scikit-learn user guide

Release 0.22.2

scikit-learn developers

Mar 04, 2020

CONTENTS

CHAPTER

ONE

WELCOME TO SCIKIT-LEARN

1.1 Installing scikit-learn

There are different ways to install scikit-learn:

- *Install the latest official release*. This is the best approach for most users. It will provide a stable version and pre-built packages are available for most platforms.
- Install the version of scikit-learn provided by your *operating system or Python distribution*. This is a quick option for those who have operating systems or Python distributions that distribute scikit-learn. It might not provide the latest release version.
- *Building the package from source*. This is best for users who want the latest-and-greatest features and aren't afraid of running brand-new code. This is also needed for users who wish to contribute to the project.

1.1.1 Installing the latest release

Then run:

In order to check your installation you can use

Note that in order to avoid potential conflicts with other packages it is strongly recommended to use a virtual environment, e.g. python3 `virtualenv` (see python3 virtualenv documentation) or conda environments.

Using an isolated environment makes possible to install a specific version of scikit-learn and its dependencies independently of any previously installed Python packages. In particular under Linux is it discouraged to install pip packages alongside the packages managed by the package manager of the distribution (apt, dnf, pacman...).

Note that you should always remember to activate the environment of your choice prior to running any Python command whenever you start a new terminal session.

If you have not installed NumPy or SciPy yet, you can also install these using conda or pip. When using pip, please ensure that *binary wheels* are used, and NumPy and SciPy are not recompiled from source, which can happen when using particular configurations of operating system and hardware (such as Linux on a Raspberry Pi).

If you must install scikit-learn and its dependencies with pip, you can install it as `scikit-learn[alldeps]`.

Scikit-learn plotting capabilities (i.e., functions start with "plot_" and classes end with "Display") require Matplotlib (>= 1.5.1). For running the examples Matplotlib >= 1.5.1 is required. A few examples require scikit-image >= 0.12.3, a few examples require pandas >= 0.18.0.

Warning: Scikit-learn 0.20 was the last version to support Python 2.7 and Python 3.4. Scikit-learn now requires Python 3.5 or newer.

Note: For installing on PyPy, PyPy3-v5.10+, Numpy 1.14.0+, and scipy 1.1.0+ are required.

1.1.2 Third party distributions of scikit-learn

Some third-party distributions provide versions of scikit-learn integrated with their package-management systems.

These can make installation and upgrading much easier for users since the integration includes the ability to automatically install dependencies (numpy, scipy) that scikit-learn requires.

The following is an incomplete list of OS and python distributions that provide their own version of scikit-learn.

Arch Linux

Arch Linux's package is provided through the official repositories as `python-scikit-learn` for Python. It can be installed by typing the following command:

```
$ sudo pacman -S python-scikit-learn
```

Debian/Ubuntu

The Debian/Ubuntu package is splitted in three different packages called `python3-sklearn` (python modules), `python3-sklearn-lib` (low-level implementations and bindings), `python3-sklearn-doc` (documentation). Only the Python 3 version is available in the Debian Buster (the more recent Debian distribution). Packages can be installed using `apt-get`:

```
$ sudo apt-get install python3-sklearn python3-sklearn-lib python3-sklearn-doc
```

Fedora

The Fedora package is called `python3-scikit-learn` for the python 3 version, the only one available in Fedora30. It can be installed using `dnf`:

```
$ sudo dnf install python3-scikit-learn
```

NetBSD

scikit-learn is available via pkgsrc-wip:

http://pkgsrc.se/math/py-scikit-learn

MacPorts for Mac OSX

The MacPorts package is named `py<XY>-scikits-learn`, where `XY` denotes the Python version. It can be installed by typing the following command:

```
$ sudo port install py36-scikit-learn
```

Canopy and Anaconda for all supported platforms

Canopy and Anaconda both ship a recent version of scikit-learn, in addition to a large set of scientific python library for Windows, Mac OSX and Linux.

Anaconda offers scikit-learn as part of its free distribution.

Intel conda channel

Intel maintains a dedicated conda channel that ships scikit-learn:

```
$ conda install -c intel scikit-learn
```

This version of scikit-learn comes with alternative solvers for some common estimators. Those solvers come from the DAAL C++ library and are optimized for multi-core Intel CPUs.

Note that those solvers are not enabled by default, please refer to the daal4py documentation for more details.

Compatibility with the standard scikit-learn solvers is checked by running the full scikit-learn test suite via automated continuous integration as reported on https://github.com/IntelPython/daal4py.

WinPython for Windows

The WinPython project distributes scikit-learn as an additional plugin.

1.1.3 Troubleshooting

Error caused by file path length limit on Windows

It can happen that pip fails to install packages when reaching the default path size limit of Windows if Python is installed in a nested location such as the `AppData` folder structure under the user home directory, for instance:

```
C:\Users\username>C:\Users\username\AppData\Local\Microsoft\WindowsApps\python.exe -m␣
↪pip install scikit-learn
Collecting scikit-learn
...
Installing collected packages: scikit-learn
ERROR: Could not install packages due to an EnvironmentError: [Errno 2] No such file␣
↪or directory:
↪'C:\\Users\\username\\AppData\\Local\\Packages\\PythonSoftwareFoundation.Python.3.7_
↪qbz5n2kfra8p0\\LocalCache\\local-packages\\Python37\\site-
↪packages\\sklearn\\datasets\\tests\\data\\openml\\292\\api-v1-json-data-list-data_
↪name-australian-limit-2-data_version-1-status-deactivated.json.gz'
```

In this case it is possible to lift that limit in the Windows registry by using the `regedit` tool:

1. Type "regedit" in the Windows start menu to launch `regedit`.

2. Go to the `Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem` key.
3. Edit the value of the `LongPathsEnabled` property of that key and set it to 1.
4. Reinstall scikit-learn (ignoring the previous broken installation):

```
pip install --exists-action=i scikit-learn
```

1.2 Frequently Asked Questions

Here we try to give some answers to questions that regularly pop up on the mailing list.

1.2.1 What is the project name (a lot of people get it wrong)?

scikit-learn, but not scikit or SciKit nor sci-kit learn. Also not scikits.learn or scikits-learn, which were previously used.

1.2.2 How do you pronounce the project name?

sy-kit learn. sci stands for science!

1.2.3 Why scikit?

There are multiple scikits, which are scientific toolboxes built around SciPy. You can find a list at https://scikits.appspot.com/scikits. Apart from scikit-learn, another popular one is scikit-image.

1.2.4 How can I contribute to scikit-learn?

See *Contributing*. Before wanting to add a new algorithm, which is usually a major and lengthy undertaking, it is recommended to start with *known issues*. Please do not contact the contributors of scikit-learn directly regarding contributing to scikit-learn.

1.2.5 What's the best way to get help on scikit-learn usage?

For general machine learning questions, please use Cross Validated with the `[machine-learning]` tag.

For scikit-learn usage questions, please use Stack Overflow with the `[scikit-learn]` and `[python]` tags. You can alternatively use the mailing list.

Please make sure to include a minimal reproduction code snippet (ideally shorter than 10 lines) that highlights your problem on a toy dataset (for instance from `sklearn.datasets` or randomly generated with functions of `numpy.random` with a fixed random seed). Please remove any line of code that is not necessary to reproduce your problem.

The problem should be reproducible by simply copy-pasting your code snippet in a Python shell with scikit-learn installed. Do not forget to include the import statements.

More guidance to write good reproduction code snippets can be found at:

https://stackoverflow.com/help/mcve

If your problem raises an exception that you do not understand (even after googling it), please make sure to include the full traceback that you obtain when running the reproduction script.

For bug reports or feature requests, please make use of the issue tracker on GitHub.

There is also a scikit-learn Gitter channel where some users and developers might be found.

Please do not email any authors directly to ask for assistance, report bugs, or for any other issue related to scikit-learn.

1.2.6 How should I save, export or deploy estimators for production?

See *Model persistence*.

1.2.7 How can I create a bunch object?

Don't make a bunch object! They are not part of the scikit-learn API. Bunch objects are just a way to package some numpy arrays. As a scikit-learn user you only ever need numpy arrays to feed your model with data.

For instance to train a classifier, all you need is a 2D array `X` for the input variables and a 1D array `y` for the target variables. The array `X` holds the features as columns and samples as rows . The array `y` contains integer values to encode the class membership of each sample in `X`.

1.2.8 How can I load my own datasets into a format usable by scikit-learn?

Generally, scikit-learn works on any numeric data stored as numpy arrays or scipy sparse matrices. Other types that are convertible to numeric arrays such as pandas DataFrame are also acceptable.

For more information on loading your data files into these usable data structures, please refer to *loading external datasets*.

1.2.9 What are the inclusion criteria for new algorithms ?

We only consider well-established algorithms for inclusion. A rule of thumb is at least 3 years since publication, 200+ citations and wide use and usefulness. A technique that provides a clear-cut improvement (e.g. an enhanced data structure or a more efficient approximation technique) on a widely-used method will also be considered for inclusion.

From the algorithms or techniques that meet the above criteria, only those which fit well within the current API of scikit-learn, that is a `fit`, `predict/transform` interface and ordinarily having input/output that is a numpy array or sparse matrix, are accepted.

The contributor should support the importance of the proposed addition with research papers and/or implementations in other similar packages, demonstrate its usefulness via common use-cases/applications and corroborate performance improvements, if any, with benchmarks and/or plots. It is expected that the proposed algorithm should outperform the methods that are already implemented in scikit-learn at least in some areas.

Inclusion of a new algorithm speeding up an existing model is easier if:

- it does not introduce new hyper-parameters (as it makes the library more future-proof),
- it is easy to document clearly when the contribution improves the speed and when it does not, for instance "when n_features >> n_samples",
- benchmarks clearly show a speed up.

Also note that your implementation need not be in scikit-learn to be used together with scikit-learn tools. You can implement your favorite algorithm in a scikit-learn compatible way, upload it to GitHub and let us know. We will be happy to list it under *Related Projects*. If you already have a package on GitHub following the scikit-learn API, you may also be interested to look at scikit-learn-contrib.

1.2.10 Why are you so selective on what algorithms you include in scikit-learn?

Code is maintenance cost, and we need to balance the amount of code we have with the size of the team (and add to this the fact that complexity scales non linearly with the number of features). The package relies on core developers using their free time to fix bugs, maintain code and review contributions. Any algorithm that is added needs future attention by the developers, at which point the original author might long have lost interest. See also *What are the inclusion criteria for new algorithms ?*. For a great read about long-term maintenance issues in open-source software, look at the Executive Summary of Roads and Bridges

1.2.11 Why did you remove HMMs from scikit-learn?

See *Will you add graphical models or sequence prediction to scikit-learn?*.

1.2.12 Will you add graphical models or sequence prediction to scikit-learn?

Not in the foreseeable future. scikit-learn tries to provide a unified API for the basic tasks in machine learning, with pipelines and meta-algorithms like grid search to tie everything together. The required concepts, APIs, algorithms and expertise required for structured learning are different from what scikit-learn has to offer. If we started doing arbitrary structured learning, we'd need to redesign the whole package and the project would likely collapse under its own weight.

There are two project with API similar to scikit-learn that do structured prediction:

- pystruct handles general structured learning (focuses on SSVMs on arbitrary graph structures with approximate inference; defines the notion of sample as an instance of the graph structure)
- seqlearn handles sequences only (focuses on exact inference; has HMMs, but mostly for the sake of completeness; treats a feature vector as a sample and uses an offset encoding for the dependencies between feature vectors)

1.2.13 Will you add GPU support?

No, or at least not in the near future. The main reason is that GPU support will introduce many software dependencies and introduce platform specific issues. scikit-learn is designed to be easy to install on a wide variety of platforms. Outside of neural networks, GPUs don't play a large role in machine learning today, and much larger gains in speed can often be achieved by a careful choice of algorithms.

1.2.14 Do you support PyPy?

In case you didn't know, PyPy is an alternative Python implementation with a built-in just-in-time compiler. Experimental support for PyPy3-v5.10+ has been added, which requires Numpy 1.14.0+, and scipy 1.1.0+.

1.2.15 How do I deal with string data (or trees, graphs...)?

scikit-learn estimators assume you'll feed them real-valued feature vectors. This assumption is hard-coded in pretty much all of the library. However, you can feed non-numerical inputs to estimators in several ways.

If you have text documents, you can use a term frequency features; see *Text feature extraction* for the built-in *text vectorizers*. For more general feature extraction from any kind of data, see *Loading features from dicts* and *Feature hashing*.

Another common case is when you have non-numerical data and a custom distance (or similarity) metric on these data. Examples include strings with edit distance (aka. Levenshtein distance; e.g., DNA or RNA sequences). These can be encoded as numbers, but doing so is painful and error-prone. Working with distance metrics on arbitrary data can be done in two ways.

Firstly, many estimators take precomputed distance/similarity matrices, so if the dataset is not too large, you can compute distances for all pairs of inputs. If the dataset is large, you can use feature vectors with only one "feature", which is an index into a separate data structure, and supply a custom metric function that looks up the actual data in this data structure. E.g., to use DBSCAN with Levenshtein distances:

```
>>> from leven import levenshtein
>>> import numpy as np
>>> from sklearn.cluster import dbscan
>>> data = ["ACCTCCTAGAAG", "ACCTACTAGAAGTT", "GAATATTAGGCCGA"]
>>> def lev_metric(x, y):
...     i, j = int(x[0]), int(y[0])     # extract indices
...     return levenshtein(data[i], data[j])
...
>>> X = np.arange(len(data)).reshape(-1, 1)
>>> X
array([[0],
       [1],
       [2]])
>>> # We need to specify algoritum='brute' as the default assumes
>>> # a continuous feature space.
>>> dbscan(X, metric=lev_metric, eps=5, min_samples=2, algorithm='brute')
...
([0, 1], array([ 0,  0, -1]))
```

(This uses the third-party edit distance package `leven`.)

Similar tricks can be used, with some care, for tree kernels, graph kernels, etc.

1.2.16 Why do I sometime get a crash/freeze with n_jobs > 1 under OSX or Linux?

Several scikit-learn tools such as `GridSearchCV` and `cross_val_score` rely internally on Python's `multiprocessing` module to parallelize execution onto several Python processes by passing `n_jobs > 1` as argument.

The problem is that Python `multiprocessing` does a `fork` system call without following it with an `exec` system call for performance reasons. Many libraries like (some versions of) Accelerate / vecLib under OSX, (some versions of) MKL, the OpenMP runtime of GCC, nvidia's Cuda (and probably many others), manage their own internal thread pool. Upon a call to `fork`, the thread pool state in the child process is corrupted: the thread pool believes it has many threads while only the main thread state has been forked. It is possible to change the libraries to make them detect when a fork happens and reinitialize the thread pool in that case: we did that for OpenBLAS (merged upstream in master since 0.2.10) and we contributed a patch to GCC's OpenMP runtime (not yet reviewed).

But in the end the real culprit is Python's `multiprocessing` that does `fork` without `exec` to reduce the overhead of starting and using new Python processes for parallel computing. Unfortunately this is a violation of the POSIX

standard and therefore some software editors like Apple refuse to consider the lack of fork-safety in Accelerate / vecLib as a bug.

In Python 3.4+ it is now possible to configure `multiprocessing` to use the 'forkserver' or 'spawn' start methods (instead of the default 'fork') to manage the process pools. To work around this issue when using scikit-learn, you can set the `JOBLIB_START_METHOD` environment variable to 'forkserver'. However the user should be aware that using the 'forkserver' method prevents joblib.Parallel to call function interactively defined in a shell session.

If you have custom code that uses `multiprocessing` directly instead of using it via joblib you can enable the 'forkserver' mode globally for your program: Insert the following instructions in your main script:

```
import multiprocessing

# other imports, custom code, load data, define model...

if __name__ == '__main__':
    multiprocessing.set_start_method('forkserver')

    # call scikit-learn utils with n_jobs > 1 here
```

You can find more default on the new start methods in the multiprocessing documentation.

1.2.17 Why does my job use more cores than specified with n_jobs?

This is because `n_jobs` only controls the number of jobs for routines that are parallelized with `joblib`, but parallel code can come from other sources:

- some routines may be parallelized with OpenMP (for code written in C or Cython).
- scikit-learn relies a lot on numpy, which in turn may rely on numerical libraries like MKL, OpenBLAS or BLIS which can provide parallel implementations.

For more details, please refer to our *Parallelism notes*.

1.2.18 Why is there no support for deep or reinforcement learning / Will there be support for deep or reinforcement learning in scikit-learn?

Deep learning and reinforcement learning both require a rich vocabulary to define an architecture, with deep learning additionally requiring GPUs for efficient computing. However, neither of these fit within the design constraints of scikit-learn; as a result, deep learning and reinforcement learning are currently out of scope for what scikit-learn seeks to achieve.

You can find more information about addition of gpu support at *Will you add GPU support?*.

1.2.19 Why is my pull request not getting any attention?

The scikit-learn review process takes a significant amount of time, and contributors should not be discouraged by a lack of activity or review on their pull request. We care a lot about getting things right the first time, as maintenance and later change comes at a high cost. We rarely release any "experimental" code, so all of our contributions will be subject to high use immediately and should be of the highest quality possible initially.

Beyond that, scikit-learn is limited in its reviewing bandwidth; many of the reviewers and core developers are working on scikit-learn on their own time. If a review of your pull request comes slowly, it is likely because the reviewers are busy. We ask for your understanding and request that you not close your pull request or discontinue your work solely because of this reason.

1.2.20 How do I set a `random_state` for an entire execution?

For testing and replicability, it is often important to have the entire execution controlled by a single seed for the pseudo-random number generator used in algorithms that have a randomized component. Scikit-learn does not use its own global random state; whenever a RandomState instance or an integer random seed is not provided as an argument, it relies on the numpy global random state, which can be set using `numpy.random.seed`. For example, to set an execution's numpy global random state to 42, one could execute the following in his or her script:

```
import numpy as np
np.random.seed(42)
```

However, a global random state is prone to modification by other code during execution. Thus, the only way to ensure replicability is to pass `RandomState` instances everywhere and ensure that both estimators and cross-validation splitters have their `random_state` parameter set.

1.2.21 Why do categorical variables need preprocessing in scikit-learn, compared to other tools?

Most of scikit-learn assumes data is in NumPy arrays or SciPy sparse matrices of a single numeric dtype. These do not explicitly represent categorical variables at present. Thus, unlike R's data.frames or pandas.DataFrame, we require explicit conversion of categorical features to numeric values, as discussed in *Encoding categorical features*. See also *Column Transformer with Mixed Types* for an example of working with heterogeneous (e.g. categorical and numeric) data.

1.2.22 Why does Scikit-learn not directly work with, for example, pandas.DataFrame?

The homogeneous NumPy and SciPy data objects currently expected are most efficient to process for most operations. Extensive work would also be needed to support Pandas categorical types. Restricting input to homogeneous types therefore reduces maintenance cost and encourages usage of efficient data structures.

1.2.23 Do you plan to implement transform for target y in a pipeline?

Currently transform only works for features X in a pipeline. There's a long-standing discussion about not being able to transform y in a pipeline. Follow on github issue #4143. Meanwhile check out *`sklearn.compose.TransformedTargetRegressor`*, pipegraph, imbalanced-learn. Note that Scikit-learn solved for the case where y has an invertible transformation applied before training and inverted after prediction. Scikit-learn intends to solve for use cases where y should be transformed at training time and not at test time, for resampling and similar uses, like at imbalanced learn. In general, these use cases can be solved with a custom meta estimator rather than a Pipeline

1.3 Support

There are several ways to get in touch with the developers.

1.3.1 Mailing List

- The main mailing list is scikit-learn.
- There is also a commit list scikit-learn-commits, where updates to the main repository and test failures get notified.

1.3.2 User questions

- Some scikit-learn developers support users on StackOverflow using the [scikit-learn] tag.
- For general theoretical or methodological Machine Learning questions stack exchange is probably a more suitable venue.

In both cases please use a descriptive question in the title field (e.g. no "Please help with scikit-learn!" as this is not a question) and put details on what you tried to achieve, what were the expected results and what you observed instead in the details field.

Code and data snippets are welcome. Minimalistic (up to ~20 lines long) reproduction script very helpful.

Please describe the nature of your data and the how you preprocessed it: what is the number of samples, what is the number and type of features (i.d. categorical or numerical) and for supervised learning tasks, what target are your trying to predict: binary, multiclass (1 out of `n_classes`) or multilabel (`k` out of `n_classes`) classification or continuous variable regression.

1.3.3 Bug tracker

If you think you've encountered a bug, please report it to the issue tracker:

https://github.com/scikit-learn/scikit-learn/issues

Don't forget to include:

- steps (or better script) to reproduce,
- expected outcome,
- observed outcome or python (or gdb) tracebacks

To help developers fix your bug faster, please link to a https://gist.github.com holding a standalone minimalistic python script that reproduces your bug and optionally a minimalistic subsample of your dataset (for instance exported as CSV files using `numpy.savetxt`).

Note: gists are git cloneable repositories and thus you can use git to push datafiles to them.

1.3.4 IRC

Some developers like to hang out on channel `#scikit-learn` on `irc.freenode.net`.

If you do not have an IRC client or are behind a firewall this web client works fine: https://webchat.freenode.net

1.3.5 Documentation resources

This documentation is relative to 0.22.2. Documentation for other versions can be found here.

Printable pdf documentation for old versions can be found here.

1.4 Related Projects

Projects implementing the scikit-learn estimator API are encouraged to use the scikit-learn-contrib template which facilitates best practices for testing and documenting estimators. The scikit-learn-contrib GitHub organisation also accepts high-quality contributions of repositories conforming to this template.

Below is a list of sister-projects, extensions and domain specific packages.

1.4.1 Interoperability and framework enhancements

These tools adapt scikit-learn for use with other technologies or otherwise enhance the functionality of scikit-learn's estimators.

Data formats

- sklearn_pandas bridge for scikit-learn pipelines and pandas data frame with dedicated transformers.
- sklearn_xarray provides compatibility of scikit-learn estimators with xarray data structures.

Auto-ML

- auto_ml Automated machine learning for production and analytics, built on scikit-learn and related projects. Trains a pipeline wth all the standard machine learning steps. Tuned for prediction speed and ease of transfer to production environments.
- auto-sklearn An automated machine learning toolkit and a drop-in replacement for a scikit-learn estimator
- TPOT An automated machine learning toolkit that optimizes a series of scikit-learn operators to design a machine learning pipeline, including data and feature preprocessors as well as the estimators. Works as a drop-in replacement for a scikit-learn estimator.
- scikit-optimize A library to minimize (very) expensive and noisy black-box functions. It implements several methods for sequential model-based optimization, and includes a replacement for `GridSearchCV` or `RandomizedSearchCV` to do cross-validated parameter search using any of these strategies.

Experimentation frameworks

- REP Environment for conducting data-driven research in a consistent and reproducible way
- ML Frontend provides dataset management and SVM fitting/prediction through web-based and programmatic interfaces.
- Scikit-Learn Laboratory A command-line wrapper around scikit-learn that makes it easy to run machine learning experiments with multiple learners and large feature sets.
- Xcessiv is a notebook-like application for quick, scalable, and automated hyperparameter tuning and stacked ensembling. Provides a framework for keeping track of model-hyperparameter combinations.

Model inspection and visualisation

- eli5 A library for debugging/inspecting machine learning models and explaining their predictions.
- mlxtend Includes model visualization utilities.
- scikit-plot A visualization library for quick and easy generation of common plots in data analysis and machine learning.
- yellowbrick A suite of custom matplotlib visualizers for scikit-learn estimators to support visual feature analysis, model selection, evaluation, and diagnostics.

Model export for production

- onnxmltools Serializes many Scikit-learn pipelines to ONNX for interchange and prediction.
- sklearn2pmml Serialization of a wide variety of scikit-learn estimators and transformers into PMML with the help of JPMML-SkLearn library.
- sklearn-porter Transpile trained scikit-learn models to C, Java, Javascript and others.
- sklearn-compiledtrees Generate a C++ implementation of the predict function for decision trees (and ensembles) trained by sklearn. Useful for latency-sensitive production environments.

1.4.2 Other estimators and tasks

Not everything belongs or is mature enough for the central scikit-learn project. The following are projects providing interfaces similar to scikit-learn for additional learning algorithms, infrastructures and tasks.

Structured learning

- sktime A scikit-learn compatible toolbox for machine learning with time series including time series classification/regression and (supervised/panel) forecasting.
- Seqlearn Sequence classification using HMMs or structured perceptron.
- HMMLearn Implementation of hidden markov models that was previously part of scikit-learn.
- PyStruct General conditional random fields and structured prediction.
- pomegranate Probabilistic modelling for Python, with an emphasis on hidden Markov models.
- sklearn-crfsuite Linear-chain conditional random fields (CRFsuite wrapper with sklearn-like API).

Deep neural networks etc.

- pylearn2 A deep learning and neural network library build on theano with scikit-learn like interface.
- sklearn_theano scikit-learn compatible estimators, transformers, and datasets which use Theano internally
- nolearn A number of wrappers and abstractions around existing neural network libraries
- keras Deep Learning library capable of running on top of either TensorFlow or Theano.
- lasagne A lightweight library to build and train neural networks in Theano.
- skorch A scikit-learn compatible neural network library that wraps PyTorch.

Broad scope

- mlxtend Includes a number of additional estimators as well as model visualization utilities.
- sparkit-learn Scikit-learn API and functionality for PySpark's distributed modelling.

Other regression and classification

- xgboost Optimised gradient boosted decision tree library.
- ML-Ensemble Generalized ensemble learning (stacking, blending, subsemble, deep ensembles, etc.).
- lightning Fast state-of-the-art linear model solvers (SDCA, AdaGrad, SVRG, SAG, etc...).
- py-earth Multivariate adaptive regression splines
- Kernel Regression Implementation of Nadaraya-Watson kernel regression with automatic bandwidth selection
- gplearn Genetic Programming for symbolic regression tasks.
- multiisotonic Isotonic regression on multidimensional features.
- scikit-multilearn Multi-label classification with focus on label space manipulation.
- seglearn Time series and sequence learning using sliding window segmentation.

Decomposition and clustering

- lda: Fast implementation of latent Dirichlet allocation in Cython which uses Gibbs sampling to sample from the true posterior distribution. (scikit-learn's `sklearn.decomposition.LatentDirichletAllocation` implementation uses variational inference to sample from a tractable approximation of a topic model's posterior distribution.)
- Sparse Filtering Unsupervised feature learning based on sparse-filtering

- kmodes k-modes clustering algorithm for categorical data, and several of its variations.
- hdbscan HDBSCAN and Robust Single Linkage clustering algorithms for robust variable density clustering.
- spherecluster Spherical K-means and mixture of von Mises Fisher clustering routines for data on the unit hypersphere.

Pre-processing

- categorical-encoding A library of sklearn compatible categorical variable encoders.
- imbalanced-learn Various methods to under- and over-sample datasets.

1.4.3 Statistical learning with Python

Other packages useful for data analysis and machine learning.

- Pandas Tools for working with heterogeneous and columnar data, relational queries, time series and basic statistics.
- theano A CPU/GPU array processing framework geared towards deep learning research.
- statsmodels Estimating and analysing statistical models. More focused on statistical tests and less on prediction than scikit-learn.
- PyMC Bayesian statistical models and fitting algorithms.
- Sacred Tool to help you configure, organize, log and reproduce experiments
- Seaborn Visualization library based on matplotlib. It provides a high-level interface for drawing attractive statistical graphics.
- Deep Learning A curated list of deep learning software libraries.

Recommendation Engine packages

- GraphLab Implementation of classical recommendation techniques (in C++, with Python bindings).
- implicit, Library for implicit feedback datasets.
- lightfm A Python/Cython implementation of a hybrid recommender system.
- OpenRec TensorFlow-based neural-network inspired recommendation algorithms.
- Spotlight Pytorch-based implementation of deep recommender models.
- Surprise Lib Library for explicit feedback datasets.

Domain specific packages

- scikit-image Image processing and computer vision in python.
- Natural language toolkit (nltk) Natural language processing and some machine learning.
- gensim A library for topic modelling, document indexing and similarity retrieval
- NiLearn Machine learning for neuro-imaging.
- AstroML Machine learning for astronomy.
- MSMBuilder Machine learning for protein conformational dynamics time series.
- scikit-surprise A scikit for building and evaluating recommender systems.

1.4.4 Snippets and tidbits

The wiki has more!

1.5 About us

1.5.1 History

This project was started in 2007 as a Google Summer of Code project by David Cournapeau. Later that year, Matthieu Brucher started work on this project as part of his thesis.

In 2010 Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort and Vincent Michel of INRIA took leadership of the project and made the first public release, February the 1st 2010. Since then, several releases have appeared following a ~3 month cycle, and a thriving international community has been leading the development.

1.5.2 Governance

The decision making process and governance structure of scikit-learn is laid out in the *governance document*.

1.5.3 Authors

The following people are currently core contributors to scikit-learn's development and maintenance:

Please do not email the authors directly to ask for assistance or report issues. Instead, please see What's the best way to ask questions about scikit-learn in the FAQ.

See also:

How you can contribute to the project

1.5.4 Emeritus Core Developers

The following people have been active contributors in the past, but are no longer active in the project:

- Mathieu Blondel
- Matthieu Brucher
- Lars Buitinck
- David Cournapeau
- Noel Dawe
- Shiqiao Du
- Vincent Dubourg
- Edouard Duchesnay
- Alexander Fabisch
- Virgile Fritsch
- Satrajit Ghosh
- Angel Soler Gollonet

- Chris Gorgolewski
- Jaques Grobler
- Brian Holt
- Arnaud Joly
- Thouis (Ray) Jones
- Kyle Kastner
- manoj kumar
- Robert Layton
- Wei Li
- Paolo Losi
- Gilles Louppe
- Vincent Michel
- Jarrod Millman
- Alexandre Passos
- Fabian Pedregosa
- Peter Prettenhofer
- (Venkat) Raghav, Rajagopalan
- Jacob Schreiber
- Jake Vanderplas
- David Warde-Farley
- Ron Weiss

1.5.5 Citing scikit-learn

If you use scikit-learn in a scientific publication, we would appreciate citations to the following paper:

> Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, JMLR 12, pp. 2825-2830, 2011.
>
> Bibtex entry:

```
@article{scikit-learn,
 title={Scikit-learn: Machine Learning in {P}ython},
 author={Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V.
         and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P.
         and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and
         Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E.},
 journal={Journal of Machine Learning Research},
 volume={12},
 pages={2825--2830},
 year={2011}
}
```

If you want to cite scikit-learn for its API or design, you may also want to consider the following paper:

API design for machine learning software: experiences from the scikit-learn project, Buitinck *et al.*, 2013.

Bibtex entry:

```
@inproceedings{sklearn_api,
  author    = {Lars Buitinck and Gilles Louppe and Mathieu Blondel and
               Fabian Pedregosa and Andreas Mueller and Olivier Grisel and
               Vlad Niculae and Peter Prettenhofer and Alexandre Gramfort
               and Jaques Grobler and Robert Layton and Jake VanderPlas and
               Arnaud Joly and Brian Holt and Ga{\"{e}}l Varoquaux},
  title     = {{API} design for machine learning software: experiences from
↪the scikit-learn
               project},
  booktitle = {ECML PKDD Workshop: Languages for Data Mining and Machine
↪Learning},
  year      = {2013},
  pages = {108--122},
}
```

1.5.6 Artwork

High quality PNG and SVG logos are available in the doc/logos/ source directory.

1.5.7 Funding

Scikit-Learn is a community driven project, however institutional and private grants help to assure its sustainability.

The project would like to thank the following funders.

The Members of the Scikit-Learn Consortium at Inria Foundation fund Olivier Grisel, Guillaume Lemaitre, Jérémie du Boisberranger and Chiara Marmo.

Columbia University funds Andreas Müller since 2016

Andreas Müller received a grant to improve scikit-learn from the Alfred P. Sloan Foundation . This grant supports the position of Nicolas Hug and Thomas J. Fan.

The University of Sydney funds Joel Nothman since July 2017.

Anaconda, Inc funds Adrin Jalali since 2019.

Past Sponsors

INRIA actively supports this project. It has provided funding for Fabian Pedregosa (2010-2012), Jaques Grobler (2012-2013) and Olivier Grisel (2013-2017) to work on this project full-time. It also hosts coding sprints and other events.

Paris-Saclay Center for Data Science funded one year for a developer to work on the project full-time (2014-2015), 50% of the time of Guillaume Lemaitre (2016-2017) and 50% of the time of Joris van den Bossche (2017-2018).

NYU Moore-Sloan Data Science Environment funded Andreas Mueller (2014-2016) to work on this project. The Moore-Sloan Data Science Environment also funds several students to work on the project part-time.

Télécom Paristech funded Manoj Kumar (2014), Tom Dupré la Tour (2015), Raghav RV (2015-2017), Thierry Guillemot (2016-2017) and Albert Thomas (2017) to work on scikit-learn.

The Labex DigiCosme funded Nicolas Goix (2015-2016), Tom Dupré la Tour (2015-2016 and 2017-2018), Mathurin Massias (2018-2019) to work part time on scikit-learn during their PhDs. It also funded a scikit-learn coding sprint in 2015.

The following students were sponsored by Google to work on scikit-learn through the Google Summer of Code program.

- 2007 - David Cournapeau
- 2011 - Vlad Niculae
- 2012 - Vlad Niculae, Immanuel Bayer.
- 2013 - Kemal Eren, Nicolas Trésegnie
- 2014 - Hamzeh Alsalhi, Issam Laradji, Maheshakya Wijewardena, Manoj Kumar.
- 2015 - Raghav RV, Wei Xue

- 2016 - Nelson Liu, YenChen Lin

The NeuroDebian project providing Debian packaging and contributions is supported by Dr. James V. Haxby (Dartmouth College).

1.5.8 Sprints

The International 2019 Paris sprint was kindly hosted by AXA. Also some participants could attend thanks to the support of the Alfred P. Sloan Foundation, the Python Software Foundation (PSF) and the DATAIA Institute.

The 2013 International Paris Sprint was made possible thanks to the support of Télécom Paristech, tinyclues, the French Python Association and the Fonds de la Recherche Scientifique.

The 2011 International Granada sprint was made possible thanks to the support of the PSF and tinyclues.

Donating to the project

If you are interested in donating to the project or to one of our code-sprints, you can use the *Paypal* button below or the NumFOCUS Donations Page (if you use the latter, please indicate that you are donating for the scikit-learn project).

All donations will be handled by NumFOCUS, a non-profit-organization which is managed by a board of Scipy community members. NumFOCUS's mission is to foster scientific computing software, in particular in Python. As a fiscal home of scikit-learn, it ensures that money is available when needed to keep the project funded and available while in compliance with tax regulations.

The received donations for the scikit-learn project mostly will go towards covering travel-expenses for code sprints, as well as towards the organization budget of the project[1].

Notes

1.5.9 Infrastructure support

- We would like to thank Rackspace for providing us with a free Rackspace Cloud account to automatically build the documentation and the example gallery from for the development version of scikit-learn using this tool.
- We would also like to thank Microsoft Azure, Travis Cl, CircleCl for free CPU time on their Continuous Integration servers.

1.6 Who is using scikit-learn?

1.6.1 J.P.Morgan

Scikit-learn is an indispensable part of the Python machine learning toolkit at JPMorgan. It is very widely used across all parts of the bank for classification, predictive analytics, and very many other machine learning tasks. Its

[1] Regarding the organization budget in particular, we might use some of the donated funds to pay for other project expenses such as DNS, hosting or continuous integration services.

straightforward API, its breadth of algorithms, and the quality of its documentation combine to make scikit-learn simultaneously very approachable and very powerful.

Stephen Simmons, VP, Athena Research, JPMorgan

J.P.Morgan

1.6.2 Spotify

Scikit-learn provides a toolbox with solid implementations of a bunch of state-of-the-art models and makes it easy to plug them into existing applications. We've been using it quite a lot for music recommendations at Spotify and I think it's the most well-designed ML package I've seen so far.

Erik Bernhardsson, Engineering Manager Music Discovery & Machine Learning, Spotify

1.6.3 Inria

At INRIA, we use scikit-learn to support leading-edge basic research in many teams: Parietal for neuroimaging, Lear for computer vision, Visages for medical image analysis, Privatics for security. The project is a fantastic tool to address difficult applications of machine learning in an academic environment as it is performant and versatile, but all easy-to-use and well documented, which makes it well suited to grad students.

Gaël Varoquaux, research at Parietal

1.6.4 betaworks

Betaworks is a NYC-based startup studio that builds new products, grows companies, and invests in others. Over the past 8 years we've launched a handful of social data analytics-driven services, such as Bitly, Chartbeat, digg and Scale Model. Consistently the betaworks data science team uses Scikit-learn for a variety of tasks. From exploratory analysis, to product development, it is an essential part of our toolkit. Recent uses are included in digg's new video recommender system, and Poncho's dynamic heuristic subspace clustering.

Gilad Lotan, Chief Data Scientist

1.6.5 Hugging Face

At Hugging Face we're using NLP and probabilistic models to generate conversational Artificial intelligences that are fun to chat with. Despite using deep neural nets for a few of our NLP tasks, scikit-learn is still the bread-and-butter of our daily machine learning routine. The ease of use and predictability of the interface, as well as the straightforward mathematical explanations that are here when you need them, is the killer feature. We use a variety of scikit-learn models in production and they are also operationally very pleasant to work with.

Julien Chaumond, Chief Technology Officer

1.6.6 Evernote

Building a classifier is typically an iterative process of exploring the data, selecting the features (the attributes of the data believed to be predictive in some way), training the models, and finally evaluating them. For many of these tasks, we relied on the excellent scikit-learn package for Python.

Read more

Mark Ayzenshtat, VP, Augmented Intelligence

1.6.7 Télécom ParisTech

At Telecom ParisTech, scikit-learn is used for hands-on sessions and home assignments in introductory and advanced machine learning courses. The classes are for undergrads and masters students. The great benefit of scikit-learn is its fast learning curve that allows students to quickly start working on interesting and motivating problems.

Alexandre Gramfort, Assistant Professor

1.6.8 Booking.com

At Booking.com, we use machine learning algorithms for many different applications, such as recommending hotels and destinations to our customers, detecting fraudulent reservations, or scheduling our customer service agents. Scikit-learn is one of the tools we use when implementing standard algorithms for prediction tasks. Its API and documentations are excellent and make it easy to use. The scikit-learn developers do a great job of incorporating state of the art implementations and new algorithms into the package. Thus, scikit-learn provides convenient access to a wide spectrum of algorithms, and allows us to readily find the right tool for the right job.

Melanie Mueller, Data Scientist

1.6.9 AWeber

The scikit-learn toolkit is indispensable for the Data Analysis and Management team at AWeber. It allows us to do AWesome stuff we would not otherwise have the time or resources to accomplish. The documentation is excellent, allowing new engineers to quickly evaluate and apply many different algorithms to our data. The text feature extraction utilities are useful when working with the large volume of email content we have at AWeber. The RandomizedPCA implementation, along with Pipelining and FeatureUnions, allows us to develop complex machine learning algorithms efficiently and reliably.

Anyone interested in learning more about how AWeber deploys scikit-learn in a production environment should check out talks from PyData Boston by AWeber's Michael Becker available at https://github.com/mdbecker/pydata_2013

Michael Becker, Software Engineer, Data Analysis and Management Ninjas

1.6.10 Yhat

The combination of consistent APIs, thorough documentation, and top notch implementation make scikit-learn our favorite machine learning package in Python. scikit-learn makes doing advanced analysis in Python accessible to anyone. At Yhat, we make it easy to integrate these models into your production applications. Thus eliminating the unnecessary dev time encountered productionizing analytical work.

Greg Lamp, Co-founder Yhat

1.6.11 Rangespan

The Python scikit-learn toolkit is a core tool in the data science group at Rangespan. Its large collection of well documented models and algorithms allow our team of data scientists to prototype fast and quickly iterate to find the right solution to our learning problems. We find that scikit-learn is not only the right tool for prototyping, but its careful and well tested implementation give us the confidence to run scikit-learn models in production.

Jurgen Van Gael, Data Science Director at Rangespan Ltd

1.6.12 Birchbox

At Birchbox, we face a range of machine learning problems typical to E-commerce: product recommendation, user clustering, inventory prediction, trends detection, etc. Scikit-learn lets us experiment with many models, especially in the exploration phase of a new project: the data can be passed around in a consistent way; models are easy to save and reuse; updates keep us informed of new developments from the pattern discovery research community. Scikit-learn is an important tool for our team, built the right way in the right language.

Thierry Bertin-Mahieux, Birchbox, Data Scientist

1.6.13 Bestofmedia Group

Scikit-learn is our #1 toolkit for all things machine learning at Bestofmedia. We use it for a variety of tasks (e.g. spam fighting, ad click prediction, various ranking models) thanks to the varied, state-of-the-art algorithm implementations packaged into it. In the lab it accelerates prototyping of complex pipelines. In production I can say it has proven to be robust and efficient enough to be deployed for business critical components.

Eustache Diemert, Lead Scientist Bestofmedia Group

1.6.14 Change.org

At change.org we automate the use of scikit-learn's RandomForestClassifier in our production systems to drive email targeting that reaches millions of users across the world each week. In the lab, scikit-learn's ease-of-use, performance, and overall variety of algorithms implemented has proved invaluable in giving us a single reliable source to turn to for our machine-learning needs.

Vijay Ramesh, Software Engineer in Data/science at Change.org

1.6.15 PHIMECA Engineering

At PHIMECA Engineering, we use scikit-learn estimators as surrogates for expensive-to-evaluate numerical models (mostly but not exclusively finite-element mechanical models) for speeding up the intensive post-processing operations involved in our simulation-based decision making framework. Scikit-learn's fit/predict API together with its efficient cross-validation tools considerably eases the task of selecting the best-fit estimator. We are also using scikit-learn for illustrating concepts in our training sessions. Trainees are always impressed by the ease-of-use of scikit-learn despite the apparent theoretical complexity of machine learning.

Vincent Dubourg, PHIMECA Engineering, PhD Engineer

1.6.16 HowAboutWe

At HowAboutWe, scikit-learn lets us implement a wide array of machine learning techniques in analysis and in production, despite having a small team. We use scikit-learn's classification algorithms to predict user behavior, enabling us to (for example) estimate the value of leads from a given traffic source early in the lead's tenure on our site. Also, our users' profiles consist of primarily unstructured data (answers to open-ended questions), so we use scikit-learn's feature extraction and dimensionality reduction tools to translate these unstructured data into inputs for our matchmaking system.

Daniel Weitzenfeld, Senior Data Scientist at HowAboutWe

1.6.17 PeerIndex

At PeerIndex we use scientific methodology to build the Influence Graph - a unique dataset that allows us to identify who's really influential and in which context. To do this, we have to tackle a range of machine learning and predictive modeling problems. Scikit-learn has emerged as our primary tool for developing prototypes and making quick progress. From predicting missing data and classifying tweets to clustering communities of social media users, scikit-learn proved useful in a variety of applications. Its very intuitive interface and excellent compatibility with other python tools makes it and indispensable tool in our daily research efforts.

Ferenc Huszar - Senior Data Scientist at Peerindex

1.6.18 DataRobot

DataRobot is building next generation predictive analytics software to make data scientists more productive, and scikit-learn is an integral part of our system. The variety of machine learning techniques in combination with the solid implementations that scikit-learn offers makes it a one-stop-shopping library for machine learning in Python. Moreover, its consistent API, well-tested code and permissive licensing allow us to use it in a production environment. Scikit-learn has literally saved us years of work we would have had to do ourselves to bring our product to market.

Jeremy Achin, CEO & Co-founder DataRobot Inc.

1.6.19 OkCupid

We're using scikit-learn at OkCupid to evaluate and improve our matchmaking system. The range of features it has, especially preprocessing utilities, means we can use it for a wide variety of projects, and it's performant enough to handle the volume of data that we need to sort through. The documentation is really thorough, as well, which makes the library quite easy to use.

David Koh - Senior Data Scientist at OkCupid

1.6.20 Lovely

At Lovely, we strive to deliver the best apartment marketplace, with respect to our users and our listings. From understanding user behavior, improving data quality, and detecting fraud, scikit-learn is a regular tool for gathering insights, predictive modeling and improving our product. The easy-to-read documentation and intuitive architecture of the API makes machine learning both explorable and accessible to a wide range of python developers. I'm constantly recommending that more developers and scientists try scikit-learn.

Simon Frid - Data Scientist, Lead at Lovely

1.6.21 Data Publica

Data Publica builds a new predictive sales tool for commercial and marketing teams called C-Radar. We extensively use scikit-learn to build segmentations of customers through clustering, and to predict future customers based on past partnerships success or failure. We also categorize companies using their website communication thanks to scikit-learn and its machine learning algorithm implementations. Eventually, machine learning makes it possible to detect weak signals that traditional tools cannot see. All these complex tasks are performed in an easy and straightforward way thanks to the great quality of the scikit-learn framework.

Guillaume Lebourgeois & Samuel Charron - Data Scientists at Data Publica

1.6.22 Machinalis

Scikit-learn is the cornerstone of all the machine learning projects carried at Machinalis. It has a consistent API, a wide selection of algorithms and lots of auxiliary tools to deal with the boilerplate. We have used it in production environments on a variety of projects including click-through rate prediction, information extraction, and even counting sheep!

In fact, we use it so much that we've started to freeze our common use cases into Python packages, some of them open-sourced, like FeatureForge . Scikit-learn in one word: Awesome.

Rafael Carrascosa, Lead developer

1.6.23 solido

Scikit-learn is helping to drive Moore's Law, via Solido. Solido creates computer-aided design tools used by the majority of top-20 semiconductor companies and fabs, to design the bleeding-edge chips inside smartphones, automobiles, and more. Scikit-learn helps to power Solido's algorithms for rare-event estimation, worst-case verification, optimization, and more. At Solido, we are particularly fond of scikit-learn's libraries for Gaussian Process models, large-scale regularized linear regression, and classification. Scikit-learn has increased our productivity, because for many ML problems we no longer need to "roll our own" code. This PyData 2014 talk has details.

Trent McConaghy, founder, Solido Design Automation Inc.

1.6.24 INFONEA

We employ scikit-learn for rapid prototyping and custom-made Data Science solutions within our in-memory based Business Intelligence Software INFONEA®. As a well-documented and comprehensive collection of state-of-the-art algorithms and pipelining methods, scikit-learn enables us to provide flexible and scalable scientific analysis solutions. Thus, scikit-learn is immensely valuable in realizing a powerful integration of Data Science technology within self-service business analytics.

Thorsten Kranz, Data Scientist, Coma Soft AG.

1.6.25 Dataiku

Our software, Data Science Studio (DSS), enables users to create data services that combine ETL with Machine Learning. Our Machine Learning module integrates many scikit-learn algorithms. The scikit-learn library is a perfect integration with DSS because it offers algorithms for virtually all business cases. Our goal is to offer a transparent and flexible tool that makes it easier to optimize time consuming aspects of building a data service, preparing data, and training machine learning algorithms on all types of data.

Florian Douetteau, CEO, Dataiku

1.6.26 Otto Group

Here at Otto Group, one of global Big Five B2C online retailers, we are using scikit-learn in all aspects of our daily work from data exploration to development of machine learning application to the productive deployment of those services. It helps us to tackle machine learning problems ranging from e-commerce to logistics. It consistent APIs enabled us to build the Palladium REST-API framework around it and continuously deliver scikit-learn based services.

Christian Rammig, Head of Data Science, Otto Group

1.6.27 Zopa

At Zopa, the first ever Peer-to-Peer lending platform, we extensively use scikit-learn to run the business and optimize our users' experience. It powers our Machine Learning models involved in credit risk, fraud risk, marketing, and pricing, and has been used for originating at least 1 billion GBP worth of Zopa loans. It is very well documented, powerful, and simple to use. We are grateful for the capabilities it has provided, and for allowing us to deliver on our mission of making money simple and fair.

Vlasios Vasileiou, Head of Data Science, Zopa

1.6.28 MARS

Scikit-Learn is integral to the Machine Learning Ecosystem at Mars. Whether we're designing better recipes for petfood or closely analysing our cocoa supply chain, Scikit-Learn is used as a tool for rapidly prototyping ideas and taking them to production. This allows us to better understand and meet the needs of our consumers worldwide. Scikit-Learn's feature-rich toolset is easy to use and equips our associates with the capabilities they need to solve the business challenges they face every day.

Michael Fitzke Next Generation Technologies Sr Leader, Mars Inc.

1.7 Release History

Release notes for all scikit-learn releases are linked in this this page.

Tip: Subscribe to scikit-learn releases on libraries.io to be notified when new versions are released.

1.7.1 Version 0.22.2.post1

March 3 2020

The 0.22.2.post1 release includes a packaging fix for the source distribution but the content of the packages is otherwise identical to the content of the wheels with the 0.22.2 version (without the .post1 suffix). Both contain the following changes.

Changelog

sklearn.impute

- [Efficiency] Reduce *impute.KNNImputer* asymptotic memory usage by chunking pairwise distance computation. #16397 by Joel Nothman.

sklearn.metrics

- [Fix] Fixed a bug in *metrics.plot_roc_curve* where the name of the estimator was passed in the *metrics.RocCurveDisplay* instead of the parameter `name`. It results in a different plot when calling *metrics.RocCurveDisplay.plot* for the subsequent times. #16500 by Guillaume Lemaitre.
- [Fix] Fixed a bug in *metrics.plot_precision_recall_curve* where the name of the estimator was passed in the *metrics.PrecisionRecallDisplay* instead of the parameter `name`. It results in a different plot when calling *metrics.PrecisionRecallDisplay.plot* for the subsequent times. #16505 by Guillaume Lemaitre.

sklearn.neighbors

- [Fix] Fix a bug which converted a list of arrays into a 2-D object array instead of a 1-D array containing NumPy arrays. This bug was affecting *neighbors.NearestNeighbors.radius_neighbors*. #16076 by Guillaume Lemaitre and Alex Shacked.

1.7.2 Version 0.22.1

January 2 2020

This is a bug-fix release to primarily resolve some packaging issues in version 0.22.0. It also includes minor documentation improvements and some bug fixes.

Changelog

sklearn.cluster

- [Fix] *cluster.KMeans* with `algorithm="elkan"` now uses the same stopping criterion as with the default `algorithm="full"`. #15930 by @inder128.

sklearn.inspection

- [Fix] *inspection.permutation_importance* will return the same `importances` when a `random_state` is given for both `n_jobs=1` or `n_jobs>1` both with shared memory backends (thread-safety) and isolated memory, process-based backends. Also avoid casting the data as object dtype and avoid read-only error on large dataframes with `n_jobs>1` as reported in #15810. Follow-up of #15898 by Shivam Gargsya. #15933 by Guillaume Lemaitre and Olivier Grisel.

- [FIX] *inspection.plot_partial_dependence* and *inspection.PartialDependenceDisplay.plot* now consistently checks the number of axes passed in. #15760 by Thomas Fan.

sklearn.metrics

- [FIX] *metrics.plot_confusion_matrix* now raises error when `normalize` is invalid. Previously, it runs fine with no normalization. #15888 by Hanmin Qin.
- [FIX] *metrics.plot_confusion_matrix* now colors the label color correctly to maximize contrast with its background. #15936 by Thomas Fan and @DizietAsahi.
- [FIX] *metrics.classification_report* does no longer ignore the value of the `zero_division` keyword argument. #15879 by Bibhash Chandra Mitra.
- [FIX] Fixed a bug in *metrics.plot_confusion_matrix* to correctly pass the `values_format` parameter to the `ConfusionMatrixDisplay` plot() call. #15937 by Stephen Blystone.

sklearn.model_selection

- [FIX] *model_selection.GridSearchCV* and *model_selection.RandomizedSearchCV* accept scalar values provided in `fit_params`. Change in 0.22 was breaking backward compatibility. #15863 by Adrin Jalali and Guillaume Lemaitre.

sklearn.naive_bayes

- [FIX] Removed `abstractmethod` decorator for the method `_check_X` in `naive_bayes.BaseNB` that could break downstream projects inheriting from this deprecated public base class. #15996 by Brigitta Sipőcz.

sklearn.preprocessing

- [FIX] *preprocessing.QuantileTransformer* now guarantees the `quantiles_` attribute to be completely sorted in non-decreasing manner. #15751 by Tirth Patel.

sklearn.semi_supervised

- [FIX] *semi_supervised.LabelPropagation* and *semi_supervised.LabelSpreading* now allow callable kernel function to return sparse weight matrix. #15868 by Niklas Smedemark-Margulies.

sklearn.utils

- [FIX] *utils.check_array* now correctly converts pandas DataFrame with boolean columns to floats. #15797 by Thomas Fan.
- [FIX] `utils.check_is_fitted` accepts back an explicit `attributes` argument to check for specific attributes as explicit markers of a fitted estimator. When no explicit `attributes` are provided, only the attributes that end with a underscore and do not start with double underscore are used as "fitted" markers. The `all_or_any` argument is also no longer deprecated. This change is made to restore some backward compatibility with the behavior of this utility in version 0.21. #15947 by Thomas Fan.

1.7.3 Version 0.22.0

December 3 2019

For a short description of the main highlights of the release, please refer to *Release Highlights for scikit-learn 0.22*.

Legend for changelogs

- [MAJOR FEATURE]: something big that you couldn't do before.
- [FEATURE]: something that you couldn't do before.
- [EFFICIENCY]: an existing feature now may not require as much computation or memory.
- [ENHANCEMENT]: a miscellaneous minor improvement.
- [FIX]: something that previously didn't work as documentated – or according to reasonable expectations – should now work.
- [API CHANGE]: you will need to change your code to have the same effect in the future; or a feature will be removed in the future.

Website update

Our website was revamped and given a fresh new look. #14849 by Thomas Fan.

Clear definition of the public API

Scikit-learn has a public API, and a private API.

We do our best not to break the public API, and to only introduce backward-compatible changes that do not require any user action. However, in cases where that's not possible, any change to the public API is subject to a deprecation cycle of two minor versions. The private API isn't publicly documented and isn't subject to any deprecation cycle, so users should not rely on its stability.

A function or object is public if it is documented in the API Reference and if it can be imported with an import path without leading underscores. For example `sklearn.pipeline.make_pipeline` is public, while `sklearn.pipeline._name_estimators` is private. `sklearn.ensemble._gb.BaseEnsemble` is private too because the whole `_gb` module is private.

Up to 0.22, some tools were de-facto public (no leading underscore), while they should have been private in the first place. In version 0.22, these tools have been made properly private, and the public API space has been cleaned. In addition, importing from most sub-modules is now deprecated: you should for example use `from sklearn.cluster import Birch` instead of `from sklearn.cluster.birch import Birch` (in practice, `birch.py` has been moved to `_birch.py`).

Note: All the tools in the public API should be documented in the API Reference. If you find a public tool (without leading underscore) that isn't in the API reference, that means it should either be private or documented. Please let us know by opening an issue!

This work was tracked in issue 9250 and issue 12927.

Deprecations: using `FutureWarning` from now on

When deprecating a feature, previous versions of scikit-learn used to raise a `DeprecationWarning`. Since the `DeprecationWarnings` aren't shown by default by Python, scikit-learn needed to resort to a custom warning filter to always show the warnings. That filter would sometimes interfere with users custom warning filters.

Starting from version 0.22, scikit-learn will show `FutureWarnings` for deprecations, as recommended by the Python documentation. `FutureWarnings` are always shown by default by Python, so the custom filter has been removed and scikit-learn no longer hinders with user filters. #15080 by Nicolas Hug.

Changed models

The following estimators and functions, when fit with the same data and parameters, may produce different models from the previous version. This often occurs due to changes in the modelling logic (bug fixes or enhancements), or in random sampling procedures.

- *cluster.KMeans* when `n_jobs=1`. [FIX]
- *decomposition.SparseCoder*, *decomposition.DictionaryLearning*, and *decomposition.MiniBatchDictionaryLearning* [FIX]
- *decomposition.SparseCoder* with `algorithm='lasso_lars'` [FIX]
- *decomposition.SparsePCA* where `normalize_components` has no effect due to deprecation.
- *ensemble.HistGradientBoostingClassifier* and *ensemble.HistGradientBoostingRegressor* [FIX], [FEATURE], [ENHANCEMENT].
- *impute.IterativeImputer* when `X` has features with no missing values. [FEATURE]
- *linear_model.Ridge* when `X` is sparse. [FIX]
- *model_selection.StratifiedKFold* and any use of `cv=int` with a classifier. [FIX]
- *cross_decomposition.CCA* when using scipy >= 1.3 [FIX]

Details are listed in the changelog below.

(While we are trying to better inform users by providing this information, we cannot assure that this list is complete.)

Changelog

`sklearn.base`

- [API CHANGE] From version 0.24 *base.BaseEstimator.get_params* will raise an AttributeError rather than return None for parameters that are in the estimator's constructor but not stored as attributes on the instance. #14464 by Joel Nothman.

`sklearn.calibration`

- [FIX] Fixed a bug that made *calibration.CalibratedClassifierCV* fail when given a `sample_weight` parameter of type `list` (in the case where `sample_weights` are not supported by the wrapped estimator). #13575 by William de Vazelhes.

sklearn.cluster

- [FEATURE] *cluster.SpectralClustering* now accepts precomputed sparse neighbors graph as input. #10482 by Tom Dupre la Tour and Kumar Ashutosh.
- [ENHANCEMENT] *cluster.SpectralClustering* now accepts a `n_components` parameter. This parameter extends `SpectralClustering` class functionality to match *cluster.spectral_clustering*. #13726 by Shuzhe Xiao.
- [FIX] Fixed a bug where *cluster.KMeans* produced inconsistent results between `n_jobs=1` and `n_jobs>1` due to the handling of the random state. #9288 by Bryan Yang.
- [FIX] Fixed a bug where `elkan` algorithm in *cluster.KMeans* was producing Segmentation Fault on large arrays due to integer index overflow. #15057 by Vladimir Korolev.
- [FIX] *MeanShift* now accepts a *max_iter* with a default value of 300 instead of always using the default 300. It also now exposes an `n_iter_` indicating the maximum number of iterations performed on each seed. #15120 by Adrin Jalali.
- [FIX] *cluster.AgglomerativeClustering* and *cluster.FeatureAgglomeration* now raise an error if `affinity='cosine'` and `X` has samples that are all-zeros. #7943 by @mthorrell.

sklearn.compose

- [FEATURE] Adds *compose.make_column_selector* which is used with *compose.ColumnTransformer* to select DataFrame columns on the basis of name and dtype. #12303 by Thomas Fan.
- [FIX] Fixed a bug in *compose.ColumnTransformer* which failed to select the proper columns when using a boolean list, with NumPy older than 1.12. #14510 by Guillaume Lemaitre.
- [FIX] Fixed a bug in *compose.TransformedTargetRegressor* which did not pass `**fit_params` to the underlying regressor. #14890 by Miguel Cabrera.
- [FIX] The *compose.ColumnTransformer* now requires the number of features to be consistent between `fit` and `transform`. A `FutureWarning` is raised now, and this will raise an error in 0.24. If the number of features isn't consistent and negative indexing is used, an error is raised. #14544 by Adrin Jalali.

sklearn.cross_decomposition

- [FEATURE] *cross_decomposition.PLSCanonical* and *cross_decomposition.PLSRegression* have a new function `inverse_transform` to transform data to the original space. #15304 by Jaime Ferrando Huertas.
- [ENHANCEMENT] *decomposition.KernelPCA* now properly checks the eigenvalues found by the solver for numerical or conditioning issues. This ensures consistency of results across solvers (different choices for `eigen_solver`), including approximate solvers such as `'randomized'` and `'lobpcg'` (see #12068). #12145 by Sylvain Marié
- [FIX] Fixed a bug where *cross_decomposition.PLSCanonical* and *cross_decomposition.PLSRegression* were raising an error when fitted with a target matrix `Y` in which the first column was constant. #13609 by Camila Williamson.
- [FIX] *cross_decomposition.CCA* now produces the same results with scipy 1.3 and previous scipy versions. #15661 by Thomas Fan.

sklearn.datasets

- [FEATURE] *datasets.fetch_openml* now supports heterogeneous data using pandas by setting `as_frame=True`. #13902 by Thomas Fan.
- [FEATURE] *datasets.fetch_openml* now includes the `target_names` in the returned Bunch. #15160 by Thomas Fan.
- [ENHANCEMENT] The parameter `return_X_y` was added to *datasets.fetch_20newsgroups* and *datasets.fetch_olivetti_faces*. #14259 by Sourav Singh.
- [ENHANCEMENT] *datasets.make_classification* now accepts array-like `weights` parameter, i.e. list or numpy.array, instead of list only. #14764 by Cat Chenal.
- [ENHANCEMENT] **The parameter `normalize` was added to** *datasets.fetch_20newsgroups_vectorized*. #14740 by Stéphan Tulkens
- [FIX] Fixed a bug in *datasets.fetch_openml*, which failed to load an OpenML dataset that contains an ignored feature. #14623 by Sarra Habchi.

sklearn.decomposition

- [EFFICIENCY] `decomposition.NMF(solver='mu')` fitted on sparse input matrices now uses batching to avoid briefly allocating an array with size (#non-zero elements, n_components). #15257 by Mart Willocx.
- [ENHANCEMENT] *decomposition.dict_learning* and *decomposition.dict_learning_online* now accept `method_max_iter` and pass it to *decomposition.sparse_encode*. #12650 by Adrin Jalali.
- [ENHANCEMENT] *decomposition.SparseCoder*, *decomposition.DictionaryLearning*, and *decomposition.MiniBatchDictionaryLearning* now take a `transform_max_iter` parameter and pass it to either *decomposition.dict_learning* or *decomposition.sparse_encode*. #12650 by Adrin Jalali.
- [ENHANCEMENT] *decomposition.IncrementalPCA* now accepts sparse matrices as input, converting them to dense in batches thereby avoiding the need to store the entire dense matrix at once. #13960 by Scott Gigante.
- [FIX] *decomposition.sparse_encode* now passes the `max_iter` to the underlying *linear_model.LassoLars* when `algorithm='lasso_lars'`. #12650 by Adrin Jalali.

sklearn.dummy

- [FIX] *dummy.DummyClassifier* now handles checking the existence of the provided constant in multiouput cases. #14908 by Martina G. Vilas.
- [API CHANGE] The default value of the `strategy` parameter in *dummy.DummyClassifier* will change from `'stratified'` in version 0.22 to `'prior'` in 0.24. A FutureWarning is raised when the default value is used. #15382 by Thomas Fan.
- [API CHANGE] The `outputs_2d_` attribute is deprecated in *dummy.DummyClassifier* and *dummy.DummyRegressor*. It is equivalent to `n_outputs > 1`. #14933 by Nicolas Hug

sklearn.ensemble

- [Major Feature] Added *ensemble.StackingClassifier* and *ensemble.StackingRegressor* to stack predictors using a final classifier or regressor. #11047 by Guillaume Lemaitre and Caio Oliveira and #15138 by Jon Cusick..
- Many improvements were made to *ensemble.HistGradientBoostingClassifier* and *ensemble.HistGradientBoostingRegressor*:
  - [Major Feature] Estimators now natively support dense data with missing values both for training and predicting. They also support infinite values. #13911 and #14406 by Nicolas Hug, Adrin Jalali and Olivier Grisel.
  - [Feature] Estimators now have an additional `warm_start` parameter that enables warm starting. #14012 by Johann Faouzi.
  - [Feature] *inspection.partial_dependence* and *inspection.plot_partial_dependence* now support the fast 'recursion' method for both estimators. #13769 by Nicolas Hug.
  - [Enhancement] for *ensemble.HistGradientBoostingClassifier* the training loss or score is now monitored on a class-wise stratified subsample to preserve the class balance of the original training set. #14194 by Johann Faouzi.
  - [Enhancement] *ensemble.HistGradientBoostingRegressor* now supports the 'least_absolute_deviation' loss. #13896 by Nicolas Hug.
  - [Fix] Estimators now bin the training and validation data separately to avoid any data leak. #13933 by Nicolas Hug.
  - [Fix] Fixed a bug where early stopping would break with string targets. #14710 by Guillaume Lemaitre.
  - [Fix] *ensemble.HistGradientBoostingClassifier* now raises an error if `categorical_crossentropy` loss is given for a binary classification problem. #14869 by Adrin Jalali.

  Note that pickles from 0.21 will not work in 0.22.
- [Enhancement] Addition of `max_samples` argument allows limiting size of bootstrap samples to be less than size of dataset. Added to *ensemble.RandomForestClassifier*, *ensemble.RandomForestRegressor*, *ensemble.ExtraTreesClassifier*, *ensemble.ExtraTreesRegressor*. #14682 by Matt Hancock and #5963 by Pablo Duboue.
- [Fix] *ensemble.VotingClassifier.predict_proba* will no longer be present when `voting='hard'`. #14287 by Thomas Fan.
- [Fix] The `named_estimators_` attribute in *ensemble.VotingClassifier* and *ensemble.VotingRegressor* now correctly maps to dropped estimators. Previously, the `named_estimators_` mapping was incorrect whenever one of the estimators was dropped. #15375 by Thomas Fan.
- [Fix] Run by default *utils.estimator_checks.check_estimator* on both *ensemble.VotingClassifier* and *ensemble.VotingRegressor*. It leads to solve issues regarding shape consistency during `predict` which was failing when the underlying estimators were not outputting consistent array dimensions. Note that it should be replaced by refactoring the common tests in the future. #14305 by Guillaume Lemaitre.
- [Fix] *ensemble.AdaBoostClassifier* computes probabilities based on the decision function as in the literature. Thus, `predict` and `predict_proba` give consistent results. #14114 by Guillaume Lemaitre.
- [Fix] Stacking and Voting estimators now ensure that their underlying estimators are either all classifiers or all regressors. *ensemble.StackingClassifier*, *ensemble.StackingRegressor*, and *ensemble.*

VotingClassifier and VotingRegressor now raise consistent error messages. #15084 by Guillaume Lemaitre.

- [FIX] *ensemble.AdaBoostRegressor* where the loss should be normalized by the max of the samples with non-null weights only. #14294 by Guillaume Lemaitre.
- [API CHANGE] presort is now deprecated in *ensemble.GradientBoostingClassifier* and *ensemble.GradientBoostingRegressor*, and the parameter has no effect. Users are recommended to use *ensemble.HistGradientBoostingClassifier* and *ensemble.HistGradientBoostingRegressor* instead. #14907 by Adrin Jalali.

sklearn.feature_extraction

- [ENHANCEMENT] A warning will now be raised if a parameter choice means that another parameter will be unused on calling the fit() method for *feature_extraction.text.HashingVectorizer*, *feature_extraction.text.CountVectorizer* and *feature_extraction.text.TfidfVectorizer*. #14602 by Gaurav Chawla.
- [FIX] Functions created by build_preprocessor and build_analyzer of feature_extraction.text.VectorizerMixin can now be pickled. #14430 by Dillon Niederhut.
- [FIX] feature_extraction.text.strip_accents_unicode now correctly removes accents from strings that are in NFKD normalized form. #15100 by Daniel Grady.
- [FIX] Fixed a bug that caused *feature_extraction.DictVectorizer* to raise an OverflowError during the transform operation when producing a scipy.sparse matrix on large input data. #15463 by Norvan Sahiner.
- [API CHANGE] Deprecated unused copy param for *feature_extraction.text.TfidfVectorizer.transform* it will be removed in v0.24. #14520 by Guillem G. Subies.

sklearn.feature_selection

- [ENHANCEMENT] Updated the following feature_selection estimators to allow NaN/Inf values in transform and fit: *feature_selection.RFE*, *feature_selection.RFECV*, *feature_selection.SelectFromModel*, and *feature_selection.VarianceThreshold*. Note that if the underlying estimator of the feature selector does not allow NaN/Inf then it will still error, but the feature selectors themselves no longer enforce this restriction unnecessarily. #11635 by Alec Peters.
- [FIX] Fixed a bug where *feature_selection.VarianceThreshold* with threshold=0 did not remove constant features due to numerical instability, by using range rather than variance in this case. #13704 by Roddy MacSween.

sklearn.gaussian_process

- [FEATURE] Gaussian process models on structured data: *gaussian_process.GaussianProcessRegressor* and *gaussian_process.GaussianProcessClassifier* can now accept a list of generic objects (e.g. strings, trees, graphs, etc.) as the X argument to their training/prediction methods. A user-defined kernel should be provided for computing the kernel matrix among the generic objects, and should inherit from gaussian_process.kernels.GenericKernelMixin to notify the GPR/GPC model that it handles non-vectorial samples. #15557 by Yu-Hang Tang.
- [EFFICIENCY] *gaussian_process.GaussianProcessClassifier.log_marginal_likelihood* and *gaussian_process.GaussianProcessRegressor.log_marginal_likelihood* now accept a clone_kernel=True keyword argument. When set

to `False`, the kernel attribute is modified, but may result in a performance improvement. #14378 by Masashi Shibata.

- [API CHANGE] From version 0.24 *gaussian_process.kernels.Kernel.get_params* will raise an `AttributeError` rather than return `None` for parameters that are in the estimator's constructor but not stored as attributes on the instance. #14464 by Joel Nothman.

sklearn.impute

- [MAJOR FEATURE] Added *impute.KNNImputer*, to impute missing values using k-Nearest Neighbors. #12852 by Ashim Bhattarai and Thomas Fan and #15010 by Guillaume Lemaitre.
- [FEATURE] *impute.IterativeImputer* has new `skip_compute` flag that is False by default, which, when True, will skip computation on features that have no missing values during the fit phase. #13773 by Sergey Feldman.
- [EFFICIENCY] *impute.MissingIndicator.fit_transform* avoid repeated computation of the masked matrix. #14356 by Harsh Soni.
- [FIX] *impute.IterativeImputer* now works when there is only one feature. By Sergey Feldman.
- [FIX] Fixed a bug in *impute.IterativeImputer* where features where imputed in the reverse desired order with `imputation_order` either `"ascending"` or `"descending"`. #15393 by Venkatachalam N.

sklearn.inspection

- [MAJOR FEATURE] *inspection.permutation_importance* has been added to measure the importance of each feature in an arbitrary trained model with respect to a given scoring function. #13146 by Thomas Fan.
- [FEATURE] *inspection.partial_dependence* and *inspection.plot_partial_dependence* now support the fast 'recursion' method for *ensemble.HistGradientBoostingClassifier* and *ensemble.HistGradientBoostingRegressor*. #13769 by Nicolas Hug.
- [ENHANCEMENT] *inspection.plot_partial_dependence* has been extended to now support the new visualization API described in the *User Guide*. #14646 by Thomas Fan.
- [ENHANCEMENT] *inspection.partial_dependence* accepts pandas DataFrame and *pipeline.Pipeline* containing *compose.ColumnTransformer*. In addition *inspection.plot_partial_dependence* will use the column names by default when a dataframe is passed. #14028 and #15429 by Guillaume Lemaitre.

sklearn.kernel_approximation

- [FIX] Fixed a bug where *kernel_approximation.Nystroem* raised a `KeyError` when using `kernel="precomputed"`. #14706 by Venkatachalam N.

sklearn.linear_model

- [EFFICIENCY] The 'liblinear' logistic regression solver is now faster and requires less memory. #14108, #14170, #14296 by Alex Henrie.
- [ENHANCEMENT] *linear_model.BayesianRidge* now accepts hyperparameters `alpha_init` and `lambda_init` which can be used to set the initial value of the maximization procedure in *fit*. #13618 by Yoshihiro Uchida.

- [Fix] *linear_model.Ridge* now correctly fits an intercept when `X` is sparse, `solver="auto"` and `fit_intercept=True`, because the default solver in this configuration has changed to `sparse_cg`, which can fit an intercept with sparse data. #13995 by Jérôme Dockès.
- [Fix] *linear_model.Ridge* with `solver='sag'` now accepts F-ordered and non-contiguous arrays and makes a conversion instead of failing. #14458 by Guillaume Lemaitre.
- [Fix] *linear_model.LassoCV* no longer forces `precompute=False` when fitting the final model. #14591 by Andreas Müller.
- [Fix] *linear_model.RidgeCV* and *linear_model.RidgeClassifierCV* now correctly scores when `cv=None`. #14864 by Venkatachalam N.
- [Fix] Fixed a bug in *linear_model.LogisticRegressionCV* where the `scores_`, `n_iter_` and `coefs_paths_` attribute would have a wrong ordering with `penalty='elastic-net'`. #15044 by Nicolas Hug
- [Fix] *linear_model.MultiTaskLassoCV* and *linear_model.MultiTaskElasticNetCV* with X of dtype int and `fit_intercept=True`. #15086 by Alex Gramfort.
- [Fix] The liblinear solver now supports `sample_weight`. #15038 by Guillaume Lemaitre.

sklearn.manifold

- [Feature] *manifold.Isomap*, *manifold.TSNE*, and *manifold.SpectralEmbedding* now accept precomputed sparse neighbors graph as input. #10482 by Tom Dupre la Tour and Kumar Ashutosh.
- [Feature] Exposed the `n_jobs` parameter in *manifold.TSNE* for multi-core calculation of the neighbors graph. This parameter has no impact when `metric="precomputed"` or (`metric="euclidean"` and `method="exact"`). #15082 by Roman Yurchak.
- [Efficiency] Improved efficiency of *manifold.TSNE* when `method="barnes-hut"` by computing the gradient in parallel. #13213 by Thomas Moreau
- [Fix] Fixed a bug where *manifold.spectral_embedding* (and therefore *manifold.SpectralEmbedding* and *cluster.SpectralClustering*) computed wrong eigenvalues with `eigen_solver='amg'` when `n_samples < 5 * n_components`. #14647 by Andreas Müller.
- [Fix] Fixed a bug in *manifold.spectral_embedding* used in *manifold.SpectralEmbedding* and *cluster.SpectralClustering* where `eigen_solver="amg"` would sometimes result in a LinAlgError. #13393 by Andrew Knyazev #13707 by Scott White
- [API Change] Deprecate `training_data_` unused attribute in *manifold.Isomap*. #10482 by Tom Dupre la Tour.

sklearn.metrics

- [Major Feature] *metrics.plot_roc_curve* has been added to plot roc curves. This function introduces the visualization API described in the *User Guide*. #14357 by Thomas Fan.
- [Feature] Added a new parameter `zero_division` to multiple classification metrics: `precision_score`, `recall_score`, `f1_score`, `fbeta_score`, `precision_recall_fscore_support`, `classification_report`. This allows to set returned value for ill-defined metrics. #14900 by Marc Torrellas Socastro.
- [Feature] Added the *metrics.pairwise.nan_euclidean_distances* metric, which calculates euclidean distances in the presence of missing values. #12852 by Ashim Bhattarai and Thomas Fan.

- [FEATURE] New ranking metrics *metrics.ndcg_score* and *metrics.dcg_score* have been added to compute Discounted Cumulative Gain and Normalized Discounted Cumulative Gain. #9951 by Jérôme Dockès.
- [FEATURE] *metrics.plot_precision_recall_curve* has been added to plot precision recall curves. #14936 by Thomas Fan.
- [FEATURE] *metrics.plot_confusion_matrix* has been added to plot confusion matrices. #15083 by Thomas Fan.
- [FEATURE] Added multiclass support to *metrics.roc_auc_score* with corresponding scorers `'roc_auc_ovr'`, `'roc_auc_ovo'`, `'roc_auc_ovr_weighted'`, and `'roc_auc_ovo_weighted'`. #12789 and #15274 by Kathy Chen, Mohamed Maskani, and Thomas Fan.
- [FEATURE] Add *metrics.mean_tweedie_deviance* measuring the Tweedie deviance for a given `power` parameter. Also add mean Poisson deviance *metrics.mean_poisson_deviance* and mean Gamma deviance *metrics.mean_gamma_deviance* that are special cases of the Tweedie deviance for `power=1` and `power=2` respectively. #13938 by Christian Lorentzen and Roman Yurchak.
- [EFFICIENCY] Improved performance of *metrics.pairwise.manhattan_distances* in the case of sparse matrices. #15049 by `Paolo Toccaceli <ptocca>`.
- [ENHANCEMENT] The parameter `beta` in *metrics.fbeta_score* is updated to accept the zero and `float('+inf')` value. #13231 by Dong-hee Na.
- [ENHANCEMENT] Added parameter `squared` in *metrics.mean_squared_error* to return root mean squared error. #13467 by Urvang Patel.
- [ENHANCEMENT] Allow computing averaged metrics in the case of no true positives. #14595 by Andreas Müller.
- [ENHANCEMENT] Multilabel metrics now supports list of lists as input. #14865 Srivatsan Ramesh, Herilalaina Rakotoarison, Léonard Binet.
- [ENHANCEMENT] *metrics.median_absolute_error* now supports `multioutput` parameter. #14732 by Agamemnon Krasoulis.
- [ENHANCEMENT] 'roc_auc_ovr_weighted' and 'roc_auc_ovo_weighted' can now be used as the *scoring* parameter of model-selection tools. #14417 by Thomas Fan.
- [ENHANCEMENT] *metrics.confusion_matrix* accepts a parameters `normalize` allowing to normalize the confusion matrix by column, rows, or overall. #15625 by `Guillaume Lemaitre <glemaitre>`.
- [FIX] Raise a ValueError in *metrics.silhouette_score* when a precomputed distance matrix contains non-zero diagonal entries. #12258 by Stephen Tierney.
- [API CHANGE] `scoring="neg_brier_score"` should be used instead of `scoring="brier_score_loss"` which is now deprecated. #14898 by Stefan Matcovici.

sklearn.model_selection

- [EFFICIENCY] Improved performance of multimetric scoring in *model_selection.cross_validate*, *model_selection.GridSearchCV*, and *model_selection.RandomizedSearchCV*. #14593 by Thomas Fan.
- [ENHANCEMENT] *model_selection.learning_curve* now accepts parameter `return_times` which can be used to retrieve computation times in order to plot model scalability (see learning_curve example). #13938 by Hadrien Reboul.
- [ENHANCEMENT] *model_selection.RandomizedSearchCV* now accepts lists of parameter distributions. #14549 by Andreas Müller.

- [FIX] Reimplemented *model_selection.StratifiedKFold* to fix an issue where one test set could be `n_classes` larger than another. Test sets should now be near-equally sized. #14704 by Joel Nothman.
- [FIX] The `cv_results_` attribute of *model_selection.GridSearchCV* and *model_selection.RandomizedSearchCV* now only contains unfitted estimators. This potentially saves a lot of memory since the state of the estimators isn't stored. ##15096 by Andreas Müller.
- [API CHANGE] *model_selection.KFold* and *model_selection.StratifiedKFold* now raise a warning if `random_state` is set but `shuffle` is False. This will raise an error in 0.24.

sklearn.multioutput

- [FIX] *multioutput.MultiOutputClassifier* now has attribute `classes_`. #14629 by Agamemnon Krasoulis.
- [FIX] *multioutput.MultiOutputClassifier* now has `predict_proba` as property and can be checked with `hasattr`. #15488 #15490 by Rebekah Kim

sklearn.naive_bayes

- [MAJOR FEATURE] Added *naive_bayes.CategoricalNB* that implements the Categorical Naive Bayes classifier. #12569 by Tim Bicker and Florian Wilhelm.

sklearn.neighbors

- [MAJOR FEATURE] Added *neighbors.KNeighborsTransformer* and *neighbors.RadiusNeighborsTransformer*, which transform input dataset into a sparse neighbors graph. They give finer control on nearest neighbors computations and enable easy pipeline caching for multiple use. #10482 by Tom Dupre la Tour.
- [FEATURE] *neighbors.KNeighborsClassifier*, *neighbors.KNeighborsRegressor*, *neighbors.RadiusNeighborsClassifier*, *neighbors.RadiusNeighborsRegressor*, and *neighbors.LocalOutlierFactor* now accept precomputed sparse neighbors graph as input. #10482 by Tom Dupre la Tour and Kumar Ashutosh.
- [FEATURE] *neighbors.RadiusNeighborsClassifier* now supports predicting probabilities by using `predict_proba` and supports more outlier_label options: 'most_frequent', or different outlier_labels for multi-outputs. #9597 by Wenbo Zhao.
- [EFFICIENCY] Efficiency improvements for *neighbors.RadiusNeighborsClassifier.predict*. #9597 by Wenbo Zhao.
- [FIX] *neighbors.KNeighborsRegressor* now throws error when `metric='precomputed'` and fit on non-square data. #14336 by Gregory Dexter.

sklearn.neural_network

- [FEATURE] Add `max_fun` parameter in `neural_network.BaseMultilayerPerceptron`, *neural_network.MLPRegressor*, and *neural_network.MLPClassifier* to give control over maximum number of function evaluation to not meet `tol` improvement. #9274 by Daniel Perry.

sklearn.pipeline

- [ENHANCEMENT] *pipeline.Pipeline* now supports *score_samples* if the final estimator does. #13806 by Anaël Beaugnon.
- [FIX] The `fit` in *FeatureUnion* now accepts `fit_params` to pass to the underlying transformers. #15119 by Adrin Jalali.
- [API CHANGE] `None` as a transformer is now deprecated in *pipeline.FeatureUnion*. Please use `'drop'` instead. #15053 by Thomas Fan.

sklearn.preprocessing

- [EFFICIENCY] *preprocessing.PolynomialFeatures* is now faster when the input data is dense. #13290 by Xavier Dupré.
- [ENHANCEMENT] Avoid unnecessary data copy when fitting preprocessors *preprocessing.StandardScaler*, *preprocessing.MinMaxScaler*, *preprocessing.MaxAbsScaler*, *preprocessing.RobustScaler* and *preprocessing.QuantileTransformer* which results in a slight performance improvement. #13987 by Roman Yurchak.
- [FIX] KernelCenterer now throws error when fit on non-square *preprocessing.KernelCenterer* #14336 by Gregory Dexter.

sklearn.model_selection

- [FIX] *model_selection.GridSearchCV* and `model_selection.RandomizedSearchCV` now supports the *_pairwise* property, which prevents an error during cross-validation for estimators with pairwise inputs (such as *neighbors.KNeighborsClassifier* when *metric* is set to 'precomputed'). #13925 by Isaac S. Robson and #15524 by Xun Tang.

sklearn.svm

- [ENHANCEMENT] *svm.SVC* and *svm.NuSVC* now accept a `break_ties` parameter. This parameter results in *predict* breaking the ties according to the confidence values of *decision_function*, if `decision_function_shape='ovr'`, and the number of target classes > 2. #12557 by Adrin Jalali.
- [ENHANCEMENT] SVM estimators now throw a more specific error when `kernel='precomputed'` and fit on non-square data. #14336 by Gregory Dexter.
- [FIX] *svm.SVC*, *svm.SVR*, *svm.NuSVR* and *svm.OneClassSVM* when received values negative or zero for parameter `sample_weight` in method fit(), generated an invalid model. This behavior occurred only in some border scenarios. Now in these cases, fit() will fail with an Exception. #14286 by Alex Shacked.
- [FIX] The `n_support_` attribute of *svm.SVR* and *svm.OneClassSVM* was previously non-initialized, and had size 2. It has now size 1 with the correct value. #15099 by Nicolas Hug.
- [FIX] fixed a bug in `BaseLibSVM._sparse_fit` where n_SV=0 raised a ZeroDivisionError. #14894 by Danna Naser.
- [FIX] The liblinear solver now supports `sample_weight`. #15038 by Guillaume Lemaitre.

sklearn.tree

- [FEATURE] Adds minimal cost complexity pruning, controlled by `ccp_alpha`, to *tree.DecisionTreeClassifier*, *tree.DecisionTreeRegressor*, *tree.ExtraTreeClassifier*, *tree.ExtraTreeRegressor*, *ensemble.RandomForestClassifier*, *ensemble.RandomForestRegressor*, *ensemble.ExtraTreesClassifier*, *ensemble.ExtraTreesRegressor*, *ensemble.GradientBoostingClassifier*, and *ensemble.GradientBoostingRegressor*. #12887 by Thomas Fan.
- [API CHANGE] `presort` is now deprecated in *tree.DecisionTreeClassifier* and *tree.DecisionTreeRegressor*, and the parameter has no effect. #14907 by Adrin Jalali.
- [API CHANGE] The `classes_` and `n_classes_` attributes of *tree.DecisionTreeRegressor* are now deprecated. #15028 by Mei Guan, Nicolas Hug, and Adrin Jalali.

sklearn.utils

- [FEATURE] *check_estimator* can now generate checks by setting `generate_only=True`. Previously, running *check_estimator* will stop when the first check fails. With `generate_only=True`, all checks can run independently and report the ones that are failing. Read more in *Rolling your own estimator*. #14381 by Thomas Fan.
- [FEATURE] Added a pytest specific decorator, *parametrize_with_checks*, to parametrize estimator checks for a list of estimators. #14381 by Thomas Fan.
- [FEATURE] A new random variable, `utils.fixes.loguniform` implements a log-uniform random variable (e.g., for use in RandomizedSearchCV). For example, the outcomes `1`, `10` and `100` are all equally likely for `loguniform(1, 100)`. See #11232 by Scott Sievert and Nathaniel Saul, and `SciPy PR 10815 <https://github.com/scipy/scipy/pull/10815>`.
- [ENHANCEMENT] *utils.safe_indexing* (now deprecated) accepts an `axis` parameter to index array-like across rows and columns. The column indexing can be done on NumPy array, SciPy sparse matrix, and Pandas DataFrame. An additional refactoring was done. #14035 and #14475 by Guillaume Lemaitre.
- [ENHANCEMENT] *utils.extmath.safe_sparse_dot* works between 3D+ ndarray and sparse matrix. #14538 by Jérémie du Boisberranger.
- [FIX] *utils.check_array* is now raising an error instead of casting NaN to integer. #14872 by Roman Yurchak.
- [FIX] *utils.check_array* will now correctly detect numeric dtypes in pandas dataframes, fixing a bug where `float32` was upcast to `float64` unnecessarily. #15094 by Andreas Müller.
- [API CHANGE] The following utils have been deprecated and are now private:
  - `choose_check_classifiers_labels`
  - `enforce_estimator_tags_y`
  - `mocking.MockDataFrame`
  - `mocking.CheckingClassifier`
  - `optimize.newton_cg`
  - `random.random_choice_csc`
  - `utils.choose_check_classifiers_labels`
  - `utils.enforce_estimator_tags_y`
  - `utils.optimize.newton_cg`

  - `utils.random.random_choice_csc`
  - `utils.safe_indexing`
  - `utils.mocking`
  - `utils.fast_dict`
  - `utils.seq_dataset`
  - `utils.weight_vector`
  - `utils.fixes.parallel_helper` (removed)
  - All of `utils.testing` except for `all_estimators` which is now in `utils`.

`sklearn.isotonic`

- [Fix] Fixed a bug where *`isotonic.IsotonicRegression.fit`* raised error when `X.dtype == 'float32'` and `X.dtype != y.dtype`. #14902 by Lucas.

Miscellaneous

- [Fix] Port `lobpcg` from SciPy which implement some bug fixes but only available in 1.3+. #13609 and #14971 by Guillaume Lemaitre.
- [API Change] Scikit-learn now converts any input data structure implementing a duck array to a numpy array (using `__array__`) to ensure consistent behavior instead of relying on `__array_function__` (see NEP 18). #14702 by Andreas Müller.
- [API Change] Replace manual checks with `check_is_fitted`. Errors thrown when using a non-fitted estimators are now more uniform. #13013 by Agamemnon Krasoulis.

Changes to estimator checks

These changes mostly affect library developers.

- Estimators are now expected to raise a `NotFittedError` if `predict` or `transform` is called before `fit`; previously an `AttributeError` or `ValueError` was acceptable. #13013 by by Agamemnon Krasoulis.
- Binary only classifiers are now supported in estimator checks. Such classifiers need to have the `binary_only=True` estimator tag. #13875 by Trevor Stephens.
- Estimators are expected to convert input data (`X`, `y`, `sample_weights`) to `numpy.ndarray` and never call `__array_function__` on the original datatype that is passed (see NEP 18). #14702 by Andreas Müller.
- `requires_positive_X` estimator tag (for models that require X to be non-negative) is now used by *`utils.estimator_checks.check_estimator`* to make sure a proper error message is raised if X contains some negative entries. #14680 by Alex Gramfort.
- Added check that pairwise estimators raise error on non-square data #14336 by Gregory Dexter.
- Added two common multioutput estimator tests `check_classifier_multioutput` and `check_regressor_multioutput`. #13392 by Rok Mihevc.
- [Fix] Added `check_transformer_data_not_an_array` to checks where missing
- [Fix] The estimators tags resolution now follows the regular MRO. They used to be overridable only once. #14884 by Andreas Müller.

Code and Documentation Contributors

Thanks to everyone who has contributed to the maintenance and improvement of the project since version 0.20, including:

Aaron Alphonsus, Abbie Popa, Abdur-Rahmaan Janhangeer, abenbihi, Abhinav Sagar, Abhishek Jana, Abraham K. Lagat, Adam J. Stewart, Aditya Vyas, Adrin Jalali, Agamemnon Krasoulis, Alec Peters, Alessandro Surace, Alexandre de Siqueira, Alexandre Gramfort, alexgoryainov, Alex Henrie, Alex Itkes, alexshacked, Allen Akinkunle, Anaël Beaugnon, Anders Kaseorg, Andrea Maldonado, Andrea Navarrete, Andreas Mueller, Andreas Schuderer, Andrew Nystrom, Angela Ambroz, Anisha Keshavan, Ankit Jha, Antonio Gutierrez, Anuja Kelkar, Archana Alva, arnaud-stiegler, arpanchowdhry, ashimb9, Ayomide Bamidele, Baran Buluttekin, barrycg, Bharat Raghunathan, Bill Mill, Biswadip Mandal, blackd0t, Brian G. Barkley, Brian Wignall, Bryan Yang, c56pony, camilaagw, cartman_nabana, catajara, Cat Chenal, Cathy, cgsavard, Charles Vesteghem, Chiara Marmo, Chris Gregory, Christian Lorentzen, Christos Aridas, Dakota Grusak, Daniel Grady, Daniel Perry, Danna Naser, DatenBergwerk, David Dormagen, deeplook, Dillon Niederhut, Dong-hee Na, Dougal J. Sutherland, DrGFreeman, Dylan Cashman, edvardlindelof, Eric Larson, Eric Ndirangu, Eunseop Jeong, Fanny, federicopisanu, Felix Divo, flaviomorelli, FranciDona, Franco M. Luque, Frank Hoang, Frederic Haase, g0g0gadget, Gabriel Altay, Gabriel do Vale Rios, Gael Varoquaux, ganevgv, gdex1, getgaurav2, Gideon Sonoiya, Gordon Chen, gpapadok, Greg Mogavero, Grzegorz Szpak, Guillaume Lemaitre, Guillem García Subies, H4dr1en, hadshirt, Hailey Nguyen, Hanmin Qin, Hannah Bruce Macdonald, Harsh Mahajan, Harsh Soni, Honglu Zhang, Hossein Pourbozorg, Ian Sanders, Ingrid Spielman, J-A16, jaehong park, Jaime Ferrando Huertas, James Hill, James Myatt, Jay, jeremiedbb, Jérémie du Boisberranger, jeromedockes, Jesper Dramsch, Joan Massich, Joanna Zhang, Joel Nothman, Johann Faouzi, Jonathan Rahn, Jon Cusick, Jose Ortiz, Kanika Sabharwal, Katarina Slama, kellycarmody, Kennedy Kang'ethe, Kensuke Arai, Kesshi Jordan, Kevad, Kevin Loftis, Kevin Winata, Kevin Yu-Sheng Li, Kirill Dolmatov, Kirthi Shankar Sivamani, krishna katyal, Lakshmi Krishnan, Lakshya KD, LalliAcqua, lbfin, Leland McInnes, Léonard Binet, Loic Esteve, loopyme, lostcoaster, Louis Huynh, lrjball, Luca Ionescu, Lutz Roeder, MaggieChege, Maithreyi Venkatesh, Maltimore, Maocx, Marc Torrellas, Marie Douriez, Markus, Markus Frey, Martina G. Vilas, Martin Oywa, Martin Thoma, Masashi SHIBATA, Maxwell Aladago, mbillingr, m-clare, Meghann Agarwal, m.fab, Micah Smith, miguelbarao, Miguel Cabrera, Mina Naghshhnejad, Ming Li, motmoti, mschaffenroth, mthorrell, Natasha Borders, nezar-a, Nicolas Hug, Nidhin Pattaniyil, Nikita Titov, Nishan Singh Mann, Nitya Mandyam, norvan, notmatthancock, novaya, nxorable, Oleg Stikhin, Oleksandr Pavlyk, Olivier Grisel, Omar Saleem, Owen Flanagan, panpiort8, Paolo, Paolo Toccaceli, Paresh Mathur, Paula, Peng Yu, Peter Marko, pierretallotte, poorna-kumar, pspachtholz, qdeffense, Rajat Garg, Raphaël Bournhonesque, Ray, Ray Bell, Rebekah Kim, Reza Gharibi, Richard Payne, Richard W, rlms, Robert Juergens, Rok Mihevc, Roman Feldbauer, Roman Yurchak, R Sanjabi, RuchitaGarde, Ruth Waithera, Sackey, Sam Dixon, Samesh Lakhotia, Samuel Taylor, Sarra Habchi, Scott Gigante, Scott Sievert, Scott White, Sebastian Pölsterl, Sergey Feldman, SeWook Oh, she-dares, Shreya V, Shubham Mehta, Shuzhe Xiao, SimonCW, smarie, smujjiga, Sönke Behrends, Soumirai, Sourav Singh, stefan-matcovici, steinfurt, Stéphane Couvreur, Stephan Tulkens, Stephen Cowley, Stephen Tierney, SylvainLan, th0rwas, theoptips, theotheo, Thierno Ibrahima DIOP, Thomas Edwards, Thomas J Fan, Thomas Moreau, Thomas Schmitt, Tilen Kusterle, Tim Bicker, Timsaur, Tim Staley, Tirth Patel, Tola A, Tom Augspurger, Tom Dupré la Tour, topisan, Trevor Stephens, ttang131, Urvang Patel, Vathsala Achar, veerlosar, Venkatachalam N, Victor Luzgin, Vincent Jeanselme, Vincent Lostanlen, Vladimir Korolev, vnherdeiro, Wenbo Zhao, Wendy Hu, willdarnell, William de Vazelhes, wolframalpha, xavier dupré, xcjason, x-martian, xsat, xun-tang, Yinglr, yokasre, Yu-Hang "Maxin" Tang, Yulia Zamriy, Zhao Feng

1.7.4 Version 0.21.3

Legend for changelogs

- [MAJOR FEATURE]: something big that you couldn't do before.
- [FEATURE]: something that you couldn't do before.
- [EFFICIENCY]: an existing feature now may not require as much computation or memory.
- [ENHANCEMENT]: a miscellaneous minor improvement.

- [FIX]: something that previously didn't work as documentated – or according to reasonable expectations – should now work.
- [API CHANGE]: you will need to change your code to have the same effect in the future; or a feature will be removed in the future.

July 30, 2019

Changed models

The following estimators and functions, when fit with the same data and parameters, may produce different models from the previous version. This often occurs due to changes in the modelling logic (bug fixes or enhancements), or in random sampling procedures.

- The v0.20.0 release notes failed to mention a backwards incompatibility in *metrics.make_scorer* when `needs_proba=True` and `y_true` is binary. Now, the scorer function is supposed to accept a 1D `y_pred` (i.e., probability of the positive class, shape `(n_samples,)`), instead of a 2D `y_pred` (i.e., shape `(n_samples, 2)`).

Changelog

sklearn.cluster

- [FIX] Fixed a bug in *cluster.KMeans* where computation with `init='random'` was single threaded for `n_jobs > 1` or `n_jobs = -1`. #12955 by Prabakaran Kumaresshan.
- [FIX] Fixed a bug in *cluster.OPTICS* where users were unable to pass float `min_samples` and `min_cluster_size`. #14496 by Fabian Klopfer and Hanmin Qin.
- [FIX] Fixed a bug in *cluster.KMeans* where KMeans++ initialisation could rarely result in an IndexError. #11756 by Joel Nothman.

sklearn.compose

- [FIX] Fixed an issue in *compose.ColumnTransformer* where using DataFrames whose column order differs between :func:`fit` and :func:`transform` could lead to silently passing incorrect columns to the `remainder` transformer. #14237 by `Andreas Schuderer <schuderer>`.

sklearn.datasets

- [FIX] *datasets.fetch_california_housing*, *datasets.fetch_covtype*, *datasets.fetch_kddcup99*, *datasets.fetch_olivetti_faces*, *datasets.fetch_rcv1*, and *datasets.fetch_species_distributions* try to persist the previously cache using the new `joblib` if the cached data was persisted using the deprecated `sklearn.externals.joblib`. This behavior is set to be deprecated and removed in v0.23. #14197 by Adrin Jalali.

sklearn.ensemble

- [FIX] Fix zero division error in `HistGradientBoostingClassifier` and `HistGradientBoostingRegressor`. #14024 by `Nicolas Hug <NicolasHug>`.

sklearn.impute

- [Fix] Fixed a bug in *impute.SimpleImputer* and *impute.IterativeImputer* so that no errors are thrown when there are missing values in training data. #13974 by Frank Hoang <fhoang7>.

sklearn.inspection

- [Fix] Fixed a bug in *inspection.plot_partial_dependence* where target parameter was not being taken into account for multiclass problems. #14393 by Guillem G. Subies.

sklearn.linear_model

- [Fix] Fixed a bug in *linear_model.LogisticRegressionCV* where refit=False would fail depending on the 'multiclass' and 'penalty' parameters (regression introduced in 0.21). #14087 by Nicolas Hug.
- [Fix] Compatibility fix for *linear_model.ARDRegression* and Scipy>=1.3.0. Adapts to upstream changes to the default pinvh cutoff threshold which otherwise results in poor accuracy in some cases. #14067 by Tim Staley.

sklearn.neighbors

- [Fix] Fixed a bug in *neighbors.NeighborhoodComponentsAnalysis* where the validation of initial parameters n_components, max_iter and tol required too strict types. #14092 by Jérémie du Boisberranger.

sklearn.tree

- [Fix] Fixed bug in *tree.export_text* when the tree has one feature and a single feature name is passed in. #14053 by Thomas Fan.
- [Fix] Fixed an issue with plot_tree where it displayed entropy calculations even for gini criterion in DecisionTreeClassifiers. #13947 by Frank Hoang.

1.7.5 Version 0.21.2

24 May 2019

Changelog

sklearn.decomposition

- [Fix] Fixed a bug in *cross_decomposition.CCA* improving numerical stability when Y is close to zero. #13903 by Thomas Fan.

sklearn.metrics

- [Fix] Fixed a bug in *metrics.pairwise.euclidean_distances* where a part of the distance matrix was left un-instanciated for suffiently large float32 datasets (regression introduced in 0.21). #13910 by Jérémie du Boisberranger.

sklearn.preprocessing

- [Fix] Fixed a bug in *preprocessing.OneHotEncoder* where the new `drop` parameter was not reflected in `get_feature_names`. #13894 by James Myatt.

sklearn.utils.sparsefuncs

- [Fix] Fixed a bug where `min_max_axis` would fail on 32-bit systems for certain large inputs. This affects *preprocessing.MaxAbsScaler*, *preprocessing.normalize* and *preprocessing.LabelBinarizer*. #13741 by Roddy MacSween.

1.7.6 Version 0.21.1

17 May 2019

This is a bug-fix release to primarily resolve some packaging issues in version 0.21.0. It also includes minor documentation improvements and some bug fixes.

Changelog

sklearn.inspection

- [Fix] Fixed a bug in *inspection.partial_dependence* to only check classifier and not regressor for the multiclass-multioutput case. #14309 by Guillaume Lemaitre.

sklearn.metrics

- [Fix] Fixed a bug in *metrics.pairwise_distances* where it would raise `AttributeError` for boolean metrics when `X` had a boolean dtype and `Y == None`. #13864 by Paresh Mathur.
- [Fix] Fixed two bugs in *metrics.pairwise_distances* when `n_jobs > 1`. First it used to return a distance matrix with same dtype as input, even for integer dtype. Then the diagonal was not zeros for euclidean metric when `Y` is `X`. #13877 by Jérémie du Boisberranger.

sklearn.neighbors

- [Fix] Fixed a bug in *neighbors.KernelDensity* which could not be restored from a pickle if `sample_weight` had been used. #13772 by Aditya Vyas.

1.7.7 Version 0.21.0

May 2019

Changed models

The following estimators and functions, when fit with the same data and parameters, may produce different models from the previous version. This often occurs due to changes in the modelling logic (bug fixes or enhancements), or in random sampling procedures.

- *discriminant_analysis.LinearDiscriminantAnalysis* for multiclass classification. [FIX]
- *discriminant_analysis.LinearDiscriminantAnalysis* with 'eigen' solver. [FIX]
- *linear_model.BayesianRidge* [FIX]
- Decision trees and derived ensembles when both `max_depth` and `max_leaf_nodes` are set. [FIX]
- *linear_model.LogisticRegression* and *linear_model.LogisticRegressionCV* with 'saga' solver. [FIX]
- *ensemble.GradientBoostingClassifier* [FIX]
- *sklearn.feature_extraction.text.HashingVectorizer*, *sklearn.feature_extraction.text.TfidfVectorizer*, and *sklearn.feature_extraction.text.CountVectorizer* [FIX]
- *neural_network.MLPClassifier* [FIX]
- *svm.SVC.decision_function* and *multiclass.OneVsOneClassifier.decision_function*. [FIX]
- *linear_model.SGDClassifier* and any derived classifiers. [FIX]
- Any model using the `linear_model._sag.sag_solver` function with a `0` seed, including *linear_model.LogisticRegression*, *linear_model.LogisticRegressionCV*, *linear_model.Ridge*, and *linear_model.RidgeCV* with 'sag' solver. [FIX]
- *linear_model.RidgeCV* when using generalized cross-validation with sparse inputs. [FIX]

Details are listed in the changelog below.

(While we are trying to better inform users by providing this information, we cannot assure that this list is complete.)

Known Major Bugs

- The default `max_iter` for *linear_model.LogisticRegression* is too small for many solvers given the default `tol`. In particular, we accidentally changed the default `max_iter` for the liblinear solver from 1000 to 100 iterations in #3591 released in version 0.16. In a future release we hope to choose better default `max_iter` and `tol` heuristically depending on the solver (see #13317).

Changelog

Support for Python 3.4 and below has been officially dropped.

sklearn.base

- [API CHANGE] The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). #13157 by Hanmin Qin.

sklearn.calibration

- [Enhancement] Added support to bin the data passed into *calibration.calibration_curve* by quantiles instead of uniformly between 0 and 1. #13086 by Scott Cole.
- [Enhancement] Allow n-dimensional arrays as input for `calibration.CalibratedClassifierCV`. #13485 by William de Vazelhes.

sklearn.cluster

- [Major Feature] A new clustering algorithm: *cluster.OPTICS*: an algorithm related to *cluster.DBSCAN*, that has hyperparameters easier to set and that scales better, by Shane, Adrin Jalali, Erich Schubert, Hanmin Qin, and Assia Benbihi.
- [Fix] Fixed a bug where *cluster.Birch* could occasionally raise an AttributeError. #13651 by Joel Nothman.
- [Fix] Fixed a bug in *cluster.KMeans* where empty clusters weren't correctly relocated when using sample weights. #13486 by Jérémie du Boisberranger.
- [API Change] The `n_components_` attribute in *cluster.AgglomerativeClustering* and *cluster.FeatureAgglomeration* has been renamed to `n_connected_components_`. #13427 by Stephane Couvreur.
- [Enhancement] *cluster.AgglomerativeClustering* and *cluster.FeatureAgglomeration* now accept a `distance_threshold` parameter which can be used to find the clusters instead of `n_clusters`. #9069 by Vathsala Achar and Adrin Jalali.

sklearn.compose

- [API Change] *compose.ColumnTransformer* is no longer an experimental feature. #13835 by Hanmin Qin.

sklearn.datasets

- [Fix] Added support for 64-bit group IDs and pointers in SVMLight files. #10727 by Bryan K Woods.
- [Fix] *datasets.load_sample_images* returns images with a deterministic order. #13250 by Thomas Fan.

sklearn.decomposition

- [Enhancement] *decomposition.KernelPCA* now has deterministic output (resolved sign ambiguity in eigenvalue decomposition of the kernel matrix). #13241 by Aurélien Bellet.
- [Fix] Fixed a bug in *decomposition.KernelPCA*, `fit().transform()` now produces the correct output (the same as `fit_transform()`) in case of non-removed zero eigenvalues (`remove_zero_eig=False`). `fit_inverse_transform` was also accelerated by using the same trick as `fit_transform` to compute the transform of `X`. #12143 by Sylvain Marié
- [Fix] Fixed a bug in *decomposition.NMF* where `init = 'nndsvd'`, `init = 'nndsvda'`, and `init = 'nndsvdar'` are allowed when `n_components < n_features` instead of `n_components <= min(n_samples, n_features)`. #11650 by Hossein Pourbozorg and Zijie (ZJ) Poh.

- [API CHANGE] The default value of the `init` argument in *decomposition.non_negative_factorization* will change from `random` to `None` in version 0.23 to make it consistent with *decomposition.NMF*. A FutureWarning is raised when the default value is used. #12988 by Zijie (ZJ) Poh.

sklearn.discriminant_analysis

- [ENHANCEMENT] *discriminant_analysis.LinearDiscriminantAnalysis* now preserves `float32` and `float64` dtypes. #8769 and #11000 by Thibault Sejourne
- [FIX] A `ChangedBehaviourWarning` is now raised when *discriminant_analysis.LinearDiscriminantAnalysis* is given as parameter `n_components > min(n_features, n_classes - 1)`, and `n_components` is changed to `min(n_features, n_classes - 1)` if so. Previously the change was made, but silently. #11526 by William de Vazelhes.
- [FIX] Fixed a bug in *discriminant_analysis.LinearDiscriminantAnalysis* where the predicted probabilities would be incorrectly computed in the multiclass case. #6848, by Agamemnon Krasoulis and `Guillaume Lemaitre <glemaitre>`.
- [FIX] Fixed a bug in *discriminant_analysis.LinearDiscriminantAnalysis* where the predicted probabilities would be incorrectly computed with `eigen` solver. #11727, by Agamemnon Krasoulis.

sklearn.dummy

- [FIX] Fixed a bug in *dummy.DummyClassifier* where the `predict_proba` method was returning int32 array instead of float64 for the `stratified` strategy. #13266 by Christos Aridas.
- [FIX] Fixed a bug in *dummy.DummyClassifier* where it was throwing a dimension mismatch error in prediction time if a column vector `y` with `shape=(n, 1)` was given at `fit` time. #13545 by Nick Sorros and Adrin Jalali.

sklearn.ensemble

- [MAJOR FEATURE] Add two new implementations of gradient boosting trees: *ensemble.HistGradientBoostingClassifier* and *ensemble.HistGradientBoostingRegressor*. The implementation of these estimators is inspired by LightGBM and can be orders of magnitude faster than *ensemble.GradientBoostingRegressor* and *ensemble.GradientBoostingClassifier* when the number of samples is larger than tens of thousands of samples. The API of these new estimators is slightly different, and some of the features from *ensemble.GradientBoostingClassifier* and *ensemble.GradientBoostingRegressor* are not yet supported.

  These new estimators are experimental, which means that their results or their API might change without any deprecation cycle. To use them, you need to explicitly import `enable_hist_gradient_boosting`:

```
>>> # explicitly require this experimental feature
>>> from sklearn.experimental import enable_hist_gradient_boosting  # noqa
>>> # now you can import normally from sklearn.ensemble
>>> from sklearn.ensemble import HistGradientBoostingClassifier
```

  #12807 by Nicolas Hug.

- [FEATURE] Add *ensemble.VotingRegressor* which provides an equivalent of *ensemble.VotingClassifier* for regression problems. #12513 by Ramil Nugmanov and Mohamed Ali Jamaoui.

- [Efficiency] Make *ensemble.IsolationForest* prefer threads over processes when running with `n_jobs > 1` as the underlying decision tree fit calls do release the GIL. This changes reduces memory usage and communication overhead. #12543 by Isaac Storch and Olivier Grisel.
- [Efficiency] Make *ensemble.IsolationForest* more memory efficient by avoiding keeping in memory each tree prediction. #13260 by Nicolas Goix.
- [Efficiency] *ensemble.IsolationForest* now uses chunks of data at prediction step, thus capping the memory usage. #13283 by Nicolas Goix.
- [Efficiency] *sklearn.ensemble.GradientBoostingClassifier* and *sklearn.ensemble.GradientBoostingRegressor* now keep the input `y` as `float64` to avoid it being copied internally by trees. #13524 by Adrin Jalali.
- [Enhancement] Minimized the validation of X in *ensemble.AdaBoostClassifier* and *ensemble.AdaBoostRegressor* #13174 by Christos Aridas.
- [Enhancement] *ensemble.IsolationForest* now exposes `warm_start` parameter, allowing iterative addition of trees to an isolation forest. #13496 by Peter Marko.
- [Fix] The values of `feature_importances_` in all random forest based models (i.e. *ensemble.RandomForestClassifier*, *ensemble.RandomForestRegressor*, *ensemble.ExtraTreesClassifier*, *ensemble.ExtraTreesRegressor*, *ensemble.RandomTreesEmbedding*, *ensemble.GradientBoostingClassifier*, and *ensemble.GradientBoostingRegressor*) now:
  - sum up to `1`
  - all the single node trees in feature importance calculation are ignored
  - in case all trees have only one single node (i.e. a root node), feature importances will be an array of all zeros.

  #13636 and #13620 by Adrin Jalali.
- [Fix] Fixed a bug in *ensemble.GradientBoostingClassifier* and *ensemble.GradientBoostingRegressor*, which didn't support scikit-learn estimators as the initial estimator. Also added support of initial estimator which does not support sample weights. #12436 by Jérémie du Boisberranger and #12983 by Nicolas Hug.
- [Fix] Fixed the output of the average path length computed in *ensemble.IsolationForest* when the input is either 0, 1 or 2. #13251 by Albert Thomas and joshuakennethjones.
- [Fix] Fixed a bug in *ensemble.GradientBoostingClassifier* where the gradients would be incorrectly computed in multiclass classification problems. #12715 by Nicolas Hug.
- [Fix] Fixed a bug in *ensemble.GradientBoostingClassifier* where validation sets for early stopping were not sampled with stratification. #13164 by Nicolas Hug.
- [Fix] Fixed a bug in *ensemble.GradientBoostingClassifier* where the default initial prediction of a multiclass classifier would predict the classes priors instead of the log of the priors. #12983 by Nicolas Hug.
- [Fix] Fixed a bug in *ensemble.RandomForestClassifier* where the `predict` method would error for multiclass multioutput forests models if any targets were strings. #12834 by Elizabeth Sander.
- [Fix] Fixed a bug in `ensemble.gradient_boosting.LossFunction` and `ensemble.gradient_boosting.LeastSquaresError` where the default value of `learning_rate` in `update_terminal_regions` is not consistent with the document and the caller functions. Note however that directly using these loss functions is deprecated. #6463 by movelikeriver.
- [Fix] `ensemble.partial_dependence` (and consequently the new version *sklearn.inspection.partial_dependence*) now takes sample weights into account for the partial dependence computation when the gradient boosting model has been trained with sample weights. #13193 by Samuel O. Ronsin.

- [API CHANGE] `ensemble.partial_dependence` and `ensemble.plot_partial_dependence` are now deprecated in favor of *inspection.partial_dependence* and *inspection.plot_partial_dependence*. #12599 by Trevor Stephens and Nicolas Hug.
- [FIX] *ensemble.VotingClassifier* and *ensemble.VotingRegressor* were failing during `fit` in one of the estimators was set to `None` and `sample_weight` was not `None`. #13779 by Guillaume Lemaitre.
- [API CHANGE] *ensemble.VotingClassifier* and *ensemble.VotingRegressor* accept `'drop'` to disable an estimator in addition to `None` to be consistent with other estimators (i.e., *pipeline.FeatureUnion* and *compose.ColumnTransformer*). #13780 by Guillaume Lemaitre.

sklearn.externals

- [API CHANGE] Deprecated `externals.six` since we have dropped support for Python 2.7. #12916 by Hanmin Qin.

sklearn.feature_extraction

- [FIX] If `input='file'` or `input='filename'`, and a callable is given as the `analyzer`, *sklearn.feature_extraction.text.HashingVectorizer*, *sklearn.feature_extraction.text.TfidfVectorizer*, and *sklearn.feature_extraction.text.CountVectorizer* now read the data from the file(s) and then pass it to the given `analyzer`, instead of passing the file name(s) or the file object(s) to the analyzer. #13641 by Adrin Jalali.

sklearn.impute

- [MAJOR FEATURE] Added *impute.IterativeImputer*, which is a strategy for imputing missing values by modeling each feature with missing values as a function of other features in a round-robin fashion. #8478 and #12177 by Sergey Feldman and Ben Lawson.

  The API of IterativeImputer is experimental and subject to change without any deprecation cycle. To use them, you need to explicitly import `enable_iterative_imputer`:

```
>>> from sklearn.experimental import enable_iterative_imputer  # noqa
>>> # now you can import normally from sklearn.impute
>>> from sklearn.impute import IterativeImputer
```

- [FEATURE] The *impute.SimpleImputer* and *impute.IterativeImputer* have a new parameter `'add_indicator'`, which simply stacks a *impute.MissingIndicator* transform into the output of the imputer's transform. That allows a predictive estimator to account for missingness. #12583, #13601 by Danylo Baibak.
- [FIX] In *impute.MissingIndicator* avoid implicit densification by raising an exception if input is sparse add `missing_values` property is set to 0. #13240 by Bartosz Telenczuk.
- [FIX] Fixed two bugs in *impute.MissingIndicator*. First, when `X` is sparse, all the non-zero non missing values used to become explicit False in the transformed data. Then, when `features='missing-only'`, all features used to be kept if there were no missing values at all. #13562 by Jérémie du Boisberranger.

sklearn.inspection

(new subpackage)

- [FEATURE] Partial dependence plots (*inspection.plot_partial_dependence*) are now supported for any regressor or classifier (provided that they have a `predict_proba` method). #12599 by Trevor Stephens and Nicolas Hug.

sklearn.isotonic

- [FEATURE] Allow different dtypes (such as float32) in *isotonic.IsotonicRegression*. #8769 by Vlad Niculae

sklearn.linear_model

- [ENHANCEMENT] *linear_model.Ridge* now preserves `float32` and `float64` dtypes. #8769 and #11000 by Guillaume Lemaitre, and Joan Massich
- [FEATURE] *linear_model.LogisticRegression* and *linear_model.LogisticRegressionCV* now support Elastic-Net penalty, with the 'saga' solver. #11646 by Nicolas Hug.
- [FEATURE] Added *linear_model.lars_path_gram*, which is *linear_model.lars_path* in the sufficient stats mode, allowing users to compute *linear_model.lars_path* without providing `X` and `y`. #11699 by Kuai Yu.
- [EFFICIENCY] `linear_model.make_dataset` now preserves `float32` and `float64` dtypes, reducing memory consumption in stochastic gradient, SAG and SAGA solvers. #8769 and #11000 by Nelle Varoquaux, Arthur Imbert, Guillaume Lemaitre, and Joan Massich
- [ENHANCEMENT] *linear_model.LogisticRegression* now supports an unregularized objective when `penalty='none'` is passed. This is equivalent to setting `C=np.inf` with l2 regularization. Not supported by the liblinear solver. #12860 by Nicolas Hug.
- [ENHANCEMENT] `sparse_cg` solver in *linear_model.Ridge* now supports fitting the intercept (i.e. `fit_intercept=True`) when inputs are sparse. #13336 by Bartosz Telenczuk.
- [ENHANCEMENT] The coordinate descent solver used in `Lasso`, `ElasticNet`, etc. now issues a `ConvergenceWarning` when it completes without meeting the desired toleranbce. #11754 and #13397 by Brent Fagan and Adrin Jalali.
- [FIX] Fixed a bug in *linear_model.LogisticRegression* and *linear_model.LogisticRegressionCV* with 'saga' solver, where the weights would not be correctly updated in some cases. #11646 by Tom Dupre la Tour.
- [FIX] Fixed the posterior mean, posterior covariance and returned regularization parameters in *linear_model.BayesianRidge*. The posterior mean and the posterior covariance were not the ones computed with the last update of the regularization parameters and the returned regularization parameters were not the final ones. Also fixed the formula of the log marginal likelihood used to compute the score when `compute_score=True`. #12174 by Albert Thomas.
- [FIX] Fixed a bug in *linear_model.LassoLarsIC*, where user input `copy_X=False` at instance creation would be overridden by default parameter value `copy_X=True` in `fit`. #12972 by Lucio Fernandez-Arjona
- [FIX] Fixed a bug in *linear_model.LinearRegression* that was not returning the same coeffecients and intercepts with `fit_intercept=True` in sparse and dense case. #13279 by Alexandre Gramfort
- [FIX] Fixed a bug in *linear_model.HuberRegressor* that was broken when `X` was of dtype bool. #13328 by Alexandre Gramfort.

- [FIX] Fixed a performance issue of `saga` and `sag` solvers when called in a `joblib.Parallel` setting with `n_jobs > 1` and `backend="threading"`, causing them to perform worse than in the sequential case. #13389 by Pierre Glaser.
- [FIX] Fixed a bug in `linear_model.stochastic_gradient.BaseSGDClassifier` that was not deterministic when trained in a multi-class setting on several threads. #13422 by Clément Doumouro.
- [FIX] Fixed bug in *linear_model.ridge_regression*, *linear_model.Ridge* and *linear_model.RidgeClassifier* that caused unhandled exception for arguments `return_intercept=True` and `solver=auto` (default) or any other solver different from `sag`. #13363 by Bartosz Telenczuk
- [FIX] *linear_model.ridge_regression* will now raise an exception if `return_intercept=True` and solver is different from `sag`. Previously, only warning was issued. #13363 by Bartosz Telenczuk
- [FIX] *linear_model.ridge_regression* will choose `sparse_cg` solver for sparse inputs when `solver=auto` and `sample_weight` is provided (previously `cholesky` solver was selected). #13363 by Bartosz Telenczuk
- [API CHANGE] The use of *linear_model.lars_path* with `X=None` while passing `Gram` is deprecated in version 0.21 and will be removed in version 0.23. Use *linear_model.lars_path_gram* instead. #11699 by Kuai Yu.
- [API CHANGE] *linear_model.logistic_regression_path* is deprecated in version 0.21 and will be removed in version 0.23. #12821 by Nicolas Hug.
- [FIX] *linear_model.RidgeCV* with generalized cross-validation now correctly fits an intercept when `fit_intercept=True` and the design matrix is sparse. #13350 by Jérôme Dockès

sklearn.manifold

- [EFFICIENCY] Make `manifold.tsne.trustworthiness` use an inverted index instead of an `np.where` lookup to find the rank of neighbors in the input space. This improves efficiency in particular when computed with lots of neighbors and/or small datasets. #9907 by William de Vazelhes.

sklearn.metrics

- [FEATURE] Added the *metrics.max_error* metric and a corresponding `'max_error'` scorer for single output regression. #12232 by Krishna Sangeeth.
- [FEATURE] Add *metrics.multilabel_confusion_matrix*, which calculates a confusion matrix with true positive, false positive, false negative and true negative counts for each class. This facilitates the calculation of set-wise metrics such as recall, specificity, fall out and miss rate. #11179 by Shangwu Yao and Joel Nothman.
- [FEATURE] *metrics.jaccard_score* has been added to calculate the Jaccard coefficient as an evaluation metric for binary, multilabel and multiclass tasks, with an interface analogous to *metrics.f1_score*. #13151 by Gaurav Dhingra and Joel Nothman.
- [FEATURE] Added *metrics.pairwise.haversine_distances* which can be accessed with `metric='pairwise'` through *metrics.pairwise_distances* and estimators. (Haversine distance was previously available for nearest neighbors calculation.) #12568 by Wei Xue, Emmanuel Arias and Joel Nothman.
- [EFFICIENCY] Faster *metrics.pairwise_distances* with `n_jobs` > 1 by using a thread-based backend, instead of process-based backends. #8216 by Pierre Glaser and Romuald Menuet
- [EFFICIENCY] The pairwise manhattan distances with sparse input now uses the BLAS shipped with scipy instead of the bundled BLAS. #12732 by Jérémie du Boisberranger

- [ENHANCEMENT] Use label `accuracy` instead of `micro-average` on *metrics.classification_report* to avoid confusion. `micro-average` is only shown for multi-label or multi-class with a subset of classes because it is otherwise identical to accuracy. #12334 by Emmanuel Arias, Joel Nothman and Andreas Müller
- [ENHANCEMENT] Added `beta` parameter to *metrics.homogeneity_completeness_v_measure* and *metrics.v_measure_score* to configure the tradeoff between homogeneity and completeness. #13607 by Stephane Couvreur and and Ivan Sanchez.
- [FIX] The metric *metrics.r2_score* is degenerate with a single sample and now it returns NaN and raises *exceptions.UndefinedMetricWarning*. #12855 by Pawel Sendyk.
- [FIX] Fixed a bug where *metrics.brier_score_loss* will sometimes return incorrect result when there's only one class in `y_true`. #13628 by Hanmin Qin.
- [FIX] Fixed a bug in *metrics.label_ranking_average_precision_score* where sample_weight wasn't taken into account for samples with degenerate labels. #13447 by Dan Ellis.
- [API CHANGE] The parameter `labels` in *metrics.hamming_loss* is deprecated in version 0.21 and will be removed in version 0.23. #10580 by Reshama Shaikh and Sandra Mitrovic.
- [FIX] The function *metrics.pairwise.euclidean_distances*, and therefore several estimators with `metric='euclidean'`, suffered from numerical precision issues with `float32` features. Precision has been increased at the cost of a small drop of performance. #13554 by @Celelibi and Jérémie du Boisberranger.
- [API CHANGE] *metrics.jaccard_similarity_score* is deprecated in favour of the more consistent *metrics.jaccard_score*. The former behavior for binary and multiclass targets is broken. #13151 by Joel Nothman.

sklearn.mixture

- [FIX] Fixed a bug in `mixture.BaseMixture` and therefore on estimators based on it, i.e. *mixture.GaussianMixture* and *mixture.BayesianGaussianMixture*, where `fit_predict` and `fit.predict` were not equivalent. #13142 by Jérémie du Boisberranger.

sklearn.model_selection

- [FEATURE] Classes *GridSearchCV* and *RandomizedSearchCV* now allow for refit=callable to add flexibility in identifying the best estimator. See *Balance model complexity and cross-validated score*. #11354 by Wenhao Zhang, Joel Nothman and Adrin Jalali.
- [ENHANCEMENT] Classes *GridSearchCV*, *RandomizedSearchCV*, and methods *cross_val_score*, *cross_val_predict*, *cross_validate*, now print train scores when `return_train_scores` is True and `verbose` > 2. For *learning_curve*, and *validation_curve* only the latter is required. #12613 and #12669 by Marc Torrellas.
- [ENHANCEMENT] Some *CV splitter* classes and `model_selection.train_test_split` now raise `ValueError` when the resulting training set is empty. #12861 by Nicolas Hug.
- [FIX] Fixed a bug where *model_selection.StratifiedKFold* shuffles each class's samples with the same `random_state`, making `shuffle=True` ineffective. #13124 by Hanmin Qin.
- [FIX] Added ability for *model_selection.cross_val_predict* to handle multi-label (and multioutput-multiclass) targets with `predict_proba`-type methods. #8773 by Stephen Hoover.
- [FIX] Fixed an issue in *cross_val_predict* where `method="predict_proba"` returned always `0.0` when one of the classes was excluded in a cross-validation fold. #13366 by Guillaume Fournier

sklearn.multiclass

- [Fix] Fixed an issue in *multiclass.OneVsOneClassifier.decision_function* where the decision_function value of a given sample was different depending on whether the decision_function was evaluated on the sample alone or on a batch containing this same sample due to the scaling used in decision_function. #10440 by Jonathan Ohayon.

sklearn.multioutput

- [Fix] Fixed a bug in *multioutput.MultiOutputClassifier* where the `predict_proba` method incorrectly checked for `predict_proba` attribute in the estimator object. #12222 by Rebekah Kim

sklearn.neighbors

- [Major Feature] Added *neighbors.NeighborhoodComponentsAnalysis* for metric learning, which implements the Neighborhood Components Analysis algorithm. #10058 by William de Vazelhes and John Chiotellis.
- [API Change] Methods in *neighbors.NearestNeighbors* : *kneighbors*, *radius_neighbors*, *kneighbors_graph*, *radius_neighbors_graph* now raise `NotFittedError`, rather than `AttributeError`, when called before `fit` #12279 by Krishna Sangeeth.

sklearn.neural_network

- [Fix] Fixed a bug in *neural_network.MLPClassifier* and *neural_network.MLPRegressor* where the option `shuffle=False` was being ignored. #12582 by Sam Waterbury.
- [Fix] Fixed a bug in *neural_network.MLPClassifier* where validation sets for early stopping were not sampled with stratification. In the multilabel case however, splits are still not stratified. #13164 by Nicolas Hug.

sklearn.pipeline

- [Feature] *pipeline.Pipeline* can now use indexing notation (e.g. `my_pipeline[0:-1]`) to extract a subsequence of steps as another Pipeline instance. A Pipeline can also be indexed directly to extract a particular step (e.g. `my_pipeline['svc']`), rather than accessing `named_steps`. #2568 by Joel Nothman.
- [Feature] Added optional parameter `verbose` in *pipeline.Pipeline*, *compose.ColumnTransformer* and *pipeline.FeatureUnion* and corresponding `make_` helpers for showing progress and timing of each step. #11364 by Baze Petrushev, Karan Desai, Joel Nothman, and Thomas Fan.
- [Enhancement] *pipeline.Pipeline* now supports using `'passthrough'` as a transformer, with the same effect as `None`. #11144 by Thomas Fan.
- [Enhancement] *pipeline.Pipeline* implements `__len__` and therefore `len(pipeline)` returns the number of steps in the pipeline. #13439 by Lakshya KD.

sklearn.preprocessing

- [Feature] *preprocessing.OneHotEncoder* now supports dropping one feature per category with a new drop parameter. #12908 by Drew Johnston.

- [EFFICIENCY] *preprocessing.OneHotEncoder* and *preprocessing.OrdinalEncoder* now handle pandas DataFrames more efficiently. #13253 by @maikia.
- [EFFICIENCY] Make *preprocessing.MultiLabelBinarizer* cache class mappings instead of calculating it every time on the fly. #12116 by Ekaterina Krivich and Joel Nothman.
- [EFFICIENCY] *preprocessing.PolynomialFeatures* now supports compressed sparse row (CSR) matrices as input for degrees 2 and 3. This is typically much faster than the dense case as it scales with matrix density and expansion degree (on the order of density^degree), and is much, much faster than the compressed sparse column (CSC) case. #12197 by Andrew Nystrom.
- [EFFICIENCY] Speed improvement in *preprocessing.PolynomialFeatures*, in the dense case. Also added a new parameter `order` which controls output order for further speed performances. #12251 by Tom Dupre la Tour.
- [FIX] Fixed the calculation overflow when using a float16 dtype with *preprocessing.StandardScaler*. #13007 by Raffaello Baluyot
- [FIX] Fixed a bug in *preprocessing.QuantileTransformer* and *preprocessing.quantile_transform* to force n_quantiles to be at most equal to n_samples. Values of n_quantiles larger than n_samples were either useless or resulting in a wrong approximation of the cumulative distribution function estimator. #13333 by Albert Thomas.
- [API CHANGE] The default value of `copy` in *preprocessing.quantile_transform* will change from False to True in 0.23 in order to make it more consistent with the default `copy` values of other functions in `preprocessing` and prevent unexpected side effects by modifying the value of `X` inplace. #13459 by Hunter McGushion.

sklearn.svm

- [FIX] Fixed an issue in *svm.SVC.decision_function* when `decision_function_shape='ovr'`. The decision_function value of a given sample was different depending on whether the decision_function was evaluated on the sample alone or on a batch containing this same sample due to the scaling used in decision_function. #10440 by Jonathan Ohayon.

sklearn.tree

- [FEATURE] Decision Trees can now be plotted with matplotlib using *tree.plot_tree* without relying on the `dot` library, removing a hard-to-install dependency. #8508 by Andreas Müller.
- [FEATURE] Decision Trees can now be exported in a human readable textual format using *tree.export_text*. #6261 by `Giuseppe Vettigli <JustGlowing>`.
- [FEATURE] `get_n_leaves()` and `get_depth()` have been added to `tree.BaseDecisionTree` and consequently all estimators based on it, including *tree.DecisionTreeClassifier*, *tree.DecisionTreeRegressor*, *tree.ExtraTreeClassifier*, and *tree.ExtraTreeRegressor*. #12300 by Adrin Jalali.
- [FIX] Trees and forests did not previously `predict` multi-output classification targets with string labels, despite accepting them in `fit`. #11458 by Mitar Milutinovic.
- [FIX] Fixed an issue with `tree.BaseDecisionTree` and consequently all estimators based on it, including *tree.DecisionTreeClassifier*, *tree.DecisionTreeRegressor*, *tree.ExtraTreeClassifier*, and *tree.ExtraTreeRegressor*, where they used to exceed the given `max_depth` by 1 while expanding the tree if `max_leaf_nodes` and `max_depth` were both specified by the user. Please note that this also affects all ensemble methods using decision trees. #12344 by Adrin Jalali.

sklearn.utils

- [FEATURE] *utils.resample* now accepts a `stratify` parameter for sampling according to class distributions. #13549 by Nicolas Hug.
- [API CHANGE] Deprecated `warn_on_dtype` parameter from *utils.check_array* and *utils.check_X_y*. Added explicit warning for dtype conversion in `check_pairwise_arrays` if the `metric` being passed is a pairwise boolean metric. #13382 by Prathmesh Savale.

Multiple modules

- [MAJOR FEATURE] The `__repr__()` method of all estimators (used when calling `print(estimator)`) has been entirely re-written, building on Python's pretty printing standard library. All parameters are printed by default, but this can be altered with the `print_changed_only` option in *sklearn.set_config*. #11705 by Nicolas Hug.
- [MAJOR FEATURE] Add estimators tags: these are annotations of estimators that allow programmatic inspection of their capabilities, such as sparse matrix support, supported output types and supported methods. Estimator tags also determine the tests that are run on an estimator when `check_estimator` is called. Read more in the *User Guide*. #8022 by Andreas Müller.
- [EFFICIENCY] Memory copies are avoided when casting arrays to a different dtype in multiple estimators. #11973 by Roman Yurchak.
- [FIX] Fixed a bug in the implementation of the `our_rand_r` helper function that was not behaving consistently across platforms. #13422 by Madhura Parikh and Clément Doumouro.

Miscellaneous

- [ENHANCEMENT] Joblib is no longer vendored in scikit-learn, and becomes a dependency. Minimal supported version is joblib 0.11, however using version >= 0.13 is strongly recommended. #13531 by Roman Yurchak.

Changes to estimator checks

These changes mostly affect library developers.

- Add `check_fit_idempotent` to *check_estimator*, which checks that when `fit` is called twice with the same data, the ouput of `predict`, `predict_proba`, `transform`, and `decision_function` does not change. #12328 by Nicolas Hug
- Many checks can now be disabled or configured with *Estimator Tags*. #8022 by Andreas Müller.

Code and Documentation Contributors

Thanks to everyone who has contributed to the maintenance and improvement of the project since version 0.20, including:

adanhawth, Aditya Vyas, Adrin Jalali, Agamemnon Krasoulis, Albert Thomas, Alberto Torres, Alexandre Gramfort, amourav, Andrea Navarrete, Andreas Mueller, Andrew Nystrom, assiaben, Aurélien Bellet, Bartosz Michałowski, Bartosz Telenczuk, bauks, BenjaStudio, bertrandhaut, Bharat Raghunathan, brentfagan, Bryan Woods, Cat Chenal, Cheuk Ting Ho, Chris Choe, Christos Aridas, Clément Doumouro, Cole Smith, Connossor, Corey Levinson, Dan Ellis, Dan Stine, Danylo Baibak, daten-kieker, Denis Kataev, Didi Bar-Zev, Dillon Gardner, Dmitry Mottl, Dmitry Vukolov, Dougal J. Sutherland, Dowon, drewmjohnston, Dror Atariah, Edward J Brown, Ekaterina Krivich, Elizabeth Sander, Emmanuel Arias, Eric Chang, Eric Larson, Erich Schubert, esvhd, Falak, Feda Curic, Federico Caselli,

Frank Hoang, Fibinse Xavier‘, Finn O’Shea, Gabriel Marzinotto, Gabriel Vacaliuc, Gabriele Calvo, Gael Varoquaux, GauravAhlawat, Giuseppe Vettigli, Greg Gandenberger, Guillaume Fournier, Guillaume Lemaitre, Gustavo De Mari Pereira, Hanmin Qin, haroldfox, hhu-luqi, Hunter McGushion, Ian Sanders, JackLangerman, Jacopo Notarstefano, jakirkham, James Bourbeau, Jan Koch, Jan S, janvanrijn, Jarrod Millman, jdethurens, jeremiedbb, JF, joaak, Joan Massich, Joel Nothman, Jonathan Ohayon, Joris Van den Bossche, josephsalmon, Jérémie Méhault, Katrin Leinweber, ken, kms15, Koen, Kossori Aruku, Krishna Sangeeth, Kuai Yu, Kulbear, Kushal Chauhan, Kyle Jackson, Lakshya KD, Leandro Hermida, Lee Yi Jie Joel, Lily Xiong, Lisa Sarah Thomas, Loic Esteve, louib, luk-f-a, maikia, mail-liam, Manimaran, Manuel López-Ibáñez, Marc Torrellas, Marco Gaido, Marco Gorelli, MarcoGorelli, marineLM, Mark Hannel, Martin Gubri, Masstran, mathurinm, Matthew Roeschke, Max Copeland, melsyt, mferrari3, Mickaël Schoentgen, Ming Li, Mitar, Mohammad Aftab, Mohammed AbdelAal, Mohammed Ibraheem, Muhammad Hassaan Rafique, mwestt, Naoya Iijima, Nicholas Smith, Nicolas Goix, Nicolas Hug, Nikolay Shebanov, Oleksandr Pavlyk, Oliver Rausch, Olivier Grisel, Orestis, Osman, Owen Flanagan, Paul Paczuski, Pavel Soriano, pavlos kallis, Pawel Sendyk, peay, Peter, Peter Cock, Peter Hausamann, Peter Marko, Pierre Glaser, pierretallotte, Pim de Haan, Piotr Szymański, Prabakaran Kumaresshan, Pradeep Reddy Raamana, Prathmesh Savale, Pulkit Maloo, Quentin Batista, Radostin Stoyanov, Raf Baluyot, Rajdeep Dua, Ramil Nugmanov, Raúl García Calvo, Rebekah Kim, Reshama Shaikh, Rohan Lekhwani, Rohan Singh, Rohan Varma, Rohit Kapoor, Roman Feldbauer, Roman Yurchak, Romuald M, Roopam Sharma, Ryan, Rüdiger Busche, Sam Waterbury, Samuel O. Ronsin, SandroCasagrande, Scott Cole, Scott Lowe, Sebastian Raschka, Shangwu Yao, Shivam Kotwalia, Shiyu Duan, smarie, Sriharsha Hatwar, Stephen Hoover, Stephen Tierney, Stéphane Couvreur, surgan12, SylvainLan, TakingItCasual, Tashay Green, thibsej, Thomas Fan, Thomas J Fan, Thomas Moreau, Tom Dupré la Tour, Tommy, Tulio Casagrande, Umar Farouk Umar, Utkarsh Upadhyay, Vinayak Mehta, Vishaal Kapoor, Vivek Kumar, Vlad Niculae, vqean3, Wenhao Zhang, William de Vazelhes, xhan, Xing Han Lu, xinyuliu12, Yaroslav Halchenko, Zach Griffith, Zach Miller, Zayd Hammoudeh, Zhuyi Xue, Zijie (ZJ) Poh, ^__^

1.7.8 Version 0.20.4

July 30, 2019

This is a bug-fix release with some bug fixes applied to version 0.20.3.

Changelog

The bundled version of joblib was upgraded from 0.13.0 to 0.13.2.

sklearn.cluster

- [Fix] Fixed a bug in *cluster.KMeans* where KMeans++ initialisation could rarely result in an IndexError. #11756 by Joel Nothman.

sklearn.compose

- [Fix] Fixed an issue in *compose.ColumnTransformer* where using DataFrames whose column order differs between :func:`fit` and :func:`transform` could lead to silently passing incorrect columns to the `remainder` transformer. #14237 by `Andreas Schuderer <schuderer>`.

sklearn.decomposition

- [Fix] Fixed a bug in *cross_decomposition.CCA* improving numerical stability when `Y` is close to zero. #13903 by Thomas Fan.

sklearn.model_selection

- [Fix] Fixed a bug where *model_selection.StratifiedKFold* shuffles each class's samples with the same `random_state`, making `shuffle=True` ineffective. #13124 by Hanmin Qin.

sklearn.neighbors

- [Fix] Fixed a bug in *neighbors.KernelDensity* which could not be restored from a pickle if `sample_weight` had been used. #13772 by Aditya Vyas.

1.7.9 Version 0.20.3

March 1, 2019

This is a bug-fix release with some minor documentation improvements and enhancements to features released in 0.20.0.

Changelog

sklearn.cluster

- [Fix] Fixed a bug in *cluster.KMeans* where computation was single threaded when `n_jobs > 1` or `n_jobs = -1`. #12949 by Prabakaran Kumaresshan.

sklearn.compose

- [Fix] Fixed a bug in *compose.ColumnTransformer* to handle negative indexes in the columns list of the transformers. #12946 by Pierre Tallotte.

sklearn.covariance

- [Fix] Fixed a regression in *covariance.graphical_lasso* so that the case `n_features=2` is handled correctly. #13276 by Aurélien Bellet.

sklearn.decomposition

- [Fix] Fixed a bug in *decomposition.sparse_encode* where computation was single threaded when `n_jobs > 1` or `n_jobs = -1`. #13005 by Prabakaran Kumaresshan.

sklearn.datasets

- [Efficiency] *sklearn.datasets.fetch_openml* now loads data by streaming, avoiding high memory usage. #13312 by Joris Van den Bossche.

sklearn.feature_extraction

- [Fix] Fixed a bug in *feature_extraction.text.CountVectorizer* which would result in the sparse feature matrix having conflicting `indptr` and `indices` precisions under very large vocabularies. #11295 by Gabriel Vacaliuc.

sklearn.impute

- [Fix] add support for non-numeric data in *sklearn.impute.MissingIndicator* which was not supported while *sklearn.impute.SimpleImputer* was supporting this for some imputation strategies. #13046 by Guillaume Lemaitre.

sklearn.linear_model

- [Fix] Fixed a bug in *linear_model.MultiTaskElasticNet* and *linear_model.MultiTaskLasso* which were breaking when `warm_start = True`. #12360 by Aakanksha Joshi.

sklearn.preprocessing

- [Fix] Fixed a bug in *preprocessing.KBinsDiscretizer* where `strategy='kmeans'` fails with an error during transformation due to unsorted bin edges. #13134 by Sandro Casagrande.
- [Fix] Fixed a bug in *preprocessing.OneHotEncoder* where the deprecation of `categorical_features` was handled incorrectly in combination with `handle_unknown='ignore'`. #12881 by Joris Van den Bossche.
- [Fix] Bins whose width are too small (i.e., <= 1e-8) are removed with a warning in *preprocessing.KBinsDiscretizer*. #13165 by Hanmin Qin.

sklearn.svm

- [Fix] Fixed a bug in *svm.SVC*, *svm.NuSVC*, *svm.SVR*, *svm.NuSVR* and *svm.OneClassSVM* where the `scale` option of parameter `gamma` is erroneously defined as `1 / (n_features * X.std())`. It's now defined as `1 / (n_features * X.var())`. #13221 by Hanmin Qin.

Code and Documentation Contributors

With thanks to:

Adrin Jalali, Agamemnon Krasoulis, Albert Thomas, Andreas Mueller, Aurélien Bellet, bertrandhaut, Bharat Raghunathan, Dowon, Emmanuel Arias, Fibinse Xavier, Finn O'Shea, Gabriel Vacaliuc, Gael Varoquaux, Guillaume Lemaitre, Hanmin Qin, joaak, Joel Nothman, Joris Van den Bossche, Jérémie Méhault, kms15, Kossori Aruku, Lakshya KD, maikia, Manuel López-Ibáñez, Marco Gorelli, MarcoGorelli, mferrari3, Mickaël Schoentgen, Nicolas Hug, pavlos kallis, Pierre Glaser, pierretallotte, Prabakaran Kumaresshan, Reshama Shaikh, Rohit Kapoor, Roman Yurchak, SandroCasagrande, Tashay Green, Thomas Fan, Vishaal Kapoor, Zhuyi Xue, Zijie (ZJ) Poh

1.7.10 Version 0.20.2

December 20, 2018

This is a bug-fix release with some minor documentation improvements and enhancements to features released in 0.20.0.

Changed models

The following estimators and functions, when fit with the same data and parameters, may produce different models from the previous version. This often occurs due to changes in the modelling logic (bug fixes or enhancements), or in random sampling procedures.

- *sklearn.neighbors* when `metric=='jaccard'` (bug fix)
- use of `'seuclidean'` or `'mahalanobis'` metrics in some cases (bug fix)

Changelog

sklearn.compose

- [Fix] Fixed an issue in *compose.make_column_transformer* which raises unexpected error when columns is pandas Index or pandas Series. #12704 by Hanmin Qin.

sklearn.metrics

- [Fix] Fixed a bug in *metrics.pairwise_distances* and *metrics.pairwise_distances_chunked* where parameters `V` of `"seuclidean"` and `VI` of `"mahalanobis"` metrics were computed after the data was split into chunks instead of being pre-computed on whole data. #12701 by Jeremie du Boisberranger.

sklearn.neighbors

- [Fix] Fixed *sklearn.neighbors.DistanceMetric* jaccard distance function to return 0 when two all-zero vectors are compared. #12685 by Thomas Fan.

sklearn.utils

- [Fix] Calling *utils.check_array* on `pandas.Series` with categorical data, which raised an error in 0.20.0, now returns the expected output again. #12699 by Joris Van den Bossche.

Code and Documentation Contributors

With thanks to:

adanhawth, Adrin Jalali, Albert Thomas, Andreas Mueller, Dan Stine, Feda Curic, Hanmin Qin, Jan S, jeremiedbb, Joel Nothman, Joris Van den Bossche, josephsalmon, Katrin Leinweber, Loic Esteve, Muhammad Hassaan Rafique, Nicolas Hug, Olivier Grisel, Paul Paczuski, Reshama Shaikh, Sam Waterbury, Shivam Kotwalia, Thomas Fan

1.7.11 Version 0.20.1

November 21, 2018

This is a bug-fix release with some minor documentation improvements and enhancements to features released in 0.20.0. Note that we also include some API changes in this release, so you might get some extra warnings after updating from 0.20.0 to 0.20.1.

Changed models

The following estimators and functions, when fit with the same data and parameters, may produce different models from the previous version. This often occurs due to changes in the modelling logic (bug fixes or enhancements), or in random sampling procedures.

- *decomposition.IncrementalPCA* (bug fix)

Changelog

sklearn.cluster

- [Efficiency] make *cluster.MeanShift* no longer try to do nested parallelism as the overhead would hurt performance significantly when `n_jobs > 1`. #12159 by Olivier Grisel.
- [Fix] Fixed a bug in *cluster.DBSCAN* with precomputed sparse neighbors graph, which would add explicitly zeros on the diagonal even when already present. #12105 by Tom Dupre la Tour.

sklearn.compose

- [Fix] Fixed an issue in *compose.ColumnTransformer* when stacking columns with types not convertible to a numeric. #11912 by Adrin Jalali.
- [API Change] *compose.ColumnTransformer* now applies the `sparse_threshold` even if all transformation results are sparse. #12304 by Andreas Müller.
- [API Change] *compose.make_column_transformer* now expects `(transformer, columns)` instead of `(columns, transformer)` to keep consistent with *compose.ColumnTransformer*. #12339 by Adrin Jalali.

sklearn.datasets

- [Fix] *datasets.fetch_openml* to correctly use the local cache. #12246 by Jan N. van Rijn.
- [Fix] *datasets.fetch_openml* to correctly handle ignore attributes and row id attributes. #12330 by Jan N. van Rijn.
- [Fix] Fixed integer overflow in *datasets.make_classification* for values of `n_informative` parameter larger than 64. #10811 by Roman Feldbauer.
- [Fix] Fixed olivetti faces dataset `DESCR` attribute to point to the right location in *datasets.fetch_olivetti_faces*. #12441 by Jérémie du Boisberranger
- [Fix] *datasets.fetch_openml* to retry downloading when reading from local cache fails. #12517 by Thomas Fan.

sklearn.decomposition

- [FIX] Fixed a regression in *decomposition.IncrementalPCA* where 0.20.0 raised an error if the number of samples in the final batch for fitting IncrementalPCA was smaller than n_components. #12234 by Ming Li.

sklearn.ensemble

- [FIX] Fixed a bug mostly affecting *ensemble.RandomForestClassifier* where `class_weight='balanced_subsample'` failed with more than 32 classes. #12165 by Joel Nothman.
- [FIX] Fixed a bug affecting *ensemble.BaggingClassifier*, *ensemble.BaggingRegressor* and *ensemble.IsolationForest*, where `max_features` was sometimes rounded down to zero. #12388 by Connor Tann.

sklearn.feature_extraction

- [FIX] Fixed a regression in v0.20.0 where *feature_extraction.text.CountVectorizer* and other text vectorizers could error during stop words validation with custom preprocessors or tokenizers. #12393 by Roman Yurchak.

sklearn.linear_model

- [FIX] *linear_model.SGDClassifier* and variants with `early_stopping=True` would not use a consistent validation split in the multiclass case and this would cause a crash when using those estimators as part of parallel parameter search or cross-validation. #12122 by Olivier Grisel.
- [FIX] Fixed a bug affecting `SGDClassifier` in the multiclass case. Each one-versus-all step is run in a `joblib.Parallel` call and mutating a common parameter, causing a segmentation fault if called within a backend using processes and not threads. We now use `require=sharedmem` at the `joblib.Parallel` instance creation. #12518 by Pierre Glaser and Olivier Grisel.

sklearn.metrics

- [FIX] Fixed a bug in `metrics.pairwise.pairwise_distances_argmin_min` which returned the square root of the distance when the metric parameter was set to "euclidean". #12481 by Jérémie du Boisberranger.
- [FIX] Fixed a bug in `metrics.pairwise.pairwise_distances_chunked` which didn't ensure the diagonal is zero for euclidean distances. #12612 by Andreas Müller.
- [API CHANGE] The *metrics.calinski_harabaz_score* has been renamed to *metrics.calinski_harabasz_score* and will be removed in version 0.23. #12211 by Lisa Thomas, Mark Hannel and Melissa Ferrari.

sklearn.mixture

- [FIX] Ensure that the `fit_predict` method of *mixture.GaussianMixture* and *mixture.BayesianGaussianMixture* always yield assignments consistent with `fit` followed by `predict` even if the convergence criterion is too loose or not met. #12451 by Olivier Grisel.

sklearn.neighbors

- [FIX] force the parallelism backend to `threading` for *neighbors.KDTree* and *neighbors.BallTree* in Python 2.7 to avoid pickling errors caused by the serialization of their methods. #12171 by Thomas Moreau.

sklearn.preprocessing

- [FIX] Fixed bug in *preprocessing.OrdinalEncoder* when passing manually specified categories. #12365 by Joris Van den Bossche.
- [FIX] Fixed bug in *preprocessing.KBinsDiscretizer* where the `transform` method mutates the `_encoder` attribute. The `transform` method is now thread safe. #12514 by Hanmin Qin.
- [FIX] Fixed a bug in *preprocessing.PowerTransformer* where the Yeo-Johnson transform was incorrect for lambda parameters outside of `[0, 2]` #12522 by Nicolas Hug.
- [FIX] Fixed a bug in *preprocessing.OneHotEncoder* where transform failed when set to ignore unknown numpy strings of different lengths #12471 by Gabriel Marzinotto.
- [API CHANGE] The default value of the `method` argument in *preprocessing.power_transform* will be changed from `box-cox` to `yeo-johnson` to match *preprocessing.PowerTransformer* in version 0.23. A FutureWarning is raised when the default value is used. #12317 by Eric Chang.

sklearn.utils

- [FIX] Use float64 for mean accumulator to avoid floating point precision issues in *preprocessing.StandardScaler* and *decomposition.IncrementalPCA* when using float32 datasets. #12338 by bauks.
- [FIX] Calling *utils.check_array* on `pandas.Series`, which raised an error in 0.20.0, now returns the expected output again. #12625 by Andreas Müller

Miscellaneous

- [FIX] When using site joblib by setting the environment variable `SKLEARN_SITE_JOBLIB`, added compatibility with joblib 0.11 in addition to 0.12+. #12350 by Joel Nothman and Roman Yurchak.
- [FIX] Make sure to avoid raising `FutureWarning` when calling `np.vstack` with numpy 1.16 and later (use list comprehensions instead of generator expressions in many locations of the scikit-learn code base). #12467 by Olivier Grisel.
- [API CHANGE] Removed all mentions of `sklearn.externals.joblib`, and deprecated joblib methods exposed in `sklearn.utils`, except for *utils.parallel_backend* and *utils.register_parallel_backend*, which allow users to configure parallel computation in scikit-learn. Other functionalities are part of joblib. package and should be used directly, by installing it. The goal of this change is to prepare for unvendoring joblib in future version of scikit-learn. #12345 by Thomas Moreau

Code and Documentation Contributors

With thanks to:

^__^, Adrin Jalali, Andrea Navarrete, Andreas Mueller, bauks, BenjaStudio, Cheuk Ting Ho, Connossor, Corey Levinson, Dan Stine, daten-kieker, Denis Kataev, Dillon Gardner, Dmitry Vukolov, Dougal J. Sutherland, Edward J Brown,

Eric Chang, Federico Caselli, Gabriel Marzinotto, Gael Varoquaux, GauravAhlawat, Gustavo De Mari Pereira, Hanmin Qin, haroldfox, JackLangerman, Jacopo Notarstefano, janvanrijn, jdethurens, jeremiedbb, Joel Nothman, Joris Van den Bossche, Koen, Kushal Chauhan, Lee Yi Jie Joel, Lily Xiong, mail-liam, Mark Hannel, melsyt, Ming Li, Nicholas Smith, Nicolas Hug, Nikolay Shebanov, Oleksandr Pavlyk, Olivier Grisel, Peter Hausamann, Pierre Glaser, Pulkit Maloo, Quentin Batista, Radostin Stoyanov, Ramil Nugmanov, Rebekah Kim, Reshama Shaikh, Rohan Singh, Roman Feldbauer, Roman Yurchak, Roopam Sharma, Sam Waterbury, Scott Lowe, Sebastian Raschka, Stephen Tierney, SylvainLan, TakingItCasual, Thomas Fan, Thomas Moreau, Tom Dupré la Tour, Tulio Casagrande, Utkarsh Upadhyay, Xing Han Lu, Yaroslav Halchenko, Zach Miller

1.7.12 Version 0.20.0

September 25, 2018

This release packs in a mountain of bug fixes, features and enhancements for the Scikit-learn library, and improvements to the documentation and examples. Thanks to our contributors!

This release is dedicated to the memory of Raghav Rajagopalan.

Warning: Version 0.20 is the last version of scikit-learn to support Python 2.7 and Python 3.4. Scikit-learn 0.21 will require Python 3.5 or higher.

Highlights

We have tried to improve our support for common data-science use-cases including missing values, categorical variables, heterogeneous data, and features/targets with unusual distributions. Missing values in features, represented by NaNs, are now accepted in column-wise preprocessing such as scalers. Each feature is fitted disregarding NaNs, and data containing NaNs can be transformed. The new `impute` module provides estimators for learning despite missing data.

`ColumnTransformer` handles the case where different features or columns of a pandas.DataFrame need different preprocessing. String or pandas Categorical columns can now be encoded with *`OneHotEncoder`* or *`OrdinalEncoder`*.

`TransformedTargetRegressor` helps when the regression target needs to be transformed to be modeled. *`PowerTransformer`* and *`KBinsDiscretizer`* join *`QuantileTransformer`* as non-linear transformations.

Beyond this, we have added *sample_weight* support to several estimators (including *`KMeans`*, *`BayesianRidge`* and *`KernelDensity`*) and improved stopping criteria in others (including *`MLPRegressor`*, *`GradientBoostingRegressor`* and *`SGDRegressor`*).

This release is also the first to be accompanied by a *Glossary of Common Terms and API Elements* developed by Joel Nothman. The glossary is a reference resource to help users and contributors become familiar with the terminology and conventions used in Scikit-learn.

Sorry if your contribution didn't make it into the highlights. There's a lot here...

Changed models

The following estimators and functions, when fit with the same data and parameters, may produce different models from the previous version. This often occurs due to changes in the modelling logic (bug fixes or enhancements), or in random sampling procedures.

- *`cluster.MeanShift`* (bug fix)
- *`decomposition.IncrementalPCA`* in Python 2 (bug fix)

- *decomposition.SparsePCA* (bug fix)
- *ensemble.GradientBoostingClassifier* (bug fix affecting feature importances)
- *isotonic.IsotonicRegression* (bug fix)
- *linear_model.ARDRegression* (bug fix)
- *linear_model.LogisticRegressionCV* (bug fix)
- *linear_model.OrthogonalMatchingPursuit* (bug fix)
- *linear_model.PassiveAggressiveClassifier* (bug fix)
- *linear_model.PassiveAggressiveRegressor* (bug fix)
- *linear_model.Perceptron* (bug fix)
- *linear_model.SGDClassifier* (bug fix)
- *linear_model.SGDRegressor* (bug fix)
- *metrics.roc_auc_score* (bug fix)
- *metrics.roc_curve* (bug fix)
- neural_network.BaseMultilayerPerceptron (bug fix)
- *neural_network.MLPClassifier* (bug fix)
- *neural_network.MLPRegressor* (bug fix)
- The v0.19.0 release notes failed to mention a backwards incompatibility with *model_selection.StratifiedKFold* when `shuffle=True` due to #7823.

Details are listed in the changelog below.

(While we are trying to better inform users by providing this information, we cannot assure that this list is complete.)

Known Major Bugs

- #11924: *linear_model.LogisticRegressionCV* with `solver='lbfgs'` and `multi_class='multinomial'` may be non-deterministic or otherwise broken on macOS. This appears to be the case on Travis CI servers, but has not been confirmed on personal MacBooks! This issue has been present in previous releases.
- #9354: *metrics.pairwise.euclidean_distances* (which is used several times throughout the library) gives results with poor precision, which particularly affects its use with 32-bit float inputs. This became more problematic in versions 0.18 and 0.19 when some algorithms were changed to avoid casting 32-bit data into 64-bit.

Changelog

Support for Python 3.3 has been officially dropped.

sklearn.cluster

- [Major Feature] *cluster.AgglomerativeClustering* now supports Single Linkage clustering via `linkage='single'`. #9372 by Leland McInnes and Steve Astels.
- [Feature] *cluster.KMeans* and *cluster.MiniBatchKMeans* now support sample weights via new parameter `sample_weight` in `fit` function. #10933 by Johannes Hansen.

- [EFFICIENCY] *cluster.KMeans*, *cluster.MiniBatchKMeans* and *cluster.k_means* passed with `algorithm='full'` now enforces row-major ordering, improving runtime. #10471 by Gaurav Dhingra.
- [EFFICIENCY] *cluster.DBSCAN* now is parallelized according to `n_jobs` regardless of `algorithm`. #8003 by Joël Billaud.
- [ENHANCEMENT] *cluster.KMeans* now gives a warning if the number of distinct clusters found is smaller than `n_clusters`. This may occur when the number of distinct points in the data set is actually smaller than the number of cluster one is looking for. #10059 by Christian Braune.
- [FIX] Fixed a bug where the `fit` method of *cluster.AffinityPropagation* stored cluster centers as 3d array instead of 2d array in case of non-convergence. For the same class, fixed undefined and arbitrary behavior in case of training data where all samples had equal similarity. #9612. By Jonatan Samoocha.
- [FIX] Fixed a bug in *cluster.spectral_clustering* where the normalization of the spectrum was using a division instead of a multiplication. #8129 by Jan Margeta, Guillaume Lemaitre, and Devansh D..
- [FIX] Fixed a bug in `cluster.k_means_elkan` where the returned `iteration` was 1 less than the correct value. Also added the missing `n_iter_` attribute in the docstring of *cluster.KMeans*. #11353 by Jeremie du Boisberranger.
- [FIX] Fixed a bug in *cluster.mean_shift* where the assigned labels were not deterministic if there were multiple clusters with the same intensities. #11901 by Adrin Jalali.
- [API CHANGE] Deprecate `pooling_func` unused parameter in *cluster.AgglomerativeClustering*. #9875 by Kumar Ashutosh.

sklearn.compose

- New module.
- [MAJOR FEATURE] Added *compose.ColumnTransformer*, which allows to apply different transformers to different columns of arrays or pandas DataFrames. #9012 by Andreas Müller and Joris Van den Bossche, and #11315 by Thomas Fan.
- [MAJOR FEATURE] Added the *compose.TransformedTargetRegressor* which transforms the target y before fitting a regression model. The predictions are mapped back to the original space via an inverse transform. #9041 by Andreas Müller and Guillaume Lemaitre.

sklearn.covariance

- [EFFICIENCY] Runtime improvements to *covariance.GraphicalLasso*. #9858 by Steven Brown.
- [API CHANGE] The `covariance.graph_lasso`, `covariance.GraphLasso` and `covariance.GraphLassoCV` have been renamed to *covariance.graphical_lasso*, *covariance.GraphicalLasso* and *covariance.GraphicalLassoCV* respectively and will be removed in version 0.22. #9993 by Artiem Krinitsyn

sklearn.datasets

- [MAJOR FEATURE] Added *datasets.fetch_openml* to fetch datasets from OpenML. OpenML is a free, open data sharing platform and will be used instead of mldata as it provides better service availability. #9908 by Andreas Müller and Jan N. van Rijn.
- [FEATURE] In *datasets.make_blobs*, one can now pass a list to the `n_samples` parameter to indicate the number of samples to generate per cluster. #8617 by Maskani Filali Mohamed and Konstantinos Katrioplas.

- [FEATURE] Add `filename` attribute to `datasets` that have a CSV file. #9101 by alex-33 and Maskani Filali Mohamed.
- [FEATURE] `return_X_y` parameter has been added to several dataset loaders. #10774 by Chris Catalfo.
- [FIX] Fixed a bug in *`datasets.load_boston`* which had a wrong data point. #10795 by Takeshi Yoshizawa.
- [FIX] Fixed a bug in *`datasets.load_iris`* which had two wrong data points. #11082 by Sadhana Srinivasan and Hanmin Qin.
- [FIX] Fixed a bug in *`datasets.fetch_kddcup99`*, where data were not properly shuffled. #9731 by Nicolas Goix.
- [FIX] Fixed a bug in *`datasets.make_circles`*, where no odd number of data points could be generated. #10045 by Christian Braune.
- [API CHANGE] Deprecated `sklearn.datasets.fetch_mldata` to be removed in version 0.22. mldata.org is no longer operational. Until removal it will remain possible to load cached datasets. #11466 by Joel Nothman.

sklearn.decomposition

- [FEATURE] *`decomposition.dict_learning`* functions and models now support positivity constraints. This applies to the dictionary and sparse code. #6374 by John Kirkham.
- [FEATURE] [FIX] *`decomposition.SparsePCA`* now exposes `normalize_components`. When set to True, the train and test data are centered with the train mean respectively during the fit phase and the transform phase. This fixes the behavior of SparsePCA. When set to False, which is the default, the previous abnormal behaviour still holds. The False value is for backward compatibility and should not be used. #11585 by Ivan Panico.
- [EFFICIENCY] Efficiency improvements in *`decomposition.dict_learning`*. #11420 and others by John Kirkham.
- [FIX] Fix for uninformative error in *`decomposition.IncrementalPCA`*: now an error is raised if the number of components is larger than the chosen batch size. The `n_components=None` case was adapted accordingly. #6452. By Wally Gauze.
- [FIX] Fixed a bug where the `partial_fit` method of *`decomposition.IncrementalPCA`* used integer division instead of float division on Python 2. #9492 by James Bourbeau.
- [FIX] In *`decomposition.PCA`* selecting a n_components parameter greater than the number of samples now raises an error. Similarly, the `n_components=None` case now selects the minimum of `n_samples` and `n_features`. #8484 by Wally Gauze.
- [FIX] Fixed a bug in *`decomposition.PCA`* where users will get unexpected error with large datasets when `n_components='mle'` on Python 3 versions. #9886 by Hanmin Qin.
- [FIX] Fixed an underflow in calculating KL-divergence for *`decomposition.NMF`* #10142 by Tom Dupre la Tour.
- [FIX] Fixed a bug in *`decomposition.SparseCoder`* when running OMP sparse coding in parallel using read-only memory mapped datastructures. #5956 by Vighnesh Birodkar and Olivier Grisel.

sklearn.discriminant_analysis

- [EFFICIENCY] Memory usage improvement for `_class_means` and `_class_cov` in `discriminant_analysis`. #10898 by Nanxin Chen.

sklearn.dummy

- [FEATURE] *dummy.DummyRegressor* now has a `return_std` option in its `predict` method. The returned standard deviations will be zeros.
- [FEATURE] *dummy.DummyClassifier* and *dummy.DummyRegressor* now only require X to be an object with finite length or shape. #9832 by Vrishank Bhardwaj.
- [FEATURE] *dummy.DummyClassifier* and *dummy.DummyRegressor* can now be scored without supplying test samples. #11951 by Rüdiger Busche.

sklearn.ensemble

- [FEATURE] *ensemble.BaggingRegressor* and *ensemble.BaggingClassifier* can now be fit with missing/non-finite values in X and/or multi-output Y to support wrapping pipelines that perform their own imputation. #9707 by Jimmy Wan.
- [FEATURE] *ensemble.GradientBoostingClassifier* and *ensemble.GradientBoostingRegressor* now support early stopping via `n_iter_no_change`, `validation_fraction` and `tol`. #7071 by Raghav RV
- [FEATURE] Added `named_estimators_` parameter in *ensemble.VotingClassifier* to access fitted estimators. #9157 by Herilalaina Rakotoarison.
- [FIX] Fixed a bug when fitting *ensemble.GradientBoostingClassifier* or *ensemble.GradientBoostingRegressor* with `warm_start=True` which previously raised a segmentation fault due to a non-conversion of CSC matrix into CSR format expected by `decision_function`. Similarly, Fortran-ordered arrays are converted to C-ordered arrays in the dense case. #9991 by Guillaume Lemaitre.
- [FIX] Fixed a bug in *ensemble.GradientBoostingRegressor* and *ensemble.GradientBoostingClassifier* to have feature importances summed and then normalized, rather than normalizing on a per-tree basis. The previous behavior over-weighted the Gini importance of features that appear in later stages. This issue only affected feature importances. #11176 by Gil Forsyth.
- [API CHANGE] The default value of the `n_estimators` parameter of *ensemble.RandomForestClassifier*, *ensemble.RandomForestRegressor*, *ensemble.ExtraTreesClassifier*, *ensemble.ExtraTreesRegressor*, and *ensemble.RandomTreesEmbedding* will change from 10 in version 0.20 to 100 in 0.22. A FutureWarning is raised when the default value is used. #11542 by Anna Ayzenshtat.
- [API CHANGE] Classes derived from `ensemble.BaseBagging`. The attribute `estimators_samples_` will return a list of arrays containing the indices selected for each bootstrap instead of a list of arrays containing the mask of the samples selected for each bootstrap. Indices allows to repeat samples while mask does not allow this functionality. #9524 by Guillaume Lemaitre.
- [FIX] `ensemble.BaseBagging` where one could not deterministically reproduce `fit` result using the object attributes when `random_state` is set. #9723 by Guillaume Lemaitre.

sklearn.feature_extraction

- [FEATURE] Enable the call to *get_feature_names* in unfitted *feature_extraction.text.CountVectorizer* initialized with a vocabulary. #10908 by Mohamed Maskani.
- [ENHANCEMENT] `idf_` can now be set on a *feature_extraction.text.TfidfTransformer*. #10899 by Sergey Melderis.

- [Fix] Fixed a bug in *feature_extraction.image.extract_patches_2d* which would throw an exception if `max_patches` was greater than or equal to the number of all possible patches rather than simply returning the number of possible patches. #10101 by Varun Agrawal
- [Fix] Fixed a bug in *feature_extraction.text.CountVectorizer*, *feature_extraction.text.TfidfVectorizer*, *feature_extraction.text.HashingVectorizer* to support 64 bit sparse array indexing necessary to process large datasets with more than $2 \cdot 10^9$ tokens (words or n-grams). #9147 by Claes-Fredrik Mannby and Roman Yurchak.
- [Fix] Fixed bug in *feature_extraction.text.TfidfVectorizer* which was ignoring the parameter `dtype`. In addition, *feature_extraction.text.TfidfTransformer* will preserve `dtype` for floating and raise a warning if `dtype` requested is integer. #10441 by Mayur Kulkarni and Guillaume Lemaitre.

sklearn.feature_selection

- [Feature] Added select K best features functionality to *feature_selection.SelectFromModel*. #6689 by Nihar Sheth and Quazi Rahman.
- [Feature] Added `min_features_to_select` parameter to *feature_selection.RFECV* to bound evaluated features counts. #11293 by Brent Yi.
- [Feature] *feature_selection.RFECV*'s fit method now supports *groups*. #9656 by Adam Greenhall.
- [Fix] Fixed computation of `n_features_to_compute` for edge case with tied CV scores in *feature_selection.RFECV*. #9222 by Nick Hoh.

sklearn.gaussian_process

- [Efficiency] In *gaussian_process.GaussianProcessRegressor*, method `predict` is faster when using `return_std=True` in particular more when called several times in a row. #9234 by andrewww and Minghui Liu.

sklearn.impute

- New module, adopting `preprocessing.Imputer` as *impute.SimpleImputer* with minor changes (see under preprocessing below).
- [Major Feature] Added *impute.MissingIndicator* which generates a binary indicator for missing values. #8075 by Maniteja Nandana and Guillaume Lemaitre.
- [Feature] The *impute.SimpleImputer* has a new strategy, `'constant'`, to complete missing values with a fixed one, given by the `fill_value` parameter. This strategy supports numeric and non-numeric data, and so does the `'most_frequent'` strategy now. #11211 by Jeremie du Boisberranger.

sklearn.isotonic

- [Fix] Fixed a bug in *isotonic.IsotonicRegression* which incorrectly combined weights when fitting a model to data involving points with identical X values. #9484 by Dallas Card

sklearn.linear_model

- [FEATURE] *linear_model.SGDClassifier*, *linear_model.SGDRegressor*, *linear_model.PassiveAggressiveClassifier*, *linear_model.PassiveAggressiveRegressor* and *linear_model.Perceptron* now expose `early_stopping`, `validation_fraction` and `n_iter_no_change` parameters, to stop optimization monitoring the score on a validation set. A new learning rate `"adaptive"` strategy divides the learning rate by 5 each time `n_iter_no_change` consecutive epochs fail to improve the model. #9043 by Tom Dupre la Tour.
- [FEATURE] Add `sample_weight` parameter to the fit method of *linear_model.BayesianRidge* for weighted linear regression. #10112 by Peter St. John.
- [FIX] Fixed a bug in `logistic.logistic_regression_path` to ensure that the returned coefficients are correct when `multiclass='multinomial'`. Previously, some of the coefficients would override each other, leading to incorrect results in *linear_model.LogisticRegressionCV*. #11724 by Nicolas Hug.
- [FIX] Fixed a bug in *linear_model.LogisticRegression* where when using the parameter `multi_class='multinomial'`, the `predict_proba` method was returning incorrect probabilities in the case of binary outcomes. #9939 by Roger Westover.
- [FIX] Fixed a bug in *linear_model.LogisticRegressionCV* where the `score` method always computes accuracy, not the metric given by the `scoring` parameter. #10998 by Thomas Fan.
- [FIX] Fixed a bug in *linear_model.LogisticRegressionCV* where the 'ovr' strategy was always used to compute cross-validation scores in the multiclass setting, even if `'multinomial'` was set. #8720 by William de Vazelhes.
- [FIX] Fixed a bug in *linear_model.OrthogonalMatchingPursuit* that was broken when setting `normalize=False`. #10071 by Alexandre Gramfort.
- [FIX] Fixed a bug in *linear_model.ARDRegression* which caused incorrectly updated estimates for the standard deviation and the coefficients. #10153 by Jörg Döpfert.
- [FIX] Fixed a bug in *linear_model.ARDRegression* and *linear_model.BayesianRidge* which caused NaN predictions when fitted with a constant target. #10095 by Jörg Döpfert.
- [FIX] Fixed a bug in *linear_model.RidgeClassifierCV* where the parameter `store_cv_values` was not implemented though it was documented in `cv_values` as a way to set up the storage of cross-validation values for different alphas. #10297 by Mabel Villalba-Jiménez.
- [FIX] Fixed a bug in *linear_model.ElasticNet* which caused the input to be overridden when using parameter `copy_X=True` and `check_input=False`. #10581 by Yacine Mazari.
- [FIX] Fixed a bug in *sklearn.linear_model.Lasso* where the coefficient had wrong shape when `fit_intercept=False`. #10687 by Martin Hahn.
- [FIX] Fixed a bug in *sklearn.linear_model.LogisticRegression* where the `multi_class='multinomial'` with binary output `with warm_start=True` #10836 by Aishwarya Srinivasan.
- [FIX] Fixed a bug in *linear_model.RidgeCV* where using integer `alphas` raised an error. #10397 by Mabel Villalba-Jiménez.
- [FIX] Fixed condition triggering gap computation in *linear_model.Lasso* and *linear_model.ElasticNet* when working with sparse matrices. #10992 by Alexandre Gramfort.
- [FIX] Fixed a bug in *linear_model.SGDClassifier*, *linear_model.SGDRegressor*, *linear_model.PassiveAggressiveClassifier*, *linear_model.PassiveAggressiveRegressor* and *linear_model.Perceptron*, where the stopping criterion was stopping the algorithm before convergence. A parameter `n_iter_no_change` was added and set by default to 5. Previous behavior is equivalent to setting the parameter to 1. #9043 by Tom Dupre la Tour.

- [FIX] Fixed a bug where liblinear and libsvm-based estimators would segfault if passed a scipy.sparse matrix with 64-bit indices. They now raise a ValueError. #11327 by Karan Dhingra and Joel Nothman.
- [API CHANGE] The default values of the `solver` and `multi_class` parameters of *`linear_model.LogisticRegression`* will change respectively from `'liblinear'` and `'ovr'` in version 0.20 to `'lbfgs'` and `'auto'` in version 0.22. A FutureWarning is raised when the default values are used. #11905 by Tom Dupre la Tour and Joel Nothman.
- [API CHANGE] Deprecate `positive=True` option in *`linear_model.Lars`* as the underlying implementation is broken. Use *`linear_model.Lasso`* instead. #9837 by Alexandre Gramfort.
- [API CHANGE] `n_iter_` may vary from previous releases in *`linear_model.LogisticRegression`* with `solver='lbfgs'` and *`linear_model.HuberRegressor`*. For Scipy <= 1.0.0, the optimizer could perform more than the requested maximum number of iterations. Now both estimators will report at most `max_iter` iterations even if more were performed. #10723 by Joel Nothman.

sklearn.manifold

- [EFFICIENCY] Speed improvements for both 'exact' and 'barnes_hut' methods in *`manifold.TSNE`*. #10593 and #10610 by Tom Dupre la Tour.
- [FEATURE] Support sparse input in *`manifold.Isomap.fit`*. #8554 by Leland McInnes.
- [FEATURE] `manifold.t_sne.trustworthiness` accepts metrics other than Euclidean. #9775 by William de Vazelhes.
- [FIX] Fixed a bug in *`manifold.spectral_embedding`* where the normalization of the spectrum was using a division instead of a multiplication. #8129 by Jan Margeta, Guillaume Lemaitre, and Devansh D..
- [API CHANGE] [FEATURE] Deprecate `precomputed` parameter in function `manifold.t_sne.trustworthiness`. Instead, the new parameter `metric` should be used with any compatible metric including 'precomputed', in which case the input matrix `X` should be a matrix of pairwise distances or squared distances. #9775 by William de Vazelhes.
- [API CHANGE] Deprecate `precomputed` parameter in function `manifold.t_sne.trustworthiness`. Instead, the new parameter `metric` should be used with any compatible metric including 'precomputed', in which case the input matrix `X` should be a matrix of pairwise distances or squared distances. #9775 by William de Vazelhes.

sklearn.metrics

- [MAJOR FEATURE] Added the *`metrics.davies_bouldin_score`* metric for evaluation of clustering models without a ground truth. #10827 by Luis Osa.
- [MAJOR FEATURE] Added the *`metrics.balanced_accuracy_score`* metric and a corresponding `'balanced_accuracy'` scorer for binary and multiclass classification. #8066 by @xyguo and Aman Dalmia, and #10587 by Joel Nothman.
- [FEATURE] Partial AUC is available via `max_fpr` parameter in *`metrics.roc_auc_score`*. #3840 by Alexander Niederbühl.
- [FEATURE] A scorer based on *`metrics.brier_score_loss`* is also available. #9521 by Hanmin Qin.
- [FEATURE] Added control over the normalization in *`metrics.normalized_mutual_info_score`* and *`metrics.adjusted_mutual_info_score`* via the `average_method` parameter. In version 0.22, the default normalizer for each will become the *arithmetic* mean of the entropies of each clustering. #11124 by Arya McCarthy.

- [FEATURE] Added `output_dict` parameter in *metrics.classification_report* to return classification statistics as dictionary. #11160 by Dan Barkhorn.
- [FEATURE] *metrics.classification_report* now reports all applicable averages on the given data, including micro, macro and weighted average as well as samples average for multilabel data. #11679 by Alexander Pacha.
- [FEATURE] *metrics.average_precision_score* now supports binary `y_true` other than `{0, 1}` or `{-1, 1}` through `pos_label` parameter. #9980 by Hanmin Qin.
- [FEATURE] *metrics.label_ranking_average_precision_score* now supports `sample_weight`. #10845 by Jose Perez-Parras Toledano.
- [FEATURE] Add `dense_output` parameter to *metrics.pairwise.linear_kernel*. When False and both inputs are sparse, will return a sparse matrix. #10999 by Taylor G Smith.
- [EFFICIENCY] *metrics.silhouette_score* and *metrics.silhouette_samples* are more memory efficient and run faster. This avoids some reported freezes and MemoryErrors. #11135 by Joel Nothman.
- [FIX] Fixed a bug in *metrics.precision_recall_fscore_support* when truncated `range(n_labels)` is passed as value for `labels`. #10377 by Gaurav Dhingra.
- [FIX] Fixed a bug due to floating point error in *metrics.roc_auc_score* with non-integer sample weights. #9786 by Hanmin Qin.
- [FIX] Fixed a bug where *metrics.roc_curve* sometimes starts on y-axis instead of (0, 0), which is inconsistent with the document and other implementations. Note that this will not influence the result from *metrics.roc_auc_score* #10093 by alexryndin and Hanmin Qin.
- [FIX] Fixed a bug to avoid integer overflow. Casted product to 64 bits integer in *metrics.mutual_info_score*. #9772 by Kumar Ashutosh.
- [FIX] Fixed a bug where *metrics.average_precision_score* will sometimes return `nan` when `sample_weight` contains 0. #9980 by Hanmin Qin.
- [FIX] Fixed a bug in *metrics.fowlkes_mallows_score* to avoid integer overflow. Casted return value of `contingency_matrix` to `int64` and computed product of square roots rather than square root of product. #9515 by Alan Liddell and Manh Dao.
- [API CHANGE] Deprecate `reorder` parameter in *metrics.auc* as it's no longer required for *metrics.roc_auc_score*. Moreover using `reorder=True` can hide bugs due to floating point error in the input. #9851 by Hanmin Qin.
- [API CHANGE] In *metrics.normalized_mutual_info_score* and *metrics.adjusted_mutual_info_score*, warn that `average_method` will have a new default value. In version 0.22, the default normalizer for each will become the *arithmetic* mean of the entropies of each clustering. Currently, *metrics.normalized_mutual_info_score* uses the default of `average_method='geometric'`, and *metrics.adjusted_mutual_info_score* uses the default of `average_method='max'` to match their behaviors in version 0.19. #11124 by Arya McCarthy.
- [API CHANGE] The `batch_size` parameter to *metrics.pairwise_distances_argmin_min* and *metrics.pairwise_distances_argmin* is deprecated to be removed in v0.22. It no longer has any effect, as batch size is determined by global `working_memory` config. See *Limiting Working Memory*. #10280 by Joel Nothman and Aman Dalmia.

sklearn.mixture

- [FEATURE] Added function *fit_predict* to *mixture.GaussianMixture* and *mixture.GaussianMixture*, which is essentially equivalent to calling *fit* and *predict*. #10336 by Shu Haoran and Andrew Peng.

- [Fix] Fixed a bug in `mixture.BaseMixture` where the reported `n_iter_` was missing an iteration. It affected *mixture.GaussianMixture* and *mixture.BayesianGaussianMixture*. #10740 by Erich Schubert and Guillaume Lemaitre.
- [Fix] Fixed a bug in `mixture.BaseMixture` and its subclasses *mixture.GaussianMixture* and *mixture.BayesianGaussianMixture* where the `lower_bound_` was not the max lower bound across all initializations (when `n_init > 1`), but just the lower bound of the last initialization. #10869 by Aurélien Géron.

sklearn.model_selection

- [Feature] Add `return_estimator` parameter in *model_selection.cross_validate* to return estimators fitted on each split. #9686 by Aurélien Bellet.
- [Feature] New `refit_time_` attribute will be stored in *model_selection.GridSearchCV* and *model_selection.RandomizedSearchCV* if `refit` is set to `True`. This will allow measuring the complete time it takes to perform hyperparameter optimization and refitting the best model on the whole dataset. #11310 by Matthias Feurer.
- [Feature] Expose `error_score` parameter in *model_selection.cross_validate*, *model_selection.cross_val_score*, *model_selection.learning_curve* and *model_selection.validation_curve* to control the behavior triggered when an error occurs in `model_selection._fit_and_score`. #11576 by Samuel O. Ronsin.
- [Feature] `BaseSearchCV` now has an experimental, private interface to support customized parameter search strategies, through its `_run_search` method. See the implementations in *model_selection.GridSearchCV* and *model_selection.RandomizedSearchCV* and please provide feedback if you use this. Note that we do not assure the stability of this API beyond version 0.20. #9599 by Joel Nothman
- [Enhancement] Add improved error message in *model_selection.cross_val_score* when multiple metrics are passed in `scoring` keyword. #11006 by Ming Li.
- [API Change] The default number of cross-validation folds `cv` and the default number of splits `n_splits` in the *model_selection.KFold*-like splitters will change from 3 to 5 in 0.22 as 3-fold has a lot of variance. #11557 by Alexandre Boucaud.
- [API Change] The default of `iid` parameter of *model_selection.GridSearchCV* and *model_selection.RandomizedSearchCV* will change from `True` to `False` in version 0.22 to correspond to the standard definition of cross-validation, and the parameter will be removed in version 0.24 altogether. This parameter is of greatest practical significance where the sizes of different test sets in cross-validation were very unequal, i.e. in group-based CV strategies. #9085 by Laurent Direr and Andreas Müller.
- [API Change] The default value of the `error_score` parameter in *model_selection.GridSearchCV* and *model_selection.RandomizedSearchCV* will change to `np.NaN` in version 0.22. #10677 by Kirill Zhdanovich.
- [API Change] Changed ValueError exception raised in *model_selection.ParameterSampler* to a UserWarning for case where the class is instantiated with a greater value of `n_iter` than the total space of parameters in the parameter grid. `n_iter` now acts as an upper bound on iterations. #10982 by Juliet Lawton
- [API Change] Invalid input for *model_selection.ParameterGrid* now raises TypeError. #10928 by Solutus Immensus

sklearn.multioutput

- [Major Feature] Added *multioutput.RegressorChain* for multi-target regression. #9257 by Kumar

Ashutosh.

sklearn.naive_bayes

- [Major Feature] Added *naive_bayes.ComplementNB*, which implements the Complement Naive Bayes classifier described in Rennie et al. (2003). #8190 by Michael A. Alcorn.
- [Feature] Add `var_smoothing` parameter in *naive_bayes.GaussianNB* to give a precise control over variances calculation. #9681 by Dmitry Mottl.
- [Fix] Fixed a bug in *naive_bayes.GaussianNB* which incorrectly raised error for prior list which summed to 1. #10005 by Gaurav Dhingra.
- [Fix] Fixed a bug in *naive_bayes.MultinomialNB* which did not accept vector valued pseudocounts (alpha). #10346 by Tobias Madsen

sklearn.neighbors

- [Efficiency] *neighbors.RadiusNeighborsRegressor* and *neighbors.RadiusNeighborsClassifier* are now parallelized according to `n_jobs` regardless of `algorithm`. #10887 by Joël Billaud.
- [Efficiency] `Nearest neighbors` query methods are now more memory efficient when `algorithm='brute'`. #11136 by Joel Nothman and Aman Dalmia.
- [Feature] Add `sample_weight` parameter to the fit method of *neighbors.KernelDensity* to enable weighting in kernel density estimation. #4394 by Samuel O. Ronsin.
- [Feature] Novelty detection with *neighbors.LocalOutlierFactor*: Add a `novelty` parameter to *neighbors.LocalOutlierFactor*. When `novelty` is set to True, *neighbors.LocalOutlierFactor* can then be used for novelty detection, i.e. predict on new unseen data. Available prediction methods are `predict`, `decision_function` and `score_samples`. By default, `novelty` is set to `False`, and only the `fit_predict` method is avaiable. By Albert Thomas.
- [Fix] Fixed a bug in *neighbors.NearestNeighbors* where fitting a NearestNeighbors model fails when a) the distance metric used is a callable and b) the input to the NearestNeighbors model is sparse. #9579 by Thomas Kober.
- [Fix] Fixed a bug so `predict` in *neighbors.RadiusNeighborsRegressor* can handle empty neighbor set when using non uniform weights. Also raises a new warning when no neighbors are found for samples. #9655 by Andreas Bjerre-Nielsen.
- [Fix] [Efficiency] Fixed a bug in `KDTree` construction that results in faster construction and querying times. #11556 by Jake VanderPlas
- [Fix] Fixed a bug in *neighbors.KDTree* and *neighbors.BallTree* where pickled tree objects would change their type to the super class `BinaryTree`. #11774 by Nicolas Hug.

sklearn.neural_network

- [Feature] Add `n_iter_no_change` parameter in `neural_network.BaseMultilayerPerceptron`, *neural_network.MLPRegressor*, and *neural_network.MLPClassifier* to give control over maximum number of epochs to not meet `tol` improvement. #9456 by Nicholas Nadeau.

- [FIX] Fixed a bug in `neural_network.BaseMultilayerPerceptron`, *neural_network.MLPRegressor*, and *neural_network.MLPClassifier* with new `n_iter_no_change` parameter now at 10 from previously hardcoded 2. #9456 by Nicholas Nadeau.
- [FIX] Fixed a bug in *neural_network.MLPRegressor* where fitting quit unexpectedly early due to local minima or fluctuations. #9456 by Nicholas Nadeau

sklearn.pipeline

- [FEATURE] The `predict` method of *pipeline.Pipeline* now passes keyword arguments on to the pipeline's last estimator, enabling the use of parameters such as `return_std` in a pipeline with caution. #9304 by Breno Freitas.
- [API CHANGE] *pipeline.FeatureUnion* now supports `'drop'` as a transformer to drop features. #11144 by Thomas Fan.

sklearn.preprocessing

- [MAJOR FEATURE] Expanded *preprocessing.OneHotEncoder* to allow to encode categorical string features as a numeric array using a one-hot (or dummy) encoding scheme, and added *preprocessing.OrdinalEncoder* to convert to ordinal integers. Those two classes now handle encoding of all feature types (also handles string-valued features) and derives the categories based on the unique values in the features instead of the maximum value in the features. #9151 and #10521 by Vighnesh Birodkar and Joris Van den Bossche.
- [MAJOR FEATURE] Added *preprocessing.KBinsDiscretizer* for turning continuous features into categorical or one-hot encoded features. #7668, #9647, #10195, #10192, #11272, #11467 and #11505. by Henry Lin, Hanmin Qin, Tom Dupre la Tour and Giovanni Giuseppe Costa.
- [MAJOR FEATURE] Added *preprocessing.PowerTransformer*, which implements the Yeo-Johnson and Box-Cox power transformations. Power transformations try to find a set of feature-wise parametric transformations to approximately map data to a Gaussian distribution centered at zero and with unit variance. This is useful as a variance-stabilizing transformation in situations where normality and homoscedasticity are desirable. #10210 by Eric Chang and Maniteja Nandana, and #11520 by Nicolas Hug.
- [MAJOR FEATURE] NaN values are ignored and handled in the following preprocessing methods: *preprocessing.MaxAbsScaler*, *preprocessing.MinMaxScaler*, *preprocessing.RobustScaler*, *preprocessing.StandardScaler*, *preprocessing.PowerTransformer*, *preprocessing.QuantileTransformer* classes and *preprocessing.maxabs_scale*, *preprocessing.minmax_scale*, *preprocessing.robust_scale*, *preprocessing.scale*, *preprocessing.power_transform*, *preprocessing.quantile_transform* functions respectively addressed in issues #11011, #11005, #11308, #11206, #11306, and #10437. By Lucija Gregov and Guillaume Lemaitre.
- [FEATURE] *preprocessing.PolynomialFeatures* now supports sparse input. #10452 by Aman Dalmia and Joel Nothman.
- [FEATURE] *preprocessing.RobustScaler* and *preprocessing.robust_scale* can be fitted using sparse matrices. #11308 by Guillaume Lemaitre.
- [FEATURE] *preprocessing.OneHotEncoder* now supports the *get_feature_names* method to obtain the transformed feature names. #10181 by Nirvan Anjirbag and Joris Van den Bossche.
- [FEATURE] A parameter `check_inverse` was added to *preprocessing.FunctionTransformer* to ensure that `func` and `inverse_func` are the inverse of each other. #9399 by Guillaume Lemaitre.

- [FEATURE] The `transform` method of *sklearn.preprocessing.MultiLabelBinarizer* now ignores any unknown classes. A warning is raised stating the unknown classes classes found which are ignored. #10913 by Rodrigo Agundez.
- [FIX] Fixed bugs in *preprocessing.LabelEncoder* which would sometimes throw errors when `transform` or `inverse_transform` was called with empty arrays. #10458 by Mayur Kulkarni.
- [FIX] Fix ValueError in *preprocessing.LabelEncoder* when using `inverse_transform` on unseen labels. #9816 by Charlie Newey.
- [FIX] Fix bug in *preprocessing.OneHotEncoder* which discarded the `dtype` when returning a sparse matrix output. #11042 by Daniel Morales.
- [FIX] Fix `fit` and `partial_fit` in *preprocessing.StandardScaler* in the rare case when `with_mean=False` and `with_std=False` which was crashing by calling `fit` more than once and giving inconsistent results for `mean_` whether the input was a sparse or a dense matrix. `mean_` will be set to `None` with both sparse and dense inputs. `n_samples_seen_` will be also reported for both input types. #11235 by Guillaume Lemaitre.
- [API CHANGE] Deprecate `n_values` and `categorical_features` parameters and `active_features_`, `feature_indices_` and `n_values_` attributes of *preprocessing.OneHotEncoder*. The `n_values` parameter can be replaced with the new `categories` parameter, and the attributes with the new `categories_` attribute. Selecting the categorical features with the `categorical_features` parameter is now better supported using the *compose.ColumnTransformer*. #10521 by Joris Van den Bossche.
- [API CHANGE] Deprecate `preprocessing.Imputer` and move the corresponding module to *impute.SimpleImputer*. #9726 by Kumar Ashutosh.
- [API CHANGE] The `axis` parameter that was in `preprocessing.Imputer` is no longer present in *impute.SimpleImputer*. The behavior is equivalent to `axis=0` (impute along columns). Row-wise imputation can be performed with FunctionTransformer (e.g., `FunctionTransformer(lambda X: SimpleImputer().fit_transform(X.T).T)`). #10829 by Guillaume Lemaitre and Gilberto Olimpio.
- [API CHANGE] The NaN marker for the missing values has been changed between the `preprocessing.Imputer` and the *impute.SimpleImputer*. `missing_values='NaN'` should now be `missing_values=np.nan`. #11211 by Jeremie du Boisberranger.
- [API CHANGE] In *preprocessing.FunctionTransformer*, the default of `validate` will be from `True` to `False` in 0.22. #10655 by Guillaume Lemaitre.

sklearn.svm

- [FIX] Fixed a bug in *svm.SVC* where when the argument `kernel` is unicode in Python2, the `predict_proba` method was raising an unexpected TypeError given dense inputs. #10412 by Jiongyan Zhang.
- [API CHANGE] Deprecate `random_state` parameter in *svm.OneClassSVM* as the underlying implementation is not random. #9497 by Albert Thomas.
- [API CHANGE] The default value of `gamma` parameter of *svm.SVC*, *NuSVC*, *SVR*, *NuSVR*, *OneClassSVM* will change from `'auto'` to `'scale'` in version 0.22 to account better for unscaled features. #8361 by Gaurav Dhingra and Ting Neo.

sklearn.tree

- [ENHANCEMENT] Although private (and hence not assured API stability), `tree._criterion.`

`ClassificationCriterion` and `tree._criterion.RegressionCriterion` may now be cimported and extended. #10325 by Camil Staps.

- [FIX] Fixed a bug in `tree.BaseDecisionTree` with `splitter="best"` where split threshold could become infinite when values in X were near infinite. #10536 by Jonathan Ohayon.
- [FIX] Fixed a bug in `tree.MAE` to ensure sample weights are being used during the calculation of tree MAE impurity. Previous behaviour could cause suboptimal splits to be chosen since the impurity calculation considered all samples to be of equal weight importance. #11464 by John Stott.

`sklearn.utils`

- [FEATURE] *`utils.check_array`* and *`utils.check_X_y`* now have `accept_large_sparse` to control whether scipy.sparse matrices with 64-bit indices should be rejected. #11327 by Karan Dhingra and Joel Nothman.
- [EFFICIENCY] [FIX] Avoid copying the data in *`utils.check_array`* when the input data is a memmap (and `copy=False`). #10663 by Arthur Mensch and Loïc Estève.
- [API CHANGE] *`utils.check_array`* yield a `FutureWarning` indicating that arrays of bytes/strings will be interpreted as decimal numbers beginning in version 0.22. #10229 by Ryan Lee

Multiple modules

- [FEATURE] [API CHANGE] More consistent outlier detection API: Add a `score_samples` method in *`svm.OneClassSVM`*, *`ensemble.IsolationForest`*, *`neighbors.LocalOutlierFactor`*, *`covariance.EllipticEnvelope`*. It allows to access raw score functions from original papers. A new `offset_` parameter allows to link `score_samples` and `decision_function` methods. The `contamination` parameter of *`ensemble.IsolationForest`* and *`neighbors.LocalOutlierFactor`* `decision_function` methods is used to define this `offset_` such that outliers (resp. inliers) have negative (resp. positive) `decision_function` values. By default, `contamination` is kept unchanged to 0.1 for a deprecation period. In 0.22, it will be set to "auto", thus using method-specific score offsets. In *`covariance.EllipticEnvelope`* `decision_function` method, the `raw_values` parameter is deprecated as the shifted Mahalanobis distance will be always returned in 0.22. #9015 by Nicolas Goix.
- [FEATURE] [API CHANGE] A `behaviour` parameter has been introduced in *`ensemble.IsolationForest`* to ensure backward compatibility. In the old behaviour, the `decision_function` is independent of the `contamination` parameter. A threshold attribute depending on the `contamination` parameter is thus used. In the new behaviour the `decision_function` is dependent on the `contamination` parameter, in such a way that 0 becomes its natural threshold to detect outliers. Setting behaviour to "old" is deprecated and will not be possible in version 0.22. Beside, the behaviour parameter will be removed in 0.24. #11553 by Nicolas Goix.
- [API CHANGE] Added convergence warning to *`svm.LinearSVC`* and *`linear_model.LogisticRegression`* when `verbose` is set to 0. #10881 by Alexandre Sevin.
- [API CHANGE] Changed warning type from `UserWarning` to *`exceptions.ConvergenceWarning`* for failing convergence in *`linear_model.logistic_regression_path`*, *`linear_model.RANSACRegressor`*, *`linear_model.ridge_regression`*, *`gaussian_process.GaussianProcessRegressor`*, *`gaussian_process.GaussianProcessClassifier`*, *`decomposition.fastica`*, *`cross_decomposition.PLSCanonical`*, *`cluster.AffinityPropagation`*, and *`cluster.Birch`*. #10306 by Jonathan Siebert.

Miscellaneous

- [MAJOR FEATURE] A new configuration parameter, `working_memory` was added to control memory consumption limits in chunked operations, such as the new *`metrics.pairwise_distances_chunked`*. See *Limiting Working Memory*. #10280 by Joel Nothman and Aman Dalmia.
- [FEATURE] The version of `joblib` bundled with Scikit-learn is now 0.12. This uses a new default multiprocessing implementation, named loky. While this may incur some memory and communication overhead, it should provide greater cross-platform stability than relying on Python standard library multiprocessing. #11741 by the Joblib developers, especially Thomas Moreau and Olivier Grisel.
- [FEATURE] An environment variable to use the site joblib instead of the vendored one was added (*Environment variables*). The main API of joblib is now exposed in *`sklearn.utils`*. #11166 by Gael Varoquaux.
- [FEATURE] Add almost complete PyPy 3 support. Known unsupported functionalities are *`datasets.load_svmlight_file`*, *`feature_extraction.FeatureHasher`* and *`feature_extraction.text.HashingVectorizer`*. For running on PyPy, PyPy3-v5.10+, Numpy 1.14.0+, and scipy 1.1.0+ are required. #11010 by Ronan Lamy and Roman Yurchak.
- [FEATURE] A utility method *`sklearn.show_versions`* was added to print out information relevant for debugging. It includes the user system, the Python executable, the version of the main libraries and BLAS binding information. #11596 by Alexandre Boucaud
- [FIX] Fixed a bug when setting parameters on meta-estimator, involving both a wrapped estimator and its parameter. #9999 by Marcus Voss and Joel Nothman.
- [FIX] Fixed a bug where calling *`sklearn.base.clone`* was not thread safe and could result in a "pop from empty list" error. #9569 by Andreas Müller.
- [API CHANGE] The default value of `n_jobs` is changed from `1` to `None` in all related functions and classes. `n_jobs=None` means `unset`. It will generally be interpreted as `n_jobs=1`, unless the current `joblib.Parallel` backend context specifies otherwise (See *Glossary* for additional information). Note that this change happens immediately (i.e., without a deprecation cycle). #11741 by Olivier Grisel.
- [FIX] Fixed a bug in validation helpers where passing a Dask DataFrame results in an error. #12462 by Zachariah Miller

Changes to estimator checks

These changes mostly affect library developers.

- Checks for transformers now apply if the estimator implements *transform*, regardless of whether it inherits from *`sklearn.base.TransformerMixin`*. #10474 by Joel Nothman.
- Classifiers are now checked for consistency between *decision_function* and categorical predictions. #10500 by Narine Kokhlikyan.
- Allow tests in *`utils.estimator_checks.check_estimator`* to test functions that accept pairwise data. #9701 by Kyle Johnson
- Allow *`utils.estimator_checks.check_estimator`* to check that there is no private settings apart from parameters during estimator initialization. #9378 by Herilalaina Rakotoarison
- The set of checks in *`utils.estimator_checks.check_estimator`* now includes a `check_set_params` test which checks that `set_params` is equivalent to passing parameters in `__init__` and warns if it encounters parameter validation. #7738 by Alvin Chiang
- Add invariance tests for clustering metrics. #8102 by Ankita Sinha and Guillaume Lemaitre.

- Add `check_methods_subset_invariance` to *check_estimator*, which checks that estimator methods are invariant if applied to a data subset. #10428 by Jonathan Ohayon
- Add tests in *utils.estimator_checks.check_estimator* to check that an estimator can handle read-only memmap input data. #10663 by Arthur Mensch and Loïc Estève.
- `check_sample_weights_pandas_series` now uses 8 rather than 6 samples to accommodate for the default number of clusters in *cluster.KMeans*. #10933 by Johannes Hansen.
- Estimators are now checked for whether `sample_weight=None` equates to `sample_weight=np.ones(...)`. #11558 by Sergul Aydore.

Code and Documentation Contributors

Thanks to everyone who has contributed to the maintenance and improvement of the project since version 0.19, including:

211217613, Aarshay Jain, absolutelyNoWarranty, Adam Greenhall, Adam Kleczewski, Adam Richie-Halford, adelr, AdityaDaflapurkar, Adrin Jalali, Aidan Fitzgerald, aishgrt1, Akash Shivram, Alan Liddell, Alan Yee, Albert Thomas, Alexander Lenail, Alexander-N, Alexandre Boucaud, Alexandre Gramfort, Alexandre Sevin, Alex Egg, Alvaro Perez-Diaz, Amanda, Aman Dalmia, Andreas Bjerre-Nielsen, Andreas Mueller, Andrew Peng, Angus Williams, Aniruddha Dave, annaayzenshtat, Anthony Gitter, Antonio Quinonez, Anubhav Marwaha, Arik Pamnani, Arthur Ozga, Artiem K, Arunava, Arya McCarthy, Attractadore, Aurélien Bellet, Aurélien Geron, Ayush Gupta, Balakumaran Manoharan, Bangda Sun, Barry Hart, Bastian Venthur, Ben Lawson, Benn Roth, Breno Freitas, Brent Yi, brett koonce, Caio Oliveira, Camil Staps, cclauss, Chady Kamar, Charlie Brummitt, Charlie Newey, chris, Chris, Chris Catalfo, Chris Foster, Chris Holdgraf, Christian Braune, Christian Hirsch, Christian Hogan, Christopher Jenness, Clement Joudet, cnx, cwitte, Dallas Card, Dan Barkhorn, Daniel, Daniel Ferreira, Daniel Gomez, Daniel Klevebring, Danielle Shwed, Daniel Mohns, Danil Baibak, Darius Morawiec, David Beach, David Burns, David Kirkby, David Nicholson, David Pickup, Derek, Didi Bar-Zev, diegodlh, Dillon Gardner, Dillon Niederhut, dilutedsauce, dlovell, Dmitry Mottl, Dmitry Petrov, Dor Cohen, Douglas Duhaime, Ekaterina Tuzova, Eric Chang, Eric Dean Sanchez, Erich Schubert, Eunji, Fang-Chieh Chou, FarahSaeed, felix, Félix Raimundo, fenx, filipj8, FrankHui, Franz Wompner, Freija Descamps, frsi, Gabriele Calvo, Gael Varoquaux, Gaurav Dhingra, Georgi Peev, Gil Forsyth, Giovanni Giuseppe Costa, gkevinyen5418, goncalo-rodrigues, Gryllos Prokopis, Guillaume Lemaitre, Guillaume "Vermeille" Sanchez, Gustavo De Mari Pereira, hakaa1, Hanmin Qin, Henry Lin, Hong, Honghe, Hossein Pourbozorg, Hristo, Hunan Rostomyan, iampat, Ivan PANICO, Jaewon Chung, Jake VanderPlas, jakirkham, James Bourbeau, James Malcolm, Jamie Cox, Jan Koch, Jan Margeta, Jan Schlüter, janvanrijn, Jason Wolosonovich, JC Liu, Jeb Bearer, jeremiedbb, Jimmy Wan, Jinkun Wang, Jiongyan Zhang, jjabl, jkleint, Joan Massich, Joël Billaud, Joel Nothman, Johannes Hansen, JohnStott, Jonatan Samoocha, Jonathan Ohayon, Jörg Döpfert, Joris Van den Bossche, Jose Perez-Parras Toledano, josephsalmon, jotasi, jschendel, Julian Kuhlmann, Julien Chaumond, julietcl, Justin Shenk, Karl F, Kasper Primdal Lauritzen, Katrin Leinweber, Kirill, ksemb, Kuai Yu, Kumar Ashutosh, Kyeongpil Kang, Kye Taylor, kyledrogo, Leland McInnes, Léo DS, Liam Geron, Liutong Zhou, Lizao Li, lkjcalc, Loic Esteve, louib, Luciano Viola, Lucija Gregov, Luis Osa, Luis Pedro Coelho, Luke M Craig, Luke Persola, Mabel, Mabel Villalba, Maniteja Nandana, MarkIwanchyshyn, Mark Roth, Markus Müller, MarsGuy, Martin Gubri, martin-hahn, martin-kokos, mathurinm, Matthias Feurer, Max Copeland, Mayur Kulkarni, Meghann Agarwal, Melanie Goetz, Michael A. Alcorn, Minghui Liu, Ming Li, Minh Le, Mohamed Ali Jamaoui, Mohamed Maskani, Mohammad Shahebaz, Muayyad Alsadi, Nabarun Pal, Nagarjuna Kumar, Naoya Kanai, Narendran Santhanam, NarineK, Nathaniel Saul, Nathan Suh, Nicholas Nadeau, P.Eng., AVS, Nick Hoh, Nicolas Goix, Nicolas Hug, Nicolau Werneck, nielsenmarkus11, Nihar Sheth, Nikita Titov, Nilesh Kevlani, Nirvan Anjirbag, notmatthancock, nzw, Oleksandr Pavlyk, oliblum90, Oliver Rausch, Olivier Grisel, Oren Milman, Osaid Rehman Nasir, pasbi, Patrick Fernandes, Patrick Olden, Paul Paczuski, Pedro Morales, Peter, Peter St. John, pierreablin, pietruh, Pinaki Nath Chowdhury, Piotr Szymański, Pradeep Reddy Raamana, Pravar D Mahajan, pravarmahajan, QingYing Chen, Raghav RV, Rajendra arora, RAKOTOARISON Herilalaina, Rameshwar Bhaskaran, RankyLau, Rasul Kerimov, Reiichiro Nakano, Rob, Roman Kosobrodov, Roman Yurchak, Ronan Lamy, rragundez, Rüdiger Busche, Ryan, Sachin Kelkar, Sagnik Bhattacharya, Sailesh Choyal, Sam Radhakrishnan, Sam Steingold, Samuel Bell, Samuel O. Ronsin, Saqib Nizam Shamsi, SATISH J, Saurabh Gupta, Scott Gigante, Sebastian Flennerhag, Sebastian Raschka, Sebastien Dubois, Sébastien Lerique, Sebastin Santy, Sergey Feldman, Sergey Melderis,

Sergul Aydore, Shahebaz, Shalil Awaley, Shangwu Yao, Sharad Vijalapuram, Sharan Yalburgi, shenhanc78, Shivam Rastogi, Shu Haoran, siftikha, Sinclert Pérez, SolutusImmensus, Somya Anand, srajan paliwal, Sriharsha Hatwar, Sri Krishna, Stefan van der Walt, Stephen McDowell, Steven Brown, syonekura, Taehoon Lee, Takanori Hayashi, tarcusx, Taylor G Smith, theriley106, Thomas, Thomas Fan, Thomas Heavey, Tobias Madsen, tobycheese, Tom Augspurger, Tom Dupré la Tour, Tommy, Trevor Stephens, Trishnendu Ghorai, Tulio Casagrande, twosigmajab, Umar Farouk Umar, Urvang Patel, Utkarsh Upadhyay, Vadim Markovtsev, Varun Agrawal, Vathsala Achar, Vilhelm von Ehrenheim, Vinayak Mehta, Vinit, Vinod Kumar L, Viraj Mavani, Viraj Navkal, Vivek Kumar, Vlad Niculae, vqean3, Vrishank Bhardwaj, vufg, wallygauze, Warut Vijitbenjaronk, wdevazelhes, Wenhao Zhang, Wes Barnett, Will, William de Vazelhes, Will Rosenfeld, Xin Xiong, Yiming (Paul) Li, ymazari, Yufeng, Zach Griffith, Zé Vinícius, Zhenqing Hu, Zhiqing Xiao, Zijie (ZJ) Poh

1.7.13 Version 0.19.2

July, 2018

This release is exclusively in order to support Python 3.7.

Related changes

- `n_iter_` may vary from previous releases in *`linear_model.LogisticRegression`* with `solver='lbfgs'` and *`linear_model.HuberRegressor`*. For Scipy <= 1.0.0, the optimizer could perform more than the requested maximum number of iterations. Now both estimators will report at most `max_iter` iterations even if more were performed. #10723 by Joel Nothman.

1.7.14 Version 0.19.1

October 23, 2017

This is a bug-fix release with some minor documentation improvements and enhancements to features released in 0.19.0.

Note there may be minor differences in TSNE output in this release (due to #9623), in the case where multiple samples have equal distance to some sample.

Changelog

API changes

- Reverted the addition of `metrics.ndcg_score` and `metrics.dcg_score` which had been merged into version 0.19.0 by error. The implementations were broken and undocumented.
- `return_train_score` which was added to *`model_selection.GridSearchCV`*, *`model_selection.RandomizedSearchCV`* and *`model_selection.cross_validate`* in version 0.19.0 will be changing its default value from True to False in version 0.21. We found that calculating training score could have a great effect on cross validation runtime in some cases. Users should explicitly set `return_train_score` to False if prediction or scoring functions are slow, resulting in a deleterious effect on CV runtime, or to True if they wish to use the calculated scores. #9677 by Kumar Ashutosh and Joel Nothman.
- `correlation_models` and `regression_models` from the legacy gaussian processes implementation have been belatedly deprecated. #9717 by Kumar Ashutosh.

Bug fixes

- Avoid integer overflows in *metrics.matthews_corrcoef*. #9693 by Sam Steingold.
- Fixed a bug in the objective function for *manifold.TSNE* (both exact and with the Barnes-Hut approximation) when `n_components >= 3`. #9711 by @goncalo-rodrigues.
- Fix regression in *model_selection.cross_val_predict* where it raised an error with `method='predict_proba'` for some probabilistic classifiers. #9641 by James Bourbeau.
- Fixed a bug where *datasets.make_classification* modified its input `weights`. #9865 by Sachin Kelkar.
- *model_selection.StratifiedShuffleSplit* now works with multioutput multiclass or multilabel data with more than 1000 columns. #9922 by Charlie Brummitt.
- Fixed a bug with nested and conditional parameter setting, e.g. setting a pipeline step and its parameter at the same time. #9945 by Andreas Müller and Joel Nothman.

Regressions in 0.19.0 fixed in 0.19.1:

- Fixed a bug where parallelised prediction in random forests was not thread-safe and could (rarely) result in arbitrary errors. #9830 by Joel Nothman.
- Fix regression in *model_selection.cross_val_predict* where it no longer accepted `X` as a list. #9600 by Rasul Kerimov.
- Fixed handling of `cross_val_predict` for binary classification with `method='decision_function'`. #9593 by Reiichiro Nakano and core devs.
- Fix regression in *pipeline.Pipeline* where it no longer accepted `steps` as a tuple. #9604 by Joris Van den Bossche.
- Fix bug where `n_iter` was not properly deprecated, leaving `n_iter` unavailable for interim use in *linear_model.SGDClassifier*, *linear_model.SGDRegressor*, *linear_model.PassiveAggressiveClassifier*, *linear_model.PassiveAggressiveRegressor* and *linear_model.Perceptron*. #9558 by Andreas Müller.
- Dataset fetchers make sure temporary files are closed before removing them, which caused errors on Windows. #9847 by Joan Massich.
- Fixed a regression in *manifold.TSNE* where it no longer supported metrics other than 'euclidean' and 'precomputed'. #9623 by Oli Blum.

Enhancements

- Our test suite and `utils.estimator_checks.check_estimators` can now be run without Nose installed. #9697 by Joan Massich.
- To improve usability of version 0.19's *pipeline.Pipeline* caching, `memory` now allows `joblib.Memory` instances. This make use of the new *utils.validation.check_memory* helper. issue:`9584` by Kumar Ashutosh
- Some fixes to examples: #9750, #9788, #9815
- Made a FutureWarning in SGD-based estimators less verbose. #9802 by Vrishank Bhardwaj.

Code and Documentation Contributors

With thanks to:

Joel Nothman, Loic Esteve, Andreas Mueller, Kumar Ashutosh, Vrishank Bhardwaj, Hanmin Qin, Rasul Kerimov, James Bourbeau, Nagarjuna Kumar, Nathaniel Saul, Olivier Grisel, Roman Yurchak, Reiichiro Nakano, Sachin Kelkar, Sam Steingold, Yaroslav Halchenko, diegodlh, felix, goncalo-rodrigues, jkleint, oliblum90, pasbi, Anthony Gitter, Ben Lawson, Charlie Brummitt, Didi Bar-Zev, Gael Varoquaux, Joan Massich, Joris Van den Bossche, nielsenmarkus11

1.7.15 Version 0.19

August 12, 2017

Highlights

We are excited to release a number of great new features including *neighbors.LocalOutlierFactor* for anomaly detection, *preprocessing.QuantileTransformer* for robust feature transformation, and the *multioutput.ClassifierChain* meta-estimator to simply account for dependencies between classes in multilabel problems. We have some new algorithms in existing estimators, such as multiplicative update in *decomposition.NMF* and multinomial *linear_model.LogisticRegression* with L1 loss (use `solver='saga'`).

Cross validation is now able to return the results from multiple metric evaluations. The new *model_selection.cross_validate* can return many scores on the test data as well as training set performance and timings, and we have extended the `scoring` and `refit` parameters for grid/randomized search *to handle multiple metrics*.

You can also learn faster. For instance, the *new option to cache transformations* in *pipeline.Pipeline* makes grid search over pipelines including slow transformations much more efficient. And you can predict faster: if you're sure you know what you're doing, you can turn off validating that the input is finite using *config_context*.

We've made some important fixes too. We've fixed a longstanding implementation error in *metrics.average_precision_score*, so please be cautious with prior results reported from that function. A number of errors in the *manifold.TSNE* implementation have been fixed, particularly in the default Barnes-Hut approximation. *semi_supervised.LabelSpreading* and *semi_supervised.LabelPropagation* have had substantial fixes. LabelPropagation was previously broken. LabelSpreading should now correctly respect its alpha parameter.

Changed models

The following estimators and functions, when fit with the same data and parameters, may produce different models from the previous version. This often occurs due to changes in the modelling logic (bug fixes or enhancements), or in random sampling procedures.

- *cluster.KMeans* with sparse X and initial centroids given (bug fix)
- *cross_decomposition.PLSRegression* with `scale=True` (bug fix)
- *ensemble.GradientBoostingClassifier* and *ensemble.GradientBoostingRegressor* where `min_impurity_split` is used (bug fix)
- gradient boosting `loss='quantile'` (bug fix)
- *ensemble.IsolationForest* (bug fix)
- *feature_selection.SelectFdr* (bug fix)
- *linear_model.RANSACRegressor* (bug fix)

- *linear_model.LassoLars* (bug fix)
- *linear_model.LassoLarsIC* (bug fix)
- *manifold.TSNE* (bug fix)
- *neighbors.NearestCentroid* (bug fix)
- *semi_supervised.LabelSpreading* (bug fix)
- *semi_supervised.LabelPropagation* (bug fix)
- tree based models where `min_weight_fraction_leaf` is used (enhancement)
- *model_selection.StratifiedKFold* with `shuffle=True` (this change, due to #7823 was not mentioned in the release notes at the time)

Details are listed in the changelog below.

(While we are trying to better inform users by providing this information, we cannot assure that this list is complete.)

Changelog

New features

Classifiers and regressors

- Added *multioutput.ClassifierChain* for multi-label classification. By Adam Kleczewski.
- Added solver `'saga'` that implements the improved version of Stochastic Average Gradient, in *linear_model.LogisticRegression* and *linear_model.Ridge*. It allows the use of L1 penalty with multinomial logistic loss, and behaves marginally better than 'sag' during the first epochs of ridge and logistic regression. #8446 by Arthur Mensch.

Other estimators

- Added the *neighbors.LocalOutlierFactor* class for anomaly detection based on nearest neighbors. #5279 by Nicolas Goix and Alexandre Gramfort.
- Added *preprocessing.QuantileTransformer* class and *preprocessing.quantile_transform* function for features normalization based on quantiles. #8363 by Denis Engemann, Guillaume Lemaitre, Olivier Grisel, Raghav RV, Thierry Guillemot, and Gael Varoquaux.
- The new solver `'mu'` implements a Multiplicate Update in *decomposition.NMF*, allowing the optimization of all beta-divergences, including the Frobenius norm, the generalized Kullback-Leibler divergence and the Itakura-Saito divergence. #5295 by Tom Dupre la Tour.

Model selection and evaluation

- *model_selection.GridSearchCV* and *model_selection.RandomizedSearchCV* now support simultaneous evaluation of multiple metrics. Refer to the *Specifying multiple metrics for evaluation* section of the user guide for more information. #7388 by Raghav RV
- Added the *model_selection.cross_validate* which allows evaluation of multiple metrics. This function returns a dict with more useful information from cross-validation such as the train scores, fit times and score times. Refer to *The cross_validate function and multiple metric evaluation* section of the userguide for more information. #7388 by Raghav RV
- Added *metrics.mean_squared_log_error*, which computes the mean square error of the logarithmic transformation of targets, particularly useful for targets with an exponential trend. #7655 by Karan Desai.
- Added *metrics.dcg_score* and *metrics.ndcg_score*, which compute Discounted cumulative gain (DCG) and Normalized discounted cumulative gain (NDCG). #7739 by David Gasquez.

- Added the *model_selection.RepeatedKFold* and *model_selection.RepeatedStratifiedKFold*. #8120 by Neeraj Gangwar.

Miscellaneous

- Validation that input data contains no NaN or inf can now be suppressed using *config_context*, at your own risk. This will save on runtime, and may be particularly useful for prediction time. #7548 by Joel Nothman.
- Added a test to ensure parameter listing in docstrings match the function/class signature. #9206 by Alexandre Gramfort and Raghav RV.

Enhancements

Trees and ensembles

- The `min_weight_fraction_leaf` constraint in tree construction is now more efficient, taking a fast path to declare a node a leaf if its weight is less than 2 * the minimum. Note that the constructed tree will be different from previous versions where `min_weight_fraction_leaf` is used. #7441 by Nelson Liu.
- *ensemble.GradientBoostingClassifier* and *ensemble.GradientBoostingRegressor* now support sparse input for prediction. #6101 by Ibraim Ganiev.
- *ensemble.VotingClassifier* now allows changing estimators by using *ensemble.VotingClassifier.set_params*. An estimator can also be removed by setting it to `None`. #7674 by Yichuan Liu.
- *tree.export_graphviz* now shows configurable number of decimal places. #8698 by Guillaume Lemaitre.
- Added `flatten_transform` parameter to *ensemble.VotingClassifier* to change output shape of `transform` method to 2 dimensional. #7794 by Ibraim Ganiev and Herilalaina Rakotoarison.

Linear, kernelized and related models

- *linear_model.SGDClassifier*, *linear_model.SGDRegressor*, *linear_model.PassiveAggressiveClassifier*, *linear_model.PassiveAggressiveRegressor* and *linear_model.Perceptron* now expose `max_iter` and `tol` parameters, to handle convergence more precisely. `n_iter` parameter is deprecated, and the fitted estimator exposes a `n_iter_` attribute, with actual number of iterations before convergence. #5036 by Tom Dupre la Tour.
- Added `average` parameter to perform weight averaging in *linear_model.PassiveAggressiveClassifier*. #4939 by Andrea Esuli.
- *linear_model.RANSACRegressor* no longer throws an error when calling `fit` if no inliers are found in its first iteration. Furthermore, causes of skipped iterations are tracked in newly added attributes, `n_skips_*`. #7914 by Michael Horrell.
- In *gaussian_process.GaussianProcessRegressor*, method `predict` is a lot faster with `return_std=True`. #8591 by Hadrien Bertrand.
- Added `return_std` to `predict` method of *linear_model.ARDRegression* and *linear_model.BayesianRidge*. #7838 by Sergey Feldman.
- Memory usage enhancements: Prevent cast from float32 to float64 in: *linear_model.MultiTaskElasticNet*; *linear_model.LogisticRegression* when using newton-cg solver; and *linear_model.Ridge* when using svd, sparse_cg, cholesky or lsqr solvers. #8835, #8061 by Joan Massich and Nicolas Cordier and Thierry Guillemot.

Other predictors

- Custom metrics for the `neighbors` binary trees now have fewer constraints: they must take two 1d-arrays and return a float. #6288 by Jake Vanderplas.
- `algorithm='auto` in `neighbors` estimators now chooses the most appropriate algorithm for all input types and metrics. #9145 by Herilalaina Rakotoarison and Reddy Chinthala.

Decomposition, manifold learning and clustering

- *`cluster.MiniBatchKMeans`* and *`cluster.KMeans`* now use significantly less memory when assigning data points to their nearest cluster center. #7721 by Jon Crall.
- *`decomposition.PCA`*, *`decomposition.IncrementalPCA`* and *`decomposition.TruncatedSVD`* now expose the singular values from the underlying SVD. They are stored in the attribute `singular_values_`, like in *`decomposition.IncrementalPCA`*. #7685 by Tommy Löfstedt
- *`decomposition.NMF`* now faster when `beta_loss=0`. #9277 by @hongkahjun.
- Memory improvements for method `barnes_hut` in *`manifold.TSNE`* #7089 by Thomas Moreau and Olivier Grisel.
- Optimization schedule improvements for Barnes-Hut *`manifold.TSNE`* so the results are closer to the one from the reference implementation lvdmaaten/bhtsne by Thomas Moreau and Olivier Grisel.
- Memory usage enhancements: Prevent cast from float32 to float64 in *`decomposition.PCA`* and `decomposition.randomized_svd_low_rank`. #9067 by Raghav RV.

Preprocessing and feature selection

- Added `norm_order` parameter to *`feature_selection.SelectFromModel`* to enable selection of the norm order when `coef_` is more than 1D. #6181 by Antoine Wendlinger.
- Added ability to use sparse matrices in *`feature_selection.f_regression`* with `center=True`. #8065 by Daniel LeJeune.
- Small performance improvement to n-gram creation in `feature_extraction.text` by binding methods for loops and special-casing unigrams. #7567 by Jaye Doepke
- Relax assumption on the data for the *`kernel_approximation.SkewedChi2Sampler`*. Since the Skewed-Chi2 kernel is defined on the open interval $(-skewedness; +\infty)^d$, the transform function should not check whether `X < 0` but whether `X < -self.skewedness`. #7573 by Romain Brault.
- Made default kernel parameters kernel-dependent in *`kernel_approximation.Nystroem`*. #5229 by Saurabh Bansod and Andreas Müller.

Model evaluation and meta-estimators

- *`pipeline.Pipeline`* is now able to cache transformers within a pipeline by using the `memory` constructor parameter. #7990 by Guillaume Lemaitre.
- *`pipeline.Pipeline`* steps can now be accessed as attributes of its `named_steps` attribute. #8586 by Herilalaina Rakotoarison.
- Added `sample_weight` parameter to *`pipeline.Pipeline.score`*. #7723 by Mikhail Korobov.
- Added ability to set `n_jobs` parameter to *`pipeline.make_union`*. A `TypeError` will be raised for any other kwargs. #8028 by Alexander Booth.
- *`model_selection.GridSearchCV`*, *`model_selection.RandomizedSearchCV`* and *`model_selection.cross_val_score`* now allow estimators with callable kernels which were previously prohibited. #8005 by Andreas Müller .
- *`model_selection.cross_val_predict`* now returns output of the correct shape for all values of the argument `method`. #7863 by Aman Dalmia.

- Added `shuffle` and `random_state` parameters to shuffle training data before taking prefixes of it based on training sizes in *model_selection.learning_curve*. #7506 by Narine Kokhlikyan.
- *model_selection.StratifiedShuffleSplit* now works with multioutput multiclass (or multilabel) data. #9044 by Vlad Niculae.
- Speed improvements to *model_selection.StratifiedShuffleSplit*. #5991 by Arthur Mensch and Joel Nothman.
- Add `shuffle` parameter to *model_selection.train_test_split*. #8845 by themrmax
- *multioutput.MultiOutputRegressor* and *multioutput.MultiOutputClassifier* now support online learning using `partial_fit`. :issue: `8053` by Peng Yu.
- Add `max_train_size` parameter to *model_selection.TimeSeriesSplit* #8282 by Aman Dalmia.
- More clustering metrics are now available through *metrics.get_scorer* and `scoring` parameters. #8117 by Raghav RV.
- A scorer based on *metrics.explained_variance_score* is also available. #9259 by Hanmin Qin.

Metrics

- *metrics.matthews_corrcoef* now support multiclass classification. #8094 by Jon Crall.
- Add `sample_weight` parameter to *metrics.cohen_kappa_score*. #8335 by Victor Poughon.

Miscellaneous

- `utils.check_estimator` now attempts to ensure that methods transform, predict, etc. do not set attributes on the estimator. #7533 by Ekaterina Krivich.
- Added type checking to the `accept_sparse` parameter in `utils.validation` methods. This parameter now accepts only boolean, string, or list/tuple of strings. `accept_sparse=None` is deprecated and should be replaced by `accept_sparse=False`. #7880 by Josh Karnofsky.
- Make it possible to load a chunk of an svmlight formatted file by passing a range of bytes to *datasets.load_svmlight_file*. #935 by Olivier Grisel.
- *dummy.DummyClassifier* and *dummy.DummyRegressor* now accept non-finite features. #8931 by @Attractadore.

Bug fixes

Trees and ensembles

- Fixed a memory leak in trees when using trees with `criterion='mae'`. #8002 by Raghav RV.
- Fixed a bug where *ensemble.IsolationForest* uses an an incorrect formula for the average path length #8549 by Peter Wang.
- Fixed a bug where *ensemble.AdaBoostClassifier* throws `ZeroDivisionError` while fitting data with single class labels. #7501 by Dominik Krzeminski.
- Fixed a bug in *ensemble.GradientBoostingClassifier* and *ensemble.GradientBoostingRegressor* where a float being compared to `0.0` using `==` caused a divide by zero error. #7970 by He Chen.
- Fix a bug where *ensemble.GradientBoostingClassifier* and *ensemble.GradientBoostingRegressor* ignored the `min_impurity_split` parameter. #8006 by Sebastian Pölsterl.
- Fixed `oob_score` in *ensemble.BaggingClassifier*. #8936 by Michael Lewis

- Fixed excessive memory usage in prediction for random forests estimators. #8672 by Mike Benfield.
- Fixed a bug where `sample_weight` as a list broke random forests in Python 2 #8068 by @xor.
- Fixed a bug where *ensemble.IsolationForest* fails when `max_features` is less than 1. #5732 by Ishank Gulati.
- Fix a bug where gradient boosting with `loss='quantile'` computed negative errors for negative values of `ytrue - ypred` leading to wrong values when calling `__call__`. #8087 by Alexis Mignon
- Fix a bug where *ensemble.VotingClassifier* raises an error when a numpy array is passed in for weights. #7983 by Vincent Pham.
- Fixed a bug where *tree.export_graphviz* raised an error when the length of features_names does not match n_features in the decision tree. #8512 by Li Li.

Linear, kernelized and related models

- Fixed a bug where *linear_model.RANSACRegressor.fit* may run until `max_iter` if it finds a large inlier group early. #8251 by @aivision2020.
- Fixed a bug where *naive_bayes.MultinomialNB* and *naive_bayes.BernoulliNB* failed when `alpha=0`. #5814 by Yichuan Liu and Herilalaina Rakotoarison.
- Fixed a bug where *linear_model.LassoLars* does not give the same result as the LassoLars implementation available in R (lars library). #7849 by Jair Montoya Martinez.
- Fixed a bug in `linear_model.RandomizedLasso`, *linear_model.Lars*, *linear_model.LassoLars*, *linear_model.LarsCV* and *linear_model.LassoLarsCV*, where the parameter `precompute` was not used consistently across classes, and some values proposed in the docstring could raise errors. #5359 by Tom Dupre la Tour.
- Fix inconsistent results between *linear_model.RidgeCV* and *linear_model.Ridge* when using `normalize=True`. #9302 by Alexandre Gramfort.
- Fix a bug where *linear_model.LassoLars.fit* sometimes left `coef_` as a list, rather than an ndarray. #8160 by CJ Carey.
- Fix *linear_model.BayesianRidge.fit* to return ridge parameter `alpha_` and `lambda_` consistent with calculated coefficients `coef_` and `intercept_`. #8224 by Peter Gedeck.
- Fixed a bug in *svm.OneClassSVM* where it returned floats instead of integer classes. #8676 by Vathsala Achar.
- Fix AIC/BIC criterion computation in *linear_model.LassoLarsIC*. #9022 by Alexandre Gramfort and Mehmet Basbug.
- Fixed a memory leak in our LibLinear implementation. #9024 by Sergei Lebedev
- Fix bug where stratified CV splitters did not work with *linear_model.LassoCV*. #8973 by Paulo Haddad.
- Fixed a bug in *gaussian_process.GaussianProcessRegressor* when the standard deviation and covariance predicted without fit would fail with a unmeaningful error by default. #6573 by Quazi Marufur Rahman and Manoj Kumar.

Other predictors

- Fix `semi_supervised.BaseLabelPropagation` to correctly implement `LabelPropagation` and `LabelSpreading` as done in the referenced papers. #9239 by Andre Ambrosio Boechat, Utkarsh Upadhyay, and Joel Nothman.

Decomposition, manifold learning and clustering

- Fixed the implementation of *manifold.TSNE*:

- `early_exageration` parameter had no effect and is now used for the first 250 optimization iterations.
- Fixed the `AssertionError: Tree consistency failed` exception reported in #8992.
- Improve the learning schedule to match the one from the reference implementation lvdmaaten/bhtsne. by Thomas Moreau and Olivier Grisel.
- Fix a bug in *decomposition.LatentDirichletAllocation* where the `perplexity` method was returning incorrect results because the `transform` method returns normalized document topic distributions as of version 0.18. #7954 by Gary Foreman.
- Fix output shape and bugs with n_jobs > 1 in *decomposition.SparseCoder* transform and *decomposition.sparse_encode* for one-dimensional data and one component. This also impacts the output shape of *decomposition.DictionaryLearning*. #8086 by Andreas Müller.
- Fixed the implementation of `explained_variance_` in *decomposition.PCA*, `decomposition.RandomizedPCA` and *decomposition.IncrementalPCA*. #9105 by Hanmin Qin.
- Fixed the implementation of `noise_variance_` in *decomposition.PCA*. #9108 by Hanmin Qin.
- Fixed a bug where *cluster.DBSCAN* gives incorrect result when input is a precomputed sparse matrix with initial rows all zero. #8306 by Akshay Gupta
- Fix a bug regarding fitting *cluster.KMeans* with a sparse array X and initial centroids, where X's means were unnecessarily being subtracted from the centroids. #7872 by Josh Karnofsky.
- Fixes to the input validation in *covariance.EllipticEnvelope*. #8086 by Andreas Müller.
- Fixed a bug in *covariance.MinCovDet* where inputting data that produced a singular covariance matrix would cause the helper method `_c_step` to throw an exception. #3367 by Jeremy Steward
- Fixed a bug in *manifold.TSNE* affecting convergence of the gradient descent. #8768 by David DeTomaso.
- Fixed a bug in *manifold.TSNE* where it stored the incorrect `kl_divergence_`. #6507 by Sebastian Saeger.
- Fixed improper scaling in *cross_decomposition.PLSRegression* with `scale=True`. #7819 by jayzed82.
- `cluster.bicluster.SpectralCoclustering` and `cluster.bicluster.SpectralBiclustering` `fit` method conforms with API by accepting `y` and returning the object. #6126, #7814 by Laurent Direr and Maniteja Nandana.
- Fix bug where `mixture` `sample` methods did not return as many samples as requested. #7702 by Levi John Wolf.
- Fixed the shrinkage implementation in *neighbors.NearestCentroid*. #9219 by Hanmin Qin.

Preprocessing and feature selection

- For sparse matrices, *preprocessing.normalize* with `return_norm=True` will now raise a `NotImplementedError` with 'l1' or 'l2' norm and with norm 'max' the norms returned will be the same as for dense matrices. #7771 by Ang Lu.
- Fix a bug where *feature_selection.SelectFdr* did not exactly implement Benjamini-Hochberg procedure. It formerly may have selected fewer features than it should. #7490 by Peng Meng.
- Fixed a bug where `linear_model.RandomizedLasso` and `linear_model.RandomizedLogisticRegression` breaks for sparse input. #8259 by Aman Dalmia.
- Fix a bug where *feature_extraction.FeatureHasher* mandatorily applied a sparse random projection to the hashed features, preventing the use of *feature_extraction.text.HashingVectorizer* in a pipeline with *feature_extraction.text.TfidfTransformer*. #7565 by Roman Yurchak.

- Fix a bug where *feature_selection.mutual_info_regression* did not correctly use n_neighbors. #8181 by Guillaume Lemaitre.

Model evaluation and meta-estimators

- Fixed a bug where model_selection.BaseSearchCV.inverse_transform returns self.best_estimator_.transform() instead of self.best_estimator_.inverse_transform(). #8344 by Akshay Gupta and Rasmus Eriksson.
- Added classes_ attribute to *model_selection.GridSearchCV*, *model_selection.RandomizedSearchCV*, grid_search.GridSearchCV, and grid_search.RandomizedSearchCV that matches the classes_ attribute of best_estimator_. #7661 and #8295 by Alyssa Batula, Dylan Werner-Meier, and Stephen Hoover.
- Fixed a bug where *model_selection.validation_curve* reused the same estimator for each parameter value. #7365 by Aleksandr Sandrovskii.
- *model_selection.permutation_test_score* now works with Pandas types. #5697 by Stijn Tonk.
- Several fixes to input validation in *multiclass.OutputCodeClassifier* #8086 by Andreas Müller.
- *multiclass.OneVsOneClassifier*'s partial_fit now ensures all classes are provided up-front. #6250 by Asish Panda.
- Fix *multioutput.MultiOutputClassifier.predict_proba* to return a list of 2d arrays, rather than a 3d array. In the case where different target columns had different numbers of classes, a ValueError would be raised on trying to stack matrices with different dimensions. #8093 by Peter Bull.
- Cross validation now works with Pandas datatypes that that have a read-only index. #9507 by Loic Esteve.

Metrics

- *metrics.average_precision_score* no longer linearly interpolates between operating points, and instead weighs precisions by the change in recall since the last operating point, as per the Wikipedia entry. (#7356). By Nick Dingwall and Gael Varoquaux.
- Fix a bug in metrics.classification._check_targets which would return 'binary' if y_true and y_pred were both 'binary' but the union of y_true and y_pred was 'multiclass'. #8377 by Loic Esteve.
- Fixed an integer overflow bug in *metrics.confusion_matrix* and hence *metrics.cohen_kappa_score*. #8354, #7929 by Joel Nothman and Jon Crall.
- Fixed passing of gamma parameter to the chi2 kernel in *metrics.pairwise.pairwise_kernels* #5211 by Nick Rhinehart, Saurabh Bansod and Andreas Müller.

Miscellaneous

- Fixed a bug when *datasets.make_classification* fails when generating more than 30 features. #8159 by Herilalaina Rakotoarison.
- Fixed a bug where *datasets.make_moons* gives an incorrect result when n_samples is odd. #8198 by Josh Levy.
- Some fetch_ functions in datasets were ignoring the download_if_missing keyword. #7944 by Ralf Gommers.
- Fix estimators to accept a sample_weight parameter of type pandas.Series in their fit function. #7825 by Kathleen Chen.
- Fix a bug in cases where numpy.cumsum may be numerically unstable, raising an exception if instability is identified. #7376 and #7331 by Joel Nothman and @yangarbiter.

- Fix a bug where `base.BaseEstimator.__getstate__` obstructed pickling customizations of child-classes, when used in a multiple inheritance context. #8316 by Holger Peters.
- Update Sphinx-Gallery from 0.1.4 to 0.1.7 for resolving links in documentation build with Sphinx>1.5 #8010, #7986 by Oscar Najera
- Add `data_home` parameter to *`sklearn.datasets.fetch_kddcup99`*. #9289 by Loic Esteve.
- Fix dataset loaders using Python 3 version of makedirs to also work in Python 2. #9284 by Sebastin Santy.
- Several minor issues were fixed with thanks to the alerts of lgtm.com. #9278 by Jean Helie, among others.

API changes summary

Trees and ensembles

- Gradient boosting base models are no longer estimators. By Andreas Müller.
- All tree based estimators now accept a `min_impurity_decrease` parameter in lieu of the `min_impurity_split`, which is now deprecated. The `min_impurity_decrease` helps stop splitting the nodes in which the weighted impurity decrease from splitting is no longer at least `min_impurity_decrease`. #8449 by Raghav RV.

Linear, kernelized and related models

- `n_iter` parameter is deprecated in *`linear_model.SGDClassifier`*, *`linear_model.SGDRegressor`*, *`linear_model.PassiveAggressiveClassifier`*, *`linear_model.PassiveAggressiveRegressor`* and *`linear_model.Perceptron`*. By Tom Dupre la Tour.

Other predictors

- `neighbors.LSHForest` has been deprecated and will be removed in 0.21 due to poor performance. #9078 by Laurent Direr.
- *`neighbors.NearestCentroid`* no longer purports to support `metric='precomputed'` which now raises an error. #8515 by Sergul Aydore.
- The `alpha` parameter of *`semi_supervised.LabelPropagation`* now has no effect and is deprecated to be removed in 0.21. #9239 by Andre Ambrosio Boechat, Utkarsh Upadhyay, and Joel Nothman.

Decomposition, manifold learning and clustering

- Deprecate the `doc_topic_distr` argument of the `perplexity` method in *`decomposition.LatentDirichletAllocation`* because the user no longer has access to the unnormalized document topic distribution needed for the perplexity calculation. #7954 by Gary Foreman.
- The `n_topics` parameter of *`decomposition.LatentDirichletAllocation`* has been renamed to `n_components` and will be removed in version 0.21. #8922 by @Attractadore.
- *`decomposition.SparsePCA.transform`*'s `ridge_alpha` parameter is deprecated in preference for class parameter. #8137 by Naoya Kanai.
- *`cluster.DBSCAN`* now has a `metric_params` parameter. #8139 by Naoya Kanai.

Preprocessing and feature selection

- *`feature_selection.SelectFromModel`* now has a `partial_fit` method only if the underlying estimator does. By Andreas Müller.
- *`feature_selection.SelectFromModel`* now validates the `threshold` parameter and sets the `threshold_` attribute during the call to `fit`, and no longer during the call to `transform`. By Andreas Müller.

- The non_negative parameter in *feature_extraction.FeatureHasher* has been deprecated, and replaced with a more principled alternative, alternate_sign. #7565 by Roman Yurchak.
- linear_model.RandomizedLogisticRegression, and linear_model.RandomizedLasso have been deprecated and will be removed in version 0.21. #8995 by Ramana.S.

Model evaluation and meta-estimators

- Deprecate the fit_params constructor input to the *model_selection.GridSearchCV* and *model_selection.RandomizedSearchCV* in favor of passing keyword parameters to the fit methods of those classes. Data-dependent parameters needed for model training should be passed as keyword arguments to fit, and conforming to this convention will allow the hyperparameter selection classes to be used with tools such as *model_selection.cross_val_predict*. #2879 by Stephen Hoover.
- In version 0.21, the default behavior of splitters that use the test_size and train_size parameter will change, such that specifying train_size alone will cause test_size to be the remainder. #7459 by Nelson Liu.
- *multiclass.OneVsRestClassifier* now has partial_fit, decision_function and predict_proba methods only when the underlying estimator does. #7812 by Andreas Müller and Mikhail Korobov.
- *multiclass.OneVsRestClassifier* now has a partial_fit method only if the underlying estimator does. By Andreas Müller.
- The decision_function output shape for binary classification in *multiclass.OneVsRestClassifier* and *multiclass.OneVsOneClassifier* is now (n_samples,) to conform to scikit-learn conventions. #9100 by Andreas Müller.
- The *multioutput.MultiOutputClassifier.predict_proba* function used to return a 3d array (n_samples, n_classes, n_outputs). In the case where different target columns had different numbers of classes, a ValueError would be raised on trying to stack matrices with different dimensions. This function now returns a list of arrays where the length of the list is n_outputs, and each array is (n_samples, n_classes) for that particular output. #8093 by Peter Bull.
- Replace attribute named_steps dict to utils.Bunch in *pipeline.Pipeline* to enable tab completion in interactive environment. In the case conflict value on named_steps and dict attribute, dict behavior will be prioritized. #8481 by Herilalaina Rakotoarison.

Miscellaneous

- Deprecate the y parameter in transform and inverse_transform. The method should not accept y parameter, as it's used at the prediction time. #8174 by Tahar Zanouda, Alexandre Gramfort and Raghav RV.
- SciPy >= 0.13.3 and NumPy >= 1.8.2 are now the minimum supported versions for scikit-learn. The following backported functions in utils have been removed or deprecated accordingly. #8854 and #8874 by Naoya Kanai
- The store_covariances and covariances_ parameters of *discriminant_analysis.QuadraticDiscriminantAnalysis* has been renamed to store_covariance and covariance_ to be consistent with the corresponding parameter names of the *discriminant_analysis.LinearDiscriminantAnalysis*. They will be removed in version 0.21. #7998 by Jiacheng

  Removed in 0.19:

  - utils.fixes.argpartition
  - utils.fixes.array_equal
  - utils.fixes.astype
  - utils.fixes.bincount

  - `utils.fixes.expit`
  - `utils.fixes.frombuffer_empty`
  - `utils.fixes.in1d`
  - `utils.fixes.norm`
  - `utils.fixes.rankdata`
  - `utils.fixes.safe_copy`

  Deprecated in 0.19, to be removed in 0.21:

  - `utils.arpack.eigs`
  - `utils.arpack.eigsh`
  - `utils.arpack.svds`
  - `utils.extmath.fast_dot`
  - `utils.extmath.logsumexp`
  - `utils.extmath.norm`
  - `utils.extmath.pinvh`
  - `utils.graph.graph_laplacian`
  - `utils.random.choice`
  - `utils.sparsetools.connected_components`
  - `utils.stats.rankdata`

- Estimators with both methods `decision_function` and `predict_proba` are now required to have a monotonic relation between them. The method `check_decision_proba_consistency` has been added in **utils.estimator_checks** to check their consistency. #7578 by Shubham Bhardwaj
- All checks in `utils.estimator_checks`, in particular *`utils.estimator_checks.check_estimator`* now accept estimator instances. Most other checks do not accept estimator classes any more. #9019 by Andreas Müller.
- Ensure that estimators' attributes ending with _ are not set in the constructor but only in the `fit` method. Most notably, ensemble estimators (deriving from `ensemble.BaseEnsemble`) now only have `self.estimators_` available after `fit`. #7464 by Lars Buitinck and Loic Esteve.

Code and Documentation Contributors

Thanks to everyone who has contributed to the maintenance and improvement of the project since version 0.18, including:

Joel Nothman, Loic Esteve, Andreas Mueller, Guillaume Lemaitre, Olivier Grisel, Hanmin Qin, Raghav RV, Alexandre Gramfort, themrmax, Aman Dalmia, Gael Varoquaux, Naoya Kanai, Tom Dupré la Tour, Rishikesh, Nelson Liu, Taehoon Lee, Nelle Varoquaux, Aashil, Mikhail Korobov, Sebastin Santy, Joan Massich, Roman Yurchak, RAKOTOARISON Herilalaina, Thierry Guillemot, Alexandre Abadie, Carol Willing, Balakumaran Manoharan, Josh Karnofsky, Vlad Niculae, Utkarsh Upadhyay, Dmitry Petrov, Minghui Liu, Srivatsan, Vincent Pham, Albert Thomas, Jake VanderPlas, Attractadore, JC Liu, alexandercbooth, chkoar, Óscar Nájera, Aarshay Jain, Kyle Gilliam, Ramana Subramanyam, CJ Carey, Clement Joudet, David Robles, He Chen, Joris Van den Bossche, Karan Desai, Katie Luangkote, Leland McInnes, Maniteja Nandana, Michele Lacchia, Sergei Lebedev, Shubham Bhardwaj, akshay0724, omtcyfz, rickiepark, waterponey, Vathsala Achar, jbDelafosse, Ralf Gommers, Ekaterina Krivich, Vivek Kumar, Ishank Gulati, Dave Elliott, ldirer, Reiichiro Nakano, Levi John Wolf, Mathieu Blondel, Sid Kapur, Dougal J. Sutherland, midinas, mikebenfield, Sourav Singh, Aseem Bansal, Ibraim Ganiev, Stephen Hoover, AishwaryaRK, Steven C. Howell, Gary

Foreman, Neeraj Gangwar, Tahar, Jon Crall, dokato, Kathy Chen, ferria, Thomas Moreau, Charlie Brummitt, Nicolas Goix, Adam Kleczewski, Sam Shleifer, Nikita Singh, Basil Beirouti, Giorgio Patrini, Manoj Kumar, Rafael Possas, James Bourbeau, James A. Bednar, Janine Harper, Jaye, Jean Helie, Jeremy Steward, Artsiom, John Wei, Jonathan LIgo, Jonathan Rahn, seanpwilliams, Arthur Mensch, Josh Levy, Julian Kuhlmann, Julien Aubert, Jörn Hees, Kai, shivamgargsya, Kat Hempstalk, Kaushik Lakshmikanth, Kennedy, Kenneth Lyons, Kenneth Myers, Kevin Yap, Kirill Bobyrev, Konstantin Podshumok, Arthur Imbert, Lee Murray, toastedcornflakes, Lera, Li Li, Arthur Douillard, Mainak Jas, tobycheese, Manraj Singh, Manvendra Singh, Marc Meketon, MarcoFalke, Matthew Brett, Matthias Gilch, Mehul Ahuja, Melanie Goetz, Meng, Peng, Michael Dezube, Michal Baumgartner, vibrantabhi19, Artem Golubin, Milen Paskov, Antonin Carette, Morikko, MrMjauh, NALEPA Emmanuel, Namiya, Antoine Wendlinger, Narine Kokhlikyan, NarineK, Nate Guerin, Angus Williams, Ang Lu, Nicole Vavrova, Nitish Pandey, Okhlopkov Daniil Olegovich, Andy Craze, Om Prakash, Parminder Singh, Patrick Carlson, Patrick Pei, Paul Ganssle, Paulo Haddad, Paweł Lorek, Peng Yu, Pete Bachant, Peter Bull, Peter Csizsek, Peter Wang, Pieter Arthur de Jong, Ping-Yao, Chang, Preston Parry, Puneet Mathur, Quentin Hibon, Andrew Smith, Andrew Jackson, 1kastner, Rameshwar Bhaskaran, Rebecca Bilbro, Remi Rampin, Andrea Esuli, Rob Hall, Robert Bradshaw, Romain Brault, Aman Pratik, Ruifeng Zheng, Russell Smith, Sachin Agarwal, Sailesh Choyal, Samson Tan, Samuël Weber, Sarah Brown, Sebastian Pölsterl, Sebastian Raschka, Sebastian Saeger, Alyssa Batula, Abhyuday Pratap Singh, Sergey Feldman, Sergul Aydore, Sharan Yalburgi, willduan, Siddharth Gupta, Sri Krishna, Almer, Stijn Tonk, Allen Riddell, Theofilos Papapanagiotou, Alison, Alexis Mignon, Tommy Boucher, Tommy Löfstedt, Toshihiro Kamishima, Tyler Folkman, Tyler Lanigan, Alexander Junge, Varun Shenoy, Victor Poughon, Vilhelm von Ehrenheim, Aleksandr Sandrovskii, Alan Yee, Vlasios Vasileiou, Warut Vijitbenjaronk, Yang Zhang, Yaroslav Halchenko, Yichuan Liu, Yuichi Fujikawa, affanv14, aivision2020, xor, andreh7, brady salz, campustrampus, Agamemnon Krasoulis, ditenberg, elena-sharova, filipj8, fukatani, gedeck, guiniol, guoci, hakaa1, hongkahjun, i-am-xhy, jakirkham, jaroslaw-weber, jayzed82, jeroko, jmontoyam, jonathan.striebel, josephsalmon, jschendel, leereeves, martin-hahn, mathurinm, mehak-sachdeva, mlewis1729, mlliou112, mthorrell, ndingwall, nuffe, yangarbiter, plagree, pldtc325, Breno Freitas, Brett Olsen, Brian A. Alfano, Brian Burns, polmauri, Brandon Carter, Charlton Austin, Chayant T15h, Chinmaya Pancholi, Christian Danielsen, Chung Yen, Chyi-Kwei Yau, pravarmahajan, DOHMATOB Elvis, Daniel LeJeune, Daniel Hnyk, Darius Morawiec, David DeTomaso, David Gasquez, David Haberthür, David Heryanto, David Kirkby, David Nicholson, rashchedrin, Deborah Gertrude Digges, Denis Engemann, Devansh D, Dickson, Bob Baxley, Don86, E. Lynch-Klarup, Ed Rogers, Elizabeth Ferriss, Ellen-Co2, Fabian Egli, Fang-Chieh Chou, Bing Tian Dai, Greg Stupp, Grzegorz Szpak, Bertrand Thirion, Hadrien Bertrand, Harizo Rajaona, zxcvbnius, Henry Lin, Holger Peters, Icyblade Dai, Igor Andriushchenko, Ilya, Isaac Laughlin, Iván Vallés, Aurélien Bellet, JPFrancoia, Jacob Schreiber, Asish Mahapatra

1.7.16 Version 0.18.2

June 20, 2017

Last release with Python 2.6 support

Scikit-learn 0.18 is the last major release of scikit-learn to support Python 2.6. Later versions of scikit-learn will require Python 2.7 or above.

Changelog

- Fixes for compatibility with NumPy 1.13.0: #7946 #8355 by Loic Esteve.
- Minor compatibility changes in the examples #9010 #8040 #9149.

Code Contributors

Aman Dalmia, Loic Esteve, Nate Guerin, Sergei Lebedev

1.7.17 Version 0.18.1

November 11, 2016

Changelog

Enhancements

- Improved `sample_without_replacement` speed by utilizing numpy.random.permutation for most cases. As a result, samples may differ in this release for a fixed random state. Affected estimators:
  - *ensemble.BaggingClassifier*
  - *ensemble.BaggingRegressor*
  - *linear_model.RANSACRegressor*
  - *model_selection.RandomizedSearchCV*
  - *random_projection.SparseRandomProjection*

  This also affects the *datasets.make_classification* method.

Bug fixes

- Fix issue where `min_grad_norm` and `n_iter_without_progress` parameters were not being utilised by *manifold.TSNE*. #6497 by Sebastian Säger
- Fix bug for svm's decision values when `decision_function_shape` is `ovr` in *svm.SVC*. *svm.SVC*'s decision_function was incorrect from versions 0.17.0 through 0.18.0. #7724 by Bing Tian Dai
- Attribute `explained_variance_ratio` of *discriminant_analysis.LinearDiscriminantAnalysis* calculated with SVD and Eigen solver are now of the same length. #7632 by JPFrancoia
- Fixes issue in *Univariate feature selection* where score functions were not accepting multi-label targets. #7676 by Mohammed Affan
- Fixed setting parameters when calling `fit` multiple times on *feature_selection.SelectFromModel*. #7756 by Andreas Müller
- Fixes issue in `partial_fit` method of *multiclass.OneVsRestClassifier* when number of classes used in `partial_fit` was less than the total number of classes in the data. #7786 by Srivatsan Ramesh
- Fixes issue in *calibration.CalibratedClassifierCV* where the sum of probabilities of each class for a data was not 1, and `CalibratedClassifierCV` now handles the case where the training set has less number of classes than the total data. #7799 by Srivatsan Ramesh
- Fix a bug where *sklearn.feature_selection.SelectFdr* did not exactly implement Benjamini-Hochberg procedure. It formerly may have selected fewer features than it should. #7490 by Peng Meng.
- *sklearn.manifold.LocallyLinearEmbedding* now correctly handles integer inputs. #6282 by Jake Vanderplas.
- The `min_weight_fraction_leaf` parameter of tree-based classifiers and regressors now assumes uniform sample weights by default if the `sample_weight` argument is not passed to the `fit` function. Previously, the parameter was silently ignored. #7301 by Nelson Liu.
- Numerical issue with *linear_model.RidgeCV* on centered data when `n_features > n_samples`. #6178 by Bertrand Thirion

- Tree splitting criterion classes' cloning/pickling is now memory safe #7680 by Ibraim Ganiev.
- Fixed a bug where *decomposition.NMF* sets its n_iters_ attribute in transform(). #7553 by Ekaterina Krivich.
- *sklearn.linear_model.LogisticRegressionCV* now correctly handles string labels. #5874 by Raghav RV.
- Fixed a bug where *sklearn.model_selection.train_test_split* raised an error when stratify is a list of string labels. #7593 by Raghav RV.
- Fixed a bug where *sklearn.model_selection.GridSearchCV* and *sklearn.model_selection.RandomizedSearchCV* were not pickleable because of a pickling bug in np.ma.MaskedArray. #7594 by Raghav RV.
- All cross-validation utilities in *sklearn.model_selection* now permit one time cross-validation splitters for the cv parameter. Also non-deterministic cross-validation splitters (where multiple calls to split produce dissimilar splits) can be used as cv parameter. The *sklearn.model_selection.GridSearchCV* will cross-validate each parameter setting on the split produced by the first split call to the cross-validation splitter. #7660 by Raghav RV.
- Fix bug where *preprocessing.MultiLabelBinarizer.fit_transform* returned an invalid CSR matrix. #7750 by CJ Carey.
- Fixed a bug where *metrics.pairwise.cosine_distances* could return a small negative distance. #7732 by Artsion.

API changes summary

Trees and forests

- The min_weight_fraction_leaf parameter of tree-based classifiers and regressors now assumes uniform sample weights by default if the sample_weight argument is not passed to the fit function. Previously, the parameter was silently ignored. #7301 by Nelson Liu.
- Tree splitting criterion classes' cloning/pickling is now memory safe. #7680 by Ibraim Ganiev.

Linear, kernelized and related models

- Length of explained_variance_ratio of *discriminant_analysis.LinearDiscriminantAnalysis* changed for both Eigen and SVD solvers. The attribute has now a length of min(n_components, n_classes - 1). #7632 by JPFrancoia
- Numerical issue with *linear_model.RidgeCV* on centered data when n_features > n_samples. #6178 by Bertrand Thirion

1.7.18 Version 0.18

September 28, 2016

Last release with Python 2.6 support

Scikit-learn 0.18 will be the last version of scikit-learn to support Python 2.6. Later versions of scikit-learn will require Python 2.7 or above.

Model Selection Enhancements and API Changes

- **The model_selection module**

  The new module *sklearn.model_selection*, which groups together the functionalities of formerly sklearn.cross_validation, sklearn.grid_search and sklearn.learning_curve, introduces new possibilities such as nested cross-validation and better manipulation of parameter searches with Pandas.

  Many things will stay the same but there are some key differences. Read below to know more about the changes.

- **Data-independent CV splitters enabling nested cross-validation**

  The new cross-validation splitters, defined in the *sklearn.model_selection*, are no longer initialized with any data-dependent parameters such as y. Instead they expose a split method that takes in the data and yields a generator for the different splits.

  This change makes it possible to use the cross-validation splitters to perform nested cross-validation, facilitated by *model_selection.GridSearchCV* and *model_selection.RandomizedSearchCV* utilities.

- **The enhanced cv_results_ attribute**

  The new cv_results_ attribute (of *model_selection.GridSearchCV* and *model_selection.RandomizedSearchCV*) introduced in lieu of the grid_scores_ attribute is a dict of 1D arrays with elements in each array corresponding to the parameter settings (i.e. search candidates).

  The cv_results_ dict can be easily imported into pandas as a DataFrame for exploring the search results.

  The cv_results_ arrays include scores for each cross-validation split (with keys such as 'split0_test_score'), as well as their mean ('mean_test_score') and standard deviation ('std_test_score').

  The ranks for the search candidates (based on their mean cross-validation score) is available at cv_results_['rank_test_score'].

  The parameter values for each parameter is stored separately as numpy masked object arrays. The value, for that search candidate, is masked if the corresponding parameter is not applicable. Additionally a list of all the parameter dicts are stored at cv_results_['params'].

- **Parameters n_folds and n_iter renamed to n_splits**

  Some parameter names have changed: The n_folds parameter in new *model_selection.KFold*, *model_selection.GroupKFold* (see below for the name change), and *model_selection.StratifiedKFold* is now renamed to n_splits. The n_iter parameter in *model_selection.ShuffleSplit*, the new class *model_selection.GroupShuffleSplit* and *model_selection.StratifiedShuffleSplit* is now renamed to n_splits.

- **Rename of splitter classes which accepts group labels along with data**

  The cross-validation splitters LabelKFold, LabelShuffleSplit, LeaveOneLabelOut and LeavePLabelOut have been renamed to *model_selection.GroupKFold*, *model_selection.GroupShuffleSplit*, *model_selection.LeaveOneGroupOut* and *model_selection.LeavePGroupsOut* respectively.

  Note the change from singular to plural form in *model_selection.LeavePGroupsOut*.

- **Fit parameter labels renamed to groups**

  The labels parameter in the split method of the newly renamed splitters *model_selection.GroupKFold*, *model_selection.LeaveOneGroupOut*, *model_selection.LeavePGroupsOut*, *model_selection.GroupShuffleSplit* is renamed to groups following the new nomenclature of their class names.

- **Parameter n_labels renamed to n_groups**

  The parameter `n_labels` in the newly renamed *model_selection.LeavePGroupsOut* is changed to `n_groups`.

- Training scores and Timing information

  `cv_results_` also includes the training scores for each cross-validation split (with keys such as `'split0_train_score'`), as well as their mean (`'mean_train_score'`) and standard deviation (`'std_train_score'`). To avoid the cost of evaluating training score, set `return_train_score=False`.

  Additionally the mean and standard deviation of the times taken to split, train and score the model across all the cross-validation splits is available at the key `'mean_time'` and `'std_time'` respectively.

Changelog

New features

Classifiers and Regressors

- The Gaussian Process module has been reimplemented and now offers classification and regression estimators through *gaussian_process.GaussianProcessClassifier* and *gaussian_process.GaussianProcessRegressor*. Among other things, the new implementation supports kernel engineering, gradient-based hyperparameter optimization or sampling of functions from GP prior and GP posterior. Extensive documentation and examples are provided. By Jan Hendrik Metzen.
- Added new supervised learning algorithm: *Multi-layer Perceptron* #3204 by Issam H. Laradji
- Added *linear_model.HuberRegressor*, a linear model robust to outliers. #5291 by Manoj Kumar.
- Added the *multioutput.MultiOutputRegressor* meta-estimator. It converts single output regressors to multi-output regressors by fitting one regressor per output. By Tim Head.

Other estimators

- New *mixture.GaussianMixture* and *mixture.BayesianGaussianMixture* replace former mixture models, employing faster inference for sounder results. #7295 by Wei Xue and Thierry Guillemot.
- Class `decomposition.RandomizedPCA` is now factored into *decomposition.PCA* and it is available calling with parameter `svd_solver='randomized'`. The default number of `n_iter` for `'randomized'` has changed to 4. The old behavior of PCA is recovered by `svd_solver='full'`. An additional solver calls `arpack` and performs truncated (non-randomized) SVD. By default, the best solver is selected depending on the size of the input and the number of components requested. #5299 by Giorgio Patrini.
- Added two functions for mutual information estimation: *feature_selection.mutual_info_classif* and *feature_selection.mutual_info_regression*. These functions can be used in *feature_selection.SelectKBest* and *feature_selection.SelectPercentile* as score functions. By Andrea Bravi and Nikolay Mayorov.
- Added the *ensemble.IsolationForest* class for anomaly detection based on random forests. By Nicolas Goix.
- Added `algorithm="elkan"` to *cluster.KMeans* implementing Elkan's fast K-Means algorithm. By Andreas Müller.

Model selection and evaluation

- Added `metrics.cluster.fowlkes_mallows_score`, the Fowlkes Mallows Index which measures the similarity of two clusterings of a set of points By Arnaud Fouchet and Thierry Guillemot.

- Added *metrics.calinski_harabaz_score*, which computes the Calinski and Harabaz score to evaluate the resulting clustering of a set of points. By Arnaud Fouchet and Thierry Guillemot.
- Added new cross-validation splitter *model_selection.TimeSeriesSplit* to handle time series data. #6586 by YenChen Lin
- The cross-validation iterators are replaced by cross-validation splitters available from *sklearn.model_selection*, allowing for nested cross-validation. See *Model Selection Enhancements and API Changes* for more information. #4294 by Raghav RV.

Enhancements

Trees and ensembles

- Added a new splitting criterion for *tree.DecisionTreeRegressor*, the mean absolute error. This criterion can also be used in *ensemble.ExtraTreesRegressor*, *ensemble.RandomForestRegressor*, and the gradient boosting estimators. #6667 by Nelson Liu.
- Added weighted impurity-based early stopping criterion for decision tree growth. #6954 by Nelson Liu
- The random forest, extra tree and decision tree estimators now has a method `decision_path` which returns the decision path of samples in the tree. By Arnaud Joly.
- A new example has been added unveiling the decision tree structure. By Arnaud Joly.
- Random forest, extra trees, decision trees and gradient boosting estimator accept the parameter `min_samples_split` and `min_samples_leaf` provided as a percentage of the training samples. By yelite and Arnaud Joly.
- Gradient boosting estimators accept the parameter `criterion` to specify to splitting criterion used in built decision trees. #6667 by Nelson Liu.
- The memory footprint is reduced (sometimes greatly) for `ensemble.bagging.BaseBagging` and classes that inherit from it, i.e, *ensemble.BaggingClassifier*, *ensemble.BaggingRegressor*, and *ensemble.IsolationForest*, by dynamically generating attribute `estimators_samples_` only when it is needed. By David Staub.
- Added `n_jobs` and `sample_weight` parameters for *ensemble.VotingClassifier* to fit underlying estimators in parallel. #5805 by Ibraim Ganiev.

Linear, kernelized and related models

- In *linear_model.LogisticRegression*, the SAG solver is now available in the multinomial case. #5251 by Tom Dupre la Tour.
- *linear_model.RANSACRegressor*, *svm.LinearSVC* and *svm.LinearSVR* now support `sample_weight`. By Imaculate.
- Add parameter `loss` to *linear_model.RANSACRegressor* to measure the error on the samples for every trial. By Manoj Kumar.
- Prediction of out-of-sample events with Isotonic Regression (*isotonic.IsotonicRegression*) is now much faster (over 1000x in tests with synthetic data). By Jonathan Arfa.
- Isotonic regression (*isotonic.IsotonicRegression*) now uses a better algorithm to avoid `O(n^2)` behavior in pathological cases, and is also generally faster (##6691). By Antony Lee.
- *naive_bayes.GaussianNB* now accepts data-independent class-priors through the parameter `priors`. By Guillaume Lemaitre.

- *linear_model.ElasticNet* and *linear_model.Lasso* now works with `np.float32` input data without converting it into `np.float64`. This allows to reduce the memory consumption. #6913 by YenChen Lin.
- *semi_supervised.LabelPropagation* and *semi_supervised.LabelSpreading* now accept arbitrary kernel functions in addition to strings `knn` and `rbf`. #5762 by Utkarsh Upadhyay.

Decomposition, manifold learning and clustering

- Added `inverse_transform` function to *decomposition.NMF* to compute data matrix of original shape. By Anish Shah.
- *cluster.KMeans* and *cluster.MiniBatchKMeans* now works with `np.float32` and `np.float64` input data without converting it. This allows to reduce the memory consumption by using `np.float32`. #6846 by Sebastian Säger and YenChen Lin.

Preprocessing and feature selection

- *preprocessing.RobustScaler* now accepts `quantile_range` parameter. #5929 by Konstantin Podshumok.
- *feature_extraction.FeatureHasher* now accepts string values. #6173 by Ryad Zenine and Devashish Deshpande.
- Keyword arguments can now be supplied to `func` in *preprocessing.FunctionTransformer* by means of the `kw_args` parameter. By Brian McFee.
- *feature_selection.SelectKBest* and *feature_selection.SelectPercentile* now accept score functions that take X, y as input and return only the scores. By Nikolay Mayorov.

Model evaluation and meta-estimators

- *multiclass.OneVsOneClassifier* and *multiclass.OneVsRestClassifier* now support `partial_fit`. By Asish Panda and Philipp Dowling.
- Added support for substituting or disabling *pipeline.Pipeline* and *pipeline.FeatureUnion* components using the `set_params` interface that powers `sklearn.grid_search`. See *Selecting dimensionality reduction with Pipeline and GridSearchCV* By Joel Nothman and Robert McGibbon.
- The new `cv_results_` attribute of *model_selection.GridSearchCV* (and *model_selection.RandomizedSearchCV*) can be easily imported into pandas as a `DataFrame`. Ref *Model Selection Enhancements and API Changes* for more information. #6697 by Raghav RV.
- Generalization of *model_selection.cross_val_predict*. One can pass method names such as `predict_proba` to be used in the cross validation framework instead of the default `predict`. By Ori Ziv and Sears Merritt.
- The training scores and time taken for training followed by scoring for each search candidate are now available at the `cv_results_` dict. See *Model Selection Enhancements and API Changes* for more information. #7325 by Eugene Chen and Raghav RV.

Metrics

- Added `labels` flag to *metrics.log_loss* to explicitly provide the labels when the number of classes in `y_true` and `y_pred` differ. #7239 by Hong Guangguo with help from Mads Jensen and Nelson Liu.
- Support sparse contingency matrices in cluster evaluation (`metrics.cluster.supervised`) to scale to a large number of clusters. #7419 by Gregory Stupp and Joel Nothman.
- Add `sample_weight` parameter to *metrics.matthews_corrcoef*. By Jatin Shah and Raghav RV.
- Speed up *metrics.silhouette_score* by using vectorized operations. By Manoj Kumar.
- Add `sample_weight` parameter to *metrics.confusion_matrix*. By Bernardo Stein.

Miscellaneous

- Added `n_jobs` parameter to *`feature_selection.RFECV`* to compute the score on the test folds in parallel. By Manoj Kumar
- Codebase does not contain C/C++ cython generated files: they are generated during build. Distribution packages will still contain generated C/C++ files. By Arthur Mensch.
- Reduce the memory usage for 32-bit float input arrays of `utils.sparse_func.mean_variance_axis` and `utils.sparse_func.incr_mean_variance_axis` by supporting cython fused types. By YenChen Lin.
- The `ignore_warnings` now accept a category argument to ignore only the warnings of a specified type. By Thierry Guillemot.
- Added parameter `return_X_y` and return type `(data, target) : tuple` option to `load_iris` dataset #7049, `load_breast_cancer` dataset #7152, `load_digits` dataset, `load_diabetes` dataset, `load_linnerud` dataset, `load_boston` dataset #7154 by Manvendra Singh.
- Simplification of the `clone` function, deprecate support for estimators that modify parameters in `__init__`. #5540 by Andreas Müller.
- When unpickling a scikit-learn estimator in a different version than the one the estimator was trained with, a `UserWarning` is raised, see *the documentation on model persistence* for more details. (#7248) By Andreas Müller.

Bug fixes

Trees and ensembles

- Random forest, extra trees, decision trees and gradient boosting won't accept anymore `min_samples_split=1` as at least 2 samples are required to split a decision tree node. By Arnaud Joly
- *`ensemble.VotingClassifier`* now raises `NotFittedError` if `predict`, `transform` or `predict_proba` are called on the non-fitted estimator. by Sebastian Raschka.
- Fix bug where *`ensemble.AdaBoostClassifier`* and *`ensemble.AdaBoostRegressor`* would perform poorly if the `random_state` was fixed (#7411). By Joel Nothman.
- Fix bug in ensembles with randomization where the ensemble would not set `random_state` on base estimators in a pipeline or similar nesting. (#7411). Note, results for *`ensemble.BaggingClassifier`* *`ensemble.BaggingRegressor`*, *`ensemble.AdaBoostClassifier`* and *`ensemble.AdaBoostRegressor`* will now differ from previous versions. By Joel Nothman.

Linear, kernelized and related models

- Fixed incorrect gradient computation for `loss='squared_epsilon_insensitive'` in *`linear_model.SGDClassifier`* and *`linear_model.SGDRegressor`* (#6764). By Wenhua Yang.
- Fix bug in *`linear_model.LogisticRegressionCV`* where `solver='liblinear'` did not accept `class_weights='balanced`. (#6817). By Tom Dupre la Tour.
- Fix bug in *`neighbors.RadiusNeighborsClassifier`* where an error occurred when there were outliers being labelled and a weight function specified (#6902). By LeonieBorne.
- Fix *`linear_model.ElasticNet`* sparse decision function to match output with dense in the multioutput case.

Decomposition, manifold learning and clustering

- `decomposition.RandomizedPCA` default number of `iterated_power` is 4 instead of 3. #5141 by Giorgio Patrini.
- *`utils.extmath.randomized_svd`* performs 4 power iterations by default, instead or 0. In practice this is enough for obtaining a good approximation of the true eigenvalues/vectors in the presence of noise. When `n_components` is small (`< .1 * min(X.shape)`) `n_iter` is set to 7, unless the user specifies a higher number. This improves precision with few components. #5299 by Giorgio Patrini.
- Whiten/non-whiten inconsistency between components of *`decomposition.PCA`* and `decomposition.RandomizedPCA` (now factored into PCA, see the New features) is fixed. `components_` are stored with no whitening. #5299 by Giorgio Patrini.
- Fixed bug in *`manifold.spectral_embedding`* where diagonal of unnormalized Laplacian matrix was incorrectly set to 1. #4995 by Peter Fischer.
- Fixed incorrect initialization of `utils.arpack.eigsh` on all occurrences. Affects `cluster.bicluster.SpectralBiclustering`, *`decomposition.KernelPCA`*, *`manifold.LocallyLinearEmbedding`*, and *`manifold.SpectralEmbedding`* (#5012). By Peter Fischer.
- Attribute `explained_variance_ratio_` calculated with the SVD solver of *`discriminant_analysis.LinearDiscriminantAnalysis`* now returns correct results. By JPFrancoia

Preprocessing and feature selection

- `preprocessing.data._transform_selected` now always passes a copy of `X` to transform function when `copy=True` (#7194). By Caio Oliveira.

Model evaluation and meta-estimators

- *`model_selection.StratifiedKFold`* now raises error if all n_labels for individual classes is less than n_folds. #6182 by Devashish Deshpande.
- Fixed bug in *`model_selection.StratifiedShuffleSplit`* where train and test sample could overlap in some edge cases, see #6121 for more details. By Loic Esteve.
- Fix in *`sklearn.model_selection.StratifiedShuffleSplit`* to return splits of size `train_size` and `test_size` in all cases (#6472). By Andreas Müller.
- Cross-validation of `OneVsOneClassifier` and `OneVsRestClassifier` now works with precomputed kernels. #7350 by Russell Smith.
- Fix incomplete `predict_proba` method delegation from *`model_selection.GridSearchCV`* to *`linear_model.SGDClassifier`* (#7159) by Yichuan Liu.

Metrics

- Fix bug in *`metrics.silhouette_score`* in which clusters of size 1 were incorrectly scored. They should get a score of 0. By Joel Nothman.
- Fix bug in *`metrics.silhouette_samples`* so that it now works with arbitrary labels, not just those ranging from 0 to n_clusters - 1.
- Fix bug where expected and adjusted mutual information were incorrect if cluster contingency cells exceeded `2**16`. By Joel Nothman.
- `metrics.pairwise.pairwise_distances` now converts arrays to boolean arrays when required in `scipy.spatial.distance`. #5460 by Tom Dupre la Tour.
- Fix sparse input support in *`metrics.silhouette_score`* as well as example examples/text/document_clustering.py. By YenChen Lin.
- *`metrics.roc_curve`* and *`metrics.precision_recall_curve`* no longer round `y_score` values when creating ROC curves; this was causing problems for users with very small differences in scores (#7353).

Miscellaneous

- `model_selection.tests._search._check_param_grid` now works correctly with all types that extends/implements `Sequence` (except string), including range (Python 3.x) and xrange (Python 2.x). #7323 by Viacheslav Kovalevskyi.
- *`utils.extmath.randomized_range_finder`* is more numerically stable when many power iterations are requested, since it applies LU normalization by default. If `n_iter<2` numerical issues are unlikely, thus no normalization is applied. Other normalization options are available: `'none', 'LU'` and `'QR'`. #5141 by Giorgio Patrini.
- Fix a bug where some formats of `scipy.sparse` matrix, and estimators with them as parameters, could not be passed to *`base.clone`*. By Loic Esteve.
- *`datasets.load_svmlight_file`* now is able to read long int QID values. #7101 by Ibraim Ganiev.

API changes summary

Linear, kernelized and related models

- `residual_metric` has been deprecated in *`linear_model.RANSACRegressor`*. Use `loss` instead. By Manoj Kumar.
- Access to public attributes `.X_` and `.y_` has been deprecated in *`isotonic.IsotonicRegression`*. By Jonathan Arfa.

Decomposition, manifold learning and clustering

- The old `mixture.DPGMM` is deprecated in favor of the new *`mixture.BayesianGaussianMixture`* (with the parameter `weight_concentration_prior_type='dirichlet_process'`). The new class solves the computational problems of the old class and computes the Gaussian mixture with a Dirichlet process prior faster than before. #7295 by Wei Xue and Thierry Guillemot.
- The old `mixture.VBGMM` is deprecated in favor of the new *`mixture.BayesianGaussianMixture`* (with the parameter `weight_concentration_prior_type='dirichlet_distribution'`). The new class solves the computational problems of the old class and computes the Variational Bayesian Gaussian mixture faster than before. #6651 by Wei Xue and Thierry Guillemot.
- The old `mixture.GMM` is deprecated in favor of the new *`mixture.GaussianMixture`*. The new class computes the Gaussian mixture faster than before and some of computational problems have been solved. #6666 by Wei Xue and Thierry Guillemot.

Model evaluation and meta-estimators

- The `sklearn.cross_validation`, `sklearn.grid_search` and `sklearn.learning_curve` have been deprecated and the classes and functions have been reorganized into the *`sklearn.model_selection`* module. Ref *Model Selection Enhancements and API Changes* for more information. #4294 by Raghav RV.
- The `grid_scores_` attribute of *`model_selection.GridSearchCV`* and *`model_selection.RandomizedSearchCV`* is deprecated in favor of the attribute `cv_results_`. Ref *Model Selection Enhancements and API Changes* for more information. #6697 by Raghav RV.
- The parameters `n_iter` or `n_folds` in old CV splitters are replaced by the new parameter `n_splits` since it can provide a consistent and unambiguous interface to represent the number of train-test splits. #7187 by YenChen Lin.
- `classes` parameter was renamed to `labels` in *`metrics.hamming_loss`*. #7260 by Sebastián Vanrell.

- The splitter classes `LabelKFold`, `LabelShuffleSplit`, `LeaveOneLabelOut` and `LeavePLabelsOut` are renamed to *model_selection.GroupKFold*, *model_selection.GroupShuffleSplit*, *model_selection.LeaveOneGroupOut* and *model_selection.LeavePGroupsOut* respectively. Also the parameter `labels` in the `split` method of the newly renamed splitters *model_selection.LeaveOneGroupOut* and *model_selection.LeavePGroupsOut* is renamed to `groups`. Additionally in *model_selection.LeavePGroupsOut*, the parameter `n_labels` is renamed to `n_groups`. #6660 by Raghav RV.
- Error and loss names for `scoring` parameters are now prefixed by `'neg_'`, such as `neg_mean_squared_error`. The unprefixed versions are deprecated and will be removed in version 0.20. #7261 by Tim Head.

Code Contributors

Aditya Joshi, Alejandro, Alexander Fabisch, Alexander Loginov, Alexander Minyushkin, Alexander Rudy, Alexandre Abadie, Alexandre Abraham, Alexandre Gramfort, Alexandre Saint, alexfields, Alvaro Ulloa, alyssaq, Amlan Kar, Andreas Mueller, andrew giessel, Andrew Jackson, Andrew McCulloh, Andrew Murray, Anish Shah, Arafat, Archit Sharma, Ariel Rokem, Arnaud Joly, Arnaud Rachez, Arthur Mensch, Ash Hoover, asnt, b0noI, Behzad Tabibian, Bernardo, Bernhard Kratzwald, Bhargav Mangipudi, blakeflei, Boyuan Deng, Brandon Carter, Brett Naul, Brian McFee, Caio Oliveira, Camilo Lamus, Carol Willing, Cass, CeShine Lee, Charles Truong, Chyi-Kwei Yau, CJ Carey, codevig, Colin Ni, Dan Shiebler, Daniel, Daniel Hnyk, David Ellis, David Nicholson, David Staub, David Thaler, David Warshaw, Davide Lasagna, Deborah, definitelyuncertain, Didi Bar-Zev, djipey, dsquareindia, edwinENSAE, Elias Kuthe, Elvis DOHMATOB, Ethan White, Fabian Pedregosa, Fabio Ticconi, fisache, Florian Wilhelm, Francis, Francis O'Donovan, Gael Varoquaux, Ganiev Ibraim, ghg, Gilles Louppe, Giorgio Patrini, Giovanni Cherubin, Giovanni Lanzani, Glenn Qian, Gordon Mohr, govin-vatsan, Graham Clenaghan, Greg Reda, Greg Stupp, Guillaume Lemaitre, Gustav Mörtberg, halwai, Harizo Rajaona, Harry Mavroforakis, hashcode55, hdmetor, Henry Lin, Hobson Lane, Hugo Bowne-Anderson, Igor Andriushchenko, Imaculate, Inki Hwang, Isaac Sijaranamual, Ishank Gulati, Issam Laradji, Iver Jordal, jackmartin, Jacob Schreiber, Jake Vanderplas, James Fiedler, James Routley, Jan Zikes, Janna Brettingen, jarfa, Jason Laska, jblackburne, jeff levesque, Jeffrey Blackburne, Jeffrey04, Jeremy Hintz, jeremynixon, Jeroen, Jessica Yung, Jill-Jênn Vie, Jimmy Jia, Jiyuan Qian, Joel Nothman, johannah, John, John Boersma, John Kirkham, John Moeller, jonathan.striebel, joncrall, Jordi, Joseph Munoz, Joshua Cook, JPFrancoia, jrfiedler, JulianKahnert, juliathebrave, kaichogami, KamalakerDadi, Kenneth Lyons, Kevin Wang, kingjr, kjell, Konstantin Podshumok, Kornel Kielczewski, Krishna Kalyan, krishnakalyan3, Kvle Putnam, Kyle Jackson, Lars Buitinck, ldavid, LeiG, LeightonZhang, Leland McInnes, Liang-Chi Hsieh, Lilian Besson, lizsz, Loic Esteve, Louis Tiao, Léonie Borne, Mads Jensen, Maniteja Nandana, Manoj Kumar, Manvendra Singh, Marco, Mario Krell, Mark Bao, Mark Szepieniec, Martin Madsen, MartinBpr, MaryanMorel, Massil, Matheus, Mathieu Blondel, Mathieu Dubois, Matteo, Matthias Ekman, Max Moroz, Michael Scherer, michiaki ariga, Mikhail Korobov, Moussa Taifi, mrandrewandrade, Mridul Seth, nadya-p, Naoya Kanai, Nate George, Nelle Varoquaux, Nelson Liu, Nick James, NickleDave, Nico, Nicolas Goix, Nikolay Mayorov, ningchi, nlathia, okbalefthanded, Okhlopkov, Olivier Grisel, Panos Louridas, Paul Strickland, Perrine Letellier, pestrickland, Peter Fischer, Pieter, Ping-Yao, Chang, practicalswift, Preston Parry, Qimu Zheng, Rachit Kansal, Raghav RV, Ralf Gommers, Ramana.S, Rammig, Randy Olson, Rob Alexander, Robert Lutz, Robin Schucker, Rohan Jain, Ruifeng Zheng, Ryan Yu, Rémy Léone, saihttam, Saiwing Yeung, Sam Shleifer, Samuel St-Jean, Sartaj Singh, Sasank Chilamkurthy, saurabh.bansod, Scott Andrews, Scott Lowe, seales, Sebastian Raschka, Sebastian Saeger, Sebastián Vanrell, Sergei Lebedev, shagun Sodhani, shanmuga cv, Shashank Shekhar, shawpan, shengxiduan, Shota, shuckle16, Skipper Seabold, sklearn-ci, SmedbergM, srvanrell, Sébastien Lerique, Taranjeet, themrmax, Thierry, Thierry Guillemot, Thomas, Thomas Hallock, Thomas Moreau, Tim Head, tKammy, toastedcornflakes, Tom, TomDLT, Toshihiro Kamishima, tracer0tong, Trent Hauck, trevorstephens, Tue Vo, Varun, Varun Jewalikar, Viacheslav, Vighnesh Birodkar, Vikram, Villu Ruusmann, Vinayak Mehta, walter, waterponey, Wenhua Yang, Wenjian Huang, Will Welch, wyseguy7, xyguo, yanlend, Yaroslav Halchenko, yelite, Yen, YenChenLin, Yichuan Liu, Yoav Ram, Yoshiki, Zheng RuiFeng, zivori, Óscar Nájera

1.7.19 Version 0.17.1

February 18, 2016

Changelog

Bug fixes

- Upgrade vendored joblib to version 0.9.4 that fixes an important bug in `joblib.Parallel` that can silently yield to wrong results when working on datasets larger than 1MB: https://github.com/joblib/joblib/blob/0.9.4/CHANGES.rst
- Fixed reading of Bunch pickles generated with scikit-learn version <= 0.16. This can affect users who have already downloaded a dataset with scikit-learn 0.16 and are loading it with scikit-learn 0.17. See #6196 for how this affected *datasets.fetch_20newsgroups*. By Loic Esteve.
- Fixed a bug that prevented using ROC AUC score to perform grid search on several CPU / cores on large arrays. See #6147 By Olivier Grisel.
- Fixed a bug that prevented to properly set the `presort` parameter in *ensemble.GradientBoostingRegressor*. See #5857 By Andrew McCulloh.
- Fixed a joblib error when evaluating the perplexity of a *decomposition.LatentDirichletAllocation* model. See #6258 By Chyi-Kwei Yau.

1.7.20 Version 0.17

November 5, 2015

Changelog

New features

- All the Scaler classes but *preprocessing.RobustScaler* can be fitted online by calling `partial_fit`. By Giorgio Patrini.
- The new class *ensemble.VotingClassifier* implements a "majority rule" / "soft voting" ensemble classifier to combine estimators for classification. By Sebastian Raschka.
- The new class *preprocessing.RobustScaler* provides an alternative to *preprocessing.StandardScaler* for feature-wise centering and range normalization that is robust to outliers. By Thomas Unterthiner.
- The new class *preprocessing.MaxAbsScaler* provides an alternative to *preprocessing.MinMaxScaler* for feature-wise range normalization when the data is already centered or sparse. By Thomas Unterthiner.
- The new class *preprocessing.FunctionTransformer* turns a Python function into a `Pipeline`-compatible transformer object. By Joe Jevnik.
- The new classes `cross_validation.LabelKFold` and `cross_validation.LabelShuffleSplit` generate train-test folds, respectively similar to `cross_validation.KFold` and `cross_validation.ShuffleSplit`, except that the folds are conditioned on a label array. By Brian McFee, Jean Kossaifi and Gilles Louppe.

- *decomposition.LatentDirichletAllocation* implements the Latent Dirichlet Allocation topic model with online variational inference. By Chyi-Kwei Yau, with code based on an implementation by Matt Hoffman. (#3659)
- The new solver `sag` implements a Stochastic Average Gradient descent and is available in both *linear_model.LogisticRegression* and *linear_model.Ridge*. This solver is very efficient for large datasets. By Danny Sullivan and Tom Dupre la Tour. (#4738)
- The new solver `cd` implements a Coordinate Descent in *decomposition.NMF*. Previous solver based on Projected Gradient is still available setting new parameter `solver` to `pg`, but is deprecated and will be removed in 0.19, along with `decomposition.ProjectedGradientNMF` and parameters `sparseness`, `eta`, `beta` and `nls_max_iter`. New parameters `alpha` and `l1_ratio` control L1 and L2 regularization, and `shuffle` adds a shuffling step in the `cd` solver. By Tom Dupre la Tour and Mathieu Blondel.

Enhancements

- *manifold.TSNE* now supports approximate optimization via the Barnes-Hut method, leading to much faster fitting. By Christopher Erick Moody. (#4025)
- `cluster.mean_shift_.MeanShift` now supports parallel execution, as implemented in the `mean_shift` function. By Martino Sorbaro.
- *naive_bayes.GaussianNB* now supports fitting with `sample_weight`. By Jan Hendrik Metzen.
- *dummy.DummyClassifier* now supports a prior fitting strategy. By Arnaud Joly.
- Added a `fit_predict` method for `mixture.GMM` and subclasses. By Cory Lorenz.
- Added the *metrics.label_ranking_loss* metric. By Arnaud Joly.
- Added the *metrics.cohen_kappa_score* metric.
- Added a `warm_start` constructor parameter to the bagging ensemble models to increase the size of the ensemble. By Tim Head.
- Added option to use multi-output regression metrics without averaging. By Konstantin Shmelkov and Michael Eickenberg.
- Added `stratify` option to `cross_validation.train_test_split` for stratified splitting. By Miroslav Batchkarov.
- The *tree.export_graphviz* function now supports aesthetic improvements for *tree.DecisionTreeClassifier* and *tree.DecisionTreeRegressor*, including options for coloring nodes by their majority class or impurity, showing variable names, and using node proportions instead of raw sample counts. By Trevor Stephens.
- Improved speed of `newton-cg` solver in *linear_model.LogisticRegression*, by avoiding loss computation. By Mathieu Blondel and Tom Dupre la Tour.
- The `class_weight="auto"` heuristic in classifiers supporting `class_weight` was deprecated and replaced by the `class_weight="balanced"` option, which has a simpler formula and interpretation. By Hanna Wallach and Andreas Müller.
- Add `class_weight` parameter to automatically weight samples by class frequency for *linear_model.PassiveAggressiveClassifier*. By Trevor Stephens.
- Added backlinks from the API reference pages to the user guide. By Andreas Müller.
- The `labels` parameter to *sklearn.metrics.f1_score*, *sklearn.metrics.fbeta_score*, *sklearn.metrics.recall_score* and *sklearn.metrics.precision_score* has been ex-

tended. It is now possible to ignore one or more labels, such as where a multiclass problem has a majority class to ignore. By Joel Nothman.

- Add `sample_weight` support to *linear_model.RidgeClassifier*. By Trevor Stephens.
- Provide an option for sparse output from *sklearn.metrics.pairwise.cosine_similarity*. By Jaidev Deshpande.
- Add `minmax_scale` to provide a function interface for `MinMaxScaler`. By Thomas Unterthiner.
- `dump_svmlight_file` now handles multi-label datasets. By Chih-Wei Chang.
- RCV1 dataset loader (*sklearn.datasets.fetch_rcv1*). By Tom Dupre la Tour.
- The "Wisconsin Breast Cancer" classical two-class classification dataset is now included in scikit-learn, available with `sklearn.dataset.load_breast_cancer`.
- Upgraded to joblib 0.9.3 to benefit from the new automatic batching of short tasks. This makes it possible for scikit-learn to benefit from parallelism when many very short tasks are executed in parallel, for instance by the `grid_search.GridSearchCV` meta-estimator with `n_jobs > 1` used with a large grid of parameters on a small dataset. By Vlad Niculae, Olivier Grisel and Loic Esteve.
- For more details about changes in joblib 0.9.3 see the release notes: https://github.com/joblib/joblib/blob/master/CHANGES.rst#release-093
- Improved speed (3 times per iteration) of `decomposition.DictLearning` with coordinate descent method from *linear_model.Lasso*. By Arthur Mensch.
- Parallel processing (threaded) for queries of nearest neighbors (using the ball-tree) by Nikolay Mayorov.
- Allow *datasets.make_multilabel_classification* to output a sparse `y`. By Kashif Rasul.
- *cluster.DBSCAN* now accepts a sparse matrix of precomputed distances, allowing memory-efficient distance precomputation. By Joel Nothman.
- *tree.DecisionTreeClassifier* now exposes an `apply` method for retrieving the leaf indices samples are predicted as. By Daniel Galvez and Gilles Louppe.
- Speed up decision tree regressors, random forest regressors, extra trees regressors and gradient boosting estimators by computing a proxy of the impurity improvement during the tree growth. The proxy quantity is such that the split that maximizes this value also maximizes the impurity improvement. By Arnaud Joly, Jacob Schreiber and Gilles Louppe.
- Speed up tree based methods by reducing the number of computations needed when computing the impurity measure taking into account linear relationship of the computed statistics. The effect is particularly visible with extra trees and on datasets with categorical or sparse features. By Arnaud Joly.
- *ensemble.GradientBoostingRegressor* and *ensemble.GradientBoostingClassifier* now expose an `apply` method for retrieving the leaf indices each sample ends up in under each try. By Jacob Schreiber.
- Add `sample_weight` support to *linear_model.LinearRegression*. By Sonny Hu. (##4881)
- Add `n_iter_without_progress` to *manifold.TSNE* to control the stopping criterion. By Santi Villalba. (#5186)
- Added optional parameter `random_state` in *linear_model.Ridge* , to set the seed of the pseudo random generator used in `sag` solver. By Tom Dupre la Tour.
- Added optional parameter `warm_start` in *linear_model.LogisticRegression*. If set to True, the solvers `lbfgs`, `newton-cg` and `sag` will be initialized with the coefficients computed in the previous fit. By Tom Dupre la Tour.

- Added `sample_weight` support to *linear_model.LogisticRegression* for the `lbfgs`, `newton-cg`, and `sag` solvers. By Valentin Stolbunov. Support added to the `liblinear` solver. By Manoj Kumar.
- Added optional parameter `presort` to *ensemble.GradientBoostingRegressor* and *ensemble.GradientBoostingClassifier*, keeping default behavior the same. This allows gradient boosters to turn off presorting when building deep trees or using sparse data. By Jacob Schreiber.
- Altered *metrics.roc_curve* to drop unnecessary thresholds by default. By Graham Clenaghan.
- Added *feature_selection.SelectFromModel* meta-transformer which can be used along with estimators that have `coef_` or `feature_importances_` attribute to select important features of the input data. By Maheshakya Wijewardena, Joel Nothman and Manoj Kumar.
- Added *metrics.pairwise.laplacian_kernel*. By Clyde Fare.
- `covariance.GraphLasso` allows separate control of the convergence criterion for the Elastic-Net subproblem via the `enet_tol` parameter.
- Improved verbosity in *decomposition.DictionaryLearning*.
- *ensemble.RandomForestClassifier* and *ensemble.RandomForestRegressor* no longer explicitly store the samples used in bagging, resulting in a much reduced memory footprint for storing random forest models.
- Added `positive` option to *linear_model.Lars* and *linear_model.lars_path* to force coefficients to be positive. (#5131)
- Added the `X_norm_squared` parameter to *metrics.pairwise.euclidean_distances* to provide precomputed squared norms for `X`.
- Added the `fit_predict` method to *pipeline.Pipeline*.
- Added the `preprocessing.min_max_scale` function.

Bug fixes

- Fixed non-determinism in *dummy.DummyClassifier* with sparse multi-label output. By Andreas Müller.
- Fixed the output shape of *linear_model.RANSACRegressor* to `(n_samples, )`. By Andreas Müller.
- Fixed bug in `decomposition.DictLearning` when `n_jobs < 0`. By Andreas Müller.
- Fixed bug where `grid_search.RandomizedSearchCV` could consume a lot of memory for large discrete grids. By Joel Nothman.
- Fixed bug in *linear_model.LogisticRegressionCV* where `penalty` was ignored in the final fit. By Manoj Kumar.
- Fixed bug in `ensemble.forest.ForestClassifier` while computing oob_score and X is a sparse.csc_matrix. By Ankur Ankan.
- All regressors now consistently handle and warn when given `y` that is of shape `(n_samples, 1)`. By Andreas Müller and Henry Lin. (#5431)
- Fix in *cluster.KMeans* cluster reassignment for sparse input by Lars Buitinck.
- Fixed a bug in `lda.LDA` that could cause asymmetric covariance matrices when using shrinkage. By Martin Billinger.
- Fixed `cross_validation.cross_val_predict` for estimators with sparse predictions. By Buddha Prakash.

- Fixed the `predict_proba` method of *linear_model.LogisticRegression* to use soft-max instead of one-vs-rest normalization. By Manoj Kumar. (#5182)
- Fixed the `partial_fit` method of *linear_model.SGDClassifier* when called with `average=True`. By Andrew Lamb. (#5282)
- Dataset fetchers use different filenames under Python 2 and Python 3 to avoid pickling compatibility issues. By Olivier Grisel. (#5355)
- Fixed a bug in *naive_bayes.GaussianNB* which caused classification results to depend on scale. By Jake Vanderplas.
- Fixed temporarily *linear_model.Ridge*, which was incorrect when fitting the intercept in the case of sparse data. The fix automatically changes the solver to 'sag' in this case. #5360 by Tom Dupre la Tour.
- Fixed a performance bug in `decomposition.RandomizedPCA` on data with a large number of features and fewer samples. (#4478) By Andreas Müller, Loic Esteve and Giorgio Patrini.
- Fixed bug in `cross_decomposition.PLS` that yielded unstable and platform dependent output, and failed on `fit_transform`. By Arthur Mensch.
- Fixes to the `Bunch` class used to store datasets.
- Fixed `ensemble.plot_partial_dependence` ignoring the `percentiles` parameter.
- Providing a `set` as vocabulary in `CountVectorizer` no longer leads to inconsistent results when pickling.
- Fixed the conditions on when a precomputed Gram matrix needs to be recomputed in *linear_model.LinearRegression*, *linear_model.OrthogonalMatchingPursuit*, *linear_model.Lasso* and *linear_model.ElasticNet*.
- Fixed inconsistent memory layout in the coordinate descent solver that affected `linear_model.DictionaryLearning` and `covariance.GraphLasso`. (#5337) By Olivier Grisel.
- *manifold.LocallyLinearEmbedding* no longer ignores the `reg` parameter.
- Nearest Neighbor estimators with custom distance metrics can now be pickled. (#4362)
- Fixed a bug in *pipeline.FeatureUnion* where `transformer_weights` were not properly handled when performing grid-searches.
- Fixed a bug in *linear_model.LogisticRegression* and *linear_model.LogisticRegressionCV* when using `class_weight='balanced'` or `class_weight='auto'`. By Tom Dupre la Tour.
- Fixed bug #5495 when doing OVR(SVC(decision_function_shape="ovr")). Fixed by Elvis Dohmatob.

API changes summary

- Attribute `data_min`, `data_max` and `data_range` in *preprocessing.MinMaxScaler* are deprecated and won't be available from 0.19. Instead, the class now exposes `data_min_`, `data_max_` and `data_range_`. By Giorgio Patrini.
- All Scaler classes now have an `scale_` attribute, the feature-wise rescaling applied by their `transform` methods. The old attribute `std_` in *preprocessing.StandardScaler* is deprecated and superseded by `scale_`; it won't be available in 0.19. By Giorgio Patrini.
- `svm.SVC`` and *svm.NuSVC* now have an `decision_function_shape` parameter to make their decision function of shape `(n_samples, n_classes)` by setting `decision_function_shape='ovr'`. This will be the default behavior starting in 0.19. By Andreas Müller.

- Passing 1D data arrays as input to estimators is now deprecated as it caused confusion in how the array elements should be interpreted as features or as samples. All data arrays are now expected to be explicitly shaped `(n_samples, n_features)`. By Vighnesh Birodkar.
- `lda.LDA` and `qda.QDA` have been moved to *discriminant_analysis.LinearDiscriminantAnalysis* and *discriminant_analysis.QuadraticDiscriminantAnalysis*.
- The `store_covariance` and `tol` parameters have been moved from the fit method to the constructor in *discriminant_analysis.LinearDiscriminantAnalysis* and the `store_covariances` and `tol` parameters have been moved from the fit method to the constructor in *discriminant_analysis.QuadraticDiscriminantAnalysis*.
- Models inheriting from `_LearntSelectorMixin` will no longer support the transform methods. (i.e, RandomForests, GradientBoosting, LogisticRegression, DecisionTrees, SVMs and SGD related models). Wrap these models around the metatransfomer *feature_selection.SelectFromModel* to remove features (according to `coefs_` or `feature_importances_`) which are below a certain threshold value instead.
- *cluster.KMeans* re-runs cluster-assignments in case of non-convergence, to ensure consistency of `predict(X)` and `labels_`. By Vighnesh Birodkar.
- Classifier and Regressor models are now tagged as such using the `_estimator_type` attribute.
- Cross-validation iterators always provide indices into training and test set, not boolean masks.
- The `decision_function` on all regressors was deprecated and will be removed in 0.19. Use `predict` instead.
- `datasets.load_lfw_pairs` is deprecated and will be removed in 0.19. Use *datasets.fetch_lfw_pairs* instead.
- The deprecated `hmm` module was removed.
- The deprecated `Bootstrap` cross-validation iterator was removed.
- The deprecated `Ward` and `WardAgglomerative` classes have been removed. Use `clustering.AgglomerativeClustering` instead.
- `cross_validation.check_cv` is now a public function.
- The property `residues_` of *linear_model.LinearRegression* is deprecated and will be removed in 0.19.
- The deprecated `n_jobs` parameter of *linear_model.LinearRegression* has been moved to the constructor.
- Removed deprecated `class_weight` parameter from *linear_model.SGDClassifier*'s `fit` method. Use the construction parameter instead.
- The deprecated support for the sequence of sequences (or list of lists) multilabel format was removed. To convert to and from the supported binary indicator matrix format, use *MultiLabelBinarizer*.
- The behavior of calling the `inverse_transform` method of `Pipeline.pipeline` will change in 0.19. It will no longer reshape one-dimensional input to two-dimensional input.
- The deprecated attributes `indicator_matrix_`, `multilabel_` and `classes_` of *preprocessing.LabelBinarizer* were removed.
- Using `gamma=0` in *svm.SVC* and *svm.SVR* to automatically set the gamma to `1. / n_features` is deprecated and will be removed in 0.19. Use `gamma="auto"` instead.

Code Contributors

Aaron Schumacher, Adithya Ganesh, akitty, Alexandre Gramfort, Alexey Grigorev, Ali Baharev, Allen Riddell, Ando Saabas, Andreas Mueller, Andrew Lamb, Anish Shah, Ankur Ankan, Anthony Erlinger, Ari Rouvinen, Arnaud Joly, Arnaud Rachez, Arthur Mensch, banilo, Barmaley.exe, benjaminirving, Boyuan Deng, Brett Naul, Brian McFee, Buddha Prakash, Chi Zhang, Chih-Wei Chang, Christof Angermueller, Christoph Gohlke, Christophe Bourguignat, Christopher Erick Moody, Chyi-Kwei Yau, Cindy Sridharan, CJ Carey, Clyde-fare, Cory Lorenz, Dan Blanchard, Daniel Galvez, Daniel Kronovet, Danny Sullivan, Data1010, David, David D Lowe, David Dotson, djipey, Dmitry Spikhalskiy, Donne Martin, Dougal J. Sutherland, Dougal Sutherland, edson duarte, Eduardo Caro, Eric Larson, Eric Martin, Erich Schubert, Fernando Carrillo, Frank C. Eckert, Frank Zalkow, Gael Varoquaux, Ganiev Ibraim, Gilles Louppe, Giorgio Patrini, giorgiop, Graham Clenaghan, Gryllos Prokopis, gwulfs, Henry Lin, Hsuan-Tien Lin, Immanuel Bayer, Ishank Gulati, Jack Martin, Jacob Schreiber, Jaidev Deshpande, Jake Vanderplas, Jan Hendrik Metzen, Jean Kossaifi, Jeffrey04, Jeremy, jfraj, Jiali Mei, Joe Jevnik, Joel Nothman, John Kirkham, John Wittenauer, Joseph, Joshua Loyal, Jungkook Park, KamalakerDadi, Kashif Rasul, Keith Goodman, Kian Ho, Konstantin Shmelkov, Kyler Brown, Lars Buitinck, Lilian Besson, Loic Esteve, Louis Tiao, maheshakya, Maheshakya Wijewardena, Manoj Kumar, MarkTab marktab.net, Martin Ku, Martin Spacek, MartinBpr, martinosorb, MaryanMorel, Masafumi Oyamada, Mathieu Blondel, Matt Krump, Matti Lyra, Maxim Kolganov, mbillinger, mhg, Michael Heilman, Michael Patterson, Miroslav Batchkarov, Nelle Varoquaux, Nicolas, Nikolay Mayorov, Olivier Grisel, Omer Katz, Óscar Nájera, Pauli Virtanen, Peter Fischer, Peter Prettenhofer, Phil Roth, pianomania, Preston Parry, Raghav RV, Rob Zinkov, Robert Layton, Rohan Ramanath, Saket Choudhary, Sam Zhang, santi, saurabh.bansod, scls19fr, Sebastian Raschka, Sebastian Saeger, Shivan Sornarajah, SimonPL, sinhrks, Skipper Seabold, Sonny Hu, sseg, Stephen Hoover, Steven De Gryze, Steven Seguin, Theodore Vasiloudis, Thomas Unterthiner, Tiago Freitas Pereira, Tian Wang, Tim Head, Timothy Hopper, tokoroten, Tom Dupré la Tour, Trevor Stephens, Valentin Stolbunov, Vighnesh Birodkar, Vinayak Mehta, Vincent, Vincent Michel, vstolbunov, wangz10, Wei Xue, Yucheng Low, Yury Zhauniarovich, Zac Stewart, zhai_pro, Zichen Wang

1.7.21 Version 0.16.1

April 14, 2015

Changelog

Bug fixes

- Allow input data larger than `block_size` in *covariance.LedoitWolf* by Andreas Müller.
- Fix a bug in *isotonic.IsotonicRegression* deduplication that caused unstable result in *calibration.CalibratedClassifierCV* by Jan Hendrik Metzen.
- Fix sorting of labels in func:`preprocessing.label_binarize` by Michael Heilman.
- Fix several stability and convergence issues in *cross_decomposition.CCA* and *cross_decomposition.PLSCanonical* by Andreas Müller
- Fix a bug in *cluster.KMeans* when `precompute_distances=False` on fortran-ordered data.
- Fix a speed regression in *ensemble.RandomForestClassifier*'s `predict` and `predict_proba` by Andreas Müller.
- Fix a regression where `utils.shuffle` converted lists and dataframes to arrays, by Olivier Grisel

1.7.22 Version 0.16

March 26, 2015

Highlights

- Speed improvements (notably in *cluster.DBSCAN*), reduced memory requirements, bug-fixes and better default settings.
- Multinomial Logistic regression and a path algorithm in *linear_model.LogisticRegressionCV*.
- Out-of core learning of PCA via *decomposition.IncrementalPCA*.
- Probability callibration of classifiers using *calibration.CalibratedClassifierCV*.
- *cluster.Birch* clustering method for large-scale datasets.
- Scalable approximate nearest neighbors search with Locality-sensitive hashing forests in `neighbors.LSHForest`.
- Improved error messages and better validation when using malformed input data.
- More robust integration with pandas dataframes.

Changelog

New features

- The new `neighbors.LSHForest` implements locality-sensitive hashing for approximate nearest neighbors search. By Maheshakya Wijewardena.
- Added *svm.LinearSVR*. This class uses the liblinear implementation of Support Vector Regression which is much faster for large sample sizes than *svm.SVR* with linear kernel. By Fabian Pedregosa and Qiang Luo.
- Incremental fit for *GaussianNB*.
- Added `sample_weight` support to *dummy.DummyClassifier* and *dummy.DummyRegressor*. By Arnaud Joly.
- Added the *metrics.label_ranking_average_precision_score* metrics. By Arnaud Joly.
- Add the *metrics.coverage_error* metrics. By Arnaud Joly.
- Added *linear_model.LogisticRegressionCV*. By Manoj Kumar, Fabian Pedregosa, Gael Varoquaux and Alexandre Gramfort.
- Added `warm_start` constructor parameter to make it possible for any trained forest model to grow additional trees incrementally. By Laurent Direr.
- Added `sample_weight` support to *ensemble.GradientBoostingClassifier* and *ensemble.GradientBoostingRegressor*. By Peter Prettenhofer.
- Added *decomposition.IncrementalPCA*, an implementation of the PCA algorithm that supports out-of-core learning with a `partial_fit` method. By Kyle Kastner.
- Averaged SGD for *SGDClassifier* and *SGDRegressor* By Danny Sullivan.
- Added `cross_val_predict` function which computes cross-validated estimates. By Luis Pedro Coelho
- Added *linear_model.TheilSenRegressor*, a robust generalized-median-based estimator. By Florian Wilhelm.
- Added *metrics.median_absolute_error*, a robust metric. By Gael Varoquaux and Florian Wilhelm.
- Add *cluster.Birch*, an online clustering algorithm. By Manoj Kumar, Alexandre Gramfort and Joel Nothman.

- Added shrinkage support to *discriminant_analysis.LinearDiscriminantAnalysis* using two new solvers. By Clemens Brunner and Martin Billinger.
- Added *kernel_ridge.KernelRidge*, an implementation of kernelized ridge regression. By Mathieu Blondel and Jan Hendrik Metzen.
- All solvers in *linear_model.Ridge* now support `sample_weight`. By Mathieu Blondel.
- Added `cross_validation.PredefinedSplit` cross-validation for fixed user-provided cross-validation folds. By Thomas Unterthiner.
- Added *calibration.CalibratedClassifierCV*, an approach for calibrating the predicted probabilities of a classifier. By Alexandre Gramfort, Jan Hendrik Metzen, Mathieu Blondel and Balazs Kegl.

Enhancements

- Add option `return_distance` in `hierarchical.ward_tree` to return distances between nodes for both structured and unstructured versions of the algorithm. By Matteo Visconti di Oleggio Castello. The same option was added in `hierarchical.linkage_tree`. By Manoj Kumar
- Add support for sample weights in scorer objects. Metrics with sample weight support will automatically benefit from it. By Noel Dawe and Vlad Niculae.
- Added `newton-cg` and `lbfgs` solver support in *linear_model.LogisticRegression*. By Manoj Kumar.
- Add `selection="random"` parameter to implement stochastic coordinate descent for *linear_model.Lasso*, *linear_model.ElasticNet* and related. By Manoj Kumar.
- Add `sample_weight` parameter to *metrics.jaccard_similarity_score* and *metrics.log_loss*. By Jatin Shah.
- Support sparse multilabel indicator representation in *preprocessing.LabelBinarizer* and *multiclass.OneVsRestClassifier* (by Hamzeh Alsalhi with thanks to Rohit Sivaprasad), as well as evaluation metrics (by Joel Nothman).
- Add `sample_weight` parameter to `metrics.jaccard_similarity_score`. By `Jatin Shah`.
- Add support for multiclass in `metrics.hinge_loss`. Added `labels=None` as optional parameter. By `Saurabh Jha`.
- Add `sample_weight` parameter to `metrics.hinge_loss`. By `Saurabh Jha`.
- Add `multi_class="multinomial"` option in *linear_model.LogisticRegression* to implement a Logistic Regression solver that minimizes the cross-entropy or multinomial loss instead of the default One-vs-Rest setting. Supports `lbfgs` and `newton-cg` solvers. By Lars Buitinck and Manoj Kumar. Solver option `newton-cg` by Simon Wu.
- `DictVectorizer` can now perform `fit_transform` on an iterable in a single pass, when giving the option `sort=False`. By Dan Blanchard.
- `GridSearchCV` and `RandomizedSearchCV` can now be configured to work with estimators that may fail and raise errors on individual folds. This option is controlled by the `error_score` parameter. This does not affect errors raised on re-fit. By Michal Romaniuk.
- Add `digits` parameter to `metrics.classification_report` to allow report to show different precision of floating point numbers. By Ian Gilmore.
- Add a quantile prediction strategy to the *dummy.DummyRegressor*. By Aaron Staple.
- Add `handle_unknown` option to *preprocessing.OneHotEncoder* to handle unknown categorical features more gracefully during transform. By Manoj Kumar.

- Added support for sparse input data to decision trees and their ensembles. By Fares Hedyati and Arnaud Joly.
- Optimized *cluster.AffinityPropagation* by reducing the number of memory allocations of large temporary data-structures. By Antony Lee.
- Parellization of the computation of feature importances in random forest. By Olivier Grisel and Arnaud Joly.
- Add `n_iter_` attribute to estimators that accept a `max_iter` attribute in their constructor. By Manoj Kumar.
- Added decision function for *multiclass.OneVsOneClassifier* By Raghav RV and Kyle Beauchamp.
- *neighbors.kneighbors_graph* and `radius_neighbors_graph` support non-Euclidean metrics. By Manoj Kumar
- Parameter `connectivity` in *cluster.AgglomerativeClustering* and family now accept callables that return a connectivity matrix. By Manoj Kumar.
- Sparse support for `paired_distances`. By Joel Nothman.
- *cluster.DBSCAN* now supports sparse input and sample weights and has been optimized: the inner loop has been rewritten in Cython and radius neighbors queries are now computed in batch. By Joel Nothman and Lars Buitinck.
- Add `class_weight` parameter to automatically weight samples by class frequency for *ensemble.RandomForestClassifier*, *tree.DecisionTreeClassifier*, *ensemble.ExtraTreesClassifier* and *tree.ExtraTreeClassifier*. By Trevor Stephens.
- `grid_search.RandomizedSearchCV` now does sampling without replacement if all parameters are given as lists. By Andreas Müller.
- Parallelized calculation of `pairwise_distances` is now supported for scipy metrics and custom callables. By Joel Nothman.
- Allow the fitting and scoring of all clustering algorithms in *pipeline.Pipeline*. By Andreas Müller.
- More robust seeding and improved error messages in *cluster.MeanShift* by Andreas Müller.
- Make the stopping criterion for `mixture.GMM`, `mixture.DPGMM` and `mixture.VBGMM` less dependent on the number of samples by thresholding the average log-likelihood change instead of its sum over all samples. By Hervé Bredin.
- The outcome of *manifold.spectral_embedding* was made deterministic by flipping the sign of eigenvectors. By Hasil Sharma.
- Significant performance and memory usage improvements in *preprocessing.PolynomialFeatures*. By Eric Martin.
- Numerical stability improvements for *preprocessing.StandardScaler* and *preprocessing.scale*. By Nicolas Goix
- *svm.SVC* fitted on sparse input now implements `decision_function`. By Rob Zinkov and Andreas Müller.
- `cross_validation.train_test_split` now preserves the input type, instead of converting to numpy arrays.

Documentation improvements

- Added example of using `FeatureUnion` for heterogeneous input. By Matt Terry
- Documentation on scorers was improved, to highlight the handling of loss functions. By Matt Pico.
- A discrepancy between liblinear output and scikit-learn's wrappers is now noted. By Manoj Kumar.

- Improved documentation generation: examples referring to a class or function are now shown in a gallery on the class/function's API reference page. By Joel Nothman.
- More explicit documentation of sample generators and of data transformation. By Joel Nothman.
- *sklearn.neighbors.BallTree* and *sklearn.neighbors.KDTree* used to point to empty pages stating that they are aliases of BinaryTree. This has been fixed to show the correct class docs. By Manoj Kumar.
- Added silhouette plots for analysis of KMeans clustering using *metrics.silhouette_samples* and *metrics.silhouette_score*. See *Selecting the number of clusters with silhouette analysis on KMeans clustering*

Bug fixes

- Metaestimators now support ducktyping for the presence of `decision_function`, `predict_proba` and other methods. This fixes behavior of `grid_search.GridSearchCV`, `grid_search.RandomizedSearchCV`, *pipeline.Pipeline*, *feature_selection.RFE*, *feature_selection.RFECV* when nested. By Joel Nothman
- The `scoring` attribute of grid-search and cross-validation methods is no longer ignored when a `grid_search.GridSearchCV` is given as a base estimator or the base estimator doesn't have predict.
- The function `hierarchical.ward_tree` now returns the children in the same order for both the structured and unstructured versions. By Matteo Visconti di Oleggio Castello.
- *feature_selection.RFECV* now correctly handles cases when `step` is not equal to 1. By Nikolay Mayorov
- The *decomposition.PCA* now undoes whitening in its `inverse_transform`. Also, its `components_` now always have unit length. By Michael Eickenberg.
- Fix incomplete download of the dataset when `datasets.download_20newsgroups` is called. By Manoj Kumar.
- Various fixes to the Gaussian processes subpackage by Vincent Dubourg and Jan Hendrik Metzen.
- Calling `partial_fit` with `class_weight=='auto'` throws an appropriate error message and suggests a work around. By Danny Sullivan.
- *RBFSampler* with `gamma=g` formerly approximated *rbf_kernel* with `gamma=g/2.`; the definition of `gamma` is now consistent, which may substantially change your results if you use a fixed value. (If you cross-validated over `gamma`, it probably doesn't matter too much.) By Dougal Sutherland.
- Pipeline object delegate the `classes_` attribute to the underlying estimator. It allows, for instance, to make bagging of a pipeline object. By Arnaud Joly
- *neighbors.NearestCentroid* now uses the median as the centroid when metric is set to `manhattan`. It was using the mean before. By Manoj Kumar
- Fix numerical stability issues in *linear_model.SGDClassifier* and *linear_model.SGDRegressor* by clipping large gradients and ensuring that weight decay rescaling is always positive (for large l2 regularization and large learning rate values). By Olivier Grisel
- When `compute_full_tree` is set to "auto", the full tree is built when n_clusters is high and is early stopped when n_clusters is low, while the behavior should be vice-versa in *cluster.AgglomerativeClustering* (and friends). This has been fixed By Manoj Kumar
- Fix lazy centering of data in *linear_model.enet_path* and *linear_model.lasso_path*. It was centered around one. It has been changed to be centered around the origin. By Manoj Kumar

- Fix handling of precomputed affinity matrices in *cluster.AgglomerativeClustering* when using connectivity constraints. By Cathy Deng
- Correct `partial_fit` handling of `class_prior` for *sklearn.naive_bayes.MultinomialNB* and *sklearn.naive_bayes.BernoulliNB*. By Trevor Stephens.
- Fixed a crash in *metrics.precision_recall_fscore_support* when using unsorted `labels` in the multi-label setting. By Andreas Müller.
- Avoid skipping the first nearest neighbor in the methods `radius_neighbors`, `kneighbors`, `kneighbors_graph` and `radius_neighbors_graph` in *sklearn.neighbors.NearestNeighbors* and family, when the query data is not the same as fit data. By Manoj Kumar.
- Fix log-density calculation in the `mixture.GMM` with tied covariance. By Will Dawson
- Fixed a scaling error in *feature_selection.SelectFdr* where a factor `n_features` was missing. By Andrew Tulloch
- Fix zero division in *neighbors.KNeighborsRegressor* and related classes when using distance weighting and having identical data points. By Garret-R.
- Fixed round off errors with non positive-definite covariance matrices in GMM. By Alexis Mignon.
- Fixed a error in the computation of conditional probabilities in *naive_bayes.BernoulliNB*. By Hanna Wallach.
- Make the method `radius_neighbors` of *neighbors.NearestNeighbors* return the samples lying on the boundary for `algorithm='brute'`. By Yan Yi.
- Flip sign of `dual_coef_` of *svm.SVC* to make it consistent with the documentation and `decision_function`. By Artem Sobolev.
- Fixed handling of ties in *isotonic.IsotonicRegression*. We now use the weighted average of targets (secondary method). By Andreas Müller and Michael Bommarito.

API changes summary

- `GridSearchCV` and `cross_val_score` and other meta-estimators don't convert pandas DataFrames into arrays any more, allowing DataFrame specific operations in custom estimators.
- `multiclass.fit_ovr`, `multiclass.predict_ovr`, `predict_proba_ovr`, `multiclass.fit_ovo`, `multiclass.predict_ovo`, `multiclass.fit_ecoc` and `multiclass.predict_ecoc` are deprecated. Use the underlying estimators instead.
- Nearest neighbors estimators used to take arbitrary keyword arguments and pass these to their distance metric. This will no longer be supported in scikit-learn 0.18; use the `metric_params` argument instead.
- **`n_jobs` parameter of the fit method shifted to the constructor of the** LinearRegression class.
- The `predict_proba` method of *multiclass.OneVsRestClassifier* now returns two probabilities per sample in the multiclass case; this is consistent with other estimators and with the method's documentation, but previous versions accidentally returned only the positive probability. Fixed by Will Lamond and Lars Buitinck.
- Change default value of precompute in `ElasticNet` and `Lasso` to False. Setting precompute to "auto" was found to be slower when n_samples > n_features since the computation of the Gram matrix is computationally expensive and outweighs the benefit of fitting the Gram for just one alpha. `precompute="auto"` is now deprecated and will be removed in 0.18 By Manoj Kumar.
- Expose `positive` option in *linear_model.enet_path* and *linear_model.enet_path* which constrains coefficients to be positive. By Manoj Kumar.

- Users should now supply an explicit `average` parameter to *sklearn.metrics.f1_score*, *sklearn.metrics.fbeta_score*, *sklearn.metrics.recall_score* and *sklearn.metrics.precision_score* when performing multiclass or multilabel (i.e. not binary) classification. By Joel Nothman.
- `scoring` parameter for cross validation now accepts `'f1_micro'`, `'f1_macro'` or `'f1_weighted'`. `'f1'` is now for binary classification only. Similar changes apply to `'precision'` and `'recall'`. By Joel Nothman.
- The `fit_intercept`, `normalize` and `return_models` parameters in *linear_model.enet_path* and *linear_model.lasso_path* have been removed. They were deprecated since 0.14
- From now onwards, all estimators will uniformly raise `NotFittedError` (`utils.validation.NotFittedError`), when any of the `predict` like methods are called before the model is fit. By Raghav RV.
- Input data validation was refactored for more consistent input validation. The `check_arrays` function was replaced by `check_array` and `check_X_y`. By Andreas Müller.
- Allow `X=None` in the methods `radius_neighbors`, `kneighbors`, `kneighbors_graph` and `radius_neighbors_graph` in *sklearn.neighbors.NearestNeighbors* and family. If set to None, then for every sample this avoids setting the sample itself as the first nearest neighbor. By Manoj Kumar.
- Add parameter `include_self` in *neighbors.kneighbors_graph* and *neighbors.radius_neighbors_graph* which has to be explicitly set by the user. If set to True, then the sample itself is considered as the first nearest neighbor.
- `thresh` parameter is deprecated in favor of new `tol` parameter in `GMM`, `DPGMM` and `VBGMM`. See `Enhancements` section for details. By Hervé Bredin.
- Estimators will treat input with dtype object as numeric when possible. By Andreas Müller
- Estimators now raise `ValueError` consistently when fitted on empty data (less than 1 sample or less than 1 feature for 2D input). By Olivier Grisel.
- The `shuffle` option of *linear_model.SGDClassifier*, *linear_model.SGDRegressor*, *linear_model.Perceptron*, *linear_model.PassiveAggressiveClassifier* and *linear_model.PassiveAggressiveRegressor* now defaults to `True`.
- *cluster.DBSCAN* now uses a deterministic initialization. The `random_state` parameter is deprecated. By Erich Schubert.

Code Contributors

A. Flaxman, Aaron Schumacher, Aaron Staple, abhishek thakur, Akshay, akshayah3, Aldrian Obaja, Alexander Fabisch, Alexandre Gramfort, Alexis Mignon, Anders Aagaard, Andreas Mueller, Andreas van Cranenburgh, Andrew Tulloch, Andrew Walker, Antony Lee, Arnaud Joly, banilo, Barmaley.exe, Ben Davies, Benedikt Koehler, bhsu, Boris Feld, Borja Ayerdi, Boyuan Deng, Brent Pedersen, Brian Wignall, Brooke Osborn, Calvin Giles, Cathy Deng, Celeo, cgohlke, chebee7i, Christian Stade-Schuldt, Christof Angermueller, Chyi-Kwei Yau, CJ Carey, Clemens Brunner, Daiki Aminaka, Dan Blanchard, danfrankj, Danny Sullivan, David Fletcher, Dmitrijs Milajevs, Dougal J. Sutherland, Erich Schubert, Fabian Pedregosa, Florian Wilhelm, floydsoft, Félix-Antoine Fortin, Gael Varoquaux, Garrett-R, Gilles Louppe, gpassino, gwulfs, Hampus Bengtsson, Hamzeh Alsalhi, Hanna Wallach, Harry Mavroforakis, Hasil Sharma, Helder, Herve Bredin, Hsiang-Fu Yu, Hugues SALAMIN, Ian Gilmore, Ilambharathi Kanniah, Imran Haque, isms, Jake VanderPlas, Jan Dlabal, Jan Hendrik Metzen, Jatin Shah, Javier López Peña, jdcaballero, Jean Kossaifi, Jeff Hammerbacher, Joel Nothman, Jonathan Helmus, Joseph, Kaicheng Zhang, Kevin Markham, Kyle Beauchamp, Kyle Kastner, Lagacherie Matthieu, Lars Buitinck, Laurent Direr, leepei, Loic Esteve, Luis Pedro Coelho, Lukas Michelbacher, maheshakya, Manoj Kumar, Manuel, Mario Michael Krell, Martin, Martin Billinger, Martin Ku, Mateusz Susik, Mathieu Blondel, Matt Pico, Matt Terry, Matteo Visconti dOC, Matti Lyra, Max Linke, Mehdi Cherti, Michael Bommarito, Michael Eickenberg, Michal Romaniuk, MLG, mr.Shu, Nelle Varoquaux, Nicola Montecchio, Nicolas,

Nikolay Mayorov, Noel Dawe, Okal Billy, Olivier Grisel, Óscar Nájera, Paolo Puggioni, Peter Prettenhofer, Pratap Vardhan, pvnguyen, queqichao, Rafael Carrascosa, Raghav R V, Rahiel Kasim, Randall Mason, Rob Zinkov, Robert Bradshaw, Saket Choudhary, Sam Nicholls, Samuel Charron, Saurabh Jha, sethdandridge, sinhrks, snuderl, Stefan Otte, Stefan van der Walt, Steve Tjoa, swu, Sylvain Zimmer, tejesh95, terrycojones, Thomas Delteil, Thomas Unterthiner, Tomas Kazmar, trevorstephens, tttthomasssss, Tzu-Ming Kuo, ugurcaliskan, ugurthemaster, Vinayak Mehta, Vincent Dubourg, Vjacheslav Murashkin, Vlad Niculae, wadawson, Wei Xue, Will Lamond, Wu Jiang, x0l, Xinfan Meng, Yan Yi, Yu-Chin

1.7.23 Version 0.15.2

September 4, 2014

Bug fixes

- Fixed handling of the `p` parameter of the Minkowski distance that was previously ignored in nearest neighbors models. By Nikolay Mayorov.
- Fixed duplicated alphas in *`linear_model.LassoLars`* with early stopping on 32 bit Python. By Olivier Grisel and Fabian Pedregosa.
- Fixed the build under Windows when scikit-learn is built with MSVC while NumPy is built with MinGW. By Olivier Grisel and Federico Vaggi.
- Fixed an array index overflow bug in the coordinate descent solver. By Gael Varoquaux.
- Better handling of numpy 1.9 deprecation warnings. By Gael Varoquaux.
- Removed unnecessary data copy in *`cluster.KMeans`*. By Gael Varoquaux.
- Explicitly close open files to avoid `ResourceWarnings` under Python 3. By Calvin Giles.
- The `transform` of *`discriminant_analysis.LinearDiscriminantAnalysis`* now projects the input on the most discriminant directions. By Martin Billinger.
- Fixed potential overflow in `_tree.safe_realloc` by Lars Buitinck.
- Performance optimization in *`isotonic.IsotonicRegression`*. By Robert Bradshaw.
- `nose` is non-longer a runtime dependency to import `sklearn`, only for running the tests. By Joel Nothman.
- Many documentation and website fixes by Joel Nothman, Lars Buitinck Matt Pico, and others.

1.7.24 Version 0.15.1

August 1, 2014

Bug fixes

- Made `cross_validation.cross_val_score` use `cross_validation.KFold` instead of `cross_validation.StratifiedKFold` on multi-output classification problems. By Nikolay Mayorov.
- Support unseen labels *`preprocessing.LabelBinarizer`* to restore the default behavior of 0.14.1 for backward compatibility. By Hamzeh Alsalhi.
- Fixed the *`cluster.KMeans`* stopping criterion that prevented early convergence detection. By Edward Raff and Gael Varoquaux.

- Fixed the behavior of *multiclass.OneVsOneClassifier*. in case of ties at the per-class vote level by computing the correct per-class sum of prediction scores. By Andreas Müller.
- Made `cross_validation.cross_val_score` and `grid_search.GridSearchCV` accept Python lists as input data. This is especially useful for cross-validation and model selection of text processing pipelines. By Andreas Müller.
- Fixed data input checks of most estimators to accept input data that implements the NumPy `__array__` protocol. This is the case for for `pandas.Series` and `pandas.DataFrame` in recent versions of pandas. By Gael Varoquaux.
- Fixed a regression for *linear_model.SGDClassifier* with `class_weight="auto"` on data with non-contiguous labels. By Olivier Grisel.

1.7.25 Version 0.15

July 15, 2014

Highlights

- Many speed and memory improvements all across the code
- Huge speed and memory improvements to random forests (and extra trees) that also benefit better from parallel computing.
- Incremental fit to *BernoulliRBM*
- Added *cluster.AgglomerativeClustering* for hierarchical agglomerative clustering with average linkage, complete linkage and ward strategies.
- Added *linear_model.RANSACRegressor* for robust regression models.
- Added dimensionality reduction with *manifold.TSNE* which can be used to visualize high-dimensional data.

Changelog

New features

- Added *ensemble.BaggingClassifier* and *ensemble.BaggingRegressor* meta-estimators for ensembling any kind of base estimator. See the *Bagging* section of the user guide for details and examples. By Gilles Louppe.
- New unsupervised feature selection algorithm *feature_selection.VarianceThreshold*, by Lars Buitinck.
- Added *linear_model.RANSACRegressor* meta-estimator for the robust fitting of regression models. By Johannes Schönberger.
- Added *cluster.AgglomerativeClustering* for hierarchical agglomerative clustering with average linkage, complete linkage and ward strategies, by Nelle Varoquaux and Gael Varoquaux.
- Shorthand constructors *pipeline.make_pipeline* and *pipeline.make_union* were added by Lars Buitinck.
- Shuffle option for `cross_validation.StratifiedKFold`. By Jeffrey Blackburne.
- Incremental learning (`partial_fit`) for Gaussian Naive Bayes by Imran Haque.
- Added `partial_fit` to *BernoulliRBM* By Danny Sullivan.

- Added `learning_curve` utility to chart performance with respect to training size. See *Plotting Learning Curves*. By Alexander Fabisch.
- Add positive option in *LassoCV* and *ElasticNetCV*. By Brian Wignall and Alexandre Gramfort.
- Added *linear_model.MultiTaskElasticNetCV* and *linear_model.MultiTaskLassoCV*. By Manoj Kumar.
- Added *manifold.TSNE*. By Alexander Fabisch.

Enhancements

- Add sparse input support to *ensemble.AdaBoostClassifier* and *ensemble.AdaBoostRegressor* meta-estimators. By Hamzeh Alsalhi.
- Memory improvements of decision trees, by Arnaud Joly.
- Decision trees can now be built in best-first manner by using `max_leaf_nodes` as the stopping criteria. Refactored the tree code to use either a stack or a priority queue for tree building. By Peter Prettenhofer and Gilles Louppe.
- Decision trees can now be fitted on fortran- and c-style arrays, and non-continuous arrays without the need to make a copy. If the input array has a different dtype than `np.float32`, a fortran- style copy will be made since fortran-style memory layout has speed advantages. By Peter Prettenhofer and Gilles Louppe.
- Speed improvement of regression trees by optimizing the the computation of the mean square error criterion. This lead to speed improvement of the tree, forest and gradient boosting tree modules. By Arnaud Joly
- The `img_to_graph` and `grid_tograph` functions in *sklearn.feature_extraction.image* now return `np.ndarray` instead of `np.matrix` when `return_as=np.ndarray`. See the Notes section for more information on compatibility.
- Changed the internal storage of decision trees to use a struct array. This fixed some small bugs, while improving code and providing a small speed gain. By Joel Nothman.
- Reduce memory usage and overhead when fitting and predicting with forests of randomized trees in parallel with `n_jobs != 1` by leveraging new threading backend of joblib 0.8 and releasing the GIL in the tree fitting Cython code. By Olivier Grisel and Gilles Louppe.
- Speed improvement of the `sklearn.ensemble.gradient_boosting` module. By Gilles Louppe and Peter Prettenhofer.
- Various enhancements to the `sklearn.ensemble.gradient_boosting` module: a `warm_start` argument to fit additional trees, a `max_leaf_nodes` argument to fit GBM style trees, a `monitor` fit argument to inspect the estimator during training, and refactoring of the verbose code. By Peter Prettenhofer.
- Faster `sklearn.ensemble.ExtraTrees` by caching feature values. By Arnaud Joly.
- Faster depth-based tree building algorithm such as decision tree, random forest, extra trees or gradient tree boosting (with depth based growing strategy) by avoiding trying to split on found constant features in the sample subset. By Arnaud Joly.
- Add `min_weight_fraction_leaf` pre-pruning parameter to tree-based methods: the minimum weighted fraction of the input samples required to be at a leaf node. By Noel Dawe.
- Added *metrics.pairwise_distances_argmin_min*, by Philippe Gervais.
- Added predict method to *cluster.AffinityPropagation* and *cluster.MeanShift*, by Mathieu Blondel.
- Vector and matrix multiplications have been optimised throughout the library by Denis Engemann, and Alexandre Gramfort. In particular, they should take less memory with older NumPy versions (prior to 1.7.2).

- Precision-recall and ROC examples now use train_test_split, and have more explanation of why these metrics are useful. By Kyle Kastner
- The training algorithm for *decomposition.NMF* is faster for sparse matrices and has much lower memory complexity, meaning it will scale up gracefully to large datasets. By Lars Buitinck.
- Added svd_method option with default value to "randomized" to *decomposition.FactorAnalysis* to save memory and significantly speedup computation by Denis Engemann, and Alexandre Gramfort.
- Changed `cross_validation.StratifiedKFold` to try and preserve as much of the original ordering of samples as possible so as not to hide overfitting on datasets with a non-negligible level of samples dependency. By Daniel Nouri and Olivier Grisel.
- Add multi-output support to `gaussian_process.GaussianProcess` by John Novak.
- Support for precomputed distance matrices in nearest neighbor estimators by Robert Layton and Joel Nothman.
- Norm computations optimized for NumPy 1.6 and later versions by Lars Buitinck. In particular, the k-means algorithm no longer needs a temporary data structure the size of its input.
- *dummy.DummyClassifier* can now be used to predict a constant output value. By Manoj Kumar.
- *dummy.DummyRegressor* has now a strategy parameter which allows to predict the mean, the median of the training set or a constant output value. By Maheshakya Wijewardena.
- Multi-label classification output in multilabel indicator format is now supported by *metrics.roc_auc_score* and *metrics.average_precision_score* by Arnaud Joly.
- Significant performance improvements (more than 100x speedup for large problems) in *isotonic.IsotonicRegression* by Andrew Tulloch.
- Speed and memory usage improvements to the SGD algorithm for linear models: it now uses threads, not separate processes, when `n_jobs>1`. By Lars Buitinck.
- Grid search and cross validation allow NaNs in the input arrays so that preprocessors such as `preprocessing.Imputer` can be trained within the cross validation loop, avoiding potentially skewed results.
- Ridge regression can now deal with sample weights in feature space (only sample space until then). By Michael Eickenberg. Both solutions are provided by the Cholesky solver.
- Several classification and regression metrics now support weighted samples with the new `sample_weight` argument: *metrics.accuracy_score*, *metrics.zero_one_loss*, *metrics.precision_score*, *metrics.average_precision_score*, *metrics.f1_score*, *metrics.fbeta_score*, *metrics.recall_score*, *metrics.roc_auc_score*, *metrics.explained_variance_score*, *metrics.mean_squared_error*, *metrics.mean_absolute_error*, *metrics.r2_score*. By Noel Dawe.
- Speed up of the sample generator *datasets.make_multilabel_classification*. By Joel Nothman.

Documentation improvements

- The *Working With Text Data* tutorial has now been worked in to the main documentation's tutorial section. Includes exercises and skeletons for tutorial presentation. Original tutorial created by several authors including Olivier Grisel, Lars Buitinck and many others. Tutorial integration into the scikit-learn documentation by Jaques Grobler
- Added *Computational Performance* documentation. Discussion and examples of prediction latency / throughput and different factors that have influence over speed. Additional tips for building faster models and choosing a relevant compromise between speed and predictive power. By Eustache Diemert.

Bug fixes

- Fixed bug in *decomposition.MiniBatchDictionaryLearning* : `partial_fit` was not working properly.
- Fixed bug in `linear_model.stochastic_gradient` : `l1_ratio` was used as `(1.0 - l1_ratio)` .
- Fixed bug in *multiclass.OneVsOneClassifier* with string labels
- Fixed a bug in *LassoCV* and *ElasticNetCV*: they would not pre-compute the Gram matrix with `precompute=True` or `precompute="auto"` and `n_samples > n_features`. By Manoj Kumar.
- Fixed incorrect estimation of the degrees of freedom in *feature_selection.f_regression* when variates are not centered. By Virgile Fritsch.
- Fixed a race condition in parallel processing with `pre_dispatch != "all"` (for instance, in `cross_val_score`). By Olivier Grisel.
- Raise error in *cluster.FeatureAgglomeration* and `cluster.WardAgglomeration` when no samples are given, rather than returning meaningless clustering.
- Fixed bug in `gradient_boosting.GradientBoostingRegressor` with `loss='huber'`: `gamma` might have not been initialized.
- Fixed feature importances as computed with a forest of randomized trees when fit with `sample_weight != None` and/or with `bootstrap=True`. By Gilles Louppe.

API changes summary

- `sklearn.hmm` is deprecated. Its removal is planned for the 0.17 release.
- Use of `covariance.EllipticEnvelop` has now been removed after deprecation. Please use *covariance.EllipticEnvelope* instead.
- `cluster.Ward` is deprecated. Use *cluster.AgglomerativeClustering* instead.
- `cluster.WardClustering` is deprecated. Use
- *cluster.AgglomerativeClustering* instead.
- `cross_validation.Bootstrap` is deprecated. `cross_validation.KFold` or `cross_validation.ShuffleSplit` are recommended instead.
- Direct support for the sequence of sequences (or list of lists) multilabel format is deprecated. To convert to and from the supported binary indicator matrix format, use *MultiLabelBinarizer*. By Joel Nothman.
- Add score method to *PCA* following the model of probabilistic PCA and deprecate `ProbabilisticPCA` model whose score implementation is not correct. The computation now also exploits the matrix inversion lemma for faster computation. By Alexandre Gramfort.
- The score method of *FactorAnalysis* now returns the average log-likelihood of the samples. Use score_samples to get log-likelihood of each sample. By Alexandre Gramfort.
- Generating boolean masks (the setting `indices=False`) from cross-validation generators is deprecated. Support for masks will be removed in 0.17. The generators have produced arrays of indices by default since 0.10. By Joel Nothman.
- 1-d arrays containing strings with `dtype=object` (as used in Pandas) are now considered valid classification targets. This fixes a regression from version 0.13 in some classifiers. By Joel Nothman.
- Fix wrong `explained_variance_ratio_` attribute in `RandomizedPCA`. By Alexandre Gramfort.

- Fit alphas for each `l1_ratio` instead of `mean_l1_ratio` in *linear_model.ElasticNetCV* and *linear_model.LassoCV*. This changes the shape of `alphas_` from `(n_alphas,)` to `(n_l1_ratio, n_alphas)` if the `l1_ratio` provided is a 1-D array like object of length greater than one. By Manoj Kumar.
- Fix *linear_model.ElasticNetCV* and *linear_model.LassoCV* when fitting intercept and input data is sparse. The automatic grid of alphas was not computed correctly and the scaling with normalize was wrong. By Manoj Kumar.
- Fix wrong maximal number of features drawn (`max_features`) at each split for decision trees, random forests and gradient tree boosting. Previously, the count for the number of drawn features started only after one non constant features in the split. This bug fix will affect computational and generalization performance of those algorithms in the presence of constant features. To get back previous generalization performance, you should modify the value of `max_features`. By Arnaud Joly.
- Fix wrong maximal number of features drawn (`max_features`) at each split for *ensemble.ExtraTreesClassifier* and *ensemble.ExtraTreesRegressor*. Previously, only non constant features in the split was counted as drawn. Now constant features are counted as drawn. Furthermore at least one feature must be non constant in order to make a valid split. This bug fix will affect computational and generalization performance of extra trees in the presence of constant features. To get back previous generalization performance, you should modify the value of `max_features`. By Arnaud Joly.
- Fix `utils.compute_class_weight` when `class_weight=="auto"`. Previously it was broken for input of non-integer `dtype` and the weighted array that was returned was wrong. By Manoj Kumar.
- Fix `cross_validation.Bootstrap` to return `ValueError` when `n_train + n_test > n`. By Ronald Phlypo.

People

List of contributors for release 0.15 by number of commits.

- 312 Olivier Grisel
- 275 Lars Buitinck
- 221 Gael Varoquaux
- 148 Arnaud Joly
- 134 Johannes Schönberger
- 119 Gilles Louppe
- 113 Joel Nothman
- 111 Alexandre Gramfort
- 95 Jaques Grobler
- 89 Denis Engemann
- 83 Peter Prettenhofer
- 83 Alexander Fabisch
- 62 Mathieu Blondel
- 60 Eustache Diemert
- 60 Nelle Varoquaux
- 49 Michael Bommarito
- 45 Manoj-Kumar-S

- 28 Kyle Kastner
- 26 Andreas Mueller
- 22 Noel Dawe
- 21 Maheshakya Wijewardena
- 21 Brooke Osborn
- 21 Hamzeh Alsalhi
- 21 Jake VanderPlas
- 21 Philippe Gervais
- 19 Bala Subrahmanyam Varanasi
- 12 Ronald Phlypo
- 10 Mikhail Korobov
- 8 Thomas Unterthiner
- 8 Jeffrey Blackburne
- 8 eltermann
- 8 bwignall
- 7 Ankit Agrawal
- 7 CJ Carey
- 6 Daniel Nouri
- 6 Chen Liu
- 6 Michael Eickenberg
- 6 ugurthemaster
- 5 Aaron Schumacher
- 5 Baptiste Lagarde
- 5 Rajat Khanduja
- 5 Robert McGibbon
- 5 Sergio Pascual
- 4 Alexis Metaireau
- 4 Ignacio Rossi
- 4 Virgile Fritsch
- 4 Sebastian Säger
- 4 Ilambharathi Kanniah
- 4 sdenton4
- 4 Robert Layton
- 4 Alyssa
- 4 Amos Waterland
- 3 Andrew Tulloch

- 3 murad
- 3 Steven Maude
- 3 Karol Pysniak
- 3 Jacques Kvam
- 3 cgohlke
- 3 cjlin
- 3 Michael Becker
- 3 hamzeh
- 3 Eric Jacobsen
- 3 john collins
- 3 kaushik94
- 3 Erwin Marsi
- 2 csytracy
- 2 LK
- 2 Vlad Niculae
- 2 Laurent Direr
- 2 Erik Shilts
- 2 Raul Garreta
- 2 Yoshiki Vázquez Baeza
- 2 Yung Siang Liau
- 2 abhishek thakur
- 2 James Yu
- 2 Rohit Sivaprasad
- 2 Roland Szabo
- 2 amormachine
- 2 Alexis Mignon
- 2 Oscar Carlsson
- 2 Nantas Nardelli
- 2 jess010
- 2 kowalski87
- 2 Andrew Clegg
- 2 Federico Vaggi
- 2 Simon Frid
- 2 Félix-Antoine Fortin
- 1 Ralf Gommers
- 1 t-aft

- 1 Ronan Amicel
- 1 Rupesh Kumar Srivastava
- 1 Ryan Wang
- 1 Samuel Charron
- 1 Samuel St-Jean
- 1 Fabian Pedregosa
- 1 Skipper Seabold
- 1 Stefan Walk
- 1 Stefan van der Walt
- 1 Stephan Hoyer
- 1 Allen Riddell
- 1 Valentin Haenel
- 1 Vijay Ramesh
- 1 Will Myers
- 1 Yaroslav Halchenko
- 1 Yoni Ben-Meshulam
- 1 Yury V. Zaytsev
- 1 adrinjalali
- 1 ai8rahim
- 1 alemagnani
- 1 alex
- 1 benjamin wilson
- 1 chalmerlowe
- 1 dzikie drożdże
- 1 jamestwebber
- 1 matrixorz
- 1 popo
- 1 samuela
- 1 François Boulogne
- 1 Alexander Measure
- 1 Ethan White
- 1 Guilherme Trein
- 1 Hendrik Heuer
- 1 IvicaJovic
- 1 Jan Hendrik Metzen
- 1 Jean Michel Rouly

- 1 Eduardo Ariño de la Rubia
- 1 Jelle Zijlstra
- 1 Eddy L O Jansson
- 1 Denis
- 1 John
- 1 John Schmidt
- 1 Jorge Cañardo Alastuey
- 1 Joseph Perla
- 1 Joshua Vredevoogd
- 1 José Ricardo
- 1 Julien Miotte
- 1 Kemal Eren
- 1 Kenta Sato
- 1 David Cournapeau
- 1 Kyle Kelley
- 1 Daniele Medri
- 1 Laurent Luce
- 1 Laurent Pierron
- 1 Luis Pedro Coelho
- 1 DanielWeitzenfeld
- 1 Craig Thompson
- 1 Chyi-Kwei Yau
- 1 Matthew Brett
- 1 Matthias Feurer
- 1 Max Linke
- 1 Chris Filo Gorgolewski
- 1 Charles Earl
- 1 Michael Hanke
- 1 Michele Orrù
- 1 Bryan Lunt
- 1 Brian Kearns
- 1 Paul Butler
- 1 Paweł Mandera
- 1 Peter
- 1 Andrew Ash
- 1 Pietro Zambelli

- 1 staubda

1.7.26 Version 0.14

August 7, 2013

Changelog

- Missing values with sparse and dense matrices can be imputed with the transformer `preprocessing.Imputer` by Nicolas Trésegnie.
- The core implementation of decisions trees has been rewritten from scratch, allowing for faster tree induction and lower memory consumption in all tree-based estimators. By Gilles Louppe.
- Added *ensemble.AdaBoostClassifier* and *ensemble.AdaBoostRegressor*, by Noel Dawe and Gilles Louppe. See the *AdaBoost* section of the user guide for details and examples.
- Added `grid_search.RandomizedSearchCV` and `grid_search.ParameterSampler` for randomized hyperparameter optimization. By Andreas Müller.
- Added *biclustering* algorithms (`sklearn.cluster.bicluster.SpectralCoclustering` and `sklearn.cluster.bicluster.SpectralBiclustering`), data generation methods (*sklearn.datasets.make_biclusters* and *sklearn.datasets.make_checkerboard*), and scoring metrics (*sklearn.metrics.consensus_score*). By Kemal Eren.
- Added *Restricted Boltzmann Machines* (*neural_network.BernoulliRBM*). By Yann Dauphin.
- Python 3 support by Justin Vincent, Lars Buitinck, Subhodeep Moitra and Olivier Grisel. All tests now pass under Python 3.3.
- Ability to pass one penalty (alpha value) per target in *linear_model.Ridge*, by @eickenberg and Mathieu Blondel.
- Fixed `sklearn.linear_model.stochastic_gradient.py` L2 regularization issue (minor practical significance). By Norbert Crombach and Mathieu Blondel .
- Added an interactive version of Andreas Müller's Machine Learning Cheat Sheet (for scikit-learn) to the documentation. See *Choosing the right estimator*. By Jaques Grobler.
- `grid_search.GridSearchCV` and `cross_validation.cross_val_score` now support the use of advanced scoring function such as area under the ROC curve and f-beta scores. See *The scoring parameter: defining model evaluation rules* for details. By Andreas Müller and Lars Buitinck. Passing a function from *sklearn.metrics* as `score_func` is deprecated.
- Multi-label classification output is now supported by *metrics.accuracy_score*, *metrics.zero_one_loss*, *metrics.f1_score*, *metrics.fbeta_score*, *metrics.classification_report*, *metrics.precision_score* and *metrics.recall_score* by Arnaud Joly.
- Two new metrics *metrics.hamming_loss* and *metrics.jaccard_similarity_score* are added with multi-label support by Arnaud Joly.
- Speed and memory usage improvements in *feature_extraction.text.CountVectorizer* and *feature_extraction.text.TfidfVectorizer*, by Jochen Wersdörfer and Roman Sinayev.
- The `min_df` parameter in *feature_extraction.text.CountVectorizer* and *feature_extraction.text.TfidfVectorizer*, which used to be 2, has been reset to 1 to avoid unpleasant surprises (empty vocabularies) for novice users who try it out on tiny document collections. A value of at least 2 is still recommended for practical use.

- *svm.LinearSVC*, *linear_model.SGDClassifier* and *linear_model.SGDRegressor* now have a sparsify method that converts their coef_ into a sparse matrix, meaning stored models trained using these estimators can be made much more compact.
- *linear_model.SGDClassifier* now produces multiclass probability estimates when trained under log loss or modified Huber loss.
- Hyperlinks to documentation in example code on the website by Martin Luessi.
- Fixed bug in *preprocessing.MinMaxScaler* causing incorrect scaling of the features for non-default feature_range settings. By Andreas Müller.
- max_features in *tree.DecisionTreeClassifier*, *tree.DecisionTreeRegressor* and all derived ensemble estimators now supports percentage values. By Gilles Louppe.
- Performance improvements in *isotonic.IsotonicRegression* by Nelle Varoquaux.
- *metrics.accuracy_score* has an option normalize to return the fraction or the number of correctly classified sample by Arnaud Joly.
- Added *metrics.log_loss* that computes log loss, aka cross-entropy loss. By Jochen Wersdörfer and Lars Buitinck.
- A bug that caused *ensemble.AdaBoostClassifier*'s to output incorrect probabilities has been fixed.
- Feature selectors now share a mixin providing consistent transform, inverse_transform and get_support methods. By Joel Nothman.
- A fitted grid_search.GridSearchCV or grid_search.RandomizedSearchCV can now generally be pickled. By Joel Nothman.
- Refactored and vectorized implementation of *metrics.roc_curve* and *metrics.precision_recall_curve*. By Joel Nothman.
- The new estimator *sklearn.decomposition.TruncatedSVD* performs dimensionality reduction using SVD on sparse matrices, and can be used for latent semantic analysis (LSA). By Lars Buitinck.
- Added self-contained example of out-of-core learning on text data *Out-of-core classification of text documents*. By Eustache Diemert.
- The default number of components for sklearn.decomposition.RandomizedPCA is now correctly documented to be n_features. This was the default behavior, so programs using it will continue to work as they did.
- *sklearn.cluster.KMeans* now fits several orders of magnitude faster on sparse data (the speedup depends on the sparsity). By Lars Buitinck.
- Reduce memory footprint of FastICA by Denis Engemann and Alexandre Gramfort.
- Verbose output in sklearn.ensemble.gradient_boosting now uses a column format and prints progress in decreasing frequency. It also shows the remaining time. By Peter Prettenhofer.
- sklearn.ensemble.gradient_boosting provides out-of-bag improvement oob_improvement_ rather than the OOB score for model selection. An example that shows how to use OOB estimates to select the number of trees was added. By Peter Prettenhofer.
- Most metrics now support string labels for multiclass classification by Arnaud Joly and Lars Buitinck.
- New OrthogonalMatchingPursuitCV class by Alexandre Gramfort and Vlad Niculae.
- Fixed a bug in sklearn.covariance.GraphLassoCV: the 'alphas' parameter now works as expected when given a list of values. By Philippe Gervais.

- Fixed an important bug in `sklearn.covariance.GraphLassoCV` that prevented all folds provided by a CV object to be used (only the first 3 were used). When providing a CV object, execution time may thus increase significantly compared to the previous version (bug results are correct now). By Philippe Gervais.
- `cross_validation.cross_val_score` and the `grid_search` module is now tested with multi-output data by Arnaud Joly.
- *`datasets.make_multilabel_classification`* can now return the output in label indicator multilabel format by Arnaud Joly.
- K-nearest neighbors, *`neighbors.KNeighborsRegressor`* and *`neighbors.RadiusNeighborsRegressor`*, and radius neighbors, *`neighbors.RadiusNeighborsRegressor`* and *`neighbors.RadiusNeighborsClassifier`* support multioutput data by Arnaud Joly.
- Random state in LibSVM-based estimators (*`svm.SVC`*, `NuSVC`, `OneClassSVM`, *`svm.SVR`*, *`svm.NuSVR`*) can now be controlled. This is useful to ensure consistency in the probability estimates for the classifiers trained with `probability=True`. By Vlad Niculae.
- Out-of-core learning support for discrete naive Bayes classifiers *`sklearn.naive_bayes.MultinomialNB`* and *`sklearn.naive_bayes.BernoulliNB`* by adding the `partial_fit` method by Olivier Grisel.
- New website design and navigation by Gilles Louppe, Nelle Varoquaux, Vincent Michel and Andreas Müller.
- Improved documentation on *multi-class, multi-label and multi-output classification* by Yannick Schwartz and Arnaud Joly.
- Better input and error handling in the `metrics` module by Arnaud Joly and Joel Nothman.
- Speed optimization of the `hmm` module by Mikhail Korobov
- Significant speed improvements for *`sklearn.cluster.DBSCAN`* by cleverless

API changes summary

- The `auc_score` was renamed `roc_auc_score`.
- Testing scikit-learn with `sklearn.test()` is deprecated. Use `nosetests sklearn` from the command line.
- Feature importances in *`tree.DecisionTreeClassifier`*, *`tree.DecisionTreeRegressor`* and all derived ensemble estimators are now computed on the fly when accessing the `feature_importances_` attribute. Setting `compute_importances=True` is no longer required. By Gilles Louppe.
- *`linear_model.lasso_path`* and *`linear_model.enet_path`* can return its results in the same format as that of *`linear_model.lars_path`*. This is done by setting the `return_models` parameter to `False`. By Jaques Grobler and Alexandre Gramfort
- `grid_search.IterGrid` was renamed to `grid_search.ParameterGrid`.
- Fixed bug in `KFold` causing imperfect class balance in some cases. By Alexandre Gramfort and Tadej Janež.
- *`sklearn.neighbors.BallTree`* has been refactored, and a *`sklearn.neighbors.KDTree`* has been added which shares the same interface. The Ball Tree now works with a wide variety of distance metrics. Both classes have many new methods, including single-tree and dual-tree queries, breadth-first and depth-first searching, and more advanced queries such as kernel density estimation and 2-point correlation functions. By Jake Vanderplas
- Support for scipy.spatial.cKDTree within neighbors queries has been removed, and the functionality replaced with the new `KDTree` class.

- *sklearn.neighbors.KernelDensity* has been added, which performs efficient kernel density estimation with a variety of kernels.
- *sklearn.decomposition.KernelPCA* now always returns output with n_components components, unless the new parameter remove_zero_eig is set to True. This new behavior is consistent with the way kernel PCA was always documented; previously, the removal of components with zero eigenvalues was tacitly performed on all data.
- gcv_mode="auto" no longer tries to perform SVD on a densified sparse matrix in *sklearn.linear_model.RidgeCV*.
- Sparse matrix support in sklearn.decomposition.RandomizedPCA is now deprecated in favor of the new TruncatedSVD.
- cross_validation.KFold and cross_validation.StratifiedKFold now enforce n_folds >= 2 otherwise a ValueError is raised. By Olivier Grisel.
- *datasets.load_files*'s charset and charset_errors parameters were renamed encoding and decode_errors.
- Attribute oob_score_ in *sklearn.ensemble.GradientBoostingRegressor* and *sklearn.ensemble.GradientBoostingClassifier* is deprecated and has been replaced by oob_improvement_ .
- Attributes in OrthogonalMatchingPursuit have been deprecated (copy_X, Gram, ...) and precompute_gram renamed precompute for consistency. See #2224.
- *sklearn.preprocessing.StandardScaler* now converts integer input to float, and raises a warning. Previously it rounded for dense integer input.
- *sklearn.multiclass.OneVsRestClassifier* now has a decision_function method. This will return the distance of each sample from the decision boundary for each class, as long as the underlying estimators implement the decision_function method. By Kyle Kastner.
- Better input validation, warning on unexpected shapes for y.

People

List of contributors for release 0.14 by number of commits.

- 277 Gilles Louppe
- 245 Lars Buitinck
- 187 Andreas Mueller
- 124 Arnaud Joly
- 112 Jaques Grobler
- 109 Gael Varoquaux
- 107 Olivier Grisel
- 102 Noel Dawe
- 99 Kemal Eren
- 79 Joel Nothman
- 75 Jake VanderPlas
- 73 Nelle Varoquaux
- 71 Vlad Niculae

- 65 Peter Prettenhofer
- 64 Alexandre Gramfort
- 54 Mathieu Blondel
- 38 Nicolas Trésegnie
- 35 eustache
- 27 Denis Engemann
- 25 Yann N. Dauphin
- 19 Justin Vincent
- 17 Robert Layton
- 15 Doug Coleman
- 14 Michael Eickenberg
- 13 Robert Marchman
- 11 Fabian Pedregosa
- 11 Philippe Gervais
- 10 Jim Holmström
- 10 Tadej Janež
- 10 syhw
- 9 Mikhail Korobov
- 9 Steven De Gryze
- 8 sergeyf
- 7 Ben Root
- 7 Hrishikesh Huilgolkar
- 6 Kyle Kastner
- 6 Martin Luessi
- 6 Rob Speer
- 5 Federico Vaggi
- 5 Raul Garreta
- 5 Rob Zinkov
- 4 Ken Geis
- 3 A. Flaxman
- 3 Denton Cockburn
- 3 Dougal Sutherland
- 3 Ian Ozsvald
- 3 Johannes Schönberger
- 3 Robert McGibbon
- 3 Roman Sinayev

- 3 Szabo Roland
- 2 Diego Molla
- 2 Imran Haque
- 2 Jochen Wersdörfer
- 2 Sergey Karayev
- 2 Yannick Schwartz
- 2 jamestwebber
- 1 Abhijeet Kolhe
- 1 Alexander Fabisch
- 1 Bastiaan van den Berg
- 1 Benjamin Peterson
- 1 Daniel Velkov
- 1 Fazlul Shahriar
- 1 Felix Brockherde
- 1 Félix-Antoine Fortin
- 1 Harikrishnan S
- 1 Jack Hale
- 1 JakeMick
- 1 James McDermott
- 1 John Benediktsson
- 1 John Zwinck
- 1 Joshua Vredevoogd
- 1 Justin Pati
- 1 Kevin Hughes
- 1 Kyle Kelley
- 1 Matthias Ekman
- 1 Miroslav Shubernetskiy
- 1 Naoki Orii
- 1 Norbert Crombach
- 1 Rafael Cunha de Almeida
- 1 Rolando Espinoza La fuente
- 1 Seamus Abshere
- 1 Sergey Feldman
- 1 Sergio Medina
- 1 Stefano Lattarini
- 1 Steve Koch

- 1 Sturla Molden
- 1 Thomas Jarosch
- 1 Yaroslav Halchenko

1.7.27 Version 0.13.1

February 23, 2013

The 0.13.1 release only fixes some bugs and does not add any new functionality.

Changelog

- Fixed a testing error caused by the function `cross_validation.train_test_split` being interpreted as a test by Yaroslav Halchenko.
- Fixed a bug in the reassignment of small clusters in the *`cluster.MiniBatchKMeans`* by Gael Varoquaux.
- Fixed default value of `gamma` in *`decomposition.KernelPCA`* by Lars Buitinck.
- Updated joblib to `0.7.0d` by Gael Varoquaux.
- Fixed scaling of the deviance in *`ensemble.GradientBoostingClassifier`* by Peter Prettenhofer.
- Better tie-breaking in *`multiclass.OneVsOneClassifier`* by Andreas Müller.
- Other small improvements to tests and documentation.

People

List of contributors for release 0.13.1 by number of commits.

- 16 Lars Buitinck
- 12 Andreas Müller
- 8 Gael Varoquaux
- 5 Robert Marchman
- 3 Peter Prettenhofer
- 2 Hrishikesh Huilgolkar
- 1 Bastiaan van den Berg
- 1 Diego Molla
- 1 Gilles Louppe
- 1 Mathieu Blondel
- 1 Nelle Varoquaux
- 1 Rafael Cunha de Almeida
- 1 Rolando Espinoza La fuente
- 1 Vlad Niculae
- 1 Yaroslav Halchenko

1.7.28 Version 0.13

January 21, 2013

New Estimator Classes

- `dummy.DummyClassifier` and `dummy.DummyRegressor`, two data-independent predictors by Mathieu Blondel. Useful to sanity-check your estimators. See *Dummy estimators* in the user guide. Multioutput support added by Arnaud Joly.
- `decomposition.FactorAnalysis`, a transformer implementing the classical factor analysis, by Christian Osendorfer and Alexandre Gramfort. See *Factor Analysis* in the user guide.
- `feature_extraction.FeatureHasher`, a transformer implementing the "hashing trick" for fast, low-memory feature extraction from string fields by Lars Buitinck and `feature_extraction.text.HashingVectorizer` for text documents by Olivier Grisel See *Feature hashing* and *Vectorizing a large text corpus with the hashing trick* for the documentation and sample usage.
- `pipeline.FeatureUnion`, a transformer that concatenates results of several other transformers by Andreas Müller. See *FeatureUnion: composite feature spaces* in the user guide.
- `random_projection.GaussianRandomProjection`, `random_projection.SparseRandomProjection` and the function `random_projection.johnson_lindenstrauss_min_dim`. The first two are transformers implementing Gaussian and sparse random projection matrix by Olivier Grisel and Arnaud Joly. See *Random Projection* in the user guide.
- `kernel_approximation.Nystroem`, a transformer for approximating arbitrary kernels by Andreas Müller. See *Nystroem Method for Kernel Approximation* in the user guide.
- `preprocessing.OneHotEncoder`, a transformer that computes binary encodings of categorical features by Andreas Müller. See *Encoding categorical features* in the user guide.
- `linear_model.PassiveAggressiveClassifier` and `linear_model.PassiveAggressiveRegressor`, predictors implementing an efficient stochastic optimization for linear models by Rob Zinkov and Mathieu Blondel. See *Passive Aggressive Algorithms* in the user guide.
- `ensemble.RandomTreesEmbedding`, a transformer for creating high-dimensional sparse representations using ensembles of totally random trees by Andreas Müller. See *Totally Random Trees Embedding* in the user guide.
- `manifold.SpectralEmbedding` and function `manifold.spectral_embedding`, implementing the "laplacian eigenmaps" transformation for non-linear dimensionality reduction by Wei Li. See *Spectral Embedding* in the user guide.
- `isotonic.IsotonicRegression` by Fabian Pedregosa, Alexandre Gramfort and Nelle Varoquaux,

Changelog

- `metrics.zero_one_loss` (formerly `metrics.zero_one`) now has option for normalized output that reports the fraction of misclassifications, rather than the raw number of misclassifications. By Kyle Beauchamp.
- `tree.DecisionTreeClassifier` and all derived ensemble models now support sample weighting, by Noel Dawe and Gilles Louppe.
- Speedup improvement when using bootstrap samples in forests of randomized trees, by Peter Prettenhofer and Gilles Louppe.
- Partial dependence plots for *Gradient Tree Boosting* in `ensemble.partial_dependence.partial_dependence` by Peter Prettenhofer. See *Partial Dependence Plots* for an example.

- The table of contents on the website has now been made expandable by Jaques Grobler.
- *feature_selection.SelectPercentile* now breaks ties deterministically instead of returning all equally ranked features.
- *feature_selection.SelectKBest* and *feature_selection.SelectPercentile* are more numerically stable since they use scores, rather than p-values, to rank results. This means that they might sometimes select different features than they did previously.
- Ridge regression and ridge classification fitting with `sparse_cg` solver no longer has quadratic memory complexity, by Lars Buitinck and Fabian Pedregosa.
- Ridge regression and ridge classification now support a new fast solver called `lsqr`, by Mathieu Blondel.
- Speed up of *metrics.precision_recall_curve* by Conrad Lee.
- Added support for reading/writing svmlight files with pairwise preference attribute (qid in svmlight file format) in *datasets.dump_svmlight_file* and *datasets.load_svmlight_file* by Fabian Pedregosa.
- Faster and more robust *metrics.confusion_matrix* and *Clustering performance evaluation* by Wei Li.
- `cross_validation.cross_val_score` now works with precomputed kernels and affinity matrices, by Andreas Müller.
- LARS algorithm made more numerically stable with heuristics to drop regressors too correlated as well as to stop the path when numerical noise becomes predominant, by Gael Varoquaux.
- Faster implementation of *metrics.precision_recall_curve* by Conrad Lee.
- New kernel `metrics.chi2_kernel` by Andreas Müller, often used in computer vision applications.
- Fix of longstanding bug in *naive_bayes.BernoulliNB* fixed by Shaun Jackman.
- Implemented `predict_proba` in *multiclass.OneVsRestClassifier*, by Andrew Winterman.
- Improve consistency in gradient boosting: estimators *ensemble.GradientBoostingRegressor* and *ensemble.GradientBoostingClassifier* use the estimator *tree.DecisionTreeRegressor* instead of the `tree._tree.Tree` data structure by Arnaud Joly.
- Fixed a floating point exception in the *decision trees* module, by Seberg.
- Fix *metrics.roc_curve* fails when y_true has only one class by Wei Li.
- Add the *metrics.mean_absolute_error* function which computes the mean absolute error. The *metrics.mean_squared_error*, *metrics.mean_absolute_error* and *metrics.r2_score* metrics support multioutput by Arnaud Joly.
- Fixed `class_weight` support in *svm.LinearSVC* and *linear_model.LogisticRegression* by Andreas Müller. The meaning of `class_weight` was reversed as erroneously higher weight meant less positives of a given class in earlier releases.
- Improve narrative documentation and consistency in *sklearn.metrics* for regression and classification metrics by Arnaud Joly.
- Fixed a bug in *sklearn.svm.SVC* when using csr-matrices with unsorted indices by Xinfan Meng and Andreas Müller.
- `MiniBatchKMeans`: Add random reassignment of cluster centers with little observations attached to them, by Gael Varoquaux.

API changes summary

- Renamed all occurrences of `n_atoms` to `n_components` for consistency. This applies to *decomposition.DictionaryLearning*, *decomposition.*

MiniBatchDictionaryLearning, *decomposition.dict_learning*, *decomposition.dict_learning_online*.

- Renamed all occurrences of `max_iters` to `max_iter` for consistency. This applies to *semi_supervised.LabelPropagation* and `semi_supervised.label_propagation.LabelSpreading`.
- Renamed all occurrences of `learn_rate` to `learning_rate` for consistency in `ensemble.BaseGradientBoosting` and *ensemble.GradientBoostingRegressor*.
- The module `sklearn.linear_model.sparse` is gone. Sparse matrix support was already integrated into the "regular" linear models.
- `sklearn.metrics.mean_square_error`, which incorrectly returned the accumulated error, was removed. Use `mean_squared_error` instead.
- Passing `class_weight` parameters to `fit` methods is no longer supported. Pass them to estimator constructors instead.
- GMMs no longer have `decode` and `rvs` methods. Use the `score`, `predict` or `sample` methods instead.
- The `solver` fit option in Ridge regression and classification is now deprecated and will be removed in v0.14. Use the constructor option instead.
- `feature_extraction.text.DictVectorizer` now returns sparse matrices in the CSR format, instead of COO.
- Renamed `k` in `cross_validation.KFold` and `cross_validation.StratifiedKFold` to `n_folds`, renamed `n_bootstraps` to `n_iter` in `cross_validation.Bootstrap`.
- Renamed all occurrences of `n_iterations` to `n_iter` for consistency. This applies to `cross_validation.ShuffleSplit`, `cross_validation.StratifiedShuffleSplit`, `utils.randomized_range_finder` and `utils.randomized_svd`.
- Replaced `rho` in *linear_model.ElasticNet* and *linear_model.SGDClassifier* by `l1_ratio`. The `rho` parameter had different meanings; `l1_ratio` was introduced to avoid confusion. It has the same meaning as previously `rho` in *linear_model.ElasticNet* and `(1-rho)` in *linear_model.SGDClassifier*.
- *linear_model.LassoLars* and *linear_model.Lars* now store a list of paths in the case of multiple targets, rather than an array of paths.
- The attribute `gmm` of `hmm.GMMHMM` was renamed to `gmm_` to adhere more strictly with the API.
- `cluster.spectral_embedding` was moved to *manifold.spectral_embedding*.
- Renamed `eig_tol` in *manifold.spectral_embedding*, *cluster.SpectralClustering* to `eigen_tol`, renamed `mode` to `eigen_solver`.
- Renamed `mode` in *manifold.spectral_embedding* and *cluster.SpectralClustering* to `eigen_solver`.
- `classes_` and `n_classes_` attributes of *tree.DecisionTreeClassifier* and all derived ensemble models are now flat in case of single output problems and nested in case of multi-output problems.
- The `estimators_` attribute of `ensemble.gradient_boosting.GradientBoostingRegressor` and `ensemble.gradient_boosting.GradientBoostingClassifier` is now an array of :class:'tree.DecisionTreeRegressor'.
- Renamed `chunk_size` to `batch_size` in *decomposition.MiniBatchDictionaryLearning* and *decomposition.MiniBatchSparsePCA* for consistency.
- *svm.SVC* and *svm.NuSVC* now provide a `classes_` attribute and support arbitrary dtypes for labels `y`. Also, the dtype returned by `predict` now reflects the dtype of `y` during `fit` (used to be `np.float`).

- Changed default test_size in `cross_validation.train_test_split` to None, added possibility to infer `test_size` from `train_size` in `cross_validation.ShuffleSplit` and `cross_validation.StratifiedShuffleSplit`.
- Renamed function `sklearn.metrics.zero_one` to *`sklearn.metrics.zero_one_loss`*. Be aware that the default behavior in *`sklearn.metrics.zero_one_loss`* is different from `sklearn.metrics.zero_one`: `normalize=False` is changed to `normalize=True`.
- Renamed function `metrics.zero_one_score` to *`metrics.accuracy_score`*.
- *`datasets.make_circles`* now has the same number of inner and outer points.
- In the Naive Bayes classifiers, the `class_prior` parameter was moved from `fit` to `__init__`.

People

List of contributors for release 0.13 by number of commits.

- 364 Andreas Müller
- 143 Arnaud Joly
- 137 Peter Prettenhofer
- 131 Gael Varoquaux
- 117 Mathieu Blondel
- 108 Lars Buitinck
- 106 Wei Li
- 101 Olivier Grisel
- 65 Vlad Niculae
- 54 Gilles Louppe
- 40 Jaques Grobler
- 38 Alexandre Gramfort
- 30 Rob Zinkov
- 19 Aymeric Masurelle
- 18 Andrew Winterman
- 17 Fabian Pedregosa
- 17 Nelle Varoquaux
- 16 Christian Osendorfer
- 14 Daniel Nouri
- 13 Virgile Fritsch
- 13 syhw
- 12 Satrajit Ghosh
- 10 Corey Lynch
- 10 Kyle Beauchamp
- 9 Brian Cheung

- 9 Immanuel Bayer
- 9 mr.Shu
- 8 Conrad Lee
- 8 James Bergstra
- 7 Tadej Janež
- 6 Brian Cajes
- 6 Jake Vanderplas
- 6 Michael
- 6 Noel Dawe
- 6 Tiago Nunes
- 6 cow
- 5 Anze
- 5 Shiqiao Du
- 4 Christian Jauvin
- 4 Jacques Kvam
- 4 Richard T. Guy
- 4 Robert Layton
- 3 Alexandre Abraham
- 3 Doug Coleman
- 3 Scott Dickerson
- 2 ApproximateIdentity
- 2 John Benediktsson
- 2 Mark Veronda
- 2 Matti Lyra
- 2 Mikhail Korobov
- 2 Xinfan Meng
- 1 Alejandro Weinstein
- 1 Alexandre Passos
- 1 Christoph Deil
- 1 Eugene Nizhibitsky
- 1 Kenneth C. Arnold
- 1 Luis Pedro Coelho
- 1 Miroslav Batchkarov
- 1 Pavel
- 1 Sebastian Berg
- 1 Shaun Jackman

- 1 Subhodeep Moitra
- 1 bob
- 1 dengemann
- 1 emanuele
- 1 x006

1.7.29 Version 0.12.1

October 8, 2012

The 0.12.1 release is a bug-fix release with no additional features, but is instead a set of bug fixes

Changelog

- Improved numerical stability in spectral embedding by Gael Varoquaux
- Doctest under windows 64bit by Gael Varoquaux
- Documentation fixes for elastic net by Andreas Müller and Alexandre Gramfort
- Proper behavior with fortran-ordered NumPy arrays by Gael Varoquaux
- Make GridSearchCV work with non-CSR sparse matrix by Lars Buitinck
- Fix parallel computing in MDS by Gael Varoquaux
- Fix Unicode support in count vectorizer by Andreas Müller
- Fix MinCovDet breaking with X.shape = (3, 1) by Virgile Fritsch
- Fix clone of SGD objects by Peter Prettenhofer
- Stabilize GMM by Virgile Fritsch

People

- 14 Peter Prettenhofer
- 12 Gael Varoquaux
- 10 Andreas Müller
- 5 Lars Buitinck
- 3 Virgile Fritsch
- 1 Alexandre Gramfort
- 1 Gilles Louppe
- 1 Mathieu Blondel

1.7.30 Version 0.12

September 4, 2012

Changelog

- Various speed improvements of the *decision trees* module, by Gilles Louppe.
- *ensemble.GradientBoostingRegressor* and *ensemble.GradientBoostingClassifier* now support feature subsampling via the `max_features` argument, by Peter Prettenhofer.
- Added Huber and Quantile loss functions to *ensemble.GradientBoostingRegressor*, by Peter Prettenhofer.
- *Decision trees* and *forests of randomized trees* now support multi-output classification and regression problems, by Gilles Louppe.
- Added *preprocessing.LabelEncoder*, a simple utility class to normalize labels or transform non-numerical labels, by Mathieu Blondel.
- Added the epsilon-insensitive loss and the ability to make probabilistic predictions with the modified huber loss in *Stochastic Gradient Descent*, by Mathieu Blondel.
- Added *Multi-dimensional Scaling (MDS)*, by Nelle Varoquaux.
- SVMlight file format loader now detects compressed (gzip/bzip2) files and decompresses them on the fly, by Lars Buitinck.
- SVMlight file format serializer now preserves double precision floating point values, by Olivier Grisel.
- A common testing framework for all estimators was added, by Andreas Müller.
- Understandable error messages for estimators that do not accept sparse input by Gael Varoquaux
- Speedups in hierarchical clustering by Gael Varoquaux. In particular building the tree now supports early stopping. This is useful when the number of clusters is not small compared to the number of samples.
- Add MultiTaskLasso and MultiTaskElasticNet for joint feature selection, by Alexandre Gramfort.
- Added `metrics.auc_score` and *metrics.average_precision_score* convenience functions by Andreas Müller.
- Improved sparse matrix support in the *Feature selection* module by Andreas Müller.
- New word boundaries-aware character n-gram analyzer for the *Text feature extraction* module by @kernc.
- Fixed bug in spectral clustering that led to single point clusters by Andreas Müller.
- In *feature_extraction.text.CountVectorizer*, added an option to ignore infrequent words, `min_df` by Andreas Müller.
- Add support for multiple targets in some linear models (ElasticNet, Lasso and OrthogonalMatchingPursuit) by Vlad Niculae and Alexandre Gramfort.
- Fixes in `decomposition.ProbabilisticPCA` score function by Wei Li.
- Fixed feature importance computation in *Gradient Tree Boosting*.

API changes summary

- The old `scikits.learn` package has disappeared; all code should import from `sklearn` instead, which was introduced in 0.9.
- In *metrics.roc_curve*, the `thresholds` array is now returned with it's order reversed, in order to keep it consistent with the order of the returned `fpr` and `tpr`.
- In `hmm` objects, like `hmm.GaussianHMM`, `hmm.MultinomialHMM`, etc., all parameters must be passed to the object when initialising it and not through `fit`. Now `fit` will only accept the data as an input parameter.

- For all SVM classes, a faulty behavior of `gamma` was fixed. Previously, the default gamma value was only computed the first time `fit` was called and then stored. It is now recalculated on every call to `fit`.
- All `Base` classes are now abstract meta classes so that they can not be instantiated.
- *`cluster.ward_tree`* now also returns the parent array. This is necessary for early-stopping in which case the tree is not completely built.
- In *`feature_extraction.text.CountVectorizer`* the parameters `min_n` and `max_n` were joined to the parameter `n_gram_range` to enable grid-searching both at once.
- In *`feature_extraction.text.CountVectorizer`*, words that appear only in one document are now ignored by default. To reproduce the previous behavior, set `min_df=1`.
- Fixed API inconsistency: *`linear_model.SGDClassifier.predict_proba`* now returns 2d array when fit on two classes.
- Fixed API inconsistency: *`discriminant_analysis.QuadraticDiscriminantAnalysis.decision_function`* and *`discriminant_analysis.LinearDiscriminantAnalysis.decision_function`* now return 1d arrays when fit on two classes.
- Grid of alphas used for fitting *`linear_model.LassoCV`* and *`linear_model.ElasticNetCV`* is now stored in the attribute `alphas_` rather than overriding the init parameter `alphas`.
- Linear models when alpha is estimated by cross-validation store the estimated value in the `alpha_` attribute rather than just `alpha` or `best_alpha`.
- *`ensemble.GradientBoostingClassifier`* now supports *`ensemble.GradientBoostingClassifier.staged_predict_proba`*, and *`ensemble.GradientBoostingClassifier.staged_predict`*.
- `svm.sparse.SVC` and other sparse SVM classes are now deprecated. The all classes in the *Support Vector Machines* module now automatically select the sparse or dense representation base on the input.
- All clustering algorithms now interpret the array `X` given to `fit` as input data, in particular *`cluster.SpectralClustering`* and *`cluster.AffinityPropagation`* which previously expected affinity matrices.
- For clustering algorithms that take the desired number of clusters as a parameter, this parameter is now called `n_clusters`.

People

- 267 Andreas Müller
- 94 Gilles Louppe
- 89 Gael Varoquaux
- 79 Peter Prettenhofer
- 60 Mathieu Blondel
- 57 Alexandre Gramfort
- 52 Vlad Niculae
- 45 Lars Buitinck
- 44 Nelle Varoquaux
- 37 Jaques Grobler
- 30 Alexis Mignon

- 30 Immanuel Bayer
- 27 Olivier Grisel
- 16 Subhodeep Moitra
- 13 Yannick Schwartz
- 12 @kernc
- 11 Virgile Fritsch
- 9 Daniel Duckworth
- 9 Fabian Pedregosa
- 9 Robert Layton
- 8 John Benediktsson
- 7 Marko Burjek
- 5 Nicolas Pinto
- 4 Alexandre Abraham
- 4 Jake Vanderplas
- 3 Brian Holt
- 3 Edouard Duchesnay
- 3 Florian Hoenig
- 3 flyingimmidev
- 2 Francois Savard
- 2 Hannes Schulz
- 2 Peter Welinder
- 2 Yaroslav Halchenko
- 2 Wei Li
- 1 Alex Companioni
- 1 Brandyn A. White
- 1 Bussonnier Matthias
- 1 Charles-Pierre Astolfi
- 1 Dan O'Huiginn
- 1 David Cournapeau
- 1 Keith Goodman
- 1 Ludwig Schwardt
- 1 Olivier Hervieu
- 1 Sergio Medina
- 1 Shiqiao Du
- 1 Tim Sheerman-Chase
- 1 buguen

1.7.31 Version 0.11

May 7, 2012

Changelog

Highlights

- Gradient boosted regression trees (*Gradient Tree Boosting*) for classification and regression by Peter Prettenhofer and Scott White .
- Simple dict-based feature loader with support for categorical variables (`feature_extraction.DictVectorizer`) by Lars Buitinck.
- Added Matthews correlation coefficient (`metrics.matthews_corrcoef`) and added macro and micro average options to `metrics.precision_score`, `metrics.recall_score` and `metrics.f1_score` by Satrajit Ghosh.
- *Out of Bag Estimates* of generalization error for *Ensemble methods* by Andreas Müller.
- Randomized sparse linear models for feature selection, by Alexandre Gramfort and Gael Varoquaux
- *Label Propagation* for semi-supervised learning, by Clay Woolam. **Note** the semi-supervised API is still work in progress, and may change.
- Added BIC/AIC model selection to classical *Gaussian mixture models* and unified the API with the remainder of scikit-learn, by Bertrand Thirion
- Added `sklearn.cross_validation.StratifiedShuffleSplit`, which is a `sklearn.cross_validation.ShuffleSplit` with balanced splits, by Yannick Schwartz.
- `sklearn.neighbors.NearestCentroid` classifier added, along with a `shrink_threshold` parameter, which implements **shrunken centroid classification**, by Robert Layton.

Other changes

- Merged dense and sparse implementations of *Stochastic Gradient Descent* module and exposed utility extension types for sequential datasets `seq_dataset` and weight vectors `weight_vector` by Peter Prettenhofer.
- Added `partial_fit` (support for online/minibatch learning) and warm_start to the *Stochastic Gradient Descent* module by Mathieu Blondel.
- Dense and sparse implementations of *Support Vector Machines* classes and `linear_model.LogisticRegression` merged by Lars Buitinck.
- Regressors can now be used as base estimator in the *Multiclass and multilabel algorithms* module by Mathieu Blondel.
- Added n_jobs option to `metrics.pairwise.pairwise_distances` and `metrics.pairwise.pairwise_kernels` for parallel computation, by Mathieu Blondel.
- *K-means* can now be run in parallel, using the `n_jobs` argument to either *K-means* or `KMeans`, by Robert Layton.
- Improved *Cross-validation: evaluating estimator performance* and *Tuning the hyper-parameters of an estimator* documentation and introduced the new `cross_validation.train_test_split` helper function by Olivier Grisel

- *svm.SVC* members coef_ and intercept_ changed sign for consistency with decision_function; for kernel==linear, coef_ was fixed in the one-vs-one case, by Andreas Müller.
- Performance improvements to efficient leave-one-out cross-validated Ridge regression, esp. for the n_samples > n_features case, in *linear_model.RidgeCV*, by Reuben Fletcher-Costin.
- Refactoring and simplification of the *Text feature extraction* API and fixed a bug that caused possible negative IDF, by Olivier Grisel.
- Beam pruning option in _BaseHMM module has been removed since it is difficult to Cythonize. If you are interested in contributing a Cython version, you can use the python version in the git history as a reference.
- Classes in *Nearest Neighbors* now support arbitrary Minkowski metric for nearest neighbors searches. The metric can be specified by argument p.

API changes summary

- covariance.EllipticEnvelop is now deprecated - Please use *covariance.EllipticEnvelope* instead.
- NeighborsClassifier and NeighborsRegressor are gone in the module *Nearest Neighbors*. Use the classes KNeighborsClassifier, RadiusNeighborsClassifier, KNeighborsRegressor and/or RadiusNeighborsRegressor instead.
- Sparse classes in the *Stochastic Gradient Descent* module are now deprecated.
- In mixture.GMM, mixture.DPGMM and mixture.VBGMM, parameters must be passed to an object when initialising it and not through fit. Now fit will only accept the data as an input parameter.
- methods rvs and decode in GMM module are now deprecated. sample and score or predict should be used instead.
- attribute _scores and _pvalues in univariate feature selection objects are now deprecated. scores_ or pvalues_ should be used instead.
- In LogisticRegression, LinearSVC, SVC and NuSVC, the class_weight parameter is now an initialization parameter, not a parameter to fit. This makes grid searches over this parameter possible.
- LFW data is now always shape (n_samples, n_features) to be consistent with the Olivetti faces dataset. Use images and pairs attribute to access the natural images shapes instead.
- In *svm.LinearSVC*, the meaning of the multi_class parameter changed. Options now are 'ovr' and 'crammer_singer', with 'ovr' being the default. This does not change the default behavior but hopefully is less confusing.
- Class feature_selection.text.Vectorizer is deprecated and replaced by feature_selection.text.TfidfVectorizer.
- The preprocessor / analyzer nested structure for text feature extraction has been removed. All those features are now directly passed as flat constructor arguments to feature_selection.text.TfidfVectorizer and feature_selection.text.CountVectorizer, in particular the following parameters are now used:
- analyzer can be 'word' or 'char' to switch the default analysis scheme, or use a specific python callable (as previously).
- tokenizer and preprocessor have been introduced to make it still possible to customize those steps with the new API.
- input explicitly control how to interpret the sequence passed to fit and predict: filenames, file objects or direct (byte or Unicode) strings.

- charset decoding is explicit and strict by default.
- the `vocabulary`, fitted or not is now stored in the `vocabulary_` attribute to be consistent with the project conventions.
- Class `feature_selection.text.TfidfVectorizer` now derives directly from `feature_selection.text.CountVectorizer` to make grid search trivial.
- methods `rvs` in `_BaseHMM` module are now deprecated. `sample` should be used instead.
- Beam pruning option in `_BaseHMM` module is removed since it is difficult to be Cythonized. If you are interested, you can look in the history codes by git.
- The SVMlight format loader now supports files with both zero-based and one-based column indices, since both occur "in the wild".
- Arguments in class `ShuffleSplit` are now consistent with `StratifiedShuffleSplit`. Arguments `test_fraction` and `train_fraction` are deprecated and renamed to `test_size` and `train_size` and can accept both `float` and `int`.
- Arguments in class `Bootstrap` are now consistent with `StratifiedShuffleSplit`. Arguments `n_test` and `n_train` are deprecated and renamed to `test_size` and `train_size` and can accept both `float` and `int`.
- Argument `p` added to classes in *Nearest Neighbors* to specify an arbitrary Minkowski metric for nearest neighbors searches.

People

- 282 Andreas Müller
- 239 Peter Prettenhofer
- 198 Gael Varoquaux
- 129 Olivier Grisel
- 114 Mathieu Blondel
- 103 Clay Woolam
- 96 Lars Buitinck
- 88 Jaques Grobler
- 82 Alexandre Gramfort
- 50 Bertrand Thirion
- 42 Robert Layton
- 28 flyingimmidev
- 26 Jake Vanderplas
- 26 Shiqiao Du
- 21 Satrajit Ghosh
- 17 David Marek
- 17 Gilles Louppe
- 14 Vlad Niculae
- 11 Yannick Schwartz

- 10 Fabian Pedregosa
- 9 fcostin
- 7 Nick Wilson
- 5 Adrien Gaidon
- 5 Nicolas Pinto
- 4 David Warde-Farley
- 5 Nelle Varoquaux
- 5 Emmanuelle Gouillart
- 3 Joonas Sillanpää
- 3 Paolo Losi
- 2 Charles McCarthy
- 2 Roy Hyunjin Han
- 2 Scott White
- 2 ibayer
- 1 Brandyn White
- 1 Carlos Scheidegger
- 1 Claire Revillet
- 1 Conrad Lee
- 1 Edouard Duchesnay
- 1 Jan Hendrik Metzen
- 1 Meng Xinfan
- 1 Rob Zinkov
- 1 Shiqiao
- 1 Udi Weinsberg
- 1 Virgile Fritsch
- 1 Xinfan Meng
- 1 Yaroslav Halchenko
- 1 jansoe
- 1 Leon Palafox

1.7.32 Version 0.10

January 11, 2012

Changelog

- Python 2.5 compatibility was dropped; the minimum Python version needed to use scikit-learn is now 2.6.
- *Sparse inverse covariance* estimation using the graph Lasso, with associated cross-validated estimator, by Gael Varoquaux
- New *Tree* module by Brian Holt, Peter Prettenhofer, Satrajit Ghosh and Gilles Louppe. The module comes with complete documentation and examples.
- Fixed a bug in the RFE module by Gilles Louppe (issue #378).
- Fixed a memory leak in *Support Vector Machines* module by Brian Holt (issue #367).
- Faster tests by Fabian Pedregosa and others.
- Silhouette Coefficient cluster analysis evaluation metric added as `sklearn.metrics.silhouette_score` by Robert Layton.
- Fixed a bug in *K-means* in the handling of the `n_init` parameter: the clustering algorithm used to be run `n_init` times but the last solution was retained instead of the best solution by Olivier Grisel.
- Minor refactoring in *Stochastic Gradient Descent* module; consolidated dense and sparse predict methods; Enhanced test time performance by converting model parameters to fortran-style arrays after fitting (only multiclass).
- Adjusted Mutual Information metric added as `sklearn.metrics.adjusted_mutual_info_score` by Robert Layton.
- Models like SVC/SVR/LinearSVC/LogisticRegression from libsvm/liblinear now support scaling of C regularization parameter by the number of samples by Alexandre Gramfort.
- New *Ensemble Methods* module by Gilles Louppe and Brian Holt. The module comes with the random forest algorithm and the extra-trees method, along with documentation and examples.
- *Novelty and Outlier Detection*: outlier and novelty detection, by Virgile Fritsch.
- *Kernel Approximation*: a transform implementing kernel approximation for fast SGD on non-linear kernels by Andreas Müller.
- Fixed a bug due to atom swapping in *Orthogonal Matching Pursuit (OMP)* by Vlad Niculae.
- *Sparse coding with a precomputed dictionary* by Vlad Niculae.
- *Mini Batch K-Means* performance improvements by Olivier Grisel.
- *K-means* support for sparse matrices by Mathieu Blondel.
- Improved documentation for developers and for the `sklearn.utils` module, by Jake Vanderplas.
- Vectorized 20newsgroups dataset loader (`sklearn.datasets.fetch_20newsgroups_vectorized`) by Mathieu Blondel.
- *Multiclass and multilabel algorithms* by Lars Buitinck.
- Utilities for fast computation of mean and variance for sparse matrices by Mathieu Blondel.
- Make `sklearn.preprocessing.scale` and `sklearn.preprocessing.Scaler` work on sparse matrices by Olivier Grisel
- Feature importances using decision trees and/or forest of trees, by Gilles Louppe.
- Parallel implementation of forests of randomized trees by Gilles Louppe.
- `sklearn.cross_validation.ShuffleSplit` can subsample the train sets as well as the test sets by Olivier Grisel.

- Errors in the build of the documentation fixed by Andreas Müller.

API changes summary

Here are the code migration instructions when upgrading from scikit-learn version 0.9:

- Some estimators that may overwrite their inputs to save memory previously had `overwrite_` parameters; these have been replaced with `copy_` parameters with exactly the opposite meaning.

  This particularly affects some of the estimators in `linear_model`. The default behavior is still to copy everything passed in.

- The SVMlight dataset loader *`sklearn.datasets.load_svmlight_file`* no longer supports loading two files at once; use `load_svmlight_files` instead. Also, the (unused) `buffer_mb` parameter is gone.

- Sparse estimators in the *Stochastic Gradient Descent* module use dense parameter vector `coef_` instead of `sparse_coef_`. This significantly improves test time performance.

- The *Covariance estimation* module now has a robust estimator of covariance, the Minimum Covariance Determinant estimator.

- Cluster evaluation metrics in `metrics.cluster` have been refactored but the changes are backwards compatible. They have been moved to the `metrics.cluster.supervised`, along with `metrics.cluster.unsupervised` which contains the Silhouette Coefficient.

- The `permutation_test_score` function now behaves the same way as `cross_val_score` (i.e. uses the mean score across the folds.)

- Cross Validation generators now use integer indices (`indices=True`) by default instead of boolean masks. This make it more intuitive to use with sparse matrix data.

- The functions used for sparse coding, `sparse_encode` and `sparse_encode_parallel` have been combined into *`sklearn.decomposition.sparse_encode`*, and the shapes of the arrays have been transposed for consistency with the matrix factorization setting, as opposed to the regression setting.

- Fixed an off-by-one error in the SVMlight/LibSVM file format handling; files generated using *`sklearn.datasets.dump_svmlight_file`* should be re-generated. (They should continue to work, but accidentally had one extra column of zeros prepended.)

- `BaseDictionaryLearning` class replaced by `SparseCodingMixin`.

- `sklearn.utils.extmath.fast_svd` has been renamed *`sklearn.utils.extmath.randomized_svd`* and the default oversampling is now fixed to 10 additional random vectors instead of doubling the number of components to extract. The new behavior follows the reference paper.

People

The following people contributed to scikit-learn since last release:

- 246 Andreas Müller
- 242 Olivier Grisel
- 220 Gilles Louppe
- 183 Brian Holt
- 166 Gael Varoquaux
- 144 Lars Buitinck
- 73 Vlad Niculae

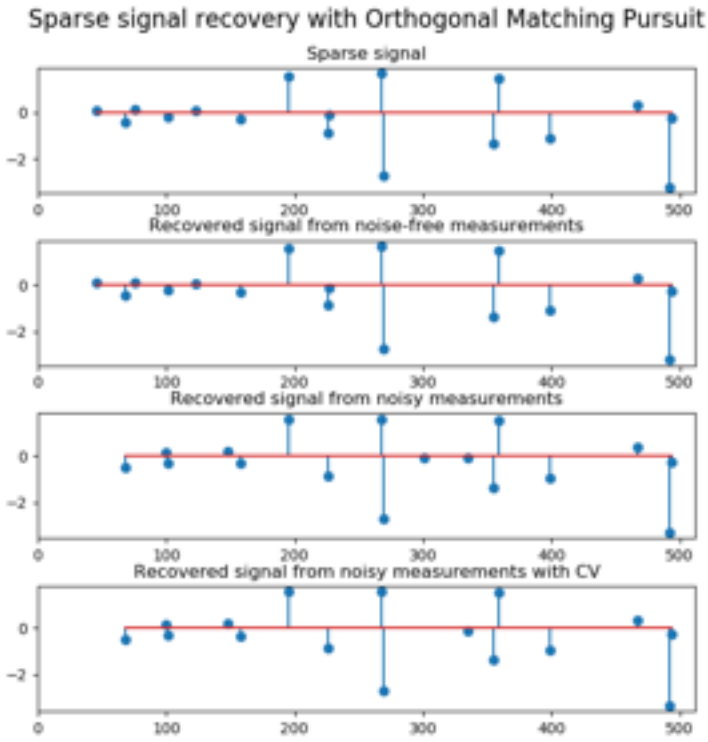
- 65 Peter Prettenhofer
- 64 Fabian Pedregosa
- 60 Robert Layton
- 55 Mathieu Blondel
- 52 Jake Vanderplas
- 44 Noel Dawe
- 38 Alexandre Gramfort
- 24 Virgile Fritsch
- 23 Satrajit Ghosh
- 3 Jan Hendrik Metzen
- 3 Kenneth C. Arnold
- 3 Shiqiao Du
- 3 Tim Sheerman-Chase
- 3 Yaroslav Halchenko
- 2 Bala Subrahmanyam Varanasi
- 2 DraXus
- 2 Michael Eickenberg
- 1 Bogdan Trach
- 1 Félix-Antoine Fortin
- 1 Juan Manuel Caicedo Carvajal
- 1 Nelle Varoquaux
- 1 Nicolas Pinto
- 1 Tiziano Zito
- 1 Xinfan Meng

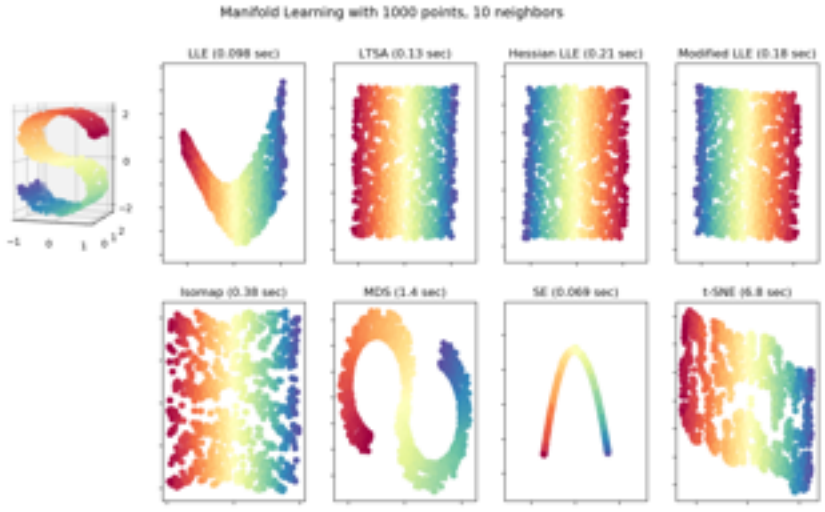
1.7.33 Version 0.9

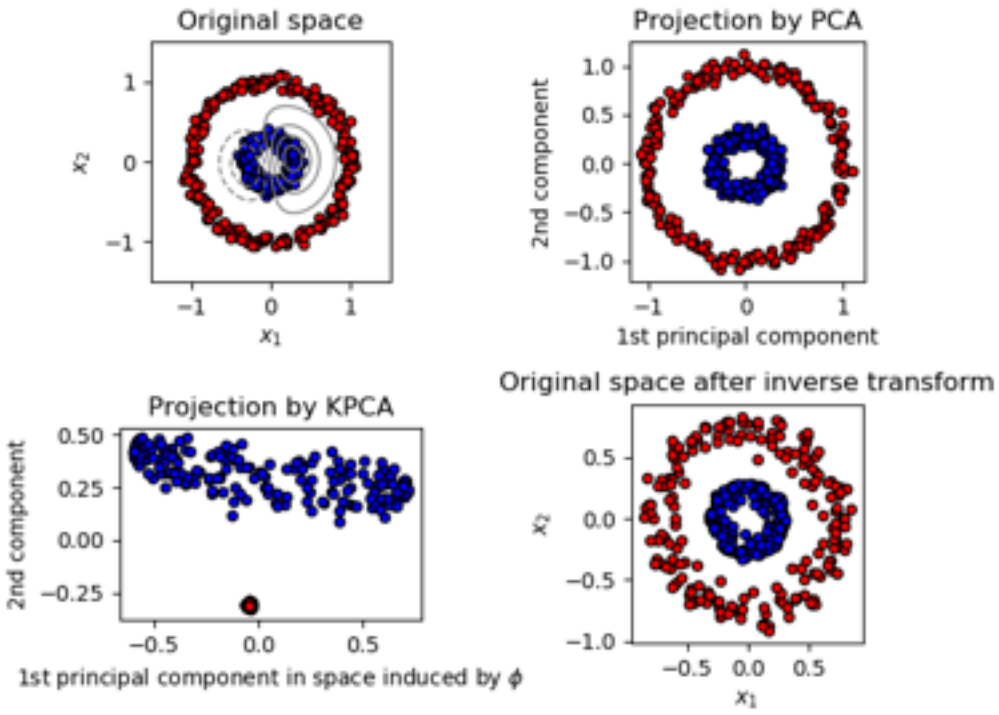
September 21, 2011

scikit-learn 0.9 was released on September 2011, three months after the 0.8 release and includes the new modules *Manifold learning*, *The Dirichlet Process* as well as several new algorithms and documentation improvements.

This release also includes the dictionary-learning work developed by Vlad Niculae as part of the Google Summer of Code program.


Sparse signal recovery with Orthogonal Matching Pursuit
Sparse signal
Recovered signal from noise-free measurements
Recovered signal from noisy measurements
Recovered signal from noisy measurements with CV


Manifold Learning with 1000 points, 10 neighbors
LLE (0.098 sec)
LTSA (0.13 sec)
Hessian LLE (0.21 sec)
Modified LLE (0.18 sec)
Isomap (0.38 sec)
MDS (1.4 sec)
SE (0.069 sec)
t-SNE (6.8 sec)



Changelog

- New *Manifold learning* module by Jake Vanderplas and Fabian Pedregosa.
- New *Dirichlet Process* Gaussian Mixture Model by Alexandre Passos
- *Nearest Neighbors* module refactoring by Jake Vanderplas : general refactoring, support for sparse matrices in input, speed and documentation improvements. See the next section for a full list of API changes.
- Improvements on the *Feature selection* module by Gilles Louppe : refactoring of the RFE classes, documentation rewrite, increased efficiency and minor API changes.
- *Sparse principal components analysis (SparsePCA and MiniBatchSparsePCA)* by Vlad Niculae, Gael Varoquaux and Alexandre Gramfort
- Printing an estimator now behaves independently of architectures and Python version thanks to Jean Kossaifi.
- *Loader for libsvm/svmlight format* by Mathieu Blondel and Lars Buitinck
- Documentation improvements: thumbnails in example gallery by Fabian Pedregosa.
- Important bugfixes in *Support Vector Machines* module (segfaults, bad performance) by Fabian Pedregosa.
- Added *Multinomial Naive Bayes* and *Bernoulli Naive Bayes* by Lars Buitinck
- Text feature extraction optimizations by Lars Buitinck
- Chi-Square feature selection (`feature_selection.univariate_selection.chi2`) by Lars Buitinck.
- *Generated datasets* module refactoring by Gilles Louppe
- *Multiclass and multilabel algorithms* by Mathieu Blondel
- Ball tree rewrite by Jake Vanderplas

- Implementation of *DBSCAN* algorithm by Robert Layton
- Kmeans predict and transform by Robert Layton
- Preprocessing module refactoring by Olivier Grisel
- Faster mean shift by Conrad Lee
- New `Bootstrap`, *Random permutations cross-validation a.k.a. Shuffle & Split* and various other improvements in cross validation schemes by Olivier Grisel and Gael Varoquaux
- Adjusted Rand index and V-Measure clustering evaluation metrics by Olivier Grisel
- Added *`Orthogonal Matching Pursuit`* by Vlad Niculae
- Added 2D-patch extractor utilities in the *Feature extraction* module by Vlad Niculae
- Implementation of *`linear_model.LassoLarsCV`* (cross-validated Lasso solver using the Lars algorithm) and *`linear_model.LassoLarsIC`* (BIC/AIC model selection in Lars) by Gael Varoquaux and Alexandre Gramfort
- Scalability improvements to *`metrics.roc_curve`* by Olivier Hervieu
- Distance helper functions `metrics.pairwise.pairwise_distances` and *`metrics.pairwise.pairwise_kernels`* by Robert Layton
- *`Mini-Batch K-Means`* by Nelle Varoquaux and Peter Prettenhofer.
- mldata utilities by Pietro Berkes.
- *The Olivetti faces dataset* by David Warde-Farley.

API changes summary

Here are the code migration instructions when upgrading from scikit-learn version 0.8:

- The `scikits.learn` package was renamed `sklearn`. There is still a `scikits.learn` package alias for backward compatibility.

  Third-party projects with a dependency on scikit-learn 0.9+ should upgrade their codebase. For instance, under Linux / MacOSX just run (make a backup first!):

  ```
  find -name "*.py" | xargs sed -i 's/\bscikits.learn\b/sklearn/g'
  ```

- Estimators no longer accept model parameters as `fit` arguments: instead all parameters must be only be passed as constructor arguments or using the now public `set_params` method inherited from *`base.BaseEstimator`*.

  Some estimators can still accept keyword arguments on the `fit` but this is restricted to data-dependent values (e.g. a Gram matrix or an affinity matrix that are precomputed from the `X` data matrix.

- The `cross_val` package has been renamed to `cross_validation` although there is also a `cross_val` package alias in place for backward compatibility.

  Third-party projects with a dependency on scikit-learn 0.9+ should upgrade their codebase. For instance, under Linux / MacOSX just run (make a backup first!):

  ```
  find -name "*.py" | xargs sed -i 's/\bcross_val\b/cross_validation/g'
  ```

- The `score_func` argument of the `sklearn.cross_validation.cross_val_score` function is now expected to accept `y_test` and `y_predicted` as only arguments for classification and regression tasks or `X_test` for unsupervised estimators.

- `gamma` parameter for support vector machine algorithms is set to `1 / n_features` by default, instead of `1 / n_samples`.
- The `sklearn.hmm` has been marked as orphaned: it will be removed from scikit-learn in version 0.11 unless someone steps up to contribute documentation, examples and fix lurking numerical stability issues.
- `sklearn.neighbors` has been made into a submodule. The two previously available estimators, `NeighborsClassifier` and `NeighborsRegressor` have been marked as deprecated. Their functionality has been divided among five new classes: `NearestNeighbors` for unsupervised neighbors searches, `KNeighborsClassifier` & `RadiusNeighborsClassifier` for supervised classification problems, and `KNeighborsRegressor` & `RadiusNeighborsRegressor` for supervised regression problems.
- `sklearn.ball_tree.BallTree` has been moved to `sklearn.neighbors.BallTree`. Using the former will generate a warning.
- `sklearn.linear_model.LARS()` and related classes (LassoLARS, LassoLARSCV, etc.) have been renamed to `sklearn.linear_model.Lars()`.
- All distance metrics and kernels in `sklearn.metrics.pairwise` now have a Y parameter, which by default is None. If not given, the result is the distance (or kernel similarity) between each sample in Y. If given, the result is the pairwise distance (or kernel similarity) between samples in X to Y.
- `sklearn.metrics.pairwise.l1_distance` is now called `manhattan_distance`, and by default returns the pairwise distance. For the component wise distance, set the parameter `sum_over_features` to `False`.

Backward compatibility package aliases and other deprecated classes and functions will be removed in version 0.11.

People

38 people contributed to this release.

- 387 Vlad Niculae
- 320 Olivier Grisel
- 192 Lars Buitinck
- 179 Gael Varoquaux
- 168 Fabian Pedregosa (INRIA, Parietal Team)
- 127 Jake Vanderplas
- 120 Mathieu Blondel
- 85 Alexandre Passos
- 67 Alexandre Gramfort
- 57 Peter Prettenhofer
- 56 Gilles Louppe
- 42 Robert Layton
- 38 Nelle Varoquaux
- 32 Jean Kossaifi
- 30 Conrad Lee
- 22 Pietro Berkes
- 18 andy

- 17 David Warde-Farley
- 12 Brian Holt
- 11 Robert
- 8 Amit Aides
- 8 Virgile Fritsch
- 7 Yaroslav Halchenko
- 6 Salvatore Masecchia
- 5 Paolo Losi
- 4 Vincent Schut
- 3 Alexis Metaireau
- 3 Bryan Silverthorn
- 3 Andreas Müller
- 2 Minwoo Jake Lee
- 1 Emmanuelle Gouillart
- 1 Keith Goodman
- 1 Lucas Wiman
- 1 Nicolas Pinto
- 1 Thouis (Ray) Jones
- 1 Tim Sheerman-Chase

1.7.34 Version 0.8

May 11, 2011

scikit-learn 0.8 was released on May 2011, one month after the first "international" scikit-learn coding sprint and is marked by the inclusion of important modules: *Hierarchical clustering*, *Cross decomposition*, *Non-negative matrix factorization (NMF or NNMF)*, initial support for Python 3 and by important enhancements and bug fixes.

Changelog

Several new modules where introduced during this release:

- New *Hierarchical clustering* module by Vincent Michel, Bertrand Thirion, Alexandre Gramfort and Gael Varoquaux.
- *Kernel PCA* implementation by Mathieu Blondel
- *The Labeled Faces in the Wild face recognition dataset* by Olivier Grisel.
- New *Cross decomposition* module by Edouard Duchesnay.
- *Non-negative matrix factorization (NMF or NNMF)* module Vlad Niculae
- Implementation of the *Oracle Approximating Shrinkage* algorithm by Virgile Fritsch in the *Covariance estimation* module.

Some other modules benefited from significant improvements or cleanups.

- Initial support for Python 3: builds and imports cleanly, some modules are usable while others have failing tests by Fabian Pedregosa.
- `decomposition.PCA` is now usable from the Pipeline object by Olivier Grisel.
- Guide *How to optimize for speed* by Olivier Grisel.
- Fixes for memory leaks in libsvm bindings, 64-bit safer BallTree by Lars Buitinck.
- bug and style fixing in *K-means* algorithm by Jan Schlüter.
- Add attribute converged to Gaussian Mixture Models by Vincent Schut.
- Implemented `transform`, `predict_log_proba` in `discriminant_analysis.LinearDiscriminantAnalysis` By Mathieu Blondel.
- Refactoring in the *Support Vector Machines* module and bug fixes by Fabian Pedregosa, Gael Varoquaux and Amit Aides.
- Refactored SGD module (removed code duplication, better variable naming), added interface for sample weight by Peter Prettenhofer.
- Wrapped BallTree with Cython by Thouis (Ray) Jones.
- Added function `svm.l1_min_c` by Paolo Losi.
- Typos, doc style, etc. by Yaroslav Halchenko, Gael Varoquaux, Olivier Grisel, Yann Malet, Nicolas Pinto, Lars Buitinck and Fabian Pedregosa.

People

People that made this release possible preceded by number of commits:

- 159 Olivier Grisel
- 96 Gael Varoquaux
- 96 Vlad Niculae
- 94 Fabian Pedregosa
- 36 Alexandre Gramfort
- 32 Paolo Losi
- 31 Edouard Duchesnay
- 30 Mathieu Blondel
- 25 Peter Prettenhofer
- 22 Nicolas Pinto
- **11 Virgile Fritsch**
    - 7 Lars Buitinck
    - 6 Vincent Michel
    - 5 Bertrand Thirion
    - 4 Thouis (Ray) Jones
    - 4 Vincent Schut
    - 3 Jan Schlüter
    - 2 Julien Miotte

- 2 Matthieu Perrot
- 2 Yann Malet
- 2 Yaroslav Halchenko
- 1 Amit Aides
- 1 Andreas Müller
- 1 Feth Arezki
- 1 Meng Xinfan

1.7.35 Version 0.7

March 2, 2011

scikit-learn 0.7 was released in March 2011, roughly three months after the 0.6 release. This release is marked by the speed improvements in existing algorithms like k-Nearest Neighbors and K-Means algorithm and by the inclusion of an efficient algorithm for computing the Ridge Generalized Cross Validation solution. Unlike the preceding release, no new modules where added to this release.

Changelog

- Performance improvements for Gaussian Mixture Model sampling [Jan Schlüter].
- Implementation of efficient leave-one-out cross-validated Ridge in *linear_model.RidgeCV* [Mathieu Blondel]
- Better handling of collinearity and early stopping in *linear_model.lars_path* [Alexandre Gramfort and Fabian Pedregosa].
- Fixes for liblinear ordering of labels and sign of coefficients [Dan Yamins, Paolo Losi, Mathieu Blondel and Fabian Pedregosa].
- Performance improvements for Nearest Neighbors algorithm in high-dimensional spaces [Fabian Pedregosa].
- Performance improvements for *cluster.KMeans* [Gael Varoquaux and James Bergstra].
- Sanity checks for SVM-based classes [Mathieu Blondel].
- Refactoring of `neighbors.NeighborsClassifier` and *neighbors.kneighbors_graph*: added different algorithms for the k-Nearest Neighbor Search and implemented a more stable algorithm for finding barycenter weights. Also added some developer documentation for this module, see notes_neighbors for more information [Fabian Pedregosa].
- Documentation improvements: Added `pca.RandomizedPCA` and *linear_model.LogisticRegression* to the class reference. Also added references of matrices used for clustering and other fixes [Gael Varoquaux, Fabian Pedregosa, Mathieu Blondel, Olivier Grisel, Virgile Fritsch , Emmanuelle Gouillart]
- Binded decision_function in classes that make use of liblinear, dense and sparse variants, like *svm.LinearSVC* or *linear_model.LogisticRegression* [Fabian Pedregosa].
- Performance and API improvements to `metrics.euclidean_distances` and to `pca.RandomizedPCA` [James Bergstra].
- Fix compilation issues under NetBSD [Kamel Ibn Hassen Derouiche]
- Allow input sequences of different lengths in `hmm.GaussianHMM` [Ron Weiss].

- Fix bug in affinity propagation caused by incorrect indexing [Xinfan Meng]

People

People that made this release possible preceded by number of commits:

- 85 Fabian Pedregosa
- 67 Mathieu Blondel
- 20 Alexandre Gramfort
- 19 James Bergstra
- 14 Dan Yamins
- 13 Olivier Grisel
- 12 Gael Varoquaux
- 4 Edouard Duchesnay
- 4 Ron Weiss
- 2 Satrajit Ghosh
- 2 Vincent Dubourg
- 1 Emmanuelle Gouillart
- 1 Kamel Ibn Hassen Derouiche
- 1 Paolo Losi
- 1 VirgileFritsch
- 1 Yaroslav Halchenko
- 1 Xinfan Meng

1.7.36 Version 0.6

December 21, 2010

scikit-learn 0.6 was released on December 2010. It is marked by the inclusion of several new modules and a general renaming of old ones. It is also marked by the inclusion of new example, including applications to real-world datasets.

Changelog

- New stochastic gradient descent module by Peter Prettenhofer. The module comes with complete documentation and examples.
- Improved svm module: memory consumption has been reduced by 50%, heuristic to automatically set class weights, possibility to assign weights to samples (see *SVM: Weighted samples* for an example).
- New *Gaussian Processes* module by Vincent Dubourg. This module also has great documentation and some very neat examples. See example_gaussian_process_plot_gp_regression.py or example_gaussian_process_plot_gp_probabilistic_classification_after_regression.py for a taste of what can be done.
- It is now possible to use liblinear's Multi-class SVC (option multi_class in `svm.LinearSVC`)
- New features and performance improvements of text feature extraction.

- Improved sparse matrix support, both in main classes (`grid_search.GridSearchCV`) as in modules sklearn.svm.sparse and sklearn.linear_model.sparse.
- Lots of cool new examples and a new section that uses real-world datasets was created. These include: *Faces recognition example using eigenfaces and SVMs*, *Species distribution modeling*, *Libsvm GUI*, *Wikipedia principal eigenvector* and others.
- Faster *Least Angle Regression* algorithm. It is now 2x faster than the R version on worst case and up to 10x times faster on some cases.
- Faster coordinate descent algorithm. In particular, the full path version of lasso (*linear_model.lasso_path*) is more than 200x times faster than before.
- It is now possible to get probability estimates from a *linear_model.LogisticRegression* model.
- module renaming: the glm module has been renamed to linear_model, the gmm module has been included into the more general mixture model and the sgd module has been included in linear_model.
- Lots of bug fixes and documentation improvements.

People

People that made this release possible preceded by number of commits:

- 207 Olivier Grisel
- 167 Fabian Pedregosa
- 97 Peter Prettenhofer
- 68 Alexandre Gramfort
- 59 Mathieu Blondel
- 55 Gael Varoquaux
- 33 Vincent Dubourg
- 21 Ron Weiss
- 9 Bertrand Thirion
- 3 Alexandre Passos
- 3 Anne-Laure Fouque
- 2 Ronan Amicel
- 1 Christian Osendorfer

1.7.37 Version 0.5

October 11, 2010

Changelog

New classes

- Support for sparse matrices in some classifiers of modules `svm` and `linear_model` (see `svm.sparse.SVC`, `svm.sparse.SVR`, `svm.sparse.LinearSVC`, `linear_model.sparse.Lasso`, `linear_model.sparse.ElasticNet`)

- New *pipeline.Pipeline* object to compose different estimators.
- Recursive Feature Elimination routines in module *Feature selection*.
- Addition of various classes capable of cross validation in the linear_model module (*linear_model.LassoCV*, *linear_model.ElasticNetCV*, etc.).
- New, more efficient LARS algorithm implementation. The Lasso variant of the algorithm is also implemented. See *linear_model.lars_path*, *linear_model.Lars* and *linear_model.LassoLars*.
- New Hidden Markov Models module (see classes `hmm.GaussianHMM`, `hmm.MultinomialHMM`, `hmm.GMMHMM`)
- New module feature_extraction (see *class reference*)
- New FastICA algorithm in module sklearn.fastica

Documentation

- Improved documentation for many modules, now separating narrative documentation from the class reference. As an example, see documentation for the SVM module and the complete class reference.

Fixes

- API changes: adhere variable names to PEP-8, give more meaningful names.
- Fixes for svm module to run on a shared memory context (multiprocessing).
- It is again possible to generate latex (and thus PDF) from the sphinx docs.

Examples

- new examples using some of the mlcomp datasets: `sphx_glr_auto_examples_mlcomp_sparse_document_classif` `py` (since removed) and *Classification of text documents using sparse features*
- Many more examples. See here the full list of examples.

External dependencies

- Joblib is now a dependency of this package, although it is shipped with (sklearn.externals.joblib).

Removed modules

- Module ann (Artificial Neural Networks) has been removed from the distribution. Users wanting this sort of algorithms should take a look into pybrain.

Misc

- New sphinx theme for the web page.

Authors

The following is a list of authors for this release, preceded by number of commits:

- 262 Fabian Pedregosa
- 240 Gael Varoquaux
- 149 Alexandre Gramfort
- 116 Olivier Grisel
- 40 Vincent Michel
- 38 Ron Weiss
- 23 Matthieu Perrot
- 10 Bertrand Thirion
- 7 Yaroslav Halchenko
- 9 VirgileFritsch
- 6 Edouard Duchesnay
- 4 Mathieu Blondel
- 1 Ariel Rokem
- 1 Matthieu Brucher

1.7.38 Version 0.4

August 26, 2010

Changelog

Major changes in this release include:

- Coordinate Descent algorithm (Lasso, ElasticNet) refactoring & speed improvements (roughly 100x times faster).
- Coordinate Descent Refactoring (and bug fixing) for consistency with R's package GLMNET.
- New metrics module.
- New GMM module contributed by Ron Weiss.
- Implementation of the LARS algorithm (without Lasso variant for now).
- feature_selection module redesign.
- Migration to GIT as version control system.
- Removal of obsolete attrselect module.
- Rename of private compiled extensions (added underscore).
- Removal of legacy unmaintained code.
- Documentation improvements (both docstring and rst).
- Improvement of the build system to (optionally) link with MKL. Also, provide a lite BLAS implementation in case no system-wide BLAS is found.

- Lots of new examples.
- Many, many bug fixes ...

Authors

The committer list for this release is the following (preceded by number of commits):

- 143 Fabian Pedregosa
- 35 Alexandre Gramfort
- 34 Olivier Grisel
- 11 Gael Varoquaux
- 5 Yaroslav Halchenko
- 2 Vincent Michel
- 1 Chris Filo Gorgolewski

1.7.39 Earlier versions

Earlier versions included contributions by Fred Mailhot, David Cooke, David Huard, Dave Morrill, Ed Schofield, Travis Oliphant, Pearu Peterson.

1.8 Roadmap

1.8.1 Purpose of this document

This document list general directions that core contributors are interested to see developed in scikit-learn. The fact that an item is listed here is in no way a promise that it will happen, as resources are limited. Rather, it is an indication that help is welcomed on this topic.

1.8.2 Statement of purpose: Scikit-learn in 2018

Eleven years after the inception of Scikit-learn, much has changed in the world of machine learning. Key changes include:

- Computational tools: The exploitation of GPUs, distributed programming frameworks like Scala/Spark, etc.
- High-level Python libraries for experimentation, processing and data management: Jupyter notebook, Cython, Pandas, Dask, Numba...
- Changes in the focus of machine learning research: artificial intelligence applications (where input structure is key) with deep learning, representation learning, reinforcement learning, domain transfer, etc.

A more subtle change over the last decade is that, due to changing interests in ML, PhD students in machine learning are more likely to contribute to PyTorch, Dask, etc. than to Scikit-learn, so our contributor pool is very different to a decade ago.

Scikit-learn remains very popular in practice for trying out canonical machine learning techniques, particularly for applications in experimental science and in data science. A lot of what we provide is now very mature. But it can be costly to maintain, and we cannot therefore include arbitrary new implementations. Yet Scikit-learn is also essential

in defining an API framework for the development of interoperable machine learning components external to the core library.

Thus our main goals in this era are to:

- continue maintaining a high-quality, well-documented collection of canonical tools for data processing and machine learning within the current scope (i.e. rectangular data largely invariant to column and row order; predicting targets with simple structure)
- improve the ease for users to develop and publish external components
- improve inter-operability with modern data science tools (e.g. Pandas, Dask) and infrastructures (e.g. distributed processing)

Many of the more fine-grained goals can be found under the API tag on the issue tracker.

1.8.3 Architectural / general goals

The list is numbered not as an indication of the order of priority, but to make referring to specific points easier. Please add new entries only at the bottom. Note that the crossed out entries are already done, and we try to keep the document up to date as we work on these issues.

1. Improved handling of Pandas DataFrames
    - document current handling
    - column reordering issue #7242
    - avoiding unnecessary conversion to ndarray #12147
    - returning DataFrames from transformers #5523
    - getting DataFrames from dataset loaders #10733, #13902
    - Sparse currently not considered #12800
2. Improved handling of categorical features
    - Tree-based models should be able to handle both continuous and categorical features #12866 and #15550.
    - In dataset loaders #13902
    - As generic transformers to be used with ColumnTransforms (e.g. ordinal encoding supervised by correlation with target variable) #5853, #11805
    - Handling mixtures of categorical and continuous variables
3. Improved handling of missing data
    - Making sure meta-estimators are lenient towards missing data, #15319
    - Non-trivial imputers #11977, #12852
    - Learners directly handling missing data #13911
    - An amputation sample generator to make parts of a dataset go missing #6284
4. More didactic documentation
    - More and more options have been added to scikit-learn. As a result, the documentation is crowded which makes it hard for beginners to get the big picture. Some work could be done in prioritizing the information.
5. Passing around information that is not (X, y): Sample properties
    - We need to be able to pass sample weights to scorers in cross validation.

- We should have standard/generalised ways of passing sample-wise properties around in meta-estimators. #4497 #7646

6. Passing around information that is not (X, y): Feature properties
   - Feature names or descriptions should ideally be available to fit for, e.g. . #6425 #6424
   - Per-feature handling (e.g. "is this a nominal / ordinal / English language text?") should also not need to be provided to estimator constructors, ideally, but should be available as metadata alongside X. #8480
7. Passing around information that is not (X, y): Target information
   - We have problems getting the full set of classes to all components when the data is split/sampled. #6231 #8100
   - We have no way to handle a mixture of categorical and continuous targets.
8. Make it easier for external users to write Scikit-learn-compatible components
   - More flexible estimator checks that do not select by estimator name #6599 #6715
   - Example of how to develop an estimator or a meta-estimator, #14582
   - More self-sufficient running of scikit-learn-contrib or a similar resource
9. Support resampling and sample reduction
   - Allow subsampling of majority classes (in a pipeline?) #3855
   - Implement random forests with resampling #8732
10. Better interfaces for interactive development
    - __repr__ and HTML visualisations of estimators #6323 and #14180.
    - Include plotting tools, not just as examples. #9173
11. Improved tools for model diagnostics and basic inference
    - alternative feature importances implementations, #13146
    - better ways to handle validation sets when fitting
    - better ways to find thresholds / create decision rules #8614
12. Better tools for selecting hyperparameters with transductive estimators
    - Grid search and cross validation are not applicable to most clustering tasks. Stability-based selection is more relevant.
13. Better support for manual and automatic pipeline building
    - Easier way to construct complex pipelines and valid search spaces #7608 #5082 #8243
    - provide search ranges for common estimators??
    - cf. searchgrid
14. Improved tracking of fitting
    - Verbose is not very friendly and should use a standard logging library #6929, #78
    - Callbacks or a similar system would facilitate logging and early stopping
15. Distributed parallelism
    - Accept data which complies with `__array_function__`
16. A way forward for more out of core

- Dask enables easy out-of-core computation. While the Dask model probably cannot be adaptable to all machine-learning algorithms, most machine learning is on smaller data than ETL, hence we can maybe adapt to very large scale while supporting only a fraction of the patterns.

17. Support for working with pre-trained models
    - Estimator "freezing". In particular, right now it's impossible to clone a `CalibratedClassifierCV` with prefit. #8370. #6451
18. Backwards-compatible de/serialization of some estimators
    - Currently serialization (with pickle) breaks across versions. While we may not be able to get around other limitations of pickle re security etc, it would be great to offer cross-version safety from version 1.0. Note: Gael and Olivier think that this can cause heavy maintenance burden and we should manage the trade-offs. A possible alternative is presented in the following point.
19. Documentation and tooling for model lifecycle management
    - Document good practices for model deployments and lifecycle: before deploying a model: snapshot the code versions (numpy, scipy, scikit-learn, custom code repo), the training script and an alias on how to retrieve historical training data + snapshot a copy of a small validation set + snapshot of the predictions (predicted probabilities for classifiers) on that validation set.
    - Document and tools to make it easy to manage upgrade of scikit-learn versions:
      - Try to load the old pickle, if it works, use the validation set prediction snapshot to detect that the serialized model still behave the same;
      - If joblib.load / pickle.load not work, use the versioned control training script + historical training set to retrain the model and use the validation set prediction snapshot to assert that it is possible to recover the previous predictive performance: if this is not the case there is probably a bug in scikit-learn that needs to be reported.
20. Everything in Scikit-learn should probably conform to our API contract. We are still in the process of making decisions on some of these related issues.
    - `Pipeline <pipeline.Pipeline>` and `FeatureUnion` modify their input parameters in fit. Fixing this requires making sure we have a good grasp of their use cases to make sure all current functionality is maintained. #8157 #7382
21. (Optional) Improve scikit-learn common tests suite to make sure that (at least for frequently used) models have stable predictions across-versions (to be discussed);
    - Extend documentation to mention how to deploy models in Python-free environments for instance ONNX. and use the above best practices to assess predictive consistency between scikit-learn and ONNX prediction functions on validation set.
    - Document good practices to detect temporal distribution drift for deployed model and good practices for re-training on fresh data without causing catastrophic predictive performance regressions.

1.8.4 Subpackage-specific goals

sklearn.ensemble

- a stacking implementation, #11047

sklearn.cluster

- kmeans variants for non-Euclidean distances, if we can show these have benefits beyond hierarchical clustering.

sklearn.model_selection

- multi-metric scoring is slow #9326

- perhaps we want to be able to get back more than multiple metrics
- the handling of random states in CV splitters is a poor design and contradicts the validation of similar parameters in estimators, #15177
- exploit warm-starting and path algorithms so the benefits of `EstimatorCV` objects can be accessed via `GridSearchCV` and used in Pipelines. #1626
- Cross-validation should be able to be replaced by OOB estimates whenever a cross-validation iterator is used.
- Redundant computations in pipelines should be avoided (related to point above) cf daskml

`sklearn.neighbors`

- Ability to substitute a custom/approximate/precomputed nearest neighbors implementation for ours in all/most contexts that nearest neighbors are used for learning. #10463

`sklearn.pipeline`

- Performance issues with `Pipeline.memory`
- see "Everything in Scikit-learn should conform to our API contract" above

1.9 Scikit-learn governance and decision-making

The purpose of this document is to formalize the governance process used by the scikit-learn project, to clarify how decisions are made and how the various elements of our community interact. This document establishes a decision-making structure that takes into account feedback from all members of the community and strives to find consensus, while avoiding any deadlocks.

This is a meritocratic, consensus-based community project. Anyone with an interest in the project can join the community, contribute to the project design and participate in the decision making process. This document describes how that participation takes place and how to set about earning merit within the project community.

1.9.1 Roles And Responsibilities

Contributors

Contributors are community members who contribute in concrete ways to the project. Anyone can become a contributor, and contributions can take many forms – not only code – as detailed in the *contributors guide*.

Core developers

Core developers are community members who have shown that they are dedicated to the continued development of the project through ongoing engagement with the community. They have shown they can be trusted to maintain scikit-learn with care. Being a core developer allows contributors to more easily carry on with their project related activities by giving them direct access to the project's repository and is represented as being an organization member on the scikit-learn GitHub organization. Core developers are expected to review code contributions, can merge approved pull requests, can cast votes for and against merging a pull-request, and can be involved in deciding major changes to the API.

New core developers can be nominated by any existing core developers. Once they have been nominated, there will be a vote by the current core developers. Voting on new core developers is one of the few activities that takes place on the project's private management list. While it is expected that most votes will be unanimous, a two-thirds majority of the cast votes is enough. The vote needs to be open for at least 1 week.

Core developers that have not contributed to the project (commits or GitHub comments) in the past 12 months will be asked if they want to become emeritus core developers and recant their commit and voting rights until they become active again. The list of core developers, active and emeritus (with dates at which they became active) is public on the scikit-learn website.

Technical Committee

The Technical Committee (TC) members are core developers who have additional responsibilities to ensure the smooth running of the project. TC members are expected to participate in strategic planning, and approve changes to the governance model. The purpose of the TC is to ensure a smooth progress from the big-picture perspective. Indeed changes that impact the full project require a synthetic analysis and a consensus that is both explicit and informed. In cases that the core developer community (which includes the TC members) fails to reach such a consensus in the required time frame, the TC is the entity to resolve the issue. Membership of the TC is by nomination by a core developer. A nomination will result in discussion which cannot take more than a month and then a vote by the core developers which will stay open for a week. TC membership votes are subject to a two-third majority of all cast votes as well as a simple majority approval of all the current TC members. TC members who do not actively engage with the TC duties are expected to resign.

The initial Technical Committee of scikit-learn consists of Alexandre Gramfort, Olivier Grisel, Andreas Müller, Joel Nothman, Hanmin Qin, Gaël Varoquaux, and Roman Yurchak.

1.9.2 Decision Making Process

Decisions about the future of the project are made through discussion with all members of the community. All non-sensitive project management discussion takes place on the project contributors' mailing list and the issue tracker. Occasionally, sensitive discussion occurs on a private list.

Scikit-learn uses a "consensus seeking" process for making decisions. The group tries to find a resolution that has no open objections among core developers. At any point during the discussion, any core-developer can call for a vote, which will conclude one month from the call for the vote. Any vote must be backed by a `SLEP <slep>`. If no option can gather two thirds of the votes cast, the decision is escalated to the TC, which in turn will use consensus seeking with the fallback option of a simple majority vote if no consensus can be found within a month. This is what we hereafter may refer to as "the decision making process".

Decisions (in addition to adding core developers and TC membership as above) are made according to the following rules:

- **Minor Documentation changes**, such as typo fixes, or addition / correction of a sentence, but no change of the scikit-learn.org landing page or the "about" page: Requires +1 by a core developer, no -1 by a core developer (lazy consensus), happens on the issue or pull request page. Core developers are expected to give "reasonable time" to others to give their opinion on the pull request if they're not confident others would agree.
- **Code changes and major documentation changes** require +1 by two core developers, no -1 by a core developer (lazy consensus), happens on the issue of pull-request page.
- **Changes to the API principles and changes to dependencies or supported versions** happen via a *Enhancement proposals (SLEPs)* and follows the decision-making process outlined above.
- **Changes to the governance model** use the same decision process outlined above.

If a veto -1 vote is cast on a lazy consensus, the proposer can appeal to the community and core developers and the change can be approved or rejected using the decision making procedure outlined above.

1.9.3 Enhancement proposals (SLEPs)

For all votes, a proposal must have been made public and discussed before the vote. Such proposal must be a consolidated document, in the form of a 'Scikit-Learn Enhancement Proposal' (SLEP), rather than a long discussion on an issue. A SLEP must be submitted as a pull-request to enhancement proposals using the SLEP template.

CHAPTER

TWO

SCIKIT-LEARN TUTORIALS

2.1 An introduction to machine learning with scikit-learn

Section contents

In this section, we introduce the machine learning vocabulary that we use throughout scikit-learn and give a simple learning example.

2.1.1 Machine learning: the problem setting

In general, a learning problem considers a set of n samples of data and then tries to predict properties of unknown data. If each sample is more than a single number and, for instance, a multi-dimensional entry (aka multivariate data), it is said to have several attributes or **features**.

Learning problems fall into a few categories:

- supervised learning, in which the data comes with additional attributes that we want to predict (*Click here* to go to the scikit-learn supervised learning page).This problem can be either:
  - classification: samples belong to two or more classes and we want to learn from already labeled data how to predict the class of unlabeled data. An example of a classification problem would be handwritten digit recognition, in which the aim is to assign each input vector to one of a finite number of discrete categories. Another way to think of classification is as a discrete (as opposed to continuous) form of supervised learning where one has a limited number of categories and for each of the n samples provided, one is to try to label them with the correct category or class.
  - regression: if the desired output consists of one or more continuous variables, then the task is called *regression*. An example of a regression problem would be the prediction of the length of a salmon as a function of its age and weight.
- unsupervised learning, in which the training data consists of a set of input vectors x without any corresponding target values. The goal in such problems may be to discover groups of similar examples within the data, where it is called clustering, or to determine the distribution of data within the input space, known as density estimation, or to project the data from a high-dimensional space down to two or three dimensions for the purpose of *visualization* (*Click here* to go to the Scikit-Learn unsupervised learning page).

Training set and testing set

Machine learning is about learning some properties of a data set and then testing those properties against another data set. A common practice in machine learning is to evaluate an algorithm by splitting a data set into two. We call one of those sets the **training set**, on which we learn some properties; we call the other set the **testing set**, on which we test the learned properties.

2.1.2 Loading an example dataset

`scikit-learn` comes with a few standard datasets, for instance the iris and digits datasets for classification and the boston house prices dataset for regression.

In the following, we start a Python interpreter from our shell and then load the `iris` and `digits` datasets. Our notational convention is that `$` denotes the shell prompt while `>>>` denotes the Python interpreter prompt:

```
$ python
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> digits = datasets.load_digits()
```

A dataset is a dictionary-like object that holds all the data and some metadata about the data. This data is stored in the `.data` member, which is a `n_samples, n_features` array. In the case of supervised problem, one or more response variables are stored in the `.target` member. More details on the different datasets can be found in the *dedicated section*.

For instance, in the case of the digits dataset, `digits.data` gives access to the features that can be used to classify the digits samples:

```
>>> print(digits.data)
[[ 0.   0.   5. ...   0.   0.   0.]
 [ 0.   0.   0. ...  10.   0.   0.]
 [ 0.   0.   0. ...  16.   9.   0.]
 ...
 [ 0.   0.   1. ...   6.   0.   0.]
 [ 0.   0.   2. ...  12.   0.   0.]
 [ 0.   0.  10. ...  12.   1.   0.]]
```

and `digits.target` gives the ground truth for the digit dataset, that is the number corresponding to each digit image that we are trying to learn:

```
>>> digits.target
array([0, 1, 2, ..., 8, 9, 8])
```

Shape of the data arrays

The data is always a 2D array, shape `(n_samples, n_features)`, although the original data may have had a different shape. In the case of the digits, each original sample is an image of shape `(8, 8)` and can be accessed using:

```
>>> digits.images[0]
array([[  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.],
       [  0.,   0.,  13.,  15.,  10.,  15.,   5.,   0.],
       [  0.,   3.,  15.,   2.,   0.,  11.,   8.,   0.],
       [  0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.],
       [  0.,   5.,   8.,   0.,   0.,   9.,   8.,   0.],
       [  0.,   4.,  11.,   0.,   1.,  12.,   7.,   0.],
       [  0.,   2.,  14.,   5.,  10.,  12.,   0.,   0.],
       [  0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.]])
```

The *simple example on this dataset* illustrates how starting from the original problem one can shape the data for consumption in scikit-learn.

Loading from external datasets

To load from an external dataset, please refer to *loading external datasets*.

2.1.3 Learning and predicting

In the case of the digits dataset, the task is to predict, given an image, which digit it represents. We are given samples of each of the 10 possible classes (the digits zero through nine) on which we *fit* an estimator to be able to *predict* the classes to which unseen samples belong.

In scikit-learn, an estimator for classification is a Python object that implements the methods `fit(X, y)` and `predict(T)`.

An example of an estimator is the class `sklearn.svm.SVC`, which implements support vector classification. The estimator's constructor takes as arguments the model's parameters.

For now, we will consider the estimator as a black box:

```
>>> from sklearn import svm
>>> clf = svm.SVC(gamma=0.001, C=100.)
```

Choosing the parameters of the model

In this example, we set the value of `gamma` manually. To find good values for these parameters, we can use tools such as *grid search* and *cross validation*.

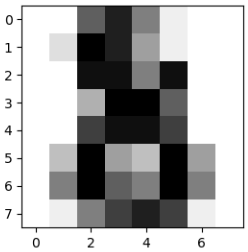
The `clf` (for classifier) estimator instance is first fitted to the model; that is, it must *learn* from the model. This is done by passing our training set to the `fit` method. For the training set, we'll use all the images from our dataset, except for the last image, which we'll reserve for our predicting. We select the training set with the `[:-1]` Python syntax, which produces a new array that contains all but the last item from `digits.data`:

```
>>> clf.fit(digits.data[:-1], digits.target[:-1])
SVC(C=100.0, gamma=0.001)
```

Now you can *predict* new values. In this case, you'll predict using the last image from `digits.data`. By predicting, you'll determine the image from the training set that best matches the last image.

```
>>> clf.predict(digits.data[-1:])
array([8])
```

The corresponding image is:



As you can see, it is a challenging task: after all, the images are of poor resolution. Do you agree with the classifier?

A complete example of this classification problem is available as an example that you can run and study: *Recognizing hand-written digits*.

2.1.4 Model persistence

It is possible to save a model in scikit-learn by using Python's built-in persistence model, pickle:

```
>>> from sklearn import svm
>>> from sklearn import datasets
>>> clf = svm.SVC()
>>> X, y = datasets.load_iris(return_X_y=True)
>>> clf.fit(X, y)
SVC()

>>> import pickle
>>> s = pickle.dumps(clf)
>>> clf2 = pickle.loads(s)
>>> clf2.predict(X[0:1])
array([0])
>>> y[0]
0
```

In the specific case of scikit-learn, it may be more interesting to use joblib's replacement for pickle (`joblib.dump` & `joblib.load`), which is more efficient on big data but it can only pickle to the disk and not to a string:

```
>>> from joblib import dump, load
>>> dump(clf, 'filename.joblib')
```

Later, you can reload the pickled model (possibly in another Python process) with:

```
>>> clf = load('filename.joblib')
```

Note: `joblib.dump` and `joblib.load` functions also accept file-like object instead of filenames. More information on data persistence with Joblib is available here.

Note that pickle has some security and maintainability issues. Please refer to section *Model persistence* for more detailed information about model persistence with scikit-learn.

2.1.5 Conventions

scikit-learn estimators follow certain rules to make their behavior more predictive. These are described in more detail in the *Glossary of Common Terms and API Elements*.

Type casting

Unless otherwise specified, input will be cast to `float64`:

```
>>> import numpy as np
>>> from sklearn import random_projection

>>> rng = np.random.RandomState(0)
>>> X = rng.rand(10, 2000)
>>> X = np.array(X, dtype='float32')
>>> X.dtype
dtype('float32')

>>> transformer = random_projection.GaussianRandomProjection()
>>> X_new = transformer.fit_transform(X)
>>> X_new.dtype
dtype('float64')
```

In this example, `X` is `float32`, which is cast to `float64` by `fit_transform(X)`.

Regression targets are cast to `float64` and classification targets are maintained:

```
>>> from sklearn import datasets
>>> from sklearn.svm import SVC
>>> iris = datasets.load_iris()
>>> clf = SVC()
>>> clf.fit(iris.data, iris.target)
SVC()

>>> list(clf.predict(iris.data[:3]))
[0, 0, 0]

>>> clf.fit(iris.data, iris.target_names[iris.target])
SVC()

>>> list(clf.predict(iris.data[:3]))
['setosa', 'setosa', 'setosa']
```

Here, the first `predict()` returns an integer array, since `iris.target` (an integer array) was used in `fit`. The second `predict()` returns a string array, since `iris.target_names` was for fitting.

Refitting and updating parameters

Hyper-parameters of an estimator can be updated after it has been constructed via the *set_params()* method. Calling `fit()` more than once will overwrite what was learned by any previous `fit()`:

```
>>> import numpy as np
>>> from sklearn.datasets import load_iris
>>> from sklearn.svm import SVC
>>> X, y = load_iris(return_X_y=True)

>>> clf = SVC()
>>> clf.set_params(kernel='linear').fit(X, y)
SVC(kernel='linear')
>>> clf.predict(X[:5])
array([0, 0, 0, 0, 0])
```

(continues on next page)

(continued from previous page)

```
>>> clf.set_params(kernel='rbf').fit(X, y)
SVC()
>>> clf.predict(X[:5])
array([0, 0, 0, 0, 0])
```

Here, the default kernel `rbf` is first changed to `linear` via *SVC.set_params()* after the estimator has been constructed, and changed back to `rbf` to refit the estimator and to make a second prediction.

Multiclass vs. multilabel fitting

When using *multiclass classifiers*, the learning and prediction task that is performed is dependent on the format of the target data fit upon:

```
>>> from sklearn.svm import SVC
>>> from sklearn.multiclass import OneVsRestClassifier
>>> from sklearn.preprocessing import LabelBinarizer

>>> X = [[1, 2], [2, 4], [4, 5], [3, 2], [3, 1]]
>>> y = [0, 0, 1, 1, 2]

>>> classif = OneVsRestClassifier(estimator=SVC(random_state=0))
>>> classif.fit(X, y).predict(X)
array([0, 0, 1, 1, 2])
```

In the above case, the classifier is fit on a 1d array of multiclass labels and the `predict()` method therefore provides corresponding multiclass predictions. It is also possible to fit upon a 2d array of binary label indicators:

```
>>> y = LabelBinarizer().fit_transform(y)
>>> classif.fit(X, y).predict(X)
array([[1, 0, 0],
       [1, 0, 0],
       [0, 1, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

Here, the classifier is `fit()` on a 2d binary label representation of `y`, using the *LabelBinarizer*. In this case `predict()` returns a 2d array representing the corresponding multilabel predictions.

Note that the fourth and fifth instances returned all zeroes, indicating that they matched none of the three labels `fit` upon. With multilabel outputs, it is similarly possible for an instance to be assigned multiple labels:

```
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> y = [[0, 1], [0, 2], [1, 3], [0, 2, 3], [2, 4]]
>>> y = MultiLabelBinarizer().fit_transform(y)
>>> classif.fit(X, y).predict(X)
array([[1, 1, 0, 0, 0],
       [1, 0, 1, 0, 0],
       [0, 1, 0, 1, 0],
       [1, 0, 1, 0, 0],
       [1, 0, 1, 0, 0]])
```

In this case, the classifier is fit upon instances each assigned multiple labels. The *MultiLabelBinarizer* is used to binarize the 2d array of multilabels to `fit` upon. As a result, `predict()` returns a 2d array with multiple predicted labels for each instance.

2.2 A tutorial on statistical-learning for scientific data processing

Statistical learning

Machine learning is a technique with a growing importance, as the size of the datasets experimental sciences are facing is rapidly growing. Problems it tackles range from building a prediction function linking different observations, to classifying observations, or learning the structure in an unlabeled dataset.

This tutorial will explore *statistical learning*, the use of machine learning techniques with the goal of statistical inference: drawing conclusions on the data at hand.

Scikit-learn is a Python module integrating classic machine learning algorithms in the tightly-knit world of scientific Python packages (NumPy, SciPy, matplotlib).

2.2.1 Statistical learning: the setting and the estimator object in scikit-learn

Datasets

Scikit-learn deals with learning information from one or more datasets that are represented as 2D arrays. They can be understood as a list of multi-dimensional observations. We say that the first axis of these arrays is the **samples** axis, while the second is the **features** axis.

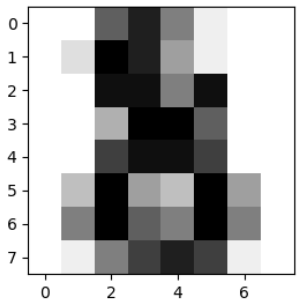
A simple example shipped with scikit-learn: iris dataset

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> data = iris.data
>>> data.shape
(150, 4)
```

It is made of 150 observations of irises, each described by 4 features: their sepal and petal length and width, as detailed in `iris.DESCR`.

When the data is not initially in the `(n_samples, n_features)` shape, it needs to be preprocessed in order to be used by scikit-learn.

An example of reshaping data would be the digits dataset



The digits dataset is made of 1797 8x8 images of hand-written digits

```
>>> digits = datasets.load_digits()
>>> digits.images.shape
(1797, 8, 8)
>>> import matplotlib.pyplot as plt
>>> plt.imshow(digits.images[-1], cmap=plt.cm.gray_r)
<matplotlib.image.AxesImage object at ...>
```

To use this dataset with scikit-learn, we transform each 8x8 image into a feature vector of length 64

```
>>> data = digits.images.reshape((digits.images.shape[0], -1))
```

Estimators objects

Fitting data: the main API implemented by scikit-learn is that of the `estimator`. An estimator is any object that learns from data; it may be a classification, regression or clustering algorithm or a *transformer* that extracts/filters useful features from raw data.

All estimator objects expose a `fit` method that takes a dataset (usually a 2-d array):

```
>>> estimator.fit(data)
```

Estimator parameters: All the parameters of an estimator can be set when it is instantiated or by modifying the corresponding attribute:

```
>>> estimator = Estimator(param1=1, param2=2)
>>> estimator.param1
1
```

Estimated parameters: When data is fitted with an estimator, parameters are estimated from the data at hand. All the estimated parameters are attributes of the estimator object ending by an underscore:

```
>>> estimator.estimated_param_
```

2.2.2 Supervised learning: predicting an output variable from high-dimensional observations

The problem solved in supervised learning

Supervised learning consists in learning the link between two datasets: the observed data `X` and an external variable `y` that we are trying to predict, usually called "target" or "labels". Most often, `y` is a 1D array of length `n_samples`.

All supervised estimators in scikit-learn implement a `fit(X, y)` method to fit the model and a `predict(X)` method that, given unlabeled observations `X`, returns the predicted labels `y`.

Vocabulary: classification and regression

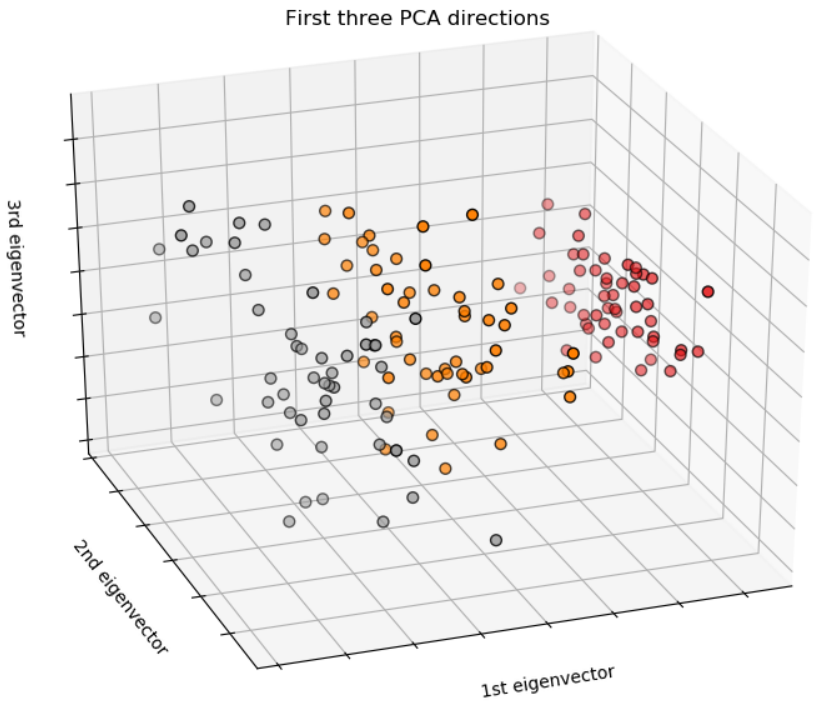
If the prediction task is to classify the observations in a set of finite labels, in other words to "name" the objects observed, the task is said to be a **classification** task. On the other hand, if the goal is to predict a continuous target variable, it is said to be a **regression** task.

When doing classification in scikit-learn, `y` is a vector of integers or strings.

Note: See the *Introduction to machine learning with scikit-learn Tutorial* for a quick run-through on the basic machine learning vocabulary used within scikit-learn.

Nearest neighbor and the curse of dimensionality

Classifying irises:



The iris dataset is a classification task consisting in identifying 3 different types of irises (Setosa, Versicolour, and Virginica) from their petal and sepal length and width:

```
>>> import numpy as np
>>> from sklearn import datasets
>>> iris_X, iris_y = datasets.load_iris(return_X_y=True)
>>> np.unique(iris_y)
array([0, 1, 2])
```

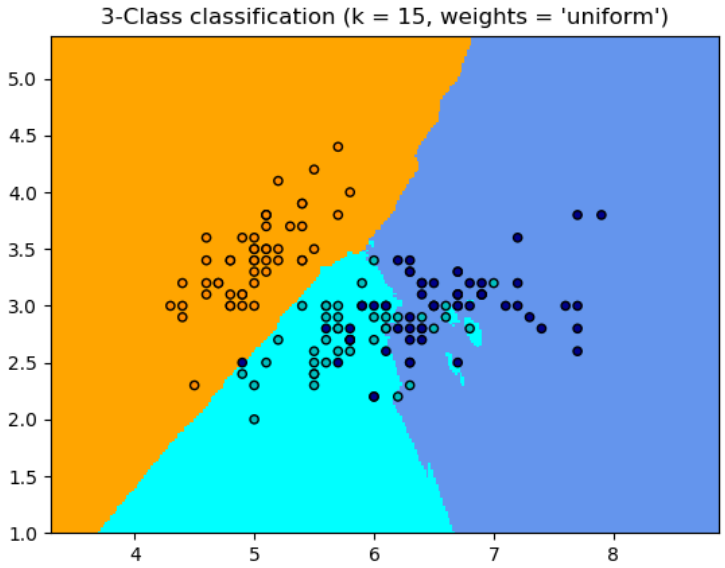
k-Nearest neighbors classifier

The simplest possible classifier is the nearest neighbor: given a new observation `X_test`, find in the training set (i.e. the data used to train the estimator) the observation with the closest feature vector. (Please see the *Nearest Neighbors section* of the online Scikit-learn documentation for more information about this type of classifier.)

Training set and testing set

While experimenting with any learning algorithm, it is important not to test the prediction of an estimator on the data used to fit the estimator as this would not be evaluating the performance of the estimator on **new data**. This is why datasets are often split into *train* and *test* data.

KNN (k nearest neighbors) classification example:




```
>>> # Split iris data in train and test data
>>> # A random permutation, to split the data randomly
>>> np.random.seed(0)
>>> indices = np.random.permutation(len(iris_X))
>>> iris_X_train = iris_X[indices[:-10]]
>>> iris_y_train = iris_y[indices[:-10]]
>>> iris_X_test = iris_X[indices[-10:]]
>>> iris_y_test = iris_y[indices[-10:]]
>>> # Create and fit a nearest-neighbor classifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier()
>>> knn.fit(iris_X_train, iris_y_train)
KNeighborsClassifier()
>>> knn.predict(iris_X_test)
array([1, 2, 1, 0, 0, 0, 2, 1, 2, 0])
>>> iris_y_test
array([1, 1, 1, 0, 0, 0, 2, 1, 2, 0])
```

The curse of dimensionality

For an estimator to be effective, you need the distance between neighboring points to be less than some value d, which depends on the problem. In one dimension, this requires on average $n \sim 1/d$ points. In the context of the above k-NN example, if the data is described by just one feature with values ranging from 0 to 1 and with n training observations, then new data will be no further away than $1/n$. Therefore, the nearest neighbor decision rule will be efficient as soon

as $1/n$ is small compared to the scale of between-class feature variations.

If the number of features is p, you now require $n \sim 1/d^p$ points. Let's say that we require 10 points in one dimension: now 10^p points are required in p dimensions to pave the $[0, 1]$ space. As p becomes large, the number of training points required for a good estimator grows exponentially.

For example, if each point is just a single number (8 bytes), then an effective k-NN estimator in a paltry $p \sim 20$ dimensions would require more training data than the current estimated size of the entire internet ($\pm$1000 Exabytes or so).

This is called the curse of dimensionality and is a core problem that machine learning addresses.

Linear model: from regression to sparsity

Diabetes dataset

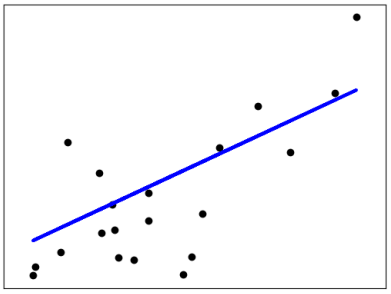
The diabetes dataset consists of 10 physiological variables (age, sex, weight, blood pressure) measure on 442 patients, and an indication of disease progression after one year:

```
>>> diabetes_X, diabetes_y = datasets.load_diabetes(return_X_y=True)
>>> diabetes_X_train = diabetes_X[:-20]
>>> diabetes_X_test  = diabetes_X[-20:]
>>> diabetes_y_train = diabetes_y[:-20]
>>> diabetes_y_test  = diabetes_y[-20:]
```

The task at hand is to predict disease progression from physiological variables.

Linear regression

`LinearRegression`, in its simplest form, fits a linear model to the data set by adjusting a set of parameters in order to make the sum of the squared residuals of the model as small as possible.



Linear models: $y = X\beta + \epsilon$

- X: data
- y: target variable
- β: Coefficients
- ϵ: Observation noise

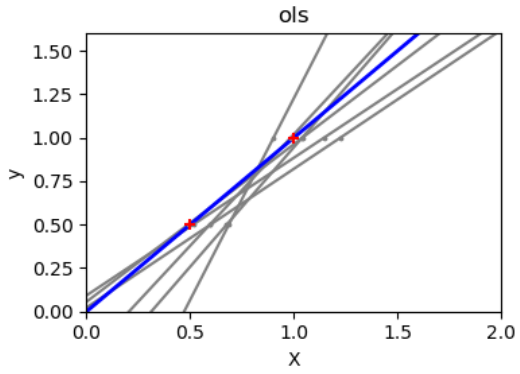
```
>>> from sklearn import linear_model
>>> regr = linear_model.LinearRegression()
>>> regr.fit(diabetes_X_train, diabetes_y_train)
LinearRegression()
>>> print(regr.coef_)
[   0.30349955 -237.63931533  510.53060544  327.73698041 -814.13170937
  492.81458798  102.84845219  184.60648906  743.51961675   76.09517222]


>>> # The mean square error
>>> np.mean((regr.predict(diabetes_X_test) - diabetes_y_test)**2)
2004.56760268...

>>> # Explained variance score: 1 is perfect prediction
>>> # and 0 means that there is no linear relationship
>>> # between X and y.
>>> regr.score(diabetes_X_test, diabetes_y_test)
0.5850753022690...
```

Shrinkage

If there are few data points per dimension, noise in the observations induces high variance:

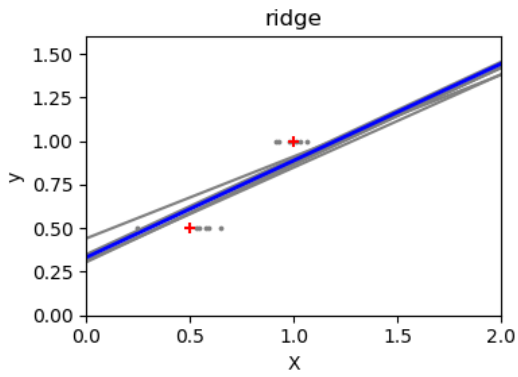



```
>>> X = np.c_[ .5, 1].T
>>> y = [.5, 1]
>>> test = np.c_[ 0, 2].T
>>> regr = linear_model.LinearRegression()

>>> import matplotlib.pyplot as plt
>>> plt.figure()

>>> np.random.seed(0)
>>> for _ in range(6):
...     this_X = .1 * np.random.normal(size=(2, 1)) + X
...     regr.fit(this_X, y)
...     plt.plot(test, regr.predict(test))
...     plt.scatter(this_X, y, s=3)
```

A solution in high-dimensional statistical learning is to *shrink* the regression coefficients to zero: any two randomly chosen set of observations are likely to be uncorrelated. This is called `Ridge` regression:




```
>>> regr = linear_model.Ridge(alpha=.1)

>>> plt.figure()

>>> np.random.seed(0)
>>> for _ in range(6):
...     this_X = .1 * np.random.normal(size=(2, 1)) + X
...     regr.fit(this_X, y)
...     plt.plot(test, regr.predict(test))
...     plt.scatter(this_X, y, s=3)
```

This is an example of **bias/variance tradeoff**: the larger the ridge `alpha` parameter, the higher the bias and the lower the variance.

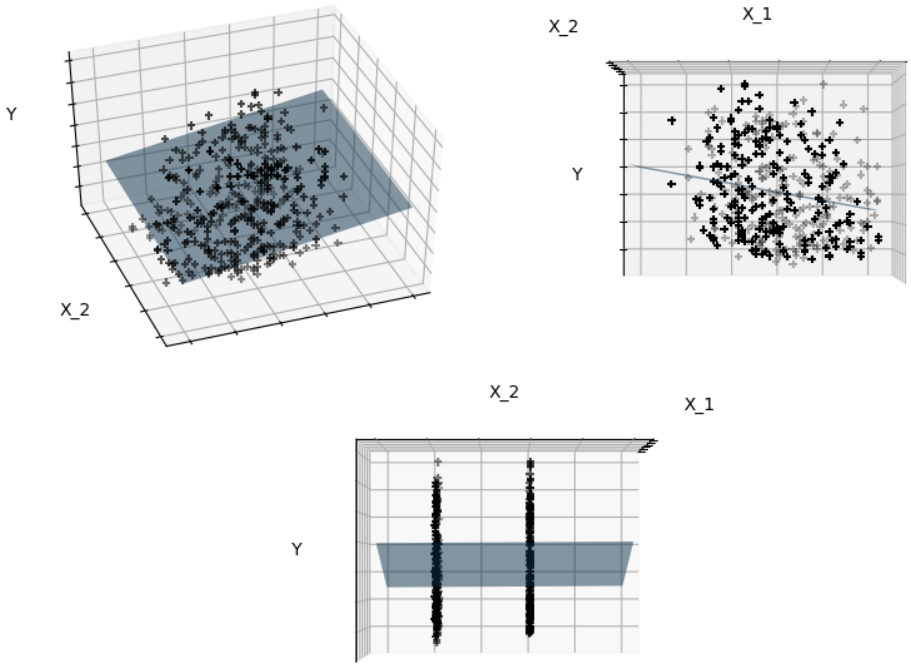
We can choose `alpha` to minimize left out error, this time using the diabetes dataset rather than our synthetic data:

```
>>> alphas = np.logspace(-4, -1, 6)
>>> print([regr.set_params(alpha=alpha)
...             .fit(diabetes_X_train, diabetes_y_train)
...             .score(diabetes_X_test, diabetes_y_test)
...         for alpha in alphas])
[0.5851110683883..., 0.5852073015444..., 0.5854677540698...,
 0.5855512036503..., 0.5830717085554..., 0.57058999437...]
```

Note: Capturing in the fitted parameters noise that prevents the model to generalize to new data is called overfitting. The bias introduced by the ridge regression is called a regularization.

Sparsity

Fitting only features 1 and 2



Note: A representation of the full diabetes dataset would involve 11 dimensions (10 feature dimensions and one of the target variable). It is hard to develop an intuition on such representation, but it may be useful to keep in mind that it would be a fairly *empty* space.

We can see that, although feature 2 has a strong coefficient on the full model, it conveys little information on `y` when considered with feature 1.

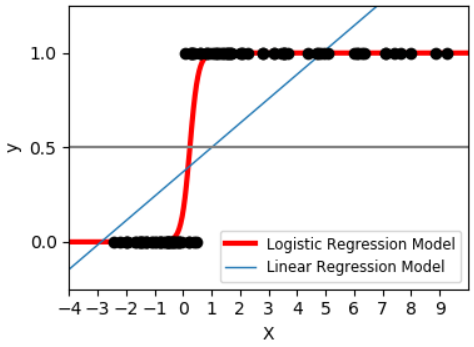
To improve the conditioning of the problem (i.e. mitigating the *The curse of dimensionality*), it would be interesting to select only the informative features and set non-informative ones, like feature 2 to 0. Ridge regression will decrease their contribution, but not set them to zero. Another penalization approach, called *Lasso* (least absolute shrinkage and selection operator), can set some coefficients to zero. Such methods are called **sparse method** and sparsity can be seen as an application of Occam's razor: *prefer simpler models*.

```
>>> regr = linear_model.Lasso()
>>> scores = [regr.set_params(alpha=alpha)
...               .fit(diabetes_X_train, diabetes_y_train)
...               .score(diabetes_X_test, diabetes_y_test)
...           for alpha in alphas]
>>> best_alpha = alphas[scores.index(max(scores))]
>>> regr.alpha = best_alpha
>>> regr.fit(diabetes_X_train, diabetes_y_train)
Lasso(alpha=0.025118864315095794)
>>> print(regr.coef_)
[   0.         -212.43764548  517.19478111  313.77959962 -160.8303982    -0.
 -187.19554705   69.38229038  508.66011217   71.84239008]
```

Different algorithms for the same problem

Different algorithms can be used to solve the same mathematical problem. For instance the `Lasso` object in scikit-learn solves the lasso regression problem using a coordinate descent method, that is efficient on large datasets. However, scikit-learn also provides the *LassoLars* object using the *LARS* algorithm, which is very efficient for problems in which the weight vector estimated is very sparse (i.e. problems with very few observations).

Classification

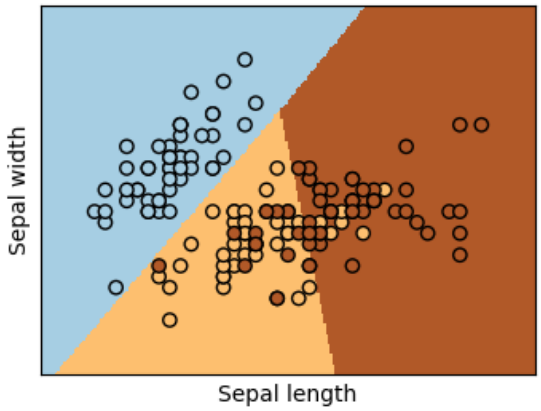


For classification, as in the labeling iris task, linear regression is not the right approach as it will give too much weight to data far from the decision frontier. A linear approach is to fit a sigmoid function or **logistic** function:

$$y = \text{sigmoid}(X\beta - \text{offset}) + \epsilon = \frac{1}{1 + \exp(-X\beta + \text{offset})} + \epsilon$$

```
>>> log = linear_model.LogisticRegression(C=1e5)
>>> log.fit(iris_X_train, iris_y_train)
LogisticRegression(C=100000.0)
```

This is known as *LogisticRegression*.



Multiclass classification

If you have several classes to predict, an option often used is to fit one-versus-all classifiers and then use a voting heuristic for the final decision.

Shrinkage and sparsity with logistic regression

The `C` parameter controls the amount of regularization in the *LogisticRegression* object: a large value for `C` results in less regularization. `penalty="l2"` gives *Shrinkage* (i.e. non-sparse coefficients), while `penalty="l1"` gives *Sparsity*.

Exercise

Try classifying the digits dataset with nearest neighbors and a linear model. Leave out the last 10% and test prediction performance on these observations.

```
from sklearn import datasets, neighbors, linear_model

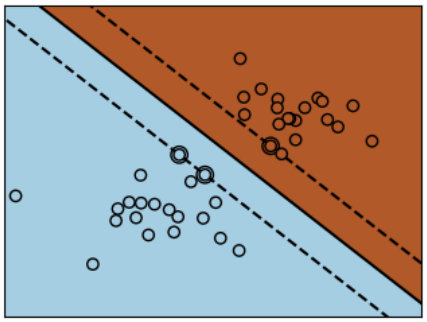
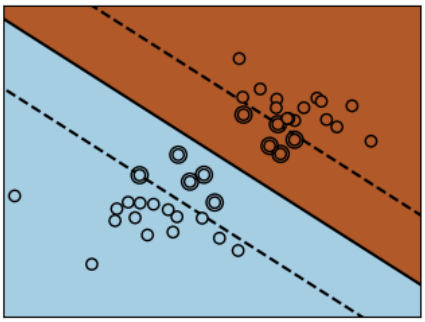
X_digits, y_digits = datasets.load_digits(return_X_y=True)
X_digits = X_digits / X_digits.max()
```

Solution: `../../auto_examples/exercises/plot_digits_classification_exercise.py`

Support vector machines (SVMs)

Linear SVMs

Support Vector Machines belong to the discriminant model family: they try to find a combination of samples to build a plane maximizing the margin between the two classes. Regularization is set by the `C` parameter: a small value for `C` means the margin is calculated using many or all of the observations around the separating line (more regularization); a large value for `C` means the margin is calculated on observations close to the separating line (less regularization).

Unregularized SVM	Regularized SVM (default)
	

Example:

- *Plot different SVM classifiers in the iris dataset*

SVMs can be used in regression –*SVR* (Support Vector Regression)–, or in classification –*SVC* (Support Vector Classification).

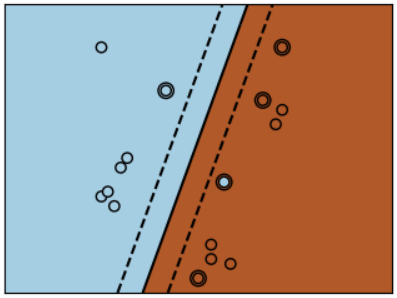
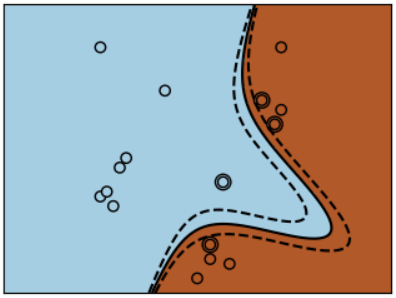
```
>>> from sklearn import svm
>>> svc = svm.SVC(kernel='linear')
>>> svc.fit(iris_X_train, iris_y_train)
SVC(kernel='linear')
```

Warning: Normalizing data

For many estimators, including the SVMs, having datasets with unit standard deviation for each feature is important to get good prediction.

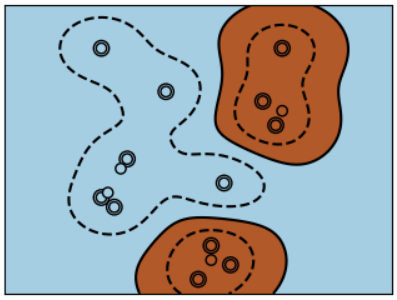
Using kernels

Classes are not always linearly separable in feature space. The solution is to build a decision function that is not linear but may be polynomial instead. This is done using the *kernel trick* that can be seen as creating a decision energy by positioning *kernels* on observations:

Linear kernel	Polynomial kernel
	

`>>> svc = svm.SVC(kernel='linear')`	`>>> svc = svm.SVC(kernel='poly',`
	`...  degree=3)`
	`>>> # degree: polynomial degree`

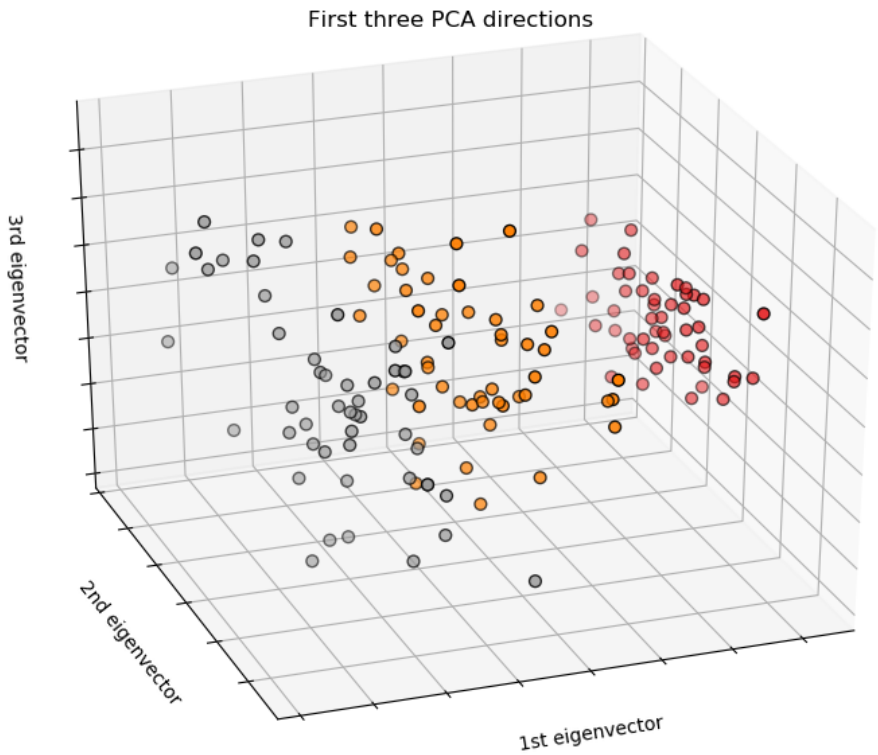
RBF kernel (Radial Basis Function)



```
>>> svc = svm.SVC(kernel='rbf')
>>> # gamma: inverse of size of
>>> # radial kernel
```

Interactive example

See the *SVM GUI* to download `svm_gui.py`; add data points of both classes with right and left button, fit the model and change parameters and data.



Exercise

Try classifying classes 1 and 2 from the iris dataset with SVMs, with the 2 first features. Leave out 10% of each class and test prediction performance on these observations.

Warning: the classes are ordered, do not leave out the last 10%, you would be testing on only one class.

Hint: You can use the `decision_function` method on a grid to get intuitions.

```
iris = datasets.load_iris()
X = iris.data
y = iris.target

X = X[y != 0, :2]
y = y[y != 0]
```

Solution: `../../auto_examples/exercises/plot_iris_exercise.py`

2.2.3 Model selection: choosing estimators and their parameters

Score, and cross-validated scores

As we have seen, every estimator exposes a `score` method that can judge the quality of the fit (or the prediction) on new data. **Bigger is better**.

```
>>> from sklearn import datasets, svm
>>> X_digits, y_digits = datasets.load_digits(return_X_y=True)
>>> svc = svm.SVC(C=1, kernel='linear')
>>> svc.fit(X_digits[:-100], y_digits[:-100]).score(X_digits[-100:], y_digits[-100:])
0.98
```

To get a better measure of prediction accuracy (which we can use as a proxy for goodness of fit of the model), we can successively split the data in *folds* that we use for training and testing:

```
>>> import numpy as np
>>> X_folds = np.array_split(X_digits, 3)
>>> y_folds = np.array_split(y_digits, 3)
>>> scores = list()
>>> for k in range(3):
...     # We use 'list' to copy, in order to 'pop' later on
...     X_train = list(X_folds)
...     X_test = X_train.pop(k)
...     X_train = np.concatenate(X_train)
...     y_train = list(y_folds)
...     y_test = y_train.pop(k)
...     y_train = np.concatenate(y_train)
...     scores.append(svc.fit(X_train, y_train).score(X_test, y_test))
>>> print(scores)
[0.934..., 0.956..., 0.939...]
```

This is called a *KFold* cross-validation.

Cross-validation generators

Scikit-learn has a collection of classes which can be used to generate lists of train/test indices for popular cross-validation strategies.

They expose a `split` method which accepts the input dataset to be split and yields the train/test set indices for each iteration of the chosen cross-validation strategy.

This example shows an example usage of the `split` method.

```
>>> from sklearn.model_selection import KFold, cross_val_score
>>> X = ["a", "a", "a", "b", "b", "c", "c", "c", "c", "c"]
>>> k_fold = KFold(n_splits=5)
>>> for train_indices, test_indices in k_fold.split(X):
...      print('Train: %s | test: %s' % (train_indices, test_indices))
Train: [2 3 4 5 6 7 8 9] | test: [0 1]
Train: [0 1 4 5 6 7 8 9] | test: [2 3]
Train: [0 1 2 3 6 7 8 9] | test: [4 5]
Train: [0 1 2 3 4 5 8 9] | test: [6 7]
Train: [0 1 2 3 4 5 6 7] | test: [8 9]
```

The cross-validation can then be performed easily:

```
>>> [svc.fit(X_digits[train], y_digits[train]).score(X_digits[test], y_digits[test])
...  for train, test in k_fold.split(X_digits)]
[0.963..., 0.922..., 0.963..., 0.963..., 0.930...]
```

The cross-validation score can be directly calculated using the *cross_val_score* helper. Given an estimator, the cross-validation object and the input dataset, the *cross_val_score* splits the data repeatedly into a training and a testing set, trains the estimator using the training set and computes the scores based on the testing set for each iteration of cross-validation.

By default the estimator's `score` method is used to compute the individual scores.

Refer the *metrics module* to learn more on the available scoring methods.

```
>>> cross_val_score(svc, X_digits, y_digits, cv=k_fold, n_jobs=-1)
array([0.96388889, 0.92222222, 0.9637883 , 0.9637883 , 0.93036212])
```

`n_jobs=-1` means that the computation will be dispatched on all the CPUs of the computer.

Alternatively, the `scoring` argument can be provided to specify an alternative scoring method.

```
>>> cross_val_score(svc, X_digits, y_digits, cv=k_fold,
...                 scoring='precision_macro')
array([0.96578289, 0.92708922, 0.96681476, 0.96362897, 0.93192644])
```

Cross-validation generators

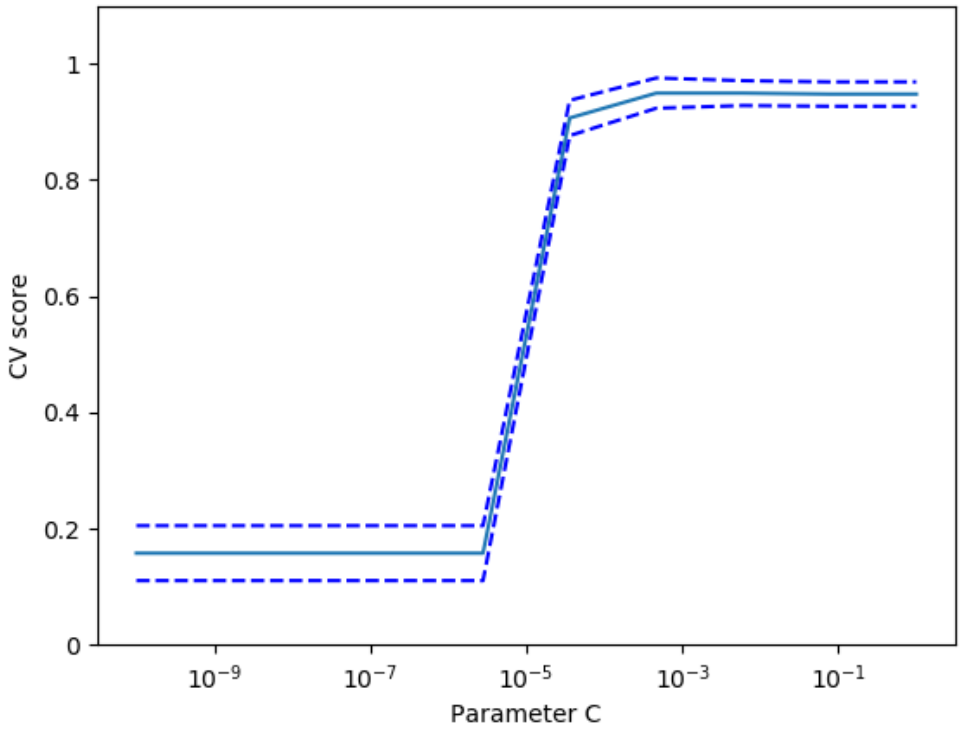
KFold **(n_splits, shuffle, random_state)**	*StratifiedKFold* **(n_splits, shuffle, random_state)**	*GroupKFold* **(n_splits)**
Splits it into K folds, trains on K-1 and then tests on the left-out.	Same as K-Fold but preserves the class distribution within each fold.	Ensures that the same group is not in both testing and training sets.

ShuffleSplit **(n_splits, test_size, train_size, random_state)**	*StratifiedShuffleSplit*	*GroupShuffleSplit*
Generates train/test indices based on random permutation.	Same as shuffle split but preserves the class distribution within each iteration.	Ensures that the same group is not in both testing and training sets.

LeaveOneGroupOut ()	*LeavePGroupsOut* (n_groups)	*LeaveOneOut* ()
Takes a group array to group observations.	Leave P groups out.	Leave one observation out.

LeavePOut (p)	*PredefinedSplit*
Leave P observations out.	Generates train/test indices based on predefined splits.

Exercise



On the digits dataset, plot the cross-validation score of a *SVC* estimator with an linear kernel as a function of parameter `C` (use a logarithmic grid of points, from 1 to 10).

```
import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn import datasets, svm

X, y = datasets.load_digits(return_X_y=True)

svc = svm.SVC(kernel='linear')
C_s = np.logspace(-10, 0, 10)

scores = list()
```

Solution: *Cross-validation on Digits Dataset Exercise*

Grid-search and cross-validated estimators

Grid-search

scikit-learn provides an object that, given data, computes the score during the fit of an estimator on a parameter grid and chooses the parameters to maximize the cross-validation score. This object takes an estimator during the construction and exposes an estimator API:

```
>>> from sklearn.model_selection import GridSearchCV, cross_val_score
>>> Cs = np.logspace(-6, -1, 10)
>>> clf = GridSearchCV(estimator=svc, param_grid=dict(C=Cs),
...                    n_jobs=-1)
>>> clf.fit(X_digits[:1000], y_digits[:1000])
GridSearchCV(cv=None,...
>>> clf.best_score_
0.925...
>>> clf.best_estimator_.C
0.0077...

>>> # Prediction performance on test set is not as good as on train set
>>> clf.score(X_digits[1000:], y_digits[1000:])
0.943...
```

By default, the *GridSearchCV* uses a 3-fold cross-validation. However, if it detects that a classifier is passed, rather than a regressor, it uses a stratified 3-fold. The default will change to a 5-fold cross-validation in version 0.22.

Nested cross-validation

```
>>> cross_val_score(clf, X_digits, y_digits)
array([0.938..., 0.963..., 0.944...])
```

Two cross-validation loops are performed in parallel: one by the *GridSearchCV* estimator to set `gamma` and the other one by `cross_val_score` to measure the prediction performance of the estimator. The resulting scores are unbiased estimates of the prediction score on new data.

Warning: You cannot nest objects with parallel computing (`n_jobs` different than 1).

Cross-validated estimators

Cross-validation to set a parameter can be done more efficiently on an algorithm-by-algorithm basis. This is why, for certain estimators, scikit-learn exposes *Cross-validation: evaluating estimator performance* estimators that set their parameter automatically by cross-validation:

```
>>> from sklearn import linear_model, datasets
>>> lasso = linear_model.LassoCV()
>>> X_diabetes, y_diabetes = datasets.load_diabetes(return_X_y=True)
>>> lasso.fit(X_diabetes, y_diabetes)
LassoCV()
>>> # The estimator chose automatically its lambda:
>>> lasso.alpha_
0.00375...
```

These estimators are called similarly to their counterparts, with 'CV' appended to their name.

Exercise

On the diabetes dataset, find the optimal regularization parameter alpha.

Bonus: How much can you trust the selection of alpha?

```
from sklearn import datasets
from sklearn.linear_model import LassoCV
from sklearn.linear_model import Lasso
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV

X, y = datasets.load_diabetes(return_X_y=True)
X = X[:150]
```

Solution: *Cross-validation on diabetes Dataset Exercise*

2.2.4 Unsupervised learning: seeking representations of the data

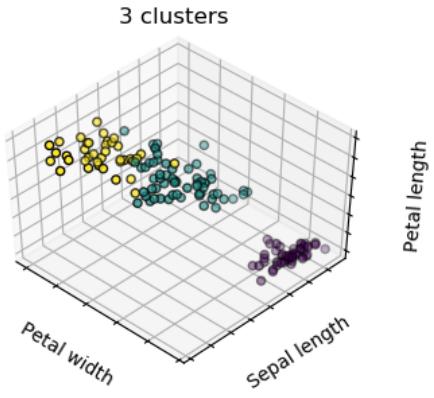
Clustering: grouping observations together

The problem solved in clustering

Given the iris dataset, if we knew that there were 3 types of iris, but did not have access to a taxonomist to label them: we could try a **clustering task**: split the observations into well-separated group called *clusters*.

K-means clustering

Note that there exist a lot of different clustering criteria and associated algorithms. The simplest clustering algorithm is *K-means*.




```
>>> from sklearn import cluster, datasets
>>> X_iris, y_iris = datasets.load_iris(return_X_y=True)

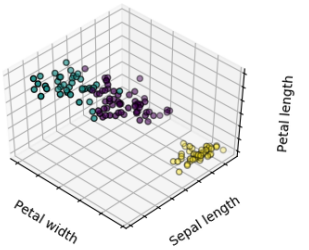
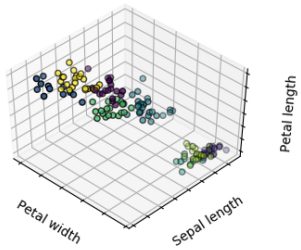
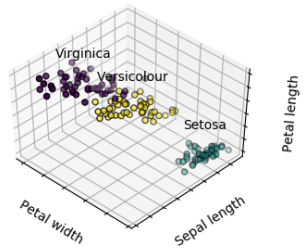
>>> k_means = cluster.KMeans(n_clusters=3)
>>> k_means.fit(X_iris)
KMeans(n_clusters=3)
```

(continues on next page)

(continued from previous page)

```
>>> print(k_means.labels_[::10])
[1 1 1 1 1 0 0 0 0 0 2 2 2 2 2]
>>> print(y_iris[::10])
[0 0 0 0 0 1 1 1 1 1 2 2 2 2 2]
```

Warning: There is absolutely no guarantee of recovering a ground truth. First, choosing the right number of clusters is hard. Second, the algorithm is sensitive to initialization, and can fall into local minima, although scikit-learn employs several tricks to mitigate this issue.

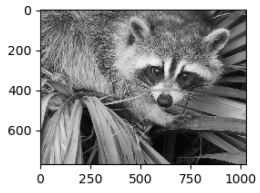
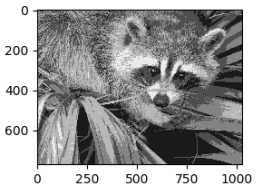
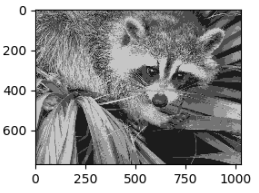
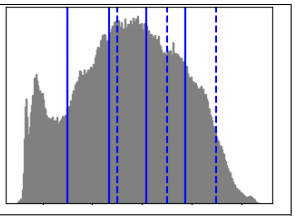
		
Bad initialization	**8 clusters**	**Ground truth**

Don't over-interpret clustering results

Application example: vector quantization

Clustering in general and KMeans, in particular, can be seen as a way of choosing a small number of exemplars to compress the information. The problem is sometimes known as vector quantization. For instance, this can be used to posterize an image:

```
>>> import scipy as sp
>>> try:
...     face = sp.face(gray=True)
... except AttributeError:
...     from scipy import misc
...     face = misc.face(gray=True)
>>> X = face.reshape((-1, 1)) # We need an (n_sample, n_feature) array
>>> k_means = cluster.KMeans(n_clusters=5, n_init=1)
>>> k_means.fit(X)
KMeans(n_clusters=5, n_init=1)
>>> values = k_means.cluster_centers_.squeeze()
>>> labels = k_means.labels_
>>> face_compressed = np.choose(labels, values)
>>> face_compressed.shape = face.shape
```

			
Raw image	K-means quantization	Equal bins	Image histogram

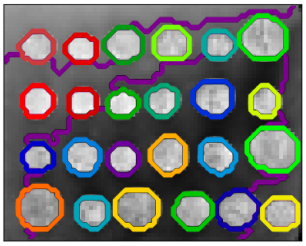
Hierarchical agglomerative clustering: Ward

A *Hierarchical clustering* method is a type of cluster analysis that aims to build a hierarchy of clusters. In general, the various approaches of this technique are either:

- **Agglomerative** - bottom-up approaches: each observation starts in its own cluster, and clusters are iteratively merged in such a way to minimize a *linkage* criterion. This approach is particularly interesting when the clusters of interest are made of only a few observations. When the number of clusters is large, it is much more computationally efficient than k-means.
- **Divisive** - top-down approaches: all observations start in one cluster, which is iteratively split as one moves down the hierarchy. For estimating large numbers of clusters, this approach is both slow (due to all observations starting as one cluster, which it splits recursively) and statistically ill-posed.

Connectivity-constrained clustering

With agglomerative clustering, it is possible to specify which samples can be clustered together by giving a connectivity graph. Graphs in scikit-learn are represented by their adjacency matrix. Often, a sparse matrix is used. This can be useful, for instance, to retrieve connected regions (sometimes also referred to as connected components) when clustering an image:



```
from scipy.ndimage.filters import gaussian_filter

import matplotlib.pyplot as plt

import skimage
from skimage.data import coins
from skimage.transform import rescale

from sklearn.feature_extraction.image import grid_to_graph
from sklearn.cluster import AgglomerativeClustering

# these were introduced in skimage-0.14
if LooseVersion(skimage.__version__) >= '0.14':
    rescale_params = {'anti_aliasing': False, 'multichannel': False}
else:
    rescale_params = {}

# ###############################################################################
# Generate data
orig_coins = coins()
```

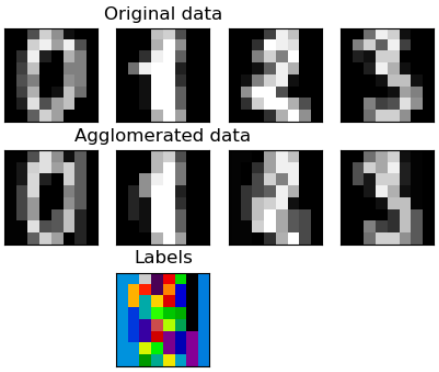
(continues on next page)

(continued from previous page)

```
# Resize it to 20% of the original size to speed up the processing
# Applying a Gaussian filter for smoothing prior to down-scaling
# reduces aliasing artifacts.
smoothened_coins = gaussian_filter(orig_coins, sigma=2)
```

Feature agglomeration

We have seen that sparsity could be used to mitigate the curse of dimensionality, *i.e* an insufficient amount of observations compared to the number of features. Another approach is to merge together similar features: **feature agglomeration**. This approach can be implemented by clustering in the feature direction, in other words clustering the transposed data.




```
>>> digits = datasets.load_digits()
>>> images = digits.images
>>> X = np.reshape(images, (len(images), -1))
>>> connectivity = grid_to_graph(*images[0].shape)

>>> agglo = cluster.FeatureAgglomeration(connectivity=connectivity,
...                                      n_clusters=32)
>>> agglo.fit(X)
FeatureAgglomeration(connectivity=..., n_clusters=32)
>>> X_reduced = agglo.transform(X)

>>> X_approx = agglo.inverse_transform(X_reduced)
>>> images_approx = np.reshape(X_approx, images.shape)
```

`transform` and `inverse_transform` methods

Some estimators expose a `transform` method, for instance to reduce the dimensionality of the dataset.

Decompositions: from a signal to components and loadings

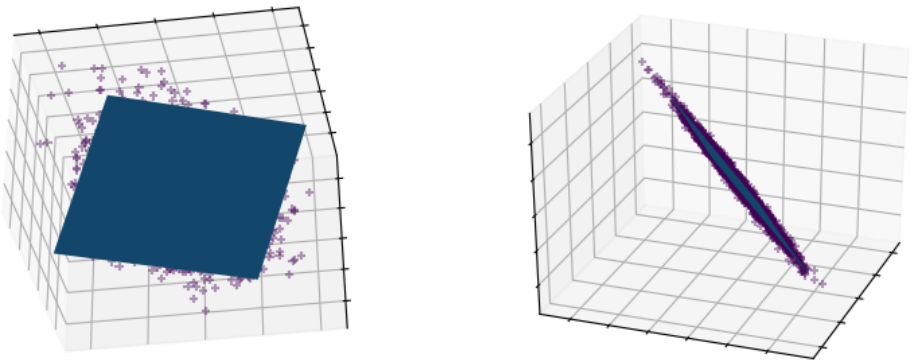
Components and loadings

If X is our multivariate data, then the problem that we are trying to solve is to rewrite it on a different observational basis: we want to learn loadings L and a set of components C such that $X = L\ C$. Different criteria exist to choose

the components

Principal component analysis: PCA

Principal component analysis (PCA) selects the successive components that explain the maximum variance in the signal.



The point cloud spanned by the observations above is very flat in one direction: one of the three univariate features can almost be exactly computed using the other two. PCA finds the directions in which the data is not *flat*

When used to *transform* data, PCA can reduce the dimensionality of the data by projecting on a principal subspace.

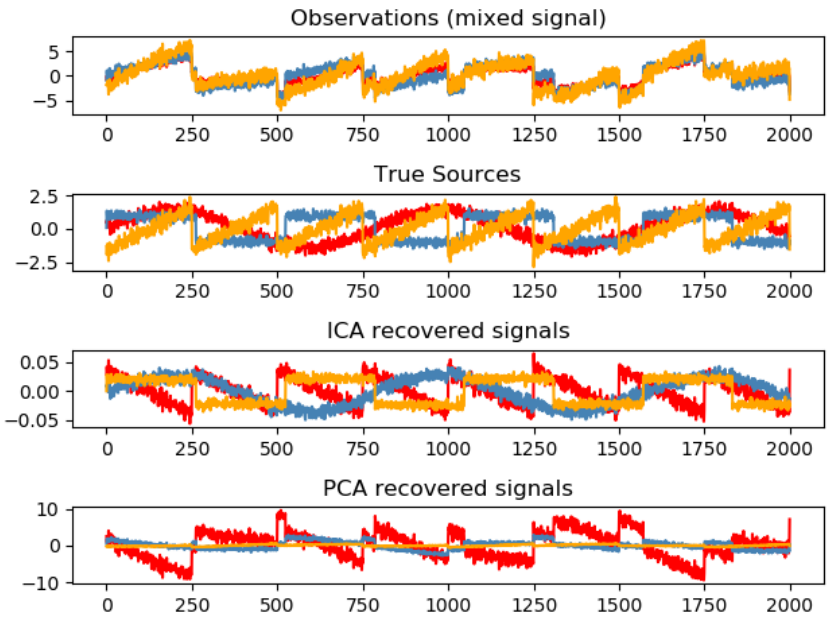
```
>>> # Create a signal with only 2 useful dimensions
>>> x1 = np.random.normal(size=100)
>>> x2 = np.random.normal(size=100)
>>> x3 = x1 + x2
>>> X = np.c_[x1, x2, x3]

>>> from sklearn import decomposition
>>> pca = decomposition.PCA()
>>> pca.fit(X)
PCA()
>>> print(pca.explained_variance_)
[  2.18565811e+00   1.19346747e+00   8.43026679e-32]

>>> # As we can see, only the 2 first components are useful
>>> pca.n_components = 2
>>> X_reduced = pca.fit_transform(X)
>>> X_reduced.shape
(100, 2)
```

Independent Component Analysis: ICA

Independent component analysis (ICA) selects components so that the distribution of their loadings carries a maximum amount of independent information. It is able to recover **non-Gaussian** independent signals:



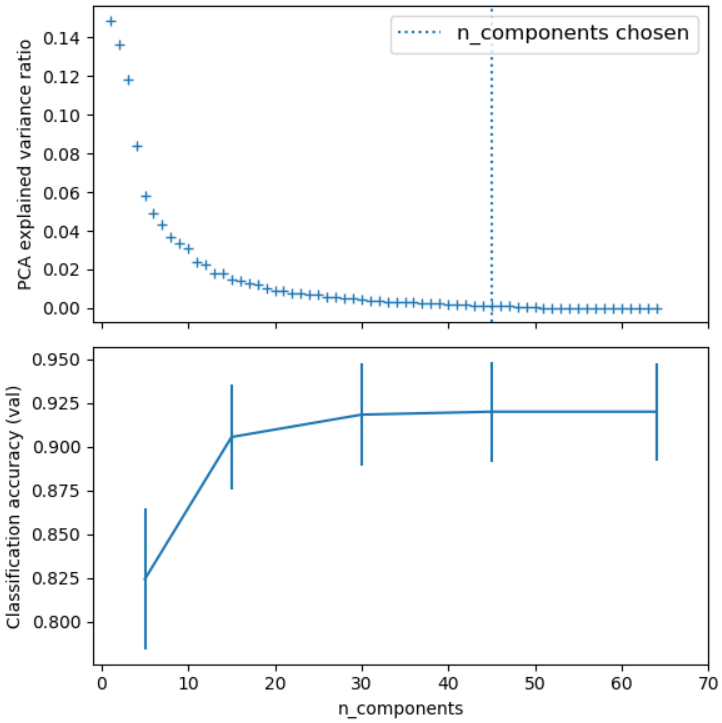

```
>>> # Generate sample data
>>> import numpy as np
>>> from scipy import signal
>>> time = np.linspace(0, 10, 2000)
>>> s1 = np.sin(2 * time)  # Signal 1 : sinusoidal signal
>>> s2 = np.sign(np.sin(3 * time))  # Signal 2 : square signal
>>> s3 = signal.sawtooth(2 * np.pi * time)  # Signal 3: saw tooth signal
>>> S = np.c_[s1, s2, s3]
>>> S += 0.2 * np.random.normal(size=S.shape)  # Add noise
>>> S /= S.std(axis=0)  # Standardize data
>>> # Mix data
>>> A = np.array([[1, 1, 1], [0.5, 2, 1], [1.5, 1, 2]])  # Mixing matrix
>>> X = np.dot(S, A.T)  # Generate observations

>>> # Compute ICA
>>> ica = decomposition.FastICA()
>>> S_ = ica.fit_transform(X)  # Get the estimated sources
>>> A_ = ica.mixing_.T
>>> np.allclose(X,  np.dot(S_, A_) + ica.mean_)
True
```

2.2.5 Putting it all together

Pipelining

We have seen that some estimators can transform data and that some estimators can predict variables. We can also create combined estimators:




```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV


# Define a pipeline to search for the best combination of PCA truncation
# and classifier regularization.
pca = PCA()
# set the tolerance to a large value to make the example faster
logistic = LogisticRegression(max_iter=10000, tol=0.1)
pipe = Pipeline(steps=[('pca', pca), ('logistic', logistic)])

X_digits, y_digits = datasets.load_digits(return_X_y=True)

# Parameters of pipelines can be set using '__' separated parameter names:
param_grid = {
    'pca__n_components': [5, 15, 30, 45, 64],
    'logistic__C': np.logspace(-4, 4, 4),
}
search = GridSearchCV(pipe, param_grid, n_jobs=-1)
search.fit(X_digits, y_digits)
print("Best parameter (CV score=%0.3f):" % search.best_score_)
print(search.best_params_)

# Plot the PCA spectrum
```

(continues on next page)

(continued from previous page)

```
pca.fit(X_digits)

fig, (ax0, ax1) = plt.subplots(nrows=2, sharex=True, figsize=(6, 6))
ax0.plot(np.arange(1, pca.n_components_ + 1),
         pca.explained_variance_ratio_, '+', linewidth=2)
ax0.set_ylabel('PCA explained variance ratio')

ax0.axvline(search.best_estimator_.named_steps['pca'].n_components,
            linestyle=':', label='n_components chosen')
ax0.legend(prop=dict(size=12))
```

Face recognition with eigenfaces

The dataset used in this example is a preprocessed excerpt of the "Labeled Faces in the Wild", also known as LFW:

http://vis-www.cs.umass.edu/lfw/lfw-funneled.tgz (233MB)

```
"""
===================================================
Faces recognition example using eigenfaces and SVMs
===================================================

The dataset used in this example is a preprocessed excerpt of the
"Labeled Faces in the Wild", aka LFW_:

  http://vis-www.cs.umass.edu/lfw/lfw-funneled.tgz (233MB)

.. _LFW: http://vis-www.cs.umass.edu/lfw/

Expected results for the top 5 most represented people in the dataset:

================== ============ ======= ========== =======
                   precision    recall  f1-score   support
================== ============ ======= ========== =======
     Ariel Sharon       0.67      0.92      0.77        13
     Colin Powell       0.75      0.78      0.76        60
  Donald Rumsfeld       0.78      0.67      0.72        27
    George W Bush       0.86      0.86      0.86       146
Gerhard Schroeder       0.76      0.76      0.76        25
      Hugo Chavez       0.67      0.67      0.67        15
       Tony Blair       0.81      0.69      0.75        36

      avg / total       0.80      0.80      0.80       322
================== ============ ======= ========== =======

"""
from time import time
import logging
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.datasets import fetch_lfw_people
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
```

(continues on next page)

(continued from previous page)

```
from sklearn.decomposition import PCA
from sklearn.svm import SVC


print(__doc__)

# Display progress logs on stdout
logging.basicConfig(level=logging.INFO, format='%(asctime)s %(message)s')


# ###############################################################################
# Download the data, if not already on disk and load it as numpy arrays

lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)

# introspect the images arrays to find the shapes (for plotting)
n_samples, h, w = lfw_people.images.shape

# for machine learning we use the 2 data directly (as relative pixel
# positions info is ignored by this model)
X = lfw_people.data
n_features = X.shape[1]

# the label to predict is the id of the person
y = lfw_people.target
target_names = lfw_people.target_names
n_classes = target_names.shape[0]

print("Total dataset size:")
print("n_samples: %d" % n_samples)
print("n_features: %d" % n_features)
print("n_classes: %d" % n_classes)


# ###############################################################################
# Split into a training set and a test set using a stratified k fold

# split into a training and testing set
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42)


# ###############################################################################
# Compute a PCA (eigenfaces) on the face dataset (treated as unlabeled
# dataset): unsupervised feature extraction / dimensionality reduction
n_components = 150

print("Extracting the top %d eigenfaces from %d faces"
      % (n_components, X_train.shape[0]))
t0 = time()
pca = PCA(n_components=n_components, svd_solver='randomized',
          whiten=True).fit(X_train)
print("done in %0.3fs" % (time() - t0))

eigenfaces = pca.components_.reshape((n_components, h, w))

print("Projecting the input data on the eigenfaces orthonormal basis")
```

(continues on next page)

(continued from previous page)

```
t0 = time()
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
print("done in %0.3fs" % (time() - t0))


# ###############################################################################
# Train a SVM classification model

print("Fitting the classifier to the training set")
t0 = time()
param_grid = {'C': [1e3, 5e3, 1e4, 5e4, 1e5],
              'gamma': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1], }
clf = GridSearchCV(
    SVC(kernel='rbf', class_weight='balanced'), param_grid
)
clf = clf.fit(X_train_pca, y_train)
print("done in %0.3fs" % (time() - t0))
print("Best estimator found by grid search:")
print(clf.best_estimator_)


# ###############################################################################
# Quantitative evaluation of the model quality on the test set

print("Predicting people's names on the test set")
t0 = time()
y_pred = clf.predict(X_test_pca)
print("done in %0.3fs" % (time() - t0))

print(classification_report(y_test, y_pred, target_names=target_names))
print(confusion_matrix(y_test, y_pred, labels=range(n_classes)))


# ###############################################################################
# Qualitative evaluation of the predictions using matplotlib

def plot_gallery(images, titles, h, w, n_row=3, n_col=4):
    """Helper function to plot a gallery of portraits"""
    plt.figure(figsize=(1.8 * n_col, 2.4 * n_row))
    plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray)
        plt.title(titles[i], size=12)
        plt.xticks(())
        plt.yticks(())


# plot the result of the prediction on a portion of the test set

def title(y_pred, y_test, target_names, i):
    pred_name = target_names[y_pred[i]].rsplit(' ', 1)[-1]
    true_name = target_names[y_test[i]].rsplit(' ', 1)[-1]
    return 'predicted: %s\ntrue:      %s' % (pred_name, true_name)

prediction_titles = [title(y_pred, y_test, target_names, i)
```

(continues on next page)

(continued from previous page)

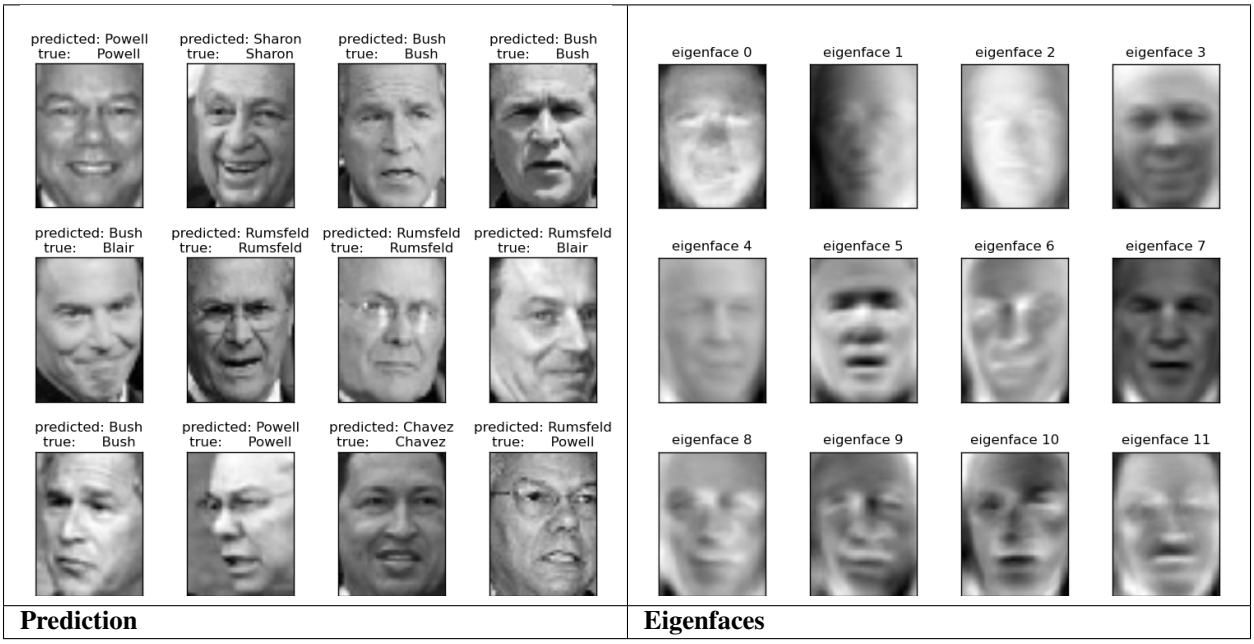
```
                    for i in range(y_pred.shape[0])]

plot_gallery(X_test, prediction_titles, h, w)

# plot the gallery of the most significative eigenfaces

eigenface_titles = ["eigenface %d" % i for i in range(eigenfaces.shape[0])]
plot_gallery(eigenfaces, eigenface_titles, h, w)

plt.show()
```



Expected results for the top 5 most represented people in the dataset:

```
                   precision    recall  f1-score   support

Gerhard_Schroeder       0.91      0.75      0.82        28
  Donald_Rumsfeld       0.84      0.82      0.83        33
        Tony_Blair       0.65      0.82      0.73        34
      Colin_Powell       0.78      0.88      0.83        58
   George_W_Bush       0.93      0.86      0.90       129

       avg / total       0.86      0.84      0.85       282
```

Open problem: Stock Market Structure

Can we predict the variation in stock prices for Google over a given time frame?

Learning a graph structure

2.2.6 Finding help

The project mailing list

If you encounter a bug with `scikit-learn` or something that needs clarification in the docstring or the online documentation, please feel free to ask on the Mailing List

Q&A communities with Machine Learning practitioners

Quora.com Quora has a topic for Machine Learning related questions that also features some interesting discussions: https://www.quora.com/topic/Machine-Learning

Stack Exchange The Stack Exchange family of sites hosts multiple subdomains for Machine Learning questions.

– _'An excellent free online course for Machine Learning taught by Professor Andrew Ng of Stanford': https://www.coursera.org/learn/machine-learning

– _'Another excellent free online course that takes a more general approach to Artificial Intelligence': https://www.udacity.com/course/intro-to-artificial-intelligence–cs271

2.3 Working With Text Data

The goal of this guide is to explore some of the main `scikit-learn` tools on a single practical task: analyzing a collection of text documents (newsgroups posts) on twenty different topics.

In this section we will see how to:

- load the file contents and the categories
- extract feature vectors suitable for machine learning
- train a linear model to perform categorization
- use a grid search strategy to find a good configuration of both the feature extraction components and the classifier

2.3.1 Tutorial setup

To get started with this tutorial, you must first install *scikit-learn* and all of its required dependencies.

Please refer to the *installation instructions* page for more information and for system-specific instructions.

The source of this tutorial can be found within your scikit-learn folder:

```
scikit-learn/doc/tutorial/text_analytics/
```

The source can also be found on Github.

The tutorial folder should contain the following sub-folders:

- `*.rst files` - the source of the tutorial document written with sphinx
- `data` - folder to put the datasets used during the tutorial
- `skeletons` - sample incomplete scripts for the exercises
- `solutions` - solutions of the exercises

You can already copy the skeletons into a new folder somewhere on your hard-drive named `sklearn_tut_workspace` where you will edit your own files for the exercises while keeping the original skeletons intact:

```
% cp -r skeletons work_directory/sklearn_tut_workspace
```

Machine learning algorithms need data. Go to each `$TUTORIAL_HOME/data` sub-folder and run the `fetch_data.py` script from there (after having read them first).

For instance:

```
% cd $TUTORIAL_HOME/data/languages
% less fetch_data.py
% python fetch_data.py
```

2.3.2 Loading the 20 newsgroups dataset

The dataset is called "Twenty Newsgroups". Here is the official description, quoted from the website:

> The 20 Newsgroups data set is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different newsgroups. To the best of our knowledge, it was originally collected by Ken Lang, probably for his paper "Newsweeder: Learning to filter netnews," though he does not explicitly mention this collection. The 20 newsgroups collection has become a popular data set for experiments in text applications of machine learning techniques, such as text classification and text clustering.

In the following we will use the built-in dataset loader for 20 newsgroups from scikit-learn. Alternatively, it is possible to download the dataset manually from the website and use the *`sklearn.datasets.load_files`* function by pointing it to the `20news-bydate-train` sub-folder of the uncompressed archive folder.

In order to get faster execution times for this first example we will work on a partial dataset with only 4 categories out of the 20 available in the dataset:

```
>>> categories = ['alt.atheism', 'soc.religion.christian',
...               'comp.graphics', 'sci.med']
```

We can now load the list of files matching those categories as follows:

```
>>> from sklearn.datasets import fetch_20newsgroups
>>> twenty_train = fetch_20newsgroups(subset='train',
...     categories=categories, shuffle=True, random_state=42)
```

The returned dataset is a `scikit-learn` "bunch": a simple holder object with fields that can be both accessed as python `dict` keys or `object` attributes for convenience, for instance the `target_names` holds the list of the requested category names:

```
>>> twenty_train.target_names
['alt.atheism', 'comp.graphics', 'sci.med', 'soc.religion.christian']
```

The files themselves are loaded in memory in the `data` attribute. For reference the filenames are also available:

```
>>> len(twenty_train.data)
2257
>>> len(twenty_train.filenames)
2257
```

Let's print the first lines of the first loaded file:

```
>>> print("\n".join(twenty_train.data[0].split("\n")[:3]))
From: sd345@city.ac.uk (Michael Collier)
Subject: Converting images to HP LaserJet III?
```

(continues on next page)

(continued from previous page)

```
Nntp-Posting-Host: hampton

>>> print(twenty_train.target_names[twenty_train.target[0]])
comp.graphics
```

Supervised learning algorithms will require a category label for each document in the training set. In this case the category is the name of the newsgroup which also happens to be the name of the folder holding the individual documents.

For speed and space efficiency reasons `scikit-learn` loads the target attribute as an array of integers that corresponds to the index of the category name in the `target_names` list. The category integer id of each sample is stored in the `target` attribute:

```
>>> twenty_train.target[:10]
array([1, 1, 3, 3, 3, 3, 3, 2, 2, 2])
```

It is possible to get back the category names as follows:

```
>>> for t in twenty_train.target[:10]:
...     print(twenty_train.target_names[t])
...
comp.graphics
comp.graphics
soc.religion.christian
soc.religion.christian
soc.religion.christian
soc.religion.christian
soc.religion.christian
sci.med
sci.med
sci.med
```

You might have noticed that the samples were shuffled randomly when we called `fetch_20newsgroups(..., shuffle=True, random_state=42)`: this is useful if you wish to select only a subset of samples to quickly train a model and get a first idea of the results before re-training on the complete dataset later.

2.3.3 Extracting features from text files

In order to perform machine learning on text documents, we first need to turn the text content into numerical feature vectors.

Bags of words

The most intuitive way to do so is to use a bags of words representation:

1. Assign a fixed integer id to each word occurring in any document of the training set (for instance by building a dictionary from words to integer indices).
2. For each document `#i`, count the number of occurrences of each word `w` and store it in `X[i, j]` as the value of feature `#j` where `j` is the index of word `w` in the dictionary.

The bags of words representation implies that `n_features` is the number of distinct words in the corpus: this number is typically larger than 100,000.

If `n_samples == 10000`, storing `X` as a NumPy array of type float32 would require 10000 x 100000 x 4 bytes = **4GB in RAM** which is barely manageable on today's computers.

Fortunately, **most values in X will be zeros** since for a given document less than a few thousand distinct words will be used. For this reason we say that bags of words are typically **high-dimensional sparse datasets**. We can save a lot of memory by only storing the non-zero parts of the feature vectors in memory.

`scipy.sparse` matrices are data structures that do exactly this, and `scikit-learn` has built-in support for these structures.

Tokenizing text with `scikit-learn`

Text preprocessing, tokenizing and filtering of stopwords are all included in *`CountVectorizer`*, which builds a dictionary of features and transforms documents to feature vectors:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count_vect = CountVectorizer()
>>> X_train_counts = count_vect.fit_transform(twenty_train.data)
>>> X_train_counts.shape
(2257, 35788)
```

`CountVectorizer` supports counts of N-grams of words or consecutive characters. Once fitted, the vectorizer has built a dictionary of feature indices:

```
>>> count_vect.vocabulary_.get(u'algorithm')
4690
```

The index value of a word in the vocabulary is linked to its frequency in the whole training corpus.

From occurrences to frequencies

Occurrence count is a good start but there is an issue: longer documents will have higher average count values than shorter documents, even though they might talk about the same topics.

To avoid these potential discrepancies it suffices to divide the number of occurrences of each word in a document by the total number of words in the document: these new features are called `tf` for Term Frequencies.

Another refinement on top of tf is to downscale weights for words that occur in many documents in the corpus and are therefore less informative than those that occur only in a smaller portion of the corpus.

This downscaling is called tf–idf for "Term Frequency times Inverse Document Frequency".

Both **tf** and **tf–idf** can be computed as follows using *`TfidfTransformer`*:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> tf_transformer = TfidfTransformer(use_idf=False).fit(X_train_counts)
>>> X_train_tf = tf_transformer.transform(X_train_counts)
>>> X_train_tf.shape
(2257, 35788)
```

In the above example-code, we firstly use the `fit(..)` method to fit our estimator to the data and secondly the `transform(..)` method to transform our count-matrix to a tf-idf representation. These two steps can be combined to achieve the same end result faster by skipping redundant processing. This is done through using the `fit_transform(..)` method as shown below, and as mentioned in the note in the previous section:

```
>>> tfidf_transformer = TfidfTransformer()
>>> X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
>>> X_train_tfidf.shape
(2257, 35788)
```

2.3.4 Training a classifier

Now that we have our features, we can train a classifier to try to predict the category of a post. Let's start with a *naïve Bayes* classifier, which provides a nice baseline for this task. `scikit-learn` includes several variants of this classifier; the one most suitable for word counts is the multinomial variant:

```
>>> from sklearn.naive_bayes import MultinomialNB
>>> clf = MultinomialNB().fit(X_train_tfidf, twenty_train.target)
```

To try to predict the outcome on a new document we need to extract the features using almost the same feature extracting chain as before. The difference is that we call `transform` instead of `fit_transform` on the transformers, since they have already been fit to the training set:

```
>>> docs_new = ['God is love', 'OpenGL on the GPU is fast']
>>> X_new_counts = count_vect.transform(docs_new)
>>> X_new_tfidf = tfidf_transformer.transform(X_new_counts)

>>> predicted = clf.predict(X_new_tfidf)

>>> for doc, category in zip(docs_new, predicted):
...     print('%r => %s' % (doc, twenty_train.target_names[category]))
...
'God is love' => soc.religion.christian
'OpenGL on the GPU is fast' => comp.graphics
```

2.3.5 Building a pipeline

In order to make the vectorizer => transformer => classifier easier to work with, `scikit-learn` provides a *`Pipeline`* class that behaves like a compound classifier:

```
>>> from sklearn.pipeline import Pipeline
>>> text_clf = Pipeline([
...     ('vect', CountVectorizer()),
...     ('tfidf', TfidfTransformer()),
...     ('clf', MultinomialNB()),
... ])
```

The names `vect`, `tfidf` and `clf` (classifier) are arbitrary. We will use them to perform grid search for suitable hyperparameters below. We can now train the model with a single command:

```
>>> text_clf.fit(twenty_train.data, twenty_train.target)
Pipeline(...)
```

2.3.6 Evaluation of the performance on the test set

Evaluating the predictive accuracy of the model is equally easy:

```
>>> import numpy as np
>>> twenty_test = fetch_20newsgroups(subset='test',
...     categories=categories, shuffle=True, random_state=42)
>>> docs_test = twenty_test.data
>>> predicted = text_clf.predict(docs_test)
>>> np.mean(predicted == twenty_test.target)
0.8348...
```

We achieved 83.5% accuracy. Let's see if we can do better with a linear *support vector machine (SVM)*, which is widely regarded as one of the best text classification algorithms (although it's also a bit slower than naïve Bayes). We can change the learner by simply plugging a different classifier object into our pipeline:

```
>>> from sklearn.linear_model import SGDClassifier
>>> text_clf = Pipeline([
...     ('vect', CountVectorizer()),
...     ('tfidf', TfidfTransformer()),
...     ('clf', SGDClassifier(loss='hinge', penalty='l2',
...                           alpha=1e-3, random_state=42,
...                           max_iter=5, tol=None)),
... ])

>>> text_clf.fit(twenty_train.data, twenty_train.target)
Pipeline(...)
>>> predicted = text_clf.predict(docs_test)
>>> np.mean(predicted == twenty_test.target)
0.9101...
```

We achieved 91.3% accuracy using the SVM. `scikit-learn` provides further utilities for more detailed performance analysis of the results:

```
>>> from sklearn import metrics
>>> print(metrics.classification_report(twenty_test.target, predicted,
...     target_names=twenty_test.target_names))
                        precision    recall  f1-score   support

           alt.atheism       0.95      0.80      0.87       319
         comp.graphics       0.87      0.98      0.92       389
               sci.med       0.94      0.89      0.91       396
soc.religion.christian       0.90      0.95      0.93       398

              accuracy                           0.91      1502
             macro avg       0.91      0.91      0.91      1502
          weighted avg       0.91      0.91      0.91      1502


>>> metrics.confusion_matrix(twenty_test.target, predicted)
array([[256,  11,  16,  36],
       [  4, 380,   3,   2],
       [  5,  35, 353,   3],
       [  5,  11,   4, 378]])
```

As expected the confusion matrix shows that posts from the newsgroups on atheism and Christianity are more often confused for one another than with computer graphics.

2.3.7 Parameter tuning using grid search

We've already encountered some parameters such as `use_idf` in the `TfidfTransformer`. Classifiers tend to have many parameters as well; e.g., `MultinomialNB` includes a smoothing parameter `alpha` and `SGDClassifier` has a penalty parameter `alpha` and configurable loss and penalty terms in the objective function (see the module documentation, or use the Python `help` function to get a description of these).

Instead of tweaking the parameters of the various components of the chain, it is possible to run an exhaustive search of the best parameters on a grid of possible values. We try out all classifiers on either words or bigrams, with or without idf, and with a penalty parameter of either 0.01 or 0.001 for the linear SVM:

```
>>> from sklearn.model_selection import GridSearchCV
>>> parameters = {
...     'vect__ngram_range': [(1, 1), (1, 2)],
...     'tfidf__use_idf': (True, False),
...     'clf__alpha': (1e-2, 1e-3),
... }
```

Obviously, such an exhaustive search can be expensive. If we have multiple CPU cores at our disposal, we can tell the grid searcher to try these eight parameter combinations in parallel with the `n_jobs` parameter. If we give this parameter a value of `-1`, grid search will detect how many cores are installed and use them all:

```
>>> gs_clf = GridSearchCV(text_clf, parameters, cv=5, n_jobs=-1)
```

The grid search instance behaves like a normal `scikit-learn` model. Let's perform the search on a smaller subset of the training data to speed up the computation:

```
>>> gs_clf = gs_clf.fit(twenty_train.data[:400], twenty_train.target[:400])
```

The result of calling `fit` on a `GridSearchCV` object is a classifier that we can use to `predict`:

```
>>> twenty_train.target_names[gs_clf.predict(['God is love'])[0]]
'soc.religion.christian'
```

The object's `best_score_` and `best_params_` attributes store the best mean score and the parameters setting corresponding to that score:

```
>>> gs_clf.best_score_
0.9...
>>> for param_name in sorted(parameters.keys()):
...     print("%s: %r" % (param_name, gs_clf.best_params_[param_name]))
...
clf__alpha: 0.001
tfidf__use_idf: True
vect__ngram_range: (1, 1)
```

A more detailed summary of the search is available at `gs_clf.cv_results_`.

The `cv_results_` parameter can be easily imported into pandas as a `DataFrame` for further inspection.

Exercises

To do the exercises, copy the content of the 'skeletons' folder as a new folder named 'workspace':

```
% cp -r skeletons workspace
```

You can then edit the content of the workspace without fear of losing the original exercise instructions.

Then fire an ipython shell and run the work-in-progress script with:

```
[1] %run workspace/exercise_XX_script.py arg1 arg2 arg3
```

If an exception is triggered, use `%debug` to fire-up a post mortem ipdb session.

Refine the implementation and iterate until the exercise is solved.

For each exercise, the skeleton file provides all the necessary import statements, boilerplate code to load the data and sample code to evaluate the predictive accuracy of the model.

2.3.8 Exercise 1: Language identification

- Write a text classification pipeline using a custom preprocessor and `CharNGramAnalyzer` using data from Wikipedia articles as training set.
- Evaluate the performance on some held out test set.

ipython command line:

```
%run workspace/exercise_01_language_train_model.py data/languages/paragraphs/
```

2.3.9 Exercise 2: Sentiment Analysis on movie reviews

- Write a text classification pipeline to classify movie reviews as either positive or negative.
- Find a good set of parameters using grid search.
- Evaluate the performance on a held out test set.

ipython command line:

```
%run workspace/exercise_02_sentiment.py data/movie_reviews/txt_sentoken/
```

2.3.10 Exercise 3: CLI text classification utility

Using the results of the previous exercises and the `cPickle` module of the standard library, write a command line utility that detects the language of some text provided on `stdin` and estimate the polarity (positive or negative) if the text is written in English.

Bonus point if the utility is able to give a confidence level for its predictions.

2.3.11 Where to from here

Here are a few suggestions to help further your scikit-learn intuition upon the completion of this tutorial:

- Try playing around with the `analyzer` and `token normalisation` under *`CountVectorizer`*.
- If you don't have labels, try using *Clustering* on your problem.
- If you have multiple labels per document, e.g categories, have a look at the *Multiclass and multilabel section*.
- Try using *Truncated SVD* for latent semantic analysis.
- Have a look at using *Out-of-core Classification* to learn from data that would not fit into the computer main memory.
- Have a look at the *Hashing Vectorizer* as a memory efficient alternative to *`CountVectorizer`*.

2.4 Choosing the right estimator

Often the hardest part of solving a machine learning problem can be finding the right estimator for the job.

Different estimators are better suited for different types of data and different problems.

The flowchart below is designed to give users a bit of a rough guide on how to approach problems with regard to which estimators to try on your data.

Click on any estimator in the chart below to see its documentation.

2.5 External Resources, Videos and Talks

For written tutorials, see the *Tutorial section* of the documentation.

2.5.1 New to Scientific Python?

For those that are still new to the scientific Python ecosystem, we highly recommend the Python Scientific Lecture Notes. This will help you find your footing a bit and will definitely improve your scikit-learn experience. A basic understanding of NumPy arrays is recommended to make the most of scikit-learn.

2.5.2 External Tutorials

There are several online tutorials available which are geared toward specific subject areas:

- Machine Learning for NeuroImaging in Python
- Machine Learning for Astronomical Data Analysis

2.5.3 Videos

- An introduction to scikit-learn Part I and Part II at Scipy 2013 by Gael Varoquaux, Jake Vanderplas and Olivier Grisel. Notebooks on github.
- Introduction to scikit-learn by Gael Varoquaux at ICML 2010

  A three minute video from a very early stage of scikit-learn, explaining the basic idea and approach we are following.
- Introduction to statistical learning with scikit-learn by Gael Varoquaux at SciPy 2011

  An extensive tutorial, consisting of four sessions of one hour. The tutorial covers the basics of machine learning, many algorithms and how to apply them using scikit-learn. The material corresponding is now in the scikit-learn documentation section *A tutorial on statistical-learning for scientific data processing*.
- Statistical Learning for Text Classification with scikit-learn and NLTK (and slides) by Olivier Grisel at PyCon 2011

  Thirty minute introduction to text classification. Explains how to use NLTK and scikit-learn to solve real-world text classification tasks and compares against cloud-based solutions.
- Introduction to Interactive Predictive Analytics in Python with scikit-learn by Olivier Grisel at PyCon 2012

  3-hours long introduction to prediction tasks using scikit-learn.
- scikit-learn - Machine Learning in Python by Jake Vanderplas at the 2012 PyData workshop at Google

  Interactive demonstration of some scikit-learn features. 75 minutes.
- scikit-learn tutorial by Jake Vanderplas at PyData NYC 2012

  Presentation using the online tutorial, 45 minutes.

Note: Doctest Mode

The code-examples in the above tutorials are written in a *python-console* format. If you wish to easily execute these examples in **IPython**, use:

```
%doctest_mode
```

in the IPython-console. You can then simply copy and paste the examples directly into IPython without having to worry about removing the **>>>** manually.

CHAPTER

THREE

GETTING STARTED

The purpose of this guide is to illustrate some of the main features that `scikit-learn` provides. It assumes a very basic working knowledge of machine learning practices (model fitting, predicting, cross-validation, etc.). Please refer to our *installation instructions* for installing `scikit-learn`.

`Scikit-learn` is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities.

3.1 Fitting and predicting: estimator basics

`Scikit-learn` provides dozens of built-in machine learning algorithms and models, called *estimators*. Each estimator can be fitted to some data using its *fit* method.

Here is a simple example where we fit a *`RandomForestClassifier`* to some very basic data:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> clf = RandomForestClassifier(random_state=0)
>>> X = [[ 1,  2,  3],  # 2 samples, 3 features
...      [11, 12, 13]]
>>> y = [0, 1]  # classes of each sample
>>> clf.fit(X, y)
RandomForestClassifier(random_state=0)
```

The *fit* method generally accepts 2 inputs:

- The samples matrix (or design matrix) *X*. The size of `X` is typically `(n_samples, n_features)`, which means that samples are represented as rows and features are represented as columns.
- The target values *y* which are real numbers for regression tasks, or integers for classification (or any other discrete set of values). For unsupervized learning tasks, `y` does not need to be specified. `y` is usually 1d array where the `i` th entry corresponds to the target of the `i` th sample (row) of `X`.

Both `X` and `y` are usually expected to be numpy arrays or equivalent *array-like* data types, though some estimators work with other formats such as sparse matrices.

Once the estimator is fitted, it can be used for predicting target values of new data. You don't need to re-train the estimator:

```
>>> clf.predict(X)  # predict classes of the training data
array([0, 1])
>>> clf.predict([[4, 5, 6], [14, 15, 16]])  # predict classes of new data
array([0, 1])
```

3.2 Transformers and pre-processors

Machine learning workflows are often composed of different parts. A typical pipeline consists of a pre-processing step that transforms or imputes the data, and a final predictor that predicts target values.

In `scikit-learn`, pre-processors and transformers follow the same API as the estimator objects (they actually all inherit from the same `BaseEstimator` class). The transformer objects don't have a *predict* method but rather a *transform* method that outputs a newly transformed sample matrix `X`:

```
>>> from sklearn.preprocessing import StandardScaler
>>> X = [[0, 15],
...      [1, -10]]
>>> StandardScaler().fit(X).transform(X)
array([[-1.,  1.],
       [ 1., -1.]])
```

Sometimes, you want to apply different transformations to different features: the *ColumnTransformer* is designed for these use-cases.

3.3 Pipelines: chaining pre-processors and estimators

Transformers and estimators (predictors) can be combined together into a single unifying object: a `Pipeline`. The pipeline offers the same API as a regular estimator: it can be fitted and used for prediction with `fit` and `predict`. As we will see later, using a pipeline will also prevent you from data leakage, i.e. disclosing some testing data in your training data.

In the following example, we *load the Iris dataset*, split it into train and test sets, and compute the accuracy score of a pipeline on the test data:

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score
...
>>> # create a pipeline object
>>> pipe = make_pipeline(
...     StandardScaler(),
...     LogisticRegression(random_state=0)
... )
...
>>> # load the iris dataset and split it into train and test sets
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
...
>>> # fit the whole pipeline
>>> pipe.fit(X_train, y_train)
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('logisticregression', LogisticRegression(random_state=0))])
>>> # we can now use it like any other estimator
>>> accuracy_score(pipe.predict(X_test), y_test)
0.97...
```

3.4 Model evaluation

Fitting a model to some data does not entail that it will predict well on unseen data. This needs to be directly evaluated. We have just seen the *train_test_split* helper that splits a dataset into train and test sets, but scikit-learn provides many other tools for model evaluation, in particular for *cross-validation*.

We here briefly show how to perform a 5-fold cross-validation procedure, using the *cross_validate* helper. Note that it is also possible to manually iterate over the folds, use different data splitting strategies, and use custom scoring functions. Please refer to our *User Guide* for more details:

```
>>> from sklearn.datasets import make_regression
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.model_selection import cross_validate
...
>>> X, y = make_regression(n_samples=1000, random_state=0)
>>> lr = LinearRegression()
...
>>> result = cross_validate(lr, X, y)  # defaults to 5-fold CV
>>> result['test_score']  # r_squared score is high because dataset is easy
array([1., 1., 1., 1., 1.])
```

3.5 Automatic parameter searches

All estimators have parameters (often called hyper-parameters in the literature) that can be tuned. The generalization power of an estimator often critically depends on a few parameters. For example a *RandomForestRegressor* has a n_estimators parameter that determines the number of trees in the forest, and a max_depth parameter that determines the maximum depth of each tree. Quite often, it is not clear what the exact values of these parameters should be since they depend on the data at hand.

Scikit-learn provides tools to automatically find the best parameter combinations (via cross-validation). In the following example, we randomly search over the parameter space of a random forest with a *RandomizedSearchCV* object. When the search is over, the *RandomizedSearchCV* behaves as a *RandomForestRegressor* that has been fitted with the best set of parameters. Read more in the *User Guide*:

```
>>> from sklearn.datasets import fetch_california_housing
>>> from sklearn.ensemble import RandomForestRegressor
>>> from sklearn.model_selection import RandomizedSearchCV
>>> from sklearn.model_selection import train_test_split
>>> from scipy.stats import randint
...
>>> X, y = fetch_california_housing(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
...
>>> # define the parameter space that will be searched over
>>> param_distributions = {'n_estimators': randint(1, 5),
...                        'max_depth': randint(5, 10)}
...
>>> # now create a searchCV object and fit it to the data
>>> search = RandomizedSearchCV(estimator=RandomForestRegressor(random_state=0),
...                             n_iter=5,
...                             param_distributions=param_distributions,
...                             random_state=0)
>>> search.fit(X_train, y_train)
RandomizedSearchCV(estimator=RandomForestRegressor(random_state=0), n_iter=5,
```

(continues on next page)

(continued from previous page)

```
                    param_distributions={'max_depth': ...,
                                         'n_estimators': ...},
                    random_state=0)
>>> search.best_params_
{'max_depth': 9, 'n_estimators': 4}

>>> # the search object now acts like a normal random forest estimator
>>> # with max_depth=9 and n_estimators=4
>>> search.score(X_test, y_test)
0.73...
```

Note: In practice, you almost always want to *search over a pipeline*, instead of a single estimator. One of the main reasons is that if you apply a pre-processing step to the whole dataset without using a pipeline, and then perform any kind of cross-validation, you would be breaking the fundamental assumption of independence between training and testing data. Indeed, since you pre-processed the data using the whole dataset, some information about the test sets are available to the train sets. This will lead to over-estimating the generalization power of the estimator (you can read more in this kaggle post).

Using a pipeline for cross-validation and searching will largely keep you from this common pitfall.

3.6 Next steps

We have briefly covered estimator fitting and predicting, pre-processing steps, pipelines, cross-validation tools and automatic hyper-parameter searches. This guide should give you an overview of some of the main features of the library, but there is much more to `scikit-learn`!

Please refer to our *User Guide* for details on all the tools that we provide. You can also find an exhaustive list of the public API in the *API Reference*.

You can also look at our numerous *examples* that illustrate the use of `scikit-learn` in many different contexts.

The *tutorials* also contain additional learning resources.

CHAPTER

FOUR

USER GUIDE

4.1 Supervised learning

4.1.1 Linear Models

The following are a set of methods intended for regression in which the target value is expected to be a linear combination of the features. In mathematical notation, if $\hat{y}$ is the predicted value.

$$\hat{y}(w, x) = w_0 + w_1 x_1 + ... + w_p x_p$$

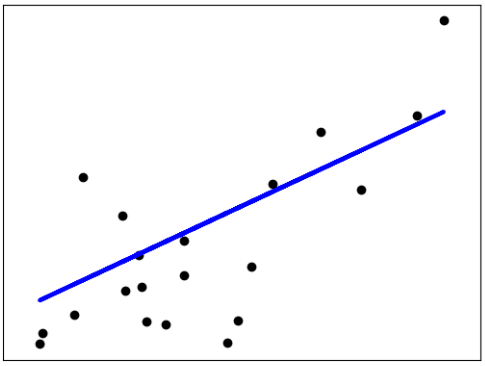
Across the module, we designate the vector $w = (w_1, ..., w_p)$ as `coef_` and w_0 as `intercept_`.

To perform classification with generalized linear models, see *Logistic regression*.

Ordinary Least Squares

`LinearRegression` fits a linear model with coefficients $w = (w_1, ..., w_p)$ to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation. Mathematically it solves a problem of the form:

$$\min_w ||Xw - y||_2^2$$



`LinearRegression` will take in its `fit` method arrays X, y and will store the coefficients w of the linear model in its `coef_` member:

```
>>> from sklearn import linear_model
>>> reg = linear_model.LinearRegression()
>>> reg.fit([[0, 0], [1, 1], [2, 2]], [0, 1, 2])
LinearRegression()
>>> reg.coef_
array([0.5, 0.5])
```

The coefficient estimates for Ordinary Least Squares rely on the independence of the features. When features are correlated and the columns of the design matrix X have an approximate linear dependence, the design matrix becomes close to singular and as a result, the least-squares estimate becomes highly sensitive to random errors in the observed target, producing a large variance. This situation of *multicollinearity* can arise, for example, when data are collected without an experimental design.

Examples:

- *Linear Regression Example*

Ordinary Least Squares Complexity

The least squares solution is computed using the singular value decomposition of X. If X is a matrix of shape `(n_samples, n_features)` this method has a cost of $O(n_{\text{samples}} n_{\text{features}}^2)$, assuming that $n_{\text{samples}} \geq n_{\text{features}}$.

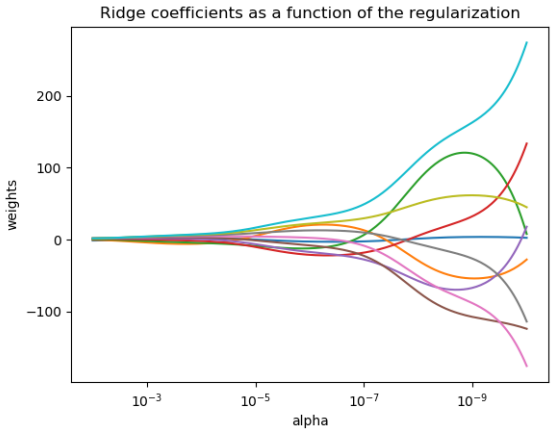
Ridge regression and classification

Regression

`Ridge` regression addresses some of the problems of *Ordinary Least Squares* by imposing a penalty on the size of the coefficients. The ridge coefficients minimize a penalized residual sum of squares:

$$\min_w ||Xw - y||_2^2 + \alpha ||w||_2^2$$

The complexity parameter $\alpha \geq 0$ controls the amount of shrinkage: the larger the value of α, the greater the amount of shrinkage and thus the coefficients become more robust to collinearity.



As with other linear models, `Ridge` will take in its `fit` method arrays X, y and will store the coefficients w of the linear model in its `coef_` member:

```
>>> from sklearn import linear_model
>>> reg = linear_model.Ridge(alpha=.5)
>>> reg.fit([[0, 0], [0, 0], [1, 1]], [0, .1, 1])
Ridge(alpha=0.5)
>>> reg.coef_
array([0.34545455, 0.34545455])
>>> reg.intercept_
0.13636...
```

Classification

The *Ridge* regressor has a classifier variant: *RidgeClassifier*. This classifier first converts binary targets to `{-1, 1}` and then treats the problem as a regression task, optimizing the same objective as above. The predicted class corresponds to the sign of the regressor's prediction. For multiclass classification, the problem is treated as multi-output regression, and the predicted class corresponds to the output with the highest value.

It might seem questionable to use a (penalized) Least Squares loss to fit a classification model instead of the more traditional logistic or hinge losses. However in practice all those models can lead to similar cross-validation scores in terms of accuracy or precision/recall, while the penalized least squares loss used by the *RidgeClassifier* allows for a very different choice of the numerical solvers with distinct computational performance profiles.

The *RidgeClassifier* can be significantly faster than e.g. *LogisticRegression* with a high number of classes, because it is able to compute the projection matrix $(X^T X)^{-1} X^T$ only once.

This classifier is sometimes referred to as a Least Squares Support Vector Machines with a linear kernel.

Examples:

- *Plot Ridge coefficients as a function of the regularization*
- *Classification of text documents using sparse features*

Ridge Complexity

This method has the same order of complexity as *Ordinary Least Squares*.

Setting the regularization parameter: generalized Cross-Validation

RidgeCV implements ridge regression with built-in cross-validation of the alpha parameter. The object works in the same way as GridSearchCV except that it defaults to Generalized Cross-Validation (GCV), an efficient form of leave-one-out cross-validation:

```
>>> import numpy as np
>>> from sklearn import linear_model
>>> reg = linear_model.RidgeCV(alphas=np.logspace(-6, 6, 13))
>>> reg.fit([[0, 0], [0, 0], [1, 1]], [0, .1, 1])
RidgeCV(alphas=array([1.e-06, 1.e-05, 1.e-04, 1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01,
      1.e+02, 1.e+03, 1.e+04, 1.e+05, 1.e+06]))
>>> reg.alpha_
0.01
```

Specifying the value of the *cv* attribute will trigger the use of cross-validation with `GridSearchCV`, for example `cv=10` for 10-fold cross-validation, rather than Generalized Cross-Validation.

References

- "Notes on Regularized Least Squares", Rifkin & Lippert (technical report, course slides).

Lasso

The `Lasso` is a linear model that estimates sparse coefficients. It is useful in some contexts due to its tendency to prefer solutions with fewer non-zero coefficients, effectively reducing the number of features upon which the given solution is dependent. For this reason Lasso and its variants are fundamental to the field of compressed sensing. Under certain conditions, it can recover the exact set of non-zero coefficients (see *Compressive sensing: tomography reconstruction with L1 prior (Lasso)*).

Mathematically, it consists of a linear model with an added regularization term. The objective function to minimize is:

$$\min_w \frac{1}{2n_{\text{samples}}}||Xw - y||_2^2 + \alpha||w||_1$$

The lasso estimate thus solves the minimization of the least-squares penalty with $\alpha||w||_1$ added, where α is a constant and $||w||_1$ is the ℓ_1-norm of the coefficient vector.

The implementation in the class `Lasso` uses coordinate descent as the algorithm to fit the coefficients. See *Least Angle Regression* for another implementation:

```
>>> from sklearn import linear_model
>>> reg = linear_model.Lasso(alpha=0.1)
>>> reg.fit([[0, 0], [1, 1]], [0, 1])
Lasso(alpha=0.1)
>>> reg.predict([[1, 1]])
array([0.8])
```

The function `lasso_path` is useful for lower-level tasks, as it computes the coefficients along the full path of possible values.

Examples:

- *Lasso and Elastic Net for Sparse Signals*
- *Compressive sensing: tomography reconstruction with L1 prior (Lasso)*

Note: Feature selection with Lasso

As the Lasso regression yields sparse models, it can thus be used to perform feature selection, as detailed in *L1-based feature selection*.

The following two references explain the iterations used in the coordinate descent solver of scikit-learn, as well as the duality gap computation used for convergence control.

References

- "Regularization Path For Generalized linear Models by Coordinate Descent", Friedman, Hastie & Tibshirani, J Stat Softw, 2010 (Paper).
- "An Interior-Point Method for Large-Scale L1-Regularized Least Squares," S. J. Kim, K. Koh, M. Lustig, S. Boyd and D. Gorinevsky, in IEEE Journal of Selected Topics in Signal Processing, 2007 (Paper)

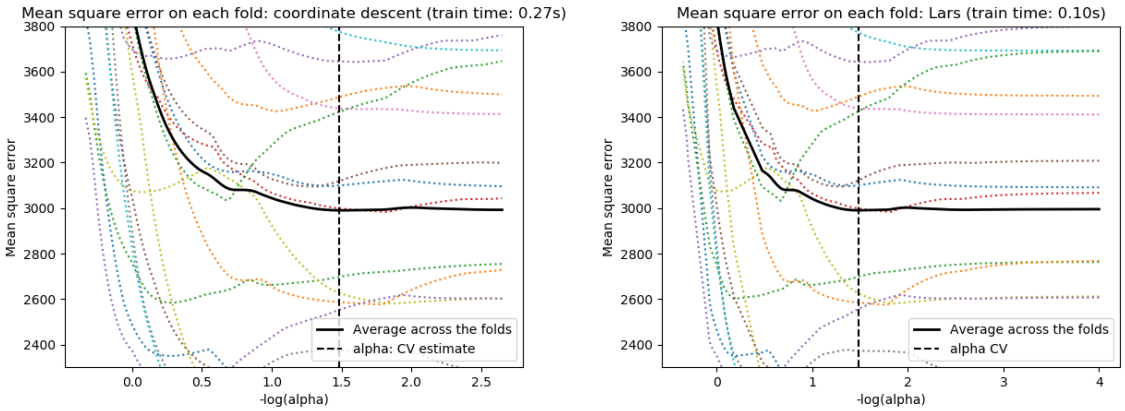
Setting regularization parameter

The `alpha` parameter controls the degree of sparsity of the estimated coefficients.

Using cross-validation

scikit-learn exposes objects that set the Lasso `alpha` parameter by cross-validation: `LassoCV` and `LassoLarsCV`. `LassoLarsCV` is based on the *Least Angle Regression* algorithm explained below.

For high-dimensional datasets with many collinear features, `LassoCV` is most often preferable. However, `LassoLarsCV` has the advantage of exploring more relevant values of `alpha` parameter, and if the number of samples is very small compared to the number of features, it is often faster than `LassoCV`.

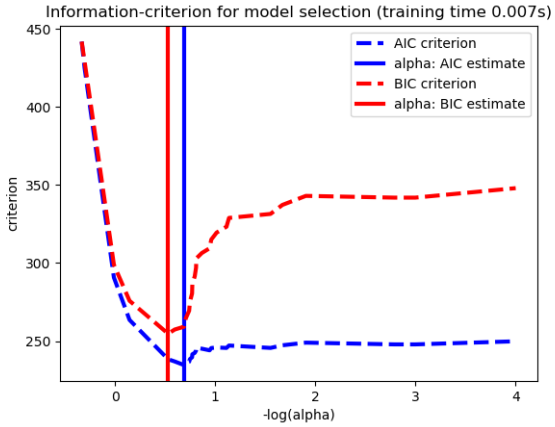


Information-criteria based model selection

Alternatively, the estimator `LassoLarsIC` proposes to use the Akaike information criterion (AIC) and the Bayes Information criterion (BIC). It is a computationally cheaper alternative to find the optimal value of alpha as the regularization path is computed only once instead of k+1 times when using k-fold cross-validation. However, such criteria needs a proper estimation of the degrees of freedom of the solution, are derived for large samples (asymptotic results) and assume the model is correct, i.e. that the data are actually generated by this model. They also tend to break when the problem is badly conditioned (more features than samples).

Examples:

- *Lasso model selection: Cross-Validation / AIC / BIC*



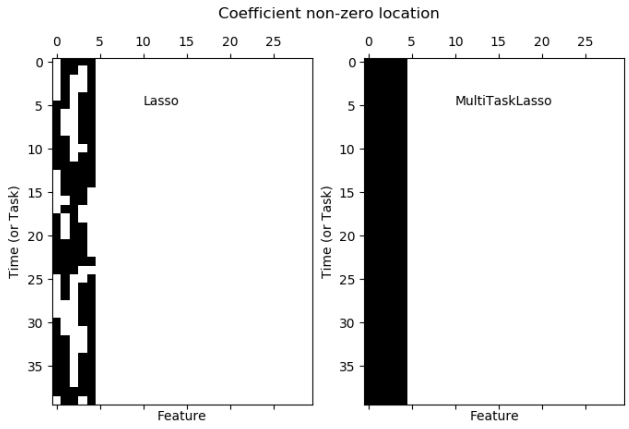
Comparison with the regularization parameter of SVM

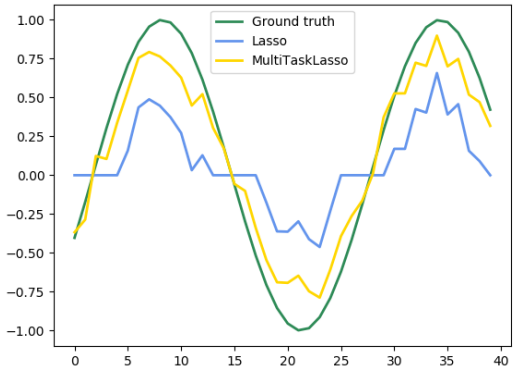
The equivalence between `alpha` and the regularization parameter of SVM, `C` is given by `alpha = 1 / C` or `alpha = 1 / (n_samples * C)`, depending on the estimator and the exact objective function optimized by the model.

Multi-task Lasso

The *`MultiTaskLasso`* is a linear model that estimates sparse coefficients for multiple regression problems jointly: `y` is a 2D array, of shape `(n_samples, n_tasks)`. The constraint is that the selected features are the same for all the regression problems, also called tasks.

The following figure compares the location of the non-zero entries in the coefficient matrix W obtained with a simple Lasso or a MultiTaskLasso. The Lasso estimates yield scattered non-zeros while the non-zeros of the MultiTaskLasso are full columns.





Fitting a time-series model, imposing that any active feature be active at all times.

Examples:

- *Joint feature selection with multi-task Lasso*

Mathematically, it consists of a linear model trained with a mixed ℓ_1 ℓ_2-norm for regularization. The objective function to minimize is:

$$\min_w \frac{1}{2n_{\text{samples}}}||XW - Y||^2_{\text{Fro}} + \alpha||W||_{21}$$

where Fro indicates the Frobenius norm

$$||A||_{\text{Fro}} = \sqrt{\sum_{ij} a_{ij}^2}$$

and ℓ_1 ℓ_2 reads

$$||A||_{21} = \sum_i \sqrt{\sum_j a_{ij}^2}.$$

The implementation in the class `MultiTaskLasso` uses coordinate descent as the algorithm to fit the coefficients.

Elastic-Net

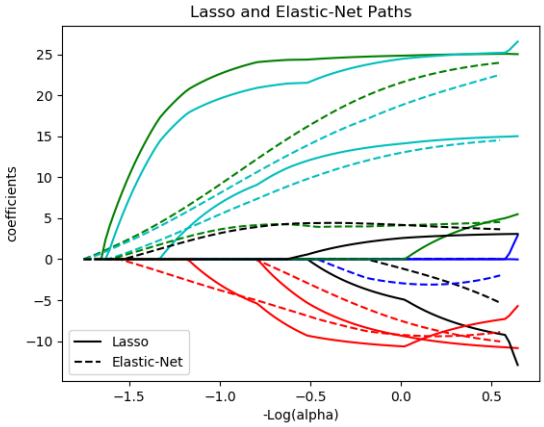
`ElasticNet` is a linear regression model trained with both ℓ_1 and ℓ_2-norm regularization of the coefficients. This combination allows for learning a sparse model where few of the weights are non-zero like `Lasso`, while still maintaining the regularization properties of `Ridge`. We control the convex combination of ℓ_1 and ℓ_2 using the `l1_ratio` parameter.

Elastic-net is useful when there are multiple features which are correlated with one another. Lasso is likely to pick one of these at random, while elastic-net is likely to pick both.

A practical advantage of trading-off between Lasso and Ridge is that it allows Elastic-Net to inherit some of Ridge's stability under rotation.

The objective function to minimize is in this case

$$\min_w \frac{1}{2n_{\text{samples}}}||Xw - y||_2^2 + \alpha\rho||w||_1 + \frac{\alpha(1-\rho)}{2}||w||_2^2$$



The class `ElasticNetCV` can be used to set the parameters `alpha` (α) and `l1_ratio` (ρ) by cross-validation.

Examples:

- *Lasso and Elastic Net for Sparse Signals*
- *Lasso and Elastic Net*

The following two references explain the iterations used in the coordinate descent solver of scikit-learn, as well as the duality gap computation used for convergence control.

References

- "Regularization Path For Generalized linear Models by Coordinate Descent", Friedman, Hastie & Tibshirani, J Stat Softw, 2010 (Paper).
- "An Interior-Point Method for Large-Scale L1-Regularized Least Squares," S. J. Kim, K. Koh, M. Lustig, S. Boyd and D. Gorinevsky, in IEEE Journal of Selected Topics in Signal Processing, 2007 (Paper)

Multi-task Elastic-Net

The `MultiTaskElasticNet` is an elastic-net model that estimates sparse coefficients for multiple regression problems jointly: `Y` is a 2D array of shape `(n_samples, n_tasks)`. The constraint is that the selected features are the same for all the regression problems, also called tasks.

Mathematically, it consists of a linear model trained with a mixed ℓ_1 ℓ_2-norm and ℓ_2-norm for regularization. The objective function to minimize is:

$$\min_W \frac{1}{2n_{\text{samples}}}||XW - Y||^2_{\text{Fro}} + \alpha\rho||W||_{21} + \frac{\alpha(1-\rho)}{2}||W||^2_{\text{Fro}}$$

The implementation in the class `MultiTaskElasticNet` uses coordinate descent as the algorithm to fit the coefficients.

The class `MultiTaskElasticNetCV` can be used to set the parameters `alpha` (α) and `l1_ratio` (ρ) by cross-validation.

Least Angle Regression

Least-angle regression (LARS) is a regression algorithm for high-dimensional data, developed by Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani. LARS is similar to forward stepwise regression. At each step, it finds the feature most correlated with the target. When there are multiple features having equal correlation, instead of continuing along the same feature, it proceeds in a direction equiangular between the features.

The advantages of LARS are:

- It is numerically efficient in contexts where the number of features is significantly greater than the number of samples.
- It is computationally just as fast as forward selection and has the same order of complexity as ordinary least squares.
- It produces a full piecewise linear solution path, which is useful in cross-validation or similar attempts to tune the model.
- If two features are almost equally correlated with the target, then their coefficients should increase at approximately the same rate. The algorithm thus behaves as intuition would expect, and also is more stable.
- It is easily modified to produce solutions for other estimators, like the Lasso.

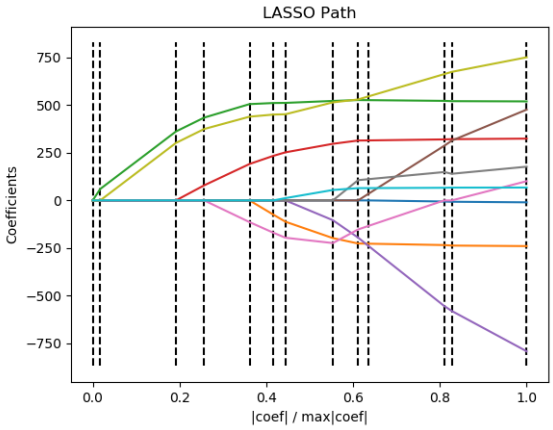
The disadvantages of the LARS method include:

- Because LARS is based upon an iterative refitting of the residuals, it would appear to be especially sensitive to the effects of noise. This problem is discussed in detail by Weisberg in the discussion section of the Efron et al. (2004) Annals of Statistics article.

The LARS model can be used using estimator *Lars*, or its low-level implementation *lars_path* or *lars_path_gram*.

LARS Lasso

LassoLars is a lasso model implemented using the LARS algorithm, and unlike the implementation based on coordinate descent, this yields the exact solution, which is piecewise linear as a function of the norm of its coefficients.




```
>>> from sklearn import linear_model
>>> reg = linear_model.LassoLars(alpha=.1)
>>> reg.fit([[0, 0], [1, 1]], [0, 1])
LassoLars(alpha=0.1)
```

(continues on next page)

(continued from previous page)

```
>>> reg.coef_
array([0.717157..., 0.        ])
```

Examples:

- *Lasso path using LARS*

The Lars algorithm provides the full path of the coefficients along the regularization parameter almost for free, thus a common operation is to retrieve the path with one of the functions `lars_path` or `lars_path_gram`.

Mathematical formulation

The algorithm is similar to forward stepwise regression, but instead of including features at each step, the estimated coefficients are increased in a direction equiangular to each one's correlations with the residual.

Instead of giving a vector result, the LARS solution consists of a curve denoting the solution for each value of the ℓ_1 norm of the parameter vector. The full coefficients path is stored in the array `coef_path_`, which has size (n_features, max_features+1). The first column is always zero.

References:

- Original Algorithm is detailed in the paper Least Angle Regression by Hastie et al.

Orthogonal Matching Pursuit (OMP)

`OrthogonalMatchingPursuit` and `orthogonal_mp` implements the OMP algorithm for approximating the fit of a linear model with constraints imposed on the number of non-zero coefficients (ie. the ℓ_0 pseudo-norm).

Being a forward feature selection method like *Least Angle Regression*, orthogonal matching pursuit can approximate the optimum solution vector with a fixed number of non-zero elements:

$$\arg\min_{\gamma} ||y - X\gamma||_2^2 \text{ subject to } ||\gamma||_0 \leq n_{\text{nonzero\_coefs}}$$

Alternatively, orthogonal matching pursuit can target a specific error instead of a specific number of non-zero coefficients. This can be expressed as:

$$\arg\min_{\gamma} ||\gamma||_0 \text{ subject to } ||y - X\gamma||_2^2 \leq \text{tol}$$

OMP is based on a greedy algorithm that includes at each step the atom most highly correlated with the current residual. It is similar to the simpler matching pursuit (MP) method, but better in that at each iteration, the residual is recomputed using an orthogonal projection on the space of the previously chosen dictionary elements.

Examples:

- *Orthogonal Matching Pursuit*

References:

- https://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf
- Matching pursuits with time-frequency dictionaries, S. G. Mallat, Z. Zhang,

Bayesian Regression

Bayesian regression techniques can be used to include regularization parameters in the estimation procedure: the regularization parameter is not set in a hard sense but tuned to the data at hand.

This can be done by introducing uninformative priors over the hyper parameters of the model. The ℓ_2 regularization used in *Ridge regression and classification* is equivalent to finding a maximum a posteriori estimation under a Gaussian prior over the coefficients w with precision λ^{-1}. Instead of setting `lambda` manually, it is possible to treat it as a random variable to be estimated from the data.

To obtain a fully probabilistic model, the output y is assumed to be Gaussian distributed around Xw:

$$p(y|X, w, \alpha) = \mathcal{N}(y|Xw, \alpha)$$

where α is again treated as a random variable that is to be estimated from the data.

The advantages of Bayesian Regression are:

- It adapts to the data at hand.
- It can be used to include regularization parameters in the estimation procedure.

The disadvantages of Bayesian regression include:

- Inference of the model can be time consuming.

References

- A good introduction to Bayesian methods is given in C. Bishop: Pattern Recognition and Machine learning
- Original Algorithm is detailed in the book `Bayesian learning for neural networks` by Radford M. Neal

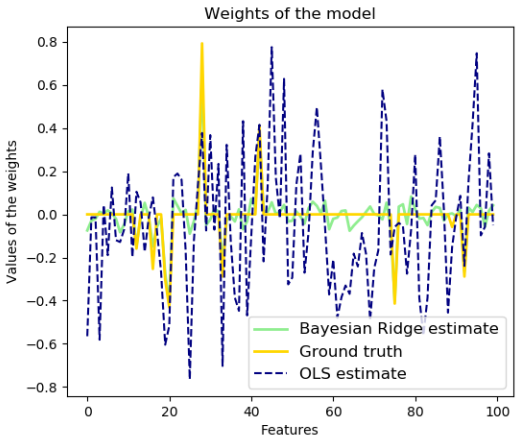
Bayesian Ridge Regression

`BayesianRidge` estimates a probabilistic model of the regression problem as described above. The prior for the coefficient w is given by a spherical Gaussian:

$$p(w|\lambda) = \mathcal{N}(w|0, \lambda^{-1}\mathbf{I}_p)$$

The priors over α and λ are chosen to be gamma distributions, the conjugate prior for the precision of the Gaussian. The resulting model is called *Bayesian Ridge Regression*, and is similar to the classical `Ridge`.

The parameters w, α and λ are estimated jointly during the fit of the model, the regularization parameters α and λ being estimated by maximizing the *log marginal likelihood*. The scikit-learn implementation is based on the algorithm described in Appendix A of (Tipping, 2001) where the update of the parameters α and λ is done as suggested in (MacKay, 1992). The initial value of the maximization procedure can be set with the hyperparameters `alpha_init` and `lambda_init`.



There are four more hyperparameters, α_1, α_2, λ_1 and λ_2 of the gamma prior distributions over α and λ. These are usually chosen to be *non-informative*. By default $\alpha_1 = \alpha_2 = \lambda_1 = \lambda_2 = 10^{-6}$.

Bayesian Ridge Regression is used for regression:

```
>>> from sklearn import linear_model
>>> X = [[0., 0.], [1., 1.], [2., 2.], [3., 3.]]
>>> Y = [0., 1., 2., 3.]
>>> reg = linear_model.BayesianRidge()
>>> reg.fit(X, Y)
BayesianRidge()
```

After being fitted, the model can then be used to predict new values:

```
>>> reg.predict([[1, 0.]])
array([0.50000013])
```

The coefficients w of the model can be accessed:

```
>>> reg.coef_
array([0.49999993, 0.49999993])
```

Due to the Bayesian framework, the weights found are slightly different to the ones found by *Ordinary Least Squares*. However, Bayesian Ridge Regression is more robust to ill-posed problems.

Examples:

- *Bayesian Ridge Regression*
- *Curve Fitting with Bayesian Ridge Regression*

References:

- Section 3.3 in Christopher M. Bishop: Pattern Recognition and Machine Learning, 2006
- David J. C. MacKay, Bayesian Interpolation, 1992.
- Michael E. Tipping, Sparse Bayesian Learning and the Relevance Vector Machine, 2001.

Automatic Relevance Determination - ARD

`ARDRegression` is very similar to *Bayesian Ridge Regression*, but can lead to sparser coefficients w[1][2]. `ARDRegression` poses a different prior over w, by dropping the assumption of the Gaussian being spherical.

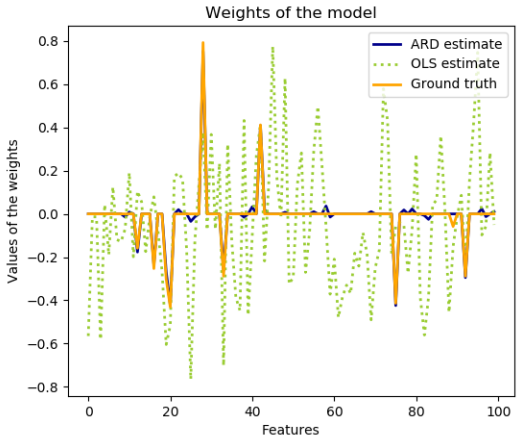
Instead, the distribution over w is assumed to be an axis-parallel, elliptical Gaussian distribution.

This means each coefficient w_i is drawn from a Gaussian distribution, centered on zero and with a precision λ_i:

$$p(w|\lambda) = \mathcal{N}(w|0, A^{-1})$$

with $\text{diag}(A) = \lambda = \{\lambda_1, ..., \lambda_p\}$.

In contrast to *Bayesian Ridge Regression*, each coordinate of w_i has its own standard deviation λ_i. The prior over all λ_i is chosen to be the same gamma distribution given by hyperparameters λ_1 and λ_2.



ARD is also known in the literature as *Sparse Bayesian Learning* and *Relevance Vector Machine*[3][4].

Examples:

- *Automatic Relevance Determination Regression (ARD)*

References:

Logistic regression

Logistic regression, despite its name, is a linear model for classification rather than regression. Logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

Logistic regression is implemented in `LogisticRegression`. This implementation can fit binary, One-vs-Rest, or multinomial logistic regression with optional ℓ_1, ℓ_2 or Elastic-Net regularization.

[1] Christopher M. Bishop: Pattern Recognition and Machine Learning, Chapter 7.2.1
[2] David Wipf and Srikantan Nagarajan: A new view of automatic relevance determination
[3] Michael E. Tipping: Sparse Bayesian Learning and the Relevance Vector Machine
[4] Tristan Fletcher: Relevance Vector Machines explained

Note: Regularization is applied by default, which is common in machine learning but not in statistics. Another advantage of regularization is that it improves numerical stability. No regularization amounts to setting C to a very high value.

As an optimization problem, binary class ℓ_2 penalized logistic regression minimizes the following cost function:

$$\min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1).$$

Similarly, ℓ_1 regularized logistic regression solves the following optimization problem:

$$\min_{w,c} \|w\|_1 + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1).$$

Elastic-Net regularization is a combination of ℓ_1 and ℓ_2, and minimizes the following cost function:

$$\min_{w,c} \frac{1-\rho}{2} w^T w + \rho \|w\|_1 + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1),$$

where ρ controls the strength of ℓ_1 regularization vs. ℓ_2 regularization (it corresponds to the `l1_ratio` parameter).

Note that, in this notation, it's assumed that the target y_i takes values in the set $-1, 1$ at trial i. We can also see that Elastic-Net is equivalent to ℓ_1 when $\rho = 1$ and equivalent to ℓ_2 when $\rho = 0$.

The solvers implemented in the class *LogisticRegression* are "liblinear", "newton-cg", "lbfgs", "sag" and "saga":

The solver "liblinear" uses a coordinate descent (CD) algorithm, and relies on the excellent C++ LIBLINEAR library, which is shipped with scikit-learn. However, the CD algorithm implemented in liblinear cannot learn a true multinomial (multiclass) model; instead, the optimization problem is decomposed in a "one-vs-rest" fashion so separate binary classifiers are trained for all classes. This happens under the hood, so *LogisticRegression* instances using this solver behave as multiclass classifiers. For ℓ_1 regularization *sklearn.svm.l1_min_c* allows to calculate the lower bound for C in order to get a non "null" (all feature weights to zero) model.

The "lbfgs", "sag" and "newton-cg" solvers only support ℓ_2 regularization or no regularization, and are found to converge faster for some high-dimensional data. Setting `multi_class` to "multinomial" with these solvers learns a true multinomial logistic regression model[5], which means that its probability estimates should be better calibrated than the default "one-vs-rest" setting.

The "sag" solver uses Stochastic Average Gradient descent[6]. It is faster than other solvers for large datasets, when both the number of samples and the number of features are large.

The "saga" solver[7] is a variant of "sag" that also supports the non-smooth `penalty="l1"`. This is therefore the solver of choice for sparse multinomial logistic regression. It is also the only solver that supports `penalty="elasticnet"`.

The "lbfgs" is an optimization algorithm that approximates the Broyden–Fletcher–Goldfarb–Shanno algorithm[8], which belongs to quasi-Newton methods. The "lbfgs" solver is recommended for use for small data-sets but for larger datasets its performance suffers.[9]

The following table summarizes the penalties supported by each solver:

[5] Christopher M. Bishop: Pattern Recognition and Machine Learning, Chapter 4.3.4

[6] Mark Schmidt, Nicolas Le Roux, and Francis Bach: Minimizing Finite Sums with the Stochastic Average Gradient.

[7] Aaron Defazio, Francis Bach, Simon Lacoste-Julien: SAGA: A Fast Incremental Gradient Method With Support for Non-Strongly Convex Composite Objectives.

[8] https://en.wikipedia.org/wiki/Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithm

[9] "Performance Evaluation of Lbfgs vs other solvers"

	Solvers				
Penalties	**'liblinear'**	**'lbfgs'**	**'newton-cg'**	**'sag'**	**'saga'**
Multinomial + L2 penalty	no	yes	yes	yes	yes
OVR + L2 penalty	yes	yes	yes	yes	yes
Multinomial + L1 penalty	no	no	no	no	yes
OVR + L1 penalty	yes	no	no	no	yes
Elastic-Net	no	no	no	no	yes
No penalty ('none')	no	yes	yes	yes	yes
Behaviors					
Penalize the intercept (bad)	yes	no	no	no	no
Faster for large datasets	no	no	no	yes	yes
Robust to unscaled datasets	yes	yes	yes	no	no

The "lbfgs" solver is used by default for its robustness. For large datasets the "saga" solver is usually faster. For large dataset, you may also consider using `SGDClassifier` with 'log' loss, which might be even faster but requires more tuning.

Examples:

- *L1 Penalty and Sparsity in Logistic Regression*
- *Regularization path of L1- Logistic Regression*
- *Plot multinomial and One-vs-Rest Logistic Regression*
- *Multiclass sparse logistic regression on 20newgroups*
- *MNIST classification using multinomial logistic + L1*

Differences from liblinear:

There might be a difference in the scores obtained between `LogisticRegression` with `solver=liblinear` or `LinearSVC` and the external liblinear library directly, when `fit_intercept=False` and the fit `coef_` (or) the data to be predicted are zeroes. This is because for the sample(s) with `decision_function` zero, `LogisticRegression` and `LinearSVC` predict the negative class, while liblinear predicts the positive class. Note that a model with `fit_intercept=False` and having many samples with `decision_function` zero, is likely to be a underfit, bad model and you are advised to set `fit_intercept=True` and increase the intercept_scaling.

Note: Feature selection with sparse logistic regression

A logistic regression with ℓ_1 penalty yields sparse models, and can thus be used to perform feature selection, as detailed in *L1-based feature selection*.

Note: P-value estimation

It is possible to obtain the p-values and confidence intervals for coefficients in cases of regression without penalization. The `statsmodels package <https://pypi.org/project/statsmodels/>` natively supports this. Within sklearn, one could use bootstrapping instead as well.

LogisticRegressionCV implements Logistic Regression with built-in cross-validation support, to find the optimal `C` and `l1_ratio` parameters according to the `scoring` attribute. The "newton-cg", "sag", "saga" and "lbfgs" solvers are found to be faster for high-dimensional dense data, due to warm-starting (see *Glossary*).

References:

Stochastic Gradient Descent - SGD

Stochastic gradient descent is a simple yet very efficient approach to fit linear models. It is particularly useful when the number of samples (and the number of features) is very large. The `partial_fit` method allows online/out-of-core learning.

The classes *SGDClassifier* and *SGDRegressor* provide functionality to fit linear models for classification and regression using different (convex) loss functions and different penalties. E.g., with `loss="log"`, *SGDClassifier* fits a logistic regression model, while with `loss="hinge"` it fits a linear support vector machine (SVM).

References

- *Stochastic Gradient Descent*

Perceptron

The *Perceptron* is another simple classification algorithm suitable for large scale learning. By default:

- It does not require a learning rate.
- It is not regularized (penalized).
- It updates its model only on mistakes.

The last characteristic implies that the Perceptron is slightly faster to train than SGD with the hinge loss and that the resulting models are sparser.

Passive Aggressive Algorithms

The passive-aggressive algorithms are a family of algorithms for large-scale learning. They are similar to the Perceptron in that they do not require a learning rate. However, contrary to the Perceptron, they include a regularization parameter `C`.

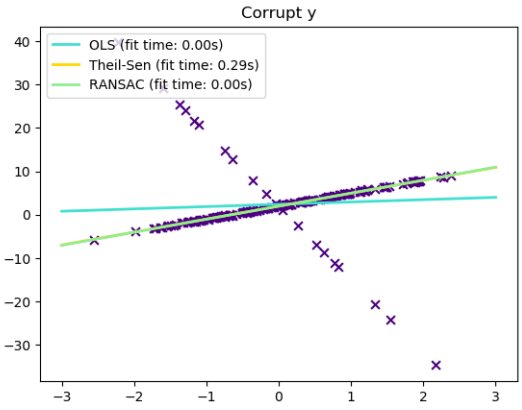
For classification, *PassiveAggressiveClassifier* can be used with `loss='hinge'` (PA-I) or `loss='squared_hinge'` (PA-II). For regression, *PassiveAggressiveRegressor* can be used with `loss='epsilon_insensitive'` (PA-I) or `loss='squared_epsilon_insensitive'` (PA-II).

References:

- "Online Passive-Aggressive Algorithms" K. Crammer, O. Dekel, J. Keshat, S. Shalev-Shwartz, Y. Singer - JMLR 7 (2006)

Robustness regression: outliers and modeling errors

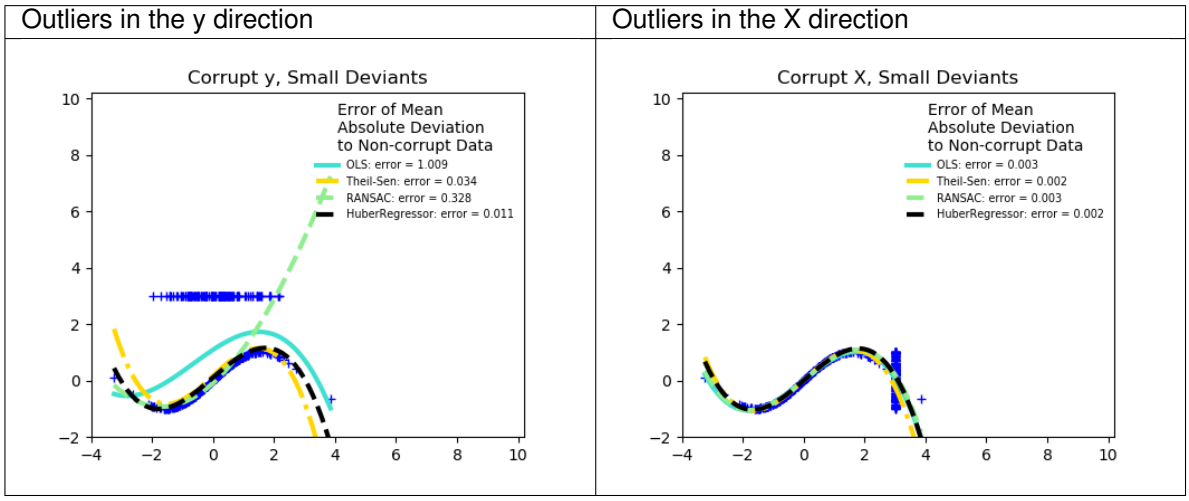
Robust regression aims to fit a regression model in the presence of corrupt data: either outliers, or error in the model.



Different scenario and useful concepts

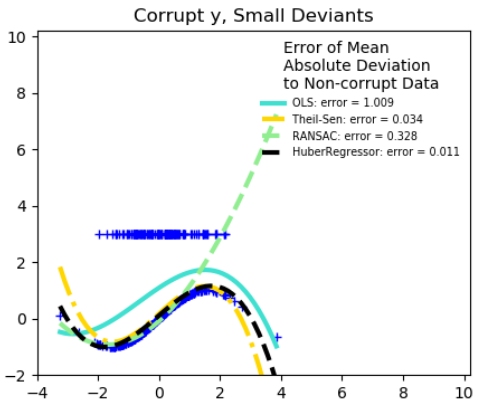
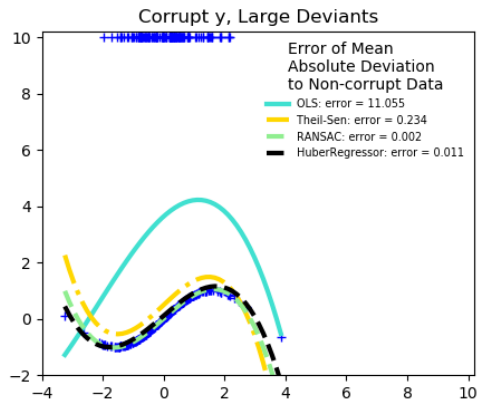
There are different things to keep in mind when dealing with data corrupted by outliers:

- **Outliers in X or in y**?



- **Fraction of outliers versus amplitude of error**

  The number of outlying points matters, but also how much they are outliers.

Small outliers	Large outliers
	

An important notion of robust fitting is that of breakdown point: the fraction of data that can be outlying for the fit to start missing the inlying data.

Note that in general, robust fitting in high-dimensional setting (large `n_features`) is very hard. The robust models here will probably not work in these settings.

Trade-offs: which estimator?

Scikit-learn provides 3 robust regression estimators: *RANSAC*, *Theil Sen* and *HuberRegressor*.

- *HuberRegressor* should be faster than *RANSAC* and *Theil Sen* unless the number of samples are very large, i.e `n_samples` >> `n_features`. This is because *RANSAC* and *Theil Sen* fit on smaller subsets of the data. However, both *Theil Sen* and *RANSAC* are unlikely to be as robust as *HuberRegressor* for the default parameters.
- *RANSAC* is faster than *Theil Sen* and scales much better with the number of samples.
- *RANSAC* will deal better with large outliers in the y direction (most common situation).
- *Theil Sen* will cope better with medium-size outliers in the X direction, but this property will disappear in high-dimensional settings.

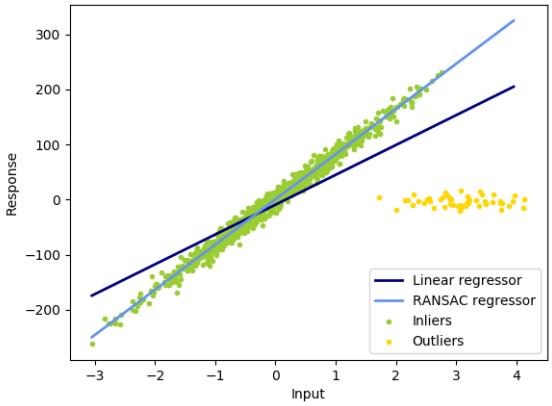
When in doubt, use *RANSAC*.

RANSAC: RANdom SAmple Consensus

RANSAC (RANdom SAmple Consensus) fits a model from random subsets of inliers from the complete data set.

RANSAC is a non-deterministic algorithm producing only a reasonable result with a certain probability, which is dependent on the number of iterations (see `max_trials` parameter). It is typically used for linear and non-linear regression problems and is especially popular in the field of photogrammetric computer vision.

The algorithm splits the complete input sample data into a set of inliers, which may be subject to noise, and outliers, which are e.g. caused by erroneous measurements or invalid hypotheses about the data. The resulting model is then estimated only from the determined inliers.



Details of the algorithm

Each iteration performs the following steps:

1. Select `min_samples` random samples from the original data and check whether the set of data is valid (see `is_data_valid`).
2. Fit a model to the random subset (`base_estimator.fit`) and check whether the estimated model is valid (see `is_model_valid`).
3. Classify all data as inliers or outliers by calculating the residuals to the estimated model (`base_estimator.predict(X) - y`) - all data samples with absolute residuals smaller than the `residual_threshold` are considered as inliers.
4. Save fitted model as best model if number of inlier samples is maximal. In case the current estimated model has the same number of inliers, it is only considered as the best model if it has better score.

These steps are performed either a maximum number of times (`max_trials`) or until one of the special stop criteria are met (see `stop_n_inliers` and `stop_score`). The final model is estimated using all inlier samples (consensus set) of the previously determined best model.

The `is_data_valid` and `is_model_valid` functions allow to identify and reject degenerate combinations of random sub-samples. If the estimated model is not needed for identifying degenerate cases, `is_data_valid` should be used as it is called prior to fitting the model and thus leading to better computational performance.

Examples:

- *Robust linear model estimation using RANSAC*
- *Robust linear estimator fitting*

References:

- https://en.wikipedia.org/wiki/RANSAC
- "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography" Martin A. Fischler and Robert C. Bolles - SRI International (1981)
- "Performance Evaluation of RANSAC Family" Sunglok Choi, Taemin Kim and Wonpil Yu - BMVC (2009)

Theil-Sen estimator: generalized-median-based estimator

The `TheilSenRegressor` estimator uses a generalization of the median in multiple dimensions. It is thus robust to multivariate outliers. Note however that the robustness of the estimator decreases quickly with the dimensionality of the problem. It loses its robustness properties and becomes no better than an ordinary least squares in high dimension.

Examples:

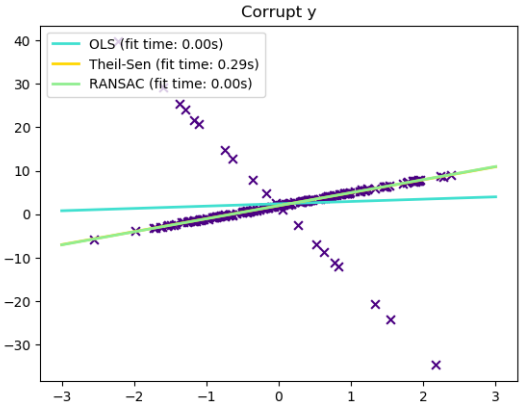
- *Theil-Sen Regression*
- *Robust linear estimator fitting*

References:

- https://en.wikipedia.org/wiki/Theil%E2%80%93Sen_estimator

Theoretical considerations

`TheilSenRegressor` is comparable to the *Ordinary Least Squares (OLS)* in terms of asymptotic efficiency and as an unbiased estimator. In contrast to OLS, Theil-Sen is a non-parametric method which means it makes no assumption about the underlying distribution of the data. Since Theil-Sen is a median-based estimator, it is more robust against corrupted data aka outliers. In univariate setting, Theil-Sen has a breakdown point of about 29.3% in case of a simple linear regression which means that it can tolerate arbitrary corrupted data of up to 29.3%.



The implementation of `TheilSenRegressor` in scikit-learn follows a generalization to a multivariate linear regression model[10] using the spatial median which is a generalization of the median to multiple dimensions[11].

In terms of time and space complexity, Theil-Sen scales according to

$$\binom{n_{\text{samples}}}{n_{\text{subsamples}}}$$

[10] Xin Dang, Hanxiang Peng, Xueqin Wang and Heping Zhang: Theil-Sen Estimators in a Multiple Linear Regression Model.

[11] T. Kärkkäinen and S. Äyrämö: On Computation of Spatial Median for Robust Data Mining.

which makes it infeasible to be applied exhaustively to problems with a large number of samples and features. Therefore, the magnitude of a subpopulation can be chosen to limit the time and space complexity by considering only a random subset of all possible combinations.

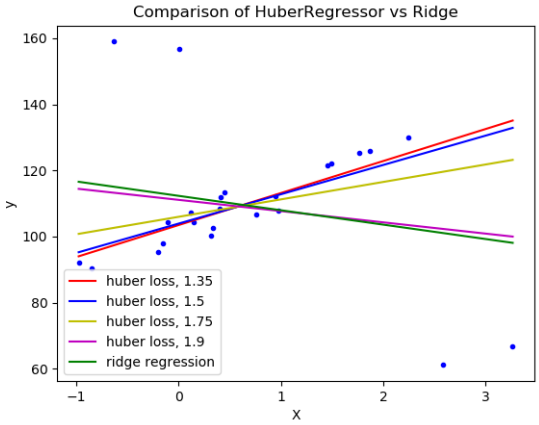
Examples:

- *Theil-Sen Regression*

References:

Huber Regression

The `HuberRegressor` is different to `Ridge` because it applies a linear loss to samples that are classified as outliers. A sample is classified as an inlier if the absolute error of that sample is lesser than a certain threshold. It differs from `TheilSenRegressor` and `RANSACRegressor` because it does not ignore the effect of the outliers but gives a lesser weight to them.



The loss function that `HuberRegressor` minimizes is given by

$$\min_{w,\sigma} \sum_{i=1}^{n} \left(\sigma + H_\epsilon \left(\frac{X_i w - y_i}{\sigma} \right) \sigma \right) + \alpha ||w||_2{}^2$$

where

$$H_\epsilon(z) = \begin{cases} z^2, & \text{if } |z| < \epsilon, \\ 2\epsilon|z| - \epsilon^2, & \text{otherwise} \end{cases}$$

It is advised to set the parameter `epsilon` to 1.35 to achieve 95% statistical efficiency.

Notes

The `HuberRegressor` differs from using `SGDRegressor` with loss set to `huber` in the following ways.

- `HuberRegressor` is scaling invariant. Once `epsilon` is set, scaling `X` and `y` down or up by different values would produce the same robustness to outliers as before. as compared to `SGDRegressor` where `epsilon` has to be set again when `X` and `y` are scaled.
- `HuberRegressor` should be more efficient to use on data with small number of samples while `SGDRegressor` needs a number of passes on the training data to produce the same robustness.

Examples:

- *HuberRegressor vs Ridge on dataset with strong outliers*

References:

- Peter J. Huber, Elvezio M. Ronchetti: Robust Statistics, Concomitant scale estimates, pg 172

Note that this estimator is different from the R implementation of Robust Regression (http://www.ats.ucla.edu/stat/r/dae/rreg.htm) because the R implementation does a weighted least squares implementation with weights given to each sample on the basis of how much the residual is greater than a certain threshold.

Polynomial regression: extending linear models with basis functions

One common pattern within machine learning is to use linear models trained on nonlinear functions of the data. This approach maintains the generally fast performance of linear methods, while allowing them to fit a much wider range of data.

For example, a simple linear regression can be extended by constructing **polynomial features** from the coefficients. In the standard linear regression case, you might have a model that looks like this for two-dimensional data:

$$\hat{y}(w, x) = w_0 + w_1 x_1 + w_2 x_2$$

If we want to fit a paraboloid to the data instead of a plane, we can combine the features in second-order polynomials, so that the model looks like this:

$$\hat{y}(w, x) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2$$

The (sometimes surprising) observation is that this is *still a linear model*: to see this, imagine creating a new set of features

$$z = [x_1, x_2, x_1 x_2, x_1^2, x_2^2]$$

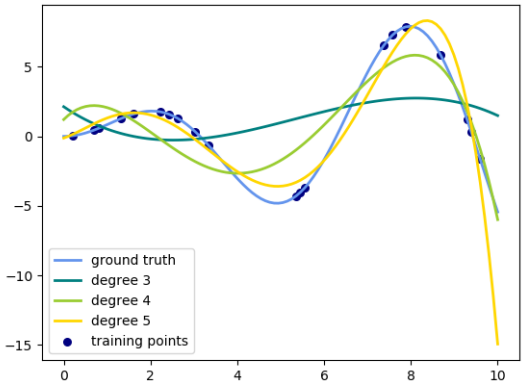
With this re-labeling of the data, our problem can be written

$$\hat{y}(w, z) = w_0 + w_1 z_1 + w_2 z_2 + w_3 z_3 + w_4 z_4 + w_5 z_5$$

We see that the resulting *polynomial regression* is in the same class of linear models we considered above (i.e. the model is linear in w) and can be solved by the same techniques. By considering linear fits within a higher-dimensional space built with these basis functions, the model has the flexibility to fit a much broader range of data.

Here is an example of applying this idea to one-dimensional data, using polynomial features of varying degrees:

This figure is created using the `PolynomialFeatures` transformer, which transforms an input data matrix into a new data matrix of a given degree. It can be used as follows:




```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> import numpy as np
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> poly = PolynomialFeatures(degree=2)
>>> poly.fit_transform(X)
array([[ 1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  2.,  3.,  4.,  6.,  9.],
       [ 1.,  4.,  5., 16., 20., 25.]])
```

The features of X have been transformed from $[x_1, x_2]$ to $[1, x_1, x_2, x_1^2, x_1x_2, x_2^2]$, and can now be used within any linear model.

This sort of preprocessing can be streamlined with the *Pipeline* tools. A single object representing a simple polynomial regression can be created and used as follows:

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
>>> model = Pipeline([('poly', PolynomialFeatures(degree=3)),
...                   ('linear', LinearRegression(fit_intercept=False))])
>>> # fit to an order-3 polynomial data
>>> x = np.arange(5)
>>> y = 3 - 2 * x + x ** 2 - x ** 3
>>> model = model.fit(x[:, np.newaxis], y)
>>> model.named_steps['linear'].coef_
array([ 3., -2.,  1., -1.])
```

The linear model trained on polynomial features is able to exactly recover the input polynomial coefficients.

In some cases it's not necessary to include higher powers of any single feature, but only the so-called *interaction features* that multiply together at most d distinct features. These can be gotten from `PolynomialFeatures` with the setting `interaction_only=True`.

For example, when dealing with boolean features, $x_i^n = x_i$ for all n and is therefore useless; but $x_i x_j$ represents the conjunction of two booleans. This way, we can solve the XOR problem with a linear classifier:

```
>>> from sklearn.linear_model import Perceptron
>>> from sklearn.preprocessing import PolynomialFeatures
>>> import numpy as np
>>> X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
>>> y = X[:, 0] ^ X[:, 1]
>>> y
array([0, 1, 1, 0])
>>> X = PolynomialFeatures(interaction_only=True).fit_transform(X).astype(int)
>>> X
array([[1, 0, 0, 0],
       [1, 0, 1, 0],
       [1, 1, 0, 0],
       [1, 1, 1, 1]])
>>> clf = Perceptron(fit_intercept=False, max_iter=10, tol=None,
...                  shuffle=False).fit(X, y)
```

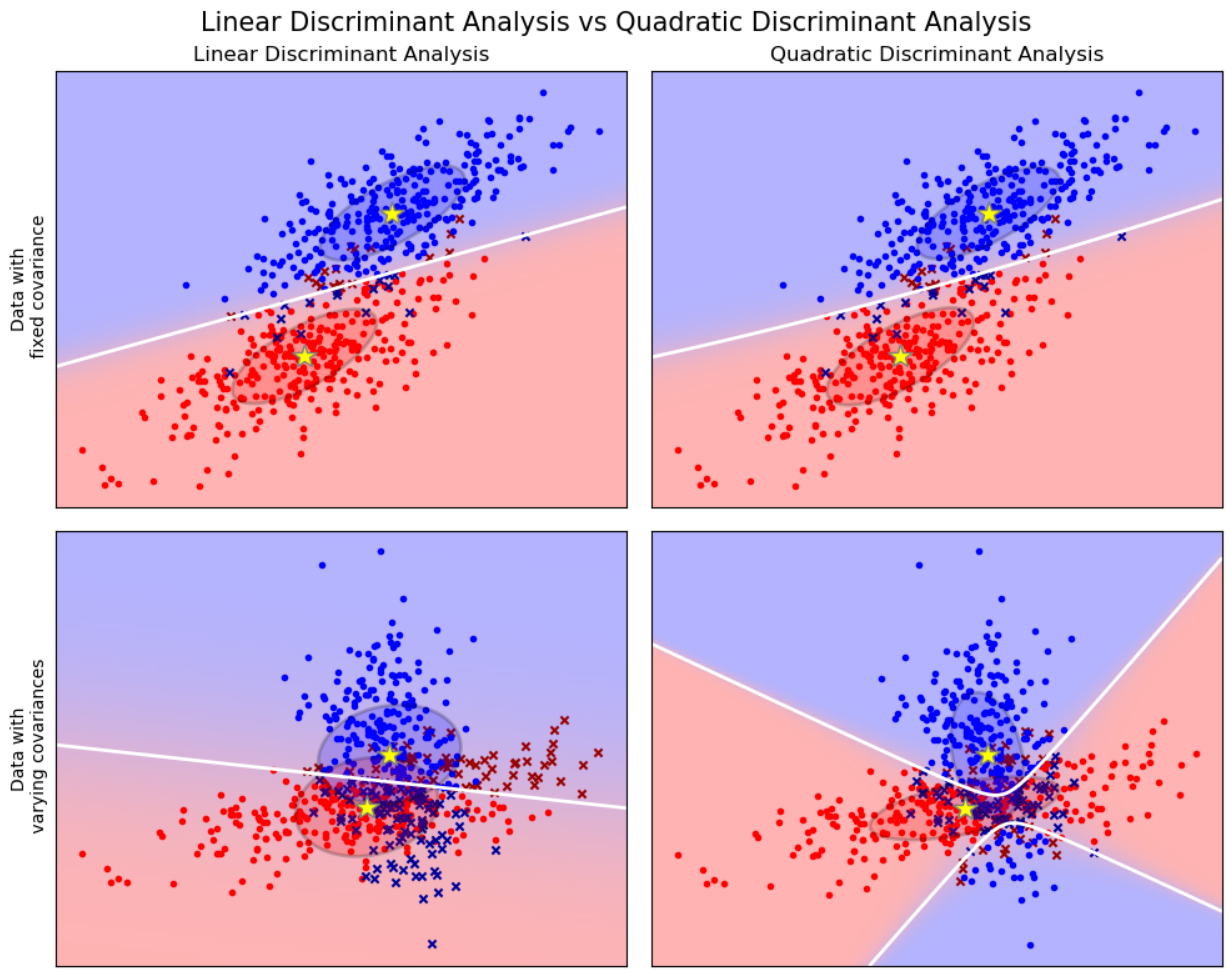
And the classifier "predictions" are perfect:

```
>>> clf.predict(X)
array([0, 1, 1, 0])
>>> clf.score(X, y)
1.0
```

4.1.2 Linear and Quadratic Discriminant Analysis

Linear Discriminant Analysis (*discriminant_analysis.LinearDiscriminantAnalysis*) and Quadratic Discriminant Analysis (*discriminant_analysis.QuadraticDiscriminantAnalysis*) are two classic classifiers, with, as their names suggest, a linear and a quadratic decision surface, respectively.

These classifiers are attractive because they have closed-form solutions that can be easily computed, are inherently multiclass, have proven to work well in practice, and have no hyperparameters to tune.



The plot shows decision boundaries for Linear Discriminant Analysis and Quadratic Discriminant Analysis. The bottom row demonstrates that Linear Discriminant Analysis can only learn linear boundaries, while Quadratic Discriminant Analysis can learn quadratic boundaries and is therefore more flexible.

Examples:

Linear and Quadratic Discriminant Analysis with covariance ellipsoid: Comparison of LDA and QDA on synthetic data.

Dimensionality reduction using Linear Discriminant Analysis

`discriminant_analysis.LinearDiscriminantAnalysis` can be used to perform supervised dimensionality reduction, by projecting the input data to a linear subspace consisting of the directions which maximize the separation between classes (in a precise sense discussed in the mathematics section below). The dimension of the output is necessarily less than the number of classes, so this is, in general, a rather strong dimensionality reduction, and only makes sense in a multiclass setting.

This is implemented in `discriminant_analysis.LinearDiscriminantAnalysis.transform`. The desired dimensionality can be set using the `n_components` constructor parameter. This parameter has no influence on `discriminant_analysis.LinearDiscriminantAnalysis.fit` or `discriminant_analysis.`

LinearDiscriminantAnalysis.predict.

Examples:

Comparison of LDA and PCA 2D projection of Iris dataset: Comparison of LDA and PCA for dimensionality reduction of the Iris dataset

Mathematical formulation of the LDA and QDA classifiers

Both LDA and QDA can be derived from simple probabilistic models which model the class conditional distribution of the data $P(X|y=k)$ for each class k. Predictions can then be obtained by using Bayes' rule:

$$P(y=k|X) = \frac{P(X|y=k)P(y=k)}{P(X)} = \frac{P(X|y=k)P(y=k)}{\sum_l P(X|y=l)\cdot P(y=l)}$$

and we select the class k which maximizes this conditional probability.

More specifically, for linear and quadratic discriminant analysis, $P(X|y)$ is modeled as a multivariate Gaussian distribution with density:

$$P(X|y=k) = \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(X-\mu_k)^t\Sigma_k^{-1}(X-\mu_k)\right)$$

where d is the number of features.

To use this model as a classifier, we just need to estimate from the training data the class priors $P(y=k)$ (by the proportion of instances of class k), the class means μ_k (by the empirical sample class means) and the covariance matrices (either by the empirical sample class covariance matrices, or by a regularized estimator: see the section on shrinkage below).

In the case of LDA, the Gaussians for each class are assumed to share the same covariance matrix: $\Sigma_k = \Sigma$ for all k. This leads to linear decision surfaces, which can be seen by comparing the log-probability ratios $\log[P(y=k|X)/P(y=l|X)]$:

$$\log\left(\frac{P(y=k|X)}{P(y=l|X)}\right) = \log\left(\frac{P(X|y=k)P(y=k)}{P(X|y=l)P(y=l)}\right) = 0 \Leftrightarrow$$
$$(\mu_k-\mu_l)^t\Sigma^{-1}X = \frac{1}{2}(\mu_k^t\Sigma^{-1}\mu_k - \mu_l^t\Sigma^{-1}\mu_l) - \log\frac{P(y=k)}{P(y=l)}$$

In the case of QDA, there are no assumptions on the covariance matrices Σ_k of the Gaussians, leading to quadratic decision surfaces. See[3] for more details.

Note: Relation with Gaussian Naive Bayes

If in the QDA model one assumes that the covariance matrices are diagonal, then the inputs are assumed to be conditionally independent in each class, and the resulting classifier is equivalent to the Gaussian Naive Bayes classifier *naive_bayes.GaussianNB*.

Mathematical formulation of LDA dimensionality reduction

To understand the use of LDA in dimensionality reduction, it is useful to start with a geometric reformulation of the LDA classification rule explained above. We write K for the total number of target classes. Since in LDA we assume

[3] "The Elements of Statistical Learning", Hastie T., Tibshirani R., Friedman J., Section 4.3, p.106-119, 2008.

that all classes have the same estimated covariance Σ, we can rescale the data so that this covariance is the identity:

$$X^* = D^{-1/2}U^t X \text{ with } \Sigma = UDU^t$$

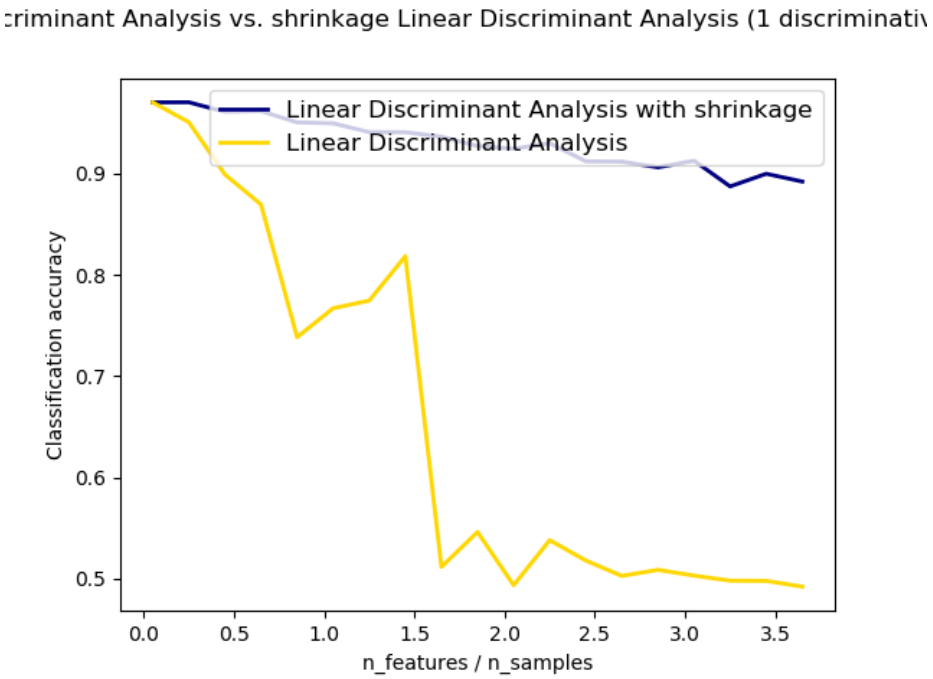
Then one can show that to classify a data point after scaling is equivalent to finding the estimated class mean μ_k^* which is closest to the data point in the Euclidean distance. But this can be done just as well after projecting on the $K-1$ affine subspace H_K generated by all the μ_k^* for all classes. This shows that, implicit in the LDA classifier, there is a dimensionality reduction by linear projection onto a $K-1$ dimensional space.

We can reduce the dimension even more, to a chosen L, by projecting onto the linear subspace H_L which maximizes the variance of the μ_k^* after projection (in effect, we are doing a form of PCA for the transformed class means μ_k^*). This L corresponds to the `n_components` parameter used in the *discriminant_analysis.LinearDiscriminantAnalysis.transform* method. See[3] for more details.

Shrinkage

Shrinkage is a tool to improve estimation of covariance matrices in situations where the number of training samples is small compared to the number of features. In this scenario, the empirical sample covariance is a poor estimator. Shrinkage LDA can be used by setting the `shrinkage` parameter of the *discriminant_analysis.LinearDiscriminantAnalysis* class to 'auto'. This automatically determines the optimal shrinkage parameter in an analytic way following the lemma introduced by Ledoit and Wolf[4]. Note that currently shrinkage only works when setting the `solver` parameter to 'lsqr' or 'eigen'.

The `shrinkage` parameter can also be manually set between 0 and 1. In particular, a value of 0 corresponds to no shrinkage (which means the empirical covariance matrix will be used) and a value of 1 corresponds to complete shrinkage (which means that the diagonal matrix of variances will be used as an estimate for the covariance matrix). Setting this parameter to a value between these two extrema will estimate a shrunk version of the covariance matrix.



[4] Ledoit O, Wolf M. Honey, I Shrunk the Sample Covariance Matrix. The Journal of Portfolio Management 30(4), 110-119, 2004.

Estimation algorithms

The default solver is 'svd'. It can perform both classification and transform, and it does not rely on the calculation of the covariance matrix. This can be an advantage in situations where the number of features is large. However, the 'svd' solver cannot be used with shrinkage.

The 'lsqr' solver is an efficient algorithm that only works for classification. It supports shrinkage.

The 'eigen' solver is based on the optimization of the between class scatter to within class scatter ratio. It can be used for both classification and transform, and it supports shrinkage. However, the 'eigen' solver needs to compute the covariance matrix, so it might not be suitable for situations with a high number of features.

Examples:

Normal and Shrinkage Linear Discriminant Analysis for classification: Comparison of LDA classifiers with and without shrinkage.

References:

4.1.3 Kernel ridge regression

Kernel ridge regression (KRR) [M2012] combines *Ridge regression and classification* (linear least squares with l2-norm regularization) with the kernel trick. It thus learns a linear function in the space induced by the respective kernel and the data. For non-linear kernels, this corresponds to a non-linear function in the original space.

The form of the model learned by `KernelRidge` is identical to support vector regression (`SVR`). However, different loss functions are used: KRR uses squared error loss while support vector regression uses ϵ-insensitive loss, both combined with l2 regularization. In contrast to `SVR`, fitting `KernelRidge` can be done in closed-form and is typically faster for medium-sized datasets. On the other hand, the learned model is non-sparse and thus slower than SVR, which learns a sparse model for $\epsilon > 0$, at prediction-time.

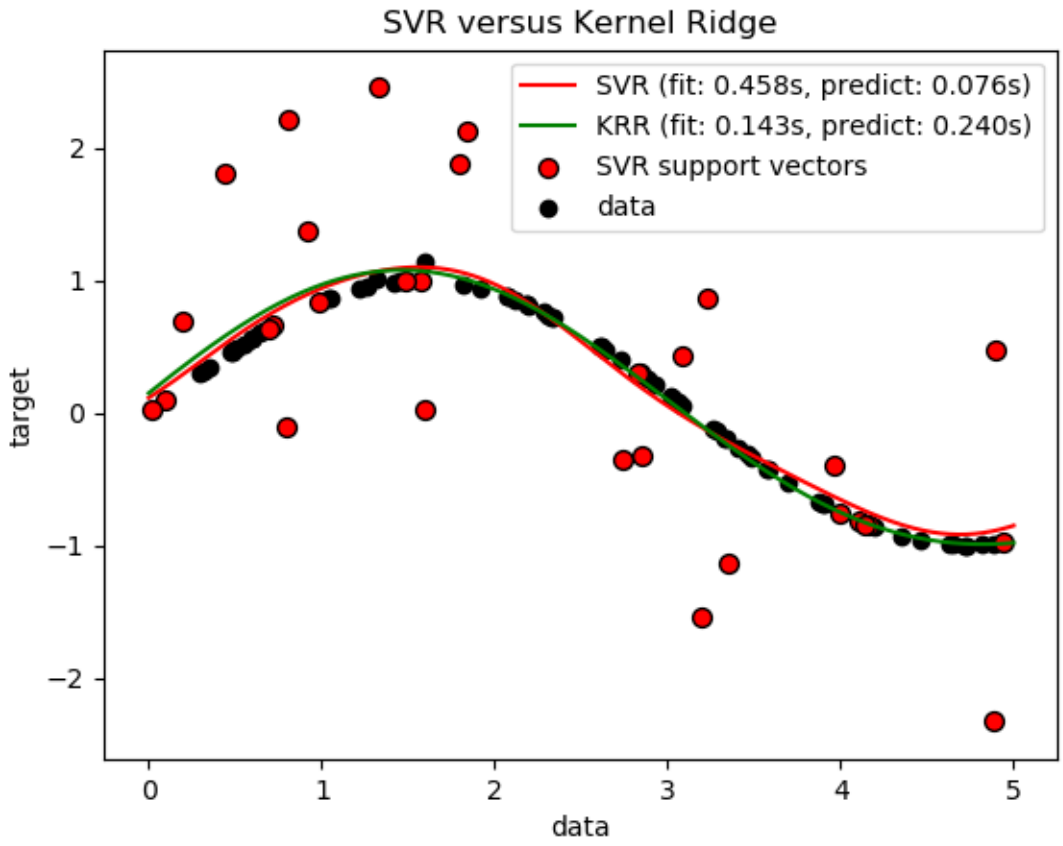
The following figure compares `KernelRidge` and `SVR` on an artificial dataset, which consists of a sinusoidal target function and strong noise added to every fifth datapoint. The learned model of `KernelRidge` and `SVR` is plotted, where both complexity/regularization and bandwidth of the RBF kernel have been optimized using grid-search. The learned functions are very similar; however, fitting `KernelRidge` is approx. seven times faster than fitting `SVR` (both with grid-search). However, prediction of 100000 target values is more than three times faster with SVR since it has learned a sparse model using only approx. 1/3 of the 100 training datapoints as support vectors.

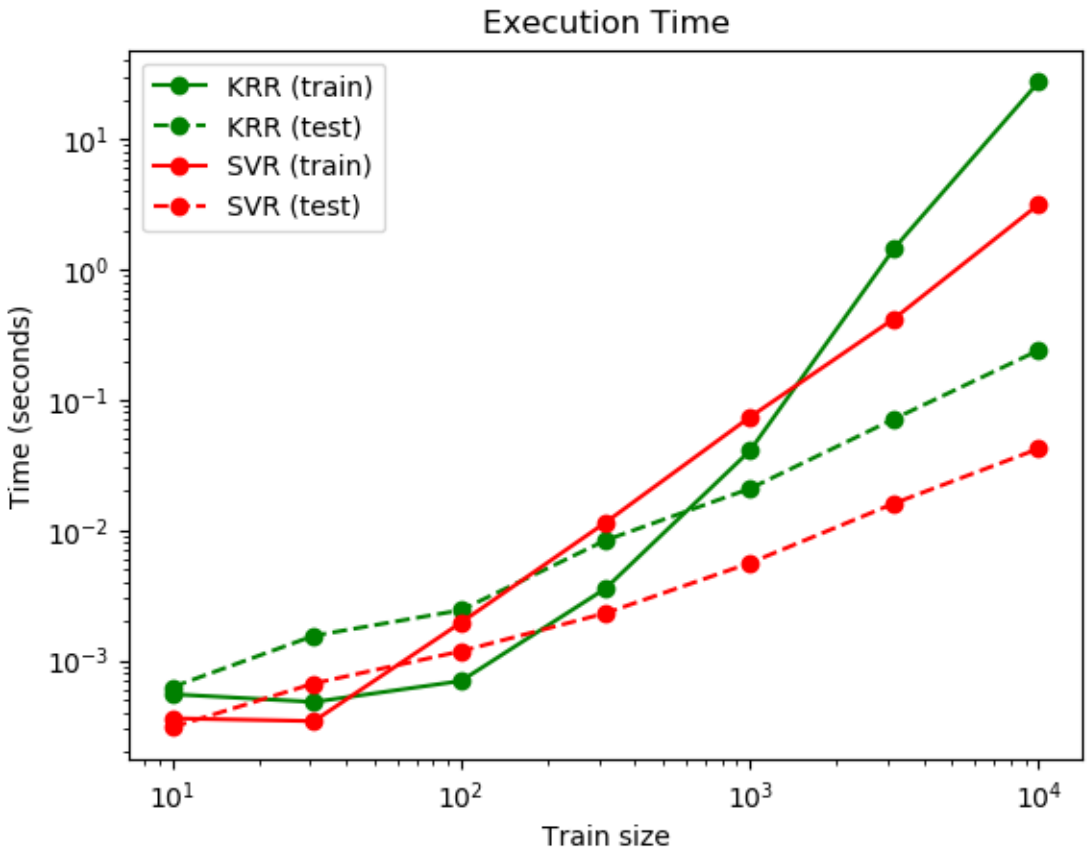
The next figure compares the time for fitting and prediction of `KernelRidge` and `SVR` for different sizes of the training set. Fitting `KernelRidge` is faster than `SVR` for medium-sized training sets (less than 1000 samples); however, for larger training sets `SVR` scales better. With regard to prediction time, `SVR` is faster than `KernelRidge` for all sizes of the training set because of the learned sparse solution. Note that the degree of sparsity and thus the prediction time depends on the parameters ϵ and C of the `SVR`; $\epsilon = 0$ would correspond to a dense model.

References:

4.1.4 Support Vector Machines

Support vector machines (SVMs) are a set of supervised learning methods used for *classification*, *regression* and *outliers detection*.


SVR versus Kernel Ridge
SVR (fit: 0.458s, predict: 0.076s)
KRR (fit: 0.143s, predict: 0.240s)
SVR support vectors
data
target
data
2
1
0
−1
−2
0
1
2
3
4
5


Execution Time
KRR (train)
KRR (test)
SVR (train)
SVR (test)
Time (seconds)
Train size
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{1}
10^{2}
10^{3}
10^{4}

The advantages of support vector machines are:

- Effective in high dimensional spaces.
- Still effective in cases where number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different *Kernel functions* can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

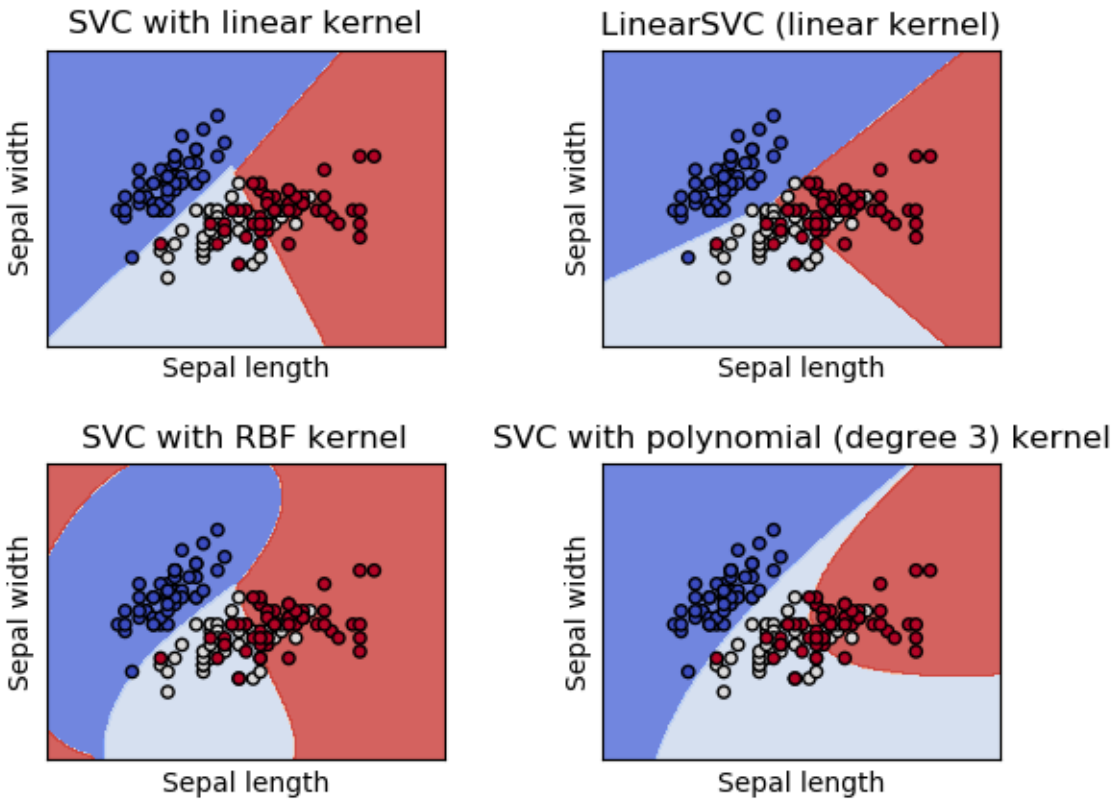
The disadvantages of support vector machines include:

- If the number of features is much greater than the number of samples, avoid over-fitting in choosing *Kernel functions* and regularization term is crucial.
- SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation (see *Scores and probabilities*, below).

The support vector machines in scikit-learn support both dense (`numpy.ndarray` and convertible to that by `numpy.asarray`) and sparse (any `scipy.sparse`) sample vectors as input. However, to use an SVM to make predictions for sparse data, it must have been fit on such data. For optimal performance, use C-ordered `numpy.ndarray` (dense) or `scipy.sparse.csr_matrix` (sparse) with `dtype=float64`.

Classification

SVC, *NuSVC* and *LinearSVC* are classes capable of performing multi-class classification on a dataset.



SVC and *NuSVC* are similar methods, but accept slightly different sets of parameters and have different mathematical formulations (see section *Mathematical formulation*). On the other hand, *LinearSVC* is another implementation of Support Vector Classification for the case of a linear kernel. Note that *LinearSVC* does not accept keyword `kernel`, as this is assumed to be linear. It also lacks some of the members of *SVC* and *NuSVC*, like `support_`.

As other classifiers, *SVC*, *NuSVC* and *LinearSVC* take as input two arrays: an array X of size `[n_samples, n_features]` holding the training samples, and an array y of class labels (strings or integers), size `[n_samples]`:

```
>>> from sklearn import svm
>>> X = [[0, 0], [1, 1]]
>>> y = [0, 1]
>>> clf = svm.SVC()
>>> clf.fit(X, y)
SVC()
```

After being fitted, the model can then be used to predict new values:

```
>>> clf.predict([[2., 2.]])
array([1])
```

SVMs decision function depends on some subset of the training data, called the support vectors. Some properties of these support vectors can be found in members `support_vectors_`, `support_` and `n_support`:

```
>>> # get support vectors
>>> clf.support_vectors_
array([[0., 0.],
       [1., 1.]])
>>> # get indices of support vectors
>>> clf.support_
array([0, 1]...)
>>> # get number of support vectors for each class
>>> clf.n_support_
array([1, 1]...)
```

Multi-class classification

SVC and *NuSVC* implement the "one-against-one" approach (Knerr et al., 1990) for multi- class classification. If `n_class` is the number of classes, then `n_class * (n_class - 1) / 2` classifiers are constructed and each one trains data from two classes. To provide a consistent interface with other classifiers, the `decision_function_shape` option allows to monotically transform the results of the "one-against-one" classifiers to a decision function of shape `(n_samples, n_classes)`.

```
>>> X = [[0], [1], [2], [3]]
>>> Y = [0, 1, 2, 3]
>>> clf = svm.SVC(decision_function_shape='ovo')
>>> clf.fit(X, Y)
SVC(decision_function_shape='ovo')
>>> dec = clf.decision_function([[1]])
>>> dec.shape[1] # 4 classes: 4*3/2 = 6
6
>>> clf.decision_function_shape = "ovr"
>>> dec = clf.decision_function([[1]])
>>> dec.shape[1] # 4 classes
4
```

On the other hand, *LinearSVC* implements "one-vs-the-rest" multi-class strategy, thus training n_class models. If there are only two classes, only one model is trained:

```
>>> lin_clf = svm.LinearSVC()
>>> lin_clf.fit(X, Y)
LinearSVC()
>>> dec = lin_clf.decision_function([[1]])
>>> dec.shape[1]
4
```

See *Mathematical formulation* for a complete description of the decision function.

Note that the *LinearSVC* also implements an alternative multi-class strategy, the so-called multi-class SVM formulated by Crammer and Singer, by using the option `multi_class='crammer_singer'`. This method is consistent, which is not true for one-vs-rest classification. In practice, one-vs-rest classification is usually preferred, since the results are mostly similar, but the runtime is significantly less.

For "one-vs-rest" *LinearSVC* the attributes `coef_` and `intercept_` have the shape `[n_class, n_features]` and `[n_class]` respectively. Each row of the coefficients corresponds to one of the `n_class` many "one-vs-rest" classifiers and similar for the intercepts, in the order of the "one" class.

In the case of "one-vs-one" *SVC*, the layout of the attributes is a little more involved. In the case of having a linear kernel, the attributes `coef_` and `intercept_` have the shape `[n_class * (n_class - 1) / 2, n_features]` and `[n_class * (n_class - 1) / 2]` respectively. This is similar to the layout for *LinearSVC* described above, with each row now corresponding to a binary classifier. The order for classes 0 to n is "0 vs 1", "0 vs 2" , ... "0 vs n", "1 vs 2", "1 vs 3", "1 vs n", . . . "n-1 vs n".

The shape of `dual_coef_` is `[n_class-1, n_SV]` with a somewhat hard to grasp layout. The columns correspond to the support vectors involved in any of the `n_class * (n_class - 1) / 2` "one-vs-one" classifiers. Each of the support vectors is used in `n_class - 1` classifiers. The `n_class - 1` entries in each row correspond to the dual coefficients for these classifiers.

This might be made more clear by an example:

Consider a three class problem with class 0 having three support vectors v_0^0, v_0^1, v_0^2 and class 1 and 2 having two support vectors v_1^0, v_1^1 and v_2^0, v_2^1 respectively. For each support vector v_i^j, there are two dual coefficients. Let's call the coefficient of support vector v_i^j in the classifier between classes i and k $\alpha_{i,k}^j$. Then `dual_coef_` looks like this:

$\alpha_{0,1}^0$	$\alpha_{0,2}^0$	Coefficients for SVs of class 0
$\alpha_{0,1}^1$	$\alpha_{0,2}^1$	
$\alpha_{0,1}^2$	$\alpha_{0,2}^2$	
$\alpha_{1,0}^0$	$\alpha_{1,2}^0$	Coefficients for SVs of class 1
$\alpha_{1,0}^1$	$\alpha_{1,2}^1$	
$\alpha_{2,0}^0$	$\alpha_{2,1}^0$	Coefficients for SVs of class 2
$\alpha_{2,0}^1$	$\alpha_{2,1}^1$	

Scores and probabilities

The `decision_function` method of *SVC* and *NuSVC* gives per-class scores for each sample (or a single score per sample in the binary case). When the constructor option `probability` is set to `True`, class membership probability estimates (from the methods `predict_proba` and `predict_log_proba`) are enabled. In the binary case, the probabilities are calibrated using Platt scaling: logistic regression on the SVM's scores, fit by an additional cross-validation on the training data. In the multiclass case, this is extended as per Wu et al. (2004).

Needless to say, the cross-validation involved in Platt scaling is an expensive operation for large datasets. In addition, the probability estimates may be inconsistent with the scores, in the sense that the "argmax" of the scores may not be the argmax of the probabilities. (E.g., in binary classification, a sample may be labeled by `predict` as belonging

to a class that has probability <½ according to `predict_proba`.) Platt's method is also known to have theoretical issues. If confidence scores are required, but these do not have to be probabilities, then it is advisable to set `probability=False` and use `decision_function` instead of `predict_proba`.

Please note that when `decision_function_shape='ovr'` and `n_classes > 2`, unlike `decision_function`, the `predict` method does not try to break ties by default. You can set `break_ties=True` for the output of `predict` to be the same as `np.argmax(clf.decision_function(...), axis=1)`, otherwise the first class among the tied classes will always be returned; but have in mind that it comes with a computational cost.

References:

- Wu, Lin and Weng, "Probability estimates for multi-class classification by pairwise coupling", JMLR 5:975-1005, 2004.
- Platt "Probabilistic outputs for SVMs and comparisons to regularized likelihood methods".

Unbalanced problems

In problems where it is desired to give more importance to certain classes or certain individual samples keywords `class_weight` and `sample_weight` can be used.

SVC (but not *NuSVC*) implement a keyword `class_weight` in the `fit` method. It's a dictionary of the form `{class_label : value}`, where value is a floating point number > 0 that sets the parameter `C` of class `class_label` to `C * value`.

SVC, *NuSVC*, *SVR*, *NuSVR*, *LinearSVC*, *LinearSVR* and *OneClassSVM* implement also weights for individual samples in method `fit` through keyword `sample_weight`. Similar to `class_weight`, these set the parameter `C` for the i-th example to `C * sample_weight[i]`.

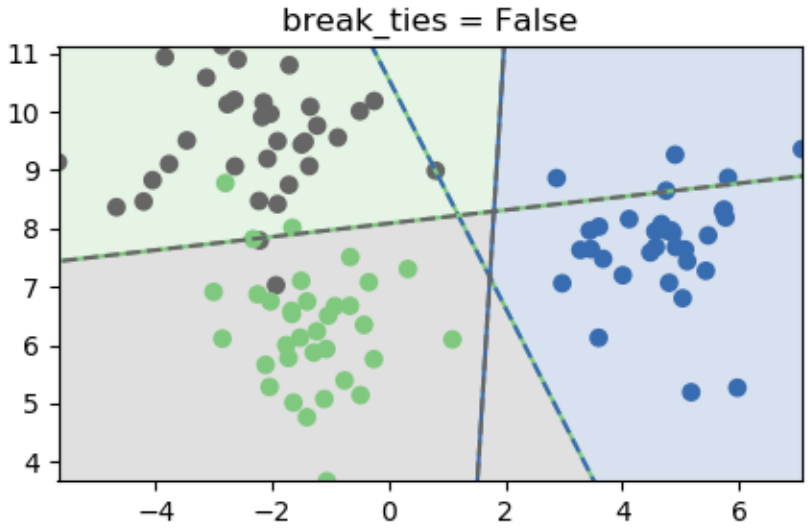
Examples:

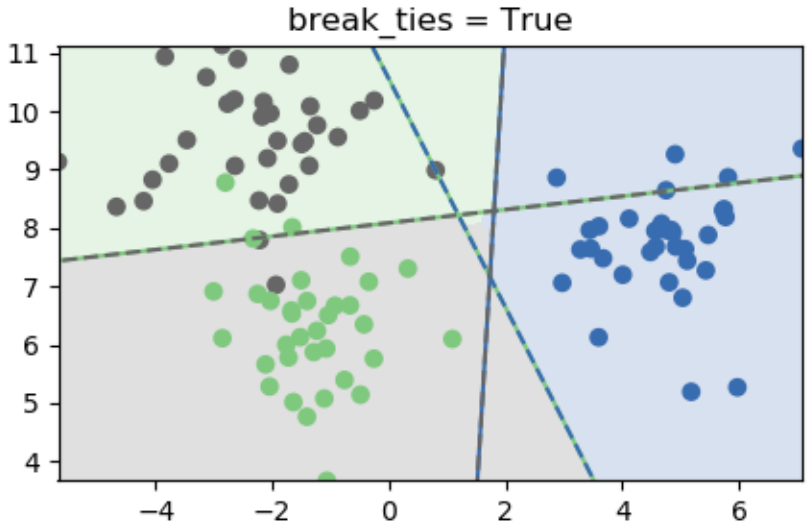
- *Plot different SVM classifiers in the iris dataset*,
- *SVM: Maximum margin separating hyperplane*,
- *SVM: Separating hyperplane for unbalanced classes*
- *SVM-Anova: SVM with univariate feature selection*,
- *Non-linear SVM*
- *SVM: Weighted samples*,

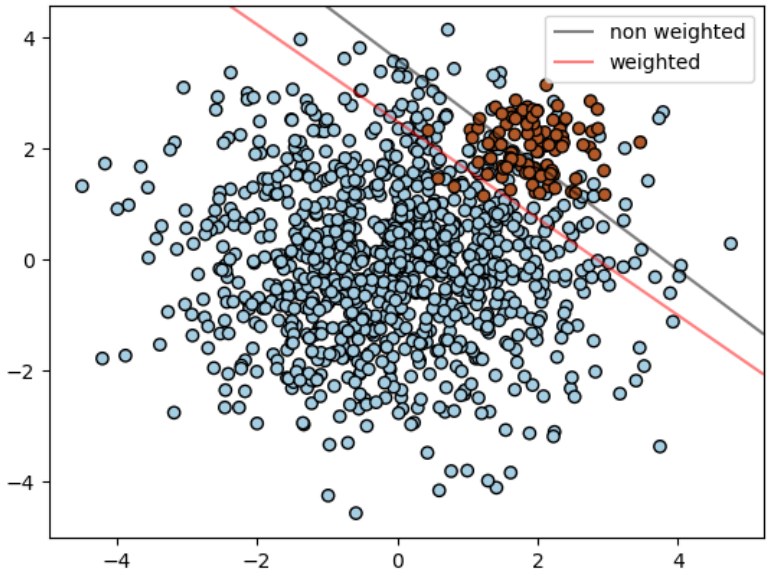
Regression

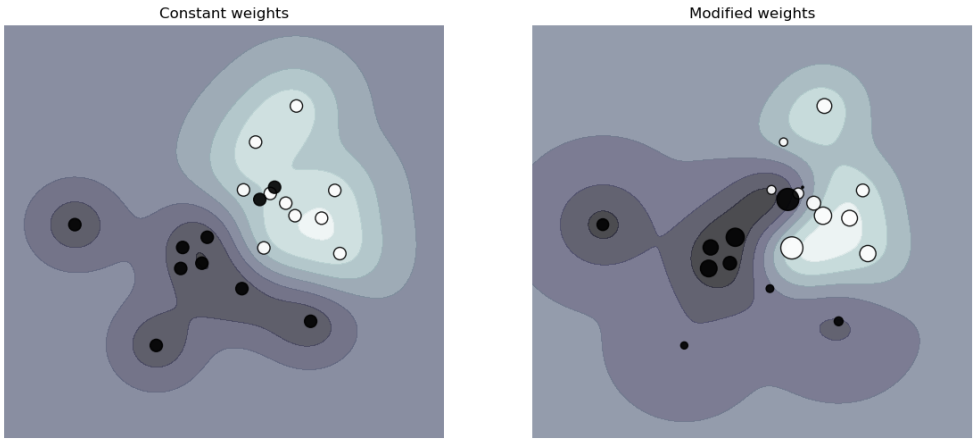
The method of Support Vector Classification can be extended to solve regression problems. This method is called Support Vector Regression.

The model produced by support vector classification (as described above) depends only on a subset of the training data, because the cost function for building the model does not care about training points that lie beyond the margin. Analogously, the model produced by Support Vector Regression depends only on a subset of the training data, because the cost function for building the model ignores any training data close to the model prediction.


break_ties = False
11
10
9
8
7
6
5
4
−4
−2
0
2
4
6


break_ties = True
11
10
9
8
7
6
5
4
−4
−2
0
2
4
6


non weighted
weighted
4
2
0
−2
−4
−4
−2
0
2
4


Constant weights
Modified weights

There are three different implementations of Support Vector Regression: *SVR*, *NuSVR* and *LinearSVR*. *LinearSVR* provides a faster implementation than *SVR* but only considers linear kernels, while *NuSVR* implements a slightly different formulation than *SVR* and *LinearSVR*. See *Implementation details* for further details.

As with classification classes, the fit method will take as argument vectors X, y, only that in this case y is expected to have floating point values instead of integer values:

```
>>> from sklearn import svm
>>> X = [[0, 0], [2, 2]]
>>> y = [0.5, 2.5]
>>> clf = svm.SVR()
>>> clf.fit(X, y)
SVR()
>>> clf.predict([[1, 1]])
array([1.5])
```

Examples:

- *Support Vector Regression (SVR) using linear and non-linear kernels*

Density estimation, novelty detection

The class *OneClassSVM* implements a One-Class SVM which is used in outlier detection.

See *Novelty and Outlier Detection* for the description and usage of OneClassSVM.

Complexity

Support Vector Machines are powerful tools, but their compute and storage requirements increase rapidly with the number of training vectors. The core of an SVM is a quadratic programming problem (QP), separating support vectors from the rest of the training data. The QP solver used by this libsvm-based implementation scales between $O(n_{features} \times n_{samples}^2)$ and $O(n_{features} \times n_{samples}^3)$ depending on how efficiently the libsvm cache is used in practice (dataset dependent). If the data is very sparse $n_{features}$ should be replaced by the average number of non-zero features in a sample vector.

Also note that for the linear case, the algorithm used in *LinearSVC* by the liblinear implementation is much more efficient than its libsvm-based *SVC* counterpart and can scale almost linearly to millions of samples and/or features.

Tips on Practical Use

- **Avoiding data copy**: For *SVC*, *SVR*, *NuSVC* and *NuSVR*, if the data passed to certain methods is not C-ordered contiguous, and double precision, it will be copied before calling the underlying C implementation. You can check whether a given numpy array is C-contiguous by inspecting its `flags` attribute.

  For *LinearSVC* (and *LogisticRegression*) any input passed as a numpy array will be copied and converted to the liblinear internal sparse data representation (double precision floats and int32 indices of non-zero components). If you want to fit a large-scale linear classifier without copying a dense numpy C-contiguous double precision array as input we suggest to use the *SGDClassifier* class instead. The objective function can be configured to be almost the same as the *LinearSVC* model.

- **Kernel cache size**: For *SVC*, *SVR*, *NuSVC* and *NuSVR*, the size of the kernel cache has a strong impact on run times for larger problems. If you have enough RAM available, it is recommended to set `cache_size` to a higher value than the default of 200(MB), such as 500(MB) or 1000(MB).

- **Setting C**: `C` is `1` by default and it's a reasonable default choice. If you have a lot of noisy observations you should decrease it. It corresponds to regularize more the estimation.

  LinearSVC and *LinearSVR* are less sensitive to `C` when it becomes large, and prediction results stop improving after a certain threshold. Meanwhile, larger `C` values will take more time to train, sometimes up to 10 times longer, as shown by Fan et al. (2008)

- Support Vector Machine algorithms are not scale invariant, so **it is highly recommended to scale your data**. For example, scale each attribute on the input vector X to [0,1] or [-1,+1], or standardize it to have mean 0 and variance 1. Note that the *same* scaling must be applied to the test vector to obtain meaningful results. See section *Preprocessing data* for more details on scaling and normalization.

- Parameter `nu` in *NuSVC*/*OneClassSVM*/*NuSVR* approximates the fraction of training errors and support vectors.

- In *SVC*, if data for classification are unbalanced (e.g. many positive and few negative), set `class_weight='balanced'` and/or try different penalty parameters `C`.

- **Randomness of the underlying implementations**: The underlying implementations of *SVC* and *NuSVC* use a random number generator only to shuffle the data for probability estimation (when `probability` is set to `True`). This randomness can be controlled with the `random_state` parameter. If `probability` is set to `False` these estimators are not random and `random_state` has no effect on the results. The underlying *OneClassSVM* implementation is similar to the ones of *SVC* and *NuSVC*. As no probability estimation is provided for *OneClassSVM*, it is not random.

  The underlying *LinearSVC* implementation uses a random number generator to select features when fitting the model with a dual coordinate descent (i.e when `dual` is set to `True`). It is thus not uncommon, to have slightly different results for the same input data. If that happens, try with a smaller tol parameter. This randomness can also be controlled with the `random_state` parameter. When `dual` is set to `False` the underlying implementation of *LinearSVC* is not random and `random_state` has no effect on the results.

- Using L1 penalization as provided by `LinearSVC(loss='l2', penalty='l1', dual=False)` yields a sparse solution, i.e. only a subset of feature weights is different from zero and contribute to the decision function. Increasing `C` yields a more complex model (more feature are selected). The `C` value that yields a "null" model (all weights equal to zero) can be calculated using *l1_min_c*.

References:

- Fan, Rong-En, et al., "LIBLINEAR: A library for large linear classification.", Journal of machine learning research 9.Aug (2008): 1871-1874.

Kernel functions

The *kernel function* can be any of the following:

- linear: $\langle x, x' \rangle$.
- polynomial: $(\gamma \langle x, x' \rangle + r)^d$. d is specified by keyword `degree`, r by `coef0`.
- rbf: $\exp(-\gamma \|x - x'\|^2)$. γ is specified by keyword `gamma`, must be greater than 0.
- sigmoid ($\tanh(\gamma \langle x, x' \rangle + r)$), where r is specified by `coef0`.

Different kernels are specified by keyword kernel at initialization:

```
>>> linear_svc = svm.SVC(kernel='linear')
>>> linear_svc.kernel
```

(continues on next page)

(continued from previous page)

```
'linear'
>>> rbf_svc = svm.SVC(kernel='rbf')
>>> rbf_svc.kernel
'rbf'
```

Custom Kernels

You can define your own kernels by either giving the kernel as a python function or by precomputing the Gram matrix.

Classifiers with custom kernels behave the same way as any other classifiers, except that:

- Field `support_vectors_` is now empty, only indices of support vectors are stored in `support_`
- A reference (and not a copy) of the first argument in the `fit()` method is stored for future reference. If that array changes between the use of `fit()` and `predict()` you will have unexpected results.

Using Python functions as kernels

You can also use your own defined kernels by passing a function to the keyword `kernel` in the constructor.

Your kernel must take as arguments two matrices of shape `(n_samples_1, n_features)`, `(n_samples_2, n_features)` and return a kernel matrix of shape `(n_samples_1, n_samples_2)`.

The following code defines a linear kernel and creates a classifier instance that will use that kernel:

```
>>> import numpy as np
>>> from sklearn import svm
>>> def my_kernel(X, Y):
...     return np.dot(X, Y.T)
...
>>> clf = svm.SVC(kernel=my_kernel)
```

Examples:

- *SVM with custom kernel*.

Using the Gram matrix

Set `kernel='precomputed'` and pass the Gram matrix instead of X in the fit method. At the moment, the kernel values between *all* training vectors and the test vectors must be provided.

```
>>> import numpy as np
>>> from sklearn import svm
>>> X = np.array([[0, 0], [1, 1]])
>>> y = [0, 1]
>>> clf = svm.SVC(kernel='precomputed')
>>> # linear kernel computation
>>> gram = np.dot(X, X.T)
>>> clf.fit(gram, y)
SVC(kernel='precomputed')
>>> # predict on training examples
```

(continues on next page)

(continued from previous page)

```
>>> clf.predict(gram)
array([0, 1])
```

Parameters of the RBF Kernel

When training an SVM with the *Radial Basis Function* (RBF) kernel, two parameters must be considered: `C` and `gamma`. The parameter `C`, common to all SVM kernels, trades off misclassification of training examples against simplicity of the decision surface. A low `C` makes the decision surface smooth, while a high `C` aims at classifying all training examples correctly. `gamma` defines how much influence a single training example has. The larger `gamma` is, the closer other examples must be to be affected.

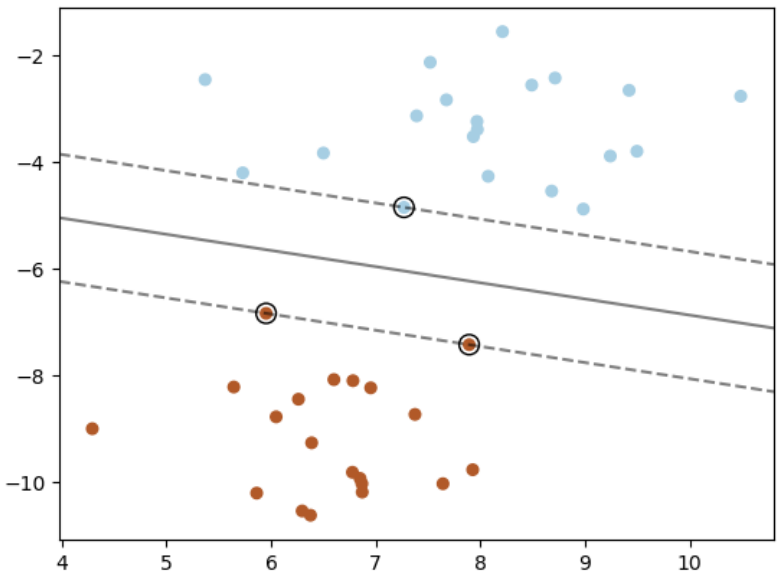
Proper choice of `C` and `gamma` is critical to the SVM's performance. One is advised to use *sklearn.model_selection.GridSearchCV* with `C` and `gamma` spaced exponentially far apart to choose good values.

Examples:

- *RBF SVM parameters*

Mathematical formulation

A support vector machine constructs a hyper-plane or set of hyper-planes in a high or infinite dimensional space, which can be used for classification, regression or other tasks. Intuitively, a good separation is achieved by the hyper-plane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier.



SVC

Given training vectors $x_i \in \mathbb{R}^p$, i=1,..., n, in two classes, and a vector $y \in \{1, -1\}^n$, SVC solves the following primal problem:

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i$$
$$\text{subject to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i,$$
$$\zeta_i \geq 0, i = 1, ..., n$$

Its dual is

$$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha$$
$$\text{subject to } y^T \alpha = 0$$
$$0 \leq \alpha_i \leq C, i = 1, ..., n$$

where e is the vector of all ones, $C > 0$ is the upper bound, Q is an n by n positive semidefinite matrix, $Q_{ij} \equiv y_i y_j K(x_i, x_j)$, where $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the kernel. Here training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function ϕ.

The decision function is:

$$\text{sgn}(\sum_{i=1}^{n} y_i \alpha_i K(x_i, x) + \rho)$$

Note: While SVM models derived from libsvm and liblinear use `C` as regularization parameter, most other estimators use `alpha`. The exact equivalence between the amount of regularization of two models depends on the exact objective function optimized by the model. For example, when the estimator used is `sklearn.linear_model.Ridge` regression, the relation between them is given as $C = \frac{1}{alpha}$.

This parameters can be accessed through the members `dual_coef_` which holds the product $y_i \alpha_i$, `support_vectors_` which holds the support vectors, and `intercept_` which holds the independent term ρ :

References:

- "Automatic Capacity Tuning of Very Large VC-dimension Classifiers", I. Guyon, B. Boser, V. Vapnik - Advances in neural information processing 1993.
- "Support-vector networks", C. Cortes, V. Vapnik - Machine Learning, 20, 273-297 (1995).

NuSVC

We introduce a new parameter ν which controls the number of support vectors and training errors. The parameter $\nu \in (0, 1]$ is an upper bound on the fraction of training errors and a lower bound of the fraction of support vectors.

It can be shown that the ν-SVC formulation is a reparameterization of the C-SVC and therefore mathematically equivalent.

SVR

Given training vectors $x_i \in \mathbb{R}^p$, i=1,..., n, and a vector $y \in \mathbb{R}^n$ ε-SVR solves the following primal problem:

$$\min_{w,b,\zeta,\zeta^*} \frac{1}{2} w^T w + C \sum_{i=1}^{n} (\zeta_i + \zeta_i^*)$$

$$\text{subject to } y_i - w^T \phi(x_i) - b \leq \varepsilon + \zeta_i,$$
$$w^T \phi(x_i) + b - y_i \leq \varepsilon + \zeta_i^*,$$
$$\zeta_i, \zeta_i^* \geq 0, i = 1, ..., n$$

Its dual is

$$\min_{\alpha,\alpha^*} \frac{1}{2} (\alpha - \alpha^*)^T Q (\alpha - \alpha^*) + \varepsilon e^T (\alpha + \alpha^*) - y^T (\alpha - \alpha^*)$$

$$\text{subject to } e^T (\alpha - \alpha^*) = 0$$
$$0 \leq \alpha_i, \alpha_i^* \leq C, i = 1, ..., n$$

where e is the vector of all ones, $C > 0$ is the upper bound, Q is an n by n positive semidefinite matrix, $Q_{ij} \equiv K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the kernel. Here training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function ϕ.

The decision function is:

$$\sum_{i=1}^{n} (\alpha_i - \alpha_i^*) K(x_i, x) + \rho$$

These parameters can be accessed through the members `dual_coef_` which holds the difference $\alpha_i - \alpha_i^*$, `support_vectors_` which holds the support vectors, and `intercept_` which holds the independent term ρ

References:

- "A Tutorial on Support Vector Regression", Alex J. Smola, Bernhard Schölkopf - Statistics and Computing archive Volume 14 Issue 3, August 2004, p. 199-222.

Implementation details

Internally, we use libsvm and liblinear to handle all computations. These libraries are wrapped using C and Cython.

References:

For a description of the implementation and details of the algorithms used, please refer to

- LIBSVM: A Library for Support Vector Machines.
- LIBLINEAR – A Library for Large Linear Classification.

4.1.5 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a simple yet very efficient approach to discriminative learning of linear classifiers under convex loss functions such as (linear) Support Vector Machines and Logistic Regression. Even though

SGD has been around in the machine learning community for a long time, it has received a considerable amount of attention just recently in the context of large-scale learning.

SGD has been successfully applied to large-scale and sparse machine learning problems often encountered in text classification and natural language processing. Given that the data is sparse, the classifiers in this module easily scale to problems with more than 10^5 training examples and more than 10^5 features.

The advantages of Stochastic Gradient Descent are:

- Efficiency.
- Ease of implementation (lots of opportunities for code tuning).

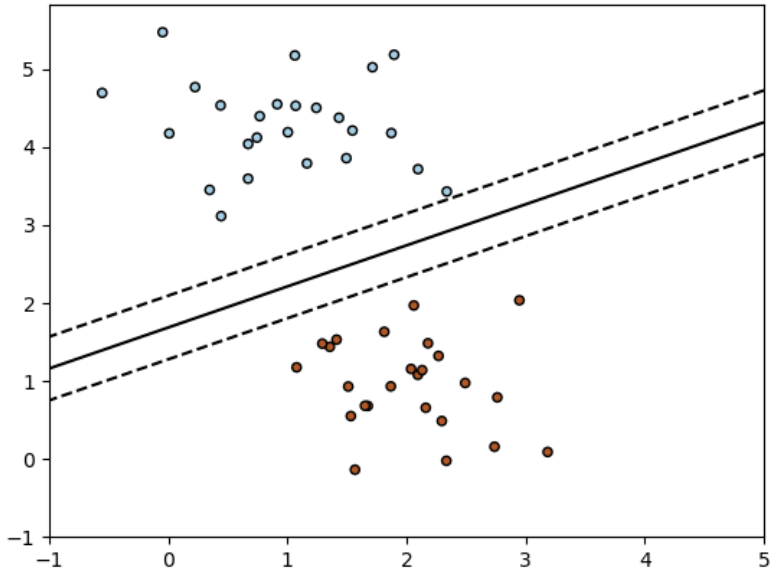
The disadvantages of Stochastic Gradient Descent include:

- SGD requires a number of hyperparameters such as the regularization parameter and the number of iterations.
- SGD is sensitive to feature scaling.

Classification

Warning: Make sure you permute (shuffle) your training data before fitting the model or use `shuffle=True` to shuffle after each iteration.

The class *`SGDClassifier`* implements a plain stochastic gradient descent learning routine which supports different loss functions and penalties for classification.



As other classifiers, SGD has to be fitted with two arrays: an array X of size [n_samples, n_features] holding the training samples, and an array Y of size [n_samples] holding the target values (class labels) for the training samples:

```
>>> from sklearn.linear_model import SGDClassifier
>>> X = [[0., 0.], [1., 1.]]
```

(continues on next page)

(continued from previous page)

```
>>> y = [0, 1]
>>> clf = SGDClassifier(loss="hinge", penalty="l2", max_iter=5)
>>> clf.fit(X, y)
SGDClassifier(max_iter=5)
```

After being fitted, the model can then be used to predict new values:

```
>>> clf.predict([[2., 2.]])
array([1])
```

SGD fits a linear model to the training data. The member `coef_` holds the model parameters:

```
>>> clf.coef_
array([[9.9..., 9.9...]])
```

Member `intercept_` holds the intercept (aka offset or bias):

```
>>> clf.intercept_
array([-9.9...])
```

Whether or not the model should use an intercept, i.e. a biased hyperplane, is controlled by the parameter `fit_intercept`.

To get the signed distance to the hyperplane use *`SGDClassifier.decision_function`*:

```
>>> clf.decision_function([[2., 2.]])
array([29.6...])
```

The concrete loss function can be set via the `loss` parameter. *`SGDClassifier`* supports the following loss functions:

- `loss="hinge"`: (soft-margin) linear Support Vector Machine,
- `loss="modified_huber"`: smoothed hinge loss,
- `loss="log"`: logistic regression,
- and all regression losses below.

The first two loss functions are lazy, they only update the model parameters if an example violates the margin constraint, which makes training very efficient and may result in sparser models, even when L2 penalty is used.

Using `loss="log"` or `loss="modified_huber"` enables the `predict_proba` method, which gives a vector of probability estimates $P(y|x)$ per sample x:

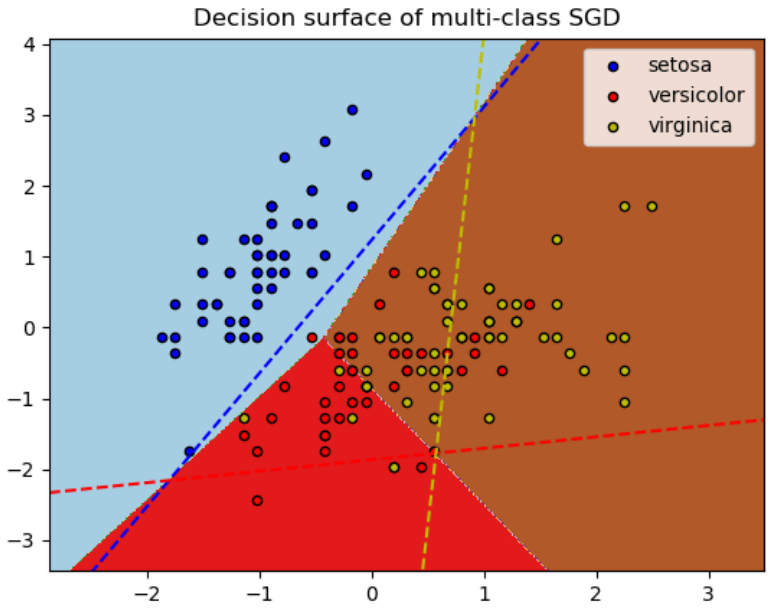
```
>>> clf = SGDClassifier(loss="log", max_iter=5).fit(X, y)
>>> clf.predict_proba([[1., 1.]])
array([[0.00..., 0.99...]])
```

The concrete penalty can be set via the `penalty` parameter. SGD supports the following penalties:

- `penalty="l2"`: L2 norm penalty on `coef_`.
- `penalty="l1"`: L1 norm penalty on `coef_`.
- `penalty="elasticnet"`: Convex combination of L2 and L1; `(1 - l1_ratio) * L2 + l1_ratio * L1`.

The default setting is `penalty="l2"`. The L1 penalty leads to sparse solutions, driving most coefficients to zero. The Elastic Net solves some deficiencies of the L1 penalty in the presence of highly correlated attributes. The parameter `l1_ratio` controls the convex combination of L1 and L2 penalty.

SGDClassifier supports multi-class classification by combining multiple binary classifiers in a "one versus all" (OVA) scheme. For each of the K classes, a binary classifier is learned that discriminates between that and all other $K-1$ classes. At testing time, we compute the confidence score (i.e. the signed distances to the hyperplane) for each classifier and choose the class with the highest confidence. The Figure below illustrates the OVA approach on the iris dataset. The dashed lines represent the three OVA classifiers; the background colors show the decision surface induced by the three classifiers.



In the case of multi-class classification `coef_` is a two-dimensional array of `shape=[n_classes, n_features]` and `intercept_` is a one-dimensional array of `shape=[n_classes]`. The i-th row of `coef_` holds the weight vector of the OVA classifier for the i-th class; classes are indexed in ascending order (see attribute `classes_`). Note that, in principle, since they allow to create a probability model, `loss="log"` and `loss="modified_huber"` are more suitable for one-vs-all classification.

SGDClassifier supports both weighted classes and weighted instances via the fit parameters `class_weight` and `sample_weight`. See the examples below and the docstring of *SGDClassifier.fit* for further information.

Examples:

- *SGD: Maximum margin separating hyperplane*,
- *Plot multi-class SGD on the iris dataset*
- *SGD: Weighted samples*
- *Comparing various online solvers*
- *SVM: Separating hyperplane for unbalanced classes* (See the `Note`)

SGDClassifier supports averaged SGD (ASGD). Averaging can be enabled by setting `` `average=True` ``. ASGD works by averaging the coefficients of the plain SGD over each iteration over a sample. When using ASGD the learning rate can be larger and even constant leading on some datasets to a speed up in training time.

For classification with a logistic loss, another variant of SGD with an averaging strategy is available with Stochastic Average Gradient (SAG) algorithm, available as a solver in *LogisticRegression*.

Regression

The class *SGDRegressor* implements a plain stochastic gradient descent learning routine which supports different loss functions and penalties to fit linear regression models. *SGDRegressor* is well suited for regression problems with a large number of training samples (> 10.000), for other problems we recommend *Ridge*, *Lasso*, or *ElasticNet*.

The concrete loss function can be set via the `loss` parameter. *SGDRegressor* supports the following loss functions:

- `loss="squared_loss"`: Ordinary least squares,
- `loss="huber"`: Huber loss for robust regression,
- `loss="epsilon_insensitive"`: linear Support Vector Regression.

The Huber and epsilon-insensitive loss functions can be used for robust regression. The width of the insensitive region has to be specified via the parameter `epsilon`. This parameter depends on the scale of the target variables.

SGDRegressor supports averaged SGD as *SGDClassifier*. Averaging can be enabled by setting `` `average=True` ``.

For regression with a squared loss and a l2 penalty, another variant of SGD with an averaging strategy is available with Stochastic Average Gradient (SAG) algorithm, available as a solver in *Ridge*.

Stochastic Gradient Descent for sparse data

Note: The sparse implementation produces slightly different results than the dense implementation due to a shrunk learning rate for the intercept.

There is built-in support for sparse data given in any matrix in a format supported by scipy.sparse. For maximum efficiency, however, use the CSR matrix format as defined in scipy.sparse.csr_matrix.

Examples:

- *Classification of text documents using sparse features*

Complexity

The major advantage of SGD is its efficiency, which is basically linear in the number of training examples. If X is a matrix of size (n, p) training has a cost of $O(kn\bar{p})$, where k is the number of iterations (epochs) and $\bar{p}$ is the average number of non-zero attributes per sample.

Recent theoretical results, however, show that the runtime to get some desired optimization accuracy does not increase as the training set size increases.

Stopping criterion

The classes *SGDClassifier* and *SGDRegressor* provide two criteria to stop the algorithm when a given level of convergence is reached:

- With `early_stopping=True`, the input data is split into a training set and a validation set. The model is then fitted on the training set, and the stopping criterion is based on the prediction score computed on the validation set. The size of the validation set can be changed with the parameter `validation_fraction`.
- With `early_stopping=False`, the model is fitted on the entire input data and the stopping criterion is based on the objective function computed on the input data.

In both cases, the criterion is evaluated once by epoch, and the algorithm stops when the criterion does not improve `n_iter_no_change` times in a row. The improvement is evaluated with a tolerance `tol`, and the algorithm stops in any case after a maximum number of iteration `max_iter`.

Tips on Practical Use

- Stochastic Gradient Descent is sensitive to feature scaling, so it is highly recommended to scale your data. For example, scale each attribute on the input vector X to [0,1] or [-1,+1], or standardize it to have mean 0 and variance 1. Note that the *same* scaling must be applied to the test vector to obtain meaningful results. This can be easily done using `StandardScaler`:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)  # Don't cheat - fit only on training data
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)  # apply same transformation to test data
```

  If your attributes have an intrinsic scale (e.g. word frequencies or indicator features) scaling is not needed.

- Finding a reasonable regularization term α is best done using `GridSearchCV`, usually in the range `10.0**-np.arange(1,7)`.
- Empirically, we found that SGD converges after observing approx. 10^6 training samples. Thus, a reasonable first guess for the number of iterations is `max_iter = np.ceil(10**6 / n)`, where `n` is the size of the training set.
- If you apply SGD to features extracted using PCA we found that it is often wise to scale the feature values by some constant `c` such that the average L2 norm of the training data equals one.
- We found that Averaged SGD works best with a larger number of features and a higher eta0

References:

- "Efficient BackProp" Y. LeCun, L. Bottou, G. Orr, K. Müller - In Neural Networks: Tricks of the Trade 1998.

Mathematical formulation

Given a set of training examples $(x_1, y_1), \ldots, (x_n, y_n)$ where $x_i \in \mathbf{R}^m$ and $y_i \in \{-1, 1\}$, our goal is to learn a linear scoring function $f(x) = w^T x + b$ with model parameters $w \in \mathbf{R}^m$ and intercept $b \in \mathbf{R}$. In order to make predictions, we simply look at the sign of $f(x)$. A common choice to find the model parameters is by minimizing the regularized training error given by

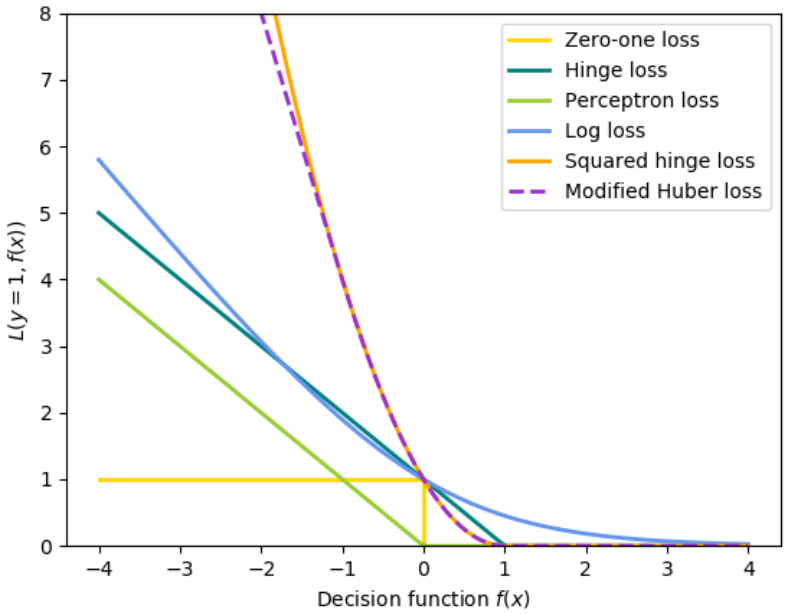
$$E(w, b) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(x_i)) + \alpha R(w)$$

where L is a loss function that measures model (mis)fit and R is a regularization term (aka penalty) that penalizes model complexity; $\alpha > 0$ is a non-negative hyperparameter.

Different choices for L entail different classifiers such as

- Hinge: (soft-margin) Support Vector Machines.
- Log: Logistic Regression.
- Least-Squares: Ridge Regression.
- Epsilon-Insensitive: (soft-margin) Support Vector Regression.

All of the above loss functions can be regarded as an upper bound on the misclassification error (Zero-one loss) as shown in the Figure below.



Popular choices for the regularization term R include:

- L2 norm: $R(w) := \frac{1}{2}\sum_{i=1}^{n} w_i^2$,
- L1 norm: $R(w) := \sum_{i=1}^{n} |w_i|$, which leads to sparse solutions.
- Elastic Net: $R(w) := \frac{\rho}{2}\sum_{i=1}^{n} w_i^2 + (1-\rho)\sum_{i=1}^{n} |w_i|$, a convex combination of L2 and L1, where ρ is given by `1 - l1_ratio`.

The Figure below shows the contours of the different regularization terms in the parameter space when $R(w) = 1$.

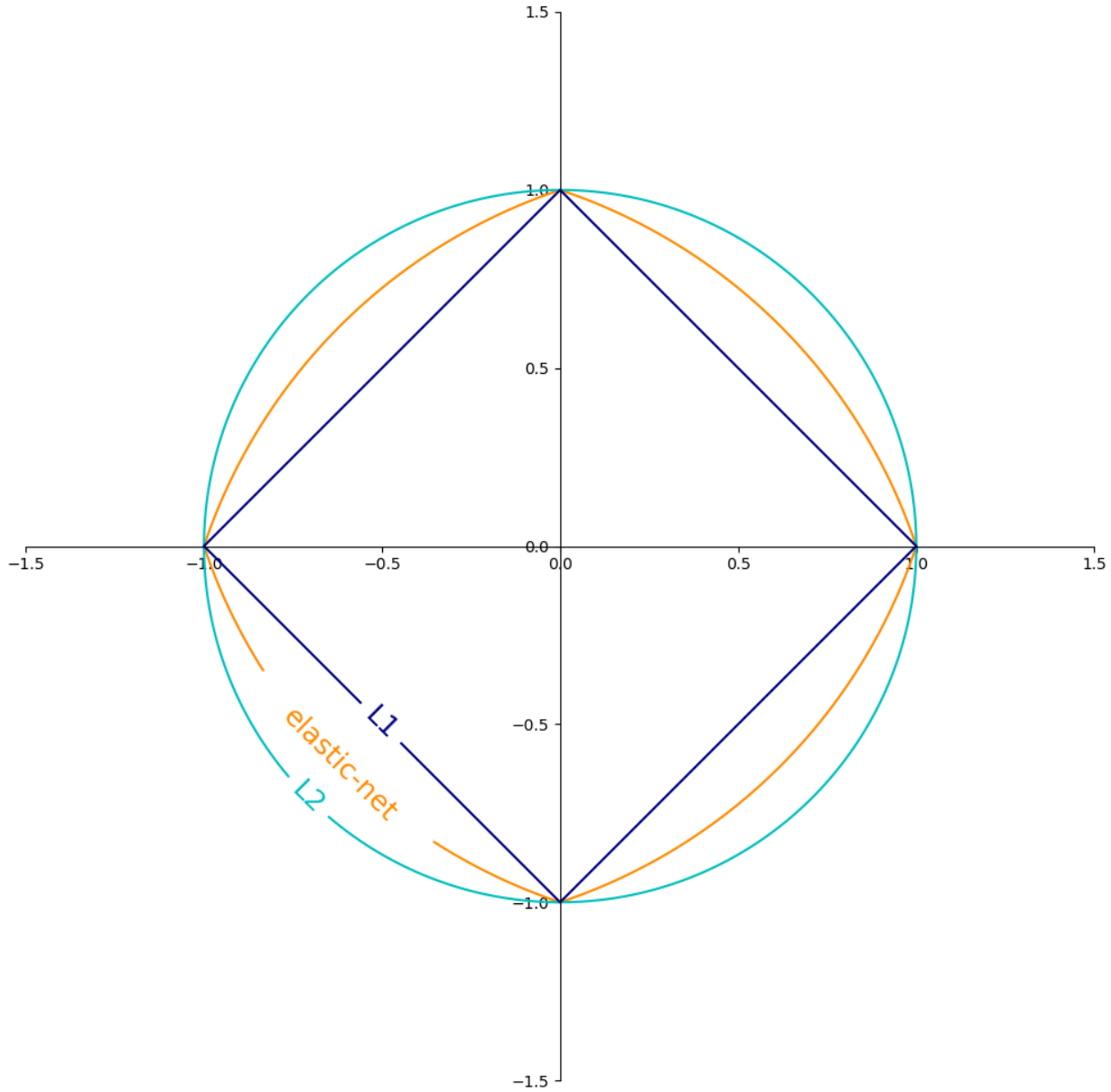
SGD

Stochastic gradient descent is an optimization method for unconstrained optimization problems. In contrast to (batch) gradient descent, SGD approximates the true gradient of $E(w, b)$ by considering a single training example at a time.

The class *`SGDClassifier`* implements a first-order SGD learning routine. The algorithm iterates over the training examples and for each example updates the model parameters according to the update rule given by

$$w \leftarrow w - \eta(\alpha\frac{\partial R(w)}{\partial w} + \frac{\partial L(w^T x_i + b, y_i)}{\partial w})$$

where η is the learning rate which controls the step-size in the parameter space. The intercept b is updated similarly but without regularization.


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
L1
elastic-net
L2

The learning rate η can be either constant or gradually decaying. For classification, the default learning rate schedule (`learning_rate='optimal'`) is given by

$$\eta^{(t)} = \frac{1}{\alpha(t_0 + t)}$$

where t is the time step (there are a total of `n_samples * n_iter` time steps), t_0 is determined based on a heuristic proposed by Léon Bottou such that the expected initial updates are comparable with the expected size of the weights (this assuming that the norm of the training samples is approx. 1). The exact definition can be found in `_init_t` in `BaseSGD`.

For regression the default learning rate schedule is inverse scaling (`learning_rate='invscaling'`), given by

$$\eta^{(t)} = \frac{eta_0}{t^{power\_t}}$$

where eta_0 and $power\_t$ are hyperparameters chosen by the user via `eta0` and `power_t`, resp.

For a constant learning rate use `learning_rate='constant'` and use `eta0` to specify the learning rate.

For an adaptively decreasing learning rate, use `learning_rate='adaptive'` and use `eta0` to specify the starting learning rate. When the stopping criterion is reached, the learning rate is divided by 5, and the algorithm does not stop. The algorithm stops when the learning rate goes below 1e-6.

The model parameters can be accessed through the members `coef_` and `intercept_`:

- Member `coef_` holds the weights w
- Member `intercept_` holds b

References:

- "Solving large scale linear prediction problems using stochastic gradient descent algorithms" T. Zhang - In Proceedings of ICML '04.
- "Regularization and variable selection via the elastic net" H. Zou, T. Hastie - Journal of the Royal Statistical Society Series B, 67 (2), 301-320.
- "Towards Optimal One Pass Large Scale Learning with Averaged Stochastic Gradient Descent" Xu, Wei

Implementation details

The implementation of SGD is influenced by the Stochastic Gradient SVM of Léon Bottou. Similar to SvmSGD, the weight vector is represented as the product of a scalar and a vector which allows an efficient weight update in the case of L2 regularization. In the case of sparse feature vectors, the intercept is updated with a smaller learning rate (multiplied by 0.01) to account for the fact that it is updated more frequently. Training examples are picked up sequentially and the learning rate is lowered after each observed example. We adopted the learning rate schedule from Shalev-Shwartz et al. 2007. For multi-class classification, a "one versus all" approach is used. We use the truncated gradient algorithm proposed by Tsuruoka et al. 2009 for L1 regularization (and the Elastic Net). The code is written in Cython.

References:

- "Stochastic Gradient Descent" L. Bottou - Website, 2010.
- "The Tradeoffs of Large Scale Machine Learning" L. Bottou - Website, 2011.

- "Pegasos: Primal estimated sub-gradient solver for svm" S. Shalev-Shwartz, Y. Singer, N. Srebro - In Proceedings of ICML '07.
- "Stochastic gradient descent training for l1-regularized log-linear models with cumulative penalty" Y. Tsuruoka, J. Tsujii, S. Ananiadou - In Proceedings of the AFNLP/ACL '09.

4.1.6 Nearest Neighbors

`sklearn.neighbors` provides functionality for unsupervised and supervised neighbors-based learning methods. Unsupervised nearest neighbors is the foundation of many other learning methods, notably manifold learning and spectral clustering. Supervised neighbors-based learning comes in two flavors: *classification* for data with discrete labels, and *regression* for data with continuous labels.

The principle behind nearest neighbor methods is to find a predefined number of training samples closest in distance to the new point, and predict the label from these. The number of samples can be a user-defined constant (k-nearest neighbor learning), or vary based on the local density of points (radius-based neighbor learning). The distance can, in general, be any metric measure: standard Euclidean distance is the most common choice. Neighbors-based methods are known as *non-generalizing* machine learning methods, since they simply "remember" all of its training data (possibly transformed into a fast indexing structure such as a *Ball Tree* or *KD Tree*).

Despite its simplicity, nearest neighbors has been successful in a large number of classification and regression problems, including handwritten digits and satellite image scenes. Being a non-parametric method, it is often successful in classification situations where the decision boundary is very irregular.

The classes in `sklearn.neighbors` can handle either NumPy arrays or `scipy.sparse` matrices as input. For dense matrices, a large number of possible distance metrics are supported. For sparse matrices, arbitrary Minkowski metrics are supported for searches.

There are many learning routines which rely on nearest neighbors at their core. One example is *kernel density estimation*, discussed in the *density estimation* section.

Unsupervised Nearest Neighbors

`NearestNeighbors` implements unsupervised nearest neighbors learning. It acts as a uniform interface to three different nearest neighbors algorithms: `BallTree`, `KDTree`, and a brute-force algorithm based on routines in `sklearn.metrics.pairwise`. The choice of neighbors search algorithm is controlled through the keyword `'algorithm'`, which must be one of `['auto', 'ball_tree', 'kd_tree', 'brute']`. When the default value `'auto'` is passed, the algorithm attempts to determine the best approach from the training data. For a discussion of the strengths and weaknesses of each option, see *Nearest Neighbor Algorithms*.

> **Warning:** Regarding the Nearest Neighbors algorithms, if two neighbors $k+1$ and k have identical distances but different labels, the result will depend on the ordering of the training data.

Finding the Nearest Neighbors

For the simple task of finding the nearest neighbors between two sets of data, the unsupervised algorithms within `sklearn.neighbors` can be used:

```
>>> from sklearn.neighbors import NearestNeighbors
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
```

(continues on next page)

(continued from previous page)

```
>>> nbrs = NearestNeighbors(n_neighbors=2, algorithm='ball_tree').fit(X)
>>> distances, indices = nbrs.kneighbors(X)
>>> indices
array([[0, 1],
       [1, 0],
       [2, 1],
       [3, 4],
       [4, 3],
       [5, 4]]...)
>>> distances
array([[0.        , 1.        ],
       [0.        , 1.        ],
       [0.        , 1.41421356],
       [0.        , 1.        ],
       [0.        , 1.        ],
       [0.        , 1.41421356]])
```

Because the query set matches the training set, the nearest neighbor of each point is the point itself, at a distance of zero.

It is also possible to efficiently produce a sparse graph showing the connections between neighboring points:

```
>>> nbrs.kneighbors_graph(X).toarray()
array([[1., 1., 0., 0., 0., 0.],
       [1., 1., 0., 0., 0., 0.],
       [0., 1., 1., 0., 0., 0.],
       [0., 0., 0., 1., 1., 0.],
       [0., 0., 0., 1., 1., 0.],
       [0., 0., 0., 0., 1., 1.]])
```

The dataset is structured such that points nearby in index order are nearby in parameter space, leading to an approximately block-diagonal matrix of K-nearest neighbors. Such a sparse graph is useful in a variety of circumstances which make use of spatial relationships between points for unsupervised learning: in particular, see *sklearn.manifold.Isomap*, *sklearn.manifold.LocallyLinearEmbedding*, and *sklearn.cluster.SpectralClustering*.

KDTree and BallTree Classes

Alternatively, one can use the *KDTree* or *BallTree* classes directly to find nearest neighbors. This is the functionality wrapped by the *NearestNeighbors* class used above. The Ball Tree and KD Tree have the same interface; we'll show an example of using the KD Tree here:

```
>>> from sklearn.neighbors import KDTree
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> kdt = KDTree(X, leaf_size=30, metric='euclidean')
>>> kdt.query(X, k=2, return_distance=False)
array([[0, 1],
       [1, 0],
       [2, 1],
       [3, 4],
       [4, 3],
       [5, 4]]...)
```

Refer to the *KDTree* and *BallTree* class documentation for more information on the options available for nearest

neighbors searches, including specification of query strategies, distance metrics, etc. For a list of available metrics, see the documentation of the `DistanceMetric` class.

Nearest Neighbors Classification

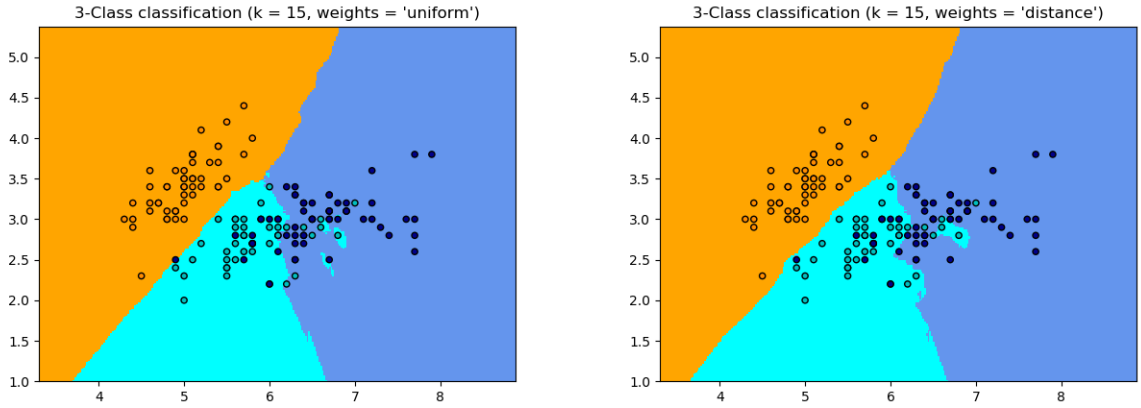
Neighbors-based classification is a type of *instance-based learning* or *non-generalizing learning*: it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point.

scikit-learn implements two different nearest neighbors classifiers: `KNeighborsClassifier` implements learning based on the k nearest neighbors of each query point, where k is an integer value specified by the user. `RadiusNeighborsClassifier` implements learning based on the number of neighbors within a fixed radius r of each training point, where r is a floating-point value specified by the user.

The k-neighbors classification in `KNeighborsClassifier` is the most commonly used technique. The optimal choice of the value k is highly data-dependent: in general a larger k suppresses the effects of noise, but makes the classification boundaries less distinct.

In cases where the data is not uniformly sampled, radius-based neighbors classification in `RadiusNeighborsClassifier` can be a better choice. The user specifies a fixed radius r, such that points in sparser neighborhoods use fewer nearest neighbors for the classification. For high-dimensional parameter spaces, this method becomes less effective due to the so-called "curse of dimensionality".

The basic nearest neighbors classification uses uniform weights: that is, the value assigned to a query point is computed from a simple majority vote of the nearest neighbors. Under some circumstances, it is better to weight the neighbors such that nearer neighbors contribute more to the fit. This can be accomplished through the `weights` keyword. The default value, `weights = 'uniform'`, assigns uniform weights to each neighbor. `weights = 'distance'` assigns weights proportional to the inverse of the distance from the query point. Alternatively, a user-defined function of the distance can be supplied to compute the weights.



Examples:

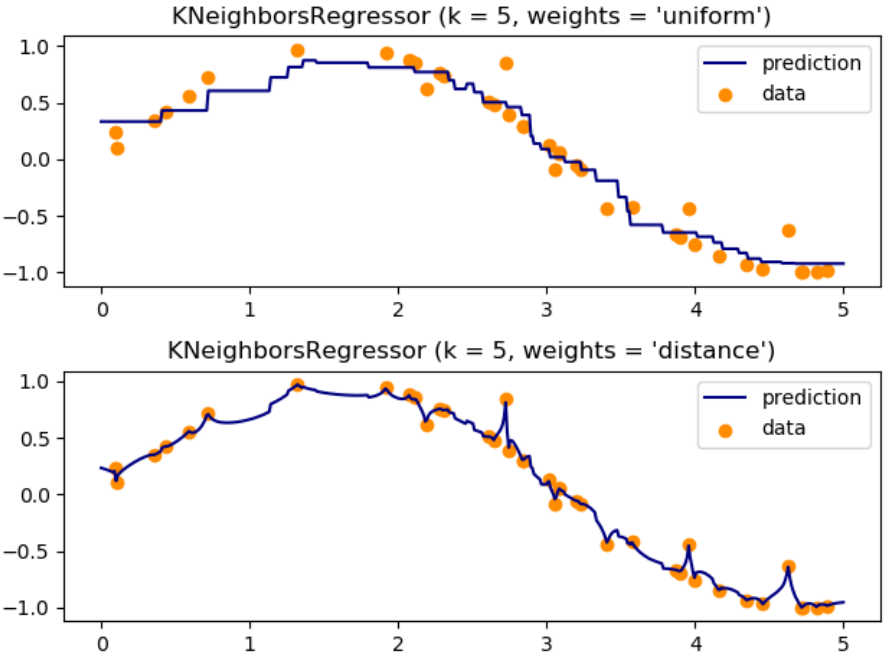
- *Nearest Neighbors Classification*: an example of classification using nearest neighbors.

Nearest Neighbors Regression

Neighbors-based regression can be used in cases where the data labels are continuous rather than discrete variables. The label assigned to a query point is computed based on the mean of the labels of its nearest neighbors.

scikit-learn implements two different neighbors regressors: `KNeighborsRegressor` implements learning based on the k nearest neighbors of each query point, where k is an integer value specified by the user. `RadiusNeighborsRegressor` implements learning based on the neighbors within a fixed radius r of the query point, where r is a floating-point value specified by the user.

The basic nearest neighbors regression uses uniform weights: that is, each point in the local neighborhood contributes uniformly to the classification of a query point. Under some circumstances, it can be advantageous to weight points such that nearby points contribute more to the regression than faraway points. This can be accomplished through the `weights` keyword. The default value, `weights = 'uniform'`, assigns equal weights to all points. `weights = 'distance'` assigns weights proportional to the inverse of the distance from the query point. Alternatively, a user-defined function of the distance can be supplied, which will be used to compute the weights.



The use of multi-output nearest neighbors for regression is demonstrated in *Face completion with a multi-output estimators*. In this example, the inputs X are the pixels of the upper half of faces and the outputs Y are the pixels of the lower half of those faces.

Examples:

- *Nearest Neighbors regression*: an example of regression using nearest neighbors.
- *Face completion with a multi-output estimators*: an example of multi-output regression using nearest neighbors.

Nearest Neighbor Algorithms


Face completion with multi-output estimators
true faces
Extra trees
K-nn
Linear regression
Ridge

Brute Force

Fast computation of nearest neighbors is an active area of research in machine learning. The most naive neighbor search implementation involves the brute-force computation of distances between all pairs of points in the dataset: for N samples in D dimensions, this approach scales as $O[DN^2]$. Efficient brute-force neighbors searches can be very competitive for small data samples. However, as the number of samples N grows, the brute-force approach quickly becomes infeasible. In the classes within *sklearn.neighbors*, brute-force neighbors searches are specified using the keyword `algorithm = 'brute'`, and are computed using the routines available in *sklearn.metrics.pairwise*.

K-D Tree

To address the computational inefficiencies of the brute-force approach, a variety of tree-based data structures have been invented. In general, these structures attempt to reduce the required number of distance calculations by efficiently encoding aggregate distance information for the sample. The basic idea is that if point A is very distant from point B, and point B is very close to point C, then we know that points A and C are very distant, *without having to explicitly calculate their distance*. In this way, the computational cost of a nearest neighbors search can be reduced to $O[DN \log(N)]$ or better. This is a significant improvement over brute-force for large N.

An early approach to taking advantage of this aggregate information was the *KD tree* data structure (short for *K-dimensional tree*), which generalizes two-dimensional *Quad-trees* and 3-dimensional *Oct-trees* to an arbitrary number of dimensions. The KD tree is a binary tree structure which recursively partitions the parameter space along the data axes, dividing it into nested orthotropic regions into which data points are filed. The construction of a KD tree is very fast: because partitioning is performed only along the data axes, no D-dimensional distances need to be computed. Once constructed, the nearest neighbor of a query point can be determined with only $O[\log(N)]$ distance computations. Though the KD tree approach is very fast for low-dimensional ($D < 20$) neighbors searches, it becomes inefficient as D grows very large: this is one manifestation of the so-called "curse of dimensionality". In scikit-learn, KD tree neighbors searches are specified using the keyword `algorithm = 'kd_tree'`, and are computed using the class *KDTree*.

References:

- "Multidimensional binary search trees used for associative searching", Bentley, J.L., Communications of the ACM (1975)

Ball Tree

To address the inefficiencies of KD Trees in higher dimensions, the *ball tree* data structure was developed. Where KD trees partition data along Cartesian axes, ball trees partition data in a series of nesting hyper-spheres. This makes tree construction more costly than that of the KD tree, but results in a data structure which can be very efficient on highly structured data, even in very high dimensions.

A ball tree recursively divides the data into nodes defined by a centroid C and radius r, such that each point in the node lies within the hyper-sphere defined by r and C. The number of candidate points for a neighbor search is reduced through use of the *triangle inequality*:

$$|x + y| \leq |x| + |y|$$

With this setup, a single distance calculation between a test point and the centroid is sufficient to determine a lower and upper bound on the distance to all points within the node. Because of the spherical geometry of the ball tree nodes, it can out-perform a *KD-tree* in high dimensions, though the actual performance is highly dependent on the structure

of the training data. In scikit-learn, ball-tree-based neighbors searches are specified using the keyword `algorithm = 'ball_tree'`, and are computed using the class *`sklearn.neighbors.BallTree`*. Alternatively, the user can work with the *`BallTree`* class directly.

References:

- "Five balltree construction algorithms", Omohundro, S.M., International Computer Science Institute Technical Report (1989)

Choice of Nearest Neighbors Algorithm

The optimal algorithm for a given dataset is a complicated choice, and depends on a number of factors:

- number of samples N (i.e. `n_samples`) and dimensionality D (i.e. `n_features`).
  - *Brute force* query time grows as $O[DN]$
  - *Ball tree* query time grows as approximately $O[D\log(N)]$
  - *KD tree* query time changes with D in a way that is difficult to precisely characterise. For small D (less than 20 or so) the cost is approximately $O[D\log(N)]$, and the KD tree query can be very efficient. For larger D, the cost increases to nearly $O[DN]$, and the overhead due to the tree structure can lead to queries which are slower than brute force.

  For small data sets (N less than 30 or so), $\log(N)$ is comparable to N, and brute force algorithms can be more efficient than a tree-based approach. Both *`KDTree`* and *`BallTree`* address this through providing a *leaf size* parameter: this controls the number of samples at which a query switches to brute-force. This allows both algorithms to approach the efficiency of a brute-force computation for small N.
- data structure: *intrinsic dimensionality* of the data and/or *sparsity* of the data. Intrinsic dimensionality refers to the dimension $d \le D$ of a manifold on which the data lies, which can be linearly or non-linearly embedded in the parameter space. Sparsity refers to the degree to which the data fills the parameter space (this is to be distinguished from the concept as used in "sparse" matrices. The data matrix may have no zero entries, but the **structure** can still be "sparse" in this sense).
  - *Brute force* query time is unchanged by data structure.
  - *Ball tree* and *KD tree* query times can be greatly influenced by data structure. In general, sparser data with a smaller intrinsic dimensionality leads to faster query times. Because the KD tree internal representation is aligned with the parameter axes, it will not generally show as much improvement as ball tree for arbitrarily structured data.

  Datasets used in machine learning tend to be very structured, and are very well-suited for tree-based queries.
- number of neighbors k requested for a query point.
  - *Brute force* query time is largely unaffected by the value of k
  - *Ball tree* and *KD tree* query time will become slower as k increases. This is due to two effects: first, a larger k leads to the necessity to search a larger portion of the parameter space. Second, using $k > 1$ requires internal queueing of results as the tree is traversed.

  As k becomes large compared to N, the ability to prune branches in a tree-based query is reduced. In this situation, Brute force queries can be more efficient.
- number of query points. Both the ball tree and the KD Tree require a construction phase. The cost of this construction becomes negligible when amortized over many queries. If only a small number of queries will

be performed, however, the construction can make up a significant fraction of the total cost. If very few query points will be required, brute force is better than a tree-based method.

Currently, `algorithm = 'auto'` selects `'brute'` if $k >= N/2$, the input data is sparse, or `effective_metric_` isn't in the `VALID_METRICS` list for either `'kd_tree'` or `'ball_tree'`. Otherwise, it selects the first out of `'kd_tree'` and `'ball_tree'` that has `effective_metric_` in its `VALID_METRICS` list. This choice is based on the assumption that the number of query points is at least the same order as the number of training points, and that `leaf_size` is close to its default value of `30`.

Effect of `leaf_size`

As noted above, for small sample sizes a brute force search can be more efficient than a tree-based query. This fact is accounted for in the ball tree and KD tree by internally switching to brute force searches within leaf nodes. The level of this switch can be specified with the parameter `leaf_size`. This parameter choice has many effects:

construction time A larger `leaf_size` leads to a faster tree construction time, because fewer nodes need to be created

query time Both a large or small `leaf_size` can lead to suboptimal query cost. For `leaf_size` approaching 1, the overhead involved in traversing nodes can significantly slow query times. For `leaf_size` approaching the size of the training set, queries become essentially brute force. A good compromise between these is `leaf_size = 30`, the default value of the parameter.

memory As `leaf_size` increases, the memory required to store a tree structure decreases. This is especially important in the case of ball tree, which stores a D-dimensional centroid for each node. The required storage space for *BallTree* is approximately `1 / leaf_size` times the size of the training set.

`leaf_size` is not referenced for brute force queries.

Nearest Centroid Classifier

The *NearestCentroid* classifier is a simple algorithm that represents each class by the centroid of its members. In effect, this makes it similar to the label updating phase of the *sklearn.cluster.KMeans* algorithm. It also has no parameters to choose, making it a good baseline classifier. It does, however, suffer on non-convex classes, as well as when classes have drastically different variances, as equal variance in all dimensions is assumed. See Linear Discriminant Analysis (*sklearn.discriminant_analysis.LinearDiscriminantAnalysis*) and Quadratic Discriminant Analysis (*sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis*) for more complex methods that do not make this assumption. Usage of the default *NearestCentroid* is simple:

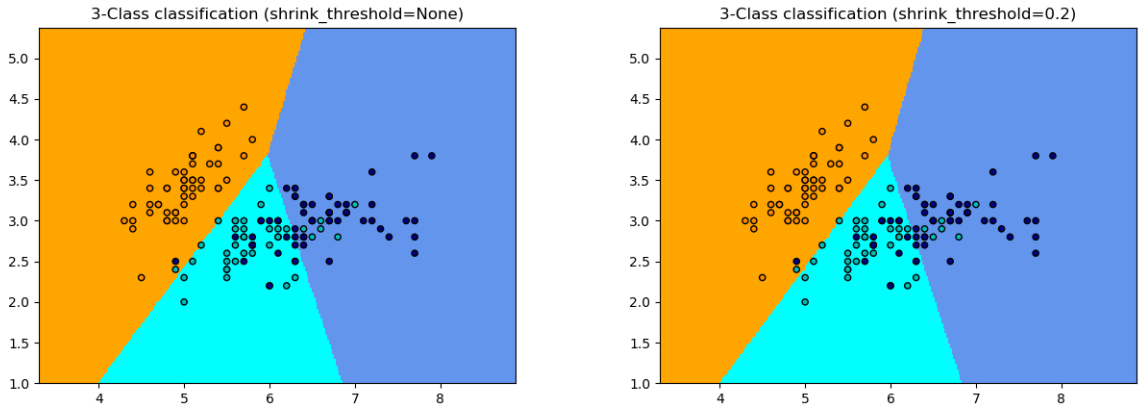
```
>>> from sklearn.neighbors import NearestCentroid
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> y = np.array([1, 1, 1, 2, 2, 2])
>>> clf = NearestCentroid()
>>> clf.fit(X, y)
NearestCentroid()
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Nearest Shrunken Centroid

The *NearestCentroid* classifier has a `shrink_threshold` parameter, which implements the nearest shrunken centroid classifier. In effect, the value of each feature for each centroid is divided by the within-class variance of that feature. The feature values are then reduced by `shrink_threshold`. Most notably, if a particular feature value

crosses zero, it is set to zero. In effect, this removes the feature from affecting the classification. This is useful, for example, for removing noisy features.

In the example below, using a small shrink threshold increases the accuracy of the model from 0.81 to 0.82.



Examples:

- *Nearest Centroid Classification*: an example of classification using nearest centroid with different shrink thresholds.

Nearest Neighbors Transformer

Many scikit-learn estimators rely on nearest neighbors: Several classifiers and regressors such as `KNeighborsClassifier` and `KNeighborsRegressor`, but also some clustering methods such as `DBSCAN` and `SpectralClustering`, and some manifold embeddings such as `TSNE` and `Isomap`.

All these estimators can compute internally the nearest neighbors, but most of them also accept precomputed nearest neighbors *sparse graph*, as given by `kneighbors_graph` and `radius_neighbors_graph`. With mode `mode='connectivity'`, these functions return a binary adjacency sparse graph as required, for instance, in `SpectralClustering`. Whereas with `mode='distance'`, they return a distance sparse graph as required, for instance, in `DBSCAN`. To include these functions in a scikit-learn pipeline, one can also use the corresponding classes `KNeighborsTransformer` and `RadiusNeighborsTransformer`. The benefits of this sparse graph API are multiple.

First, the precomputed graph can be re-used multiple times, for instance while varying a parameter of the estimator. This can be done manually by the user, or using the caching properties of the scikit-learn pipeline:

```
>>> from sklearn.manifold import Isomap
>>> from sklearn.neighbors import KNeighborsTransformer
>>> from sklearn.pipeline import make_pipeline
>>> estimator = make_pipeline(
...     KNeighborsTransformer(n_neighbors=5, mode='distance'),
...     Isomap(neighbors_algorithm='precomputed'),
...     memory='/path/to/cache')
```

Second, precomputing the graph can give finer control on the nearest neighbors estimation, for instance enabling multiprocessing though the parameter `n_jobs`, which might not be available in all estimators.

Finally, the precomputation can be performed by custom estimators to use different implementations, such as approximate nearest neighbors methods, or implementation with special data types. The precomputed neighbors *sparse graph*

needs to be formatted as in *radius_neighbors_graph* output:

- a CSR matrix (although COO, CSC or LIL will be accepted).
- only explicitly store nearest neighborhoods of each sample with respect to the training data. This should include those at 0 distance from a query point, including the matrix diagonal when computing the nearest neighborhoods between the training data and itself.
- each row's `data` should store the distance in increasing order (optional. Unsorted data will be stable-sorted, adding a computational overhead).
- all values in data should be non-negative.
- there should be no duplicate `indices` in any row (see https://github.com/scipy/scipy/issues/5807).
- if the algorithm being passed the precomputed matrix uses k nearest neighbors (as opposed to radius neighborhood), at least k neighbors must be stored in each row (or k+1, as explained in the following note).

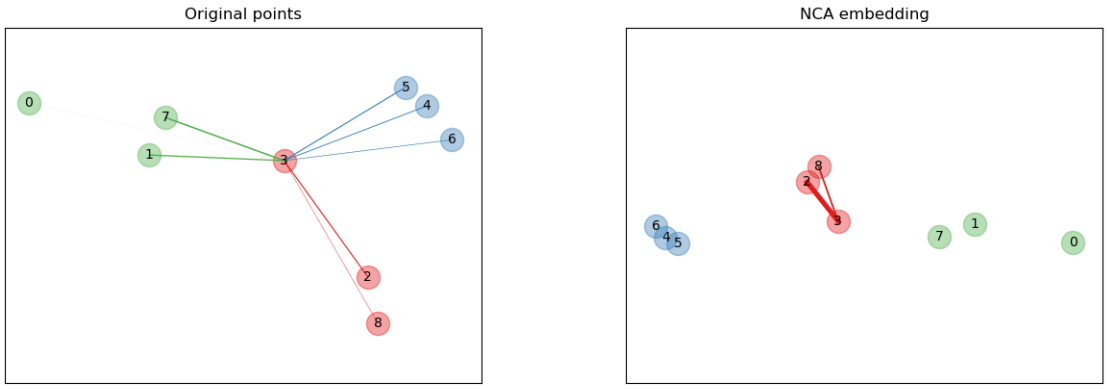
Note: When a specific number of neighbors is queried (using *KNeighborsTransformer*), the definition of `n_neighbors` is ambiguous since it can either include each training point as its own neighbor, or exclude them. Neither choice is perfect, since including them leads to a different number of non-self neighbors during training and testing, while excluding them leads to a difference between `fit(X).transform(X)` and `fit_transform(X)`, which is against scikit-learn API. In *KNeighborsTransformer* we use the definition which includes each training point as its own neighbor in the count of `n_neighbors`. However, for compatibility reasons with other estimators which use the other definition, one extra neighbor will be computed when `mode == 'distance'`. To maximise compatibility with all estimators, a safe choice is to always include one extra neighbor in a custom nearest neighbors estimator, since unnecessary neighbors will be filtered by following estimators.

Examples:

- *Approximate nearest neighbors in TSNE*: an example of pipelining *KNeighborsTransformer* and *TSNE*. Also proposes two custom nearest neighbors estimators based on external packages.
- *Caching nearest neighbors*: an example of pipelining *KNeighborsTransformer* and *KNeighborsClassifier* to enable caching of the neighbors graph during a hyper-parameter grid-search.

Neighborhood Components Analysis

Neighborhood Components Analysis (NCA, *NeighborhoodComponentsAnalysis*) is a distance metric learning algorithm which aims to improve the accuracy of nearest neighbors classification compared to the standard Euclidean distance. The algorithm directly maximizes a stochastic variant of the leave-one-out k-nearest neighbors (KNN) score on the training set. It can also learn a low-dimensional linear projection of data that can be used for data visualization and fast classification.



In the above illustrating figure, we consider some points from a randomly generated dataset. We focus on the stochastic KNN classification of point no. 3. The thickness of a link between sample 3 and another point is proportional to their distance, and can be seen as the relative weight (or probability) that a stochastic nearest neighbor prediction rule would assign to this point. In the original space, sample 3 has many stochastic neighbors from various classes, so the right class is not very likely. However, in the projected space learned by NCA, the only stochastic neighbors with non-negligible weight are from the same class as sample 3, guaranteeing that the latter will be well classified. See the *mathematical formulation* for more details.

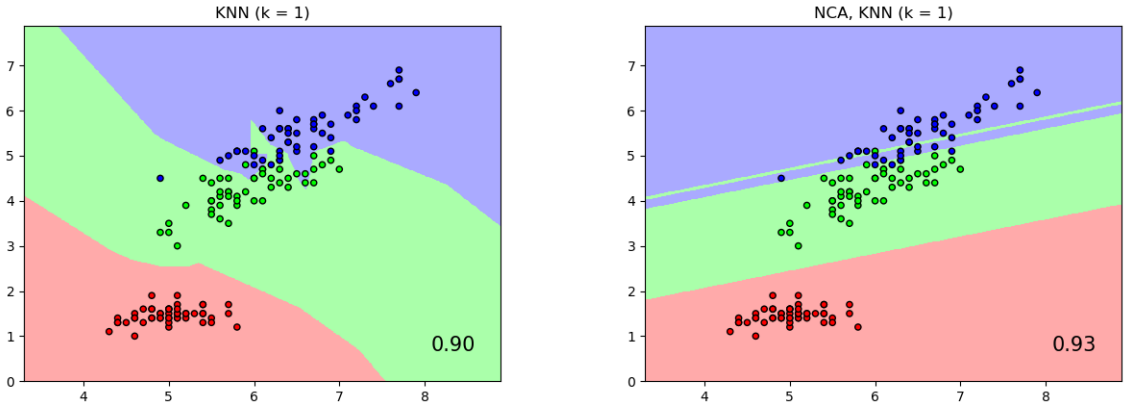
Classification

Combined with a nearest neighbors classifier (`KNeighborsClassifier`), NCA is attractive for classification because it can naturally handle multi-class problems without any increase in the model size, and does not introduce additional parameters that require fine-tuning by the user.

NCA classification has been shown to work well in practice for data sets of varying size and difficulty. In contrast to related methods such as Linear Discriminant Analysis, NCA does not make any assumptions about the class distributions. The nearest neighbor classification can naturally produce highly irregular decision boundaries.

To use this model for classification, one needs to combine a `NeighborhoodComponentsAnalysis` instance that learns the optimal transformation with a `KNeighborsClassifier` instance that performs the classification in the projected space. Here is an example using the two classes:

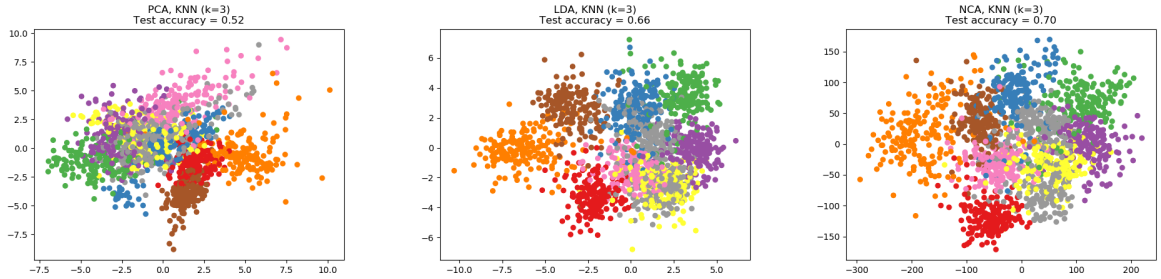
```
>>> from sklearn.neighbors import (NeighborhoodComponentsAnalysis,
... KNeighborsClassifier)
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.pipeline import Pipeline
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
... stratify=y, test_size=0.7, random_state=42)
>>> nca = NeighborhoodComponentsAnalysis(random_state=42)
>>> knn = KNeighborsClassifier(n_neighbors=3)
>>> nca_pipe = Pipeline([('nca', nca), ('knn', knn)])
>>> nca_pipe.fit(X_train, y_train)
Pipeline(...)
>>> print(nca_pipe.score(X_test, y_test))
0.96190476...
```



The plot shows decision boundaries for Nearest Neighbor Classification and Neighborhood Components Analysis classification on the iris dataset, when training and scoring on only two features, for visualisation purposes.

Dimensionality reduction

NCA can be used to perform supervised dimensionality reduction. The input data are projected onto a linear subspace consisting of the directions which minimize the NCA objective. The desired dimensionality can be set using the parameter `n_components`. For instance, the following figure shows a comparison of dimensionality reduction with Principal Component Analysis (*`sklearn.decomposition.PCA`*), Linear Discriminant Analysis (*`sklearn.discriminant_analysis.LinearDiscriminantAnalysis`*) and Neighborhood Component Analysis (*`NeighborhoodComponentsAnalysis`*) on the Digits dataset, a dataset with size $n_{samples} = 1797$ and $n_{features} = 64$. The data set is split into a training and a test set of equal size, then standardized. For evaluation the 3-nearest neighbor classification accuracy is computed on the 2-dimensional projected points found by each method. Each data sample belongs to one of 10 classes.



Examples:

- *Comparing Nearest Neighbors with and without Neighborhood Components Analysis*
- *Dimensionality Reduction with Neighborhood Components Analysis*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

Mathematical formulation

The goal of NCA is to learn an optimal linear transformation matrix of size `(n_components, n_features)`, which maximises the sum over all samples i of the probability p_i that i is correctly classified, i.e.:

$$\arg\max_{L} \sum_{i=0}^{N-1} p_i$$

with N = `n_samples` and p_i the probability of sample i being correctly classified according to a stochastic nearest neighbors rule in the learned embedded space:

$$p_i = \sum_{j \in C_i} p_{ij}$$

where C_i is the set of points in the same class as sample i, and p_{ij} is the softmax over Euclidean distances in the embedded space:

$$p_{ij} = \frac{\exp(-||Lx_i - Lx_j||^2)}{\sum_{k \neq i} \exp -(||Lx_i - Lx_k||^2)}, \quad p_{ii} = 0$$

Mahalanobis distance

NCA can be seen as learning a (squared) Mahalanobis distance metric:

$$||L(x_i - x_j)||^2 = (x_i - x_j)^T M (x_i - x_j),$$

where $M = L^T L$ is a symmetric positive semi-definite matrix of size `(n_features, n_features)`.

Implementation

This implementation follows what is explained in the original paper[1]. For the optimisation method, it currently uses scipy's L-BFGS-B with a full gradient computation at each iteration, to avoid to tune the learning rate and provide stable learning.

See the examples below and the docstring of *`NeighborhoodComponentsAnalysis.fit`* for further information.

Complexity

Training

NCA stores a matrix of pairwise distances, taking `n_samples ** 2` memory. Time complexity depends on the number of iterations done by the optimisation algorithm. However, one can set the maximum number of iterations with the argument `max_iter`. For each iteration, time complexity is `O(n_components x n_samples x min(n_samples, n_features))`.

[1] "Neighbourhood Components Analysis", J. Goldberger, S. Roweis, G. Hinton, R. Salakhutdinov, Advances in Neural Information Processing Systems, Vol. 17, May 2005, pp. 513-520.

Transform

Here the `transform` operation returns LX^T, therefore its time complexity equals `n_components * n_features * n_samples_test`. There is no added space complexity in the operation.

References:

Wikipedia entry on Neighborhood Components Analysis

4.1.7 Gaussian Processes

Gaussian Processes (GP) are a generic supervised learning method designed to solve *regression* and *probabilistic classification* problems.

The advantages of Gaussian processes are:

- The prediction interpolates the observations (at least for regular kernels).
- The prediction is probabilistic (Gaussian) so that one can compute empirical confidence intervals and decide based on those if one should refit (online fitting, adaptive fitting) the prediction in some region of interest.
- Versatile: different *kernels* can be specified. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of Gaussian processes include:

- They are not sparse, i.e., they use the whole samples/features information to perform the prediction.
- They lose efficiency in high dimensional spaces – namely when the number of features exceeds a few dozens.

Gaussian Process Regression (GPR)

The *GaussianProcessRegressor* implements Gaussian processes (GP) for regression purposes. For this, the prior of the GP needs to be specified. The prior mean is assumed to be constant and zero (for `normalize_y=False`) or the training data's mean (for `normalize_y=True`). The prior's covariance is specified by passing a *kernel* object. The hyperparameters of the kernel are optimized during fitting of GaussianProcessRegressor by maximizing the log-marginal-likelihood (LML) based on the passed `optimizer`. As the LML may have multiple local optima, the optimizer can be started repeatedly by specifying `n_restarts_optimizer`. The first run is always conducted starting from the initial hyperparameter values of the kernel; subsequent runs are conducted from hyperparameter values that have been chosen randomly from the range of allowed values. If the initial hyperparameters should be kept fixed, `None` can be passed as optimizer.

The noise level in the targets can be specified by passing it via the parameter `alpha`, either globally as a scalar or per datapoint. Note that a moderate noise level can also be helpful for dealing with numeric issues during fitting as it is effectively implemented as Tikhonov regularization, i.e., by adding it to the diagonal of the kernel matrix. An alternative to specifying the noise level explicitly is to include a WhiteKernel component into the kernel, which can estimate the global noise level from the data (see example below).

The implementation is based on Algorithm 2.1 of [RW2006]. In addition to the API of standard scikit-learn estimators, GaussianProcessRegressor:

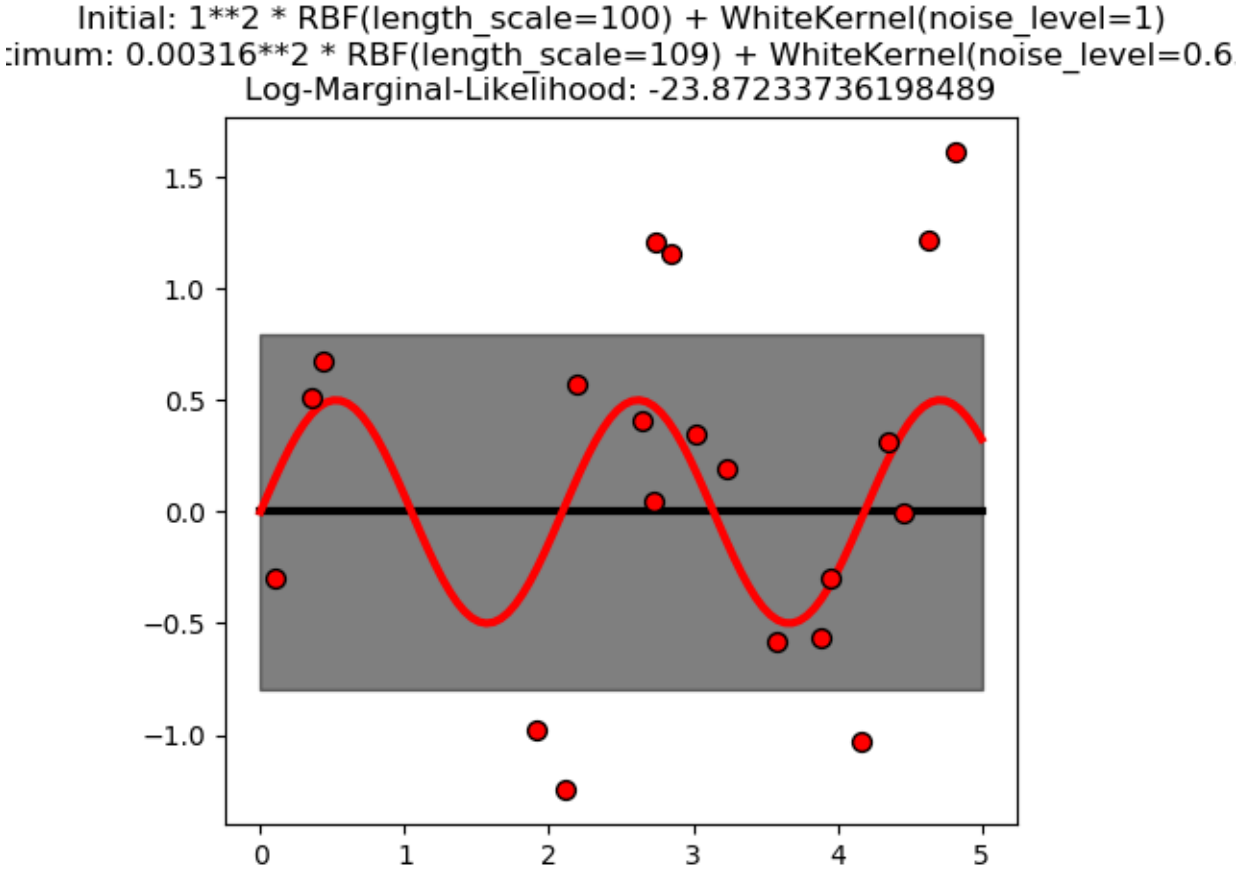
- allows prediction without prior fitting (based on the GP prior)
- provides an additional method `sample_y(X)`, which evaluates samples drawn from the GPR (prior or posterior) at given inputs

- exposes a method `log_marginal_likelihood(theta)`, which can be used externally for other ways of selecting hyperparameters, e.g., via Markov chain Monte Carlo.

GPR examples

GPR with noise-level estimation

This example illustrates that GPR with a sum-kernel including a WhiteKernel can estimate the noise level of data. An illustration of the log-marginal-likelihood (LML) landscape shows that there exist two local maxima of LML.

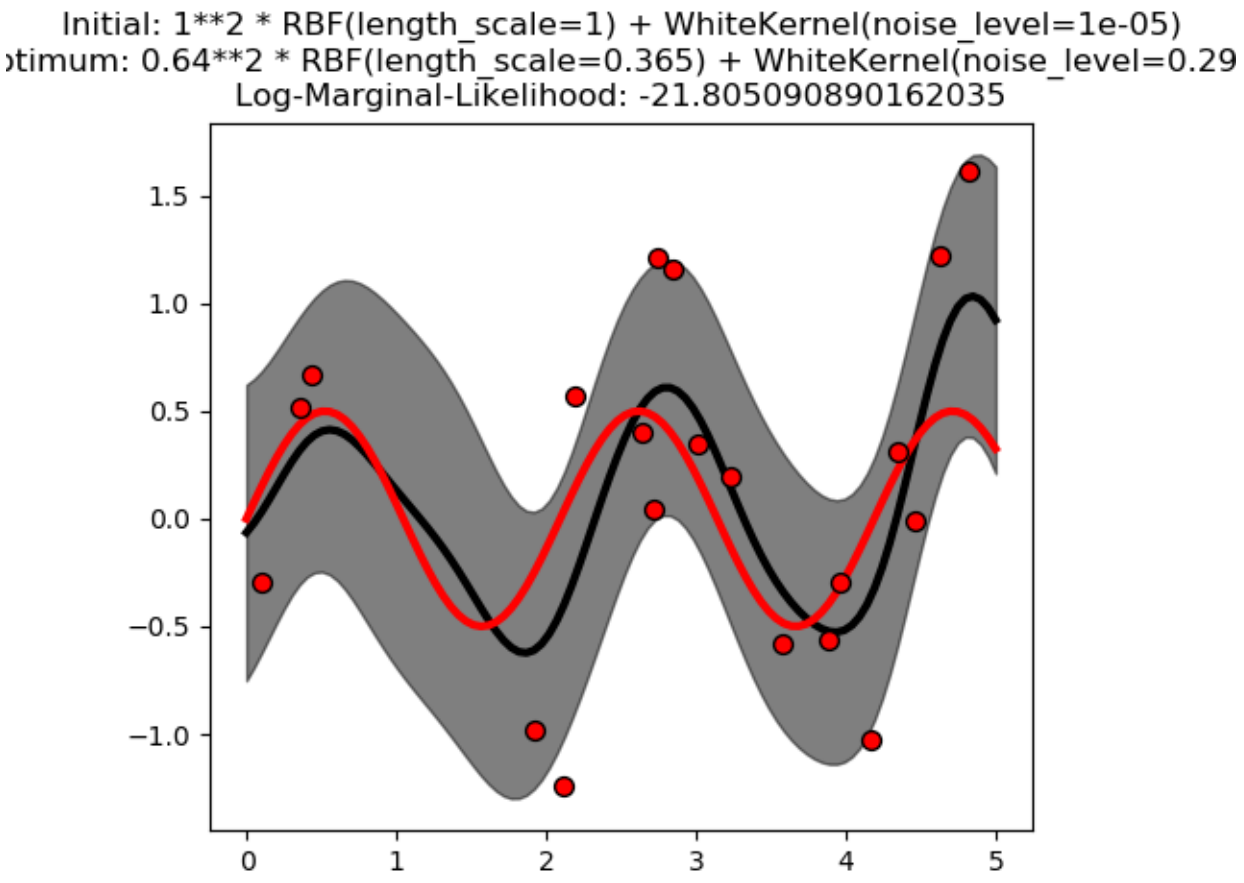


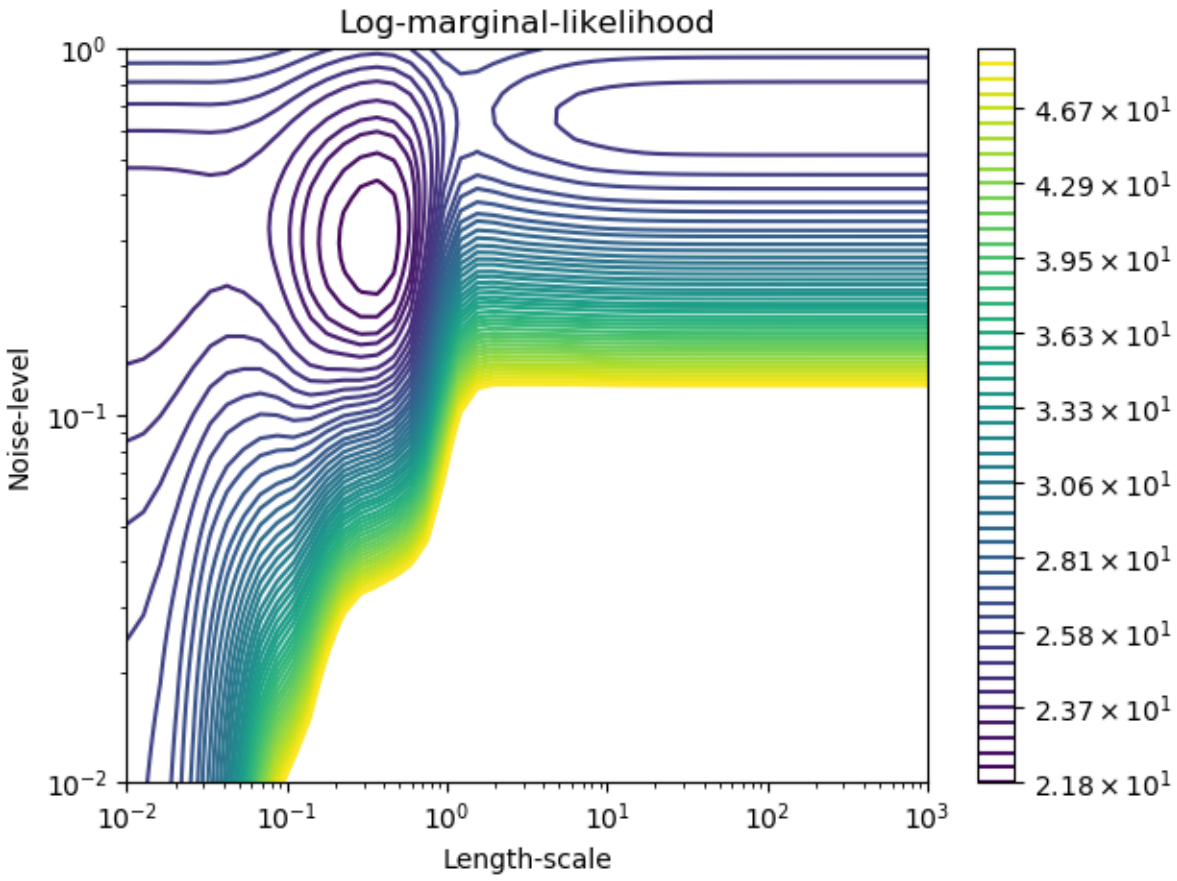
The first corresponds to a model with a high noise level and a large length scale, which explains all variations in the data by noise.

The second one has a smaller noise level and shorter length scale, which explains most of the variation by the noise-free functional relationship. The second model has a higher likelihood; however, depending on the initial value for the hyperparameters, the gradient-based optimization might also converge to the high-noise solution. It is thus important to repeat the optimization several times for different initializations.

Comparison of GPR and Kernel Ridge Regression

Both kernel ridge regression (KRR) and GPR learn a target function by employing internally the "kernel trick". KRR learns a linear function in the space induced by the respective kernel which corresponds to a non-linear function in the original space. The linear function in the kernel space is chosen based on the mean-squared error loss with ridge

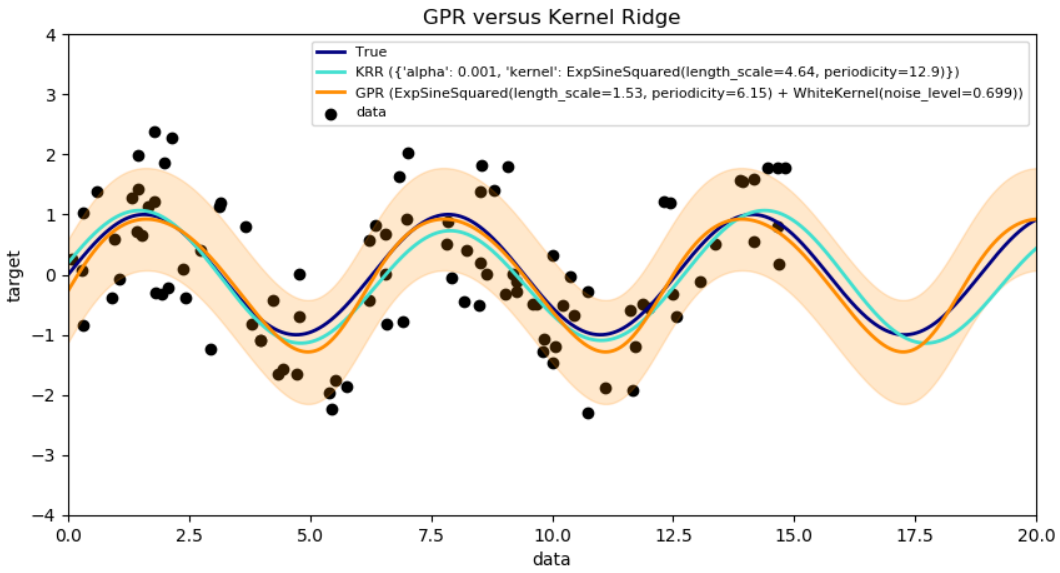

Initial: 1**2 * RBF(length_scale=1) + WhiteKernel(noise_level=1e-05)
ɔtimum: 0.64**2 * RBF(length_scale=0.365) + WhiteKernel(noise_level=0.29
Log-Marginal-Likelihood: -21.805090890162035
1.5
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5


Log-marginal-likelihood
Noise-level
Length-scale
10^{0}
10^{-1}
10^{-2}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
4.67×10^{1}
4.29×10^{1}
3.95×10^{1}
3.63×10^{1}
3.33×10^{1}
3.06×10^{1}
2.81×10^{1}
2.58×10^{1}
2.37×10^{1}
2.18×10^{1}

regularization. GPR uses the kernel to define the covariance of a prior distribution over the target functions and uses the observed training data to define a likelihood function. Based on Bayes theorem, a (Gaussian) posterior distribution over target functions is defined, whose mean is used for prediction.

A major difference is that GPR can choose the kernel's hyperparameters based on gradient-ascent on the marginal likelihood function while KRR needs to perform a grid search on a cross-validated loss function (mean-squared error loss). A further difference is that GPR learns a generative, probabilistic model of the target function and can thus provide meaningful confidence intervals and posterior samples along with the predictions while KRR only provides predictions.

The following figure illustrates both methods on an artificial dataset, which consists of a sinusoidal target function and strong noise. The figure compares the learned model of KRR and GPR based on a ExpSineSquared kernel, which is suited for learning periodic functions. The kernel's hyperparameters control the smoothness (length_scale) and periodicity of the kernel (periodicity). Moreover, the noise level of the data is learned explicitly by GPR by an additional WhiteKernel component in the kernel and by the regularization parameter alpha of KRR.



The figure shows that both methods learn reasonable models of the target function. GPR correctly identifies the periodicity of the function to be roughly $2 * \pi$ (6.28), while KRR chooses the doubled periodicity $4 * \pi$. Besides that, GPR provides reasonable confidence bounds on the prediction which are not available for KRR. A major difference between the two methods is the time required for fitting and predicting: while fitting KRR is fast in principle, the grid-search for hyperparameter optimization scales exponentially with the number of hyperparameters ("curse of dimensionality"). The gradient-based optimization of the parameters in GPR does not suffer from this exponential scaling and is thus considerable faster on this example with 3-dimensional hyperparameter space. The time for predicting is similar; however, generating the variance of the predictive distribution of GPR takes considerable longer than just predicting the mean.

GPR on Mauna Loa CO2 data

This example is based on Section 5.4.3 of [RW2006]. It illustrates an example of complex kernel engineering and hyperparameter optimization using gradient ascent on the log-marginal-likelihood. The data consists of the monthly average atmospheric CO2 concentrations (in parts per million by volume (ppmv)) collected at the Mauna Loa Observatory in Hawaii, between 1958 and 1997. The objective is to model the CO2 concentration as a function of the time t.

The kernel is composed of several terms that are responsible for explaining different properties of the signal:

- a long term, smooth rising trend is to be explained by an RBF kernel. The RBF kernel with a large length-scale enforces this component to be smooth; it is not enforced that the trend is rising which leaves this choice to the GP. The specific length-scale and the amplitude are free hyperparameters.
- a seasonal component, which is to be explained by the periodic ExpSineSquared kernel with a fixed periodicity of 1 year. The length-scale of this periodic component, controlling its smoothness, is a free parameter. In order to allow decaying away from exact periodicity, the product with an RBF kernel is taken. The length-scale of this RBF component controls the decay time and is a further free parameter.
- smaller, medium term irregularities are to be explained by a RationalQuadratic kernel component, whose length-scale and alpha parameter, which determines the diffuseness of the length-scales, are to be determined. According to [RW2006], these irregularities can better be explained by a RationalQuadratic than an RBF kernel component, probably because it can accommodate several length-scales.
- a "noise" term, consisting of an RBF kernel contribution, which shall explain the correlated noise components such as local weather phenomena, and a WhiteKernel contribution for the white noise. The relative amplitudes and the RBF's length scale are further free parameters.

Maximizing the log-marginal-likelihood after subtracting the target's mean yields the following kernel with an LML of -83.214:

```
34.4**2 * RBF(length_scale=41.8)
+ 3.27**2 * RBF(length_scale=180) * ExpSineSquared(length_scale=1.44,
                                                   periodicity=1)
+ 0.446**2 * RationalQuadratic(alpha=17.7, length_scale=0.957)
+ 0.197**2 * RBF(length_scale=0.138) + WhiteKernel(noise_level=0.0336)
```

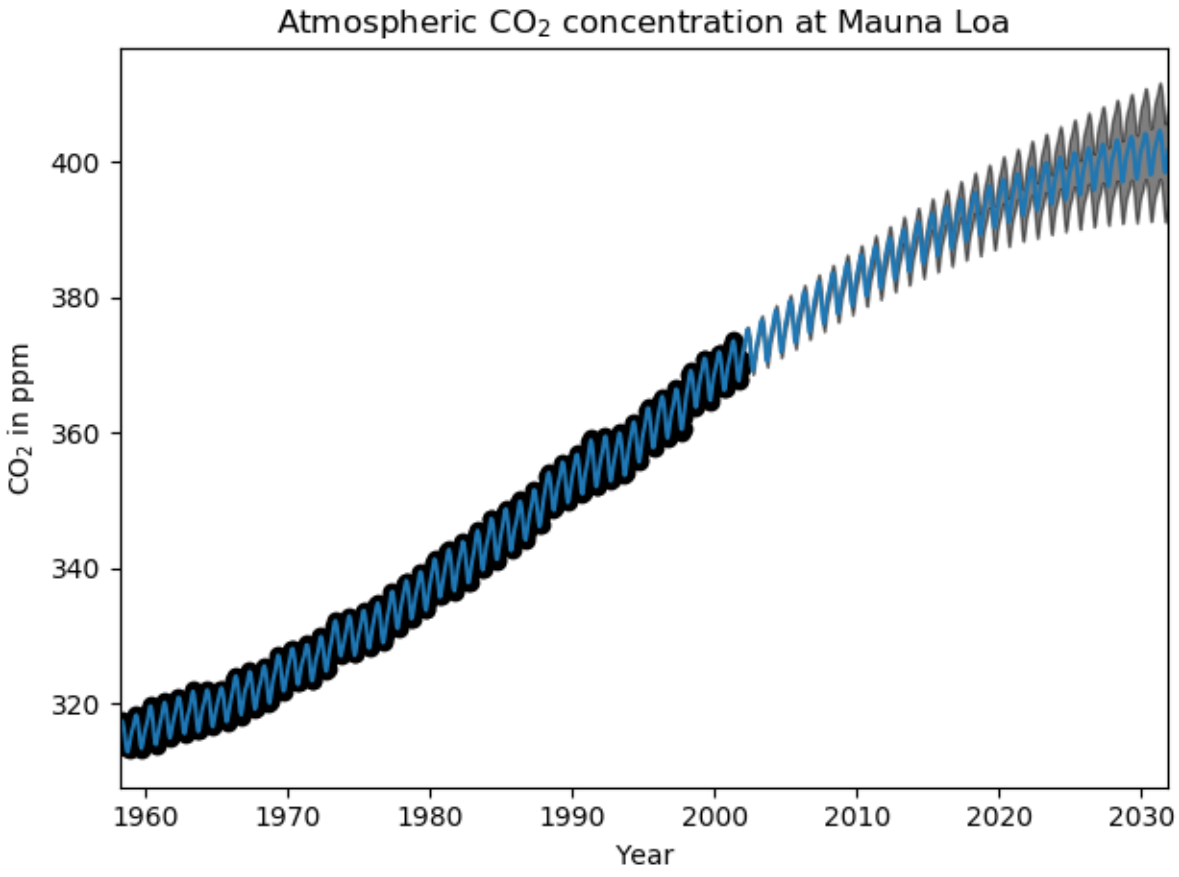
Thus, most of the target signal (34.4ppm) is explained by a long-term rising trend (length-scale 41.8 years). The periodic component has an amplitude of 3.27ppm, a decay time of 180 years and a length-scale of 1.44. The long decay time indicates that we have a locally very close to periodic seasonal component. The correlated noise has an amplitude of 0.197ppm with a length scale of 0.138 years and a white-noise contribution of 0.197ppm. Thus, the overall noise level is very small, indicating that the data can be very well explained by the model. The figure shows also that the model makes very confident predictions until around 2015

Gaussian Process Classification (GPC)

The *GaussianProcessClassifier* implements Gaussian processes (GP) for classification purposes, more specifically for probabilistic classification, where test predictions take the form of class probabilities. GaussianProcessClassifier places a GP prior on a latent function f, which is then squashed through a link function to obtain the probabilistic classification. The latent function f is a so-called nuisance function, whose values are not observed and are not relevant by themselves. Its purpose is to allow a convenient formulation of the model, and f is removed (integrated out) during prediction. GaussianProcessClassifier implements the logistic link function, for which the integral cannot be computed analytically but is easily approximated in the binary case.

In contrast to the regression setting, the posterior of the latent function f is not Gaussian even for a GP prior since a Gaussian likelihood is inappropriate for discrete class labels. Rather, a non-Gaussian likelihood corresponding to the logistic link function (logit) is used. GaussianProcessClassifier approximates the non-Gaussian posterior with a Gaussian based on the Laplace approximation. More details can be found in Chapter 3 of [RW2006].

The GP prior mean is assumed to be zero. The prior's covariance is specified by passing a *kernel* object. The hyperparameters of the kernel are optimized during fitting of GaussianProcessRegressor by maximizing the log-marginal-likelihood (LML) based on the passed `optimizer`. As the LML may have multiple local optima, the optimizer can be started repeatedly by specifying `n_restarts_optimizer`. The first run is always conducted starting from the initial hyperparameter values of the kernel; subsequent runs are conducted from hyperparameter values that have been


Atmospheric CO_2 concentration at Mauna Loa
CO_2 in ppm
400
380
360
340
320
1960
1970
1980
1990
2000
2010
2020
2030
Year

chosen randomly from the range of allowed values. If the initial hyperparameters should be kept fixed, `None` can be passed as optimizer.

GaussianProcessClassifier supports multi-class classification by performing either one-versus-rest or one-versus-one based training and prediction. In one-versus-rest, one binary Gaussian process classifier is fitted for each class, which is trained to separate this class from the rest. In "one_vs_one", one binary Gaussian process classifier is fitted for each pair of classes, which is trained to separate these two classes. The predictions of these binary predictors are combined into multi-class predictions. See the section on *multi-class classification* for more details.

In the case of Gaussian process classification, "one_vs_one" might be computationally cheaper since it has to solve many problems involving only a subset of the whole training set rather than fewer problems on the whole dataset. Since Gaussian process classification scales cubically with the size of the dataset, this might be considerably faster. However, note that "one_vs_one" does not support predicting probability estimates but only plain predictions. Moreover, note that *GaussianProcessClassifier* does not (yet) implement a true multi-class Laplace approximation internally, but as discussed above is based on solving several binary classification tasks internally, which are combined using one-versus-rest or one-versus-one.

GPC examples

Probabilistic predictions with GPC

This example illustrates the predicted probability of GPC for an RBF kernel with different choices of the hyperparameters. The first figure shows the predicted probability of GPC with arbitrarily chosen hyperparameters and with the hyperparameters corresponding to the maximum log-marginal-likelihood (LML).

While the hyperparameters chosen by optimizing LML have a considerable larger LML, they perform slightly worse according to the log-loss on test data. The figure shows that this is because they exhibit a steep change of the class probabilities at the class boundaries (which is good) but have predicted probabilities close to 0.5 far away from the class boundaries (which is bad) This undesirable effect is caused by the Laplace approximation used internally by GPC.

The second figure shows the log-marginal-likelihood for different choices of the kernel's hyperparameters, highlighting the two choices of the hyperparameters used in the first figure by black dots.

Illustration of GPC on the XOR dataset

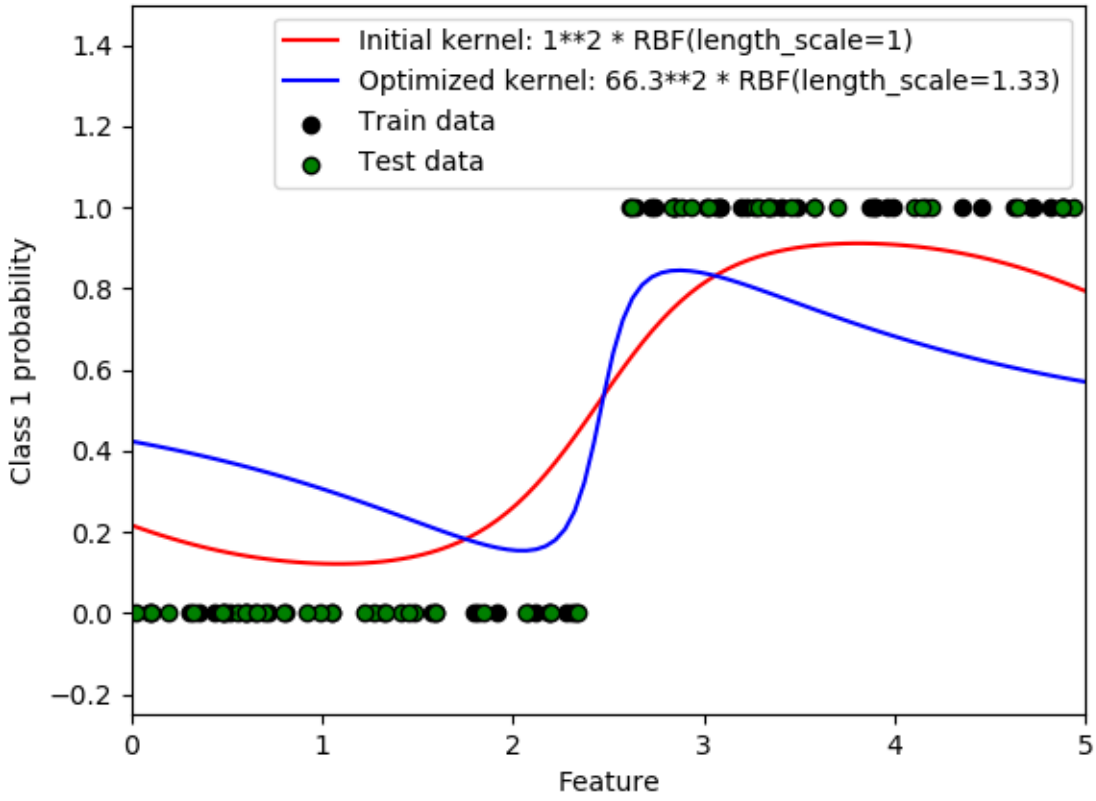
This example illustrates GPC on XOR data. Compared are a stationary, isotropic kernel (*RBF*) and a non-stationary kernel (*DotProduct*). On this particular dataset, the *DotProduct* kernel obtains considerably better results because the class-boundaries are linear and coincide with the coordinate axes. In practice, however, stationary kernels such as *RBF* often obtain better results.

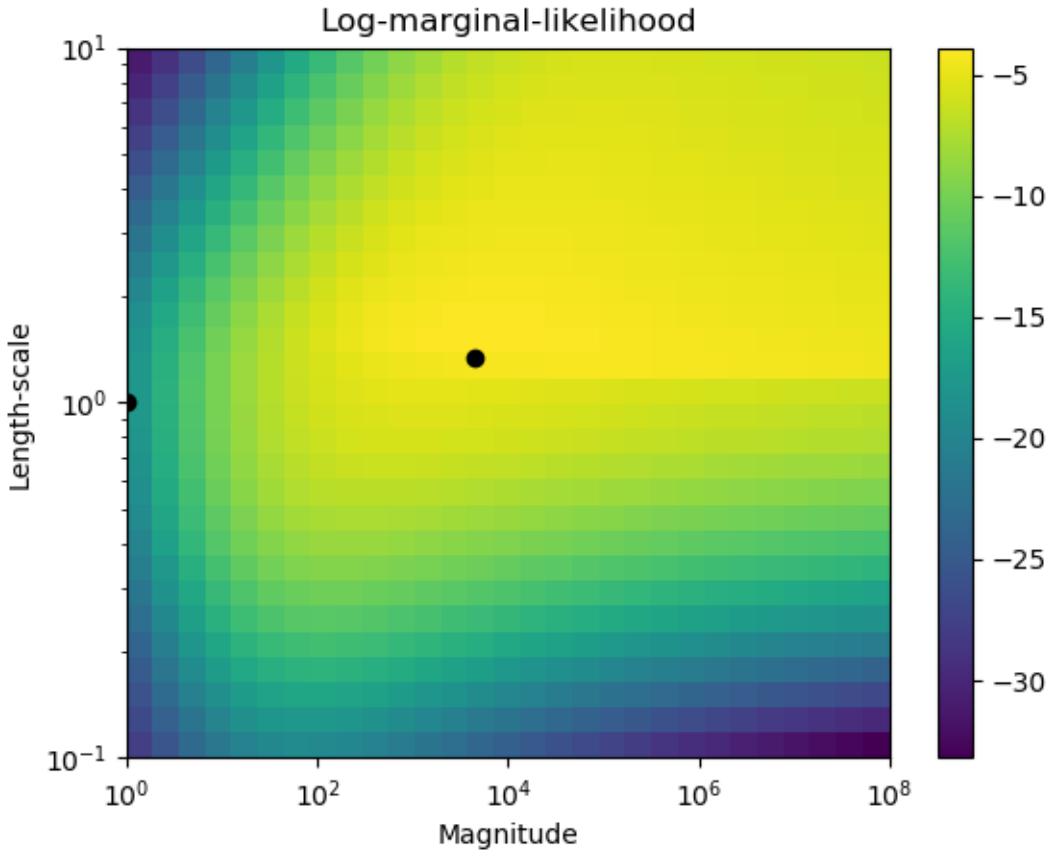
Gaussian process classification (GPC) on iris dataset

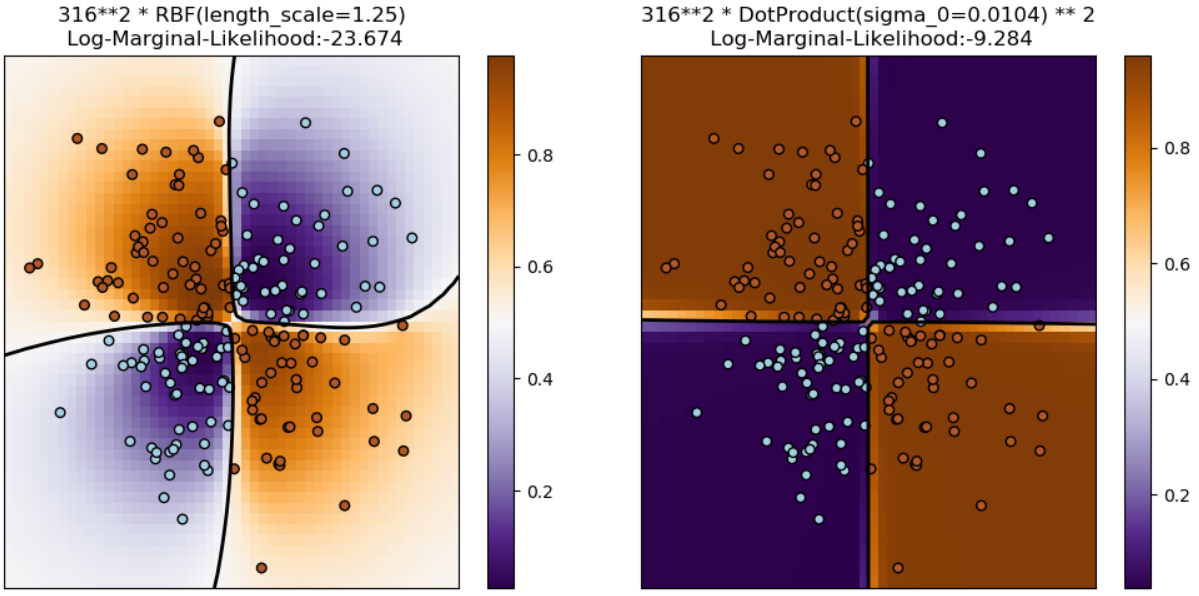
This example illustrates the predicted probability of GPC for an isotropic and anisotropic RBF kernel on a two-dimensional version for the iris-dataset. This illustrates the applicability of GPC to non-binary classification. The anisotropic RBF kernel obtains slightly higher log-marginal-likelihood by assigning different length-scales to the two feature dimensions.

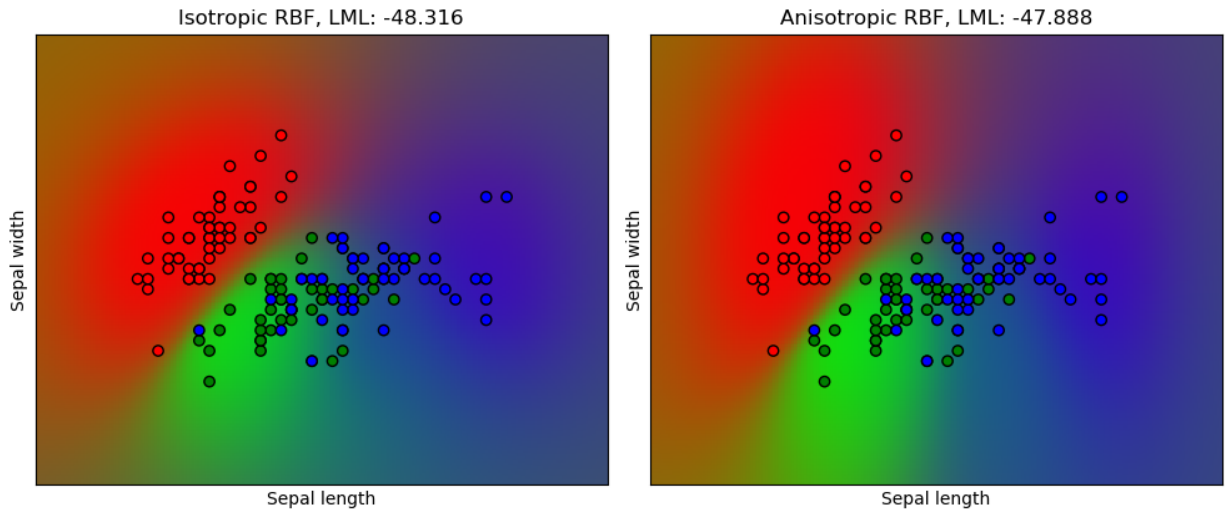
Kernels for Gaussian Processes

Kernels (also called "covariance functions" in the context of GPs) are a crucial ingredient of GPs which determine the shape of prior and posterior of the GP. They encode the assumptions on the function being learned by defining the


Initial kernel: 1**2 * RBF(length_scale=1)
Optimized kernel: 66.3**2 * RBF(length_scale=1.33)
Train data
Test data
Class 1 probability
Feature
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
1
2
3
4
5


Log-marginal-likelihood
Length-scale
Magnitude
10^1
10^0
10^{-1}
10^0
10^2
10^4
10^6
10^8
−5
−10
−15
−20
−25
−30


316**2 * RBF(length_scale=1.25)
Log-Marginal-Likelihood:-23.674
316**2 * DotProduct(sigma_0=0.0104) ** 2
Log-Marginal-Likelihood:-9.284
0.8
0.6
0.4
0.2



"similarity" of two datapoints combined with the assumption that similar datapoints should have similar target values. Two categories of kernels can be distinguished: stationary kernels depend only on the distance of two datapoints and not on their absolute values $k(x_i, x_j) = k(d(x_i, x_j))$ and are thus invariant to translations in the input space, while non-stationary kernels depend also on the specific values of the datapoints. Stationary kernels can further be subdivided into isotropic and anisotropic kernels, where isotropic kernels are also invariant to rotations in the input space. For more details, we refer to Chapter 4 of [RW2006].

Gaussian Process Kernel API

The main usage of a *Kernel* is to compute the GP's covariance between datapoints. For this, the method `__call__` of the kernel can be called. This method can either be used to compute the "auto-covariance" of all pairs of datapoints in a 2d array X, or the "cross-covariance" of all combinations of datapoints of a 2d array X with datapoints in a 2d array Y. The following identity holds true for all kernels k (except for the *WhiteKernel*): `k(X) == K(X, Y=X)`

If only the diagonal of the auto-covariance is being used, the method `diag()` of a kernel can be called, which is more computationally efficient than the equivalent call to `__call__`: `np.diag(k(X, X)) == k.diag(X)`

Kernels are parameterized by a vector θ of hyperparameters. These hyperparameters can for instance control length-scales or periodicity of a kernel (see below). All kernels support computing analytic gradients of the kernel's auto-covariance with respect to θ via setting `eval_gradient=True` in the `__call__` method. This gradient is used by the Gaussian process (both regressor and classifier) in computing the gradient of the log-marginal-likelihood, which in turn is used to determine the value of θ, which maximizes the log-marginal-likelihood, via gradient ascent. For each hyperparameter, the initial value and the bounds need to be specified when creating an instance of the kernel. The current value of θ can be get and set via the property `theta` of the kernel object. Moreover, the bounds of the hyperparameters can be accessed by the property `bounds` of the kernel. Note that both properties (theta and bounds) return log-transformed values of the internally used values since those are typically more amenable to gradient-based optimization. The specification of each hyperparameter is stored in the form of an instance of *Hyperparameter* in the respective kernel. Note that a kernel using a hyperparameter with name "x" must have the attributes self.x and self.x_bounds.

The abstract base class for all kernels is *Kernel*. Kernel implements a similar interface as `Estimator`, providing the methods `get_params()`, `set_params()`, and `clone()`. This allows setting kernel values also via meta-estimators such as `Pipeline` or `GridSearch`. Note that due to the nested structure of kernels (by applying kernel

operators, see below), the names of kernel parameters might become relatively complicated. In general, for a binary kernel operator, parameters of the left operand are prefixed with `k1__` and parameters of the right operand with `k2__`. An additional convenience method is `clone_with_theta(theta)`, which returns a cloned version of the kernel but with the hyperparameters set to `theta`. An illustrative example:

```
>>> from sklearn.gaussian_process.kernels import ConstantKernel, RBF
>>> kernel = ConstantKernel(constant_value=1.0, constant_value_bounds=(0.0, 10.0)) *␣
↪RBF(length_scale=0.5, length_scale_bounds=(0.0, 10.0)) + RBF(length_scale=2.0,␣
↪length_scale_bounds=(0.0, 10.0))
>>> for hyperparameter in kernel.hyperparameters: print(hyperparameter)
Hyperparameter(name='k1__k1__constant_value', value_type='numeric', bounds=array([[ 0.
↪, 10.]]), n_elements=1, fixed=False)
Hyperparameter(name='k1__k2__length_scale', value_type='numeric', bounds=array([[ 0.,␣
↪10.]]), n_elements=1, fixed=False)
Hyperparameter(name='k2__length_scale', value_type='numeric', bounds=array([[ 0., 10.
↪]]), n_elements=1, fixed=False)
>>> params = kernel.get_params()
>>> for key in sorted(params): print("%s : %s" % (key, params[key]))
k1 : 1**2 * RBF(length_scale=0.5)
k1__k1 : 1**2
k1__k1__constant_value : 1.0
k1__k1__constant_value_bounds : (0.0, 10.0)
k1__k2 : RBF(length_scale=0.5)
k1__k2__length_scale : 0.5
k1__k2__length_scale_bounds : (0.0, 10.0)
k2 : RBF(length_scale=2)
k2__length_scale : 2.0
k2__length_scale_bounds : (0.0, 10.0)
>>> print(kernel.theta)  # Note: log-transformed
[ 0.         -0.69314718  0.69314718]
>>> print(kernel.bounds)  # Note: log-transformed
[[      -inf 2.30258509]
 [      -inf 2.30258509]
 [      -inf 2.30258509]]
```

All Gaussian process kernels are interoperable with *sklearn.metrics.pairwise* and vice versa: instances of subclasses of *Kernel* can be passed as `metric` to `pairwise_kernels` from *sklearn.metrics.pairwise*. Moreover, kernel functions from pairwise can be used as GP kernels by using the wrapper class *PairwiseKernel*. The only caveat is that the gradient of the hyperparameters is not analytic but numeric and all those kernels support only isotropic distances. The parameter `gamma` is considered to be a hyperparameter and may be optimized. The other kernel parameters are set directly at initialization and are kept fixed.

Basic kernels

The *ConstantKernel* kernel can be used as part of a *Product* kernel where it scales the magnitude of the other factor (kernel) or as part of a *Sum* kernel, where it modifies the mean of the Gaussian process. It depends on a parameter $constant\_value$. It is defined as:

$$k(x_i, x_j) = constant\_value \;\forall\; x_1, x_2$$

The main use-case of the *WhiteKernel* kernel is as part of a sum-kernel where it explains the noise-component of the signal. Tuning its parameter $noise\_level$ corresponds to estimating the noise-level. It is defined as:

$$k(x_i, x_j) = noise\_level \text{ if } x_i == x_j \text{ else } 0$$

Kernel operators

Kernel operators take one or two base kernels and combine them into a new kernel. The `Sum` kernel takes two kernels $k1$ and $k2$ and combines them via $k_{sum}(X, Y) = k1(X, Y) + k2(X, Y)$. The `Product` kernel takes two kernels $k1$ and $k2$ and combines them via $k_{product}(X, Y) = k1(X, Y) * k2(X, Y)$. The `Exponentiation` kernel takes one base kernel and a scalar parameter $exponent$ and combines them via $k_{exp}(X, Y) = k(X, Y)^{\text{exponent}}$.

Radial-basis function (RBF) kernel

The `RBF` kernel is a stationary kernel. It is also known as the "squared exponential" kernel. It is parameterized by a length-scale parameter $l > 0$, which can either be a scalar (isotropic variant of the kernel) or a vector with the same number of dimensions as the inputs x (anisotropic variant of the kernel). The kernel is given by:

$$k(x_i, x_j) = \exp\left(-\frac{1}{2} d(x_i/l, x_j/l)^2\right)$$

This kernel is infinitely differentiable, which implies that GPs with this kernel as covariance function have mean square derivatives of all orders, and are thus very smooth. The prior and posterior of a GP resulting from an RBF kernel are shown in the following figure:

Matérn kernel

The `Matern` kernel is a stationary kernel and a generalization of the `RBF` kernel. It has an additional parameter ν which controls the smoothness of the resulting function. It is parameterized by a length-scale parameter $l > 0$, which can either be a scalar (isotropic variant of the kernel) or a vector with the same number of dimensions as the inputs x (anisotropic variant of the kernel). The kernel is given by:

$$k(x_i, x_j) = \sigma^2 \frac{1}{\Gamma(\nu)2^{\nu-1}} \left(\gamma\sqrt{2\nu} d(x_i/l, x_j/l)\right)^{\nu} K_{\nu}\left(\gamma\sqrt{2\nu} d(x_i/l, x_j/l)\right),$$

As $\nu \to \infty$, the Matérn kernel converges to the RBF kernel. When $\nu = 1/2$, the Matérn kernel becomes identical to the absolute exponential kernel, i.e.,

$$k(x_i, x_j) = \sigma^2 \exp\left(-\gamma d(x_i/l, x_j/l)\right) \qquad \nu = \tfrac{1}{2}$$

In particular, $\nu = 3/2$:

$$k(x_i, x_j) = \sigma^2 \left(1 + \gamma\sqrt{3} d(x_i/l, x_j/l)\right) \exp\left(-\gamma\sqrt{3} d(x_i/l, x_j/l)\right) \qquad \nu = \tfrac{3}{2}$$

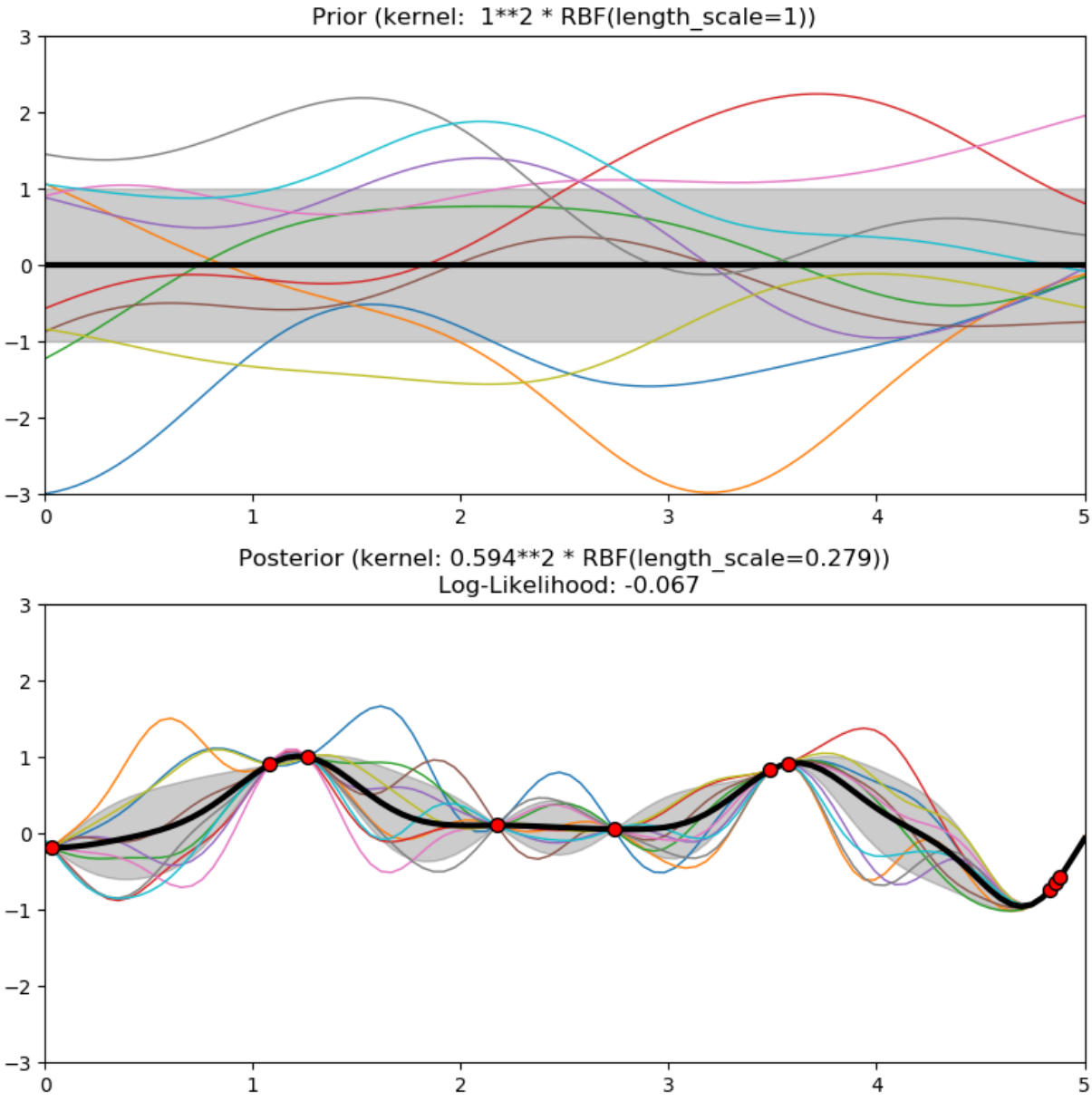
and $\nu = 5/2$:

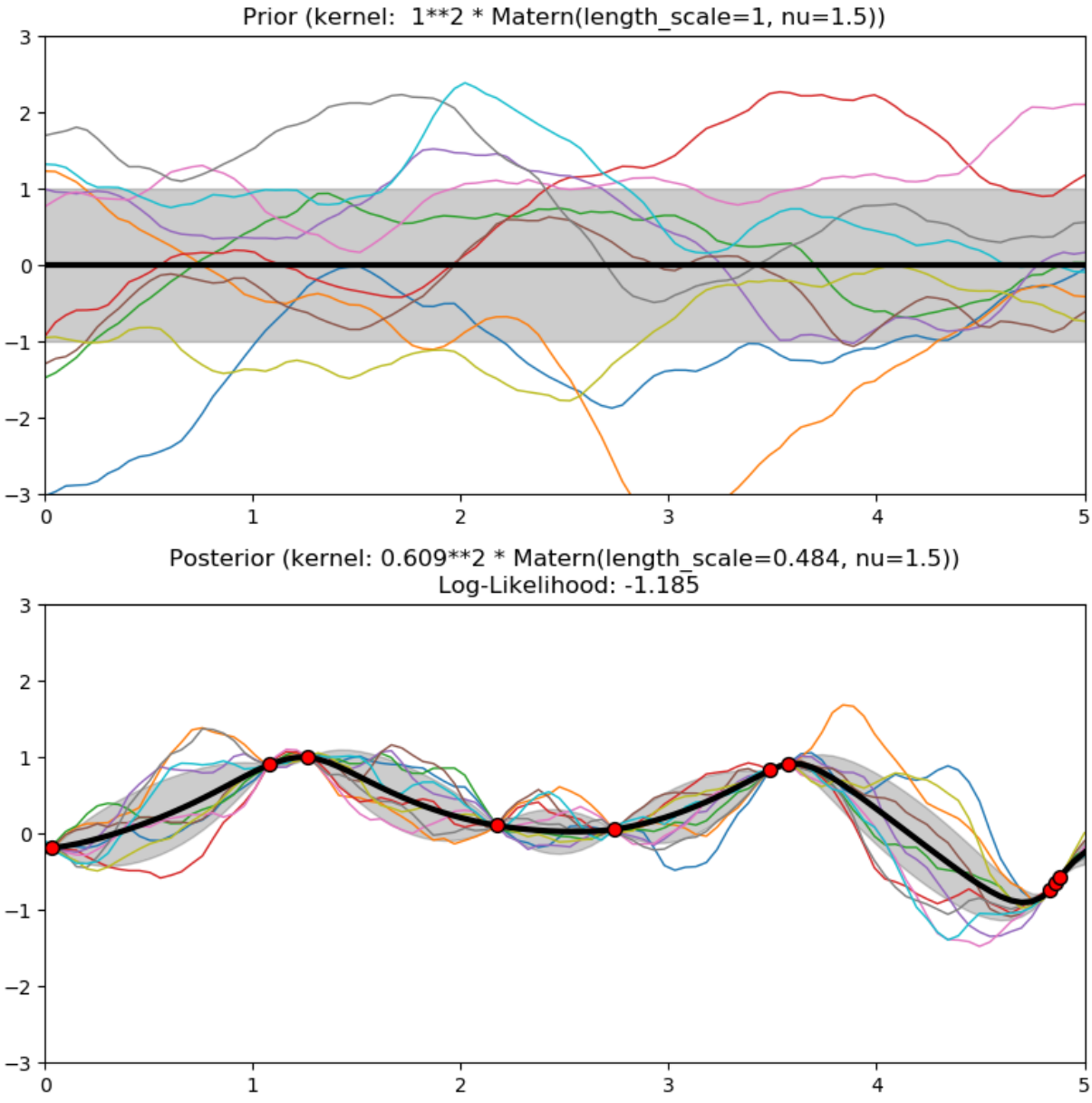
$$k(x_i, x_j) = \sigma^2 \left(1 + \gamma\sqrt{5} d(x_i/l, x_j/l) + \frac{5}{3}\gamma^2 d(x_i/l, x_j/l)^2\right) \exp\left(-\gamma\sqrt{5} d(x_i/l, x_j/l)\right) \qquad \nu = \tfrac{5}{2}$$

are popular choices for learning functions that are not infinitely differentiable (as assumed by the RBF kernel) but at least once ($\nu = 3/2$) or twice differentiable ($\nu = 5/2$).

The flexibility of controlling the smoothness of the learned function via ν allows adapting to the properties of the true underlying functional relation. The prior and posterior of a GP resulting from a Matérn kernel are shown in the following figure:

See [RW2006], pp84 for further details regarding the different variants of the Matérn kernel.


Prior (kernel: 1**2 * RBF(length_scale=1))
Posterior (kernel: 0.594**2 * RBF(length_scale=0.279))
Log-Likelihood: -0.067

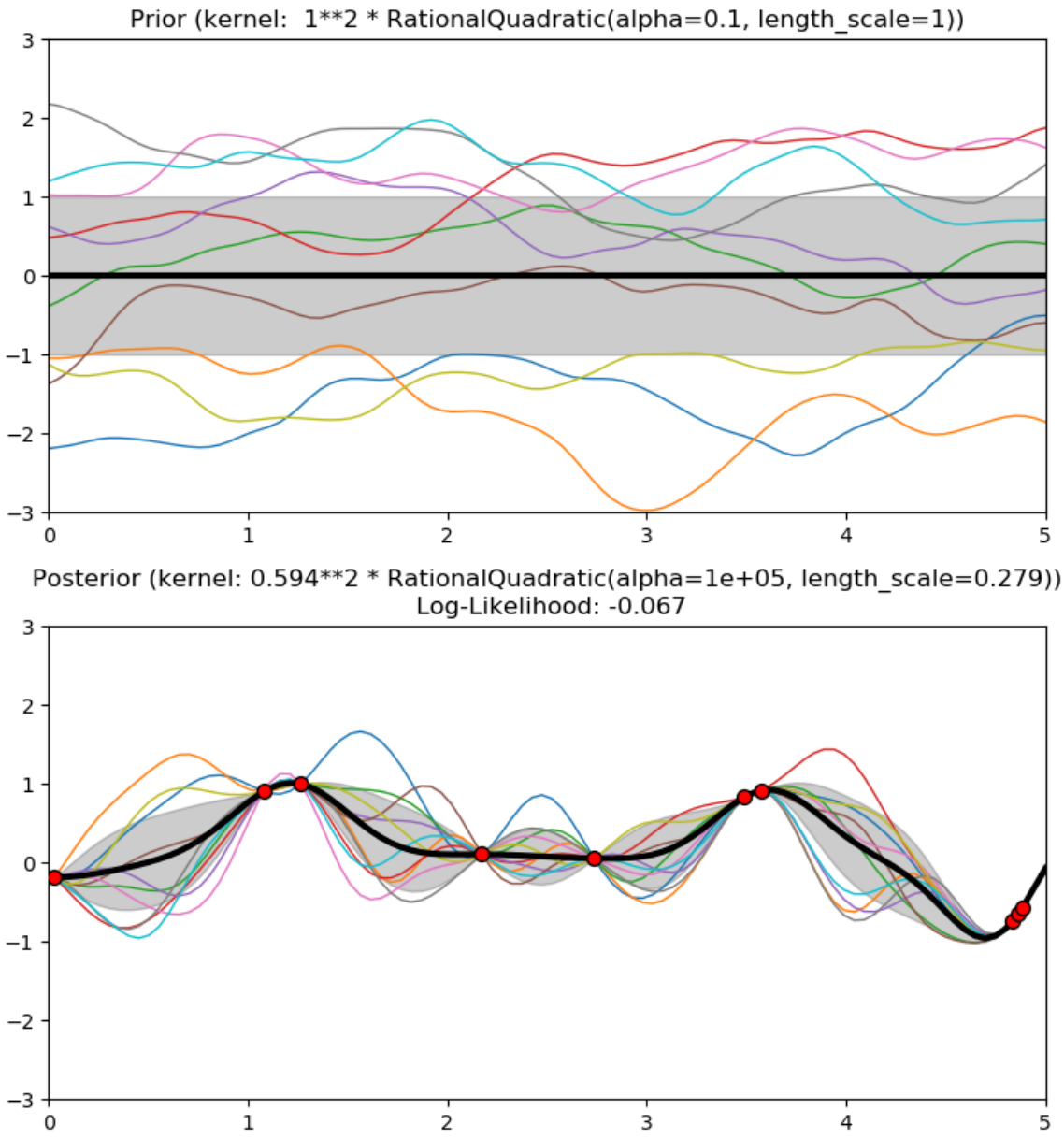

Prior (kernel: 1**2 * Matern(length_scale=1, nu=1.5))
Posterior (kernel: 0.609**2 * Matern(length_scale=0.484, nu=1.5))
Log-Likelihood: -1.185

Rational quadratic kernel

The `RationalQuadratic` kernel can be seen as a scale mixture (an infinite sum) of `RBF` kernels with different characteristic length-scales. It is parameterized by a length-scale parameter $l > 0$ and a scale mixture parameter $\alpha > 0$ Only the isotropic variant where l is a scalar is supported at the moment. The kernel is given by:

$$k(x_i, x_j) = \left(1 + \frac{d(x_i, x_j)^2}{2\alpha l^2}\right)^{-\alpha}$$

The prior and posterior of a GP resulting from a `RationalQuadratic` kernel are shown in the following figure:

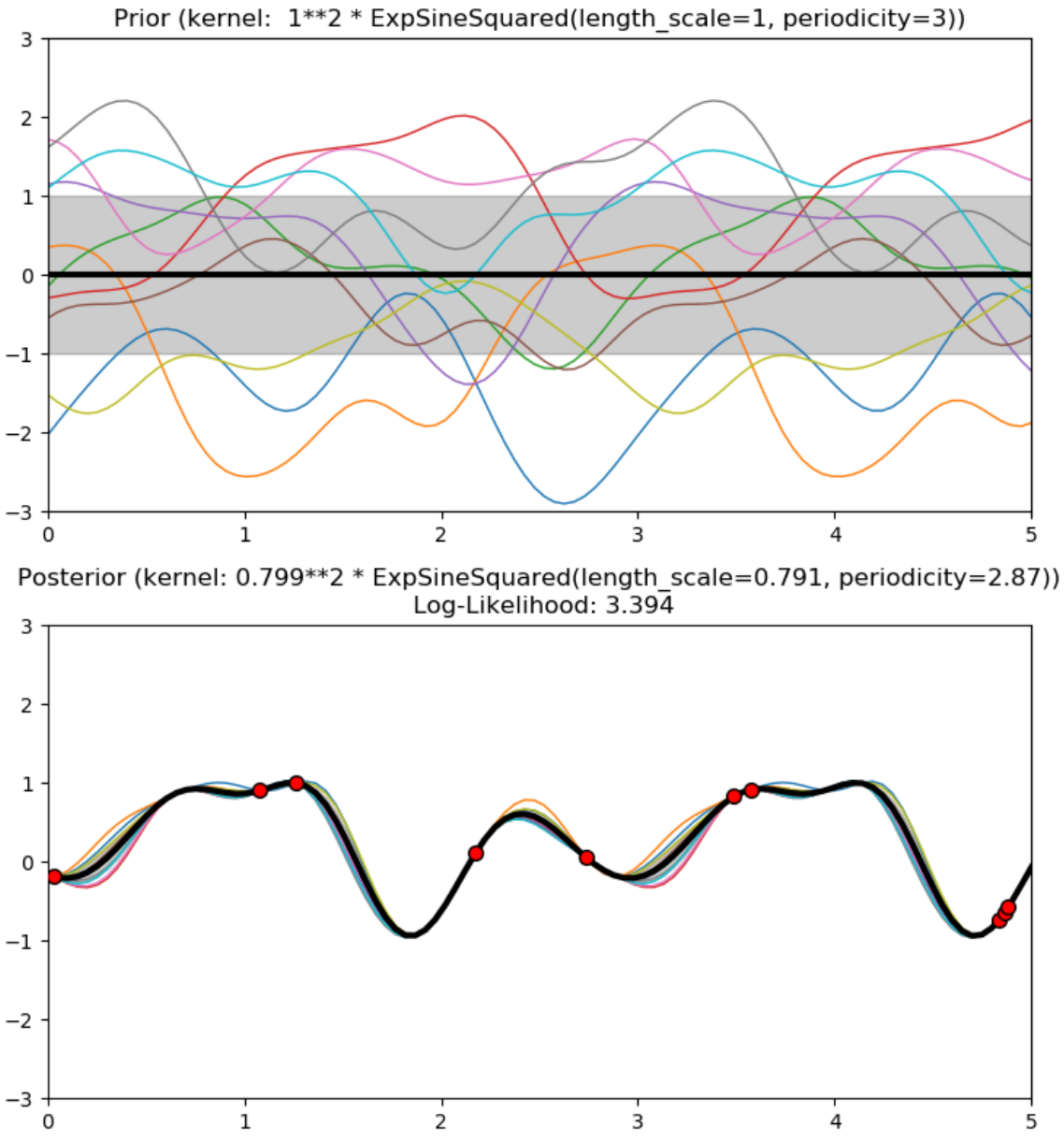


Exp-Sine-Squared kernel

The `ExpSineSquared` kernel allows modeling periodic functions. It is parameterized by a length-scale parameter $l > 0$ and a periodicity parameter $p > 0$. Only the isotropic variant where l is a scalar is supported at the moment. The kernel is given by:

$$k(x_i, x_j) = \exp\left(-2\left(\sin(\pi/p * d(x_i, x_j))/l\right)^2\right)$$

The prior and posterior of a GP resulting from an ExpSineSquared kernel are shown in the following figure:

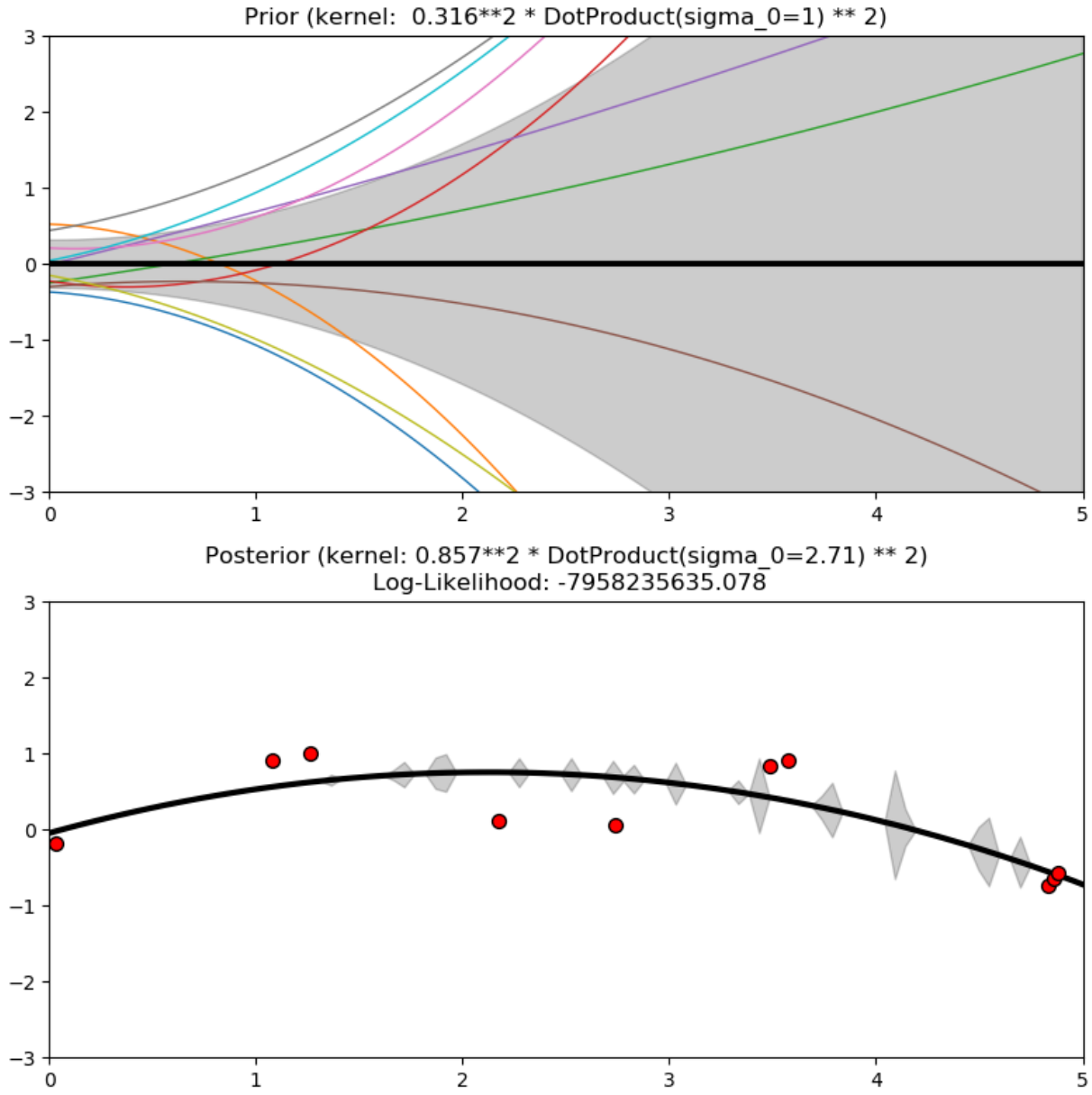


Dot-Product kernel

The `DotProduct` kernel is non-stationary and can be obtained from linear regression by putting $N(0, 1)$ priors on the coefficients of $x_d(d = 1, ..., D)$ and a prior of $N(0, \sigma_0^2)$ on the bias. The `DotProduct` kernel is invariant to a rotation of the coordinates about the origin, but not translations. It is parameterized by a parameter σ_0^2. For $\sigma_0^2 = 0$, the kernel is called the homogeneous linear kernel, otherwise it is inhomogeneous. The kernel is given by

$$k(x_i, x_j) = \sigma_0^2 + x_i \cdot x_j$$

The `DotProduct` kernel is commonly combined with exponentiation. An example with exponent 2 is shown in the following figure:

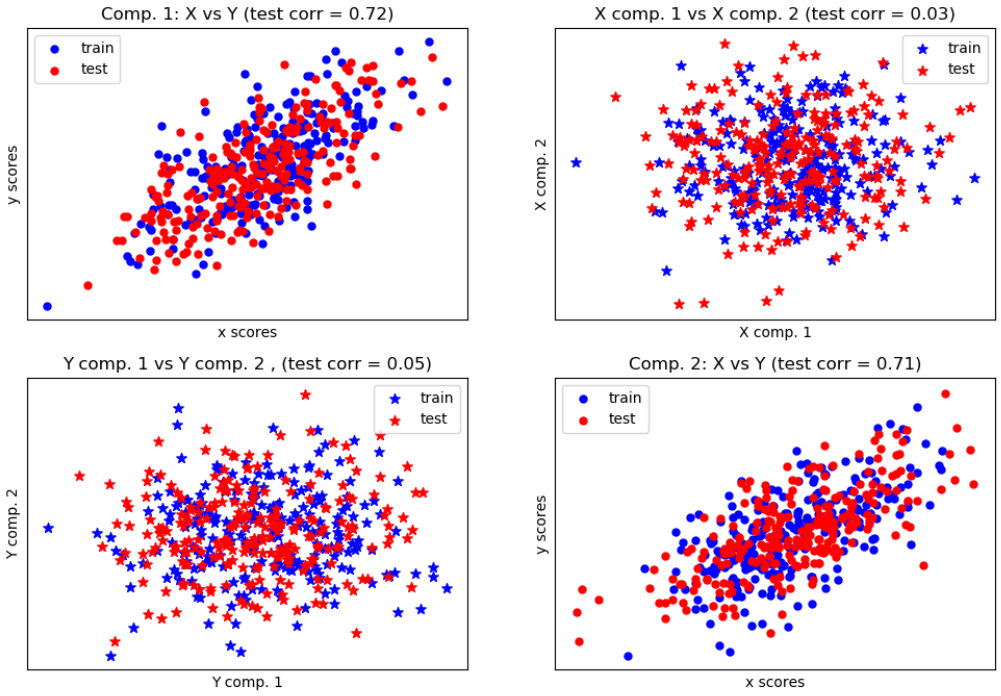


References

4.1.8 Cross decomposition

The cross decomposition module contains two main families of algorithms: the partial least squares (PLS) and the canonical correlation analysis (CCA).

These families of algorithms are useful to find linear relations between two multivariate datasets: the `X` and `Y` arguments of the `fit` method are 2D arrays.



Cross decomposition algorithms find the fundamental relations between two matrices (X and Y). They are latent variable approaches to modeling the covariance structures in these two spaces. They will try to find the multidimensional direction in the X space that explains the maximum multidimensional variance direction in the Y space. PLS-regression is particularly suited when the matrix of predictors has more variables than observations, and when there is multicollinearity among X values. By contrast, standard regression will fail in these cases.

Classes included in this module are *`PLSRegression`* *`PLSCanonical`*, *`CCA`* and *`PLSSVD`*

Reference:

- JA Wegelin A survey of Partial Least Squares (PLS) methods, with emphasis on the two-block case

Examples:

- *Compare cross decomposition methods*

4.1.9 Naive Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. Bayes' theorem states the following relationship, given class variable y and dependent feature vector x_1 through x_n, :

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots x_n \mid y)}{P(x_1, \ldots, x_n)}$$

Using the naive conditional independence assumption that

$$P(x_i|y, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = P(x_i|y),$$

for all i, this relationship is simplified to

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \ldots, x_n)}$$

Since $P(x_1, \ldots, x_n)$ is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, \ldots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

$$\Downarrow$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

and we can use Maximum A Posteriori (MAP) estimation to estimate $P(y)$ and $P(x_i \mid y)$; the former is then the relative frequency of class y in the training set.

The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(x_i \mid y)$.

In spite of their apparently over-simplified assumptions, naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering. They require a small amount of training data to estimate the necessary parameters. (For theoretical reasons why naive Bayes works well, and on which types of data it does, see the references below.)

Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods. The decoupling of the class conditional feature distributions means that each distribution can be independently estimated as a one dimensional distribution. This in turn helps to alleviate problems stemming from the curse of dimensionality.

On the flip side, although naive Bayes is known as a decent classifier, it is known to be a bad estimator, so the probability outputs from `predict_proba` are not to be taken too seriously.

References:

- H. Zhang (2004). The optimality of Naive Bayes. Proc. FLAIRS.

Gaussian Naive Bayes

`GaussianNB` implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

The parameters σ_y and μ_y are estimated using maximum likelihood.

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.naive_bayes import GaussianNB
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_
↪state=0)
>>> gnb = GaussianNB()
>>> y_pred = gnb.fit(X_train, y_train).predict(X_test)
>>> print("Number of mislabeled points out of a total %d points : %d"
...       % (X_test.shape[0], (y_test != y_pred).sum()))
Number of mislabeled points out of a total 75 points : 4
```

Multinomial Naive Bayes

MultinomialNB implements the naive Bayes algorithm for multinomially distributed data, and is one of the two classic naive Bayes variants used in text classification (where the data are typically represented as word vector counts, although tf-idf vectors are also known to work well in practice). The distribution is parametrized by vectors $\theta_y = (\theta_{y1}, \ldots, \theta_{yn})$ for each class y, where n is the number of features (in text classification, the size of the vocabulary) and θ_{yi} is the probability $P(x_i \mid y)$ of feature i appearing in a sample belonging to class y.

The parameters θ_y is estimated by a smoothed version of maximum likelihood, i.e. relative frequency counting:

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n}$$

where $N_{yi} = \sum_{x \in T} x_i$ is the number of times feature i appears in a sample of class y in the training set T, and $N_y = \sum_{i=1}^{n} N_{yi}$ is the total count of all features for class y.

The smoothing priors $\alpha \geq 0$ accounts for features not present in the learning samples and prevents zero probabilities in further computations. Setting $\alpha = 1$ is called Laplace smoothing, while $\alpha < 1$ is called Lidstone smoothing.

Complement Naive Bayes

ComplementNB implements the complement naive Bayes (CNB) algorithm. CNB is an adaptation of the standard multinomial naive Bayes (MNB) algorithm that is particularly suited for imbalanced data sets. Specifically, CNB uses statistics from the *complement* of each class to compute the model's weights. The inventors of CNB show empirically that the parameter estimates for CNB are more stable than those for MNB. Further, CNB regularly outperforms MNB (often by a considerable margin) on text classification tasks. The procedure for calculating the weights is as follows:

$$\hat{\theta}_{ci} = \frac{\alpha_i + \sum_{j:y_j \neq c} d_{ij}}{\alpha + \sum_{j:y_j \neq c} \sum_k d_{kj}}$$

$$w_{ci} = \log \hat{\theta}_{ci}$$

$$w_{ci} = \frac{w_{ci}}{\sum_j |w_{cj}|}$$

where the summations are over all documents j not in class c, d_{ij} is either the count or tf-idf value of term i in document j, α_i is a smoothing hyperparameter like that found in MNB, and $\alpha = \sum_i \alpha_i$. The second normalization addresses the tendency for longer documents to dominate parameter estimates in MNB. The classification rule is:

$$\hat{c} = \arg\min_c \sum_i t_i w_{ci}$$

i.e., a document is assigned to the class that is the *poorest* complement match.

References:

- Rennie, J. D., Shih, L., Teevan, J., & Karger, D. R. (2003). Tackling the poor assumptions of naive bayes text classifiers. In ICML (Vol. 3, pp. 616-623).

Bernoulli Naive Bayes

`BernoulliNB` implements the naive Bayes training and classification algorithms for data that is distributed according to multivariate Bernoulli distributions; i.e., there may be multiple features but each one is assumed to be a binary-valued (Bernoulli, boolean) variable. Therefore, this class requires samples to be represented as binary-valued feature vectors; if handed any other kind of data, a `BernoulliNB` instance may binarize its input (depending on the `binarize` parameter).

The decision rule for Bernoulli naive Bayes is based on

$$P(x_i \mid y) = P(i \mid y)x_i + (1 - P(i \mid y))(1 - x_i)$$

which differs from multinomial NB's rule in that it explicitly penalizes the non-occurrence of a feature i that is an indicator for class y, where the multinomial variant would simply ignore a non-occurring feature.

In the case of text classification, word occurrence vectors (rather than word count vectors) may be used to train and use this classifier. `BernoulliNB` might perform better on some datasets, especially those with shorter documents. It is advisable to evaluate both models, if time permits.

References:

- C.D. Manning, P. Raghavan and H. Schütze (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 234-265.
- A. McCallum and K. Nigam (1998). A comparison of event models for Naive Bayes text classification. Proc. AAAI/ICML-98 Workshop on Learning for Text Categorization, pp. 41-48.
- V. Metsis, I. Androutsopoulos and G. Paliouras (2006). Spam filtering with Naive Bayes – Which Naive Bayes? 3rd Conf. on Email and Anti-Spam (CEAS).

Categorical Naive Bayes

`CategoricalNB` implements the categorical naive Bayes algorithm for categorically distributed data. It assumes that each feature, which is described by the index i, has its own categorical distribution.

For each feature i in the training set X, `CategoricalNB` estimates a categorical distribution for each feature i of X conditioned on the class y. The index set of the samples is defined as $J = \{1, \ldots, m\}$, with m as the number of samples.

The probability of category t in feature i given class c is estimated as:

$$P(x_i = t \mid y = c \,;\, \alpha) = \frac{N_{tic} + \alpha}{N_c + \alpha n_i},$$

where $N_{tic} = |\{j \in J \mid x_{ij} = t, y_j = c\}|$ is the number of times category t appears in the samples x_i, which belong to class c, $N_c = |\{j \in J \mid y_j = c\}|$ is the number of samples with class c, α is a smoothing parameter and n_i is the number of available categories of feature i.

CategoricalNB assumes that the sample matrix X is encoded (for instance with the help of `OrdinalEncoder`) such that all categories for each feature i are represented with numbers $0, ..., n_i - 1$ where n_i is the number of available categories of feature i.

Out-of-core naive Bayes model fitting

Naive Bayes models can be used to tackle large scale classification problems for which the full training set might not fit in memory. To handle this case, *MultinomialNB*, *BernoulliNB*, and *GaussianNB* expose a `partial_fit` method that can be used incrementally as done with other classifiers as demonstrated in *Out-of-core classification of text documents*. All naive Bayes classifiers support sample weighting.

Contrary to the `fit` method, the first call to `partial_fit` needs to be passed the list of all the expected class labels.

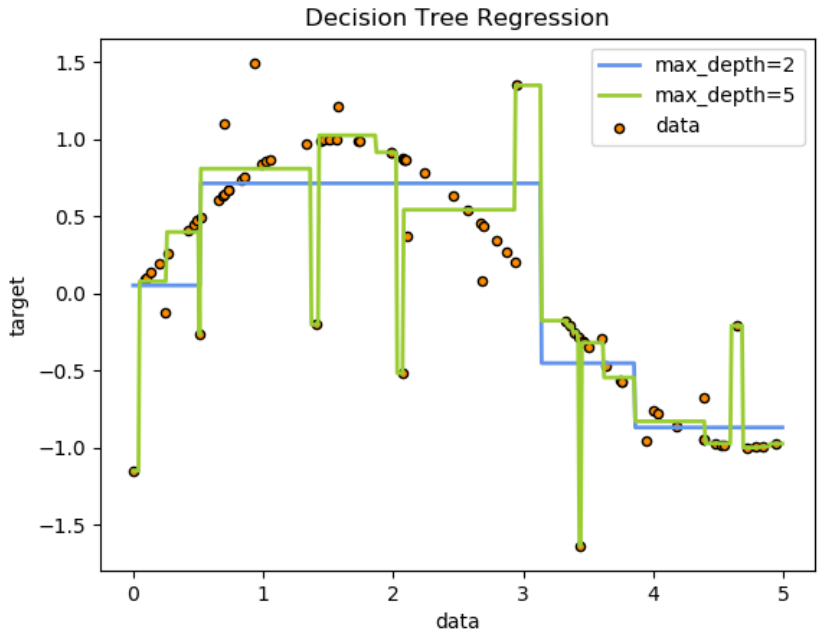
For an overview of available strategies in scikit-learn, see also the *out-of-core learning* documentation.

Note: The `partial_fit` method call of naive Bayes models introduces some computational overhead. It is recommended to use data chunk sizes that are as large as possible, that is as the available RAM allows.

4.1.10 Decision Trees

Decision Trees (DTs) are a non-parametric supervised learning method used for *classification* and *regression*. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

For instance, in the example below, decision trees learn from data to approximate a sine curve with a set of if-then-else decision rules. The deeper the tree, the more complex the decision rules and the fitter the model.



Some advantages of decision trees are:

- Simple to understand and to interpret. Trees can be visualised.

- Requires little data preparation. Other techniques often require data normalisation, dummy variables need to be created and blank values to be removed. Note however that this module does not support missing values.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data. Other techniques are usually specialised in analysing datasets that have only one type of variable. See *algorithms* for more information.
- Able to handle multi-output problems.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
- Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

The disadvantages of decision trees include:

- Decision-tree learners can create over-complex trees that do not generalise the data well. This is called overfitting. Mechanisms such as pruning (not currently supported), setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.
- The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.
- There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

Classification

DecisionTreeClassifier is a class capable of performing multi-class classification on a dataset.

As with other classifiers, *DecisionTreeClassifier* takes as input two arrays: an array X, sparse or dense, of size `[n_samples, n_features]` holding the training samples, and an array Y of integer values, size `[n_samples]`, holding the class labels for the training samples:

```
>>> from sklearn import tree
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(X, Y)
```

After being fitted, the model can then be used to predict the class of samples:

```
>>> clf.predict([[2., 2.]])
array([1])
```

Alternatively, the probability of each class can be predicted, which is the fraction of training samples of the same class in a leaf:

```
>>> clf.predict_proba([[2., 2.]])
array([[0., 1.]])
```

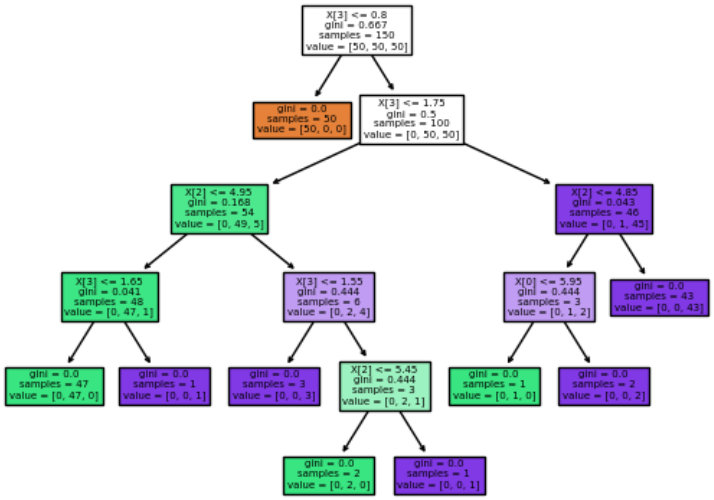
DecisionTreeClassifier is capable of both binary (where the labels are [-1, 1]) classification and multiclass (where the labels are [0, . . . , K-1]) classification.

Using the Iris dataset, we can construct a tree as follows:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn import tree
>>> X, y = load_iris(return_X_y=True)
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(X, y)
```

Once trained, you can plot the tree with the plot_tree function:

```
>>> tree.plot_tree(clf.fit(iris.data, iris.target))
```



We can also export the tree in Graphviz format using the *export_graphviz* exporter. If you use the conda package manager, the graphviz binaries

and the python package can be installed with

    conda install python-graphviz

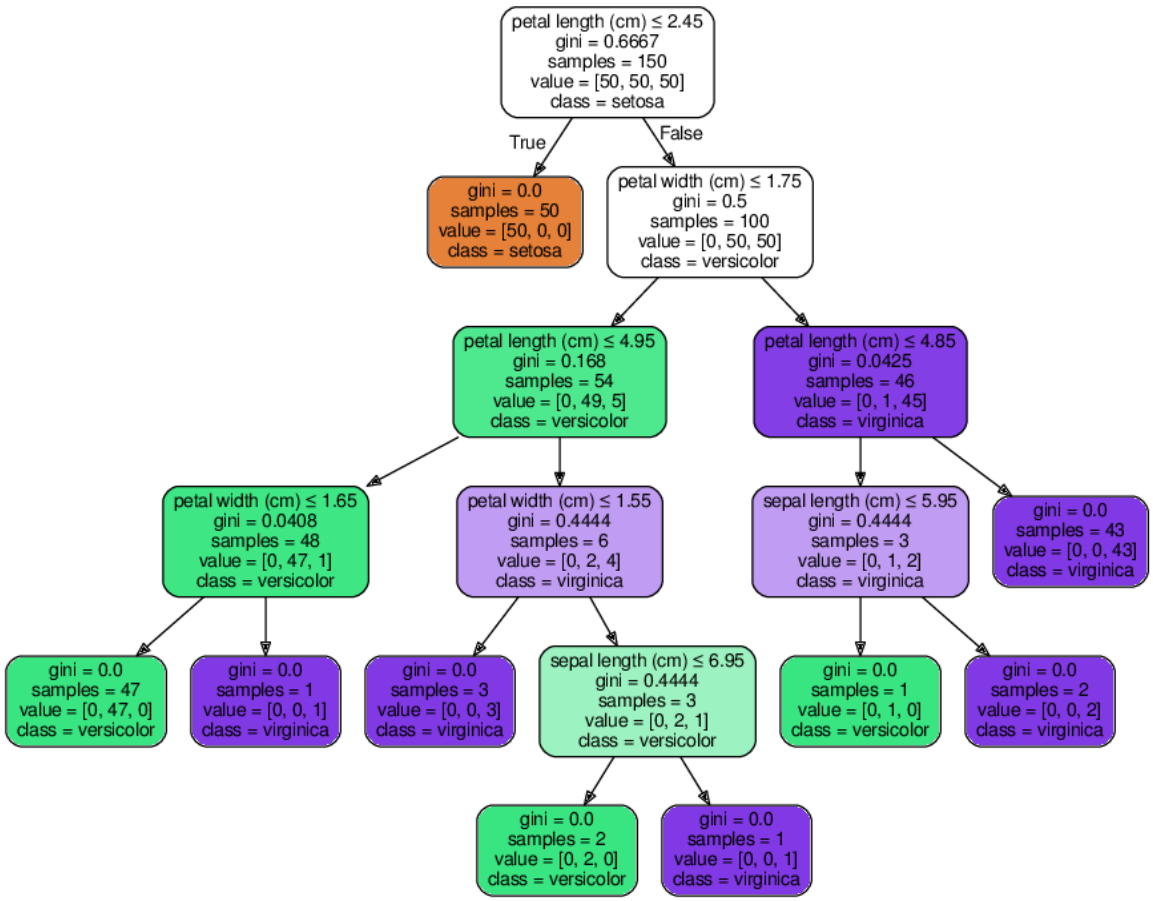
Alternatively binaries for graphviz can be downloaded from the graphviz project homepage, and the Python wrapper installed from pypi with `pip install graphviz`.

Below is an example graphviz export of the above tree trained on the entire iris dataset; the results are saved in an output file `iris.pdf`:

```
>>> import graphviz
>>> dot_data = tree.export_graphviz(clf, out_file=None)
>>> graph = graphviz.Source(dot_data)
>>> graph.render("iris")
```

The *export_graphviz* exporter also supports a variety of aesthetic options, including coloring nodes by their class (or value for regression) and using explicit variable and class names if desired. Jupyter notebooks also render these plots inline automatically:

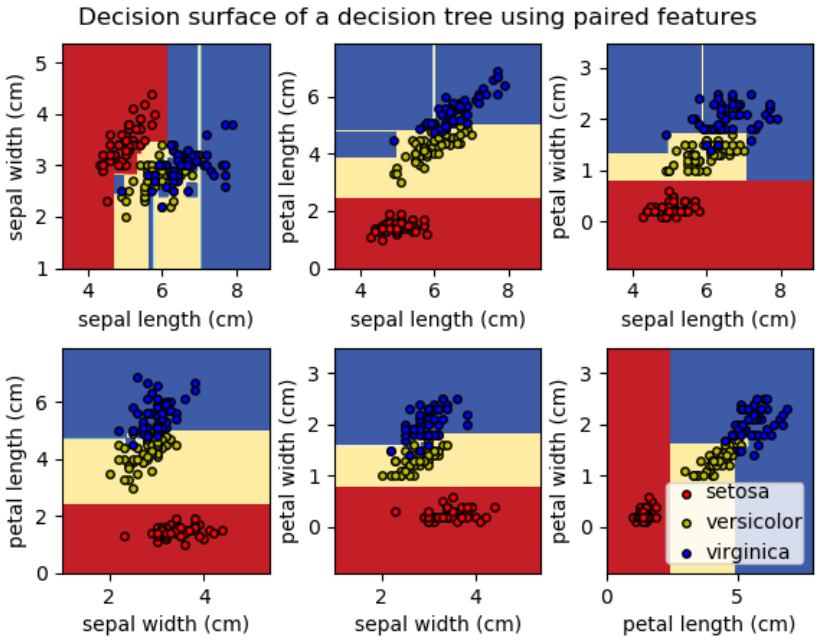
```
>>> dot_data = tree.export_graphviz(clf, out_file=None,
...                                 feature_names=iris.feature_names,
...                                 class_names=iris.target_names,
...                                 filled=True, rounded=True,
...                                 special_characters=True)
>>> graph = graphviz.Source(dot_data)
>>> graph
```



Alternatively, the tree can also be exported in textual format with the function *export_text*. This method doesn't require the installation of external libraries and is more compact:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.tree import DecisionTreeClassifier
```

(continues on next page)



(continued from previous page)

```
>>> from sklearn.tree.export import export_text
>>> iris = load_iris()
>>> decision_tree = DecisionTreeClassifier(random_state=0, max_depth=2)
>>> decision_tree = decision_tree.fit(iris.data, iris.target)
>>> r = export_text(decision_tree, feature_names=iris['feature_names'])
>>> print(r)
|--- petal width (cm) <= 0.80
|   |--- class: 0
|--- petal width (cm) >  0.80
|   |--- petal width (cm) <= 1.75
|   |   |--- class: 1
|   |--- petal width (cm) >  1.75
|   |   |--- class: 2
<BLANKLINE>
```

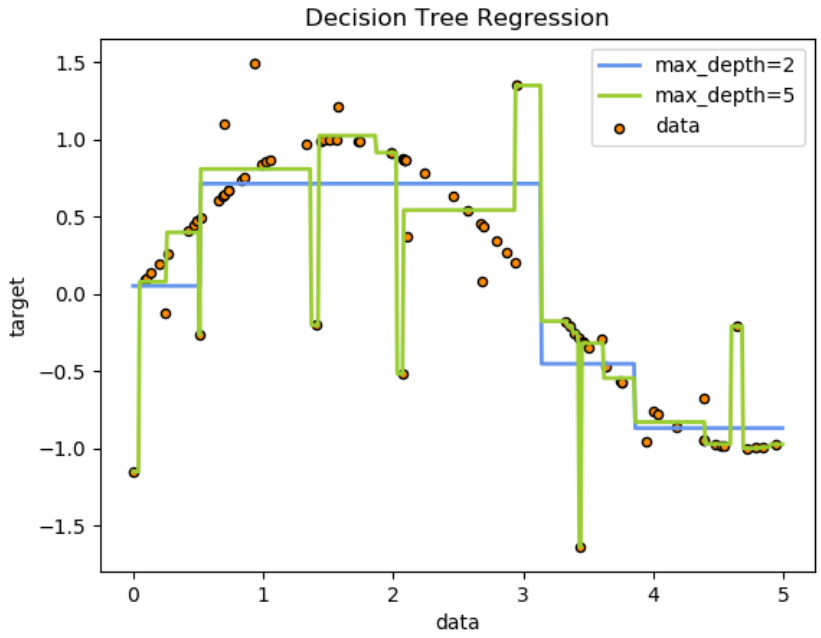
Examples:

- *Plot the decision surface of a decision tree on the iris dataset*
- *Understanding the decision tree structure*

Regression

Decision trees can also be applied to regression problems, using the `DecisionTreeRegressor` class.

As in the classification setting, the fit method will take as argument arrays X and y, only that in this case y is expected to have floating point values instead of integer values:




```
>>> from sklearn import tree
>>> X = [[0, 0], [2, 2]]
>>> y = [0.5, 2.5]
>>> clf = tree.DecisionTreeRegressor()
>>> clf = clf.fit(X, y)
>>> clf.predict([[1, 1]])
array([0.5])
```

Examples:

- *Decision Tree Regression*

Multi-output problems

A multi-output problem is a supervised learning problem with several outputs to predict, that is when Y is a 2d array of size `[n_samples, n_outputs]`.

When there is no correlation between the outputs, a very simple way to solve this kind of problem is to build n independent models, i.e. one for each output, and then to use those models to independently predict each one of the n outputs. However, because it is likely that the output values related to the same input are themselves correlated, an often better way is to build a single model capable of predicting simultaneously all n outputs. First, it requires lower training time since only a single estimator is built. Second, the generalization accuracy of the resulting estimator may often be increased.

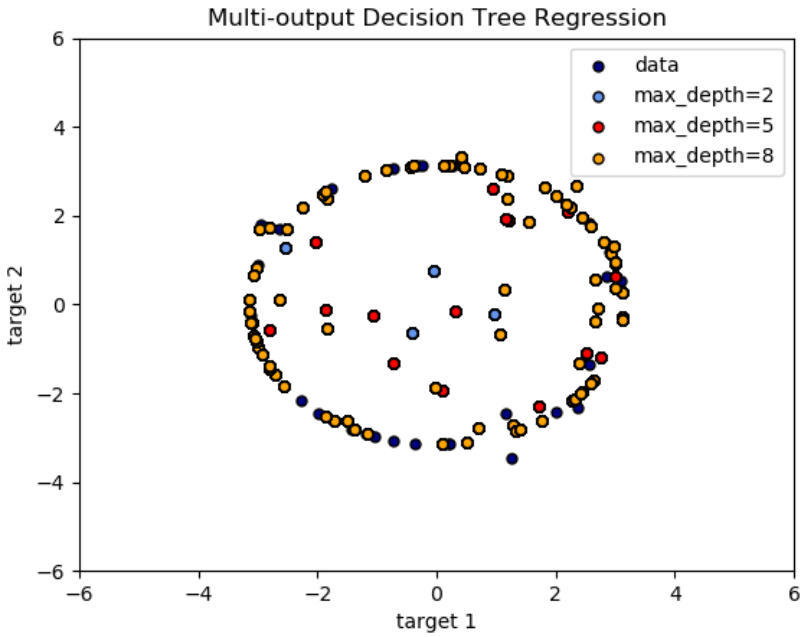
With regard to decision trees, this strategy can readily be used to support multi-output problems. This requires the following changes:

- Store n output values in leaves, instead of 1;
- Use splitting criteria that compute the average reduction across all n outputs.

This module offers support for multi-output problems by implementing this strategy in both `DecisionTreeClassifier` and `DecisionTreeRegressor`. If a decision tree is fit on an output array Y of size `[n_samples, n_outputs]` then the resulting estimator will:

- Output n_output values upon `predict`;
- Output a list of n_output arrays of class probabilities upon `predict_proba`.

The use of multi-output trees for regression is demonstrated in *Multi-output Decision Tree Regression*. In this example, the input X is a single real value and the outputs Y are the sine and cosine of X.



The use of multi-output trees for classification is demonstrated in *Face completion with a multi-output estimators*. In this example, the inputs X are the pixels of the upper half of faces and the outputs Y are the pixels of the lower half of those faces.

Examples:

- *Multi-output Decision Tree Regression*
- *Face completion with a multi-output estimators*

References:

- M. Dumont et al, Fast multi-class image annotation with random subwindows and multiple output randomized trees, International Conference on Computer Vision Theory and Applications 2009

Complexity

In general, the run time cost to construct a balanced binary tree is $O(n_{samples}n_{features}\log(n_{samples}))$ and query time $O(\log(n_{samples}))$. Although the tree construction algorithm attempts to generate balanced trees, they will not


Face completion with multi-output estimators
true faces
Extra trees
K-nn
Linear regression
Ridge

always be balanced. Assuming that the subtrees remain approximately balanced, the cost at each node consists of searching through $O(n_{features})$ to find the feature that offers the largest reduction in entropy. This has a cost of $O(n_{features}n_{samples}\log(n_{samples}))$ at each node, leading to a total cost over the entire trees (by summing the cost at each node) of $O(n_{features}n^2_{samples}\log(n_{samples}))$.

Tips on practical use

- Decision trees tend to overfit on data with a large number of features. Getting the right ratio of samples to number of features is important, since a tree with few samples in high dimensional space is very likely to overfit.
- Consider performing dimensionality reduction (*PCA*, *ICA*, or *Feature selection*) beforehand to give your tree a better chance of finding features that are discriminative.
- *Understanding the decision tree structure* will help in gaining more insights about how the decision tree makes predictions, which is important for understanding the important features in the data.
- Visualise your tree as you are training by using the `export` function. Use `max_depth=3` as an initial tree depth to get a feel for how the tree is fitting to your data, and then increase the depth.
- Remember that the number of samples required to populate the tree doubles for each additional level the tree grows to. Use `max_depth` to control the size of the tree to prevent overfitting.
- Use `min_samples_split` or `min_samples_leaf` to ensure that multiple samples inform every decision in the tree, by controlling which splits will be considered. A very small number will usually mean the tree will overfit, whereas a large number will prevent the tree from learning the data. Try `min_samples_leaf=5` as an initial value. If the sample size varies greatly, a float number can be used as percentage in these two parameters. While `min_samples_split` can create arbitrarily small leaves, `min_samples_leaf` guarantees that each leaf has a minimum size, avoiding low-variance, over-fit leaf nodes in regression problems. For classification with few classes, `min_samples_leaf=1` is often the best choice.
- Balance your dataset before training to prevent the tree from being biased toward the classes that are dominant. Class balancing can be done by sampling an equal number of samples from each class, or preferably by normalizing the sum of the sample weights (`sample_weight`) for each class to the same value. Also note that weight-based pre-pruning criteria, such as `min_weight_fraction_leaf`, will then be less biased toward dominant classes than criteria that are not aware of the sample weights, like `min_samples_leaf`.
- If the samples are weighted, it will be easier to optimize the tree structure using weight-based pre-pruning criterion such as `min_weight_fraction_leaf`, which ensure that leaf nodes contain at least a fraction of the overall sum of the sample weights.
- All decision trees use `np.float32` arrays internally. If training data is not in this format, a copy of the dataset will be made.
- If the input matrix X is very sparse, it is recommended to convert to sparse `csc_matrix` before calling fit and sparse `csr_matrix` before calling predict. Training time can be orders of magnitude faster for a sparse matrix input compared to a dense matrix when features have zero values in most of the samples.

Tree algorithms: ID3, C4.5, C5.0 and CART

What are all the various decision tree algorithms and how do they differ from each other? Which one is implemented in scikit-learn?

ID3 (Iterative Dichotomiser 3) was developed in 1986 by Ross Quinlan. The algorithm creates a multiway tree, finding for each node (i.e. in a greedy manner) the categorical feature that will yield the largest information gain for categorical targets. Trees are grown to their maximum size and then a pruning step is usually applied to improve the ability of the tree to generalise to unseen data.

C4.5 is the successor to ID3 and removed the restriction that features must be categorical by dynamically defining a discrete attribute (based on numerical variables) that partitions the continuous attribute value into a discrete set of intervals. C4.5 converts the trained trees (i.e. the output of the ID3 algorithm) into sets of if-then rules. These accuracy of each rule is then evaluated to determine the order in which they should be applied. Pruning is done by removing a rule's precondition if the accuracy of the rule improves without it.

C5.0 is Quinlan's latest version release under a proprietary license. It uses less memory and builds smaller rulesets than C4.5 while being more accurate.

CART (Classification and Regression Trees) is very similar to C4.5, but it differs in that it supports numerical target variables (regression) and does not compute rule sets. CART constructs binary trees using the feature and threshold that yield the largest information gain at each node.

scikit-learn uses an optimised version of the CART algorithm; however, scikit-learn implementation does not support categorical variables for now.

Mathematical formulation

Given training vectors $x_i \in R^n$, i=1,..., l and a label vector $y \in R^l$, a decision tree recursively partitions the space such that the samples with the same labels are grouped together.

Let the data at node m be represented by Q. For each candidate split $\theta = (j, t_m)$ consisting of a feature j and threshold t_m, partition the data into $Q_{left}(\theta)$ and $Q_{right}(\theta)$ subsets

$$Q_{left}(\theta) = (x, y)|x_j <= t_m$$
$$Q_{right}(\theta) = Q \setminus Q_{left}(\theta)$$

The impurity at m is computed using an impurity function $H()$, the choice of which depends on the task being solved (classification or regression)

$$G(Q, \theta) = \frac{n_{left}}{N_m} H(Q_{left}(\theta)) + \frac{n_{right}}{N_m} H(Q_{right}(\theta))$$

Select the parameters that minimises the impurity

$$\theta^* = \operatorname{argmin}_\theta G(Q, \theta)$$

Recurse for subsets $Q_{left}(\theta^*)$ and $Q_{right}(\theta^*)$ until the maximum allowable depth is reached, $N_m < \min_{samples}$ or $N_m = 1$.

Classification criteria

If a target is a classification outcome taking on values 0,1,...,K-1, for node m, representing a region R_m with N_m observations, let

$$p_{mk} = 1/N_m \sum_{x_i \in R_m} I(y_i = k)$$

be the proportion of class k observations in node m

Common measures of impurity are Gini

$$H(X_m) = \sum_k p_{mk}(1 - p_{mk})$$

Entropy

$$H(X_m) = -\sum_k p_{mk} \log(p_{mk})$$

and Misclassification

$$H(X_m) = 1 - \max(p_{mk})$$

where X_m is the training data in node m

Regression criteria

If the target is a continuous value, then for node m, representing a region R_m with N_m observations, common criteria to minimise as for determining locations for future splits are Mean Squared Error, which minimizes the L2 error using mean values at terminal nodes, and Mean Absolute Error, which minimizes the L1 error using median values at terminal nodes.

Mean Squared Error:

$$\bar{y}_m = \frac{1}{N_m} \sum_{i \in N_m} y_i$$

$$H(X_m) = \frac{1}{N_m} \sum_{i \in N_m} (y_i - \bar{y}_m)^2$$

Mean Absolute Error:

$$\bar{y}_m = \frac{1}{N_m} \sum_{i \in N_m} y_i$$

$$H(X_m) = \frac{1}{N_m} \sum_{i \in N_m} |y_i - \bar{y}_m|$$

where X_m is the training data in node m

Minimal Cost-Complexity Pruning

Minimal cost-complexity pruning is an algorithm used to prune a tree to avoid over-fitting, described in Chapter 3 of [BRE]. This algorithm is parameterized by $\alpha \geq 0$ known as the complexity parameter. The complexity parameter is used to define the cost-complexity measure, $R_\alpha(T)$ of a given tree T:

$$R_\alpha(T) = R(T) + \alpha|T|$$

where $|T|$ is the number of terminal nodes in T and $R(T)$ is traditionally defined as the total misclassification rate of the terminal nodes. Alternatively, scikit-learn uses the total sample weighted impurity of the terminal nodes for $R(T)$. As shown above, the impurity of a node depends on the criterion. Minimal cost-complexity pruning finds the subtree of T that minimizes $R_\alpha(T)$.

The cost complexity measure of a single node is $R_\alpha(t) = R(t) + \alpha$. The branch, T_t, is defined to be a tree where node t is its root. In general, the impurity of a node is greater than the sum of impurities of its terminal nodes, $R(T_t) < R(t)$. However, the cost complexity measure of a node, t, and its branch, T_t, can be equal depending on α. We define the effective α of a node to be the value where they are equal, $R_\alpha(T_t) = R_\alpha(t)$ or $\alpha_{eff}(t) = \frac{R(t) - R(T_t)}{|T| - 1}$. A non-terminal node with the smallest value of α_{eff} is the weakest link and will be pruned. This process stops when the pruned tree's minimal α_{eff} is greater than the `ccp_alpha` parameter.

Examples:

- *Post pruning decision trees with cost complexity pruning*

References:

- https://en.wikipedia.org/wiki/Decision_tree_learning
- https://en.wikipedia.org/wiki/Predictive_analytics
- J.R. Quinlan. C4. 5: programs for machine learning. Morgan Kaufmann, 1993.
- T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning, Springer, 2009.

4.1.11 Ensemble methods

The goal of **ensemble methods** is to combine the predictions of several base estimators built with a given learning algorithm in order to improve generalizability / robustness over a single estimator.

Two families of ensemble methods are usually distinguished:

- In **averaging methods**, the driving principle is to build several estimators independently and then to average their predictions. On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced.

  Examples: *Bagging methods*, *Forests of randomized trees*, ...

- By contrast, in **boosting methods**, base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble.

  Examples: *AdaBoost*, *Gradient Tree Boosting*, ...

Bagging meta-estimator

In ensemble algorithms, bagging methods form a class of algorithms which build several instances of a black-box estimator on random subsets of the original training set and then aggregate their individual predictions to form a final prediction. These methods are used as a way to reduce the variance of a base estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it. In many cases, bagging methods constitute a very simple way to improve with respect to a single model, without making it necessary to adapt the underlying base algorithm. As they provide a way to reduce overfitting, bagging methods work best with strong and complex models (e.g., fully developed decision trees), in contrast with boosting methods which usually work best with weak models (e.g., shallow decision trees).

Bagging methods come in many flavours but mostly differ from each other by the way they draw random subsets of the training set:

- When random subsets of the dataset are drawn as random subsets of the samples, then this algorithm is known as Pasting [B1999].
- When samples are drawn with replacement, then the method is known as Bagging [B1996].
- When random subsets of the dataset are drawn as random subsets of the features, then the method is known as Random Subspaces [H1998].
- Finally, when base estimators are built on subsets of both samples and features, then the method is known as Random Patches [LG2012].

In scikit-learn, bagging methods are offered as a unified `BaggingClassifier` meta-estimator (resp. `BaggingRegressor`), taking as input a user-specified base estimator along with parameters specifying the strategy to draw random subsets. In particular, `max_samples` and `max_features` control the size of the subsets (in terms of samples and features), while `bootstrap` and `bootstrap_features` control whether samples and features

are drawn with or without replacement. When using a subset of the available samples the generalization accuracy can be estimated with the out-of-bag samples by setting `oob_score=True`. As an example, the snippet below illustrates how to instantiate a bagging ensemble of `KNeighborsClassifier` base estimators, each built on random subsets of 50% of the samples and 50% of the features.

```
>>> from sklearn.ensemble import BaggingClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> bagging = BaggingClassifier(KNeighborsClassifier(),
...                             max_samples=0.5, max_features=0.5)
```

Examples:

- *Single estimator versus bagging: bias-variance decomposition*

References

Forests of randomized trees

The *sklearn.ensemble* module includes two averaging algorithms based on randomized *decision trees*: the RandomForest algorithm and the Extra-Trees method. Both algorithms are perturb-and-combine techniques [B1998] specifically designed for trees. This means a diverse set of classifiers is created by introducing randomness in the classifier construction. The prediction of the ensemble is given as the averaged prediction of the individual classifiers.

As other classifiers, forest classifiers have to be fitted with two arrays: a sparse or dense array X of size `[n_samples, n_features]` holding the training samples, and an array Y of size `[n_samples]` holding the target values (class labels) for the training samples:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = RandomForestClassifier(n_estimators=10)
>>> clf = clf.fit(X, Y)
```

Like *decision trees*, forests of trees also extend to *multi-output problems* (if Y is an array of size `[n_samples, n_outputs]`).

Random Forests

In random forests (see *RandomForestClassifier* and *RandomForestRegressor* classes), each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set.

Furthermore, when splitting each node during the construction of a tree, the best split is found either from all input features or a random subset of size `max_features`. (See the *parameter tuning guidelines* for more details).

The purpose of these two sources of randomness is to decrease the variance of the forest estimator. Indeed, individual decision trees typically exhibit high variance and tend to overfit. The injected randomness in forests yield decision trees with somewhat decoupled prediction errors. By taking an average of those predictions, some errors can cancel out. Random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias. In practice the variance reduction is often significant hence yielding an overall better model.

In contrast to the original publication [B2001], the scikit-learn implementation combines classifiers by averaging their probabilistic prediction, instead of letting each classifier vote for a single class.

Extremely Randomized Trees

In extremely randomized trees (see *ExtraTreesClassifier* and *ExtraTreesRegressor* classes), randomness goes one step further in the way splits are computed. As in random forests, a random subset of candidate features is used, but instead of looking for the most discriminative thresholds, thresholds are drawn at random for each candidate feature and the best of these randomly-generated thresholds is picked as the splitting rule. This usually allows to reduce the variance of the model a bit more, at the expense of a slightly greater increase in bias:

```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.datasets import make_blobs
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.tree import DecisionTreeClassifier

>>> X, y = make_blobs(n_samples=10000, n_features=10, centers=100,
...     random_state=0)

>>> clf = DecisionTreeClassifier(max_depth=None, min_samples_split=2,
...     random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
0.98...

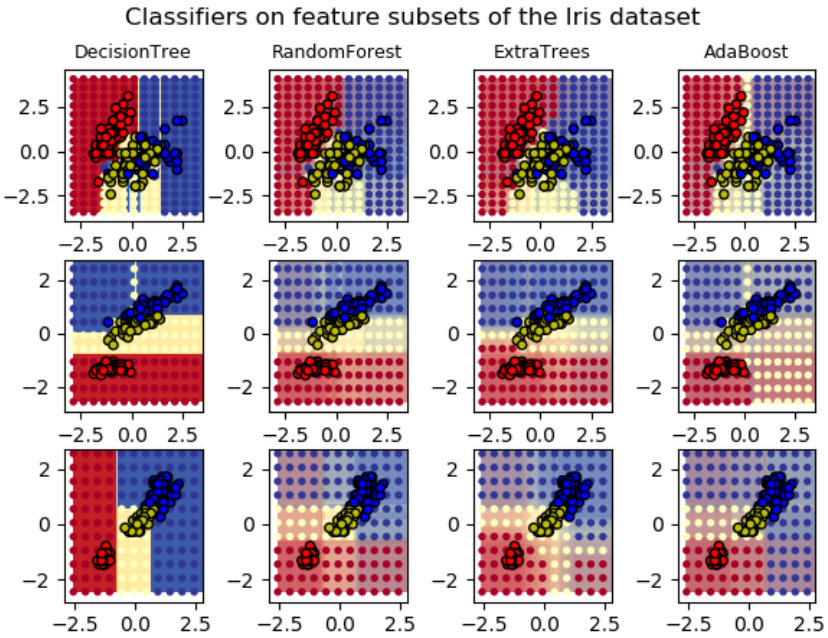
>>> clf = RandomForestClassifier(n_estimators=10, max_depth=None,
...     min_samples_split=2, random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
0.999...

>>> clf = ExtraTreesClassifier(n_estimators=10, max_depth=None,
...     min_samples_split=2, random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean() > 0.999
True
```

Parameters

The main parameters to adjust when using these methods is `n_estimators` and `max_features`. The former is the number of trees in the forest. The larger the better, but also the longer it will take to compute. In addition, note that results will stop getting significantly better beyond a critical number of trees. The latter is the size of the random subsets of features to consider when splitting a node. The lower the greater the reduction of variance, but also the greater the increase in bias. Empirical good default values are `max_features=None` (always considering all features instead of a random subset) for regression problems, and `max_features="sqrt"` (using a random subset of size `sqrt(n_features)`) for classification tasks (where `n_features` is the number of features in the data). Good results are often achieved when setting `max_depth=None` in combination with `min_samples_split=2` (i.e., when fully developing the trees). Bear in mind though that these values are usually not optimal, and might result in models that consume a lot of RAM. The best parameter values should always be cross-validated. In addition, note that in random forests, bootstrap samples are used by default (`bootstrap=True`) while the default strategy for extra-trees is to use the whole dataset (`bootstrap=False`). When using bootstrap sampling the generalization accuracy can be estimated on the left out or out-of-bag samples. This can be enabled by setting `oob_score=True`.

Note: The size of the model with the default parameters is $O(M * N * log(N))$, where M is the number of trees and N is the number of samples. In order to reduce the size of the model, you can change these parameters:



`min_samples_split`, `max_leaf_nodes`, `max_depth` and `min_samples_leaf`.

Parallelization

Finally, this module also features the parallel construction of the trees and the parallel computation of the predictions through the `n_jobs` parameter. If `n_jobs=k` then computations are partitioned into `k` jobs, and run on `k` cores of the machine. If `n_jobs=-1` then all cores available on the machine are used. Note that because of inter-process communication overhead, the speedup might not be linear (i.e., using `k` jobs will unfortunately not be `k` times as fast). Significant speedup can still be achieved though when building a large number of trees, or when building a single tree requires a fair amount of time (e.g., on large datasets).

Examples:

- *Plot the decision surfaces of ensembles of trees on the iris dataset*
- *Pixel importances with a parallel forest of trees*
- *Face completion with a multi-output estimators*

References

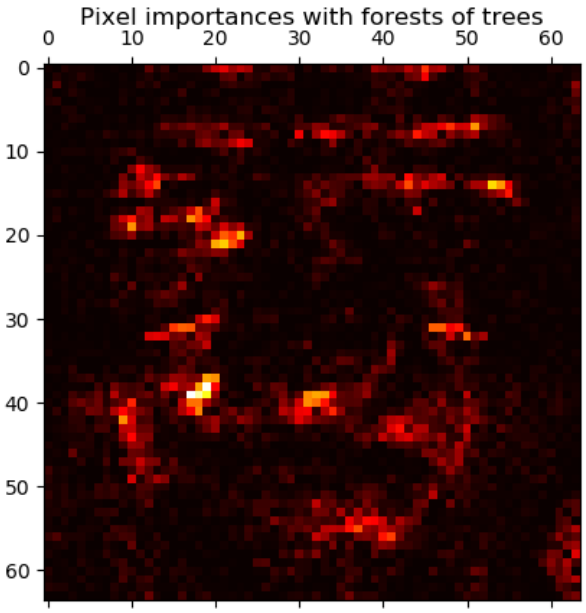
- P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.

Feature importance evaluation

The relative rank (i.e. depth) of a feature used as a decision node in a tree can be used to assess the relative importance of that feature with respect to the predictability of the target variable. Features used at the top of the tree contribute to the final prediction decision of a larger fraction of the input samples. The **expected fraction of the samples** they contribute to can thus be used as an estimate of the **relative importance of the features**. In scikit-learn, the fraction of samples a feature contributes to is combined with the decrease in impurity from splitting them to create a normalized estimate of the predictive power of that feature.

By **averaging** the estimates of predictive ability over several randomized trees one can **reduce the variance** of such an estimate and use it for feature selection. This is known as the mean decrease in impurity, or MDI. Refer to [L2014] for more information on MDI and feature importance evaluation with Random Forests.

The following example shows a color-coded representation of the relative importances of each individual pixel for a face recognition task using a `ExtraTreesClassifier` model.



In practice those estimates are stored as an attribute named `feature_importances_` on the fitted model. This is an array with shape `(n_features,)` whose values are positive and sum to 1.0. The higher the value, the more important is the contribution of the matching feature to the prediction function.

Examples:

- *Pixel importances with a parallel forest of trees*
- *Feature importances with forests of trees*

References

Totally Random Trees Embedding

`RandomTreesEmbedding` implements an unsupervised transformation of the data. Using a forest of completely random trees, `RandomTreesEmbedding` encodes the data by the indices of the leaves a data point ends up in. This index is then encoded in a one-of-K manner, leading to a high dimensional, sparse binary coding. This coding can be computed very efficiently and can then be used as a basis for other learning tasks. The size and sparsity of the code can be influenced by choosing the number of trees and the maximum depth per tree. For each tree in the ensemble, the coding contains one entry of one. The size of the coding is at most `n_estimators * 2 ** max_depth`, the maximum number of leaves in the forest.

As neighboring data points are more likely to lie within the same leaf of a tree, the transformation performs an implicit, non-parametric density estimation.

Examples:

- *Hashing feature transformation using Totally Random Trees*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...* compares non-linear dimensionality reduction techniques on handwritten digits.
- *Feature transformations with ensembles of trees* compares supervised and unsupervised tree based feature transformations.

See also:

Manifold learning techniques can also be useful to derive non-linear representations of feature space, also these approaches focus also on dimensionality reduction.

AdaBoost

The module `sklearn.ensemble` includes the popular boosting algorithm AdaBoost, introduced in 1995 by Freund and Schapire [FS1995].

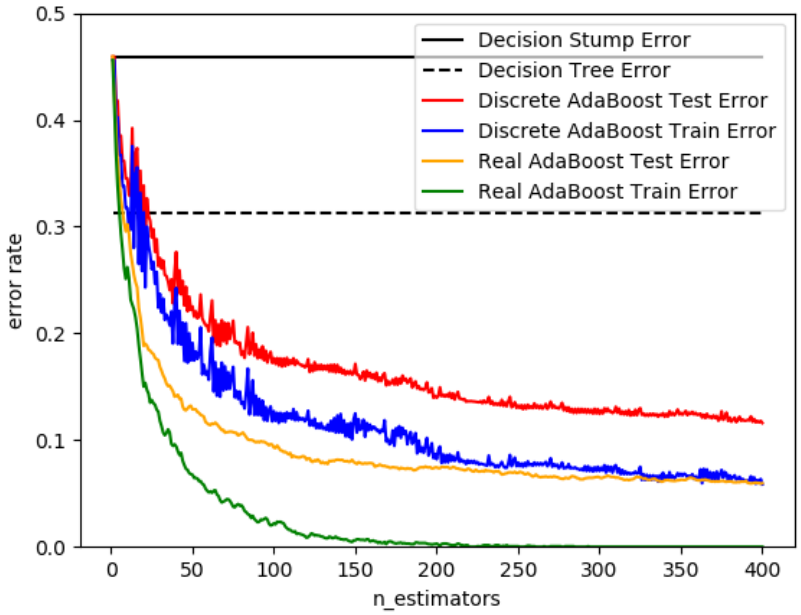
The core principle of AdaBoost is to fit a sequence of weak learners (i.e., models that are only slightly better than random guessing, such as small decision trees) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction. The data modifications at each so-called boosting iteration consist of applying weights w_1, w_2, ..., w_N to each of the training samples. Initially, those weights are all set to $w_i = 1/N$, so that the first step simply trains a weak learner on the original data. For each successive iteration, the sample weights are individually modified and the learning algorithm is reapplied to the reweighted data. At a given step, those training examples that were incorrectly predicted by the boosted model induced at the previous step have their weights increased, whereas the weights are decreased for those that were predicted correctly. As iterations proceed, examples that are difficult to predict receive ever-increasing influence. Each subsequent weak learner is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence [HTF].

AdaBoost can be used both for classification and regression problems:

- For multi-class classification, `AdaBoostClassifier` implements AdaBoost-SAMME and AdaBoost-SAMME.R [ZZRH2009].
- For regression, `AdaBoostRegressor` implements AdaBoost.R2 [D1997].

Usage

The following example shows how to fit an AdaBoost classifier with 100 weak learners:




```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.datasets import load_iris
>>> from sklearn.ensemble import AdaBoostClassifier

>>> X, y = load_iris(return_X_y=True)
>>> clf = AdaBoostClassifier(n_estimators=100)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
0.9...
```

The number of weak learners is controlled by the parameter `n_estimators`. The `learning_rate` parameter controls the contribution of the weak learners in the final combination. By default, weak learners are decision stumps. Different weak learners can be specified through the `base_estimator` parameter. The main parameters to tune to obtain good results are `n_estimators` and the complexity of the base estimators (e.g., its depth `max_depth` or minimum required number of samples to consider a split `min_samples_split`).

Examples:

- *Discrete versus Real AdaBoost* compares the classification error of a decision stump, decision tree, and a boosted decision stump using AdaBoost-SAMME and AdaBoost-SAMME.R.
- *Multi-class AdaBoosted Decision Trees* shows the performance of AdaBoost-SAMME and AdaBoost-SAMME.R on a multi-class problem.
- *Two-class AdaBoost* shows the decision boundary and decision function values for a non-linearly separable two-class problem using AdaBoost-SAMME.
- *Decision Tree Regression with AdaBoost* demonstrates regression with the AdaBoost.R2 algorithm.

References

Gradient Tree Boosting

Gradient Tree Boosting or Gradient Boosted Decision Trees (GBDT) is a generalization of boosting to arbitrary differentiable loss functions. GBDT is an accurate and effective off-the-shelf procedure that can be used for both regression and classification problems in a variety of areas including Web search ranking and ecology.

The module *sklearn.ensemble* provides methods for both classification and regression via gradient boosted decision trees.

Note: Scikit-learn 0.21 introduces two new experimental implementations of gradient boosting trees, namely *HistGradientBoostingClassifier* and *HistGradientBoostingRegressor*, inspired by LightGBM (See [LightGBM]).

These histogram-based estimators can be **orders of magnitude faster** than *GradientBoostingClassifier* and *GradientBoostingRegressor* when the number of samples is larger than tens of thousands of samples.

They also have built-in support for missing values, which avoids the need for an imputer.

These estimators are described in more detail below in *Histogram-Based Gradient Boosting*.

The following guide focuses on *GradientBoostingClassifier* and *GradientBoostingRegressor*, which might be preferred for small sample sizes since binning may lead to split points that are too approximate in this setting.

Classification

GradientBoostingClassifier supports both binary and multi-class classification. The following example shows how to fit a gradient boosting classifier with 100 decision stumps as weak learners:

```
>>> from sklearn.datasets import make_hastie_10_2
>>> from sklearn.ensemble import GradientBoostingClassifier

>>> X, y = make_hastie_10_2(random_state=0)
>>> X_train, X_test = X[:2000], X[2000:]
>>> y_train, y_test = y[:2000], y[2000:]

>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.913...
```

The number of weak learners (i.e. regression trees) is controlled by the parameter `n_estimators`; *The size of each tree* can be controlled either by setting the tree depth via `max_depth` or by setting the number of leaf nodes via `max_leaf_nodes`. The `learning_rate` is a hyper-parameter in the range (0.0, 1.0] that controls overfitting via *shrinkage* .

Note: Classification with more than 2 classes requires the induction of `n_classes` regression trees at each iteration, thus, the total number of induced trees equals `n_classes * n_estimators`. For datasets with a large number of classes we strongly recommend to use *HistGradientBoostingClassifier* as an alternative to *GradientBoostingClassifier* .

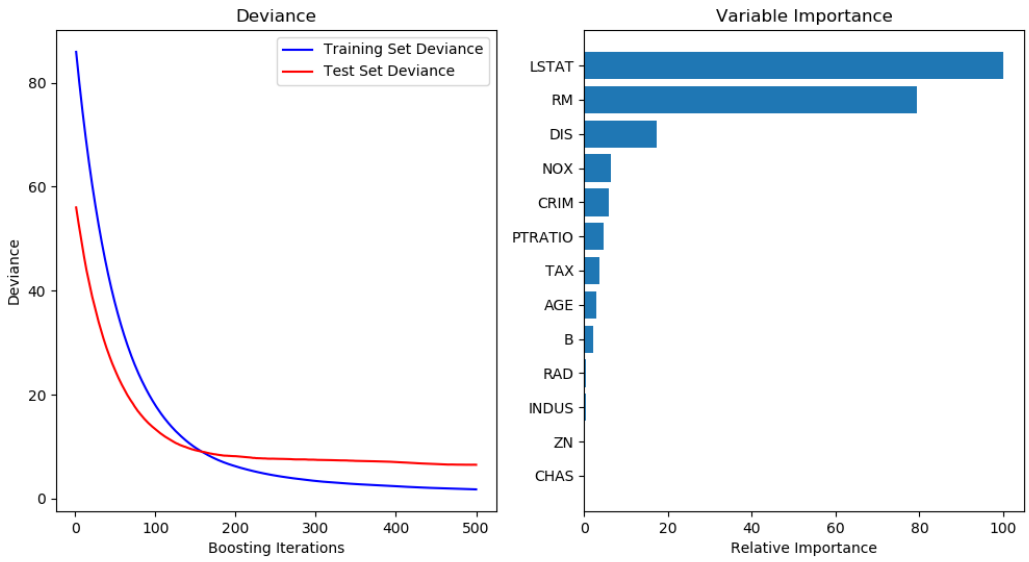
Regression

GradientBoostingRegressor supports a number of *different loss functions* for regression which can be specified via the argument `loss`; the default loss function for regression is least squares (`'ls'`).

```
>>> import numpy as np
>>> from sklearn.metrics import mean_squared_error
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.ensemble import GradientBoostingRegressor

>>> X, y = make_friedman1(n_samples=1200, random_state=0, noise=1.0)
>>> X_train, X_test = X[:200], X[200:]
>>> y_train, y_test = y[:200], y[200:]
>>> est = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1,
...     max_depth=1, random_state=0, loss='ls').fit(X_train, y_train)
>>> mean_squared_error(y_test, est.predict(X_test))
5.00...
```

The figure below shows the results of applying *GradientBoostingRegressor* with least squares loss and 500 base learners to the Boston house price dataset (*sklearn.datasets.load_boston*). The plot on the left shows the train and test error at each iteration. The train error at each iteration is stored in the `train_score_` attribute of the gradient boosting model. The test error at each iterations can be obtained via the *staged_predict* method which returns a generator that yields the predictions at each stage. Plots like these can be used to determine the optimal number of trees (i.e. `n_estimators`) by early stopping. The plot on the right shows the feature importances which can be obtained via the `feature_importances_` property.



Examples:

- *Gradient Boosting regression*
- *Gradient Boosting Out-of-Bag estimates*

Fitting additional weak-learners

Both *GradientBoostingRegressor* and *GradientBoostingClassifier* support `warm_start=True` which allows you to add more estimators to an already fitted model.

```
>>> _ = est.set_params(n_estimators=200, warm_start=True)  # set warm_start and new␣
↪nr of trees
>>> _ = est.fit(X_train, y_train) # fit additional 100 trees to est
>>> mean_squared_error(y_test, est.predict(X_test))
3.84...
```

Controlling the tree size

The size of the regression tree base learners defines the level of variable interactions that can be captured by the gradient boosting model. In general, a tree of depth `h` can capture interactions of order `h` . There are two ways in which the size of the individual regression trees can be controlled.

If you specify `max_depth=h` then complete binary trees of depth `h` will be grown. Such trees will have (at most) `2**h` leaf nodes and `2**h - 1` split nodes.

Alternatively, you can control the tree size by specifying the number of leaf nodes via the parameter `max_leaf_nodes`. In this case, trees will be grown using best-first search where nodes with the highest improvement in impurity will be expanded first. A tree with `max_leaf_nodes=k` has `k - 1` split nodes and thus can model interactions of up to order `max_leaf_nodes - 1` .

We found that `max_leaf_nodes=k` gives comparable results to `max_depth=k-1` but is significantly faster to train at the expense of a slightly higher training error. The parameter `max_leaf_nodes` corresponds to the variable `J` in the chapter on gradient boosting in [F2001] and is related to the parameter `interaction.depth` in R's gbm package where `max_leaf_nodes == interaction.depth + 1` .

Mathematical formulation

GBRT considers additive models of the following form:

$$F(x) = \sum_{m=1}^{M} \gamma_m h_m(x)$$

where $h_m(x)$ are the basis functions which are usually called *weak learners* in the context of boosting. Gradient Tree Boosting uses *decision trees* of fixed size as weak learners. Decision trees have a number of abilities that make them valuable for boosting, namely the ability to handle data of mixed type and the ability to model complex functions.

Similar to other boosting algorithms, GBRT builds the additive model in a greedy fashion:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x),$$

where the newly added tree h_m tries to minimize the loss L, given the previous ensemble F_{m-1}:

$$h_m = \arg\min_{h} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + h(x_i)).$$

The initial model F_0 is problem specific, for least-squares regression one usually chooses the mean of the target values.

Note: The initial model can also be specified via the `init` argument. The passed object has to implement `fit` and `predict`.

Gradient Boosting attempts to solve this minimization problem numerically via steepest descent: The steepest descent direction is the negative gradient of the loss function evaluated at the current model F_{m-1} which can be calculated for any differentiable loss function:

$$F_m(x) = F_{m-1}(x) - \gamma_m \sum_{i=1}^{n} \nabla_F L(y_i, F_{m-1}(x_i))$$

Where the step length γ_m is chosen using line search:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) - \gamma \frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)})$$

The algorithms for regression and classification only differ in the concrete loss function used.

Loss Functions

The following loss functions are supported and can be specified using the parameter `loss`:

- Regression
  - Least squares (`'ls'`): The natural choice for regression due to its superior computational properties. The initial model is given by the mean of the target values.
  - Least absolute deviation (`'lad'`): A robust loss function for regression. The initial model is given by the median of the target values.
  - Huber (`'huber'`): Another robust loss function that combines least squares and least absolute deviation; use `alpha` to control the sensitivity with regards to outliers (see [F2001] for more details).
  - Quantile (`'quantile'`): A loss function for quantile regression. Use `0 < alpha < 1` to specify the quantile. This loss function can be used to create prediction intervals (see *Prediction Intervals for Gradient Boosting Regression*).
- Classification
  - Binomial deviance (`'deviance'`): The negative binomial log-likelihood loss function for binary classification (provides probability estimates). The initial model is given by the log odds-ratio.
  - Multinomial deviance (`'deviance'`): The negative multinomial log-likelihood loss function for multi-class classification with `n_classes` mutually exclusive classes. It provides probability estimates. The initial model is given by the prior probability of each class. At each iteration `n_classes` regression trees have to be constructed which makes GBRT rather inefficient for data sets with a large number of classes.
  - Exponential loss (`'exponential'`): The same loss function as `AdaBoostClassifier`. Less robust to mislabeled examples than `'deviance'`; can only be used for binary classification.

Regularization

Shrinkage

[F2001] proposed a simple regularization strategy that scales the contribution of each weak learner by a factor ν:

$$F_m(x) = F_{m-1}(x) + \nu\gamma_m h_m(x)$$

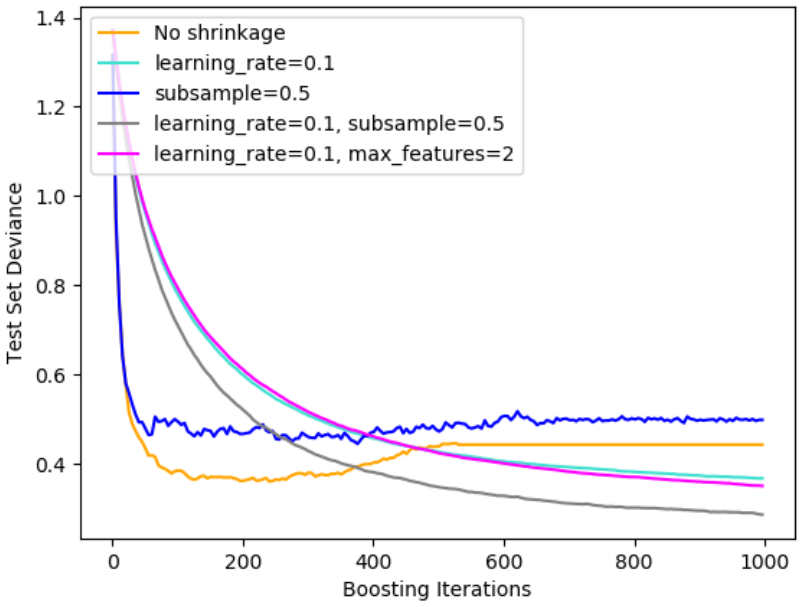
The parameter ν is also called the **learning rate** because it scales the step length the gradient descent procedure; it can be set via the `learning_rate` parameter.

The parameter `learning_rate` strongly interacts with the parameter `n_estimators`, the number of weak learners to fit. Smaller values of `learning_rate` require larger numbers of weak learners to maintain a constant training error. Empirical evidence suggests that small values of `learning_rate` favor better test error. [HTF] recommend to set the learning rate to a small constant (e.g. `learning_rate <= 0.1`) and choose `n_estimators` by early stopping. For a more detailed discussion of the interaction between `learning_rate` and `n_estimators` see [R2007].

Subsampling

[F1999] proposed stochastic gradient boosting, which combines gradient boosting with bootstrap averaging (bagging). At each iteration the base classifier is trained on a fraction `subsample` of the available training data. The subsample is drawn without replacement. A typical value of `subsample` is 0.5.

The figure below illustrates the effect of shrinkage and subsampling on the goodness-of-fit of the model. We can clearly see that shrinkage outperforms no-shrinkage. Subsampling with shrinkage can further increase the accuracy of the model. Subsampling without shrinkage, on the other hand, does poorly.



Another strategy to reduce the variance is by subsampling the features analogous to the random splits in `RandomForestClassifier`. The number of subsampled features can be controlled via the `max_features` parameter.

Note: Using a small `max_features` value can significantly decrease the runtime.

Stochastic gradient boosting allows to compute out-of-bag estimates of the test deviance by computing the improvement in deviance on the examples that are not included in the bootstrap sample (i.e. the out-of-bag examples). The improvements are stored in the attribute `oob_improvement_`. `oob_improvement_[i]` holds the improvement in terms of the loss on the OOB samples if you add the i-th stage to the current predictions. Out-of-bag estimates can be used for model selection, for example to determine the optimal number of iterations. OOB estimates are usually very pessimistic thus we recommend to use cross-validation instead and only use OOB if cross-validation is too time consuming.

Examples:

- *Gradient Boosting regularization*
- *Gradient Boosting Out-of-Bag estimates*
- *OOB Errors for Random Forests*

Interpretation

Individual decision trees can be interpreted easily by simply visualizing the tree structure. Gradient boosting models, however, comprise hundreds of regression trees thus they cannot be easily interpreted by visual inspection of the individual trees. Fortunately, a number of techniques have been proposed to summarize and interpret gradient boosting models.

Feature importance

Often features do not contribute equally to predict the target response; in many situations the majority of the features are in fact irrelevant. When interpreting a model, the first question usually is: what are those important features and how do they contributing in predicting the target response?

Individual decision trees intrinsically perform feature selection by selecting appropriate split points. This information can be used to measure the importance of each feature; the basic idea is: the more often a feature is used in the split points of a tree the more important that feature is. This notion of importance can be extended to decision tree ensembles by simply averaging the feature importance of each tree (see *Feature importance evaluation* for more details).

The feature importance scores of a fit gradient boosting model can be accessed via the `feature_importances_` property:

```
>>> from sklearn.datasets import make_hastie_10_2
>>> from sklearn.ensemble import GradientBoostingClassifier

>>> X, y = make_hastie_10_2(random_state=0)
>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X, y)
>>> clf.feature_importances_
array([0.10..., 0.10..., 0.11..., ...
```

Examples:

- *Gradient Boosting regression*

Histogram-Based Gradient Boosting

Scikit-learn 0.21 introduces two new experimental implementations of gradient boosting trees, namely `HistGradientBoostingClassifier` and `HistGradientBoostingRegressor`, inspired by LightGBM (See [LightGBM]).

These histogram-based estimators can be **orders of magnitude faster** than `GradientBoostingClassifier` and `GradientBoostingRegressor` when the number of samples is larger than tens of thousands of samples.

They also have built-in support for missing values, which avoids the need for an imputer.

These fast estimators first bin the input samples `X` into integer-valued bins (typically 256 bins) which tremendously reduces the number of splitting points to consider, and allows the algorithm to leverage integer-based data structures (histograms) instead of relying on sorted continuous values when building the trees. The API of these estimators is slightly different, and some of the features from `GradientBoostingClassifier` and `GradientBoostingRegressor` are not yet supported: in particular sample weights, and some loss functions.

These estimators are still **experimental**: their predictions and their API might change without any deprecation cycle. To use them, you need to explicitly import `enable_hist_gradient_boosting`:

```
>>> # explicitly require this experimental feature
>>> from sklearn.experimental import enable_hist_gradient_boosting  # noqa
>>> # now you can import normally from ensemble
>>> from sklearn.ensemble import HistGradientBoostingClassifier
```

Examples:

- *Partial Dependence Plots*

Usage

Most of the parameters are unchanged from `GradientBoostingClassifier` and `GradientBoostingRegressor`. One exception is the `max_iter` parameter that replaces `n_estimators`, and controls the number of iterations of the boosting process:

```
>>> from sklearn.experimental import enable_hist_gradient_boosting
>>> from sklearn.ensemble import HistGradientBoostingClassifier
>>> from sklearn.datasets import make_hastie_10_2

>>> X, y = make_hastie_10_2(random_state=0)
>>> X_train, X_test = X[:2000], X[2000:]
>>> y_train, y_test = y[:2000], y[2000:]

>>> clf = HistGradientBoostingClassifier(max_iter=100).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.8965
```

Available losses for regression are 'least_squares' and 'least_absolute_deviation', which is less sensitive to outliers. For classification, 'binary_crossentropy' is used for binary classification and 'categorical_crossentropy' is used for multiclass classification. By default the loss is 'auto' and will select the appropriate loss depending on *y* passed to *fit*.

The size of the trees can be controlled through the `max_leaf_nodes`, `max_depth`, and `min_samples_leaf` parameters.

The number of bins used to bin the data is controlled with the `max_bins` parameter. Using less bins acts as a form of regularization. It is generally recommended to use as many bins as possible, which is the default.

The `l2_regularization` parameter is a regularizer on the loss function and corresponds to λ in equation (2) of [XGBoost].

The early-stopping behaviour is controlled via the `scoring`, `validation_fraction`, `n_iter_no_change`, and `tol` parameters. It is possible to early-stop using an arbitrary *scorer*, or just the training or validation loss. By default, early-stopping is performed using the default *scorer* of the estimator on a validation set but it is also possible to perform early-stopping based on the loss value, which is significantly faster.

Missing values support

HistGradientBoostingClassifier and *HistGradientBoostingRegressor* have built-in support for missing values (NaNs).

During training, the tree grower learns at each split point whether samples with missing values should go to the left or right child, based on the potential gain. When predicting, samples with missing values are assigned to the left or right child consequently:

```
>>> from sklearn.experimental import enable_hist_gradient_boosting  # noqa
>>> from sklearn.ensemble import HistGradientBoostingClassifier
>>> import numpy as np

>>> X = np.array([0, 1, 2, np.nan]).reshape(-1, 1)
>>> y = [0, 0, 1, 1]

>>> gbdt = HistGradientBoostingClassifier(min_samples_leaf=1).fit(X, y)
>>> gbdt.predict(X)
array([0, 0, 1, 1])
```

When the missingness pattern is predictive, the splits can be done on whether the feature value is missing or not:

```
>>> X = np.array([0, np.nan, 1, 2, np.nan]).reshape(-1, 1)
>>> y = [0, 1, 0, 0, 1]
>>> gbdt = HistGradientBoostingClassifier(min_samples_leaf=1,
...                                       max_depth=2,
...                                       learning_rate=1,
...                                       max_iter=1).fit(X, y)
>>> gbdt.predict(X)
array([0, 1, 0, 0, 1])
```

If no missing values were encountered for a given feature during training, then samples with missing values are mapped to whichever child has the most samples.

Low-level parallelism

HistGradientBoostingClassifier and *HistGradientBoostingRegressor* have implementations that use OpenMP for parallelization through Cython. For more details on how to control the number of threads, please refer to our *Parallelism* notes.

The following parts are parallelized:

- mapping samples from real values to integer-valued bins (finding the bin thresholds is however sequential)

- building histograms is parallelized over features
- finding the best split point at a node is parallelized over features
- during fit, mapping samples into the left and right children is parallelized over samples
- gradient and hessians computations are parallelized over samples
- predicting is parallelized over samples

Why it's faster

The bottleneck of a gradient boosting procedure is building the decision trees. Building a traditional decision tree (as in the other GBDTs *GradientBoostingClassifier* and *GradientBoostingRegressor*) requires sorting the samples at each node (for each feature). Sorting is needed so that the potential gain of a split point can be computed efficiently. Splitting a single node has thus a complexity of $\mathcal{O}(n_{\text{features}} \times n \log(n))$ where n is the number of samples at the node.

HistGradientBoostingClassifier and *HistGradientBoostingRegressor*, in contrast, do not require sorting the feature values and instead use a data-structure called a histogram, where the samples are implicitly ordered. Building a histogram has a $\mathcal{O}(n)$ complexity, so the node splitting procedure has a $\mathcal{O}(n_{\text{features}} \times n)$ complexity, much smaller than the previous one. In addition, instead of considering n split points, we here consider only `max_bins` split points, which is much smaller.

In order to build histograms, the input data `X` needs to be binned into integer-valued bins. This binning procedure does require sorting the feature values, but it only happens once at the very beginning of the boosting process (not at each node, like in *GradientBoostingClassifier* and *GradientBoostingRegressor*).

Finally, many parts of the implementation of *HistGradientBoostingClassifier* and *HistGradientBoostingRegressor* are parallelized.

References

Voting Classifier

The idea behind the *VotingClassifier* is to combine conceptually different machine learning classifiers and use a majority vote or the average predicted probabilities (soft vote) to predict the class labels. Such a classifier can be useful for a set of equally well performing model in order to balance out their individual weaknesses.

Majority Class Labels (Majority/Hard Voting)

In majority voting, the predicted class label for a particular sample is the class label that represents the majority (mode) of the class labels predicted by each individual classifier.

E.g., if the prediction for a given sample is

- classifier 1 -> class 1
- classifier 2 -> class 1
- classifier 3 -> class 2

the VotingClassifier (with `voting='hard'`) would classify the sample as "class 1" based on the majority class label.

In the cases of a tie, the *VotingClassifier* will select the class based on the ascending sort order. E.g., in the following scenario

- classifier 1 -> class 2
- classifier 2 -> class 1

the class label 1 will be assigned to the sample.

Usage

The following example shows how to fit the majority rule classifier:

```
>>> from sklearn import datasets
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.ensemble import VotingClassifier

>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, 1:3], iris.target

>>> clf1 = LogisticRegression(random_state=1)
>>> clf2 = RandomForestClassifier(n_estimators=50, random_state=1)
>>> clf3 = GaussianNB()

>>> eclf = VotingClassifier(
...     estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],
...     voting='hard')

>>> for clf, label in zip([clf1, clf2, clf3, eclf], ['Logistic Regression', 'Random␣
↪Forest', 'naive Bayes', 'Ensemble']):
...     scores = cross_val_score(clf, X, y, scoring='accuracy', cv=5)
...     print("Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(), scores.std(),␣
↪label))
Accuracy: 0.95 (+/- 0.04) [Logistic Regression]
Accuracy: 0.94 (+/- 0.04) [Random Forest]
Accuracy: 0.91 (+/- 0.04) [naive Bayes]
Accuracy: 0.95 (+/- 0.04) [Ensemble]
```

Weighted Average Probabilities (Soft Voting)

In contrast to majority voting (hard voting), soft voting returns the class label as argmax of the sum of predicted probabilities.

Specific weights can be assigned to each classifier via the `weights` parameter. When weights are provided, the predicted class probabilities for each classifier are collected, multiplied by the classifier weight, and averaged. The final class label is then derived from the class label with the highest average probability.

To illustrate this with a simple example, let's assume we have 3 classifiers and a 3-class classification problems where we assign equal weights to all classifiers: w1=1, w2=1, w3=1.

The weighted average probabilities for a sample would then be calculated as follows:

classifier	class 1	class 2	class 3
classifier 1	w1 * 0.2	w1 * 0.5	w1 * 0.3
classifier 2	w2 * 0.6	w2 * 0.3	w2 * 0.1
classifier 3	w3 * 0.3	w3 * 0.4	w3 * 0.3
weighted average	0.37	0.4	0.23

Here, the predicted class label is 2, since it has the highest average probability.

The following example illustrates how the decision regions may change when a soft *VotingClassifier* is used based on an linear Support Vector Machine, a Decision Tree, and a K-nearest neighbor classifier:

```
>>> from sklearn import datasets
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.svm import SVC
>>> from itertools import product
>>> from sklearn.ensemble import VotingClassifier

>>> # Loading some example data
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [0, 2]]
>>> y = iris.target

>>> # Training classifiers
>>> clf1 = DecisionTreeClassifier(max_depth=4)
>>> clf2 = KNeighborsClassifier(n_neighbors=7)
>>> clf3 = SVC(kernel='rbf', probability=True)
>>> eclf = VotingClassifier(estimators=[('dt', clf1), ('knn', clf2), ('svc', clf3)],
...                         voting='soft', weights=[2, 1, 2])

>>> clf1 = clf1.fit(X, y)
>>> clf2 = clf2.fit(X, y)
>>> clf3 = clf3.fit(X, y)
>>> eclf = eclf.fit(X, y)
```

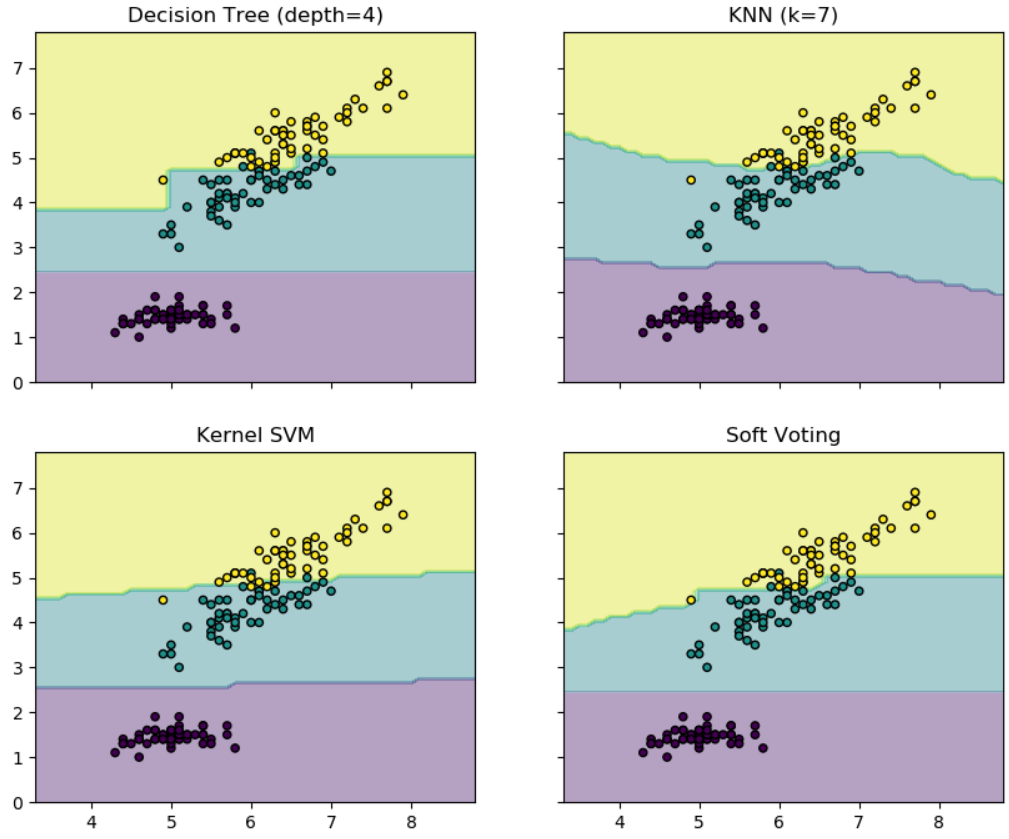
Using the `VotingClassifier` with `GridSearchCV`

The *VotingClassifier* can also be used together with *GridSearchCV* in order to tune the hyperparameters of the individual estimators:

```
>>> from sklearn.model_selection import GridSearchCV
>>> clf1 = LogisticRegression(random_state=1)
>>> clf2 = RandomForestClassifier(random_state=1)
>>> clf3 = GaussianNB()
>>> eclf = VotingClassifier(
...     estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],
...     voting='soft'
... )

>>> params = {'lr__C': [1.0, 100.0], 'rf__n_estimators': [20, 200]}

>>> grid = GridSearchCV(estimator=eclf, param_grid=params, cv=5)
>>> grid = grid.fit(iris.data, iris.target)
```


Decision Tree (depth=4)
KNN (k=7)
Kernel SVM
Soft Voting
0
1
2
3
4
5
6
7
4
5
6
7
8

Usage

In order to predict the class labels based on the predicted class-probabilities (scikit-learn estimators in the VotingClassifier must support `predict_proba` method):

```
>>> eclf = VotingClassifier(
...     estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],
...     voting='soft'
... )
```

Optionally, weights can be provided for the individual classifiers:

```
>>> eclf = VotingClassifier(
...     estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],
...     voting='soft', weights=[2,5,1]
... )
```

Voting Regressor

The idea behind the *VotingRegressor* is to combine conceptually different machine learning regressors and return the average predicted values. Such a regressor can be useful for a set of equally well performing models in order to balance out their individual weaknesses.

Usage

The following example shows how to fit the VotingRegressor:

```
>>> from sklearn.datasets import load_boston
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> from sklearn.ensemble import RandomForestRegressor
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.ensemble import VotingRegressor

>>> # Loading some example data
>>> X, y = load_boston(return_X_y=True)

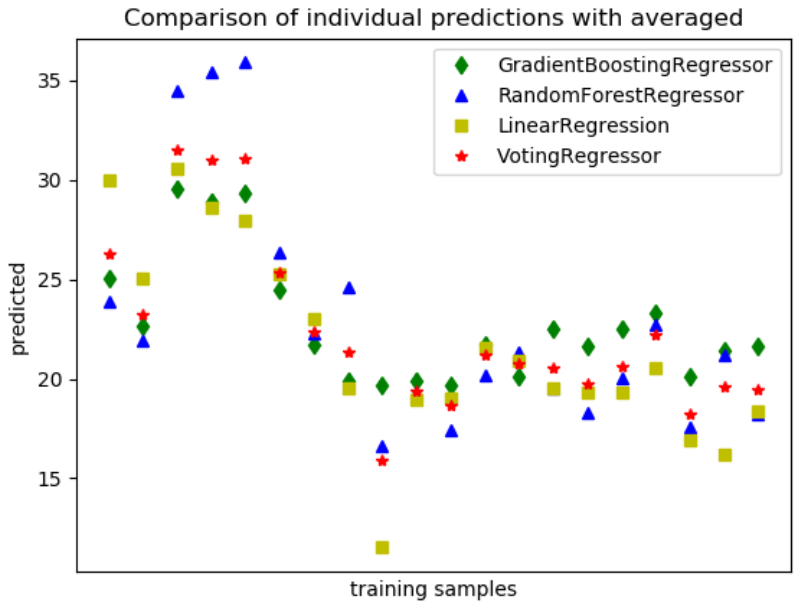
>>> # Training classifiers
>>> reg1 = GradientBoostingRegressor(random_state=1, n_estimators=10)
>>> reg2 = RandomForestRegressor(random_state=1, n_estimators=10)
>>> reg3 = LinearRegression()
>>> ereg = VotingRegressor(estimators=[('gb', reg1), ('rf', reg2), ('lr', reg3)])
>>> ereg = ereg.fit(X, y)
```

Examples:

- *Plot individual and voting regression predictions*

Stacked generalization

Stacked generalization is a method for combining estimators to reduce their biases [W1992] [HTF]. More precisely, the predictions of each individual estimator are stacked together and used as input to a final estimator to compute the prediction. This final estimator is trained through cross-validation.



The *StackingClassifier* and *StackingRegressor* provide such strategies which can be applied to classification and regression problems.

The `estimators` parameter corresponds to the list of the estimators which are stacked together in parallel on the input data. It should be given as a list of names and estimators:

```
>>> from sklearn.linear_model import RidgeCV, LassoCV
>>> from sklearn.svm import SVR
>>> estimators = [('ridge', RidgeCV()),
...               ('lasso', LassoCV(random_state=42)),
...               ('svr', SVR(C=1, gamma=1e-6))]
```

The `final_estimator` will use the predictions of the `estimators` as input. It needs to be a classifier or a regressor when using *StackingClassifier* or *StackingRegressor*, respectively:

```
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> from sklearn.ensemble import StackingRegressor
>>> reg = StackingRegressor(
...     estimators=estimators,
...     final_estimator=GradientBoostingRegressor(random_state=42))
```

To train the `estimators` and `final_estimator`, the `fit` method needs to be called on the training data:

```
>>> from sklearn.datasets import load_boston
>>> X, y = load_boston(return_X_y=True)
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
...                                                     random_state=42)
>>> reg.fit(X_train, y_train)
StackingRegressor(...)
```

During training, the `estimators` are fitted on the whole training data `X_train`. They will be used when calling `predict` or `predict_proba`. To generalize and avoid over-fitting, the `final_estimator` is trained on out-

samples using *sklearn.model_selection.cross_val_predict* internally.

For *StackingClassifier*, note that the output of the `estimators` is controlled by the parameter `stack_method` and it is called by each estimator. This parameter is either a string, being estimator method names, or `'auto'` which will automatically identify an available method depending on the availability, tested in the order of preference: `predict_proba`, `decision_function` and `predict`.

A *StackingRegressor* and *StackingClassifier* can be used as any other regressor or classifier, exposing a `predict`, `predict_proba`, and `decision_function` methods, e.g.:

```
>>> y_pred = reg.predict(X_test)
>>> from sklearn.metrics import r2_score
>>> print('R2 score: {:.2f}'.format(r2_score(y_test, y_pred)))
R2 score: 0.81
```

Note that it is also possible to get the output of the stacked outputs of the `estimators` using the `transform` method:

```
>>> reg.transform(X_test[:5])
array([[28.78..., 28.43...  , 22.62...],
       [35.96..., 32.58..., 23.68...],
       [14.97..., 14.05..., 16.45...],
       [25.19..., 25.54..., 22.92...],
       [18.93..., 19.26..., 17.03... ]])
```

In practise, a stacking predictor predict as good as the best predictor of the base layer and even sometimes outputperform it by combining the different strength of the these predictors. However, training a stacking predictor is computationally expensive.

Note: For *StackingClassifier*, when using `stack_method_='predict_proba'`, the first column is dropped when the problem is a binary classification problem. Indeed, both probability columns predicted by each estimator are perfectly collinear.

Note: Multiple stacking layers can be achieved by assigning `final_estimator` to a *StackingClassifier* or *StackingRegressor*:

```
>>> final_layer = StackingRegressor(
...     estimators=[('rf', RandomForestRegressor(random_state=42)),
...                 ('gbrt', GradientBoostingRegressor(random_state=42))],
...     final_estimator=RidgeCV()
...     )
>>> multi_layer_regressor = StackingRegressor(
...     estimators=[('ridge', RidgeCV()),
...                 ('lasso', LassoCV(random_state=42)),
...                 ('svr', SVR(C=1, gamma=1e-6, kernel='rbf'))],
...     final_estimator=final_layer
... )
>>> multi_layer_regressor.fit(X_train, y_train)
StackingRegressor(...)
>>> print('R2 score: {:.2f}'
...       .format(multi_layer_regressor.score(X_test, y_test)))
R2 score: 0.82
```

References

4.1.12 Multiclass and multilabel algorithms

Warning: All classifiers in scikit-learn do multiclass classification out-of-the-box. You don't need to use the *sklearn.multiclass* module unless you want to experiment with different multiclass strategies.

The *sklearn.multiclass* module implements *meta-estimators* to solve `multiclass` and `multilabel` classification problems by decomposing such problems into binary classification problems. `multioutput` regression is also supported.

- **Multiclass classification**: classification task with more than two classes. Each sample can only be labelled as one class.

  For example, classification using features extracted from a set of images of fruit, where each image may either be of an orange, an apple, or a pear. Each image is one sample and is labelled as one of the 3 possible classes. Multiclass classification makes the assumption that each sample is assigned to one and only one label - one sample cannot, for example, be both a pear and an apple.

  Valid *multiclass* representations for `type_of_target` (`y`) are:

  – 1d or column vector containing more than two discrete values. An example of a vector `y` for 3 samples:

  ```
  >>> import numpy as np
  >>> y = np.array(['apple', 'pear', 'apple'])
  >>> print(y)
  ['apple' 'pear' 'apple']
  ```

  – sparse *binary* matrix of shape `(n_samples, n_classes)` with a single element per row, where each column represents one class. An example of a sparse *binary* matrix `y` for 3 samples, where the columns, in order, are orange, apple and pear:

  ```
  >>> from scipy import sparse
  >>> row_ind = np.array([0, 1, 2])
  >>> col_ind = np.array([1, 2, 1])
  >>> y_sparse = sparse.csr_matrix((np.ones(3), (row_ind, col_ind)))
  >>> print(y_sparse)
    (0, 1)        1.0
    (1, 2)        1.0
    (2, 1)        1.0
  ```

- **Multilabel classification**: classification task labelling each sample with `x` labels from `n_classes` possible classes, where `x` can be 0 to `n_classes` inclusive. This can be thought of as predicting properties of a sample that are not mutually exclusive. Formally, a binary output is assigned to each class, for every sample. Positive classes are indicated with 1 and negative classes with 0 or -1. It is thus comparable to running `n_classes` binary classification tasks, for example with *sklearn.multioutput.MultiOutputClassifier*. This approach treats each label independently whereas multilabel classifiers *may* treat the multiple classes simultaneously, accounting for correlated behaviour amoung them.

  For example, prediction of the topics relevant to a text document or video. The document or video may be about one of 'religion', 'politics', 'finance' or 'education', several of the topic classes or all of the topic classes.

  Valid representation of *multilabel* `y` is either dense (or sparse) *binary* matrix of shape `(n_samples, n_classes)`. Each column represents a class. The `1`'s in each row denote the positive classes a sample

has been labelled with. An example of a dense matrix `y` for 3 samples:

```
>>> y = np.array([[1, 0, 0, 1], [0, 0, 1, 1], [0, 0, 0, 0]])
>>> print(y)
[[1 0 0 1]
 [0 0 1 1]
 [0 0 0 0]]
```

An example of the same `y` in sparse matrix form:

```
>>> y_sparse = sparse.csr_matrix(y)
>>> print(y_sparse)
  (0, 0)    1
  (0, 3)    1
  (1, 2)    1
  (1, 3)    1
```

- **Multioutput regression**: predicts multiple numerical properties for each sample. Each property is a numerical variable and the number of properties to be predicted for each sample is greater than or equal to 2. Some estimators that support multioutput regression are faster than just running `n_output` estimators.

  For example, prediction of both wind speed and wind direction, in degrees, using data obtained at a certain location. Each sample would be data obtained at one location and both wind speed and directtion would be output for each sample.

  Valid representation of *multilabel* `y` is dense matrix of shape `(n_samples, n_classes)` of floats. A column wise concatenation of *continuous* variables. An example of `y` for 3 samples:

```
>>> y = np.array([[31.4, 94], [40.5, 109], [25.0, 30]])
>>> print(y)
[[ 31.4  94. ]
 [ 40.5 109. ]
 [ 25.   30. ]]
```

- **Multioutput-multiclass classification** (also known as **multitask classification**): classification task which labels each sample with a set of **non-binary** properties. Both the number of properties and the number of classes per property is greater than 2. A single estimator thus handles several joint classification tasks. This is both a generalization of the multi*label* classification task, which only considers binary attributes, as well as a generalization of the multi*class* classification task, where only one property is considered.

  For example, classification of the properties "type of fruit" and "colour" for a set of images of fruit. The property "type of fruit" has the possible classes: "apple", "pear" and "orange". The property "colour" has the possible classes: "green", "red", "yellow" and "orange". Each sample is an image of a fruit, a label is output for both properties and each label is one of the possible classes of the corresponding property.

  Valid representation of *multilabel* `y` is dense matrix of shape `(n_samples, n_classes)` of floats. A column wise concatenation of 1d *multiclass* variables. An example of `y` for 3 samples:

```
>>> y = np.array([['apple', 'green'], ['orange', 'orange'], ['pear', 'green']])
>>> print(y)
[['apple' 'green']
 ['orange' 'orange']
 ['pear' 'green']]
```

  Note that any classifiers handling multioutput-multiclass (also known as multitask classification) tasks, support the multilabel classification task as a special case. Multitask classification is similar to the multioutput classification task with different model formulations. For more information, see the relevant estimator documentation.

All scikit-learn classifiers are capable of multiclass classification, but the meta-estimators offered by *sklearn.multiclass* permit changing the way they handle more than two classes because this may have an effect on classifier performance (either in terms of generalization error or required computational resources).

Summary

	Number of targets	Target cardinality	Valid `type_of_target`
Multiclass classification	1	>2	• 'multiclass'
Multilabel classification	>1	2 (0 or 1)	• 'multilabel-indicator'
Multioutput regression	>1	Continuous	• 'continuous-multioutput'
Multioutput- multiclass classification	>1	>2	• 'multiclass-multioutput'

Below is a summary of the classifiers supported by scikit-learn grouped by strategy; you don't need the meta-estimators in this class if you're using one of these, unless you want custom multiclass behavior:

- **Inherently multiclass:**
  - *sklearn.naive_bayes.BernoulliNB*
  - *sklearn.tree.DecisionTreeClassifier*
  - *sklearn.tree.ExtraTreeClassifier*
  - *sklearn.ensemble.ExtraTreesClassifier*
  - *sklearn.naive_bayes.GaussianNB*
  - *sklearn.neighbors.KNeighborsClassifier*
  - *sklearn.semi_supervised.LabelPropagation*
  - *sklearn.semi_supervised.LabelSpreading*
  - *sklearn.discriminant_analysis.LinearDiscriminantAnalysis*
  - *sklearn.svm.LinearSVC* (setting multi_class="crammer_singer")
  - *sklearn.linear_model.LogisticRegression* (setting multi_class="multinomial")
  - *sklearn.linear_model.LogisticRegressionCV* (setting multi_class="multinomial")
  - *sklearn.neural_network.MLPClassifier*
  - *sklearn.neighbors.NearestCentroid*
  - *sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis*
  - *sklearn.neighbors.RadiusNeighborsClassifier*
  - *sklearn.ensemble.RandomForestClassifier*
  - *sklearn.linear_model.RidgeClassifier*
  - *sklearn.linear_model.RidgeClassifierCV*

- **Multiclass as One-Vs-One:**
  - *sklearn.svm.NuSVC*
  - *sklearn.svm.SVC*.
  - *sklearn.gaussian_process.GaussianProcessClassifier* (setting multi_class = "one_vs_one")
- **Multiclass as One-Vs-The-Rest:**
  - *sklearn.ensemble.GradientBoostingClassifier*
  - *sklearn.gaussian_process.GaussianProcessClassifier* (setting multi_class = "one_vs_rest")
  - *sklearn.svm.LinearSVC* (setting multi_class="ovr")
  - *sklearn.linear_model.LogisticRegression* (setting multi_class="ovr")
  - *sklearn.linear_model.LogisticRegressionCV* (setting multi_class="ovr")
  - *sklearn.linear_model.SGDClassifier*
  - *sklearn.linear_model.Perceptron*
  - *sklearn.linear_model.PassiveAggressiveClassifier*
- **Support multilabel:**
  - *sklearn.tree.DecisionTreeClassifier*
  - *sklearn.tree.ExtraTreeClassifier*
  - *sklearn.ensemble.ExtraTreesClassifier*
  - *sklearn.neighbors.KNeighborsClassifier*
  - *sklearn.neural_network.MLPClassifier*
  - *sklearn.neighbors.RadiusNeighborsClassifier*
  - *sklearn.ensemble.RandomForestClassifier*
  - *sklearn.linear_model.RidgeClassifierCV*
- **Support multiclass-multioutput:**
  - *sklearn.tree.DecisionTreeClassifier*
  - *sklearn.tree.ExtraTreeClassifier*
  - *sklearn.ensemble.ExtraTreesClassifier*
  - *sklearn.neighbors.KNeighborsClassifier*
  - *sklearn.neighbors.RadiusNeighborsClassifier*
  - *sklearn.ensemble.RandomForestClassifier*

Warning: At present, no metric in *sklearn.metrics* supports the multioutput-multiclass classification task.

Multilabel classification format

In multilabel learning, the joint set of binary classification tasks is expressed with label binary indicator array: each sample is one row of a 2d array of shape (n_samples, n_classes) with binary values: the one, i.e. the non zero elements, corresponds to the subset of labels. An array such as `np.array([[1, 0, 0], [0, 1, 1], [0, 0, 0]])` represents label 0 in the first sample, labels 1 and 2 in the second sample, and no labels in the third sample.

Producing multilabel data as a list of sets of labels may be more intuitive. The *MultiLabelBinarizer* transformer can be used to convert between a collection of collections of labels and the indicator format.

```
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> y = [[2, 3, 4], [2], [0, 1, 3], [0, 1, 2, 3, 4], [0, 1, 2]]
>>> MultiLabelBinarizer().fit_transform(y)
array([[0, 0, 1, 1, 1],
       [0, 0, 1, 0, 0],
       [1, 1, 0, 1, 0],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 0, 0]])
```

One-Vs-The-Rest

This strategy, also known as **one-vs-all**, is implemented in *OneVsRestClassifier*. The strategy consists in fitting one classifier per class. For each classifier, the class is fitted against all the other classes. In addition to its computational efficiency (only `n_classes` classifiers are needed), one advantage of this approach is its interpretability. Since each class is represented by one and only one classifier, it is possible to gain knowledge about the class by inspecting its corresponding classifier. This is the most commonly used strategy and is a fair default choice.

Multiclass learning

Below is an example of multiclass learning using OvR:

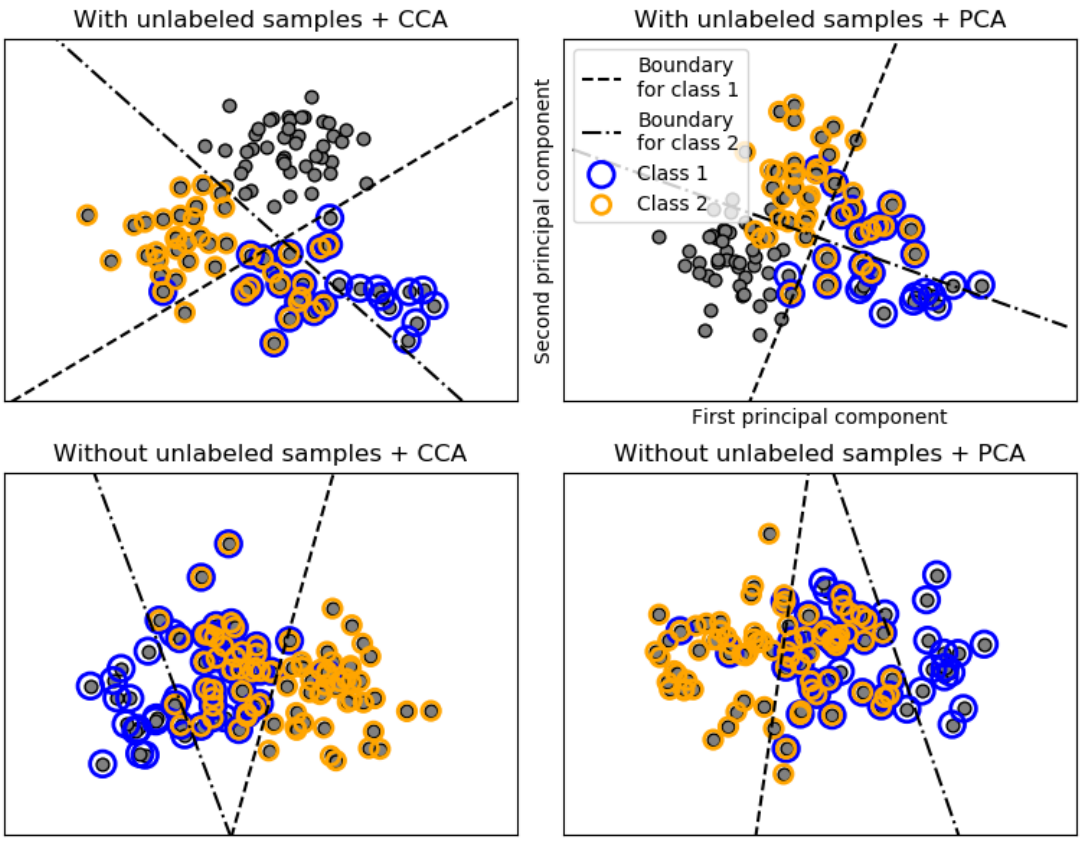
```
>>> from sklearn import datasets
>>> from sklearn.multiclass import OneVsRestClassifier
>>> from sklearn.svm import LinearSVC
>>> X, y = datasets.load_iris(return_X_y=True)
>>> OneVsRestClassifier(LinearSVC(random_state=0)).fit(X, y).predict(X)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Multilabel learning

OneVsRestClassifier also supports multilabel classification. To use this feature, feed the classifier an indicator matrix, in which cell [i, j] indicates the presence of label j in sample i.

Examples:

- *Multilabel classification*



One-Vs-One

OneVsOneClassifier constructs one classifier per pair of classes. At prediction time, the class which received the most votes is selected. In the event of a tie (among two classes with an equal number of votes), it selects the class with the highest aggregate classification confidence by summing over the pair-wise classification confidence levels computed by the underlying binary classifiers.

Since it requires to fit `n_classes * (n_classes - 1) / 2` classifiers, this method is usually slower than one-vs-the-rest, due to its O(n_classes^2) complexity. However, this method may be advantageous for algorithms such as kernel algorithms which don't scale well with `n_samples`. This is because each individual learning problem only involves a small subset of the data whereas, with one-vs-the-rest, the complete dataset is used `n_classes` times. The decision function is the result of a monotonic transformation of the one-versus-one classification.

Multiclass learning

Below is an example of multiclass learning using OvO:

```
>>> from sklearn import datasets
>>> from sklearn.multiclass import OneVsOneClassifier
>>> from sklearn.svm import LinearSVC
>>> X, y = datasets.load_iris(return_X_y=True)
>>> OneVsOneClassifier(LinearSVC(random_state=0)).fit(X, y).predict(X)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

(continues on next page)

(continued from previous page)

```
       1, 2, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

References:

- "Pattern Recognition and Machine Learning. Springer", Christopher M. Bishop, page 183, (First Edition)

Error-Correcting Output-Codes

Output-code based strategies are fairly different from one-vs-the-rest and one-vs-one. With these strategies, each class is represented in a Euclidean space, where each dimension can only be 0 or 1. Another way to put it is that each class is represented by a binary code (an array of 0 and 1). The matrix which keeps track of the location/code of each class is called the code book. The code size is the dimensionality of the aforementioned space. Intuitively, each class should be represented by a code as unique as possible and a good code book should be designed to optimize classification accuracy. In this implementation, we simply use a randomly-generated code book as advocated in[3] although more elaborate methods may be added in the future.

At fitting time, one binary classifier per bit in the code book is fitted. At prediction time, the classifiers are used to project new points in the class space and the class closest to the points is chosen.

In *`OutputCodeClassifier`*, the `code_size` attribute allows the user to control the number of classifiers which will be used. It is a percentage of the total number of classes.

A number between 0 and 1 will require fewer classifiers than one-vs-the-rest. In theory, `log2(n_classes) / n_classes` is sufficient to represent each class unambiguously. However, in practice, it may not lead to good accuracy since `log2(n_classes)` is much smaller than n_classes.

A number greater than 1 will require more classifiers than one-vs-the-rest. In this case, some classifiers will in theory correct for the mistakes made by other classifiers, hence the name "error-correcting". In practice, however, this may not happen as classifier mistakes will typically be correlated. The error-correcting output codes have a similar effect to bagging.

Multiclass learning

Below is an example of multiclass learning using Output-Codes:

```
>>> from sklearn import datasets
>>> from sklearn.multiclass import OutputCodeClassifier
>>> from sklearn.svm import LinearSVC
>>> X, y = datasets.load_iris(return_X_y=True)
>>> clf = OutputCodeClassifier(LinearSVC(random_state=0),
...                            code_size=2, random_state=0)
>>> clf.fit(X, y).predict(X)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1,
       1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
```

(continues on next page)

[3] "The error coding method and PICTs", James G., Hastie T., Journal of Computational and Graphical statistics 7, 1998.

(continued from previous page)

```
       2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 1, 1, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

References:

- "Solving multiclass learning problems via error-correcting output codes", Dietterich T., Bakiri G., Journal of Artificial Intelligence Research 2, 1995.
- "The Elements of Statistical Learning", Hastie T., Tibshirani R., Friedman J., page 606 (second-edition) 2008.

Multioutput regression

Multioutput regression support can be added to any regressor with `MultiOutputRegressor`. This strategy consists of fitting one regressor per target. Since each target is represented by exactly one regressor it is possible to gain knowledge about the target by inspecting its corresponding regressor. As `MultiOutputRegressor` fits one regressor per target it can not take advantage of correlations between targets.

Below is an example of multioutput regression:

```
>>> from sklearn.datasets import make_regression
>>> from sklearn.multioutput import MultiOutputRegressor
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> X, y = make_regression(n_samples=10, n_targets=3, random_state=1)
>>> MultiOutputRegressor(GradientBoostingRegressor(random_state=0)).fit(X, y).
↪predict(X)
array([[-154.75474165, -147.03498585,  -50.03812219],
       [   7.12165031,    5.12914884,  -81.46081961],
       [-187.8948621 , -100.44373091,   13.88978285],
       [-141.62745778,   95.02891072, -191.48204257],
       [  97.03260883,  165.34867495,  139.52003279],
       [ 123.92529176,   21.25719016,   -7.84253   ],
       [-122.25193977,  -85.16443186, -107.12274212],
       [ -30.170388  ,  -94.80956739,   12.16979946],
       [ 140.72667194,  176.50941682,  -17.50447799],
       [ 149.37967282,  -81.15699552,   -5.72850319]])
```

Multioutput classification

Multioutput classification support can be added to any classifier with `MultiOutputClassifier`. This strategy consists of fitting one classifier per target. This allows multiple target variable classifications. The purpose of this class is to extend estimators to be able to estimate a series of target functions (f1,f2,f3...,fn) that are trained on a single X predictor matrix to predict a series of responses (y1,y2,y3...,yn).

Below is an example of multioutput classification:

```
>>> from sklearn.datasets import make_classification
>>> from sklearn.multioutput import MultiOutputClassifier
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.utils import shuffle
>>> import numpy as np
>>> X, y1 = make_classification(n_samples=10, n_features=100, n_informative=30, n_
↪classes=3, random_state=1)
```

(continues on next page)

(continued from previous page)

```
>>> y2 = shuffle(y1, random_state=1)
>>> y3 = shuffle(y1, random_state=2)
>>> Y = np.vstack((y1, y2, y3)).T
>>> n_samples, n_features = X.shape # 10,100
>>> n_outputs = Y.shape[1] # 3
>>> n_classes = 3
>>> forest = RandomForestClassifier(random_state=1)
>>> multi_target_forest = MultiOutputClassifier(forest, n_jobs=-1)
>>> multi_target_forest.fit(X, Y).predict(X)
array([[2, 2, 0],
       [1, 2, 1],
       [2, 1, 0],
       [0, 0, 2],
       [0, 2, 1],
       [0, 0, 2],
       [1, 1, 0],
       [1, 1, 1],
       [0, 0, 2],
       [2, 0, 0]])
```

Classifier Chain

Classifier chains (see `ClassifierChain`) are a way of combining a number of binary classifiers into a single multi-label model that is capable of exploiting correlations among targets.

For a multi-label classification problem with N classes, N binary classifiers are assigned an integer between 0 and N-1. These integers define the order of models in the chain. Each classifier is then fit on the available training data plus the true labels of the classes whose models were assigned a lower number.

When predicting, the true labels will not be available. Instead the predictions of each model are passed on to the subsequent models in the chain to be used as features.

Clearly the order of the chain is important. The first model in the chain has no information about the other labels while the last model in the chain has features indicating the presence of all of the other labels. In general one does not know the optimal ordering of the models in the chain so typically many randomly ordered chains are fit and their predictions are averaged together.

References:

Jesse Read, Bernhard Pfahringer, Geoff Holmes, Eibe Frank, "Classifier Chains for Multi-label Classification", 2009.

Regressor Chain

Regressor chains (see `RegressorChain`) is analogous to ClassifierChain as a way of combining a number of regressions into a single multi-target model that is capable of exploiting correlations among targets.

4.1.13 Feature selection

The classes in the *sklearn.feature_selection* module can be used for feature selection/dimensionality reduction on sample sets, either to improve estimators' accuracy scores or to boost their performance on very high-dimensional datasets.

Removing features with low variance

VarianceThreshold is a simple baseline approach to feature selection. It removes all features whose variance doesn't meet some threshold. By default, it removes all zero-variance features, i.e. features that have the same value in all samples.

As an example, suppose that we have a dataset with boolean features, and we want to remove all features that are either one or zero (on or off) in more than 80% of the samples. Boolean features are Bernoulli random variables, and the variance of such variables is given by

$$\mathrm{Var}[X] = p(1-p)$$

so we can select using the threshold `.8 * (1 - .8)`:

```
>>> from sklearn.feature_selection import VarianceThreshold
>>> X = [[0, 0, 1], [0, 1, 0], [1, 0, 0], [0, 1, 1], [0, 1, 0], [0, 1, 1]]
>>> sel = VarianceThreshold(threshold=(.8 * (1 - .8)))
>>> sel.fit_transform(X)
array([[0, 1],
       [1, 0],
       [0, 0],
       [1, 1],
       [1, 0],
       [1, 1]])
```

As expected, `VarianceThreshold` has removed the first column, which has a probability $p = 5/6 > .8$ of containing a zero.

Univariate feature selection

Univariate feature selection works by selecting the best features based on univariate statistical tests. It can be seen as a preprocessing step to an estimator. Scikit-learn exposes feature selection routines as objects that implement the `transform` method:

- *SelectKBest* removes all but the k highest scoring features
- *SelectPercentile* removes all but a user-specified highest scoring percentage of features
- using common univariate statistical tests for each feature: false positive rate *SelectFpr*, false discovery rate *SelectFdr*, or family wise error *SelectFwe*.
- *GenericUnivariateSelect* allows to perform univariate feature selection with a configurable strategy. This allows to select the best univariate selection strategy with hyper-parameter search estimator.

For instance, we can perform a χ^2 test to the samples to retrieve only the two best features as follows:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectKBest
>>> from sklearn.feature_selection import chi2
>>> X, y = load_iris(return_X_y=True)
>>> X.shape
(150, 4)
>>> X_new = SelectKBest(chi2, k=2).fit_transform(X, y)
>>> X_new.shape
(150, 2)
```

These objects take as input a scoring function that returns univariate scores and p-values (or only scores for *SelectKBest* and *SelectPercentile*):

- For regression: *f_regression*, *mutual_info_regression*
- For classification: *chi2*, *f_classif*, *mutual_info_classif*

The methods based on F-test estimate the degree of linear dependency between two random variables. On the other hand, mutual information methods can capture any kind of statistical dependency, but being nonparametric, they require more samples for accurate estimation.

Feature selection with sparse data

If you use sparse data (i.e. data represented as sparse matrices), *chi2*, *mutual_info_regression*, *mutual_info_classif* will deal with the data without making it dense.

Warning: Beware not to use a regression scoring function with a classification problem, you will get useless results.

Examples:

- *Univariate Feature Selection*
- *Comparison of F-test and mutual information*

Recursive feature elimination

Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model), recursive feature elimination (*RFE*) is to select features by recursively considering smaller and smaller sets of features. First, the estimator is trained on the initial set of features and the importance of each feature is obtained either through a `coef_` attribute or through a `feature_importances_` attribute. Then, the least important features are pruned from current set of features.That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.

RFECV performs RFE in a cross-validation loop to find the optimal number of features.

Examples:

- *Recursive feature elimination*: A recursive feature elimination example showing the relevance of pixels in a digit classification task.
- *Recursive feature elimination with cross-validation*: A recursive feature elimination example with automatic tuning of the number of features selected with cross-validation.

Feature selection using SelectFromModel

SelectFromModel is a meta-transformer that can be used along with any estimator that has a `coef_` or `feature_importances_` attribute after fitting. The features are considered unimportant and removed, if the corresponding `coef_` or `feature_importances_` values are below the provided `threshold` parameter. Apart from specifying the threshold numerically, there are built-in heuristics for finding a threshold using a string argument. Available heuristics are “mean”, “median” and float multiples of these like “0.1*mean”.

For examples on how it is to be used refer to the sections below.

Examples

- *Feature selection using SelectFromModel and LassoCV*: Selecting the two most important features from the Boston dataset without knowing the threshold beforehand.

L1-based feature selection

Linear models penalized with the L1 norm have sparse solutions: many of their estimated coefficients are zero. When the goal is to reduce the dimensionality of the data to use with another classifier, they can be used along with `feature_selection.SelectFromModel` to select the non-zero coefficients. In particular, sparse estimators useful for this purpose are the `linear_model.Lasso` for regression, and of `linear_model.LogisticRegression` and `svm.LinearSVC` for classification:

```
>>> from sklearn.svm import LinearSVC
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectFromModel
>>> X, y = load_iris(return_X_y=True)
>>> X.shape
(150, 4)
>>> lsvc = LinearSVC(C=0.01, penalty="l1", dual=False).fit(X, y)
>>> model = SelectFromModel(lsvc, prefit=True)
>>> X_new = model.transform(X)
>>> X_new.shape
(150, 3)
```

With SVMs and logistic-regression, the parameter C controls the sparsity: the smaller C the fewer features selected. With Lasso, the higher the alpha parameter, the fewer features selected.

Examples:

- *Classification of text documents using sparse features*: Comparison of different algorithms for document classification including L1-based feature selection.

L1-recovery and compressive sensing

For a good choice of alpha, the *Lasso* can fully recover the exact set of non-zero variables using only few observations, provided certain specific conditions are met. In particular, the number of samples should be "sufficiently large", or L1 models will perform at random, where "sufficiently large" depends on the number of non-zero coefficients, the logarithm of the number of features, the amount of noise, the smallest absolute value of non-zero coefficients, and the structure of the design matrix X. In addition, the design matrix must display certain specific properties, such as not being too correlated.

There is no general rule to select an alpha parameter for recovery of non-zero coefficients. It can by set by cross-validation (`LassoCV` or `LassoLarsCV`), though this may lead to under-penalized models: including a small number of non-relevant variables is not detrimental to prediction score. BIC (`LassoLarsIC`) tends, on the opposite, to set high values of alpha.

Reference Richard G. Baraniuk "Compressive Sensing", IEEE Signal Processing Magazine [120] July 2007 http://users.isr.ist.utl.pt/~aguiar/CS_notes.pdf

Tree-based feature selection

Tree-based estimators (see the *sklearn.tree* module and forest of trees in the *sklearn.ensemble* module) can be used to compute feature importances, which in turn can be used to discard irrelevant features (when coupled with the *sklearn.feature_selection.SelectFromModel* meta-transformer):

```
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectFromModel
>>> X, y = load_iris(return_X_y=True)
>>> X.shape
(150, 4)
>>> clf = ExtraTreesClassifier(n_estimators=50)
>>> clf = clf.fit(X, y)
>>> clf.feature_importances_
array([ 0.04...,  0.05...,  0.4...,  0.4...])
>>> model = SelectFromModel(clf, prefit=True)
>>> X_new = model.transform(X)
>>> X_new.shape
(150, 2)
```

Examples:

- *Feature importances with forests of trees*: example on synthetic data showing the recovery of the actually meaningful features.
- *Pixel importances with a parallel forest of trees*: example on face recognition data.

Feature selection as part of a pipeline

Feature selection is usually used as a pre-processing step before doing the actual learning. The recommended way to do this in scikit-learn is to use a *sklearn.pipeline.Pipeline*:

```
clf = Pipeline([
  ('feature_selection', SelectFromModel(LinearSVC(penalty="l1"))),
  ('classification', RandomForestClassifier())
])
clf.fit(X, y)
```

In this snippet we make use of a *sklearn.svm.LinearSVC* coupled with *sklearn.feature_selection.SelectFromModel* to evaluate feature importances and select the most relevant features. Then, a *sklearn.ensemble.RandomForestClassifier* is trained on the transformed output, i.e. using only relevant features. You can perform similar operations with the other feature selection methods and also classifiers that provide a way to evaluate feature importances of course. See the *sklearn.pipeline.Pipeline* examples for more details.

4.1.14 Semi-Supervised

Semi-supervised learning is a situation in which in your training data some of the samples are not labeled. The semi-supervised estimators in *sklearn.semi_supervised* are able to make use of this additional unlabeled data to better capture the shape of the underlying data distribution and generalize better to new samples. These algorithms can perform well when we have a very small amount of labeled points and a large amount of unlabeled points.

Unlabeled entries in y

It is important to assign an identifier to unlabeled points along with the labeled data when training the model with the `fit` method. The identifier that this implementation uses is the integer value -1.

Label Propagation

Label propagation denotes a few variations of semi-supervised graph inference algorithms.

A few features available in this model:

- Can be used for classification and regression tasks
- Kernel methods to project data into alternate dimensional spaces

`scikit-learn` provides two label propagation models: *`LabelPropagation`* and *`LabelSpreading`*. Both work by constructing a similarity graph over all items in the input dataset.

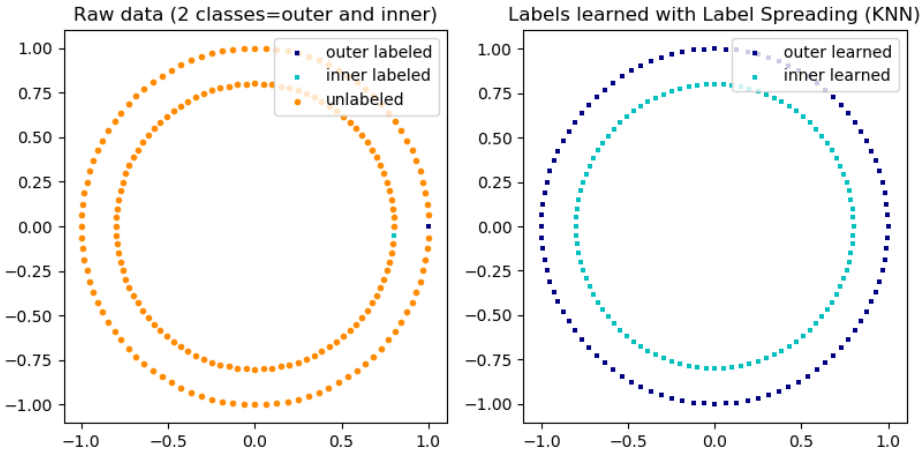


Fig. 1: **An illustration of label-propagation:** *the structure of unlabeled observations is consistent with the class structure, and thus the class label can be propagated to the unlabeled observations of the training set.*

`LabelPropagation` and *`LabelSpreading`* differ in modifications to the similarity matrix that graph and the clamping effect on the label distributions. Clamping allows the algorithm to change the weight of the true ground labeled data to some degree. The *`LabelPropagation`* algorithm performs hard clamping of input labels, which means $\alpha = 0$. This clamping factor can be relaxed, to say $\alpha = 0.2$, which means that we will always retain 80 percent of our original label distribution, but the algorithm gets to change its confidence of the distribution within 20 percent.

`LabelPropagation` uses the raw similarity matrix constructed from the data with no modifications. In contrast, *`LabelSpreading`* minimizes a loss function that has regularization properties, as such it is often more robust to noise. The algorithm iterates on a modified version of the original graph and normalizes the edge weights by computing the normalized graph Laplacian matrix. This procedure is also used in *Spectral clustering*.

Label propagation models have two built-in kernel methods. Choice of kernel effects both scalability and performance of the algorithms. The following are available:

- rbf ($\exp(-\gamma|x - y|^2), \gamma > 0$). γ is specified by keyword gamma.
- knn ($1[x' \in kNN(x)]$). k is specified by keyword n_neighbors.

The RBF kernel will produce a fully connected graph which is represented in memory by a dense matrix. This matrix may be very large and combined with the cost of performing a full matrix multiplication calculation for each iteration

of the algorithm can lead to prohibitively long running times. On the other hand, the KNN kernel will produce a much more memory-friendly sparse matrix which can drastically reduce running times.

Examples

- *Decision boundary of label propagation versus SVM on the Iris dataset*
- *Label Propagation learning a complex structure*
- *Label Propagation digits: Demonstrating performance*
- *Label Propagation digits active learning*

References

[1] Yoshua Bengio, Olivier Delalleau, Nicolas Le Roux. In Semi-Supervised Learning (2006), pp. 193-216

[2] Olivier Delalleau, Yoshua Bengio, Nicolas Le Roux. Efficient Non-Parametric Function Induction in Semi-Supervised Learning. AISTAT 2005 https://research.microsoft.com/en-us/people/nicolasl/efficient_ssl.pdf

4.1.15 Isotonic regression

The class `IsotonicRegression` fits a non-decreasing function to data. It solves the following problem:

minimize $\sum_i w_i(y_i - \hat{y}_i)^2$

subject to $\hat{y}_{min} = \hat{y}_1 \leq \hat{y}_2 ... \leq \hat{y}_n = \hat{y}_{max}$

where each w_i is strictly positive and each y_i is an arbitrary real number. It yields the vector which is composed of non-decreasing elements the closest in terms of mean squared error. In practice this list of elements forms a function that is piecewise linear.

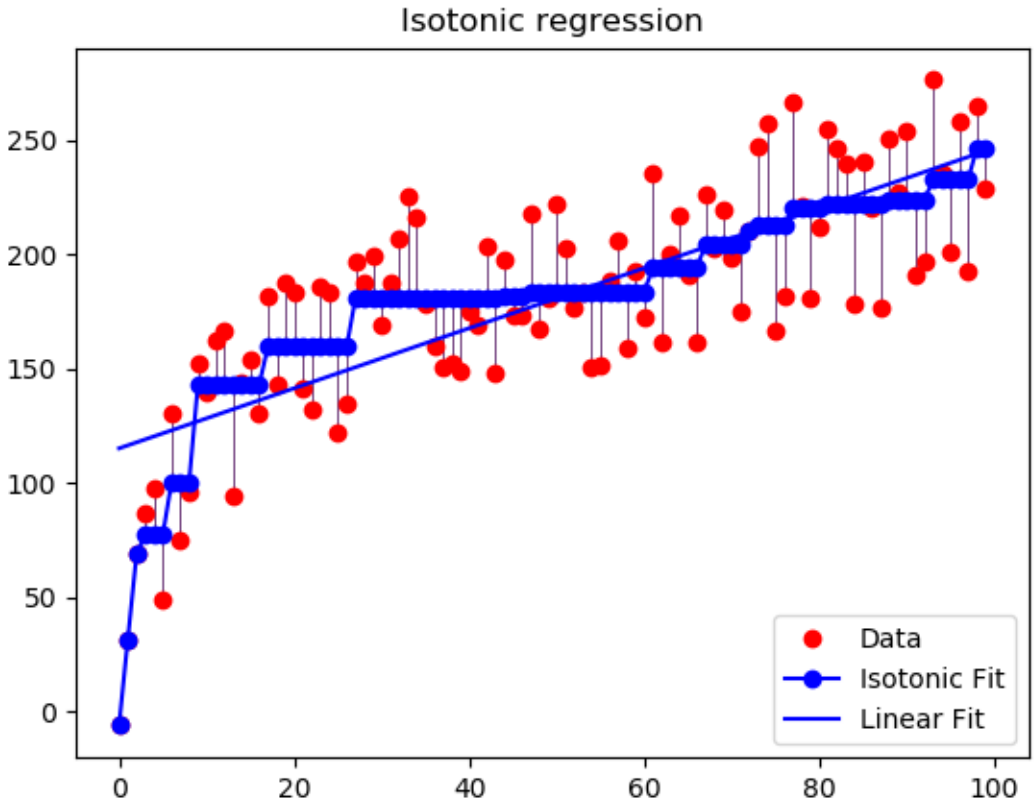
4.1.16 Probability calibration

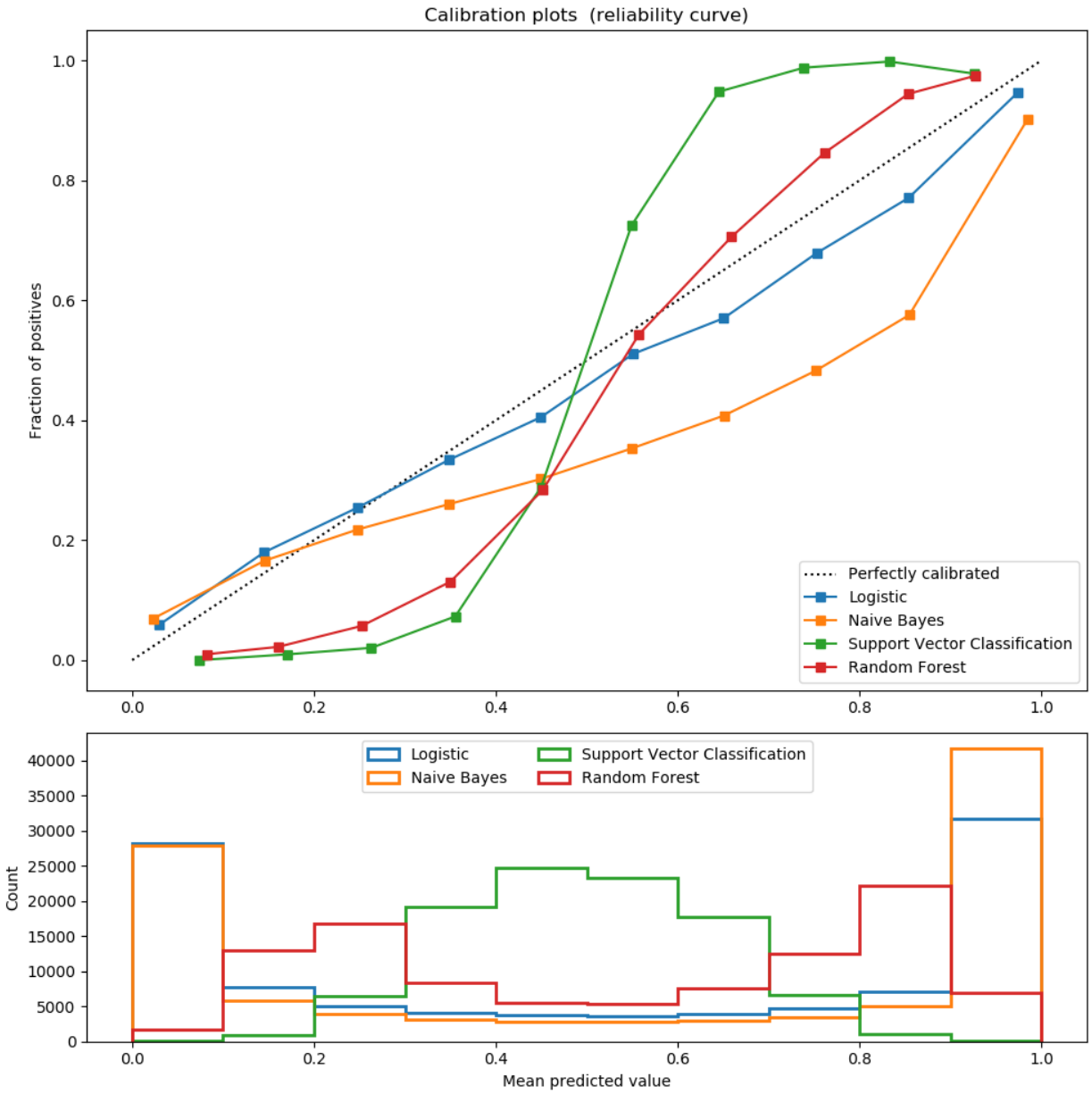
When performing classification you often want not only to predict the class label, but also obtain a probability of the respective label. This probability gives you some kind of confidence on the prediction. Some models can give you poor estimates of the class probabilities and some even do not support probability prediction. The calibration module allows you to better calibrate the probabilities of a given model, or to add support for probability prediction.

Well calibrated classifiers are probabilistic classifiers for which the output of the predict_proba method can be directly interpreted as a confidence level. For instance, a well calibrated (binary) classifier should classify the samples such that among the samples to which it gave a predict_proba value close to 0.8, approximately 80% actually belong to the positive class. The following plot compares how well the probabilistic predictions of different classifiers are calibrated:

`LogisticRegression` returns well calibrated predictions by default as it directly optimizes log-loss. In contrast, the other methods return biased probabilities; with different biases per method:

- `GaussianNB` tends to push probabilities to 0 or 1 (note the counts in the histograms). This is mainly because it makes the assumption that features are conditionally independent given the class, which is not the case in this dataset which contains 2 redundant features.
- `RandomForestClassifier` shows the opposite behavior: the histograms show peaks at approximately 0.2 and 0.9 probability, while probabilities close to 0 or 1 are very rare. An explanation for this is given by


Isotonic regression
250
200
150
100
50
0
0
20
40
60
80
100
Data
Isotonic Fit
Linear Fit


Calibration plots (reliability curve)
Fraction of positives
Perfectly calibrated
Logistic
Naive Bayes
Support Vector Classification
Random Forest
Count
Mean predicted value

Niculescu-Mizil and Caruana[4]: "Methods such as bagging and random forests that average predictions from a base set of models can have difficulty making predictions near 0 and 1 because variance in the underlying base models will bias predictions that should be near zero or one away from these values. Because predictions are restricted to the interval [0,1], errors caused by variance tend to be one-sided near zero and one. For example, if a model should predict p = 0 for a case, the only way bagging can achieve this is if all bagged trees predict zero. If we add noise to the trees that bagging is averaging over, this noise will cause some trees to predict values larger than 0 for this case, thus moving the average prediction of the bagged ensemble away from 0. We observe this effect most strongly with random forests because the base-level trees trained with random forests have relatively high variance due to feature subsetting." As a result, the calibration curve also referred to as the reliability diagram (Wilks 1995[5]) shows a characteristic sigmoid shape, indicating that the classifier could trust its "intuition" more and return probabilities closer to 0 or 1 typically.

- Linear Support Vector Classification (`LinearSVC`) shows an even more sigmoid curve as the RandomForestClassifier, which is typical for maximum-margin methods (compare Niculescu-Mizil and Caruana[4]), which focus on hard samples that are close to the decision boundary (the support vectors).

Two approaches for performing calibration of probabilistic predictions are provided: a parametric approach based on Platt's sigmoid model and a non-parametric approach based on isotonic regression (`sklearn.isotonic`). Probability calibration should be done on new data not used for model fitting. The class `CalibratedClassifierCV` uses a cross-validation generator and estimates for each split the model parameter on the train samples and the calibration of the test samples. The probabilities predicted for the folds are then averaged. Already fitted classifiers can be calibrated by `CalibratedClassifierCV` via the parameter cv="prefit". In this case, the user has to take care manually that data for model fitting and calibration are disjoint.

The following images demonstrate the benefit of probability calibration. The first image present a dataset with 2 classes and 3 blobs of data. The blob in the middle contains random samples of each class. The probability for the samples in this blob should be 0.5.

The following image shows on the data above the estimated probability using a Gaussian naive Bayes classifier without calibration, with a sigmoid calibration and with a non-parametric isotonic calibration. One can observe that the non-parametric model provides the most accurate probability estimates for samples in the middle, i.e., 0.5.

The following experiment is performed on an artificial dataset for binary classification with 100,000 samples (1,000 of them are used for model fitting) with 20 features. Of the 20 features, only 2 are informative and 10 are redundant. The figure shows the estimated probabilities obtained with logistic regression, a linear support-vector classifier (SVC), and linear SVC with both isotonic calibration and sigmoid calibration. The Brier score is a metric which is a combination of calibration loss and refinement loss, `brier_score_loss`, reported in the legend (the smaller the better). Calibration loss is defined as the mean squared deviation from empirical probabilities derived from the slope of ROC segments. Refinement loss can be defined as the expected optimal loss as measured by the area under the optimal cost curve.

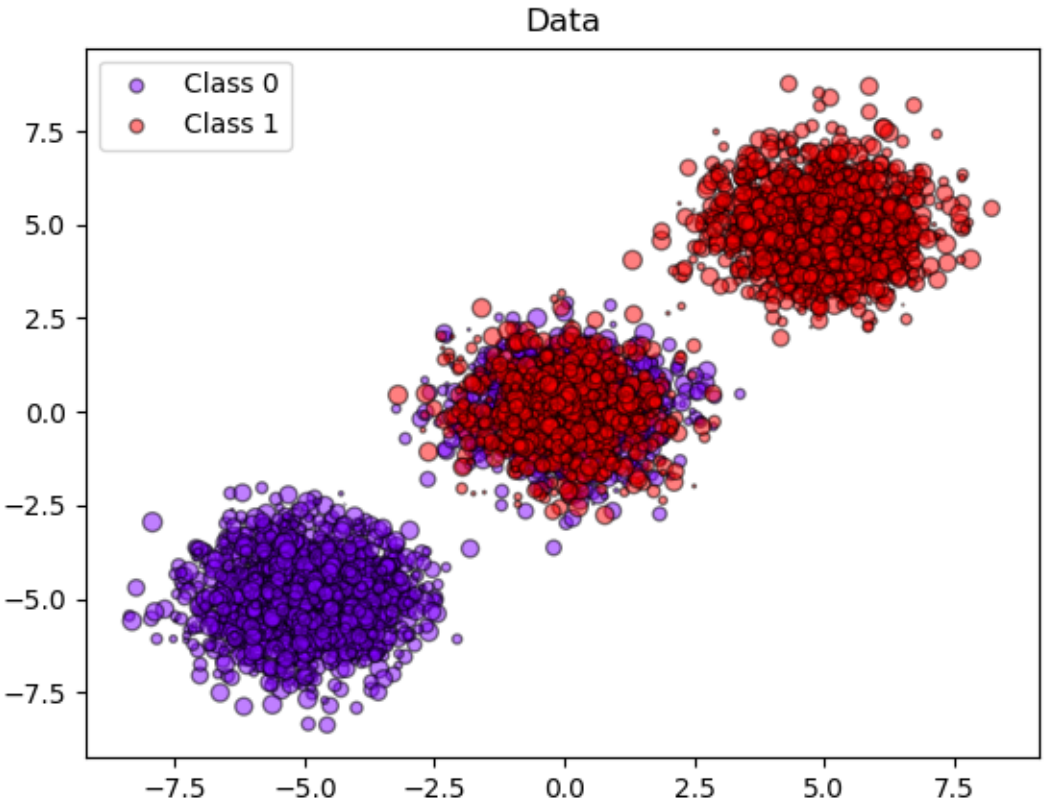
One can observe here that logistic regression is well calibrated as its curve is nearly diagonal. Linear SVC's calibration curve or reliability diagram has a sigmoid curve, which is typical for an under-confident classifier. In the case of LinearSVC, this is caused by the margin property of the hinge loss, which lets the model focus on hard samples that are close to the decision boundary (the support vectors). Both kinds of calibration can fix this issue and yield nearly identical results. The next figure shows the calibration curve of Gaussian naive Bayes on the same data, with both kinds of calibration and also without calibration.

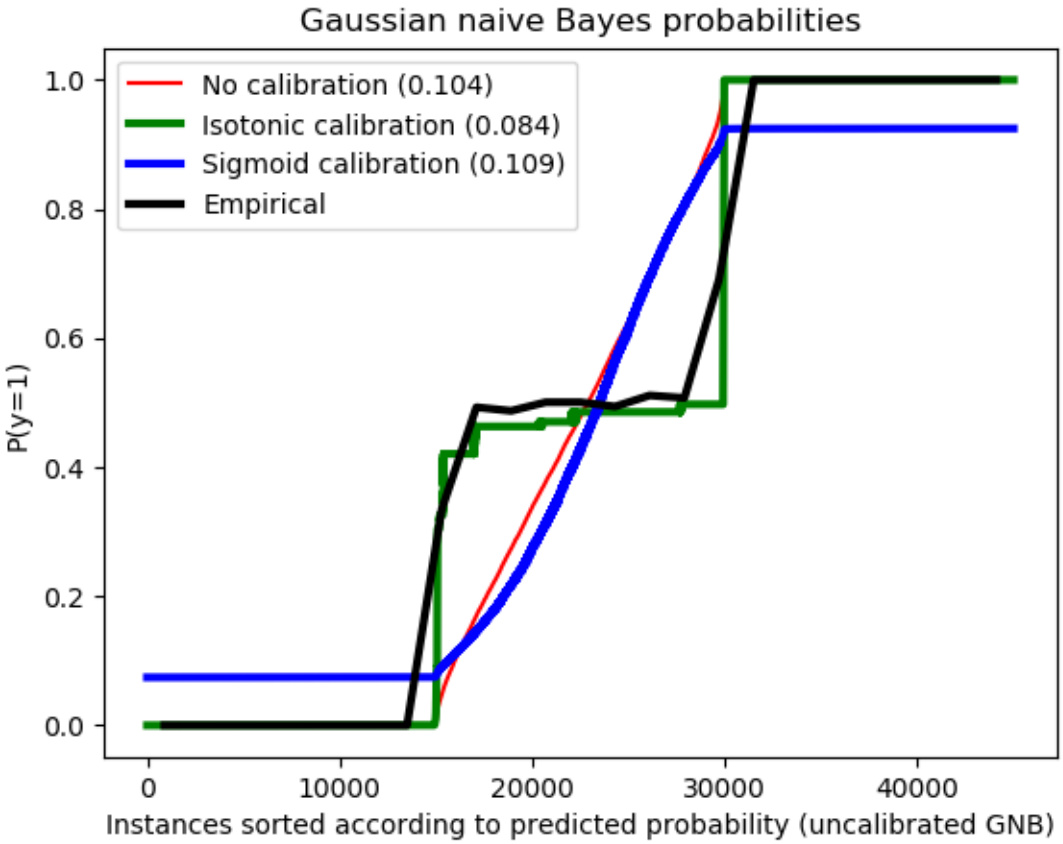
One can see that Gaussian naive Bayes performs very badly but does so in an other way than linear SVC: While linear SVC exhibited a sigmoid calibration curve, Gaussian naive Bayes' calibration curve has a transposed-sigmoid shape. This is typical for an over-confident classifier. In this case, the classifier's overconfidence is caused by the redundant features which violate the naive Bayes assumption of feature-independence.

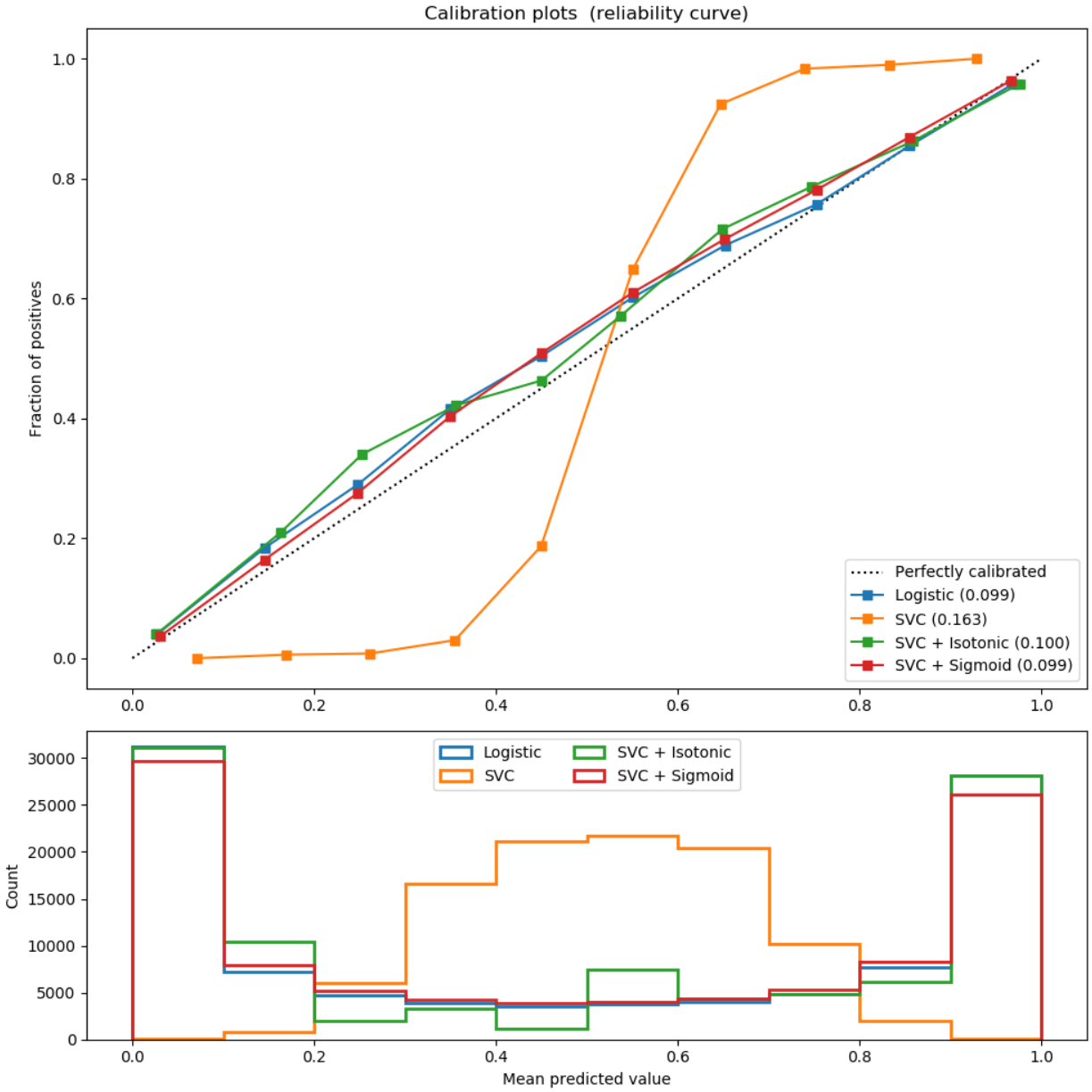
Calibration of the probabilities of Gaussian naive Bayes with isotonic regression can fix this issue as can be seen from the nearly diagonal calibration curve. Sigmoid calibration also improves the brier score slightly, albeit not as strongly as the non-parametric isotonic calibration. This is an intrinsic limitation of sigmoid calibration, whose parametric form

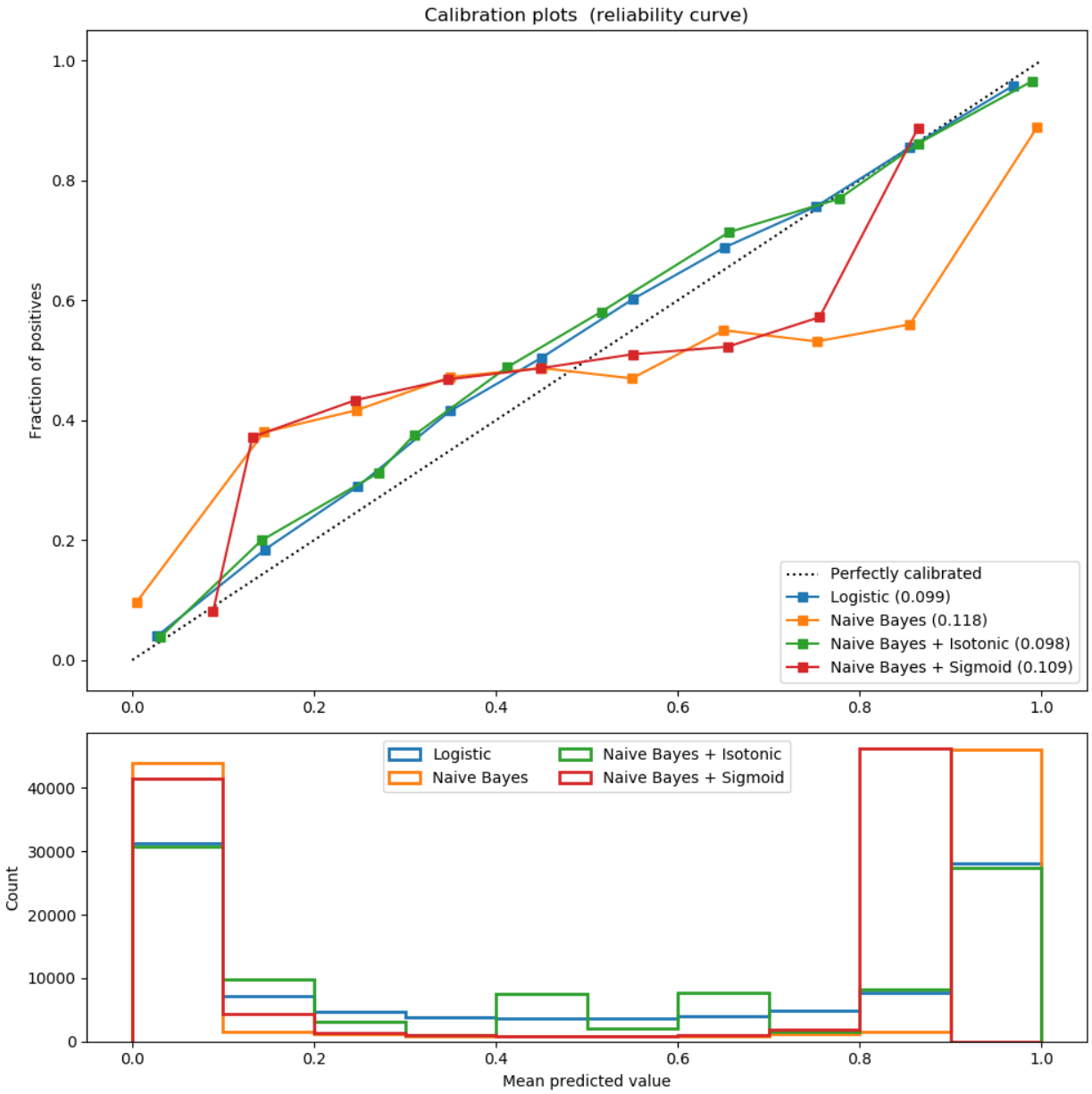
[4] Predicting Good Probabilities with Supervised Learning, A. Niculescu-Mizil & R. Caruana, ICML 2005

[5] On the combination of forecast probabilities for consecutive precipitation periods. Wea. Forecasting, 5, 640–650., Wilks, D. S., 1990a


Data
Class 0
Class 1
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5


Gaussian naive Bayes probabilities
No calibration (0.104)
Isotonic calibration (0.084)
Sigmoid calibration (0.109)
Empirical
P(y=1)
1.0
0.8
0.6
0.4
0.2
0.0
0
10000
20000
30000
40000
Instances sorted according to predicted probability (uncalibrated GNB)

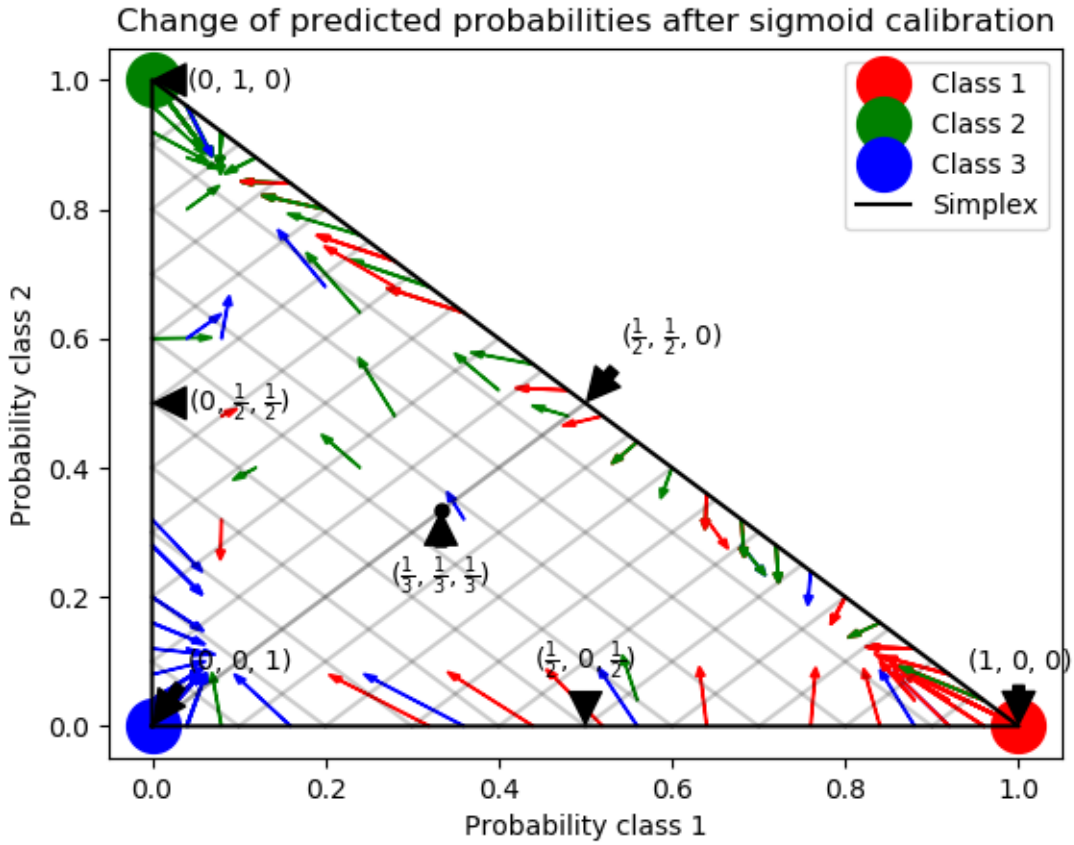

Calibration plots (reliability curve)
Fraction of positives
Perfectly calibrated
Logistic (0.099)
SVC (0.163)
SVC + Isotonic (0.100)
SVC + Sigmoid (0.099)
Logistic
SVC
SVC + Isotonic
SVC + Sigmoid
Count
Mean predicted value


Calibration plots (reliability curve)
Fraction of positives
Perfectly calibrated
Logistic (0.099)
Naive Bayes (0.118)
Naive Bayes + Isotonic (0.098)
Naive Bayes + Sigmoid (0.109)
Logistic
Naive Bayes
Naive Bayes + Isotonic
Naive Bayes + Sigmoid
Count
Mean predicted value

assumes a sigmoid rather than a transposed-sigmoid curve. The non-parametric isotonic calibration model, however, makes no such strong assumptions and can deal with either shape, provided that there is sufficient calibration data. In general, sigmoid calibration is preferable in cases where the calibration curve is sigmoid and where there is limited calibration data, while isotonic calibration is preferable for non-sigmoid calibration curves and in situations where large amounts of data are available for calibration.

`CalibratedClassifierCV` can also deal with classification tasks that involve more than two classes if the base estimator can do so. In this case, the classifier is calibrated first for each class separately in an one-vs-rest fashion. When predicting probabilities for unseen data, the calibrated probabilities for each class are predicted separately. As those probabilities do not necessarily sum to one, a postprocessing is performed to normalize them.

The next image illustrates how sigmoid calibration changes predicted probabilities for a 3-class classification problem. Illustrated is the standard 2-simplex, where the three corners correspond to the three classes. Arrows point from the probability vectors predicted by an uncalibrated classifier to the probability vectors predicted by the same classifier after sigmoid calibration on a hold-out validation set. Colors indicate the true class of an instance (red: class 1, green: class 2, blue: class 3).



The base classifier is a random forest classifier with 25 base estimators (trees). If this classifier is trained on all 800 training datapoints, it is overly confident in its predictions and thus incurs a large log-loss. Calibrating an identical classifier, which was trained on 600 datapoints, with method='sigmoid' on the remaining 200 datapoints reduces the confidence of the predictions, i.e., moves the probability vectors from the edges of the simplex towards the center:

This calibration results in a lower log-loss. Note that an alternative would have been to increase the number of base estimators which would have resulted in a similar decrease in log-loss.

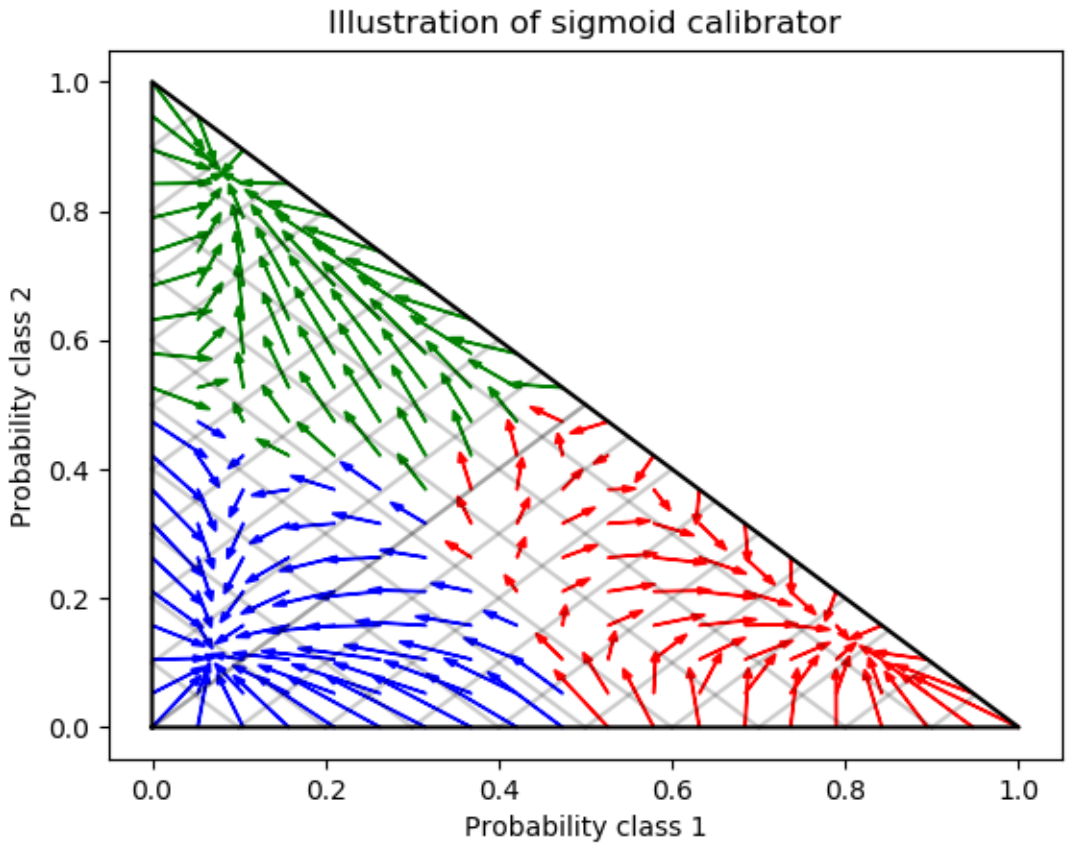


Illustration of sigmoid calibrator
Probability class 2
Probability class 1
0.0
0.2
0.4
0.6
0.8
1.0

References:

- Obtaining calibrated probability estimates from decision trees and naive Bayesian classifiers, B. Zadrozny & C. Elkan, ICML 2001
- Transforming Classifier Scores into Accurate Multiclass Probability Estimates, B. Zadrozny & C. Elkan, (KDD 2002)
- Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods, J. Platt, (1999)

4.1.17 Neural network models (supervised)

Warning: This implementation is not intended for large-scale applications. In particular, scikit-learn offers no GPU support. For much faster, GPU-based implementations, as well as frameworks offering much more flexibility to build deep learning architectures, see *Related Projects*.

Multi-layer Perceptron

Multi-layer Perceptron (MLP) is a supervised learning algorithm that learns a function $f(\cdot) : R^m \rightarrow R^o$ by training on a dataset, where m is the number of dimensions for input and o is the number of dimensions for output. Given a set of features $X = x_1, x_2, ..., x_m$ and a target y, it can learn a non-linear function approximator for either classification or regression. It is different from logistic regression, in that between the input and the output layer, there can be one or more non-linear layers, called hidden layers. Figure 1 shows a one hidden layer MLP with scalar output.

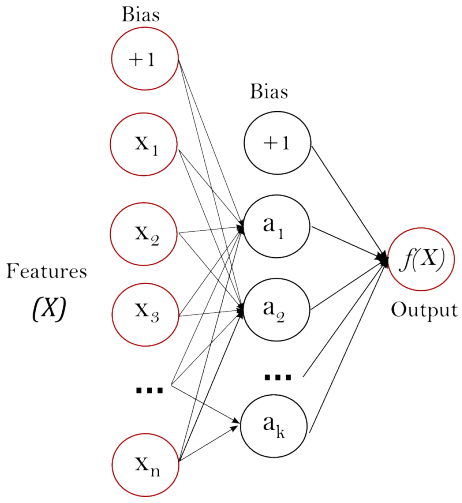


Fig. 2: **Figure 1 : One hidden layer MLP.**

The leftmost layer, known as the input layer, consists of a set of neurons $\{x_i | x_1, x_2, ..., x_m\}$ representing the input features. Each neuron in the hidden layer transforms the values from the previous layer with a weighted linear summation $w_1x_1 + w_2x_2 + ... + w_mx_m$, followed by a non-linear activation function $g(\cdot) : R \rightarrow R$ - like the hyperbolic tan function. The output layer receives the values from the last hidden layer and transforms them into output values.

The module contains the public attributes `coefs_` and `intercepts_`. `coefs_` is a list of weight matrices, where weight matrix at index i represents the weights between layer i and layer $i+1$. `intercepts_` is a list of bias vectors, where the vector at index i represents the bias values added to layer $i + 1$.

The advantages of Multi-layer Perceptron are:

- Capability to learn non-linear models.
- Capability to learn models in real-time (on-line learning) using `partial_fit`.

The disadvantages of Multi-layer Perceptron (MLP) include:

- MLP with hidden layers have a non-convex loss function where there exists more than one local minimum. Therefore different random weight initializations can lead to different validation accuracy.
- MLP requires tuning a number of hyperparameters such as the number of hidden neurons, layers, and iterations.
- MLP is sensitive to feature scaling.

Please see *Tips on Practical Use* section that addresses some of these disadvantages.

Classification

Class *MLPClassifier* implements a multi-layer perceptron (MLP) algorithm that trains using Backpropagation.

MLP trains on two arrays: array X of size (n_samples, n_features), which holds the training samples represented as floating point feature vectors; and array y of size (n_samples,), which holds the target values (class labels) for the training samples:

```
>>> from sklearn.neural_network import MLPClassifier
>>> X = [[0., 0.], [1., 1.]]
>>> y = [0, 1]
>>> clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
...                     hidden_layer_sizes=(5, 2), random_state=1)
...
>>> clf.fit(X, y)
MLPClassifier(alpha=1e-05, hidden_layer_sizes=(5, 2), random_state=1,
              solver='lbfgs')
```

After fitting (training), the model can predict labels for new samples:

```
>>> clf.predict([[2., 2.], [-1., -2.]])
array([1, 0])
```

MLP can fit a non-linear model to the training data. `clf.coefs_` contains the weight matrices that constitute the model parameters:

```
>>> [coef.shape for coef in clf.coefs_]
[(2, 5), (5, 2), (2, 1)]
```

Currently, *MLPClassifier* supports only the Cross-Entropy loss function, which allows probability estimates by running the `predict_proba` method.

MLP trains using Backpropagation. More precisely, it trains using some form of gradient descent and the gradients are calculated using Backpropagation. For classification, it minimizes the Cross-Entropy loss function, giving a vector of probability estimates $P(y|x)$ per sample x:

```
>>> clf.predict_proba([[2., 2.], [1., 2.]])
array([[1.967...e-04, 9.998...-01],
       [1.967...e-04, 9.998...-01]])
```

MLPClassifier supports multi-class classification by applying Softmax as the output function.

Further, the model supports *multi-label classification* in which a sample can belong to more than one class. For each class, the raw output passes through the logistic function. Values larger or equal to `0.5` are rounded to `1`, otherwise to `0`. For a predicted output of a sample, the indices where the value is `1` represents the assigned classes of that sample:

```
>>> X = [[0., 0.], [1., 1.]]
>>> y = [[0, 1], [1, 1]]
>>> clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
...                     hidden_layer_sizes=(15,), random_state=1)
...
>>> clf.fit(X, y)
MLPClassifier(alpha=1e-05, hidden_layer_sizes=(15,), random_state=1,
              solver='lbfgs')
>>> clf.predict([[1., 2.]])
array([[1, 1]])
>>> clf.predict([[0., 0.]])
array([[0, 1]])
```

See the examples below and the docstring of *`MLPClassifier.fit`* for further information.

Examples:

- *Compare Stochastic learning strategies for MLPClassifier*
- *Visualization of MLP weights on MNIST*

Regression

Class *`MLPRegressor`* implements a multi-layer perceptron (MLP) that trains using backpropagation with no activation function in the output layer, which can also be seen as using the identity function as activation function. Therefore, it uses the square error as the loss function, and the output is a set of continuous values.

`MLPRegressor` also supports multi-output regression, in which a sample can have more than one target.

Regularization

Both *`MLPRegressor`* and *`MLPClassifier`* use parameter `alpha` for regularization (L2 regularization) term which helps in avoiding overfitting by penalizing weights with large magnitudes. Following plot displays varying decision function with value of alpha.

See the examples below for further information.

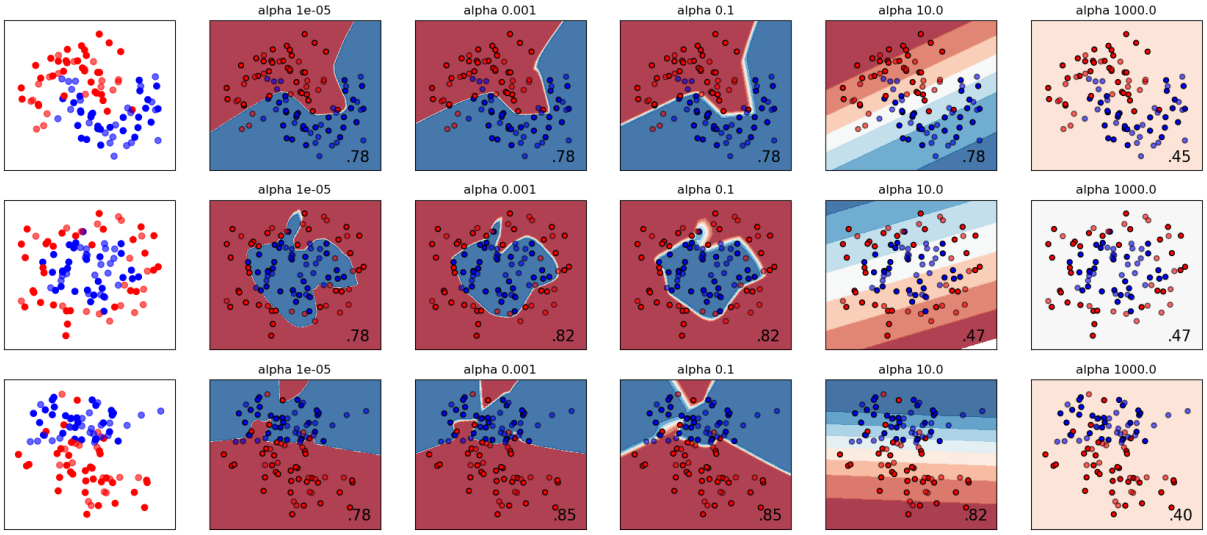
Examples:

- *Varying regularization in Multi-layer Perceptron*

Algorithms

MLP trains using Stochastic Gradient Descent, Adam, or L-BFGS. Stochastic Gradient Descent (SGD) updates parameters using the gradient of the loss function with respect to a parameter that needs adaptation, i.e.

$$w \leftarrow w - \eta(\alpha\frac{\partial R(w)}{\partial w} + \frac{\partial Loss}{\partial w})$$



where η is the learning rate which controls the step-size in the parameter space search. $Loss$ is the loss function used for the network.

More details can be found in the documentation of SGD

Adam is similar to SGD in a sense that it is a stochastic optimizer, but it can automatically adjust the amount to update parameters based on adaptive estimates of lower-order moments.

With SGD or Adam, training supports online and mini-batch learning.

L-BFGS is a solver that approximates the Hessian matrix which represents the second-order partial derivative of a function. Further it approximates the inverse of the Hessian matrix to perform parameter updates. The implementation uses the Scipy version of L-BFGS.

If the selected solver is 'L-BFGS', training does not support online nor mini-batch learning.

Complexity

Suppose there are n training samples, m features, k hidden layers, each containing h neurons - for simplicity, and o output neurons. The time complexity of backpropagation is $O(n \cdot m \cdot h^k \cdot o \cdot i)$, where i is the number of iterations. Since backpropagation has a high time complexity, it is advisable to start with smaller number of hidden neurons and few hidden layers for training.

Mathematical formulation

Given a set of training examples $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ where $x_i \in \mathbf{R}^n$ and $y_i \in \{0, 1\}$, a one hidden layer one hidden neuron MLP learns the function $f(x) = W_2 g(W_1^T x + b_1) + b_2$ where $W_1 \in \mathbf{R}^m$ and $W_2, b_1, b_2 \in \mathbf{R}$ are model parameters. W_1, W_2 represent the weights of the input layer and hidden layer, respectively; and b_1, b_2 represent the bias added to the hidden layer and the output layer, respectively. $g(\cdot) : R \rightarrow R$ is the activation function, set by default as the hyperbolic tan. It is given as,

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

For binary classification, $f(x)$ passes through the logistic function $g(z) = 1/(1+e^{-z})$ to obtain output values between zero and one. A threshold, set to 0.5, would assign samples of outputs larger or equal 0.5 to the positive class, and the rest to the negative class.

If there are more than two classes, $f(x)$ itself would be a vector of size (n_classes,). Instead of passing through logistic function, it passes through the softmax function, which is written as,

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{l=1}^{k} \exp(z_l)}$$

where z_i represents the i th element of the input to softmax, which corresponds to class i, and K is the number of classes. The result is a vector containing the probabilities that sample x belong to each class. The output is the class with the highest probability.

In regression, the output remains as $f(x)$; therefore, output activation function is just the identity function.

MLP uses different loss functions depending on the problem type. The loss function for classification is Cross-Entropy, which in binary case is given as,

$$Loss(\hat{y}, y, W) = -y \ln \hat{y} - (1-y) \ln (1-\hat{y}) + \alpha ||W||_2^2$$

where $\alpha ||W||_2^2$ is an L2-regularization term (aka penalty) that penalizes complex models; and $\alpha > 0$ is a non-negative hyperparameter that controls the magnitude of the penalty.

For regression, MLP uses the Square Error loss function; written as,

$$Loss(\hat{y}, y, W) = \frac{1}{2} ||\hat{y} - y||_2^2 + \frac{\alpha}{2} ||W||_2^2$$

Starting from initial random weights, multi-layer perceptron (MLP) minimizes the loss function by repeatedly updating these weights. After computing the loss, a backward pass propagates it from the output layer to the previous layers, providing each weight parameter with an update value meant to decrease the loss.

In gradient descent, the gradient $\nabla Loss_W$ of the loss with respect to the weights is computed and deducted from W. More formally, this is expressed as,

$$W^{i+1} = W^i - \epsilon \nabla Loss_W^i$$

where i is the iteration step, and ϵ is the learning rate with a value larger than 0.

The algorithm stops when it reaches a preset maximum number of iterations; or when the improvement in loss is below a certain, small number.

Tips on Practical Use

- Multi-layer Perceptron is sensitive to feature scaling, so it is highly recommended to scale your data. For example, scale each attribute on the input vector X to [0, 1] or [-1, +1], or standardize it to have mean 0 and variance 1. Note that you must apply the *same* scaling to the test set for meaningful results. You can use `StandardScaler` for standardization.

```
>>> from sklearn.preprocessing import StandardScaler  # doctest: +SKIP
>>> scaler = StandardScaler()  # doctest: +SKIP
>>> # Don't cheat - fit only on training data
>>> scaler.fit(X_train)  # doctest: +SKIP
>>> X_train = scaler.transform(X_train)  # doctest: +SKIP
>>> # apply same transformation to test data
>>> X_test = scaler.transform(X_test)  # doctest: +SKIP
```

  An alternative and recommended approach is to use `StandardScaler` in a `Pipeline`

- Finding a reasonable regularization parameter α is best done using `GridSearchCV`, usually in the range `10.0 ** -np.arange(1, 7)`.
- Empirically, we observed that `L-BFGS` converges faster and with better solutions on small datasets. For relatively large datasets, however, `Adam` is very robust. It usually converges quickly and gives pretty good performance. `SGD` with momentum or nesterov's momentum, on the other hand, can perform better than those two algorithms if learning rate is correctly tuned.

More control with warm_start

If you want more control over stopping criteria or learning rate in SGD, or want to do additional monitoring, using `warm_start=True` and `max_iter=1` and iterating yourself can be helpful:

```
>>> X = [[0., 0.], [1., 1.]]
>>> y = [0, 1]
>>> clf = MLPClassifier(hidden_layer_sizes=(15,), random_state=1, max_iter=1, warm_
↪start=True)
>>> for i in range(10):
...     clf.fit(X, y)
...     # additional monitoring / inspection
MLPClassifier(...
```

References:

- "Learning representations by back-propagating errors." Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams.
- "Stochastic Gradient Descent" L. Bottou - Website, 2010.
- "Backpropagation" Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen - Website, 2011.
- "Efficient BackProp" Y. LeCun, L. Bottou, G. Orr, K. Müller - In Neural Networks: Tricks of the Trade 1998.
- "Adam: A method for stochastic optimization." Kingma, Diederik, and Jimmy Ba. arXiv preprint arXiv:1412.6980 (2014).

4.2 Unsupervised learning

4.2.1 Gaussian mixture models

`sklearn.mixture` is a package which enables one to learn Gaussian Mixture Models (diagonal, spherical, tied and full covariance matrices supported), sample them, and estimate them from data. Facilities to help determine the appropriate number of components are also provided.

A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. One can think of mixture models as generalizing k-means clustering to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussians.

Scikit-learn implements different classes to estimate Gaussian mixture models, that correspond to different estimation strategies, detailed below.

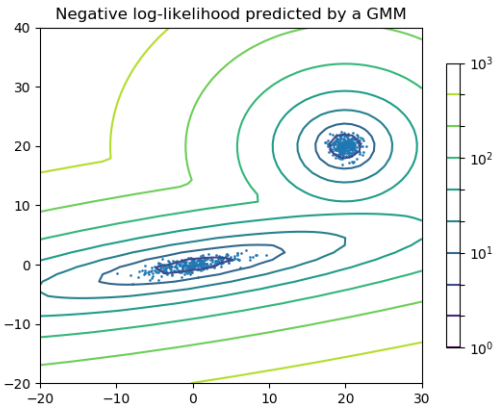


Fig. 3: **Two-component Gaussian mixture model:** *data points, and equi-probability surfaces of the model.*

Gaussian Mixture

The `GaussianMixture` object implements the *expectation-maximization* (EM) algorithm for fitting mixture-of-Gaussian models. It can also draw confidence ellipsoids for multivariate models, and compute the Bayesian Information Criterion to assess the number of clusters in the data. A `GaussianMixture.fit` method is provided that learns a Gaussian Mixture Model from train data. Given test data, it can assign to each sample the Gaussian it mostly probably belong to using the `GaussianMixture.predict` method.

The `GaussianMixture` comes with different options to constrain the covariance of the difference classes estimated: spherical, diagonal, tied or full covariance.

Examples:

- See *GMM covariances* for an example of using the Gaussian mixture as clustering on the iris dataset.
- See *Density Estimation for a Gaussian mixture* for an example on plotting the density estimation.

Pros and cons of class `GaussianMixture`

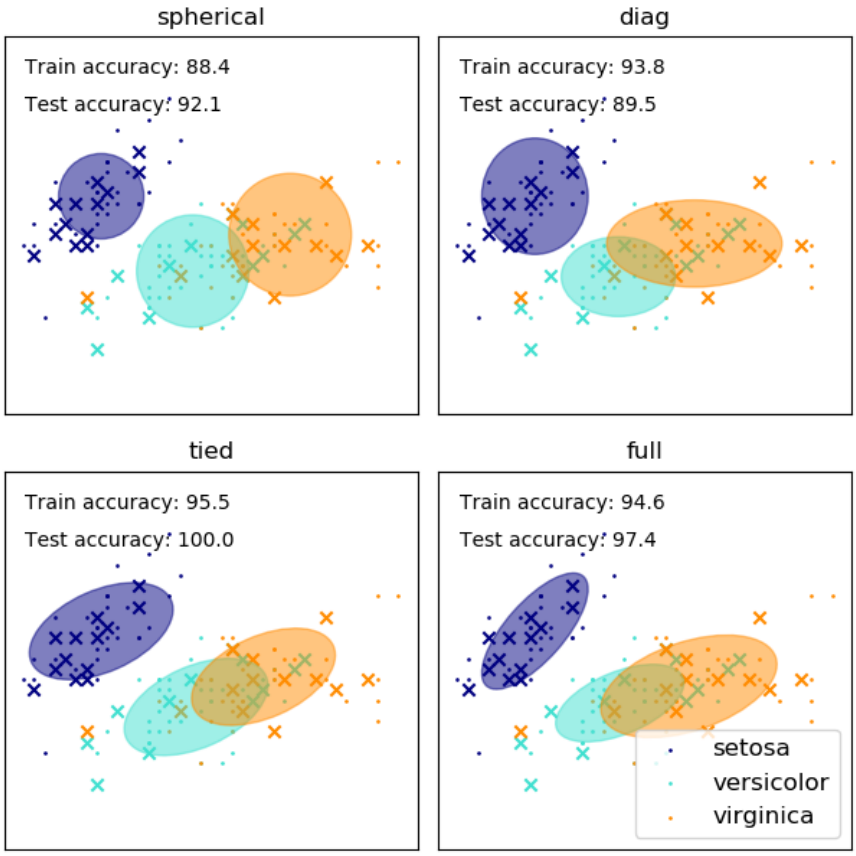
Pros

Speed It is the fastest algorithm for learning mixture models

Agnostic As this algorithm maximizes only the likelihood, it will not bias the means towards zero, or bias the cluster sizes to have specific structures that might or might not apply.

Cons

Singularities When one has insufficiently many points per mixture, estimating the covariance matrices becomes difficult, and the algorithm is known to diverge and find solutions with infinite likelihood unless one regularizes the covariances artificially.



Number of components This algorithm will always use all the components it has access to, needing held-out data or information theoretical criteria to decide how many components to use in the absence of external cues.

Selecting the number of components in a classical Gaussian Mixture Model

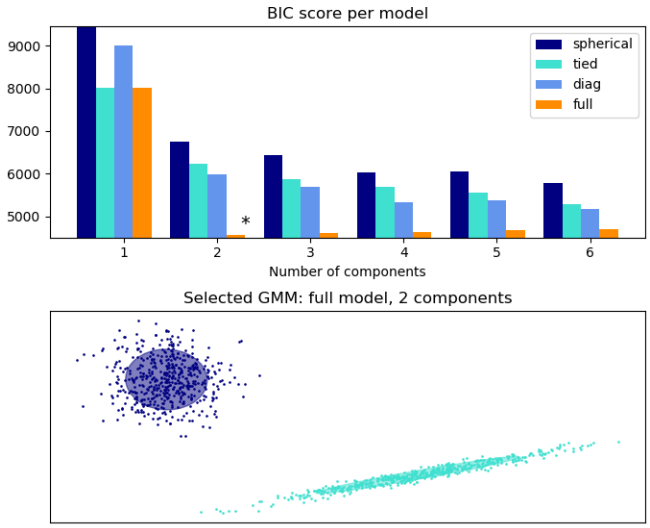
The BIC criterion can be used to select the number of components in a Gaussian Mixture in an efficient way. In theory, it recovers the true number of components only in the asymptotic regime (i.e. if much data is available and assuming that the data was actually generated i.i.d. from a mixture of Gaussian distribution). Note that using a *Variational Bayesian Gaussian mixture* avoids the specification of the number of components for a Gaussian mixture model.

Examples:

- See *Gaussian Mixture Model Selection* for an example of model selection performed with classical Gaussian mixture.

Estimation algorithm Expectation-maximization

The main difficulty in learning Gaussian mixture models from unlabeled data is that it is one usually doesn't know which points came from which latent component (if one has access to this information it gets very easy to fit a separate Gaussian distribution to each set of points). Expectation-maximization is a well-founded statistical algorithm to get around this problem by an iterative process. First one assumes random components (randomly centered on data points,



learned from k-means, or even just normally distributed around the origin) and computes for each point a probability of being generated by each component of the model. Then, one tweaks the parameters to maximize the likelihood of the data given those assignments. Repeating this process is guaranteed to always converge to a local optimum.

Variational Bayesian Gaussian Mixture

The *BayesianGaussianMixture* object implements a variant of the Gaussian mixture model with variational inference algorithms. The API is similar as the one defined by *GaussianMixture*.

Estimation algorithm: variational inference

Variational inference is an extension of expectation-maximization that maximizes a lower bound on model evidence (including priors) instead of data likelihood. The principle behind variational methods is the same as expectation-maximization (that is both are iterative algorithms that alternate between finding the probabilities for each point to be generated by each mixture and fitting the mixture to these assigned points), but variational methods add regularization by integrating information from prior distributions. This avoids the singularities often found in expectation-maximization solutions but introduces some subtle biases to the model. Inference is often notably slower, but not usually as much so as to render usage unpractical.

Due to its Bayesian nature, the variational algorithm needs more hyper- parameters than expectation-maximization, the most important of these being the concentration parameter `weight_concentration_prior`. Specifying a low value for the concentration prior will make the model put most of the weight on few components set the remaining components weights very close to zero. High values of the concentration prior will allow a larger number of components to be active in the mixture.

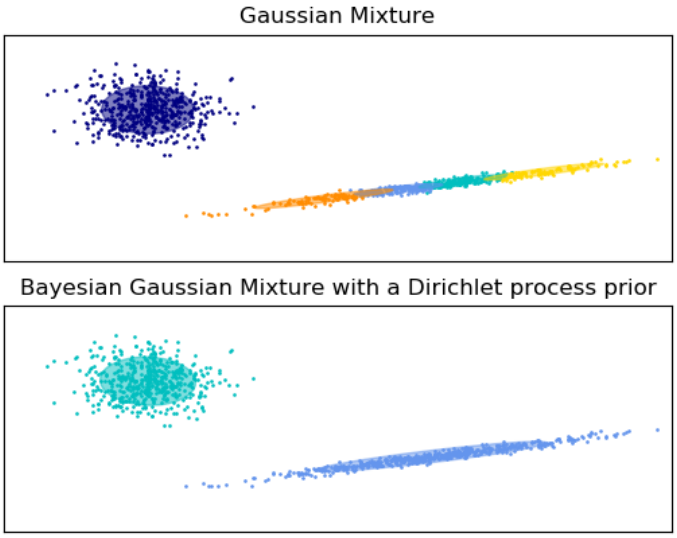
The parameters implementation of the *BayesianGaussianMixture* class proposes two types of prior for the weights distribution: a finite mixture model with Dirichlet distribution and an infinite mixture model with the Dirichlet Process. In practice Dirichlet Process inference algorithm is approximated and uses a truncated distribution with a fixed maximum number of components (called the Stick-breaking representation). The number of components actually used almost always depends on the data.

The next figure compares the results obtained for the different type of the weight concentration prior (parameter `weight_concentration_prior_type`) for different values of `weight_concentration_prior`. Here, we can see the value of the `weight_concentration_prior` parameter has a strong impact on the effective

number of active components obtained. We can also notice that large values for the concentration weight prior lead to more uniform weights when the type of prior is 'dirichlet_distribution' while this is not necessarily the case for the 'dirichlet_process' type (used by default).

The examples below compare Gaussian mixture models with a fixed number of components, to the variational Gaussian mixture models with a Dirichlet process prior. Here, a classical Gaussian mixture is fitted with 5 components on a dataset composed of 2 clusters. We can see that the variational Gaussian mixture with a Dirichlet process prior is able to limit itself to only 2 components whereas the Gaussian mixture fits the data with a fixed number of components that has to be set a priori by the user. In this case the user has selected `n_components=5` which does not match the

true generative distribution of this toy dataset. Note that with very little observations, the variational Gaussian mixture models with a Dirichlet process prior can take a conservative stand, and fit only one component.



On the following figure we are fitting a dataset not well-depicted by a Gaussian mixture. Adjusting the `weight_concentration_prior`, parameter of the *`BayesianGaussianMixture`* controls the number of components used to fit this data. We also present on the last two plots a random sampling generated from the two resulting mixtures.

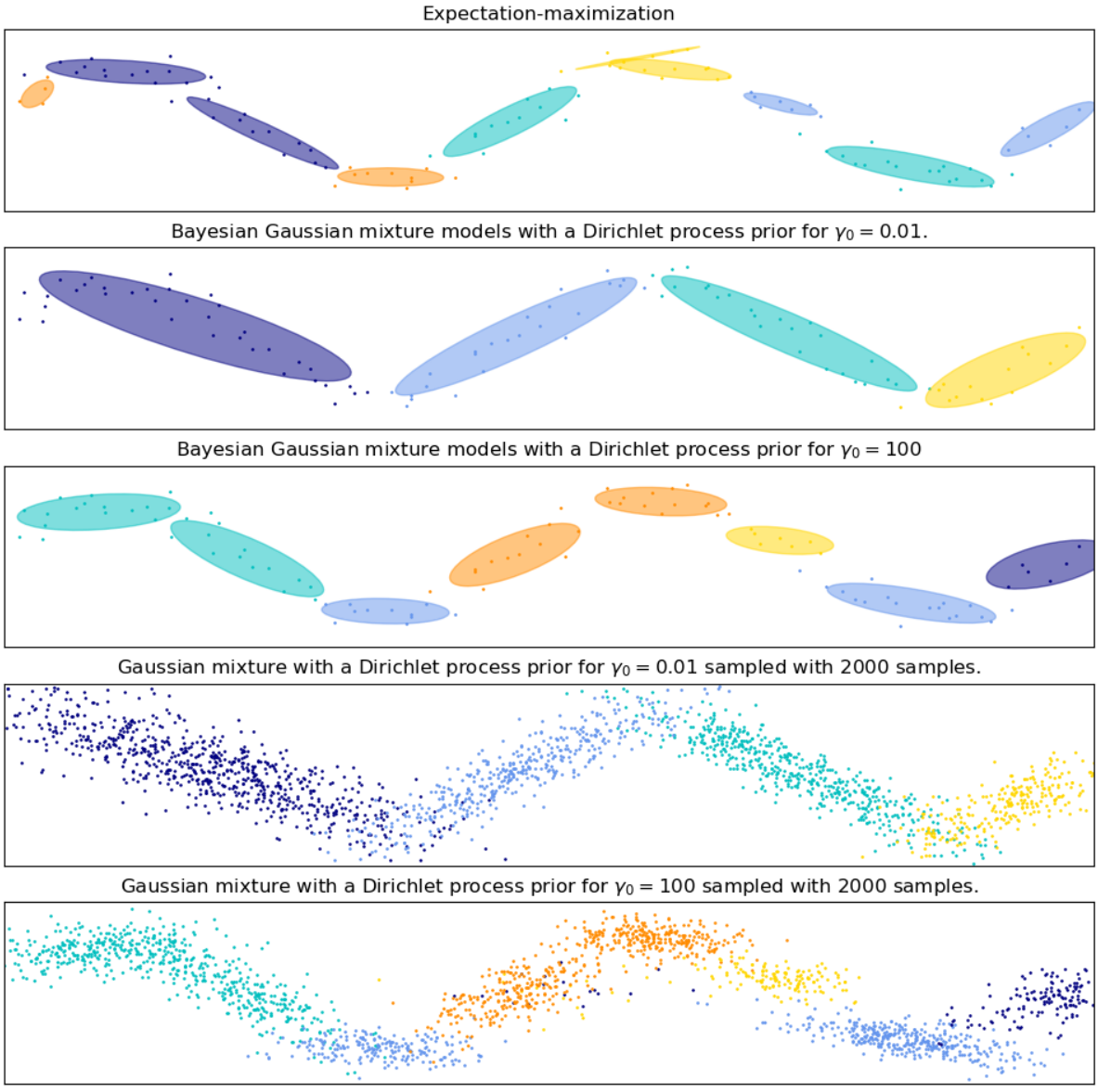
Examples:

- See *Gaussian Mixture Model Ellipsoids* for an example on plotting the confidence ellipsoids for both *`GaussianMixture`* and *`BayesianGaussianMixture`*.
- *Gaussian Mixture Model Sine Curve* shows using *`GaussianMixture`* and *`BayesianGaussianMixture`* to fit a sine wave.
- See *Concentration Prior Type Analysis of Variation Bayesian Gaussian Mixture* for an example plotting the confidence ellipsoids for the *`BayesianGaussianMixture`* with different `weight_concentration_prior_type` for different values of the parameter `weight_concentration_prior`.

Pros and cons of variational inference with `BayesianGaussianMixture`

Pros

Automatic selection when `weight_concentration_prior` is small enough and `n_components` is larger than what is found necessary by the model, the Variational Bayesian mixture model has a natural tendency to set some mixture weights values close to zero. This makes it possible to let the model choose a suitable number of effective components automatically. Only an upper bound of this number needs to be provided. Note however that the "ideal" number of active components is very application specific and is typically ill-defined in a data exploration setting.


Expectation-maximization
Bayesian Gaussian mixture models with a Dirichlet process prior for $\gamma_0 = 0.01$.
Bayesian Gaussian mixture models with a Dirichlet process prior for $\gamma_0 = 100$
Gaussian mixture with a Dirichlet process prior for $\gamma_0 = 0.01$ sampled with 2000 samples.
Gaussian mixture with a Dirichlet process prior for $\gamma_0 = 100$ sampled with 2000 samples.

Less sensitivity to the number of parameters unlike finite models, which will almost always use all components as much as they can, and hence will produce wildly different solutions for different numbers of components, the variational inference with a Dirichlet process prior (`weight_concentration_prior_type='dirichlet_process'`) won't change much with changes to the parameters, leading to more stability and less tuning.

Regularization due to the incorporation of prior information, variational solutions have less pathological special cases than expectation-maximization solutions.

Cons

Speed the extra parametrization necessary for variational inference make inference slower, although not by much.

Hyperparameters this algorithm needs an extra hyperparameter that might need experimental tuning via cross-validation.

Bias there are many implicit biases in the inference algorithms (and also in the Dirichlet process if used), and whenever there is a mismatch between these biases and the data it might be possible to fit better models using a finite mixture.

The Dirichlet Process

Here we describe variational inference algorithms on Dirichlet process mixture. The Dirichlet process is a prior probability distribution on *clusterings with an infinite, unbounded, number of partitions*. Variational techniques let us incorporate this prior structure on Gaussian mixture models at almost no penalty in inference time, comparing with a finite Gaussian mixture model.

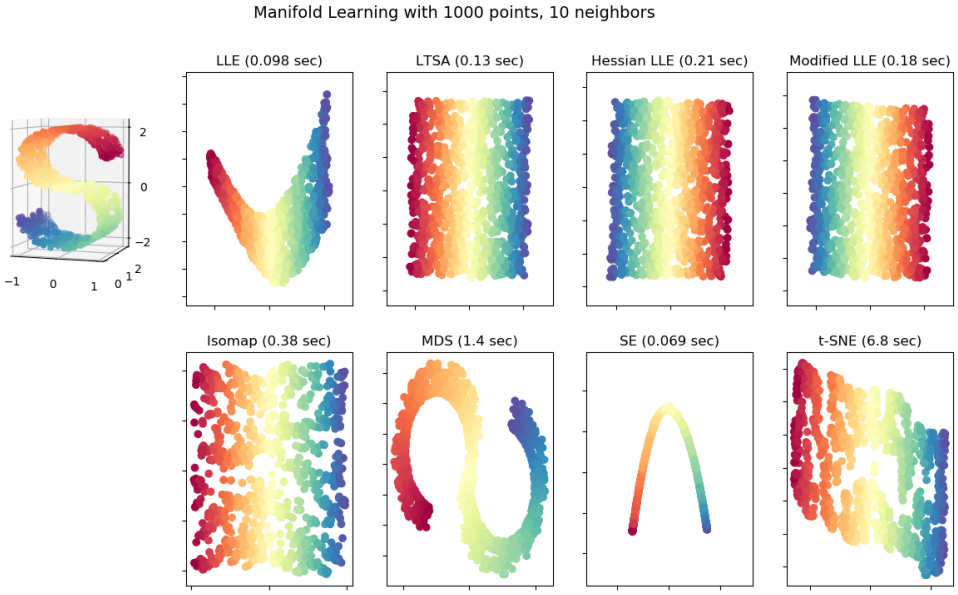
An important question is how can the Dirichlet process use an infinite, unbounded number of clusters and still be consistent. While a full explanation doesn't fit this manual, one can think of its stick breaking process analogy to help understanding it. The stick breaking process is a generative story for the Dirichlet process. We start with a unit-length stick and in each step we break off a portion of the remaining stick. Each time, we associate the length of the piece of the stick to the proportion of points that falls into a group of the mixture. At the end, to represent the infinite mixture, we associate the last remaining piece of the stick to the proportion of points that don't fall into all the other groups. The length of each piece is a random variable with probability proportional to the concentration parameter. Smaller value of the concentration will divide the unit-length into larger pieces of the stick (defining more concentrated distribution). Larger concentration values will create smaller pieces of the stick (increasing the number of components with non zero weights).

Variational inference techniques for the Dirichlet process still work with a finite approximation to this infinite mixture model, but instead of having to specify a priori how many components one wants to use, one just specifies the concentration parameter and an upper bound on the number of mixture components (this upper bound, assuming it is higher than the "true" number of components, affects only algorithmic complexity, not the actual number of components used).

4.2.2 Manifold learning

Look for the bare necessities
The simple bare necessities
Forget about your worries and your strife
I mean the bare necessities
Old Mother Nature's recipes
That bring the bare necessities of life

– Baloo's song [The Jungle Book]

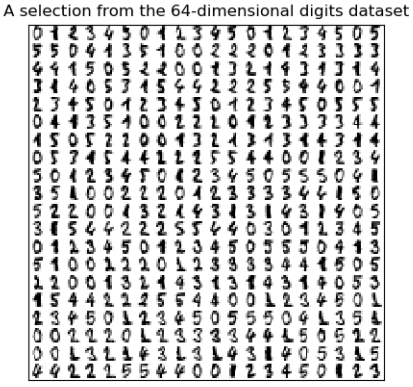


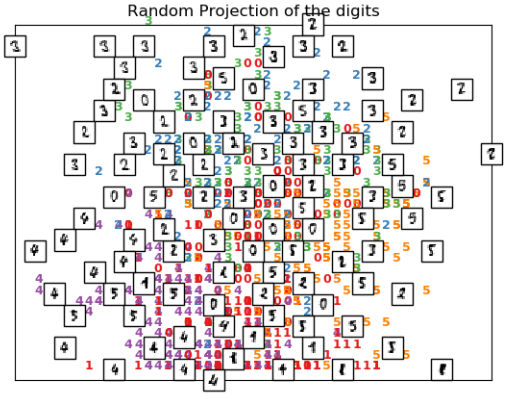
Manifold learning is an approach to non-linear dimensionality reduction. Algorithms for this task are based on the idea that the dimensionality of many data sets is only artificially high.

Introduction

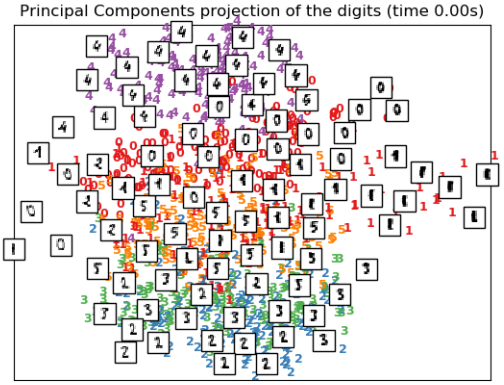
High-dimensional datasets can be very difficult to visualize. While data in two or three dimensions can be plotted to show the inherent structure of the data, equivalent high-dimensional plots are much less intuitive. To aid visualization of the structure of a dataset, the dimension must be reduced in some way.

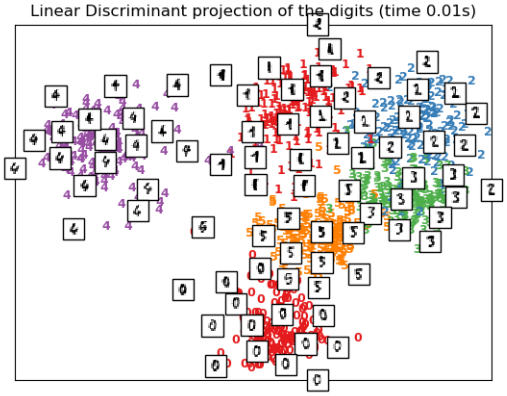
The simplest way to accomplish this dimensionality reduction is by taking a random projection of the data. Though this allows some degree of visualization of the data structure, the randomness of the choice leaves much to be desired. In a random projection, it is likely that the more interesting structure within the data will be lost.





To address this concern, a number of supervised and unsupervised linear dimensionality reduction frameworks have been designed, such as Principal Component Analysis (PCA), Independent Component Analysis, Linear Discriminant Analysis, and others. These algorithms define specific rubrics to choose an "interesting" linear projection of the data. These methods can be powerful, but often miss important non-linear structure in the data.





Manifold Learning can be thought of as an attempt to generalize linear frameworks like PCA to be sensitive to non-linear structure in data. Though supervised variants exist, the typical manifold learning problem is unsupervised: it learns the high-dimensional structure of the data from the data itself, without the use of predetermined classifications.

Examples:

- See *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap…* for an example of dimensionality reduction on handwritten digits.
- See *Comparison of Manifold Learning methods* for an example of dimensionality reduction on a toy "S-curve" dataset.

The manifold learning implementations available in scikit-learn are summarized below

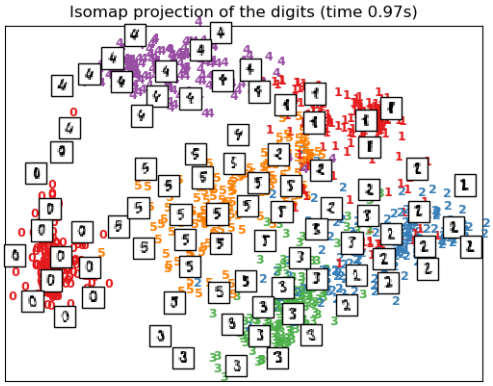
Isomap

One of the earliest approaches to manifold learning is the Isomap algorithm, short for Isometric Mapping. Isomap can be viewed as an extension of Multi-dimensional Scaling (MDS) or Kernel PCA. Isomap seeks a lower-dimensional embedding which maintains geodesic distances between all points. Isomap can be performed with the object `Isomap`.

Complexity

The Isomap algorithm comprises three stages:

1. **Nearest neighbor search.** Isomap uses `sklearn.neighbors.BallTree` for efficient neighbor search. The cost is approximately $O[D \log(k) N \log(N)]$, for k nearest neighbors of N points in D dimensions.
2. **Shortest-path graph search.** The most efficient known algorithms for this are *Dijkstra's Algorithm*, which is approximately $O[N^2(k + \log(N))]$, or the *Floyd-Warshall algorithm*, which is $O[N^3]$. The algorithm can be selected by the user with the `path_method` keyword of `Isomap`. If unspecified, the code attempts to choose the best algorithm for the input data.
3. **Partial eigenvalue decomposition.** The embedding is encoded in the eigenvectors corresponding to the d largest eigenvalues of the $N \times N$ isomap kernel. For a dense solver, the cost is approximately $O[dN^2]$. This



cost can often be improved using the `ARPACK` solver. The eigensolver can be specified by the user with the `path_method` keyword of `Isomap`. If unspecified, the code attempts to choose the best algorithm for the input data.

The overall complexity of Isomap is $O[D \log(k) N \log(N)] + O[N^2(k + \log(N))] + O[dN^2]$.

- N : number of training data points
- D : input dimension
- k : number of nearest neighbors
- d : output dimension

References:

- "A global geometric framework for nonlinear dimensionality reduction" Tenenbaum, J.B.; De Silva, V.; & Langford, J.C. Science 290 (5500)

Locally Linear Embedding

Locally linear embedding (LLE) seeks a lower-dimensional projection of the data which preserves distances within local neighborhoods. It can be thought of as a series of local Principal Component Analyses which are globally compared to find the best non-linear embedding.

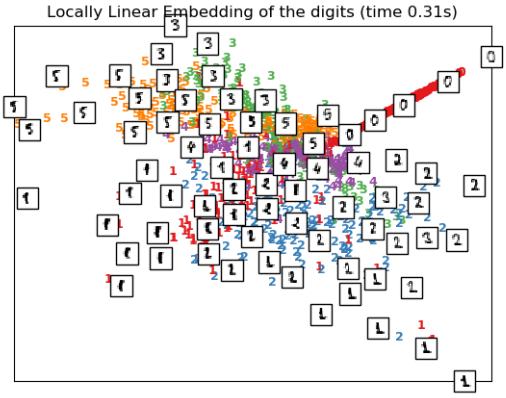
Locally linear embedding can be performed with function *`locally_linear_embedding`* or its object-oriented counterpart *`LocallyLinearEmbedding`*.

Complexity

The standard LLE algorithm comprises three stages:

1. **Nearest Neighbors Search**. See discussion under Isomap above.
2. **Weight Matrix Construction**. $O[DNk^3]$. The construction of the LLE weight matrix involves the solution of a $k \times k$ linear equation for each of the N local neighborhoods
3. **Partial Eigenvalue Decomposition**. See discussion under Isomap above.

The overall complexity of standard LLE is $O[D \log(k) N \log(N)] + O[DNk^3] + O[dN^2]$.



- N : number of training data points
- D : input dimension
- k : number of nearest neighbors
- d : output dimension

References:

- "Nonlinear dimensionality reduction by locally linear embedding" Roweis, S. & Saul, L. Science 290:2323 (2000)

Modified Locally Linear Embedding

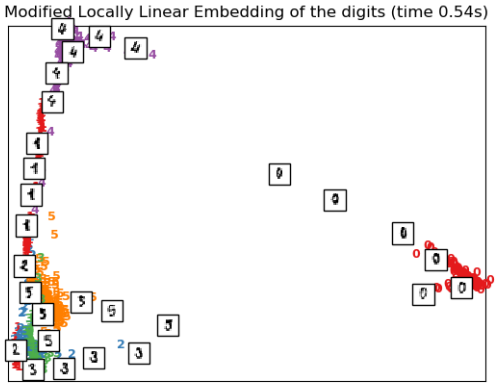
One well-known issue with LLE is the regularization problem. When the number of neighbors is greater than the number of input dimensions, the matrix defining each local neighborhood is rank-deficient. To address this, standard LLE applies an arbitrary regularization parameter r, which is chosen relative to the trace of the local weight matrix. Though it can be shown formally that as $r \to 0$, the solution converges to the desired embedding, there is no guarantee that the optimal solution will be found for $r > 0$. This problem manifests itself in embeddings which distort the underlying geometry of the manifold.

One method to address the regularization problem is to use multiple weight vectors in each neighborhood. This is the essence of *modified locally linear embedding* (MLLE). MLLE can be performed with function `locally_linear_embedding` or its object-oriented counterpart `LocallyLinearEmbedding`, with the keyword `method = 'modified'`. It requires `n_neighbors > n_components`.

Complexity

The MLLE algorithm comprises three stages:

1. **Nearest Neighbors Search**. Same as standard LLE
2. **Weight Matrix Construction**. Approximately $O[DNk^3]+O[N(k-D)k^2]$. The first term is exactly equivalent to that of standard LLE. The second term has to do with constructing the weight matrix from multiple weights. In practice, the added cost of constructing the MLLE weight matrix is relatively small compared to the cost of stages 1 and 3.



3. **Partial Eigenvalue Decomposition**. Same as standard LLE

The overall complexity of MLLE is $O[D\log(k)N\log(N)] + O[DNk^3] + O[N(k-D)k^2] + O[dN^2]$.

- N : number of training data points
- D : input dimension
- k : number of nearest neighbors
- d : output dimension

References:

- "MLLE: Modified Locally Linear Embedding Using Multiple Weights" Zhang, Z. & Wang, J.

Hessian Eigenmapping

Hessian Eigenmapping (also known as Hessian-based LLE: HLLE) is another method of solving the regularization problem of LLE. It revolves around a hessian-based quadratic form at each neighborhood which is used to recover the locally linear structure. Though other implementations note its poor scaling with data size, `sklearn` implements some algorithmic improvements which make its cost comparable to that of other LLE variants for small output dimension. HLLE can be performed with function *`locally_linear_embedding`* or its object-oriented counterpart *`LocallyLinearEmbedding`*, with the keyword `method = 'hessian'`. It requires `n_neighbors > n_components * (n_components + 3) / 2`.

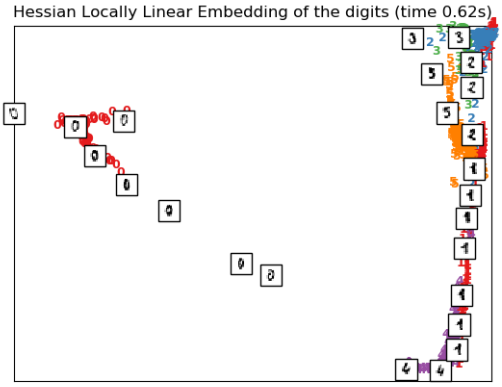
Complexity

The HLLE algorithm comprises three stages:

1. **Nearest Neighbors Search**. Same as standard LLE
2. **Weight Matrix Construction**. Approximately $O[DNk^3] + O[Nd^6]$. The first term reflects a similar cost to that of standard LLE. The second term comes from a QR decomposition of the local hessian estimator.
3. **Partial Eigenvalue Decomposition**. Same as standard LLE

The overall complexity of standard HLLE is $O[D\log(k)N\log(N)] + O[DNk^3] + O[Nd^6] + O[dN^2]$.

- N : number of training data points



- D : input dimension
- k : number of nearest neighbors
- d : output dimension

References:

- "Hessian Eigenmaps: Locally linear embedding techniques for high-dimensional data" Donoho, D. & Grimes, C. Proc Natl Acad Sci USA. 100:5591 (2003)

Spectral Embedding

Spectral Embedding is an approach to calculating a non-linear embedding. Scikit-learn implements Laplacian Eigenmaps, which finds a low dimensional representation of the data using a spectral decomposition of the graph Laplacian. The graph generated can be considered as a discrete approximation of the low dimensional manifold in the high dimensional space. Minimization of a cost function based on the graph ensures that points close to each other on the manifold are mapped close to each other in the low dimensional space, preserving local distances. Spectral embedding can be performed with the function `spectral_embedding` or its object-oriented counterpart `SpectralEmbedding`.

Complexity

The Spectral Embedding (Laplacian Eigenmaps) algorithm comprises three stages:

1. **Weighted Graph Construction**. Transform the raw input data into graph representation using affinity (adjacency) matrix representation.
2. **Graph Laplacian Construction**. unnormalized Graph Laplacian is constructed as $L = D - A$ for and normalized one as $L = D^{-\frac{1}{2}}(D - A)D^{-\frac{1}{2}}$.
3. **Partial Eigenvalue Decomposition**. Eigenvalue decomposition is done on graph Laplacian

The overall complexity of spectral embedding is $O[D \log(k) N \log(N)] + O[DNk^3] + O[dN^2]$.

- N : number of training data points
- D : input dimension
- k : number of nearest neighbors

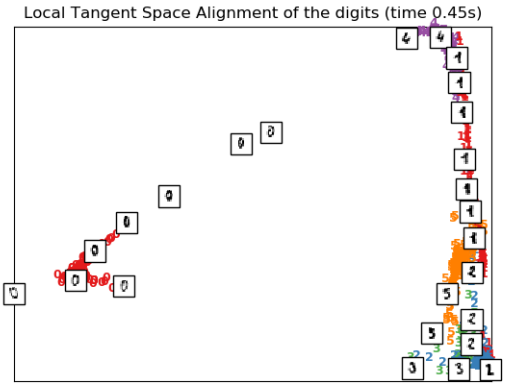
- d : output dimension

References:

- "Laplacian Eigenmaps for Dimensionality Reduction and Data Representation" M. Belkin, P. Niyogi, Neural Computation, June 2003; 15 (6):1373-1396

Local Tangent Space Alignment

Though not technically a variant of LLE, Local tangent space alignment (LTSA) is algorithmically similar enough to LLE that it can be put in this category. Rather than focusing on preserving neighborhood distances as in LLE, LTSA seeks to characterize the local geometry at each neighborhood via its tangent space, and performs a global optimization to align these local tangent spaces to learn the embedding. LTSA can be performed with function *locally_linear_embedding* or its object-oriented counterpart *LocallyLinearEmbedding*, with the keyword `method = 'ltsa'`.



Complexity

The LTSA algorithm comprises three stages:

1. **Nearest Neighbors Search**. Same as standard LLE
2. **Weight Matrix Construction**. Approximately $O[DNk^3] + O[k^2d]$. The first term reflects a similar cost to that of standard LLE.
3. **Partial Eigenvalue Decomposition**. Same as standard LLE

The overall complexity of standard LTSA is $O[D\log(k)N\log(N)] + O[DNk^3] + O[k^2d] + O[dN^2]$.

- N : number of training data points
- D : input dimension
- k : number of nearest neighbors
- d : output dimension

References:

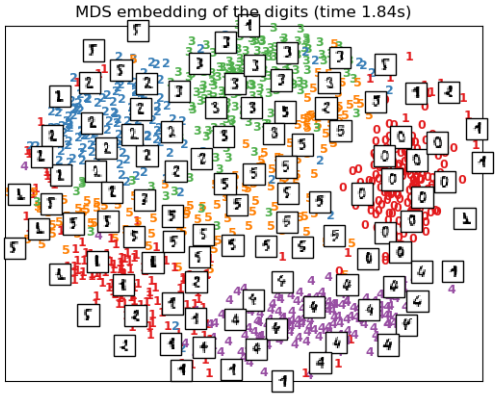
- "Principal manifolds and nonlinear dimensionality reduction via tangent space alignment" Zhang, Z. & Zha, H. Journal of Shanghai Univ. 8:406 (2004)

Multi-dimensional Scaling (MDS)

Multidimensional scaling (*MDS*) seeks a low-dimensional representation of the data in which the distances respect well the distances in the original high-dimensional space.

In general, *MDS* is a technique used for analyzing similarity or dissimilarity data. It attempts to model similarity or dissimilarity data as distances in a geometric spaces. The data can be ratings of similarity between objects, interaction frequencies of molecules, or trade indices between countries.

There exists two types of MDS algorithm: metric and non metric. In the scikit-learn, the class *MDS* implements both. In Metric MDS, the input similarity matrix arises from a metric (and thus respects the triangular inequality), the distances between output two points are then set to be as close as possible to the similarity or dissimilarity data. In the non-metric version, the algorithms will try to preserve the order of the distances, and hence seek for a monotonic relationship between the distances in the embedded space and the similarities/dissimilarities.



Let S be the similarity matrix, and X the coordinates of the n input points. Disparities $\hat{d}_{ij}$ are transformation of the similarities chosen in some optimal ways. The objective, called the stress, is then defined by $\sum_{i<j} d_{ij}(X) - \hat{d}_{ij}(X)$

Metric MDS

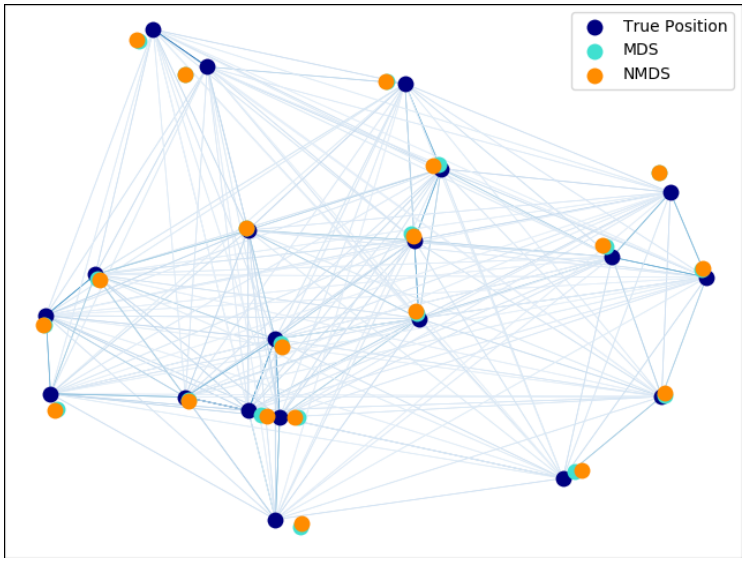
The simplest metric *MDS* model, called *absolute MDS*, disparities are defined by $\hat{d}_{ij} = S_{ij}$. With absolute MDS, the value S_{ij} should then correspond exactly to the distance between point i and j in the embedding point.

Most commonly, disparities are set to $\hat{d}_{ij} = bS_{ij}$.

Nonmetric MDS

Non metric *MDS* focuses on the ordination of the data. If $S_{ij} < S_{jk}$, then the embedding should enforce $d_{ij} < d_{jk}$. A simple algorithm to enforce that is to use a monotonic regression of d_{ij} on S_{ij}, yielding disparities $\hat{d}_{ij}$ in the same order as S_{ij}.

A trivial solution to this problem is to set all the points on the origin. In order to avoid that, the disparities $\hat{d}_{ij}$ are normalized.



References:

- "Modern Multidimensional Scaling - Theory and Applications" Borg, I.; Groenen P. Springer Series in Statistics (1997)
- "Nonmetric multidimensional scaling: a numerical method" Kruskal, J. Psychometrika, 29 (1964)
- "Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis" Kruskal, J. Psychometrika, 29, (1964)

t-distributed Stochastic Neighbor Embedding (t-SNE)

t-SNE (*TSNE*) converts affinities of data points to probabilities. The affinities in the original space are represented by Gaussian joint probabilities and the affinities in the embedded space are represented by Student's t-distributions. This allows t-SNE to be particularly sensitive to local structure and has a few other advantages over existing techniques:

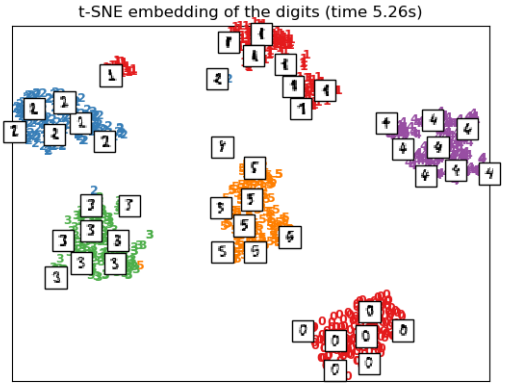
- Revealing the structure at many scales on a single map
- Revealing data that lie in multiple, different, manifolds or clusters
- Reducing the tendency to crowd points together at the center

While Isomap, LLE and variants are best suited to unfold a single continuous low dimensional manifold, t-SNE will focus on the local structure of the data and will tend to extract clustered local groups of samples as highlighted on the S-curve example. This ability to group samples based on the local structure might be beneficial to visually disentangle a dataset that comprises several manifolds at once as is the case in the digits dataset.

The Kullback-Leibler (KL) divergence of the joint probabilities in the original space and the embedded space will be minimized by gradient descent. Note that the KL divergence is not convex, i.e. multiple restarts with different initializations will end up in local minima of the KL divergence. Hence, it is sometimes useful to try different seeds and select the embedding with the lowest KL divergence.

The disadvantages to using t-SNE are roughly:

- t-SNE is computationally expensive, and can take several hours on million-sample datasets where PCA will finish in seconds or minutes
- The Barnes-Hut t-SNE method is limited to two or three dimensional embeddings.
- The algorithm is stochastic and multiple restarts with different seeds can yield different embeddings. However, it is perfectly legitimate to pick the embedding with the least error.
- Global structure is not explicitly preserved. This problem is mitigated by initializing points with PCA (using `init='pca'`).



Optimizing t-SNE

The main purpose of t-SNE is visualization of high-dimensional data. Hence, it works best when the data will be embedded on two or three dimensions.

Optimizing the KL divergence can be a little bit tricky sometimes. There are five parameters that control the optimization of t-SNE and therefore possibly the quality of the resulting embedding:

- perplexity
- early exaggeration factor
- learning rate
- maximum number of iterations
- angle (not used in the exact method)

The perplexity is defined as $k = 2^{(S)}$ where S is the Shannon entropy of the conditional probability distribution. The perplexity of a k-sided die is k, so that k is effectively the number of nearest neighbors t-SNE considers when generating the conditional probabilities. Larger perplexities lead to more nearest neighbors and less sensitive to small structure. Conversely a lower perplexity considers a smaller number of neighbors, and thus ignores more global information in favour of the local neighborhood. As dataset sizes get larger more points will be required to get a reasonable sample of the local neighborhood, and hence larger perplexities may be required. Similarly noisier datasets will require larger perplexity values to encompass enough local neighbors to see beyond the background noise.

The maximum number of iterations is usually high enough and does not need any tuning. The optimization consists of two phases: the early exaggeration phase and the final optimization. During early exaggeration the joint probabilities in the original space will be artificially increased by multiplication with a given factor. Larger factors result in larger gaps between natural clusters in the data. If the factor is too high, the KL divergence could increase during this phase. Usually it does not have to be tuned. A critical parameter is the learning rate. If it is too low gradient descent will get stuck in a bad local minimum. If it is too high the KL divergence will increase during optimization. More tips can be

found in Laurens van der Maaten's FAQ (see references). The last parameter, angle, is a tradeoff between performance and accuracy. Larger angles imply that we can approximate larger regions by a single point, leading to better speed but less accurate results.

"How to Use t-SNE Effectively" provides a good discussion of the effects of the various parameters, as well as interactive plots to explore the effects of different parameters.

Barnes-Hut t-SNE

The Barnes-Hut t-SNE that has been implemented here is usually much slower than other manifold learning algorithms. The optimization is quite difficult and the computation of the gradient is $O[dNlog(N)]$, where d is the number of output dimensions and N is the number of samples. The Barnes-Hut method improves on the exact method where t-SNE complexity is $O[dN^2]$, but has several other notable differences:

- The Barnes-Hut implementation only works when the target dimensionality is 3 or less. The 2D case is typical when building visualizations.
- Barnes-Hut only works with dense input data. Sparse data matrices can only be embedded with the exact method or can be approximated by a dense low rank projection for instance using *sklearn.decomposition.TruncatedSVD*
- Barnes-Hut is an approximation of the exact method. The approximation is parameterized with the angle parameter, therefore the angle parameter is unused when method="exact"
- Barnes-Hut is significantly more scalable. Barnes-Hut can be used to embed hundred of thousands of data points while the exact method can handle thousands of samples before becoming computationally intractable

For visualization purpose (which is the main use case of t-SNE), using the Barnes-Hut method is strongly recommended. The exact t-SNE method is useful for checking the theoretically properties of the embedding possibly in higher dimensional space but limit to small datasets due to computational constraints.

Also note that the digits labels roughly match the natural grouping found by t-SNE while the linear 2D projection of the PCA model yields a representation where label regions largely overlap. This is a strong clue that this data can be well separated by non linear methods that focus on the local structure (e.g. an SVM with a Gaussian RBF kernel). However, failing to visualize well separated homogeneously labeled groups with t-SNE in 2D does not necessarily imply that the data cannot be correctly classified by a supervised model. It might be the case that 2 dimensions are not low enough to accurately represents the internal structure of the data.

References:

- "Visualizing High-Dimensional Data Using t-SNE" van der Maaten, L.J.P.; Hinton, G. Journal of Machine Learning Research (2008)
- "t-Distributed Stochastic Neighbor Embedding" van der Maaten, L.J.P.
- "Accelerating t-SNE using Tree-Based Algorithms." L.J.P. van der Maaten. Journal of Machine Learning Research 15(Oct):3221-3245, 2014.

Tips on practical use

- Make sure the same scale is used over all features. Because manifold learning methods are based on a nearest-neighbor search, the algorithm may perform poorly otherwise. See *StandardScaler* for convenient ways of scaling heterogeneous data.

- The reconstruction error computed by each routine can be used to choose the optimal output dimension. For a d-dimensional manifold embedded in a D-dimensional parameter space, the reconstruction error will decrease as `n_components` is increased until `n_components == d`.
- Note that noisy data can "short-circuit" the manifold, in essence acting as a bridge between parts of the manifold that would otherwise be well-separated. Manifold learning on noisy and/or incomplete data is an active area of research.
- Certain input configurations can lead to singular weight matrices, for example when more than two points in the dataset are identical, or when the data is split into disjointed groups. In this case, `solver='arpack'` will fail to find the null space. The easiest way to address this is to use `solver='dense'` which will work on a singular matrix, though it may be very slow depending on the number of input points. Alternatively, one can attempt to understand the source of the singularity: if it is due to disjoint sets, increasing `n_neighbors` may help. If it is due to identical points in the dataset, removing these points may help.

See also:

Totally Random Trees Embedding can also be useful to derive non-linear representations of feature space, also it does not perform dimensionality reduction.

4.2.3 Clustering

Clustering of unlabeled data can be performed with the module *sklearn.cluster*.

Each clustering algorithm comes in two variants: a class, that implements the `fit` method to learn the clusters on train data, and a function, that, given train data, returns an array of integer labels corresponding to the different clusters. For the class, the labels over the training data can be found in the `labels_` attribute.

Input data

One important thing to note is that the algorithms implemented in this module can take different kinds of matrix as input. All the methods accept standard data matrices of shape `[n_samples, n_features]`. These can be obtained from the classes in the *sklearn.feature_extraction* module. For *AffinityPropagation*, *SpectralClustering* and *DBSCAN* one can also input similarity matrices of shape `[n_samples, n_samples]`. These can be obtained from the functions in the *sklearn.metrics.pairwise* module.

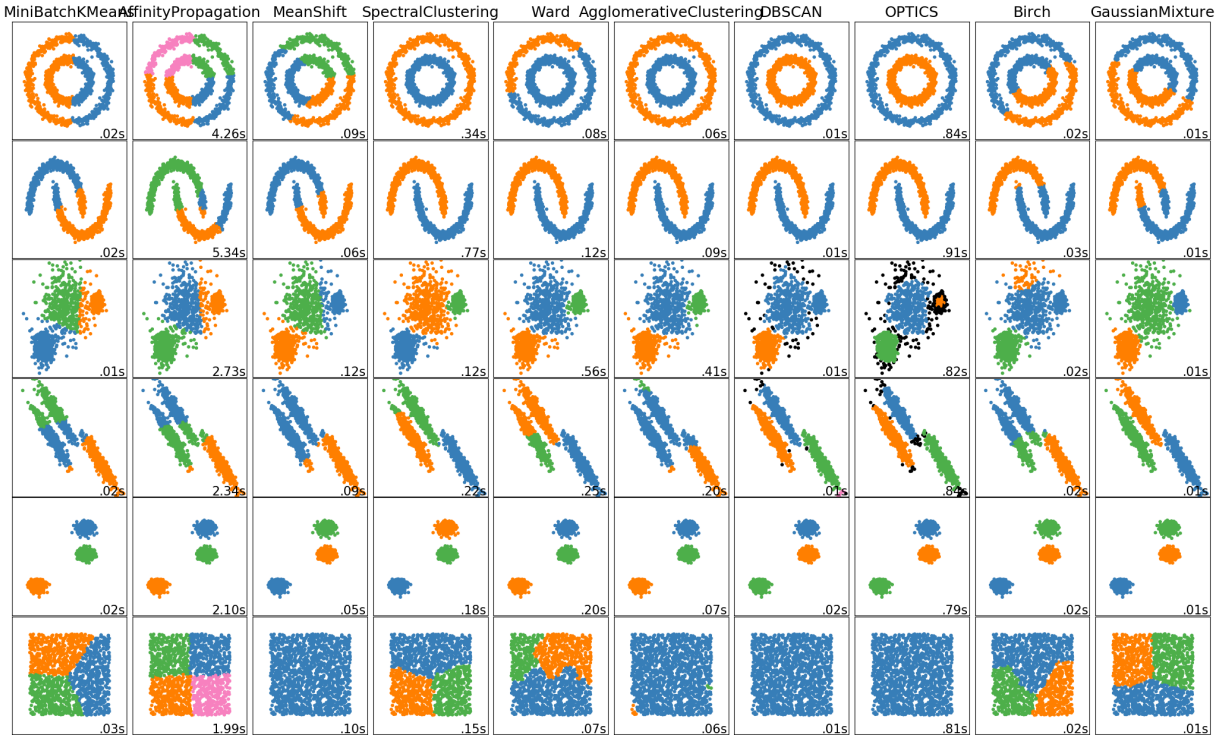


Fig. 4: A comparison of the clustering algorithms in scikit-learn

Overview of clustering methods

Method name	Parameters	Scalability	Usecase	Geometry (metric used)
K-Means	number of clusters	Very large `n_samples`, medium `n_clusters` with *MiniBatch code*	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with `n_samples`	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium `n_samples`, small `n_clusters`	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters or distance threshold	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters or distance threshold, linkage type, distance	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large `n_samples`, medium `n_clusters`	Non-flat geometry, uneven cluster sizes	Distances between nearest points
OPTICS	minimum cluster membership	Very large `n_samples`, large `n_clusters`	Non-flat geometry, uneven cluster sizes, variable cluster density	Distances between points
Gaussian mix	many	Not scalable	Flat geometry, good for den	Mahalanobis dis

Non-flat geometry clustering is useful when the clusters have a specific shape, i.e. a non-flat manifold, and the standard euclidean distance is not the right metric. This case arises in the two top rows of the figure above.

Gaussian mixture models, useful for clustering, are described in *another chapter of the documentation* dedicated to mixture models. KMeans can be seen as a special case of Gaussian mixture model with equal covariance per component.

K-means

The `KMeans` algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the *inertia* or within-cluster sum-of-squares (see below). This algorithm requires the number of clusters to be specified. It scales well to large number of samples and has been used across a large range of application areas in many different fields.

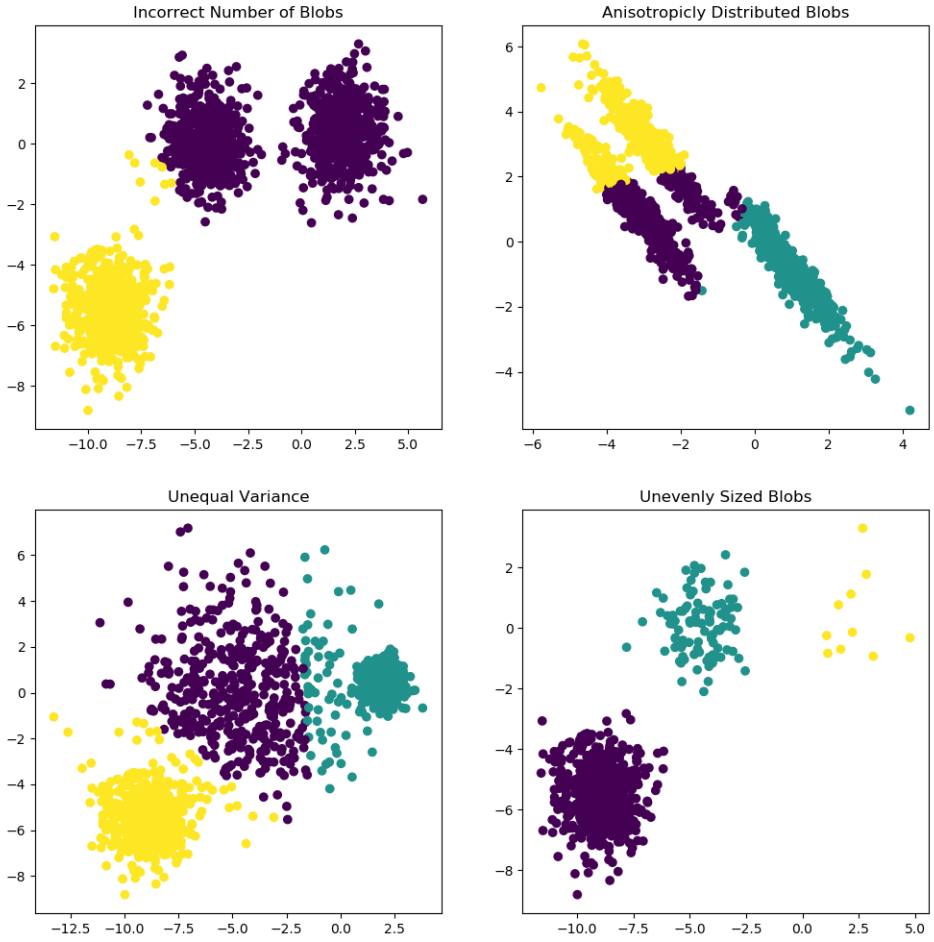
The k-means algorithm divides a set of N samples X into K disjoint clusters C, each described by the mean μ_j of the samples in the cluster. The means are commonly called the cluster "centroids"; note that they are not, in general, points from X, although they live in the same space.

The K-means algorithm aims to choose centroids that minimise the **inertia**, or **within-cluster sum-of-squares criterion**:

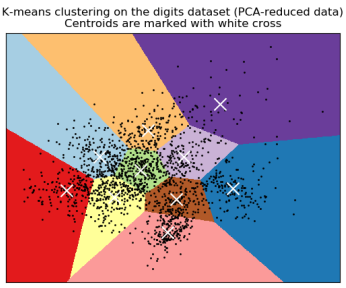
$$\sum_{i=0}^{n} \min_{\mu_j \in C} (||x_i - \mu_j||^2)$$

Inertia can be recognized as a measure of how internally coherent clusters are. It suffers from various drawbacks:

- Inertia makes the assumption that clusters are convex and isotropic, which is not always the case. It responds poorly to elongated clusters, or manifolds with irregular shapes.
- Inertia is not a normalized metric: we just know that lower values are better and zero is optimal. But in very high-dimensional spaces, Euclidean distances tend to become inflated (this is an instance of the so-called "curse of dimensionality"). Running a dimensionality reduction algorithm such as *Principal component analysis (PCA)* prior to k-means clustering can alleviate this problem and speed up the computations.



K-means is often referred to as Lloyd's algorithm. In basic terms, the algorithm has three steps. The first step chooses the initial centroids, with the most basic method being to choose k samples from the dataset X. After initialization, K-means consists of looping between the two other steps. The first step assigns each sample to its nearest centroid. The second step creates new centroids by taking the mean value of all of the samples assigned to each previous centroid. The difference between the old and the new centroids are computed and the algorithm repeats these last two steps until this value is less than a threshold. In other words, it repeats until the centroids do not move significantly.



K-means is equivalent to the expectation-maximization algorithm with a small, all-equal, diagonal covariance matrix.

The algorithm can also be understood through the concept of Voronoi diagrams. First the Voronoi diagram of the points is calculated using the current centroids. Each segment in the Voronoi diagram becomes a separate cluster. Secondly, the centroids are updated to the mean of each segment. The algorithm then repeats this until a stopping criterion is fulfilled. Usually, the algorithm stops when the relative decrease in the objective function between iterations is less than the given tolerance value. This is not the case in this implementation: iteration stops when centroids move less than the tolerance.

Given enough time, K-means will always converge, however this may be to a local minimum. This is highly dependent on the initialization of the centroids. As a result, the computation is often done several times, with different initializations of the centroids. One method to help address this issue is the k-means++ initialization scheme, which has been implemented in scikit-learn (use the `init='k-means++'` parameter). This initializes the centroids to be (generally) distant from each other, leading to provably better results than random initialization, as shown in the reference.

The algorithm supports sample weights, which can be given by a parameter `sample_weight`. This allows to assign more weight to some samples when computing cluster centers and values of inertia. For example, assigning a weight of 2 to a sample is equivalent to adding a duplicate of that sample to the dataset X.

A parameter can be given to allow K-means to be run in parallel, called `n_jobs`. Giving this parameter a positive value uses that many processors (default: 1). A value of -1 uses all available processors, with -2 using one less, and so on. Parallelization generally speeds up computation at the cost of memory (in this case, multiple copies of centroids need to be stored, one for each job).

Warning: The parallel version of K-Means is broken on OS X when `numpy` uses the `Accelerate` Framework. This is expected behavior: `Accelerate` can be called after a fork but you need to execv the subprocess with the Python binary (which multiprocessing does not do under posix).

K-means can be used for vector quantization. This is achieved using the transform method of a trained model of *KMeans*.

Examples:

- *Demonstration of k-means assumptions*: Demonstrating when k-means performs intuitively and when it does not
- *A demo of K-Means clustering on the handwritten digits data*: Clustering handwritten digits

References:

- "k-means++: The advantages of careful seeding" Arthur, David, and Sergei Vassilvitskii, *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics (2007)

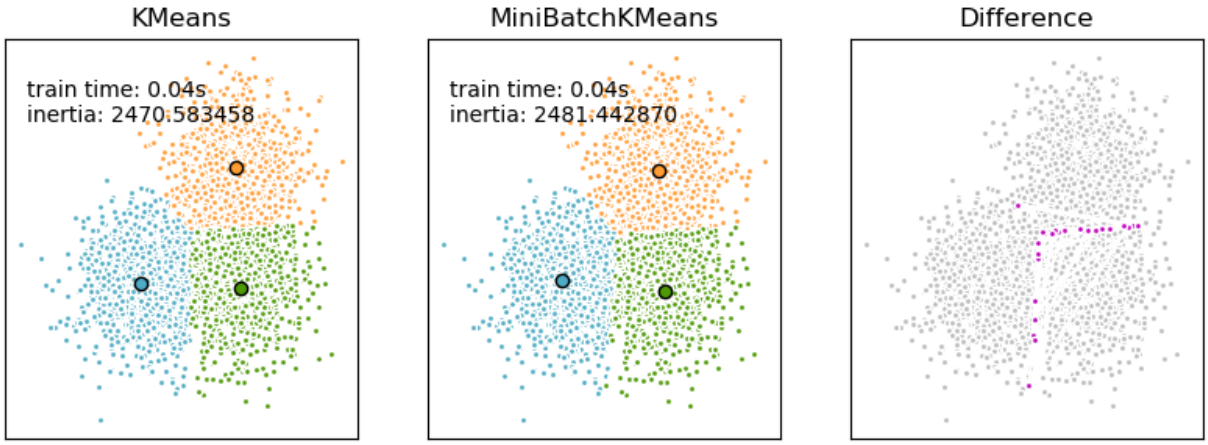
Mini Batch K-Means

The *MiniBatchKMeans* is a variant of the *KMeans* algorithm which uses mini-batches to reduce the computation time, while still attempting to optimise the same objective function. Mini-batches are subsets of the input data, randomly sampled in each training iteration. These mini-batches drastically reduce the amount of computation required

to converge to a local solution. In contrast to other algorithms that reduce the convergence time of k-means, mini-batch k-means produces results that are generally only slightly worse than the standard algorithm.

The algorithm iterates between two major steps, similar to vanilla k-means. In the first step, b samples are drawn randomly from the dataset, to form a mini-batch. These are then assigned to the nearest centroid. In the second step, the centroids are updated. In contrast to k-means, this is done on a per-sample basis. For each sample in the mini-batch, the assigned centroid is updated by taking the streaming average of the sample and all previous samples assigned to that centroid. This has the effect of decreasing the rate of change for a centroid over time. These steps are performed until convergence or a predetermined number of iterations is reached.

`MiniBatchKMeans` converges faster than `KMeans`, but the quality of the results is reduced. In practice this difference in quality can be quite small, as shown in the example and cited reference.



Examples:

- *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*: Comparison of KMeans and MiniBatchKMeans
- *Clustering text documents using k-means*: Document clustering using sparse MiniBatchKMeans
- *Online learning of a dictionary of parts of faces*

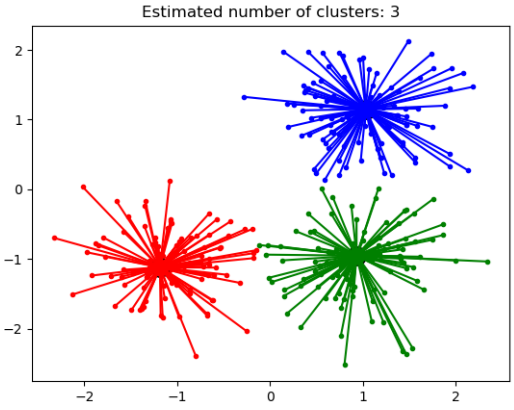
References:

- "Web Scale K-Means clustering" D. Sculley, *Proceedings of the 19th international conference on World wide web* (2010)

Affinity Propagation

`AffinityPropagation` creates clusters by sending messages between pairs of samples until convergence. A dataset is then described using a small number of exemplars, which are identified as those most representative of other samples. The messages sent between pairs represent the suitability for one sample to be the exemplar of the other, which is updated in response to the values from other pairs. This updating happens iteratively until convergence, at which point the final exemplars are chosen, and hence the final clustering is given.

Affinity Propagation can be interesting as it chooses the number of clusters based on the data provided. For this purpose, the two important parameters are the *preference*, which controls how many exemplars are used, and the *damping*



factor which damps the responsibility and availability messages to avoid numerical oscillations when updating these messages.

The main drawback of Affinity Propagation is its complexity. The algorithm has a time complexity of the order $O(N^2T)$, where N is the number of samples and T is the number of iterations until convergence. Further, the memory complexity is of the order $O(N^2)$ if a dense similarity matrix is used, but reducible if a sparse similarity matrix is used. This makes Affinity Propagation most appropriate for small to medium sized datasets.

Examples:

- *Demo of affinity propagation clustering algorithm*: Affinity Propagation on a synthetic 2D datasets with 3 classes.
- *Visualizing the stock market structure* Affinity Propagation on Financial time series to find groups of companies

Algorithm description: The messages sent between points belong to one of two categories. The first is the responsibility $r(i, k)$, which is the accumulated evidence that sample k should be the exemplar for sample i. The second is the availability $a(i, k)$ which is the accumulated evidence that sample i should choose sample k to be its exemplar, and considers the values for all other samples that k should be an exemplar. In this way, exemplars are chosen by samples if they are (1) similar enough to many samples and (2) chosen by many samples to be representative of themselves.

More formally, the responsibility of a sample k to be the exemplar of sample i is given by:

$$r(i, k) \leftarrow s(i, k) - max[a(i, k') + s(i, k')\forall k' \neq k]$$

Where $s(i, k)$ is the similarity between samples i and k. The availability of sample k to be the exemplar of sample i is given by:

$$a(i, k) \leftarrow min[0, r(k, k) + \sum_{i' \ s.t. \ i' \notin \{i,k\}} r(i', k)]$$

To begin with, all values for r and a are set to zero, and the calculation of each iterates until convergence. As discussed above, in order to avoid numerical oscillations when updating the messages, the damping factor λ is introduced to iteration process:

$$r_{t+1}(i, k) = \lambda \cdot r_t(i, k) + (1 - \lambda) \cdot r_{t+1}(i, k)$$

$$a_{t+1}(i, k) = \lambda \cdot a_t(i, k) + (1 - \lambda) \cdot a_{t+1}(i, k)$$

where t indicates the iteration times.

Mean Shift

`MeanShift` clustering aims to discover *blobs* in a smooth density of samples. It is a centroid based algorithm, which works by updating candidates for centroids to be the mean of the points within a given region. These candidates are then filtered in a post-processing stage to eliminate near-duplicates to form the final set of centroids.

Given a candidate centroid x_i for iteration t, the candidate is updated according to the following equation:

$$x_i^{t+1} = m(x_i^t)$$

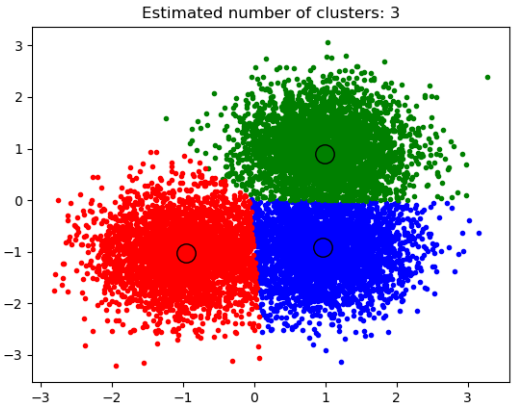
Where $N(x_i)$ is the neighborhood of samples within a given distance around x_i and m is the *mean shift* vector that is computed for each centroid that points towards a region of the maximum increase in the density of points. This is computed using the following equation, effectively updating a centroid to be the mean of the samples within its neighborhood:

$$m(x_i) = \frac{\sum_{x_j \in N(x_i)} K(x_j - x_i) x_j}{\sum_{x_j \in N(x_i)} K(x_j - x_i)}$$

The algorithm automatically sets the number of clusters, instead of relying on a parameter `bandwidth`, which dictates the size of the region to search through. This parameter can be set manually, but can be estimated using the provided `estimate_bandwidth` function, which is called if the bandwidth is not set.

The algorithm is not highly scalable, as it requires multiple nearest neighbor searches during the execution of the algorithm. The algorithm is guaranteed to converge, however the algorithm will stop iterating when the change in centroids is small.

Labelling a new sample is performed by finding the nearest centroid for a given sample.



Examples:

- *A demo of the mean-shift clustering algorithm*: Mean Shift clustering on a synthetic 2D datasets with 3 classes.

References:

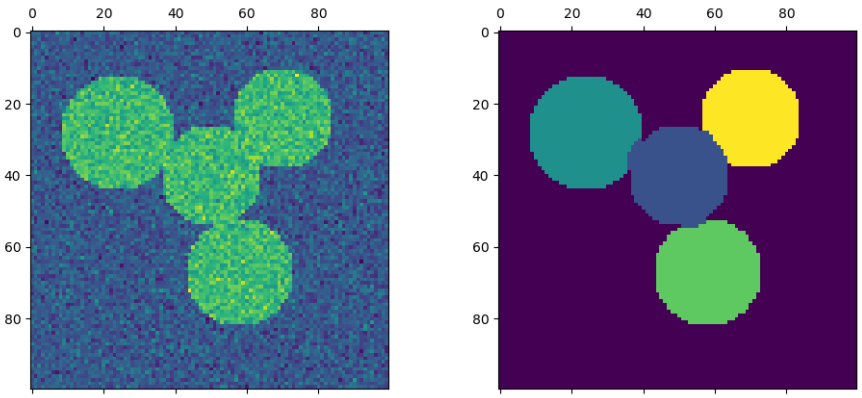
- "Mean shift: A robust approach toward feature space analysis." D. Comaniciu and P. Meer, *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2002)

Spectral clustering

SpectralClustering performs a low-dimension embedding of the affinity matrix between samples, followed by clustering, e.g., by KMeans, of the components of the eigenvectors in the low dimensional space. It is especially computationally efficient if the affinity matrix is sparse and the `amg` solver is used for the eigenvalue problem (Note, the `amg` solver requires that the pyamg module is installed.)

The present version of SpectralClustering requires the number of clusters to be specified in advance. It works well for a small number of clusters, but is not advised for many clusters.

For two clusters, SpectralClustering solves a convex relaxation of the normalised cuts problem on the similarity graph: cutting the graph in two so that the weight of the edges cut is small compared to the weights of the edges inside each cluster. This criteria is especially interesting when working on images, where graph vertices are pixels, and weights of the edges of the similarity graph are computed using a function of a gradient of the image.



Warning: Transforming distance to well-behaved similarities

Note that if the values of your similarity matrix are not well distributed, e.g. with negative values or with a distance matrix rather than a similarity, the spectral problem will be singular and the problem not solvable. In which case it is advised to apply a transformation to the entries of the matrix. For instance, in the case of a signed distance matrix, is common to apply a heat kernel:

```
similarity = np.exp(-beta * distance / distance.std())
```

See the examples for such an application.

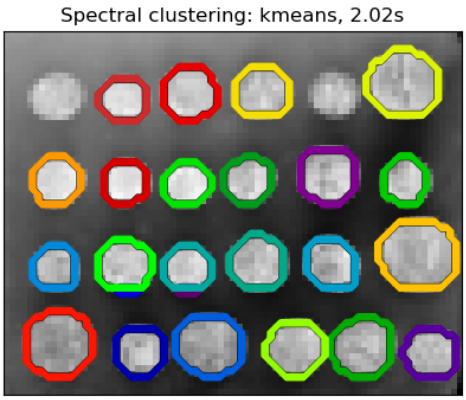
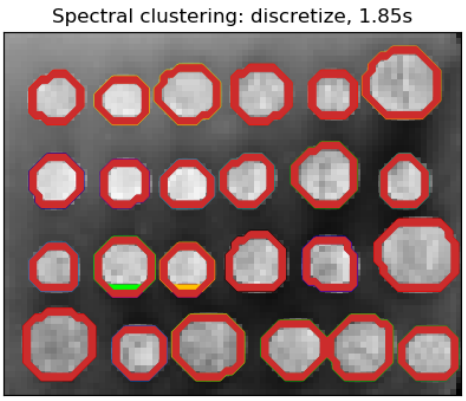
Examples:

- *Spectral clustering for image segmentation*: Segmenting objects from a noisy background using spectral clustering.
- *Segmenting the picture of greek coins in regions*: Spectral clustering to split the image of coins in regions.

Different label assignment strategies

Different label assignment strategies can be used, corresponding to the `assign_labels` parameter of *SpectralClustering*. `"kmeans"` strategy can match finer details, but can be unstable. In particular, unless

you control the `random_state`, it may not be reproducible from run-to-run, as it depends on random initialization. The alternative `"discretize"` strategy is 100% reproducible, but tends to create parcels of fairly even and geometrical shape.

`assign_labels="kmeans"`	`assign_labels="discretize"`
	

Spectral Clustering Graphs

Spectral Clustering can also be used to partition graphs via their spectral embeddings. In this case, the affinity matrix is the adjacency matrix of the graph, and SpectralClustering is initialized with `affinity='precomputed'`:

```
>>> from sklearn.cluster import SpectralClustering
>>> sc = SpectralClustering(3, affinity='precomputed', n_init=100,
...                         assign_labels='discretize')
>>> sc.fit_predict(adjacency_matrix)
```

References:

- "A Tutorial on Spectral Clustering" Ulrike von Luxburg, 2007
- "Normalized cuts and image segmentation" Jianbo Shi, Jitendra Malik, 2000
- "A Random Walks View of Spectral Segmentation" Marina Meila, Jianbo Shi, 2001
- "On Spectral Clustering: Analysis and an algorithm" Andrew Y. Ng, Michael I. Jordan, Yair Weiss, 2001
- "Preconditioned Spectral Clustering for Stochastic Block Partition Streaming Graph Challenge" David Zhuzhunashvili, Andrew Knyazev

Hierarchical clustering

Hierarchical clustering is a general family of clustering algorithms that build nested clusters by merging or splitting them successively. This hierarchy of clusters is represented as a tree (or dendrogram). The root of the tree is the unique

cluster that gathers all the samples, the leaves being the clusters with only one sample. See the Wikipedia page for more details.

The *AgglomerativeClustering* object performs a hierarchical clustering using a bottom up approach: each observation starts in its own cluster, and clusters are successively merged together. The linkage criteria determines the metric used for the merge strategy:

- **Ward** minimizes the sum of squared differences within all clusters. It is a variance-minimizing approach and in this sense is similar to the k-means objective function but tackled with an agglomerative hierarchical approach.
- **Maximum** or **complete linkage** minimizes the maximum distance between observations of pairs of clusters.
- **Average linkage** minimizes the average of the distances between all observations of pairs of clusters.
- **Single linkage** minimizes the distance between the closest observations of pairs of clusters.

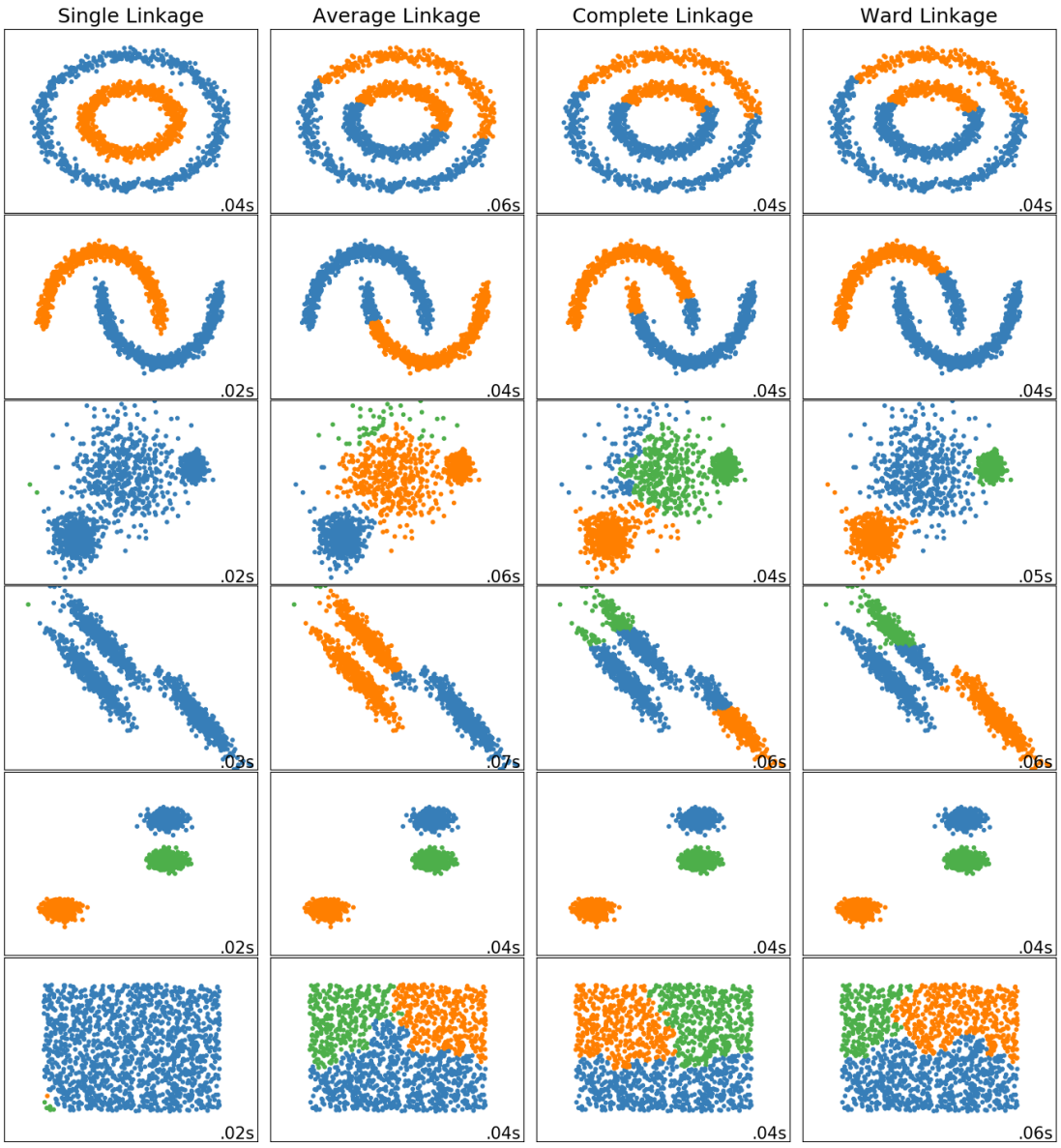
AgglomerativeClustering can also scale to large number of samples when it is used jointly with a connectivity matrix, but is computationally expensive when no connectivity constraints are added between samples: it considers at each step all the possible merges.

FeatureAgglomeration

The *FeatureAgglomeration* uses agglomerative clustering to group together features that look very similar, thus decreasing the number of features. It is a dimensionality reduction tool, see *Unsupervised dimensionality reduction*.

Different linkage type: Ward, complete, average, and single linkage

AgglomerativeClustering supports Ward, single, average, and complete linkage strategies.



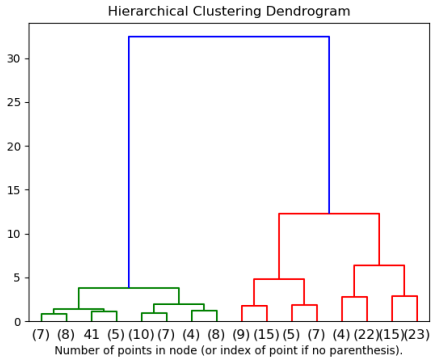
Agglomerative cluster has a "rich get richer" behavior that leads to uneven cluster sizes. In this regard, single linkage is the worst strategy, and Ward gives the most regular sizes. However, the affinity (or distance used in clustering) cannot be varied with Ward, thus for non Euclidean metrics, average linkage is a good alternative. Single linkage, while not robust to noisy data, can be computed very efficiently and can therefore be useful to provide hierarchical clustering of larger datasets. Single linkage can also perform well on non-globular data.

Examples:

- *Various Agglomerative Clustering on a 2D embedding of digits*: exploration of the different linkage strategies in a real dataset.

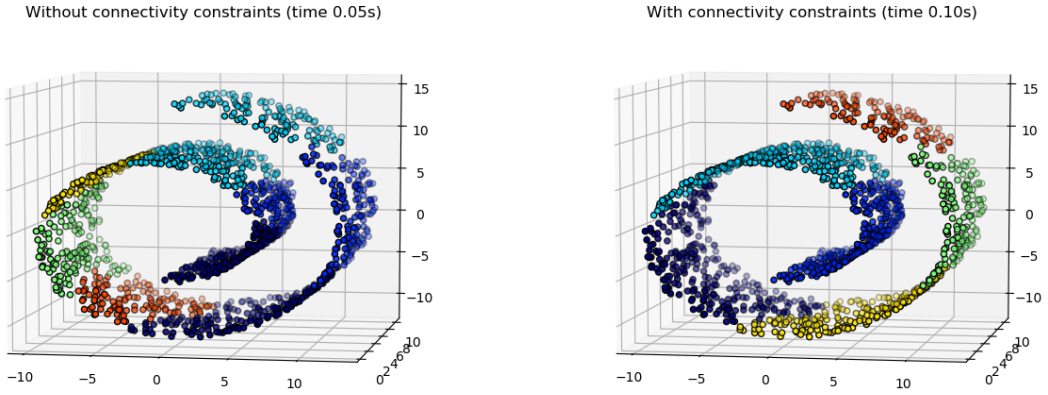
Visualization of cluster hierarchy

It's possible to visualize the tree representing the hierarchical merging of clusters as a dendrogram. Visual inspection can often be useful for understanding the structure of the data, though more so in the case of small sample sizes.



Adding connectivity constraints

An interesting aspect of *AgglomerativeClustering* is that connectivity constraints can be added to this algorithm (only adjacent clusters can be merged together), through a connectivity matrix that defines for each sample the neighboring samples following a given structure of the data. For instance, in the swiss-roll example below, the connectivity constraints forbid the merging of points that are not adjacent on the swiss roll, and thus avoid forming clusters that extend across overlapping folds of the roll.



These constraint are useful to impose a certain local structure, but they also make the algorithm faster, especially when the number of the samples is high.

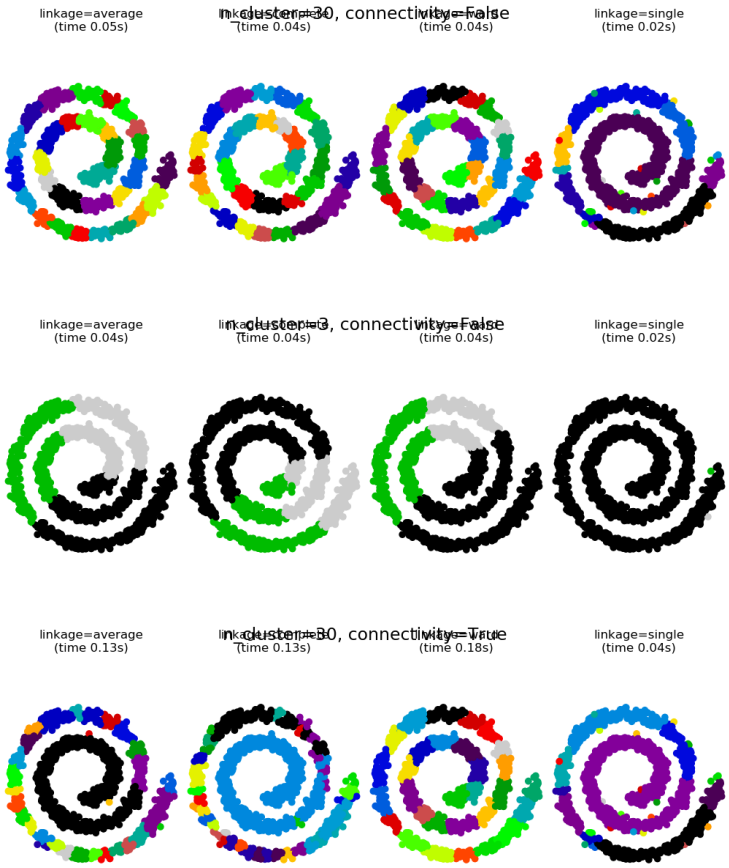
The connectivity constraints are imposed via an connectivity matrix: a scipy sparse matrix that has elements only at the intersection of a row and a column with indices of the dataset that should be connected. This matrix can be constructed from a-priori information: for instance, you may wish to cluster web pages by only merging pages with a link pointing from one to another. It can also be learned from the data, for instance using *sklearn.neighbors.kneighbors_graph* to restrict merging to nearest neighbors as in *this example*, or using *sklearn.feature_extraction.image.grid_to_graph* to enable only merging of neighboring pixels on an image, as in the *coin* example.

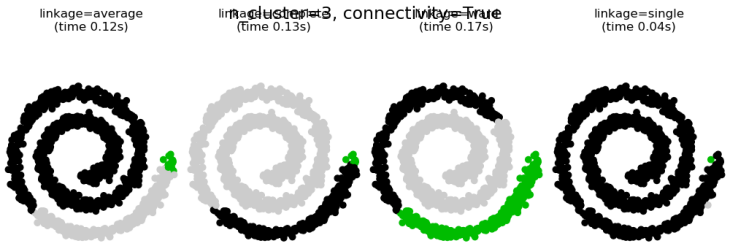
Examples:

- *A demo of structured Ward hierarchical clustering on an image of coins*: Ward clustering to split the image of coins in regions.
- *Hierarchical clustering: structured vs unstructured ward*: Example of Ward algorithm on a swiss-roll, comparison of structured approaches versus unstructured approaches.
- *Feature agglomeration vs. univariate selection*: Example of dimensionality reduction with feature agglomeration based on Ward hierarchical clustering.
- *Agglomerative clustering with and without structure*

Warning: Connectivity constraints with single, average and complete linkage

Connectivity constraints and single, complete or average linkage can enhance the 'rich getting richer' aspect of agglomerative clustering, particularly so if they are built with `sklearn.neighbors.kneighbors_graph`. In the limit of a small number of clusters, they tend to give a few macroscopically occupied clusters and almost empty ones. (see the discussion in *Agglomerative clustering with and without structure*). Single linkage is the most brittle linkage option with regard to this issue.



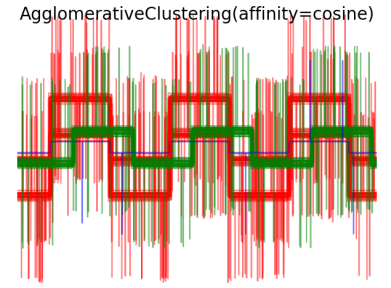


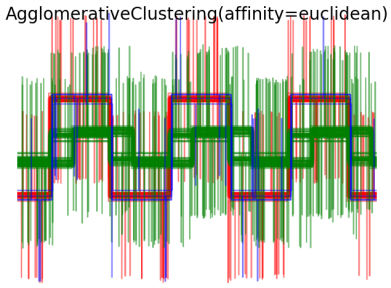
Varying the metric

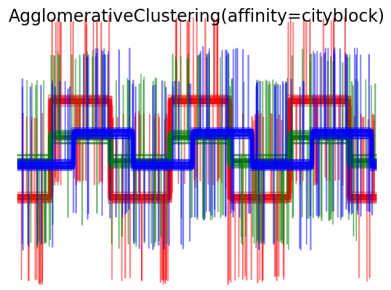
Single, average and complete linkage can be used with a variety of distances (or affinities), in particular Euclidean distance (*l2*), Manhattan distance (or Cityblock, or *l1*), cosine distance, or any precomputed affinity matrix.

- *l1* distance is often good for sparse features, or sparse noise: i.e. many of the features are zero, as in text mining using occurrences of rare words.
- *cosine* distance is interesting because it is invariant to global scalings of the signal.

The guidelines for choosing a metric is to use one that maximizes the distance between samples in different classes, and minimizes that within each class.







Examples:

- *Agglomerative clustering with different metrics*

DBSCAN

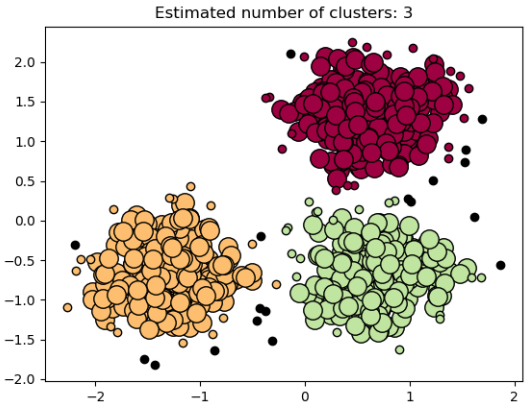
The `DBSCAN` algorithm views clusters as areas of high density separated by areas of low density. Due to this rather generic view, clusters found by DBSCAN can be any shape, as opposed to k-means which assumes that clusters are convex shaped. The central component to the DBSCAN is the concept of *core samples*, which are samples that are in areas of high density. A cluster is therefore a set of core samples, each close to each other (measured by some distance measure) and a set of non-core samples that are close to a core sample (but are not themselves core samples). There are two parameters to the algorithm, `min_samples` and `eps`, which define formally what we mean when we say *dense*. Higher `min_samples` or lower `eps` indicate higher density necessary to form a cluster.

More formally, we define a core sample as being a sample in the dataset such that there exist `min_samples` other samples within a distance of `eps`, which are defined as *neighbors* of the core sample. This tells us that the core sample is in a dense area of the vector space. A cluster is a set of core samples that can be built by recursively taking a core sample, finding all of its neighbors that are core samples, finding all of *their* neighbors that are core samples, and so on. A cluster also has a set of non-core samples, which are samples that are neighbors of a core sample in the cluster but are not themselves core samples. Intuitively, these samples are on the fringes of a cluster.

Any core sample is part of a cluster, by definition. Any sample that is not a core sample, and is at least `eps` in distance from any core sample, is considered an outlier by the algorithm.

While the parameter `min_samples` primarily controls how tolerant the algorithm is towards noise (on noisy and large data sets it may be desirable to increase this parameter), the parameter `eps` is *crucial to choose appropriately* for the data set and distance function and usually cannot be left at the default value. It controls the local neighborhood of the points. When chosen too small, most data will not be clustered at all (and labeled as `-1` for "noise"). When chosen too large, it causes close clusters to be merged into one cluster, and eventually the entire data set to be returned as a single cluster. Some heuristics for choosing this parameter have been discussed in the literature, for example based on a knee in the nearest neighbor distances plot (as discussed in the references below).

In the figure below, the color indicates cluster membership, with large circles indicating core samples found by the algorithm. Smaller circles are non-core samples that are still part of a cluster. Moreover, the outliers are indicated by black points below.



Examples:

- *Demo of DBSCAN clustering algorithm*

Implementation

The DBSCAN algorithm is deterministic, always generating the same clusters when given the same data in the same order. However, the results can differ when data is provided in a different order. First, even though the core samples will always be assigned to the same clusters, the labels of those clusters will depend on the order in which those samples are encountered in the data. Second and more importantly, the clusters to which non-core samples are assigned can differ depending on the data order. This would happen when a non-core sample has a distance lower than `eps` to two core samples in different clusters. By the triangular inequality, those two core samples must be more distant than `eps` from each other, or they would be in the same cluster. The non-core sample is assigned to whichever cluster is generated first in a pass through the data, and so the results will depend on the data ordering.

The current implementation uses ball trees and kd-trees to determine the neighborhood of points, which avoids calculating the full distance matrix (as was done in scikit-learn versions before 0.14). The possibility to use custom metrics is retained; for details, see `NearestNeighbors`.

Memory consumption for large sample sizes

This implementation is by default not memory efficient because it constructs a full pairwise similarity matrix in the case where kd-trees or ball-trees cannot be used (e.g., with sparse matrices). This matrix will consume n^2 floats. A couple of mechanisms for getting around this are:

- Use *OPTICS* clustering in conjunction with the `extract_dbscan` method. OPTICS clustering also calculates the full pairwise matrix, but only keeps one row in memory at a time (memory complexity n).
- A sparse radius neighborhood graph (where missing entries are presumed to be out of eps) can be precomputed in a memory-efficient way and dbscan can be run over this with `metric='precomputed'`. See *`sklearn.neighbors.NearestNeighbors.radius_neighbors_graph`*.
- The dataset can be compressed, either by removing exact duplicates if these occur in your data, or by using BIRCH. Then you only have a relatively small number of representatives for a large number of points. You can then provide a `sample_weight` when fitting DBSCAN.

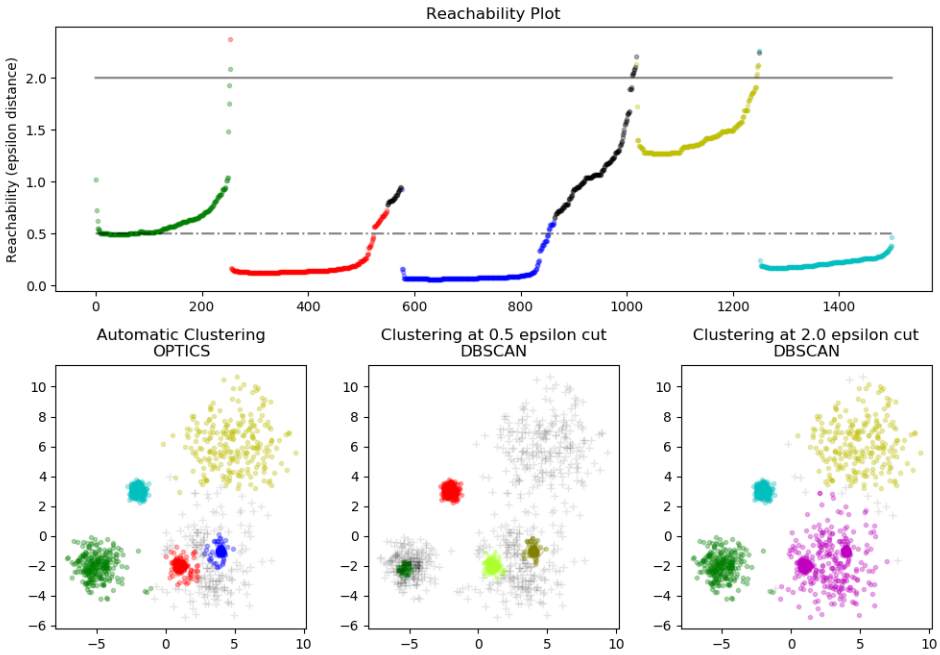
References:

- "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise" Ester, M., H. P. Kriegel, J. Sander, and X. Xu, In Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press, pp. 226–231. 1996
- "DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. Schubert, E., Sander, J., Ester, M., Kriegel, H. P., & Xu, X. (2017). In ACM Transactions on Database Systems (TODS), 42(3), 19.

OPTICS

The *`OPTICS`* algorithm shares many similarities with the *`DBSCAN`* algorithm, and can be considered a generalization of DBSCAN that relaxes the `eps` requirement from a single value to a value range. The key difference between DBSCAN and OPTICS is that the OPTICS algorithm builds a *reachability* graph, which assigns each sample both

a `reachability_` distance, and a spot within the cluster `ordering_` attribute; these two attributes are assigned when the model is fitted, and are used to determine cluster membership. If OPTICS is run with the default value of *inf* set for `max_eps`, then DBSCAN style cluster extraction can be performed repeatedly in linear time for any given `eps` value using the `cluster_optics_dbscan` method. Setting `max_eps` to a lower value will result in shorter run times, and can be thought of as the maximum neighborhood radius from each point to find other potential reachable points.



The *reachability* distances generated by OPTICS allow for variable density extraction of clusters within a single data set. As shown in the above plot, combining *reachability* distances and data set `ordering_` produces a *reachability plot*, where point density is represented on the Y-axis, and points are ordered such that nearby points are adjacent. 'Cutting' the reachability plot at a single value produces DBSCAN like results; all points above the 'cut' are classified as noise, and each time that there is a break when reading from left to right signifies a new cluster. The default cluster extraction with OPTICS looks at the steep slopes within the graph to find clusters, and the user can define what counts as a steep slope using the parameter `xi`. There are also other possibilities for analysis on the graph itself, such as generating hierarchical representations of the data through reachability-plot dendrograms, and the hierarchy of clusters detected by the algorithm can be accessed through the `cluster_hierarchy_` parameter. The plot above has been color-coded so that cluster colors in planar space match the linear segment clusters of the reachability plot. Note that the blue and red clusters are adjacent in the reachability plot, and can be hierarchically represented as children of a larger parent cluster.

Examples:

- *Demo of OPTICS clustering algorithm*

Comparison with DBSCAN

The results from OPTICS `cluster_optics_dbscan` method and DBSCAN are very similar, but not always identical; specifically, labeling of periphery and noise points. This is in part because the first samples of each dense area processed by OPTICS have a large reachability value while being close to other points in their area, and will thus sometimes be marked as noise rather than periphery. This affects adjacent points when they are considered as

candidates for being marked as either periphery or noise.

Note that for any single value of `eps`, DBSCAN will tend to have a shorter run time than OPTICS; however, for repeated runs at varying `eps` values, a single run of OPTICS may require less cumulative runtime than DBSCAN. It is also important to note that OPTICS' output is close to DBSCAN's only if `eps` and `max_eps` are close.

Computational Complexity

Spatial indexing trees are used to avoid calculating the full distance matrix, and allow for efficient memory usage on large sets of samples. Different distance metrics can be supplied via the `metric` keyword.

For large datasets, similar (but not identical) results can be obtained via HDBSCAN. The HDBSCAN implementation is multithreaded, and has better algorithmic runtime complexity than OPTICS, at the cost of worse memory scaling. For extremely large datasets that exhaust system memory using HDBSCAN, OPTICS will maintain *n* (as opposed to *n*^2) memory scaling; however, tuning of the `max_eps` parameter will likely need to be used to give a solution in a reasonable amount of wall time.

References:

- "OPTICS: ordering points to identify the clustering structure." Ankerst, Mihael, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. In ACM Sigmod Record, vol. 28, no. 2, pp. 49-60. ACM, 1999.

Birch

The *Birch* builds a tree called the Clustering Feature Tree (CFT) for the given data. The data is essentially lossy compressed to a set of Clustering Feature nodes (CF Nodes). The CF Nodes have a number of subclusters called Clustering Feature subclusters (CF Subclusters) and these CF Subclusters located in the non-terminal CF Nodes can have CF Nodes as children.

The CF Subclusters hold the necessary information for clustering which prevents the need to hold the entire input data in memory. This information includes:

- Number of samples in a subcluster.
- Linear Sum - A n-dimensional vector holding the sum of all samples
- Squared Sum - Sum of the squared L2 norm of all samples.
- Centroids - To avoid recalculation linear sum / n_samples.
- Squared norm of the centroids.

The Birch algorithm has two parameters, the threshold and the branching factor. The branching factor limits the number of subclusters in a node and the threshold limits the distance between the entering sample and the existing subclusters.

This algorithm can be viewed as an instance or data reduction method, since it reduces the input data to a set of subclusters which are obtained directly from the leaves of the CFT. This reduced data can be further processed by feeding it into a global clusterer. This global clusterer can be set by `n_clusters`. If `n_clusters` is set to None, the subclusters from the leaves are directly read off, otherwise a global clustering step labels these subclusters into global clusters (labels) and the samples are mapped to the global label of the nearest subcluster.

Algorithm description:

- A new sample is inserted into the root of the CF Tree which is a CF Node. It is then merged with the subcluster of the root, that has the smallest radius after merging, constrained by the threshold and branching factor conditions. If the subcluster has any child node, then this is done repeatedly till it reaches a leaf. After finding the nearest subcluster in the leaf, the properties of this subcluster and the parent subclusters are recursively updated.
- If the radius of the subcluster obtained by merging the new sample and the nearest subcluster is greater than the square of the threshold and if the number of subclusters is greater than the branching factor, then a space is temporarily allocated to this new sample. The two farthest subclusters are taken and the subclusters are divided into two groups on the basis of the distance between these subclusters.
- If this split node has a parent subcluster and there is room for a new subcluster, then the parent is split into two. If there is no room, then this node is again split into two and the process is continued recursively, till it reaches the root.

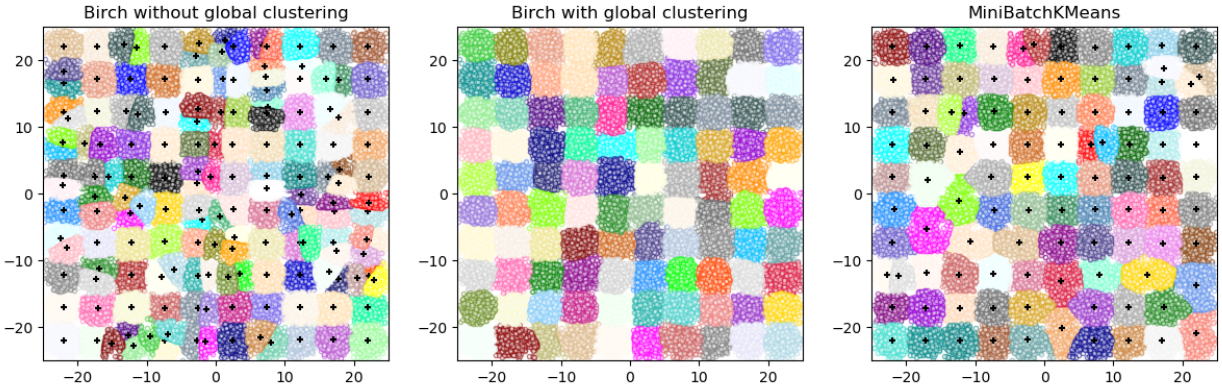
Birch or MiniBatchKMeans?

- Birch does not scale very well to high dimensional data. As a rule of thumb if `n_features` is greater than twenty, it is generally better to use MiniBatchKMeans.
- If the number of instances of data needs to be reduced, or if one wants a large number of subclusters either as a preprocessing step or otherwise, Birch is more useful than MiniBatchKMeans.

How to use partial_fit?

To avoid the computation of global clustering, for every call of `partial_fit` the user is advised

1. To set `n_clusters=None` initially
2. Train all data by multiple calls to partial_fit.
3. Set `n_clusters` to a required value using `brc.set_params(n_clusters=n_clusters)`.
4. Call `partial_fit` finally with no arguments, i.e. `brc.partial_fit()` which performs the global clustering.



References:

- Tian Zhang, Raghu Ramakrishnan, Maron Livny BIRCH: An efficient data clustering method for large databases. https://www.cs.sfu.ca/CourseCentral/459/han/papers/zhang96.pdf
- Roberto Perdisci JBirch - Java implementation of BIRCH clustering algorithm https://code.google.com/archive/p/jbirch

Clustering performance evaluation

Evaluating the performance of a clustering algorithm is not as trivial as counting the number of errors or the precision and recall of a supervised classification algorithm. In particular any evaluation metric should not take the absolute values of the cluster labels into account but rather if this clustering define separations of the data similar to some ground truth set of classes or satisfying some assumption such that members belong to the same class are more similar than members of different classes according to some similarity metric.

Adjusted Rand index

Given the knowledge of the ground truth class assignments `labels_true` and our clustering algorithm assignments of the same samples `labels_pred`, the **adjusted Rand index** is a function that measures the **similarity** of the two assignments, ignoring permutations and **with chance normalization**:

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]

>>> metrics.adjusted_rand_score(labels_true, labels_pred)
0.24...
```

One can permute 0 and 1 in the predicted labels, rename 2 to 3, and get the same score:

```
>>> labels_pred = [1, 1, 0, 0, 3, 3]
>>> metrics.adjusted_rand_score(labels_true, labels_pred)
0.24...
```

Furthermore, *adjusted_rand_score* is **symmetric**: swapping the argument does not change the score. It can thus be used as a **consensus measure**:

```
>>> metrics.adjusted_rand_score(labels_pred, labels_true)
0.24...
```

Perfect labeling is scored 1.0:

```
>>> labels_pred = labels_true[:]
>>> metrics.adjusted_rand_score(labels_true, labels_pred)
1.0
```

Bad (e.g. independent labelings) have negative or close to 0.0 scores:

```
>>> labels_true = [0, 1, 2, 0, 3, 4, 5, 1]
>>> labels_pred = [1, 1, 0, 0, 2, 2, 2, 2]
>>> metrics.adjusted_rand_score(labels_true, labels_pred)
-0.12...
```

Advantages

- **Random (uniform) label assignments have a ARI score close to 0.0** for any value of `n_clusters` and `n_samples` (which is not the case for raw Rand index or the V-measure for instance).
- **Bounded range [-1, 1]**: negative values are bad (independent labelings), similar clusterings have a positive ARI, 1.0 is the perfect match score.

- **No assumption is made on the cluster structure**: can be used to compare clustering algorithms such as k-means which assumes isotropic blob shapes with results of spectral clustering algorithms which can find cluster with "folded" shapes.

Drawbacks

- Contrary to inertia, **ARI requires knowledge of the ground truth classes** while is almost never available in practice or requires manual assignment by human annotators (as in the supervised learning setting).

  However ARI can also be useful in a purely unsupervised setting as a building block for a Consensus Index that can be used for clustering model selection (TODO).

Examples:

- *Adjustment for chance in clustering performance evaluation*: Analysis of the impact of the dataset size on the value of clustering measures for random assignments.

Mathematical formulation

If C is a ground truth class assignment and K the clustering, let us define a and b as:

- a, the number of pairs of elements that are in the same set in C and in the same set in K
- b, the number of pairs of elements that are in different sets in C and in different sets in K

The raw (unadjusted) Rand index is then given by:

$$\text{RI} = \frac{a + b}{C_2^{n_{samples}}}$$

Where $C_2^{n_{samples}}$ is the total number of possible pairs in the dataset (without ordering).

However the RI score does not guarantee that random label assignments will get a value close to zero (esp. if the number of clusters is in the same order of magnitude as the number of samples).

To counter this effect we can discount the expected RI $E[\text{RI}]$ of random labelings by defining the adjusted Rand index as follows:

$$\text{ARI} = \frac{\text{RI} - E[\text{RI}]}{\max(\text{RI}) - E[\text{RI}]}$$

References

- Comparing Partitions L. Hubert and P. Arabie, Journal of Classification 1985
- Wikipedia entry for the adjusted Rand index

Mutual Information based scores

Given the knowledge of the ground truth class assignments `labels_true` and our clustering algorithm assignments of the same samples `labels_pred`, the **Mutual Information** is a function that measures the **agreement** of the two assignments, ignoring permutations. Two different normalized versions of this measure are available, **Normalized Mutual Information (NMI)** and **Adjusted Mutual Information (AMI)**. NMI is often used in the literature, while AMI was proposed more recently and is **normalized against chance**:

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]

>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
0.22504...
```

One can permute 0 and 1 in the predicted labels, rename 2 to 3 and get the same score:

```
>>> labels_pred = [1, 1, 0, 0, 3, 3]
>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
0.22504...
```

All, *mutual_info_score*, *adjusted_mutual_info_score* and *normalized_mutual_info_score* are symmetric: swapping the argument does not change the score. Thus they can be used as a **consensus measure**:

```
>>> metrics.adjusted_mutual_info_score(labels_pred, labels_true)
0.22504...
```

Perfect labeling is scored 1.0:

```
>>> labels_pred = labels_true[:]
>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
1.0

>>> metrics.normalized_mutual_info_score(labels_true, labels_pred)
1.0
```

This is not true for `mutual_info_score`, which is therefore harder to judge:

```
>>> metrics.mutual_info_score(labels_true, labels_pred)
0.69...
```

Bad (e.g. independent labelings) have non-positive scores:

```
>>> labels_true = [0, 1, 2, 0, 3, 4, 5, 1]
>>> labels_pred = [1, 1, 0, 0, 2, 2, 2, 2]
>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
-0.10526...
```

Advantages

- **Random (uniform) label assignments have a AMI score close to 0.0** for any value of `n_clusters` and `n_samples` (which is not the case for raw Mutual Information or the V-measure for instance).
- **Upper bound of 1**: Values close to zero indicate two label assignments that are largely independent, while values close to one indicate significant agreement. Further, an AMI of exactly 1 indicates that the two label assignments are equal (with or without permutation).

Drawbacks

- Contrary to inertia, **MI-based measures require the knowledge of the ground truth classes** while almost never available in practice or requires manual assignment by human annotators (as in the supervised learning setting).

However MI-based measures can also be useful in purely unsupervised setting as a building block for a Consensus Index that can be used for clustering model selection.

- NMI and MI are not adjusted against chance.

Examples:

- *Adjustment for chance in clustering performance evaluation*: Analysis of the impact of the dataset size on the value of clustering measures for random assignments. This example also includes the Adjusted Rand Index.

Mathematical formulation

Assume two label assignments (of the same N objects), U and V. Their entropy is the amount of uncertainty for a partition set, defined by:

$$H(U) = -\sum_{i=1}^{|U|} P(i) \log(P(i))$$

where $P(i) = |U_i|/N$ is the probability that an object picked at random from U falls into class U_i. Likewise for V:

$$H(V) = -\sum_{j=1}^{|V|} P'(j) \log(P'(j))$$

With $P'(j) = |V_j|/N$. The mutual information (MI) between U and V is calculated by:

$$\text{MI}(U, V) = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} P(i, j) \log \left(\frac{P(i,j)}{P(i)P'(j)} \right)$$

where $P(i, j) = |U_i \cap V_j|/N$ is the probability that an object picked at random falls into both classes U_i and V_j.

It also can be expressed in set cardinality formulation:

$$\text{MI}(U, V) = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} \frac{|U_i \cap V_j|}{N} \log \left(\frac{N|U_i \cap V_j|}{|U_i||V_j|} \right)$$

The normalized mutual information is defined as

$$\text{NMI}(U, V) = \frac{\text{MI}(U, V)}{\text{mean}(H(U), H(V))}$$

This value of the mutual information and also the normalized variant is not adjusted for chance and will tend to increase as the number of different labels (clusters) increases, regardless of the actual amount of "mutual information" between the label assignments.

The expected value for the mutual information can be calculated using the following equation [VEB2009]. In this equation, $a_i = |U_i|$ (the number of elements in U_i) and $b_j = |V_j|$ (the number of elements in V_j).

$$E[\text{MI}(U, V)] = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} \sum_{n_{ij}=(a_i+b_j-N)^+}^{\min(a_i, b_j)} \frac{n_{ij}}{N} \log \left(\frac{N.n_{ij}}{a_i b_j} \right) \frac{a_i! b_j! (N - a_i)! (N - b_j)!}{N! n_{ij}! (a_i - n_{ij})! (b_j - n_{ij})! (N - a_i - b_j + n_{ij})!}$$

Using the expected value, the adjusted mutual information can then be calculated using a similar form to that of the adjusted Rand index:

$$\text{AMI} = \frac{\text{MI} - E[\text{MI}]}{\text{mean}(H(U), H(V)) - E[\text{MI}]}$$

For normalized mutual information and adjusted mutual information, the normalizing value is typically some *generalized* mean of the entropies of each clustering. Various generalized means exist, and no firm rules exist for preferring one over the others. The decision is largely a field-by-field basis; for instance, in community detection, the arithmetic mean is most common. Each normalizing method provides "qualitatively similar behaviours" [YAT2016]. In our implementation, this is controlled by the `average_method` parameter.

Vinh et al. (2010) named variants of NMI and AMI by their averaging method [VEB2010]. Their 'sqrt' and 'sum' averages are the geometric and arithmetic means; we use these more broadly common names.

References

- Strehl, Alexander, and Joydeep Ghosh (2002). "Cluster ensembles – a knowledge reuse framework for combining multiple partitions". Journal of Machine Learning Research 3: 583–617. doi:10.1162/153244303321897735.
- Wikipedia entry for the (normalized) Mutual Information
- Wikipedia entry for the Adjusted Mutual Information

Homogeneity, completeness and V-measure

Given the knowledge of the ground truth class assignments of the samples, it is possible to define some intuitive metric using conditional entropy analysis.

In particular Rosenberg and Hirschberg (2007) define the following two desirable objectives for any cluster assignment:

- **homogeneity**: each cluster contains only members of a single class.
- **completeness**: all members of a given class are assigned to the same cluster.

We can turn those concept as scores *homogeneity_score* and *completeness_score*. Both are bounded below by 0.0 and above by 1.0 (higher is better):

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]

>>> metrics.homogeneity_score(labels_true, labels_pred)
0.66...

>>> metrics.completeness_score(labels_true, labels_pred)
0.42...
```

Their harmonic mean called **V-measure** is computed by *v_measure_score*:

```
>>> metrics.v_measure_score(labels_true, labels_pred)
0.51...
```

This function's formula is as follows:

$$v = \frac{(1 + \beta) \times \text{homogeneity} \times \text{completeness}}{(\beta \times \text{homogeneity} + \text{completeness})}$$

`beta` defaults to a value of 1.0, but for using a value less than 1 for beta:

```
>>> metrics.v_measure_score(labels_true, labels_pred, beta=0.6)
0.54...
```

more weight will be attributed to homogeneity, and using a value greater than 1:

```
>>> metrics.v_measure_score(labels_true, labels_pred, beta=1.8)
0.48...
```

more weight will be attributed to completeness.

The V-measure is actually equivalent to the mutual information (NMI) discussed above, with the aggregation function being the arithmetic mean [B2011].

Homogeneity, completeness and V-measure can be computed at once using *homogeneity_completeness_v_measure* as follows:

```
>>> metrics.homogeneity_completeness_v_measure(labels_true, labels_pred)
(0.66..., 0.42..., 0.51...)
```

The following clustering assignment is slightly better, since it is homogeneous but not complete:

```
>>> labels_pred = [0, 0, 0, 1, 2, 2]
>>> metrics.homogeneity_completeness_v_measure(labels_true, labels_pred)
(1.0, 0.68..., 0.81...)
```

Note: *v_measure_score* is **symmetric**: it can be used to evaluate the **agreement** of two independent assignments on the same dataset.

This is not the case for *completeness_score* and *homogeneity_score*: both are bound by the relationship:

```
homogeneity_score(a, b) == completeness_score(b, a)
```

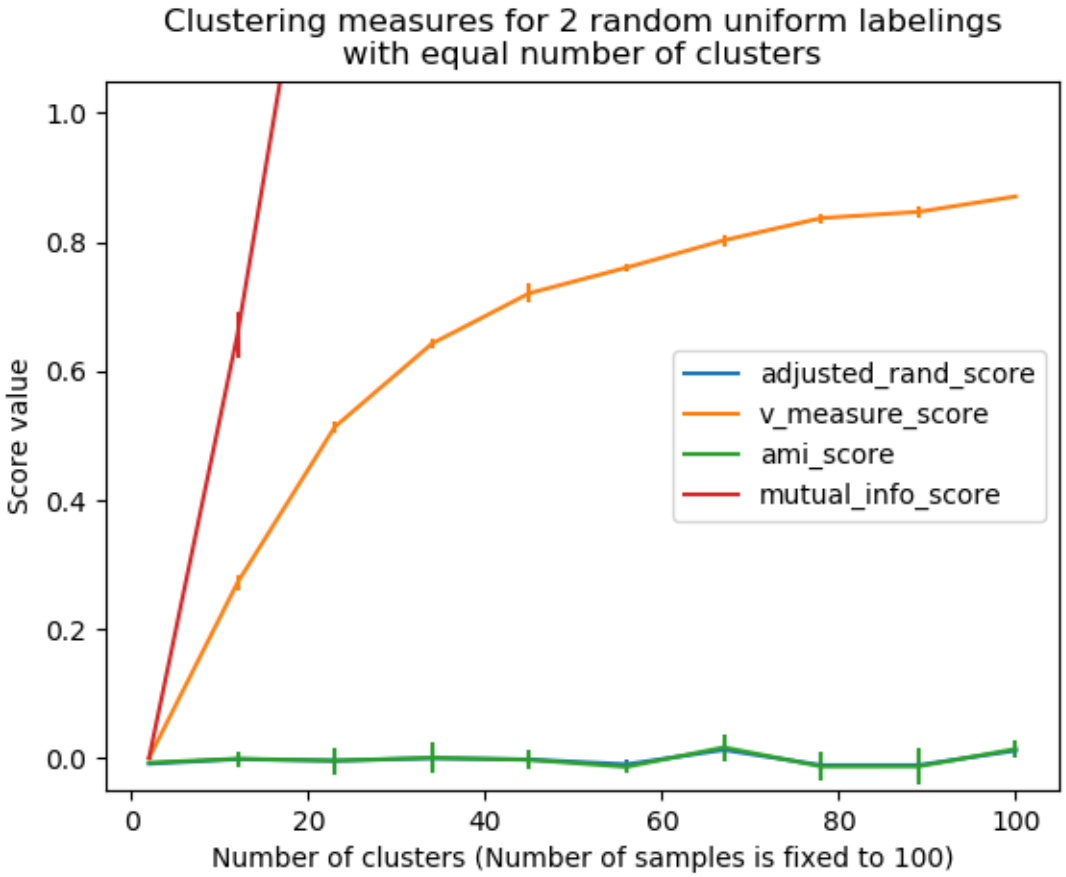
Advantages

- **Bounded scores**: 0.0 is as bad as it can be, 1.0 is a perfect score.
- Intuitive interpretation: clustering with bad V-measure can be **qualitatively analyzed in terms of homogeneity and completeness** to better feel what 'kind' of mistakes is done by the assignment.
- **No assumption is made on the cluster structure**: can be used to compare clustering algorithms such as k-means which assumes isotropic blob shapes with results of spectral clustering algorithms which can find cluster with "folded" shapes.

Drawbacks

- The previously introduced metrics are **not normalized with regards to random labeling**: this means that depending on the number of samples, clusters and ground truth classes, a completely random labeling will not always yield the same values for homogeneity, completeness and hence v-measure. In particular **random labeling won't yield zero scores especially when the number of clusters is large**.

  This problem can safely be ignored when the number of samples is more than a thousand and the number of clusters is less than 10. **For smaller sample sizes or larger number of clusters it is safer to use an adjusted index such as the Adjusted Rand Index (ARI)**.


Clustering measures for 2 random uniform labelings
with equal number of clusters
Score value
Number of clusters (Number of samples is fixed to 100)
adjusted_rand_score
v_measure_score
ami_score
mutual_info_score
0.0
0.2
0.4
0.6
0.8
1.0
0
20
40
60
80
100

- These metrics **require the knowledge of the ground truth classes** while almost never available in practice or requires manual assignment by human annotators (as in the supervised learning setting).

Examples:

- *Adjustment for chance in clustering performance evaluation*: Analysis of the impact of the dataset size on the value of clustering measures for random assignments.

Mathematical formulation

Homogeneity and completeness scores are formally given by:

$$h = 1 - \frac{H(C|K)}{H(C)}$$

$$c = 1 - \frac{H(K|C)}{H(K)}$$

where $H(C|K)$ is the **conditional entropy of the classes given the cluster assignments** and is given by:

$$H(C|K) = -\sum_{c=1}^{|C|}\sum_{k=1}^{|K|}\frac{n_{c,k}}{n}\cdot\log\left(\frac{n_{c,k}}{n_k}\right)$$

and $H(C)$ is the **entropy of the classes** and is given by:

$$H(C) = -\sum_{c=1}^{|C|}\frac{n_c}{n}\cdot\log\left(\frac{n_c}{n}\right)$$

with n the total number of samples, n_c and n_k the number of samples respectively belonging to class c and cluster k, and finally $n_{c,k}$ the number of samples from class c assigned to cluster k.

The **conditional entropy of clusters given class** $H(K|C)$ and the **entropy of clusters** $H(K)$ are defined in a symmetric manner.

Rosenberg and Hirschberg further define **V-measure** as the **harmonic mean of homogeneity and completeness**:

$$v = 2\cdot\frac{h\cdot c}{h+c}$$

References

- V-Measure: A conditional entropy-based external cluster evaluation measure Andrew Rosenberg and Julia Hirschberg, 2007

Fowlkes-Mallows scores

The Fowlkes-Mallows index (`sklearn.metrics.fowlkes_mallows_score`) can be used when the ground truth class assignments of the samples is known. The Fowlkes-Mallows score FMI is defined as the geometric mean of the pairwise precision and recall:

$$\text{FMI} = \frac{\text{TP}}{\sqrt{(\text{TP}+\text{FP})(\text{TP}+\text{FN})}}$$

Where `TP` is the number of **True Positive** (i.e. the number of pair of points that belong to the same clusters in both the true labels and the predicted labels), `FP` is the number of **False Positive** (i.e. the number of pair of points that belong to the same clusters in the true labels and not in the predicted labels) and `FN` is the number of **False Negative** (i.e the number of pair of points that belongs in the same clusters in the predicted labels and not in the true labels).

The score ranges from 0 to 1. A high value indicates a good similarity between two clusters.

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]
```

```
>>> metrics.fowlkes_mallows_score(labels_true, labels_pred)
0.47140...
```

One can permute 0 and 1 in the predicted labels, rename 2 to 3 and get the same score:

```
>>> labels_pred = [1, 1, 0, 0, 3, 3]

>>> metrics.fowlkes_mallows_score(labels_true, labels_pred)
0.47140...
```

Perfect labeling is scored 1.0:

```
>>> labels_pred = labels_true[:]
>>> metrics.fowlkes_mallows_score(labels_true, labels_pred)
1.0
```

Bad (e.g. independent labelings) have zero scores:

```
>>> labels_true = [0, 1, 2, 0, 3, 4, 5, 1]
>>> labels_pred = [1, 1, 0, 0, 2, 2, 2, 2]
>>> metrics.fowlkes_mallows_score(labels_true, labels_pred)
0.0
```

Advantages

- **Random (uniform) label assignments have a FMI score close to 0.0** for any value of `n_clusters` and `n_samples` (which is not the case for raw Mutual Information or the V-measure for instance).
- **Upper-bounded at 1**: Values close to zero indicate two label assignments that are largely independent, while values close to one indicate significant agreement. Further, values of exactly 0 indicate **purely** independent label assignments and a FMI of exactly 1 indicates that the two label assignments are equal (with or without permutation).
- **No assumption is made on the cluster structure**: can be used to compare clustering algorithms such as k-means which assumes isotropic blob shapes with results of spectral clustering algorithms which can find cluster with "folded" shapes.

Drawbacks

- Contrary to inertia, **FMI-based measures require the knowledge of the ground truth classes** while almost never available in practice or requires manual assignment by human annotators (as in the supervised learning setting).

References

- E. B. Fowkles and C. L. Mallows, 1983. "A method for comparing two hierarchical clusterings". Journal of the American Statistical Association. http://wildfire.stat.ucla.edu/pdflibrary/fowlkes.pdf
- Wikipedia entry for the Fowlkes-Mallows Index

Silhouette Coefficient

If the ground truth labels are not known, evaluation must be performed using the model itself. The Silhouette Coefficient (*sklearn.metrics.silhouette_score*) is an example of such an evaluation, where a higher Silhouette Coefficient score relates to a model with better defined clusters. The Silhouette Coefficient is defined for each sample and is composed of two scores:

- **a**: The mean distance between a sample and all other points in the same class.
- **b**: The mean distance between a sample and all other points in the *next nearest cluster*.

The Silhouette Coefficient s for a single sample is then given as:

$$s = \frac{b - a}{max(a, b)}$$

The Silhouette Coefficient for a set of samples is given as the mean of the Silhouette Coefficient for each sample.

```
>>> from sklearn import metrics
>>> from sklearn.metrics import pairwise_distances
>>> from sklearn import datasets
>>> X, y = datasets.load_iris(return_X_y=True)
```

In normal usage, the Silhouette Coefficient is applied to the results of a cluster analysis.

```
>>> import numpy as np
>>> from sklearn.cluster import KMeans
>>> kmeans_model = KMeans(n_clusters=3, random_state=1).fit(X)
>>> labels = kmeans_model.labels_
>>> metrics.silhouette_score(X, labels, metric='euclidean')
0.55...
```

References

- Peter J. Rousseeuw (1987). "Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis". Computational and Applied Mathematics 20: 53–65. doi:10.1016/0377-0427(87)90125-7.

Advantages

- The score is bounded between -1 for incorrect clustering and +1 for highly dense clustering. Scores around zero indicate overlapping clusters.
- The score is higher when clusters are dense and well separated, which relates to a standard concept of a cluster.

Drawbacks

- The Silhouette Coefficient is generally higher for convex clusters than other concepts of clusters, such as density based clusters like those obtained through DBSCAN.

Examples:

- *Selecting the number of clusters with silhouette analysis on KMeans clustering* : In this example the silhouette analysis is used to choose an optimal value for n_clusters.

Calinski-Harabasz Index

If the ground truth labels are not known, the Calinski-Harabasz index (*sklearn.metrics.calinski_harabasz_score*) - also known as the Variance Ratio Criterion - can be used to evaluate the model, where a higher Calinski-Harabasz score relates to a model with better defined clusters.

The index is the ratio of the sum of between-clusters dispersion and of inter-cluster dispersion for all clusters (where dispersion is defined as the sum of distances squared):

```
>>> from sklearn import metrics
>>> from sklearn.metrics import pairwise_distances
>>> from sklearn import datasets
>>> X, y = datasets.load_iris(return_X_y=True)
```

In normal usage, the Calinski-Harabasz index is applied to the results of a cluster analysis:

```
>>> import numpy as np
>>> from sklearn.cluster import KMeans
>>> kmeans_model = KMeans(n_clusters=3, random_state=1).fit(X)
>>> labels = kmeans_model.labels_
>>> metrics.calinski_harabasz_score(X, labels)
561.62...
```

Advantages

- The score is higher when clusters are dense and well separated, which relates to a standard concept of a cluster.
- The score is fast to compute.

Drawbacks

- The Calinski-Harabasz index is generally higher for convex clusters than other concepts of clusters, such as density based clusters like those obtained through DBSCAN.

Mathematical formulation

For a set of data E of size n_E which has been clustered into k clusters, the Calinski-Harabasz score s is defined as the ratio of the between-clusters dispersion mean and the within-cluster dispersion:

$$s = \frac{\mathrm{tr}(B_k)}{\mathrm{tr}(W_k)} \times \frac{n_E - k}{k - 1}$$

where $\mathrm{tr}(B_k)$ is trace of the between group dispersion matrix and $\mathrm{tr}(W_k)$ is the trace of the within-cluster dispersion matrix defined by:

$$W_k = \sum_{q=1}^{k} \sum_{x \in C_q} (x - c_q)(x - c_q)^T$$

$$B_k = \sum_{q=1}^{k} n_q (c_q - c_E)(c_q - c_E)^T$$

with C_q the set of points in cluster q, c_q the center of cluster q, c_E the center of E, and n_q the number of points in cluster q.

References

- Caliński, T., & Harabasz, J. (1974). "A Dendrite Method for Cluster Analysis". Communications in Statistics-theory and Methods 3: 1-27. doi:10.1080/03610927408827101.

Davies-Bouldin Index

If the ground truth labels are not known, the Davies-Bouldin index (*sklearn.metrics.davies_bouldin_score*) can be used to evaluate the model, where a lower Davies-Bouldin index relates to a model with better separation between the clusters.

This index signifies the average 'similarity' between clusters, where the similarity is a measure that compares the distance between clusters with the size of the clusters themselves.

Zero is the lowest possible score. Values closer to zero indicate a better partition.

In normal usage, the Davies-Bouldin index is applied to the results of a cluster analysis as follows:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X = iris.data
>>> from sklearn.cluster import KMeans
>>> from sklearn.metrics import davies_bouldin_score
>>> kmeans = KMeans(n_clusters=3, random_state=1).fit(X)
>>> labels = kmeans.labels_
>>> davies_bouldin_score(X, labels)
0.6619...
```

Advantages

- The computation of Davies-Bouldin is simpler than that of Silhouette scores.
- The index is computed only quantities and features inherent to the dataset.

Drawbacks

- The Davies-Boulding index is generally higher for convex clusters than other concepts of clusters, such as density based clusters like those obtained from DBSCAN.
- The usage of centroid distance limits the distance metric to Euclidean space.

Mathematical formulation

The index is defined as the average similarity between each cluster C_i for $i = 1, ..., k$ and its most similar one C_j. In the context of this index, similarity is defined as a measure R_{ij} that trades off:

- s_i, the average distance between each point of cluster i and the centroid of that cluster – also know as cluster diameter.
- d_{ij}, the distance between cluster centroids i and j.

A simple choice to construct R_ij so that it is nonnegative and symmetric is:

$$R_{ij} = \frac{s_i + s_j}{d_{ij}}$$

Then the Davies-Bouldin index is defined as:

$$DB = \frac{1}{k} \sum_{i=1}^{k} \max_{i \neq j} R_{ij}$$

References

- Davies, David L.; Bouldin, Donald W. (1979). "A Cluster Separation Measure" IEEE Transactions on Pattern Analysis and Machine Intelligence. PAMI-1 (2): 224-227. doi:10.1109/TPAMI.1979.4766909.
- Halkidi, Maria; Batistakis, Yannis; Vazirgiannis, Michalis (2001). "On Clustering Validation Techniques" Journal of Intelligent Information Systems, 17(2-3), 107-145. doi:10.1023/A:1012801612483.
- Wikipedia entry for Davies-Bouldin index.

Contingency Matrix

Contingency matrix (`sklearn.metrics.cluster.contingency_matrix`) reports the intersection cardinality for every true/predicted cluster pair. The contingency matrix provides sufficient statistics for all clustering metrics where the samples are independent and identically distributed and one doesn't need to account for some instances not being clustered.

Here is an example:

```
>>> from sklearn.metrics.cluster import contingency_matrix
>>> x = ["a", "a", "a", "b", "b", "b"]
>>> y = [0, 0, 1, 1, 2, 2]
>>> contingency_matrix(x, y)
array([[2, 1, 0],
       [0, 1, 2]])
```

The first row of output array indicates that there are three samples whose true cluster is "a". Of them, two are in predicted cluster 0, one is in 1, and none is in 2. And the second row indicates that there are three samples whose true cluster is "b". Of them, none is in predicted cluster 0, one is in 1 and two are in 2.

A *confusion matrix* for classification is a square contingency matrix where the order of rows and columns correspond to a list of classes.

Advantages

- Allows to examine the spread of each true cluster across predicted clusters and vice versa.

- The contingency table calculated is typically utilized in the calculation of a similarity statistic (like the others listed in this document) between the two clusterings.

Drawbacks

- Contingency matrix is easy to interpret for a small number of clusters, but becomes very hard to interpret for a large number of clusters.
- It doesn't give a single metric to use as an objective for clustering optimisation.

References

- Wikipedia entry for contingency matrix

4.2.4 Biclustering

Biclustering can be performed with the module `sklearn.cluster.bicluster`. Biclustering algorithms simultaneously cluster rows and columns of a data matrix. These clusters of rows and columns are known as biclusters. Each determines a submatrix of the original data matrix with some desired properties.

For instance, given a matrix of shape `(10, 10)`, one possible bicluster with three rows and two columns induces a submatrix of shape `(3, 2)`:

```
>>> import numpy as np
>>> data = np.arange(100).reshape(10, 10)
>>> rows = np.array([0, 2, 3])[:, np.newaxis]
>>> columns = np.array([1, 2])
>>> data[rows, columns]
array([[ 1,  2],
       [21, 22],
       [31, 32]])
```

For visualization purposes, given a bicluster, the rows and columns of the data matrix may be rearranged to make the bicluster contiguous.

Algorithms differ in how they define biclusters. Some of the common types include:

- constant values, constant rows, or constant columns
- unusually high or low values
- submatrices with low variance
- correlated rows or columns

Algorithms also differ in how rows and columns may be assigned to biclusters, which leads to different bicluster structures. Block diagonal or checkerboard structures occur when rows and columns are divided into partitions.

If each row and each column belongs to exactly one bicluster, then rearranging the rows and columns of the data matrix reveals the biclusters on the diagonal. Here is an example of this structure where biclusters have higher average values than the other rows and columns:

In the checkerboard case, each row belongs to all column clusters, and each column belongs to all row clusters. Here is an example of this structure where the variance of the values within each bicluster is small:

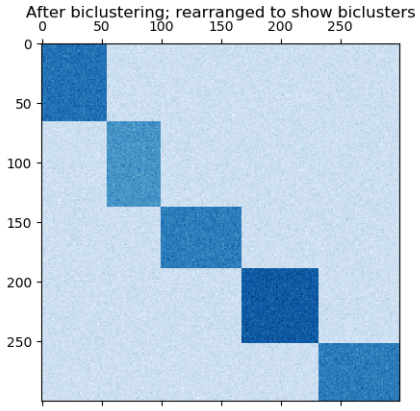


Fig. 5: An example of biclusters formed by partitioning rows and columns.

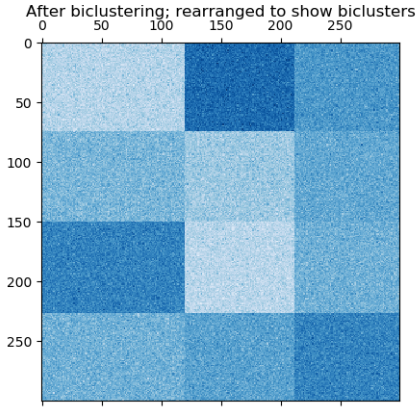


Fig. 6: An example of checkerboard biclusters.

After fitting a model, row and column cluster membership can be found in the `rows_` and `columns_` attributes. `rows_[i]` is a binary vector with nonzero entries corresponding to rows that belong to bicluster `i`. Similarly, `columns_[i]` indicates which columns belong to bicluster `i`.

Some models also have `row_labels_` and `column_labels_` attributes. These models partition the rows and columns, such as in the block diagonal and checkerboard bicluster structures.

Note: Biclustering has many other names in different fields including co-clustering, two-mode clustering, two-way clustering, block clustering, coupled two-way clustering, etc. The names of some algorithms, such as the Spectral Co-Clustering algorithm, reflect these alternate names.

Spectral Co-Clustering

The `SpectralCoclustering` algorithm finds biclusters with values higher than those in the corresponding other rows and columns. Each row and each column belongs to exactly one bicluster, so rearranging the rows and columns to make partitions contiguous reveals these high values along the diagonal:

Note: The algorithm treats the input data matrix as a bipartite graph: the rows and columns of the matrix correspond to the two sets of vertices, and each entry corresponds to an edge between a row and a column. The algorithm approximates the normalized cut of this graph to find heavy subgraphs.

Mathematical formulation

An approximate solution to the optimal normalized cut may be found via the generalized eigenvalue decomposition of the Laplacian of the graph. Usually this would mean working directly with the Laplacian matrix. If the original data matrix A has shape $m \times n$, the Laplacian matrix for the corresponding bipartite graph has shape $(m + n) \times (m + n)$. However, in this case it is possible to work directly with A, which is smaller and more efficient.

The input matrix A is preprocessed as follows:

$$A_n = R^{-1/2} A C^{-1/2}$$

Where R is the diagonal matrix with entry i equal to $\sum_j A_{ij}$ and C is the diagonal matrix with entry j equal to $\sum_i A_{ij}$.

The singular value decomposition, $A_n = U \Sigma V^\top$, provides the partitions of the rows and columns of A. A subset of the left singular vectors gives the row partitions, and a subset of the right singular vectors gives the column partitions.

The $\ell = \lceil \log_2 k \rceil$ singular vectors, starting from the second, provide the desired partitioning information. They are used to form the matrix Z:

$$Z = \begin{bmatrix} R^{-1/2} U \\ \\ C^{-1/2} V \end{bmatrix}$$

where the columns of U are $u_2, \ldots, u_{\ell+1}$, and similarly for V.

Then the rows of Z are clustered using *k-means*. The first `n_rows` labels provide the row partitioning, and the remaining `n_columns` labels provide the column partitioning.

Examples:

- *A demo of the Spectral Co-Clustering algorithm*: A simple example showing how to generate a data matrix with biclusters and apply this method to it.
- *Biclustering documents with the Spectral Co-clustering algorithm*: An example of finding biclusters in the twenty newsgroup dataset.

References:

- Dhillon, Inderjit S, 2001. Co-clustering documents and words using bipartite spectral graph partitioning.

Spectral Biclustering

The `SpectralBiclustering` algorithm assumes that the input data matrix has a hidden checkerboard structure. The rows and columns of a matrix with this structure may be partitioned so that the entries of any bicluster in the Cartesian product of row clusters and column clusters are approximately constant. For instance, if there are two row partitions and three column partitions, each row will belong to three biclusters, and each column will belong to two biclusters.

The algorithm partitions the rows and columns of a matrix so that a corresponding blockwise-constant checkerboard matrix provides a good approximation to the original matrix.

Mathematical formulation

The input matrix A is first normalized to make the checkerboard pattern more obvious. There are three possible methods:

1. *Independent row and column normalization*, as in Spectral Co-Clustering. This method makes the rows sum to a constant and the columns sum to a different constant.
2. **Bistochastization**: repeated row and column normalization until convergence. This method makes both rows and columns sum to the same constant.
3. **Log normalization**: the log of the data matrix is computed: $L = \log A$. Then the column mean $\overline{L_{i\cdot}}$, row mean $\overline{L_{\cdot j}}$, and overall mean $\overline{L_{\cdot\cdot}}$ of L are computed. The final matrix is computed according to the formula

$$K_{ij} = L_{ij} - \overline{L_{i\cdot}} - \overline{L_{\cdot j}} + \overline{L_{\cdot\cdot}}$$

After normalizing, the first few singular vectors are computed, just as in the Spectral Co-Clustering algorithm.

If log normalization was used, all the singular vectors are meaningful. However, if independent normalization or bistochastization were used, the first singular vectors, u_1 and v_1. are discarded. From now on, the "first" singular vectors refers to $u_2 \dots u_{p+1}$ and $v_2 \dots v_{p+1}$ except in the case of log normalization.

Given these singular vectors, they are ranked according to which can be best approximated by a piecewise-constant vector. The approximations for each vector are found using one-dimensional k-means and scored using the Euclidean distance. Some subset of the best left and right singular vector are selected. Next, the data is projected to this best subset of singular vectors and clustered.

For instance, if p singular vectors were calculated, the q best are found as described, where $q < p$. Let U be the matrix with columns the q best left singular vectors, and similarly V for the right. To partition the rows, the rows of A are projected to a q dimensional space: $A * V$. Treating the m rows of this $m \times q$ matrix as samples and clustering using k-means yields the row labels. Similarly, projecting the columns to $A^\top * U$ and clustering this $n \times q$ matrix yields the column labels.

Examples:

- *A demo of the Spectral Biclustering algorithm*: a simple example showing how to generate a checkerboard matrix and bicluster it.

References:

- Kluger, Yuval, et. al., 2003. Spectral biclustering of microarray data: coclustering genes and conditions.

Biclustering evaluation

There are two ways of evaluating a biclustering result: internal and external. Internal measures, such as cluster stability, rely only on the data and the result themselves. Currently there are no internal bicluster measures in scikit-learn. External measures refer to an external source of information, such as the true solution. When working with real data the true solution is usually unknown, but biclustering artificial data may be useful for evaluating algorithms precisely because the true solution is known.

To compare a set of found biclusters to the set of true biclusters, two similarity measures are needed: a similarity measure for individual biclusters, and a way to combine these individual similarities into an overall score.

To compare individual biclusters, several measures have been used. For now, only the Jaccard index is implemented:

$$J(A, B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

where A and B are biclusters, $|A \cap B|$ is the number of elements in their intersection. The Jaccard index achieves its minimum of 0 when the biclusters to not overlap at all and its maximum of 1 when they are identical.

Several methods have been developed to compare two sets of biclusters. For now, only `consensus_score` (Hochreiter et. al., 2010) is available:

1. Compute bicluster similarities for pairs of biclusters, one in each set, using the Jaccard index or a similar measure.
2. Assign biclusters from one set to another in a one-to-one fashion to maximize the sum of their similarities. This step is performed using the Hungarian algorithm.
3. The final sum of similarities is divided by the size of the larger set.

The minimum consensus score, 0, occurs when all pairs of biclusters are totally dissimilar. The maximum score, 1, occurs when both sets are identical.

References:

- Hochreiter, Bodenhofer, et. al., 2010. FABIA: factor analysis for bicluster acquisition.

4.2.5 Decomposing signals in components (matrix factorization problems)

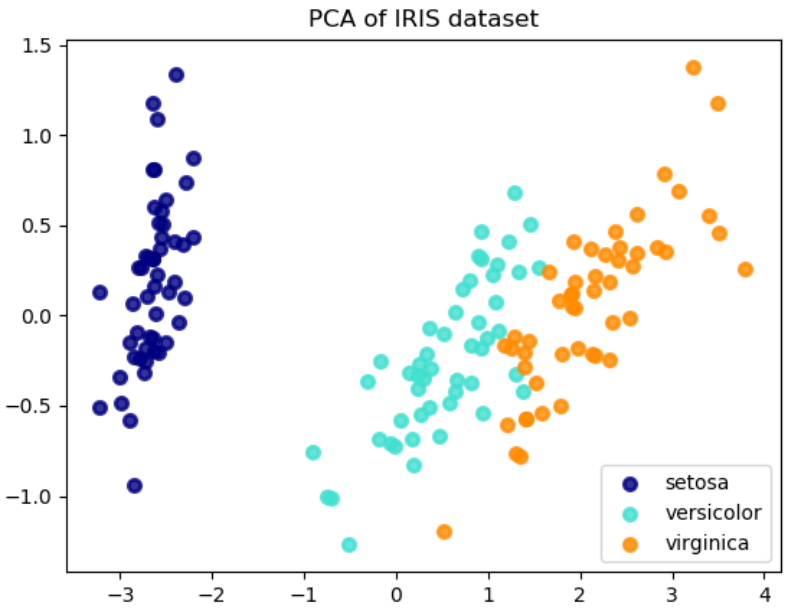
Principal component analysis (PCA)

Exact PCA and probabilistic interpretation

PCA is used to decompose a multivariate dataset in a set of successive orthogonal components that explain a maximum amount of the variance. In scikit-learn, `PCA` is implemented as a *transformer* object that learns n components in its `fit` method, and can be used on new data to project it on these components.

PCA centers but does not scale the input data for each feature before applying the SVD. The optional parameter `whiten=True` makes it possible to project the data onto the singular space while scaling each component to unit variance. This is often useful if the models down-stream make strong assumptions on the isotropy of the signal: this is for example the case for Support Vector Machines with the RBF kernel and the K-Means clustering algorithm.

Below is an example of the iris dataset, which is comprised of 4 features, projected on the 2 dimensions that explain most variance:



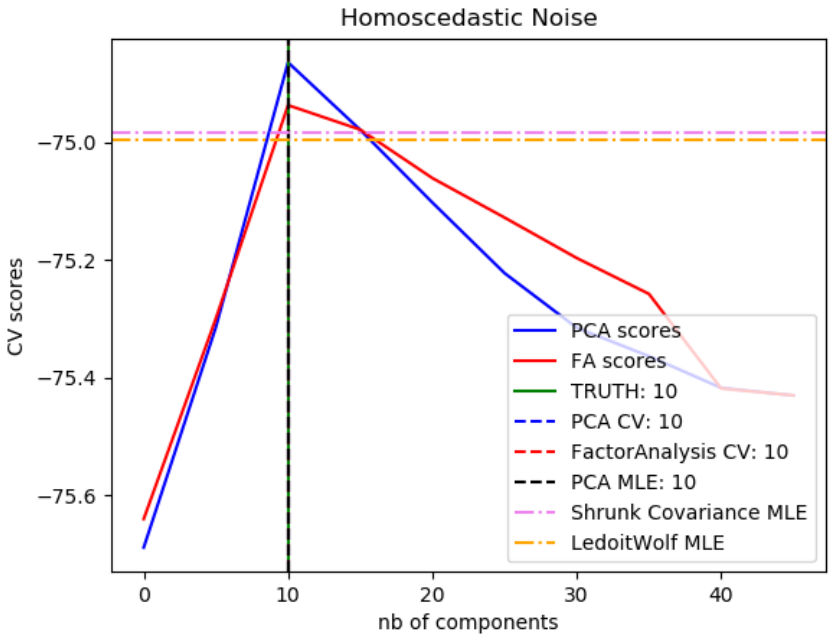
The `PCA` object also provides a probabilistic interpretation of the PCA that can give a likelihood of data based on the amount of variance it explains. As such it implements a *score* method that can be used in cross-validation:

Examples:

- *Comparison of LDA and PCA 2D projection of Iris dataset*
- *Model selection with Probabilistic PCA and Factor Analysis (FA)*

Incremental PCA

The `PCA` object is very useful, but has certain limitations for large datasets. The biggest limitation is that `PCA` only supports batch processing, which means all of the data to be processed must fit in main memory. The `IncrementalPCA` object uses a different form of processing and allows for partial computations which almost exactly match the results of `PCA` while processing the data in a minibatch fashion. `IncrementalPCA` makes it possible to implement out-of-core Principal Component Analysis either by:



- Using its `partial_fit` method on chunks of data fetched sequentially from the local hard drive or a network database.
- Calling its fit method on a sparse matrix or a memory mapped file using `numpy.memmap`.

IncrementalPCA only stores estimates of component and noise variances, in order update `explained_variance_ratio_` incrementally. This is why memory usage depends on the number of samples per batch, rather than the number of samples to be processed in the dataset.

As in *PCA*, *IncrementalPCA* centers but does not scale the input data for each feature before applying the SVD.

Examples:

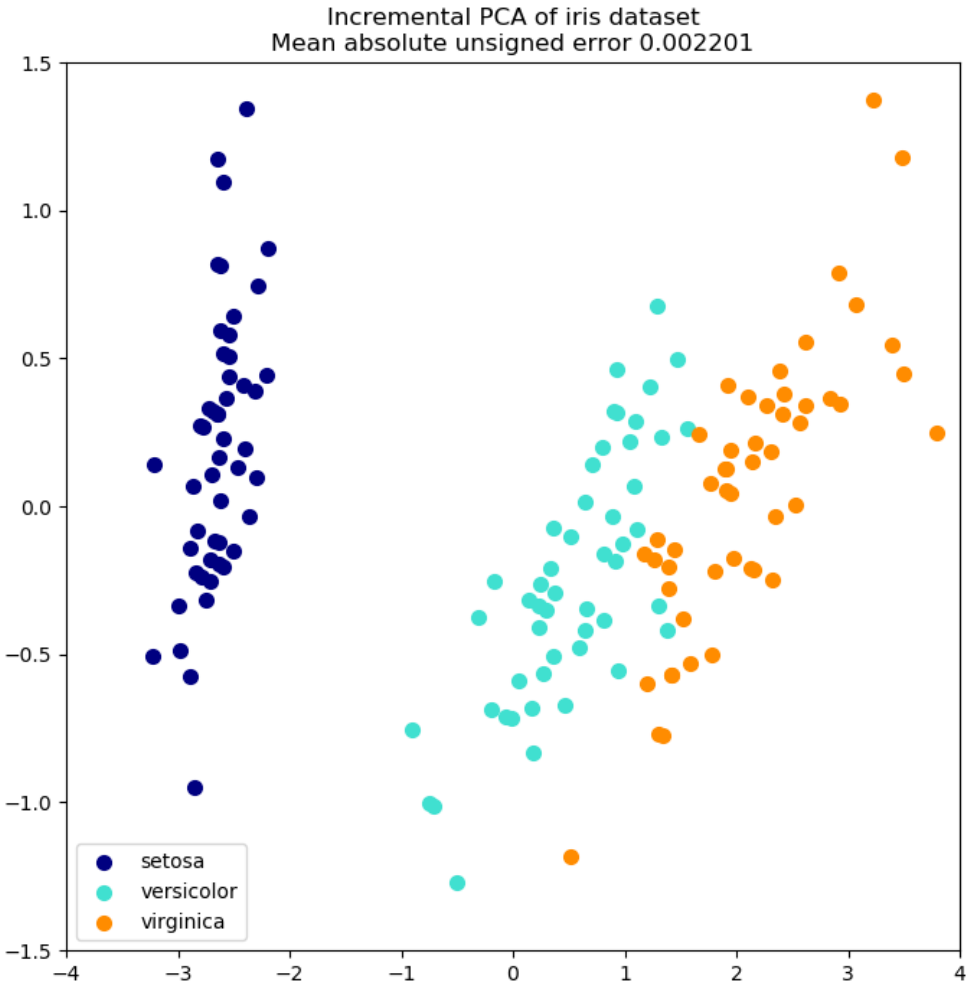
- *Incremental PCA*

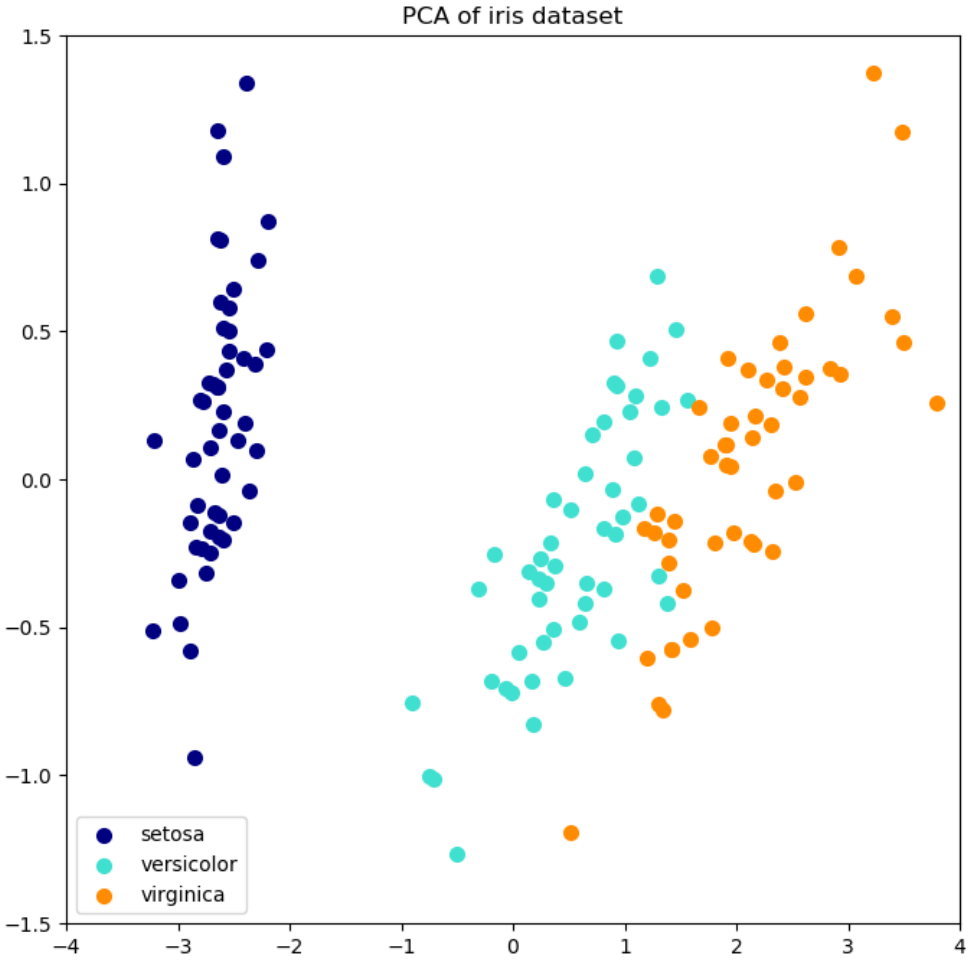
PCA using randomized SVD

It is often interesting to project data to a lower-dimensional space that preserves most of the variance, by dropping the singular vector of components associated with lower singular values.

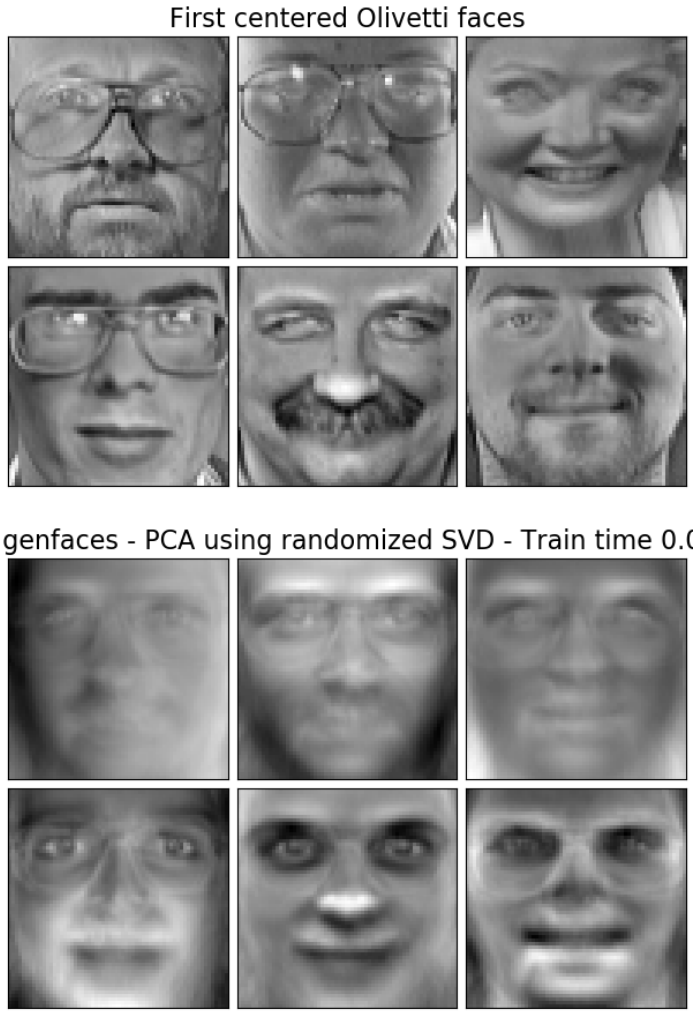
For instance, if we work with 64x64 pixel gray-level pictures for face recognition, the dimensionality of the data is 4096 and it is slow to train an RBF support vector machine on such wide data. Furthermore we know that the intrinsic dimensionality of the data is much lower than 4096 since all pictures of human faces look somewhat alike. The samples lie on a manifold of much lower dimension (say around 200 for instance). The PCA algorithm can be used to linearly transform the data while both reducing the dimensionality and preserve most of the explained variance at the same time.

The class *PCA* used with the optional parameter `svd_solver='randomized'` is very useful in that case: since we are going to drop most of the singular vectors it is much more efficient to limit the computation to an approximated estimate of the singular vectors we will keep to actually perform the transform.


Incremental PCA of iris dataset
Mean absolute unsigned error 0.002201
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−4
−3
−2
−1
0
1
2
3
4
setosa
versicolor
virginica


PCA of iris dataset
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−4
−3
−2
−1
0
1
2
3
4
setosa
versicolor
virginica

For instance, the following shows 16 sample portraits (centered around 0.0) from the Olivetti dataset. On the right hand side are the first 16 singular vectors reshaped as portraits. Since we only require the top 16 singular vectors of a dataset with size $n_{samples} = 400$ and $n_{features} = 64 \times 64 = 4096$, the computation time is less than 1s:



If we note $n_{\max} = \max(n_{\text{samples}}, n_{\text{features}})$ and $n_{\min} = \min(n_{\text{samples}}, n_{\text{features}})$, the time complexity of the randomized `PCA` is $O(n_{\max}^2 \cdot n_{\text{components}})$ instead of $O(n_{\max}^2 \cdot n_{\min})$ for the exact method implemented in `PCA`.

The memory footprint of randomized `PCA` is also proportional to $2 \cdot n_{\max} \cdot n_{\text{components}}$ instead of $n_{\max} \cdot n_{\min}$ for the exact method.

Note: the implementation of `inverse_transform` in `PCA` with `svd_solver='randomized'` is not the exact inverse transform of `transform` even when `whiten=False` (default).

Examples:

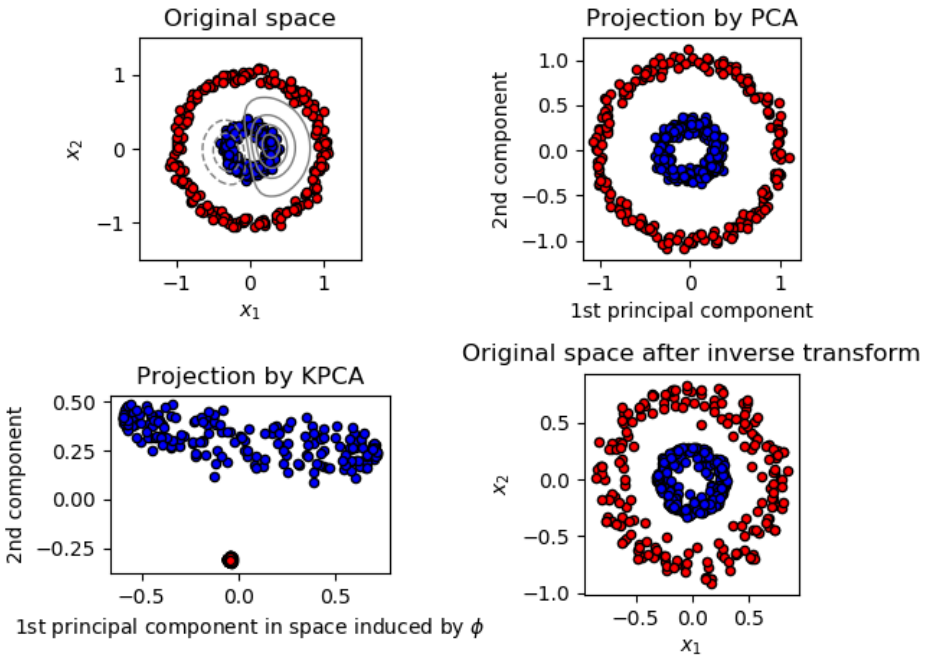
- *Faces recognition example using eigenfaces and SVMs*
- *Faces dataset decompositions*

References:

- "Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions" Halko, et al., 2009

Kernel PCA

`KernelPCA` is an extension of PCA which achieves non-linear dimensionality reduction through the use of kernels (see *Pairwise metrics, Affinities and Kernels*). It has many applications including denoising, compression and structured prediction (kernel dependency estimation). `KernelPCA` supports both `transform` and `inverse_transform`.



Examples:

- *Kernel PCA*

Sparse principal components analysis (SparsePCA and MiniBatchSparsePCA)

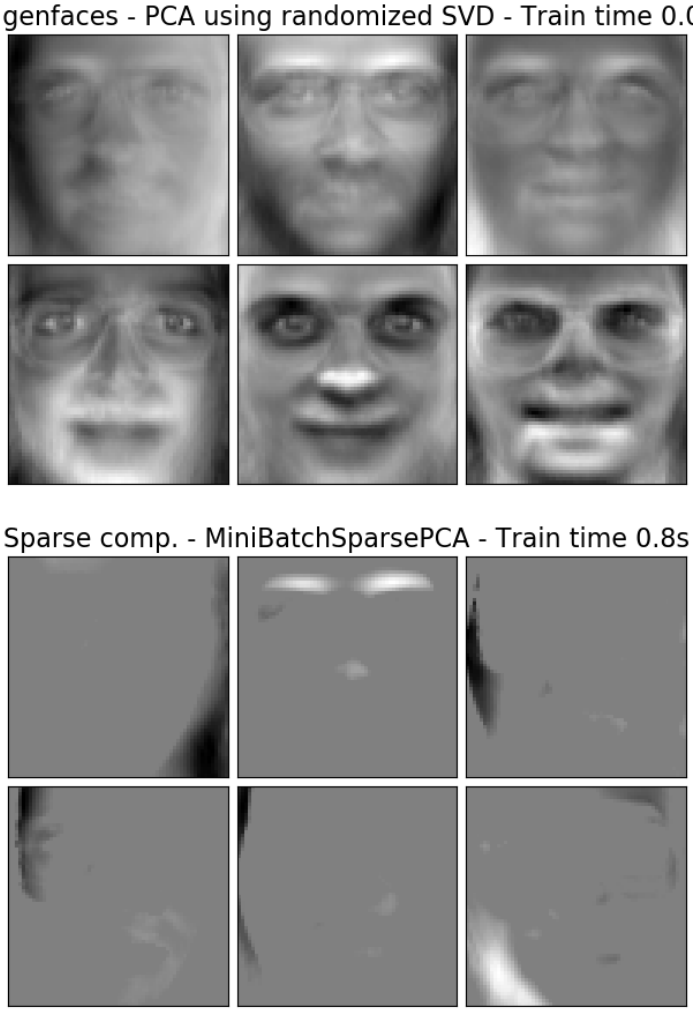
`SparsePCA` is a variant of PCA, with the goal of extracting the set of sparse components that best reconstruct the data.

Mini-batch sparse PCA (`MiniBatchSparsePCA`) is a variant of `SparsePCA` that is faster but less accurate. The increased speed is reached by iterating over small chunks of the set of features, for a given number of iterations.

Principal component analysis (`PCA`) has the disadvantage that the components extracted by this method have exclusively dense expressions, i.e. they have non-zero coefficients when expressed as linear combinations of the original variables. This can make interpretation difficult. In many cases, the real underlying components can be more naturally imagined as sparse vectors; for example in face recognition, components might naturally map to parts of faces.

Sparse principal components yields a more parsimonious, interpretable representation, clearly emphasizing which of the original features contribute to the differences between samples.

The following example illustrates 16 components extracted using sparse PCA from the Olivetti faces dataset. It can be seen how the regularization term induces many zeros. Furthermore, the natural structure of the data causes the non-zero coefficients to be vertically adjacent. The model does not enforce this mathematically: each component is a vector $h \in \mathbf{R}^{4096}$, and there is no notion of vertical adjacency except during the human-friendly visualization as 64x64 pixel images. The fact that the components shown below appear local is the effect of the inherent structure of the data, which makes such local patterns minimize reconstruction error. There exist sparsity-inducing norms that take into account adjacency and different kinds of structure; see [Jen09] for a review of such methods. For more details on how to use Sparse PCA, see the Examples section, below.



Note that there are many different formulations for the Sparse PCA problem. The one implemented here is based on [Mrl09] . The optimization problem solved is a PCA problem (dictionary learning) with an ℓ_1 penalty on the components:

$$(U^*, V^*) = \arg\min_{U,V} \frac{1}{2}||X - UV||_2^2 + \alpha||V||_1$$
$$\text{subject to } ||U_k||_2 = 1 \text{ for all } 0 \leq k < n_{components}$$

The sparsity-inducing ℓ_1 norm also prevents learning components from noise when few training samples are available. The degree of penalization (and thus sparsity) can be adjusted through the hyperparameter `alpha`. Small values lead

to a gently regularized factorization, while larger values shrink many coefficients to zero.

Note: While in the spirit of an online algorithm, the class `MiniBatchSparsePCA` does not implement `partial_fit` because the algorithm is online along the features direction, not the samples direction.

Examples:

- *Faces dataset decompositions*

References:

Truncated singular value decomposition and latent semantic analysis

`TruncatedSVD` implements a variant of singular value decomposition (SVD) that only computes the k largest singular values, where k is a user-specified parameter.

When truncated SVD is applied to term-document matrices (as returned by `CountVectorizer` or `TfidfVectorizer`), this transformation is known as latent semantic analysis (LSA), because it transforms such matrices to a "semantic" space of low dimensionality. In particular, LSA is known to combat the effects of synonymy and polysemy (both of which roughly mean there are multiple meanings per word), which cause term-document matrices to be overly sparse and exhibit poor similarity under measures such as cosine similarity.

Note: LSA is also known as latent semantic indexing, LSI, though strictly that refers to its use in persistent indexes for information retrieval purposes.

Mathematically, truncated SVD applied to training samples X produces a low-rank approximation X:

$$X \approx X_k = U_k \Sigma_k V_k^\top$$

After this operation, $U_k \Sigma_k^\top$ is the transformed training set with k features (called `n_components` in the API).

To also transform a test set X, we multiply it with V_k:

$$X' = X V_k$$

Note: Most treatments of LSA in the natural language processing (NLP) and information retrieval (IR) literature swap the axes of the matrix X so that it has shape `n_features` × `n_samples`. We present LSA in a different way that matches the scikit-learn API better, but the singular values found are the same.

`TruncatedSVD` is very similar to `PCA`, but differs in that it works on sample matrices X directly instead of their covariance matrices. When the columnwise (per-feature) means of X are subtracted from the feature values, truncated SVD on the resulting matrix is equivalent to PCA. In practical terms, this means that the `TruncatedSVD` transformer accepts `scipy.sparse` matrices without the need to densify them, as densifying may fill up memory even for medium-sized document collections.

While the `TruncatedSVD` transformer works with any (sparse) feature matrix, using it on tf–idf matrices is recommended over raw frequency counts in an LSA/document processing setting. In particular, sublinear scaling and inverse

document frequency should be turned on (`sublinear_tf=True, use_idf=True`) to bring the feature values closer to a Gaussian distribution, compensating for LSA's erroneous assumptions about textual data.

Examples:

- *Clustering text documents using k-means*

References:

- Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze (2008), *Introduction to Information Retrieval*, Cambridge University Press, chapter 18: Matrix decompositions & latent semantic indexing

Dictionary Learning

Sparse coding with a precomputed dictionary

The `SparseCoder` object is an estimator that can be used to transform signals into sparse linear combination of atoms from a fixed, precomputed dictionary such as a discrete wavelet basis. This object therefore does not implement a `fit` method. The transformation amounts to a sparse coding problem: finding a representation of the data as a linear combination of as few dictionary atoms as possible. All variations of dictionary learning implement the following transform methods, controllable via the `transform_method` initialization parameter:

- Orthogonal matching pursuit (*Orthogonal Matching Pursuit (OMP)*)
- Least-angle regression (*Least Angle Regression*)
- Lasso computed by least-angle regression
- Lasso using coordinate descent (*Lasso*)
- Thresholding

Thresholding is very fast but it does not yield accurate reconstructions. They have been shown useful in literature for classification tasks. For image reconstruction tasks, orthogonal matching pursuit yields the most accurate, unbiased reconstruction.

The dictionary learning objects offer, via the `split_code` parameter, the possibility to separate the positive and negative values in the results of sparse coding. This is useful when dictionary learning is used for extracting features that will be used for supervised learning, because it allows the learning algorithm to assign different weights to negative loadings of a particular atom, from to the corresponding positive loading.

The split code for a single sample has length `2 * n_components` and is constructed using the following rule: First, the regular code of length `n_components` is computed. Then, the first `n_components` entries of the `split_code` are filled with the positive part of the regular code vector. The second half of the split code is filled with the negative part of the code vector, only with a positive sign. Therefore, the split_code is non-negative.

Examples:

- *Sparse coding with a precomputed dictionary*

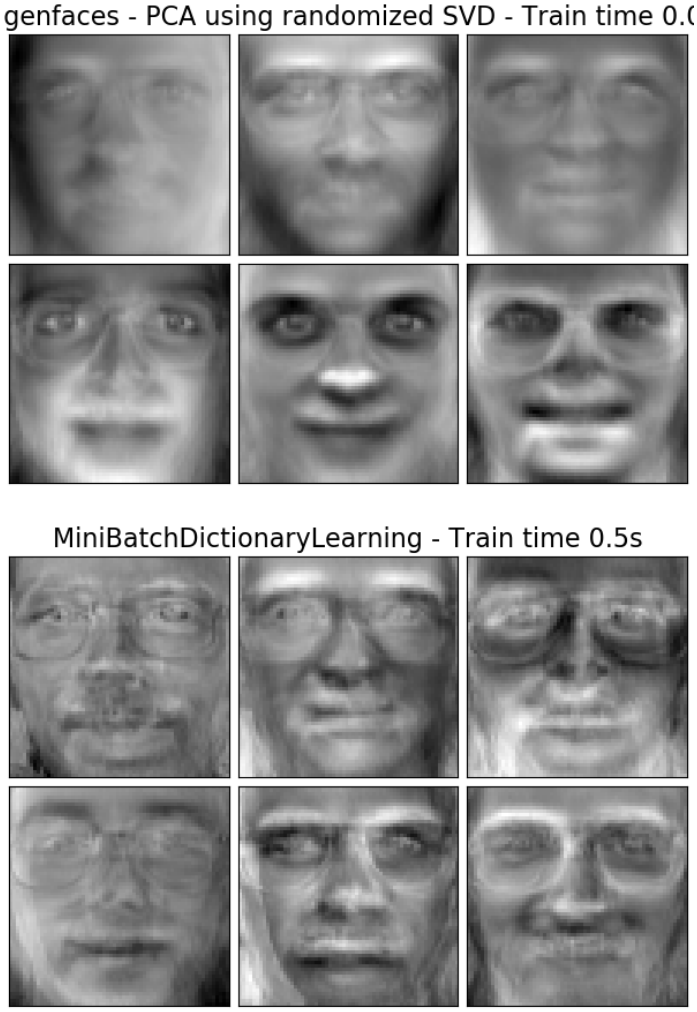
Generic dictionary learning

Dictionary learning (`DictionaryLearning`) is a matrix factorization problem that amounts to finding a (usually overcomplete) dictionary that will perform well at sparsely encoding the fitted data.

Representing data as sparse combinations of atoms from an overcomplete dictionary is suggested to be the way the mammalian primary visual cortex works. Consequently, dictionary learning applied on image patches has been shown to give good results in image processing tasks such as image completion, inpainting and denoising, as well as for supervised recognition tasks.

Dictionary learning is an optimization problem solved by alternatively updating the sparse code, as a solution to multiple Lasso problems, considering the dictionary fixed, and then updating the dictionary to best fit the sparse code.

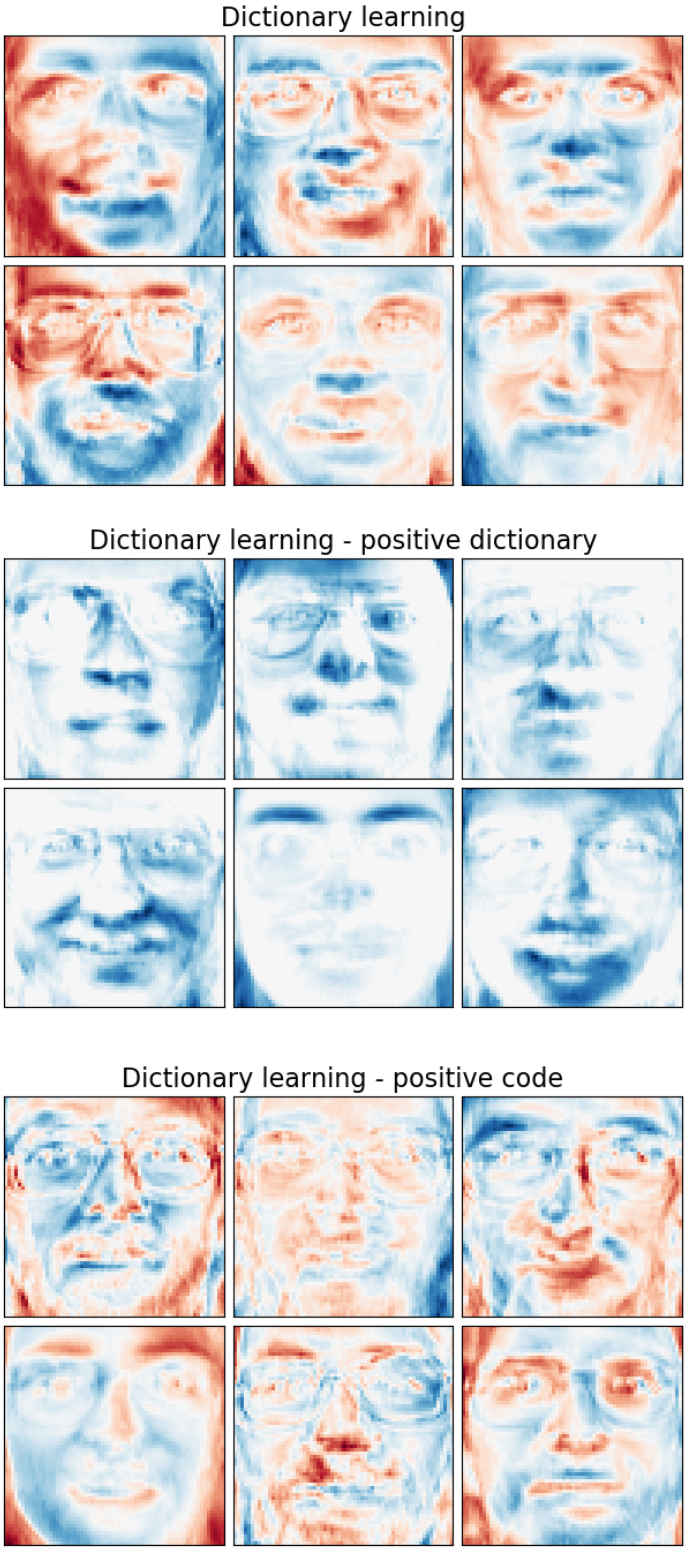
$$(U^*, V^*) = \arg\min_{U,V} \frac{1}{2}||X - UV||_2^2 + \alpha||U||_1$$
$$\text{subject to } ||V_k||_2 = 1 \text{ for all } 0 \leq k < n_{\text{atoms}}$$

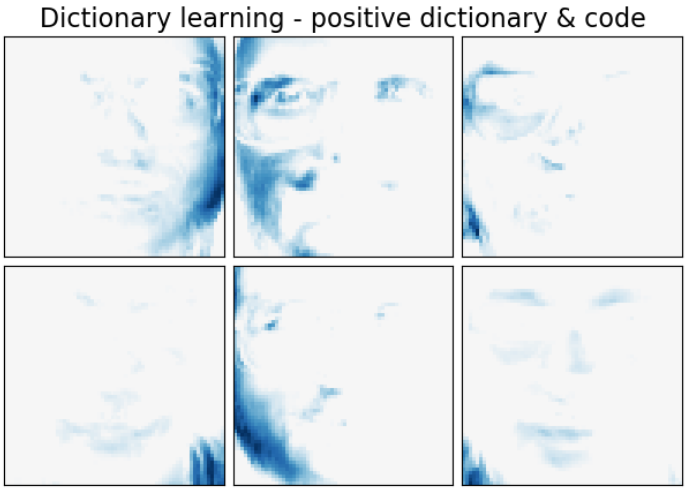


After using such a procedure to fit the dictionary, the transform is simply a sparse coding step that shares the same implementation with all dictionary learning objects (see *Sparse coding with a precomputed dictionary*).

It is also possible to constrain the dictionary and/or code to be positive to match constraints that may be present in the data. Below are the faces with different positivity constraints applied. Red indicates negative values, blue indicates

positive values, and white represents zeros.





The following image shows how a dictionary learned from 4x4 pixel image patches extracted from part of the image of a raccoon face looks like.

Examples:

- *Image denoising using dictionary learning*

References:

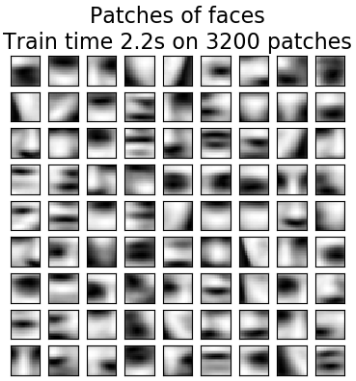
- "Online dictionary learning for sparse coding" J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009

Mini-batch dictionary learning

`MiniBatchDictionaryLearning` implements a faster, but less accurate version of the dictionary learning algorithm that is better suited for large datasets.

By default, `MiniBatchDictionaryLearning` divides the data into mini-batches and optimizes in an online manner by cycling over the mini-batches for the specified number of iterations. However, at the moment it does not implement a stopping condition.

The estimator also implements `partial_fit`, which updates the dictionary by iterating only once over a mini-batch. This can be used for online learning when the data is not readily available from the start, or for when the data does not fit into the memory.



Clustering for dictionary learning

Note that when using dictionary learning to extract a representation (e.g. for sparse coding) clustering can be a good proxy to learn the dictionary. For instance the `MiniBatchKMeans` estimator is computationally efficient and implements on-line learning with a `partial_fit` method.

Example: *Online learning of a dictionary of parts of faces*

Factor Analysis

In unsupervised learning we only have a dataset $X = \{x_1, x_2, \ldots, x_n\}$. How can this dataset be described mathematically? A very simple `continuous latent variable` model for X is

$$x_i = W h_i + \mu + \epsilon$$

The vector h_i is called "latent" because it is unobserved. ϵ is considered a noise term distributed according to a Gaussian with mean 0 and covariance Ψ (i.e. $\epsilon \sim \mathcal{N}(0, \Psi)$), μ is some arbitrary offset vector. Such a model is called "generative" as it describes how x_i is generated from h_i. If we use all the x_i's as columns to form a matrix $\mathbf{X}$ and all the h_i's as columns of a matrix $\mathbf{H}$ then we can write (with suitably defined $\mathbf{M}$ and $\mathbf{E}$):

$$\mathbf{X} = W\mathbf{H} + \mathbf{M} + \mathbf{E}$$

In other words, we *decomposed* matrix $\mathbf{X}$.

If h_i is given, the above equation automatically implies the following probabilistic interpretation:

$$p(x_i|h_i) = \mathcal{N}(W h_i + \mu, \Psi)$$

For a complete probabilistic model we also need a prior distribution for the latent variable h. The most straightforward assumption (based on the nice properties of the Gaussian distribution) is $h \sim \mathcal{N}(0, \mathbf{I})$. This yields a Gaussian as the marginal distribution of x:

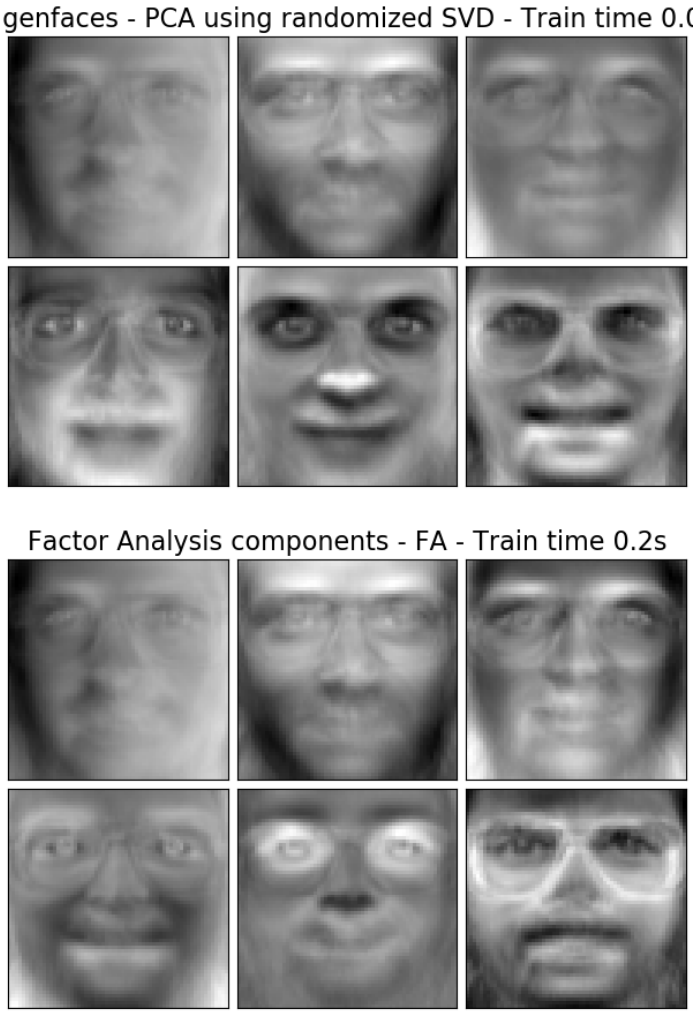
$$p(x) = \mathcal{N}(\mu, W W^T + \Psi)$$

Now, without any further assumptions the idea of having a latent variable h would be superfluous – x can be completely modelled with a mean and a covariance. We need to impose some more specific structure on one of these two parameters. A simple additional assumption regards the structure of the error covariance Ψ:

- $\Psi = \sigma^2\mathbf{I}$: This assumption leads to the probabilistic model of `PCA`.
- $\Psi = \mathrm{diag}(\psi_1, \psi_2, \ldots, \psi_n)$: This model is called `FactorAnalysis`, a classical statistical model. The matrix W is sometimes called the "factor loading matrix".

Both models essentially estimate a Gaussian with a low-rank covariance matrix. Because both models are probabilistic they can be integrated in more complex models, e.g. Mixture of Factor Analysers. One gets very different models (e.g. `FastICA`) if non-Gaussian priors on the latent variables are assumed.

Factor analysis *can* produce similar components (the columns of its loading matrix) to `PCA`. However, one can not make any general statements about these components (e.g. whether they are orthogonal):

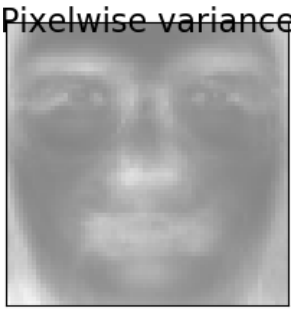


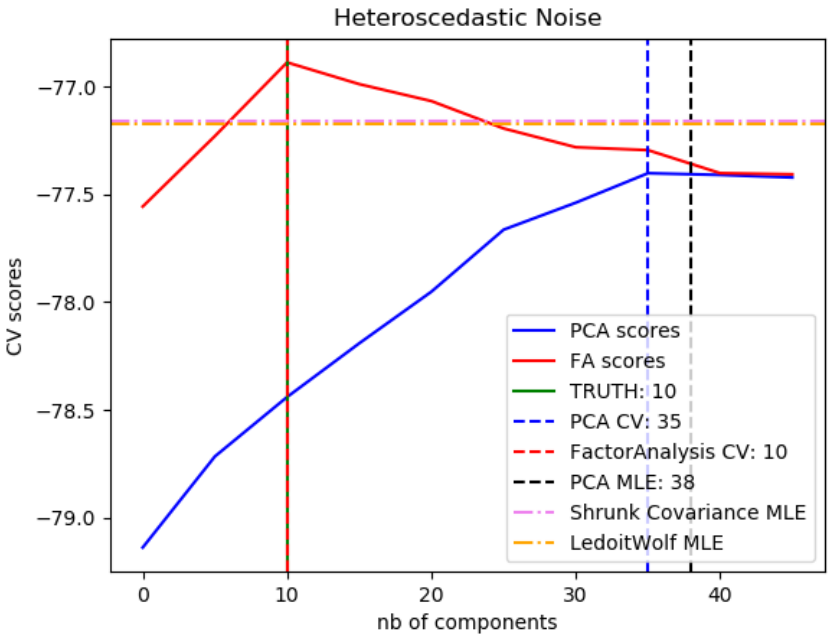
The main advantage for Factor Analysis over `PCA` is that it can model the variance in every direction of the input space independently (heteroscedastic noise):

This allows better model selection than probabilistic PCA in the presence of heteroscedastic noise:

Examples:

- *Model selection with Probabilistic PCA and Factor Analysis (FA)*

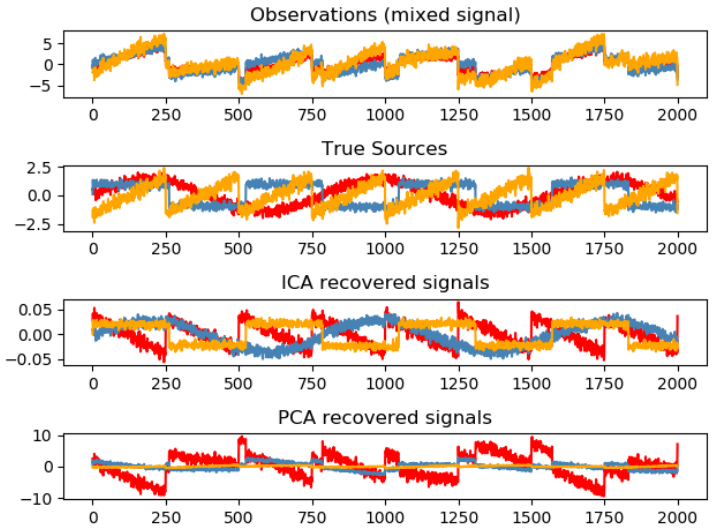

Pixelwise variance


Heteroscedastic Noise
−77.0
−77.5
−78.0
−78.5
−79.0
CV scores
0
10
20
30
40
nb of components
PCA scores
FA scores
TRUTH: 10
PCA CV: 35
FactorAnalysis CV: 10
PCA MLE: 38
Shrunk Covariance MLE
LedoitWolf MLE

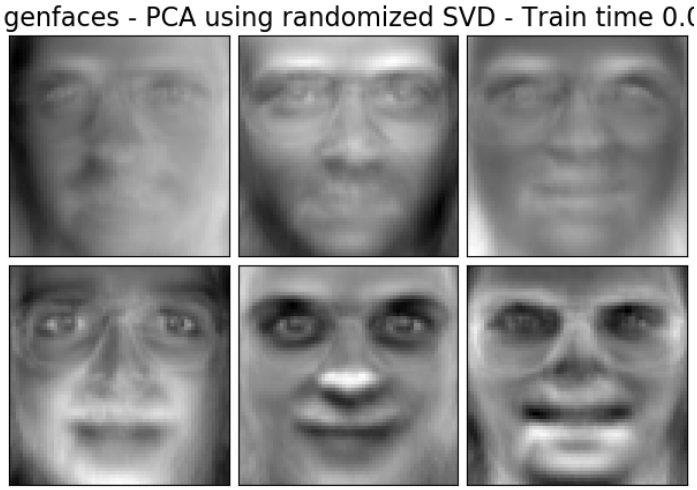
Independent component analysis (ICA)

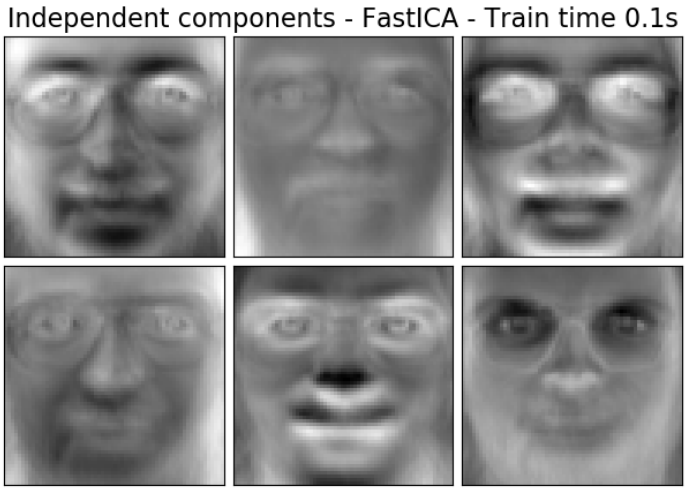
Independent component analysis separates a multivariate signal into additive subcomponents that are maximally independent. It is implemented in scikit-learn using the *Fast ICA* algorithm. Typically, ICA is not used for reducing dimensionality but for separating superimposed signals. Since the ICA model does not include a noise term, for the model to be correct, whitening must be applied. This can be done internally using the whiten argument or manually using one of the PCA variants.

It is classically used to separate mixed signals (a problem known as *blind source separation*), as in the example below:



ICA can also be used as yet another non linear decomposition that finds components with some sparsity:





Examples:

- *Blind source separation using FastICA*
- *FastICA on 2D point clouds*
- *Faces dataset decompositions*

Non-negative matrix factorization (NMF or NNMF)

NMF with the Frobenius norm

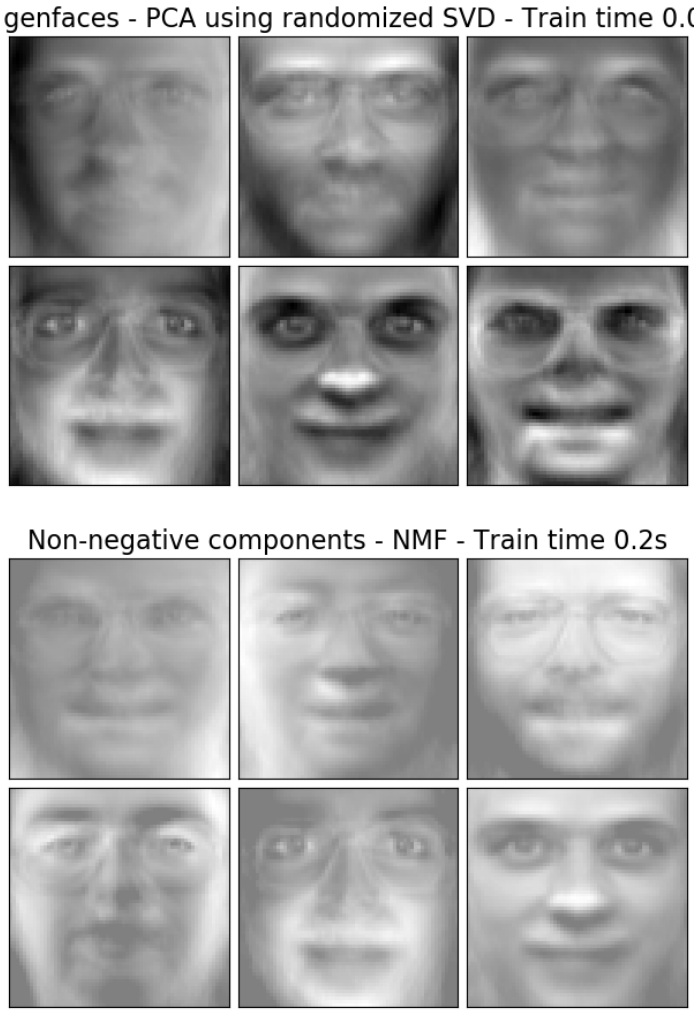
NMF[1] is an alternative approach to decomposition that assumes that the data and the components are non-negative. *NMF* can be plugged in instead of *PCA* or its variants, in the cases where the data matrix does not contain negative values. It finds a decomposition of samples X into two matrices W and H of non-negative elements, by optimizing the distance d between X and the matrix product WH. The most widely used distance function is the squared Frobenius norm, which is an obvious extension of the Euclidean norm to matrices:

$$d_{\mathrm{Fro}}(X, Y) = \frac{1}{2}||X - Y||_{\mathrm{Fro}}^2 = \frac{1}{2}\sum_{i,j}(X_{ij} - Y_{ij})^2$$

Unlike *PCA*, the representation of a vector is obtained in an additive fashion, by superimposing the components, without subtracting. Such additive models are efficient for representing images and text.

It has been observed in [Hoyer, 2004][2] that, when carefully constrained, *NMF* can produce a parts-based representation of the dataset, resulting in interpretable models. The following example displays 16 sparse components found by *NMF* from the images in the Olivetti faces dataset, in comparison with the PCA eigenfaces.

[1] "Learning the parts of objects by non-negative matrix factorization" D. Lee, S. Seung, 1999
[2] "Non-negative Matrix Factorization with Sparseness Constraints" P. Hoyer, 2004



The `init` attribute determines the initialization method applied, which has a great impact on the performance of the method. *NMF* implements the method Nonnegative Double Singular Value Decomposition. NNDSVD[4] is based on two SVD processes, one approximating the data matrix, the other approximating positive sections of the resulting partial SVD factors utilizing an algebraic property of unit rank matrices. The basic NNDSVD algorithm is better fit for sparse factorization. Its variants NNDSVDa (in which all zeros are set equal to the mean of all elements of the data), and NNDSVDar (in which the zeros are set to random perturbations less than the mean of the data divided by 100) are recommended in the dense case.

Note that the Multiplicative Update ('mu') solver cannot update zeros present in the initialization, so it leads to poorer results when used jointly with the basic NNDSVD algorithm which introduces a lot of zeros; in this case, NNDSVDa or NNDSVDar should be preferred.

NMF can also be initialized with correctly scaled random non-negative matrices by setting `init="random"`. An integer seed or a `RandomState` can also be passed to `random_state` to control reproducibility.

In *NMF*, L1 and L2 priors can be added to the loss function in order to regularize the model. The L2 prior uses the Frobenius norm, while the L1 prior uses an elementwise L1 norm. As in `ElasticNet`, we control the combination of L1 and L2 with the `l1_ratio` (ρ) parameter, and the intensity of the regularization with the `alpha` (α) parameter. Then the priors terms are:

$$\alpha\rho||W||_1 + \alpha\rho||H||_1 + \frac{\alpha(1-\rho)}{2}||W||_{\text{Fro}}^2 + \frac{\alpha(1-\rho)}{2}||H||_{\text{Fro}}^2$$

[4] "SVD based initialization: A head start for nonnegative matrix factorization" C. Boutsidis, E. Gallopoulos, 2008

and the regularized objective function is:

$$d_{\text{Fro}}(X, WH) + \alpha\rho||W||_1 + \alpha\rho||H||_1 + \frac{\alpha(1-\rho)}{2}||W||^2_{\text{Fro}} + \frac{\alpha(1-\rho)}{2}||H||^2_{\text{Fro}}$$

NMF regularizes both W and H. The public function *non_negative_factorization* allows a finer control through the `regularization` attribute, and may regularize only W, only H, or both.

NMF with a beta-divergence

As described previously, the most widely used distance function is the squared Frobenius norm, which is an obvious extension of the Euclidean norm to matrices:

$$d_{\text{Fro}}(X, Y) = \frac{1}{2}||X - Y||^2_{Fro} = \frac{1}{2}\sum_{i,j}(X_{ij} - Y_{ij})^2$$

Other distance functions can be used in NMF as, for example, the (generalized) Kullback-Leibler (KL) divergence, also referred as I-divergence:

$$d_{KL}(X, Y) = \sum_{i,j}(X_{ij}\log(\frac{X_{ij}}{Y_{ij}}) - X_{ij} + Y_{ij})$$

Or, the Itakura-Saito (IS) divergence:

$$d_{IS}(X, Y) = \sum_{i,j}(\frac{X_{ij}}{Y_{ij}} - \log(\frac{X_{ij}}{Y_{ij}}) - 1)$$

These three distances are special cases of the beta-divergence family, with $\beta = 2, 1, 0$ respectively[6]. The beta-divergence are defined by :

$$d_{\beta}(X, Y) = \sum_{i,j}\frac{1}{\beta(\beta-1)}(X_{ij}^{\beta} + (\beta-1)Y_{ij}^{\beta} - \beta X_{ij}Y_{ij}^{\beta-1})$$

Note that this definition is not valid if $\beta \in (0; 1)$, yet it can be continuously extended to the definitions of d_{KL} and d_{IS} respectively.

NMF implements two solvers, using Coordinate Descent ('cd')[5], and Multiplicative Update ('mu')[6]. The 'mu' solver can optimize every beta-divergence, including of course the Frobenius norm ($\beta = 2$), the (generalized) Kullback-Leibler divergence ($\beta = 1$) and the Itakura-Saito divergence ($\beta = 0$). Note that for $\beta \in (1; 2)$, the 'mu' solver is significantly faster than for other values of β. Note also that with a negative (or 0, i.e. 'itakura-saito') β, the input matrix cannot contain zero values.

The 'cd' solver can only optimize the Frobenius norm. Due to the underlying non-convexity of NMF, the different solvers may converge to different minima, even when optimizing the same distance function.

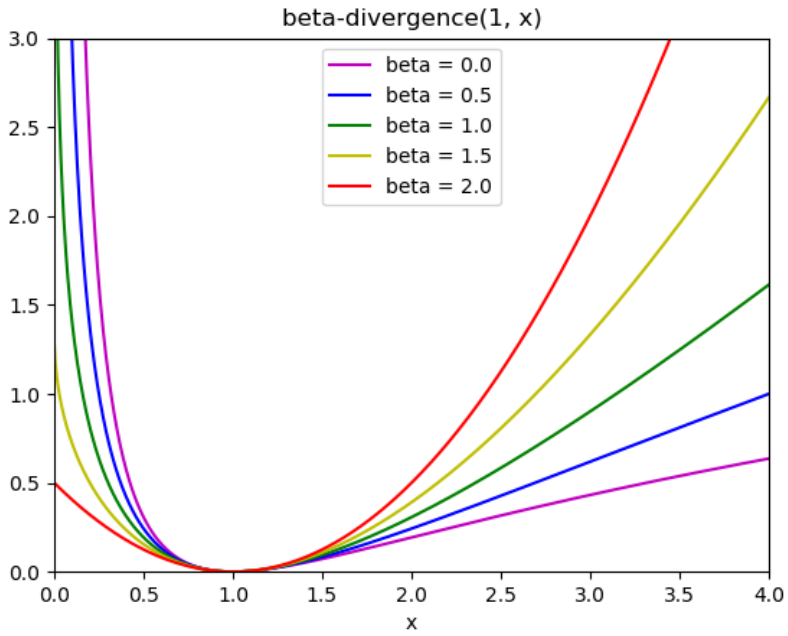
NMF is best used with the `fit_transform` method, which returns the matrix W. The matrix H is stored into the fitted model in the `components_` attribute; the method `transform` will decompose a new matrix X_new based on these stored components:

```
>>> import numpy as np
>>> X = np.array([[1, 1], [2, 1], [3, 1.2], [4, 1], [5, 0.8], [6, 1]])
>>> from sklearn.decomposition import NMF
```

(continues on next page)

[6] "Algorithms for nonnegative matrix factorization with the beta-divergence" C. Fevotte, J. Idier, 2011

[5] "Fast local algorithms for large scale nonnegative matrix and tensor factorizations." A. Cichocki, A. Phan, 2009



(continued from previous page)

```
>>> model = NMF(n_components=2, init='random', random_state=0)
>>> W = model.fit_transform(X)
>>> H = model.components_
>>> X_new = np.array([[1, 0], [1, 6.1], [1, 0], [1, 4], [3.2, 1], [0, 4]])
>>> W_new = model.transform(X_new)
```

Examples:

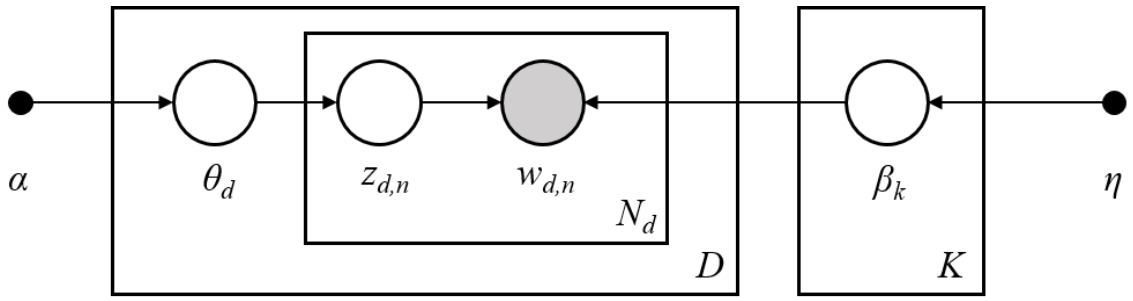
- *Faces dataset decompositions*
- *Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation*
- *Beta-divergence loss functions*

References:

Latent Dirichlet Allocation (LDA)

Latent Dirichlet Allocation is a generative probabilistic model for collections of discrete dataset such as text corpora. It is also a topic model that is used for discovering abstract topics from a collection of documents.

The graphical model of LDA is a three-level generative model:



Note on notations presented in the graphical model above, which can be found in Hoffman et al. (2013):

- The corpus is a collection of D documents.
- A document is a sequence of N words.
- There are K topics in the corpus.
- The boxes represent repeated sampling.

In the graphical model, each node is a random variable and has a role in the generative process. A shaded node indicates an observed variable and an unshaded node indicates a hidden (latent) variable. In this case, words in the corpus are the only data that we observe. The latent variables determine the random mixture of topics in the corpus and the distribution of words in the documents. The goal of LDA is to use the observed words to infer the hidden topic structure.

When modeling text corpora, the model assumes the following generative process for a corpus with D documents and K topics, with K corresponding to `n_components` in the API:

1. For each topic $k \in K$, draw $\beta_k \sim \text{Dirichlet}(\eta)$. This provides a distribution over the words, i.e. the probability of a word appearing in topic k. η corresponds to `topic_word_prior`.
2. For each document $d \in D$, draw the topic proportions $\theta_d \sim \text{Dirichlet}(\alpha)$. α corresponds to `doc_topic_prior`.
3. For each word i in document d:

a. Draw the topic assignment $z_{di} \sim \text{Multinomial}(\theta_d)$

b. Draw the observed word $w_{ij} \sim \text{Multinomial}(\beta_{z_{di}})$

For parameter estimation, the posterior distribution is:

$$p(z, \theta, \beta | w, \alpha, \eta) = \frac{p(z, \theta, \beta | \alpha, \eta)}{p(w | \alpha, \eta)}$$

Since the posterior is intractable, variational Bayesian method uses a simpler distribution $q(z, \theta, \beta | \lambda, \phi, \gamma)$ to approximate it, and those variational parameters λ, ϕ, γ are optimized to maximize the Evidence Lower Bound (ELBO):

$$\log\ P(w | \alpha, \eta) \geq L(w, \phi, \gamma, \lambda) \stackrel{\triangle}{=} E_q[\log\ p(w, z, \theta, \beta | \alpha, \eta)] - E_q[\log\ q(z, \theta, \beta)]$$

Maximizing ELBO is equivalent to minimizing the Kullback-Leibler(KL) divergence between $q(z, \theta, \beta)$ and the true posterior $p(z, \theta, \beta | w, \alpha, \eta)$.

`LatentDirichletAllocation` implements the online variational Bayes algorithm and supports both online and batch update methods. While the batch method updates variational variables after each full pass through the data, the online method updates variational variables from mini-batch data points.

Note: Although the online method is guaranteed to converge to a local optimum point, the quality of the optimum point and the speed of convergence may depend on mini-batch size and attributes related to learning rate setting.

When *LatentDirichletAllocation* is applied on a "document-term" matrix, the matrix will be decomposed into a "topic-term" matrix and a "document-topic" matrix. While "topic-term" matrix is stored as `components_` in the model, "document-topic" matrix can be calculated from `transform` method.

LatentDirichletAllocation also implements `partial_fit` method. This is used when data can be fetched sequentially.

Examples:

- *Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation*

References:

- "Latent Dirichlet Allocation" D. Blei, A. Ng, M. Jordan, 2003
- "Online Learning for Latent Dirichlet Allocation" M. Hoffman, D. Blei, F. Bach, 2010
- "Stochastic Variational Inference" M. Hoffman, D. Blei, C. Wang, J. Paisley, 2013

See also *Dimensionality reduction* for dimensionality reduction with Neighborhood Components Analysis.

4.2.6 Covariance estimation

Many statistical problems require the estimation of a population's covariance matrix, which can be seen as an estimation of data set scatter plot shape. Most of the time, such an estimation has to be done on a sample whose properties (size, structure, homogeneity) have a large influence on the estimation's quality. The *sklearn.covariance* package provides tools for accurately estimating a population's covariance matrix under various settings.

We assume that the observations are independent and identically distributed (i.i.d.).

Empirical covariance

The covariance matrix of a data set is known to be well approximated by the classical *maximum likelihood estimator* (or "empirical covariance"), provided the number of observations is large enough compared to the number of features (the variables describing the observations). More precisely, the Maximum Likelihood Estimator of a sample is an unbiased estimator of the corresponding population's covariance matrix.

The empirical covariance matrix of a sample can be computed using the *empirical_covariance* function of the package, or by fitting an *EmpiricalCovariance* object to the data sample with the *EmpiricalCovariance.fit* method. Be careful that results depend on whether the data are centered, so one may want to use the `assume_centered` parameter accurately. More precisely, if `assume_centered=False`, then the test set is supposed to have the same mean vector as the training set. If not, both should be centered by the user, and `assume_centered=True` should be used.

Examples:

- See *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood* for an example on how to fit an *EmpiricalCovariance* object to data.

Shrunk Covariance

Basic shrinkage

Despite being an unbiased estimator of the covariance matrix, the Maximum Likelihood Estimator is not a good estimator of the eigenvalues of the covariance matrix, so the precision matrix obtained from its inversion is not accurate. Sometimes, it even occurs that the empirical covariance matrix cannot be inverted for numerical reasons. To avoid such an inversion problem, a transformation of the empirical covariance matrix has been introduced: the `shrinkage`.

In scikit-learn, this transformation (with a user-defined shrinkage coefficient) can be directly applied to a pre-computed covariance with the *`shrunk_covariance`* method. Also, a shrunk estimator of the covariance can be fitted to data with a *`ShrunkCovariance`* object and its *`ShrunkCovariance.fit`* method. Again, results depend on whether the data are centered, so one may want to use the `assume_centered` parameter accurately.

Mathematically, this shrinkage consists in reducing the ratio between the smallest and the largest eigenvalues of the empirical covariance matrix. It can be done by simply shifting every eigenvalue according to a given offset, which is equivalent of finding the l2-penalized Maximum Likelihood Estimator of the covariance matrix. In practice, shrinkage boils down to a simple a convex transformation : $\Sigma_{\text{shrunk}} = (1-\alpha)\hat{\Sigma} + \alpha\frac{\text{Tr}\hat{\Sigma}}{p}\text{Id}$.

Choosing the amount of shrinkage, α amounts to setting a bias/variance trade-off, and is discussed below.

Examples:

- See *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood* for an example on how to fit a *`ShrunkCovariance`* object to data.

Ledoit-Wolf shrinkage

In their 2004 paper[1], O. Ledoit and M. Wolf propose a formula to compute the optimal shrinkage coefficient α that minimizes the Mean Squared Error between the estimated and the real covariance matrix.

The Ledoit-Wolf estimator of the covariance matrix can be computed on a sample with the *`ledoit_wolf`* function of the *`sklearn.covariance`* package, or it can be otherwise obtained by fitting a *`LedoitWolf`* object to the same sample.

Note: Case when population covariance matrix is isotropic

It is important to note that when the number of samples is much larger than the number of features, one would expect that no shrinkage would be necessary. The intuition behind this is that if the population covariance is full rank, when the number of sample grows, the sample covariance will also become positive definite. As a result, no shrinkage would necessary and the method should automatically do this.

This, however, is not the case in the Ledoit-Wolf procedure when the population covariance happens to be a multiple of the identity matrix. In this case, the Ledoit-Wolf shrinkage estimate approaches 1 as the number of samples increases. This indicates that the optimal estimate of the covariance matrix in the Ledoit-Wolf sense is multiple of the identity. Since the population covariance is already a multiple of the identity matrix, the Ledoit-Wolf solution is indeed a reasonable estimate.

---

[1] O. Ledoit and M. Wolf, "A Well-Conditioned Estimator for Large-Dimensional Covariance Matrices", Journal of Multivariate Analysis, Volume 88, Issue 2, February 2004, pages 365-411.

Examples:

- See *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood* for an example on how to fit a `LedoitWolf` object to data and for visualizing the performances of the Ledoit-Wolf estimator in terms of likelihood.

References:

Oracle Approximating Shrinkage

Under the assumption that the data are Gaussian distributed, Chen et al.[2] derived a formula aimed at choosing a shrinkage coefficient that yields a smaller Mean Squared Error than the one given by Ledoit and Wolf's formula. The resulting estimator is known as the Oracle Shrinkage Approximating estimator of the covariance.

The OAS estimator of the covariance matrix can be computed on a sample with the `oas` function of the `sklearn.covariance` package, or it can be otherwise obtained by fitting an `OAS` object to the same sample.

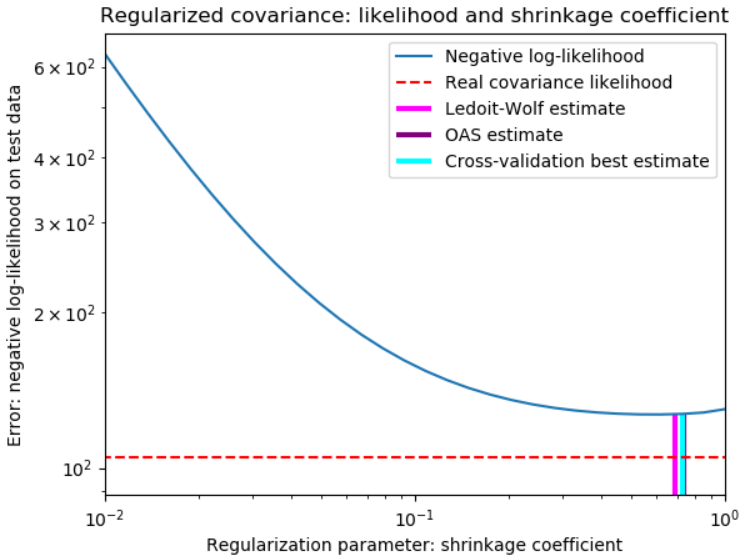


Fig. 7: Bias-variance trade-off when setting the shrinkage: comparing the choices of Ledoit-Wolf and OAS estimators

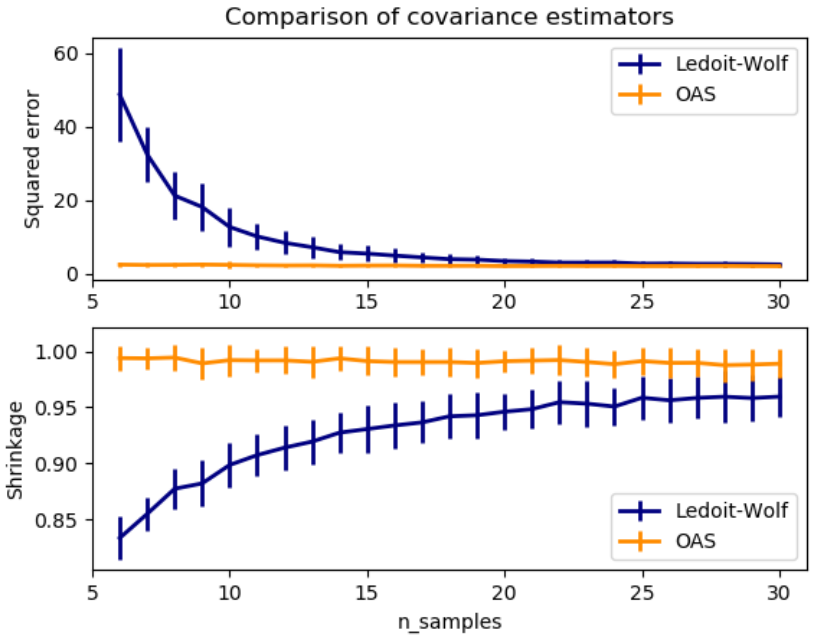
References:

Examples:

- See *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood* for an example on how to fit an `OAS` object to data.

[2] Chen et al., "Shrinkage Algorithms for MMSE Covariance Estimation", IEEE Trans. on Sign. Proc., Volume 58, Issue 10, October 2010.

- See *Ledoit-Wolf vs OAS estimation* to visualize the Mean Squared Error difference between a `LedoitWolf` and an `OAS` estimator of the covariance.



Sparse inverse covariance

The matrix inverse of the covariance matrix, often called the precision matrix, is proportional to the partial correlation matrix. It gives the partial independence relationship. In other words, if two features are independent conditionally on the others, the corresponding coefficient in the precision matrix will be zero. This is why it makes sense to estimate a sparse precision matrix: the estimation of the covariance matrix is better conditioned by learning independence relations from the data. This is known as *covariance selection*.

In the small-samples situation, in which `n_samples` is on the order of `n_features` or smaller, sparse inverse covariance estimators tend to work better than shrunk covariance estimators. However, in the opposite situation, or for very correlated data, they can be numerically unstable. In addition, unlike shrinkage estimators, sparse estimators are able to recover off-diagonal structure.

The `GraphicalLasso` estimator uses an l1 penalty to enforce sparsity on the precision matrix: the higher its `alpha` parameter, the more sparse the precision matrix. The corresponding `GraphicalLassoCV` object uses cross-validation to automatically set the `alpha` parameter.

Note: Structure recovery

Recovering a graphical structure from correlations in the data is a challenging thing. If you are interested in such recovery keep in mind that:

- Recovery is easier from a correlation matrix than a covariance matrix: standardize your observations before running `GraphicalLasso`
- If the underlying graph has nodes with much more connections than the average node, the algorithm will miss some of these connections.

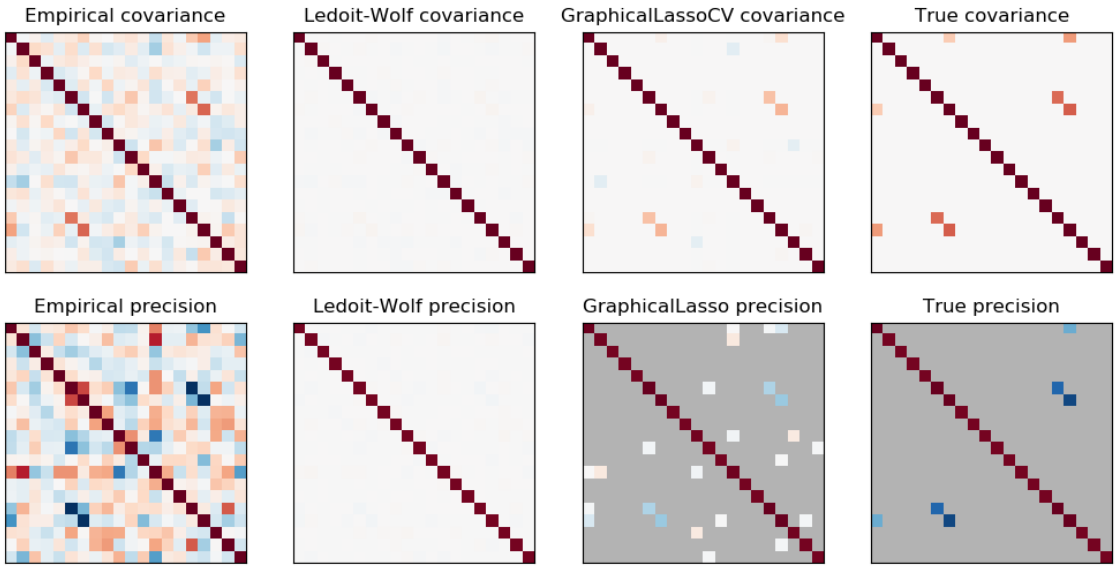


Fig. 8: *A comparison of maximum likelihood, shrinkage and sparse estimates of the covariance and precision matrix in the very small samples settings.*

- If your number of observations is not large compared to the number of edges in your underlying graph, you will not recover it.
- Even if you are in favorable recovery conditions, the alpha parameter chosen by cross-validation (e.g. using the `GraphicalLassoCV` object) will lead to selecting too many edges. However, the relevant edges will have heavier weights than the irrelevant ones.

The mathematical formulation is the following:

$$\hat{K} = \text{argmin}_K \left(\text{tr} S K - \text{logdet} K + \alpha \|K\|_1 \right)$$

Where K is the precision matrix to be estimated, and S is the sample covariance matrix. $\|K\|_1$ is the sum of the absolute values of off-diagonal coefficients of K. The algorithm employed to solve this problem is the GLasso algorithm, from the Friedman 2008 Biostatistics paper. It is the same algorithm as in the R `glasso` package.

Examples:

- *Sparse inverse covariance estimation*: example on synthetic data showing some recovery of a structure, and comparing to other covariance estimators.
- *Visualizing the stock market structure*: example on real stock market data, finding which symbols are most linked.

References:

- Friedman et al, "Sparse inverse covariance estimation with the graphical lasso", Biostatistics 9, pp 432, 2008

Robust Covariance Estimation

Real data sets are often subject to measurement or recording errors. Regular but uncommon observations may also appear for a variety of reasons. Observations which are very uncommon are called outliers. The empirical covariance estimator and the shrunk covariance estimators presented above are very sensitive to the presence of outliers in the data. Therefore, one should use robust covariance estimators to estimate the covariance of its real data sets. Alternatively, robust covariance estimators can be used to perform outlier detection and discard/downweight some observations according to further processing of the data.

The `sklearn.covariance` package implements a robust estimator of covariance, the Minimum Covariance Determinant[3].

Minimum Covariance Determinant

The Minimum Covariance Determinant estimator is a robust estimator of a data set's covariance introduced by P.J. Rousseeuw in[3]. The idea is to find a given proportion (h) of "good" observations which are not outliers and compute their empirical covariance matrix. This empirical covariance matrix is then rescaled to compensate the performed selection of observations ("consistency step"). Having computed the Minimum Covariance Determinant estimator, one can give weights to observations according to their Mahalanobis distance, leading to a reweighted estimate of the covariance matrix of the data set ("reweighting step").

Rousseeuw and Van Driessen[4] developed the FastMCD algorithm in order to compute the Minimum Covariance Determinant. This algorithm is used in scikit-learn when fitting an MCD object to data. The FastMCD algorithm also computes a robust estimate of the data set location at the same time.

Raw estimates can be accessed as `raw_location_` and `raw_covariance_` attributes of a *`MinCovDet`* robust covariance estimator object.

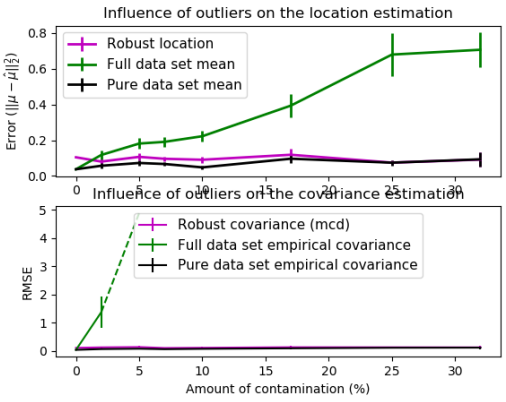
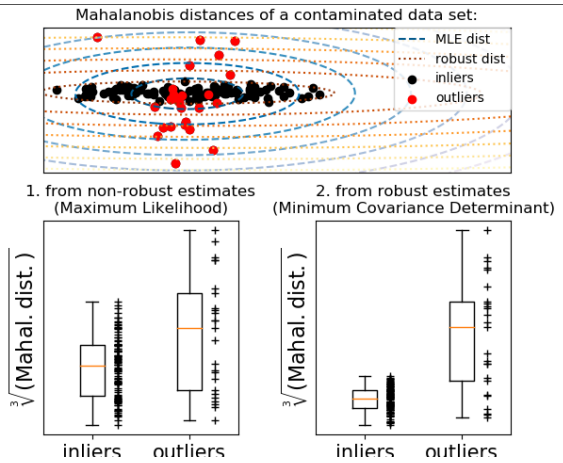
References:

Examples:

- See *Robust vs Empirical covariance estimate* for an example on how to fit a *`MinCovDet`* object to data and see how the estimate remains accurate despite the presence of outliers.
- See *Robust covariance estimation and Mahalanobis distances relevance* to visualize the difference between *`EmpiricalCovariance`* and *`MinCovDet`* covariance estimators in terms of Mahalanobis distance (so we get a better estimate of the precision matrix too).

[3] P. J. Rousseeuw. Least median of squares regression. J. Am Stat Ass, 79:871, 1984.

[4] A Fast Algorithm for the Minimum Covariance Determinant Estimator, 1999, American Statistical Association and the American Society for Quality, TECHNOMETRICS.

Influence of outliers on location and covariance estimates	Separating inliers from outliers using a Mahalanobis distance
	

4.2.7 Novelty and Outlier Detection

Many applications require being able to decide whether a new observation belongs to the same distribution as existing observations (it is an *inlier*), or should be considered as different (it is an *outlier*). Often, this ability is used to clean real data sets. Two important distinctions must be made:

outlier detection The training data contains outliers which are defined as observations that are far from the others. Outlier detection estimators thus try to fit the regions where the training data is the most concentrated, ignoring the deviant observations.

novelty detection The training data is not polluted by outliers and we are interested in detecting whether a **new** observation is an outlier. In this context an outlier is also called a novelty.

Outlier detection and novelty detection are both used for anomaly detection, where one is interested in detecting abnormal or unusual observations. Outlier detection is then also known as unsupervised anomaly detection and novelty detection as semi-supervised anomaly detection. In the context of outlier detection, the outliers/anomalies cannot form a dense cluster as available estimators assume that the outliers/anomalies are located in low density regions. On the contrary, in the context of novelty detection, novelties/anomalies can form a dense cluster as long as they are in a low density region of the training data, considered as normal in this context.

The scikit-learn project provides a set of machine learning tools that can be used both for novelty or outlier detection. This strategy is implemented with objects learning in an unsupervised way from the data:

```
estimator.fit(X_train)
```

new observations can then be sorted as inliers or outliers with a `predict` method:

```
estimator.predict(X_test)
```

Inliers are labeled 1, while outliers are labeled -1. The predict method makes use of a threshold on the raw scoring function computed by the estimator. This scoring function is accessible through the `score_samples` method, while the threshold can be controlled by the `contamination` parameter.

The `decision_function` method is also defined from the scoring function, in such a way that negative values are outliers and non-negative ones are inliers:

```
estimator.decision_function(X_test)
```

Note that *neighbors.LocalOutlierFactor* does not support predict, decision_function and score_samples methods by default but only a fit_predict method, as this estimator was originally meant to be applied for outlier detection. The scores of abnormality of the training samples are accessible through the negative_outlier_factor_ attribute.

If you really want to use *neighbors.LocalOutlierFactor* for novelty detection, i.e. predict labels or compute the score of abnormality of new unseen data, you can instantiate the estimator with the novelty parameter set to True before fitting the estimator. In this case, fit_predict is not available.

Warning: Novelty detection with Local Outlier Factor

When novelty is set to True be aware that you must only use predict, decision_function and score_samples on new unseen data and not on the training samples as this would lead to wrong results. The scores of abnormality of the training samples are always accessible through the negative_outlier_factor_ attribute.

The behavior of *neighbors.LocalOutlierFactor* is summarized in the following table.

Method	Outlier detection	Novelty detection
fit_predict	OK	Not available
predict	Not available	Use only on new data
decision_function	Not available	Use only on new data
score_samples	Use negative_outlier_factor_	Use only on new data

Overview of outlier detection methods

A comparison of the outlier detection algorithms in scikit-learn. Local Outlier Factor (LOF) does not show a decision boundary in black as it has no predict method to be applied on new data when it is used for outlier detection.

ensemble.IsolationForest and *neighbors.LocalOutlierFactor* perform reasonably well on the data sets considered here. The *svm.OneClassSVM* is known to be sensitive to outliers and thus does not perform very well for outlier detection. Finally, *covariance.EllipticEnvelope* assumes the data is Gaussian and learns an ellipse. For more details on the different estimators refer to the example *Comparing anomaly detection algorithms for outlier detection on toy datasets* and the sections hereunder.

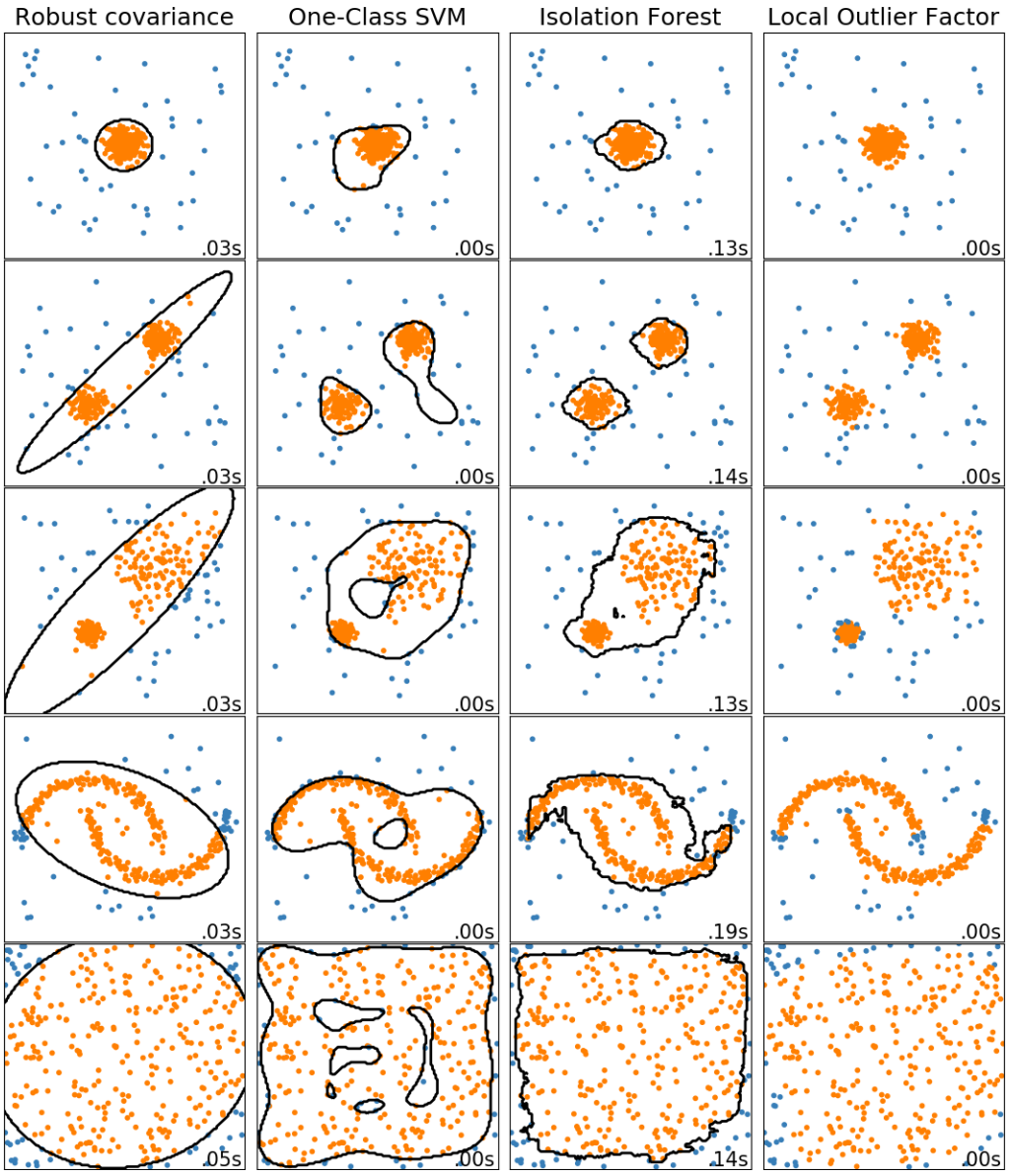
Examples:

- See *Comparing anomaly detection algorithms for outlier detection on toy datasets* for a comparison of the *svm.OneClassSVM*, the *ensemble.IsolationForest*, the *neighbors.LocalOutlierFactor* and *covariance.EllipticEnvelope*.

Novelty Detection

Consider a data set of n observations from the same distribution described by p features. Consider now that we add one more observation to that data set. Is the new observation so different from the others that we can doubt it is regular? (i.e. does it come from the same distribution?) Or on the contrary, is it so similar to the other that we cannot distinguish it from the original observations? This is the question addressed by the novelty detection tools and methods.

In general, it is about to learn a rough, close frontier delimiting the contour of the initial observations distribution, plotted in embedding p-dimensional space. Then, if further observations lay within the frontier-delimited subspace,


Robust covariance
One-Class SVM
Isolation Forest
Local Outlier Factor
.03s
.00s
.13s
.00s
.03s
.00s
.14s
.00s
.03s
.00s
.13s
.00s
.03s
.00s
.19s
.00s
.05s
.00s
.14s
.00s

they are considered as coming from the same population than the initial observations. Otherwise, if they lay outside the frontier, we can say that they are abnormal with a given confidence in our assessment.

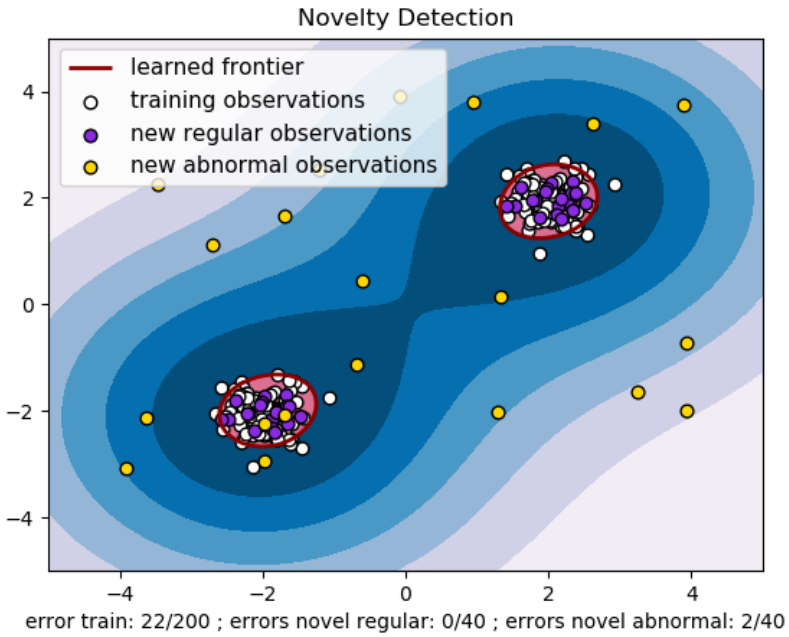
The One-Class SVM has been introduced by Schölkopf et al. for that purpose and implemented in the *Support Vector Machines* module in the `svm.OneClassSVM` object. It requires the choice of a kernel and a scalar parameter to define a frontier. The RBF kernel is usually chosen although there exists no exact formula or algorithm to set its bandwidth parameter. This is the default in the scikit-learn implementation. The ν parameter, also known as the margin of the One-Class SVM, corresponds to the probability of finding a new, but regular, observation outside the frontier.

References:

- Estimating the support of a high-dimensional distribution Schölkopf, Bernhard, et al. Neural computation 13.7 (2001): 1443-1471.

Examples:

- See *One-class SVM with non-linear kernel (RBF)* for visualizing the frontier learned around some data by a `svm.OneClassSVM` object.
- *Species distribution modeling*



Outlier Detection

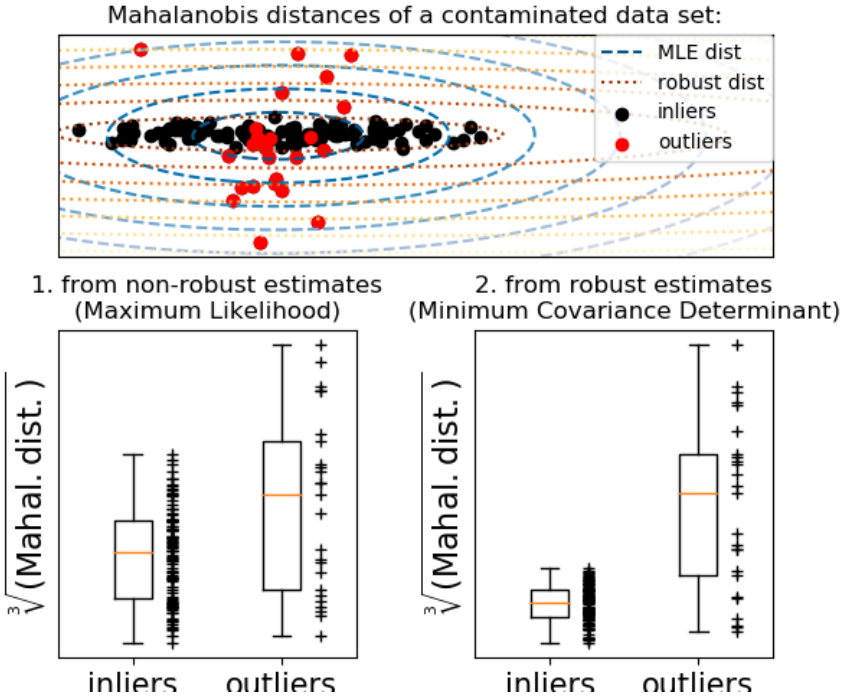
Outlier detection is similar to novelty detection in the sense that the goal is to separate a core of regular observations from some polluting ones, called *outliers*. Yet, in the case of outlier detection, we don't have a clean data set representing the population of regular observations that can be used to train any tool.

Fitting an elliptic envelope

One common way of performing outlier detection is to assume that the regular data come from a known distribution (e.g. data are Gaussian distributed). From this assumption, we generally try to define the "shape" of the data, and can define outlying observations as observations which stand far enough from the fit shape.

The scikit-learn provides an object `covariance.EllipticEnvelope` that fits a robust covariance estimate to the data, and thus fits an ellipse to the central data points, ignoring points outside the central mode.

For instance, assuming that the inlier data are Gaussian distributed, it will estimate the inlier location and covariance in a robust way (i.e. without being influenced by outliers). The Mahalanobis distances obtained from this estimate is used to derive a measure of outlyingness. This strategy is illustrated below.



Examples:

- See *Robust covariance estimation and Mahalanobis distances relevance* for an illustration of the difference between using a standard (`covariance.EmpiricalCovariance`) or a robust estimate (`covariance.MinCovDet`) of location and covariance to assess the degree of outlyingness of an observation.

References:

- Rousseeuw, P.J., Van Driessen, K. "A fast algorithm for the minimum covariance determinant estimator" Technometrics 41(3), 212 (1999)

Isolation Forest

One efficient way of performing outlier detection in high-dimensional datasets is to use random forests. The *ensemble.IsolationForest* 'isolates' observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature.

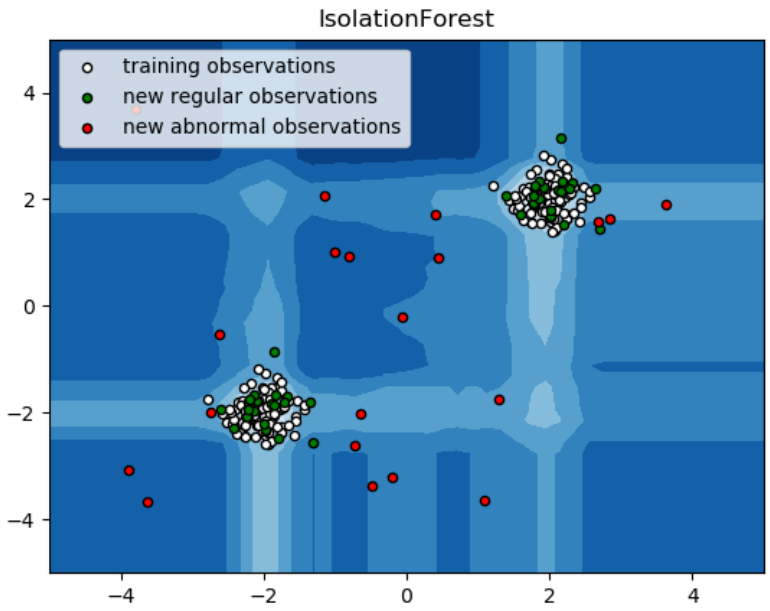
Since recursive partitioning can be represented by a tree structure, the number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node.

This path length, averaged over a forest of such random trees, is a measure of normality and our decision function.

Random partitioning produces noticeably shorter paths for anomalies. Hence, when a forest of random trees collectively produce shorter path lengths for particular samples, they are highly likely to be anomalies.

The implementation of *ensemble.IsolationForest* is based on an ensemble of *tree.ExtraTreeRegressor*. Following Isolation Forest original paper, the maximum depth of each tree is set to $\lceil \log_2(n) \rceil$ where n is the number of samples used to build the tree (see (Liu et al., 2008) for more details).

This algorithm is illustrated below.



The *ensemble.IsolationForest* supports `warm_start=True` which allows you to add more trees to an already fitted model:

```
>>> from sklearn.ensemble import IsolationForest
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [0, 0], [-20, 50], [3, 5]])
>>> clf = IsolationForest(n_estimators=10, warm_start=True)
>>> clf.fit(X)  # fit 10 trees
>>> clf.set_params(n_estimators=20)  # add 10 more trees
>>> clf.fit(X)  # fit the added trees
```

Examples:

- See *IsolationForest example* for an illustration of the use of IsolationForest.
- See *Comparing anomaly detection algorithms for outlier detection on toy datasets* for a comparison of `ensemble.IsolationForest` with `neighbors.LocalOutlierFactor`, `svm.OneClassSVM` (tuned to perform like an outlier detection method) and a covariance-based outlier detection with `covariance.EllipticEnvelope`.

References:

- Liu, Fei Tony, Ting, Kai Ming and Zhou, Zhi-Hua. "Isolation forest." Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on.

Local Outlier Factor

Another efficient way to perform outlier detection on moderately high dimensional datasets is to use the Local Outlier Factor (LOF) algorithm.

The `neighbors.LocalOutlierFactor` (LOF) algorithm computes a score (called local outlier factor) reflecting the degree of abnormality of the observations. It measures the local density deviation of a given data point with respect to its neighbors. The idea is to detect the samples that have a substantially lower density than their neighbors.

In practice the local density is obtained from the k-nearest neighbors. The LOF score of an observation is equal to the ratio of the average local density of his k-nearest neighbors, and its own local density: a normal instance is expected to have a local density similar to that of its neighbors, while abnormal data are expected to have much smaller local density.

The number k of neighbors considered, (alias parameter n_neighbors) is typically chosen 1) greater than the minimum number of objects a cluster has to contain, so that other objects can be local outliers relative to this cluster, and 2) smaller than the maximum number of close by objects that can potentially be local outliers. In practice, such informations are generally not available, and taking n_neighbors=20 appears to work well in general. When the proportion of outliers is high (i.e. greater than 10 %, as in the example below), n_neighbors should be greater (n_neighbors=35 in the example below).

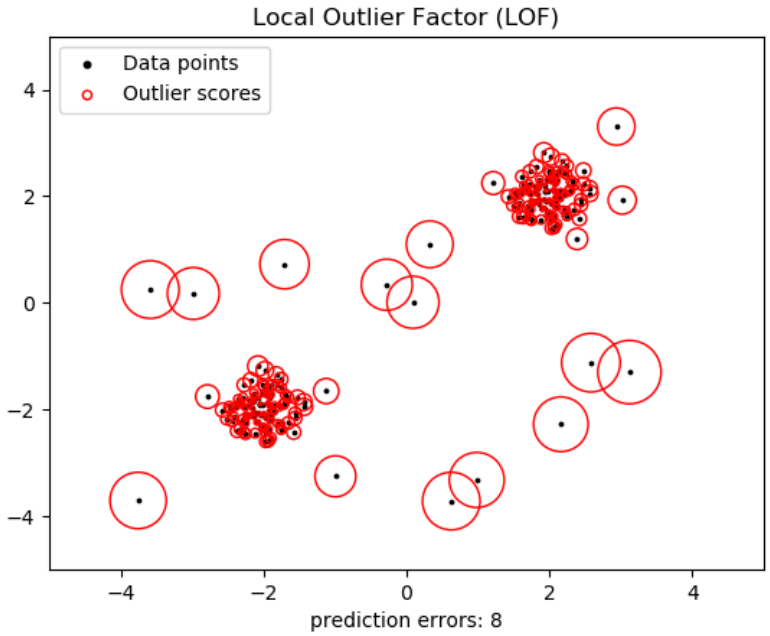
The strength of the LOF algorithm is that it takes both local and global properties of datasets into consideration: it can perform well even in datasets where abnormal samples have different underlying densities. The question is not, how isolated the sample is, but how isolated it is with respect to the surrounding neighborhood.

When applying LOF for outlier detection, there are no `predict`, `decision_function` and `score_samples` methods but only a `fit_predict` method. The scores of abnormality of the training samples are accessible through the `negative_outlier_factor_` attribute. Note that `predict`, `decision_function` and `score_samples` can be used on new unseen data when LOF is applied for novelty detection, i.e. when the `novelty` parameter is set to `True`. See *Novelty detection with Local Outlier Factor*.

This strategy is illustrated below.

Examples:

- See *Outlier detection with Local Outlier Factor (LOF)* for an illustration of the use of `neighbors.LocalOutlierFactor`.



- See *Comparing anomaly detection algorithms for outlier detection on toy datasets* for a comparison with other anomaly detection methods.

References:

- Breunig, Kriegel, Ng, and Sander (2000) LOF: identifying density-based local outliers. Proc. ACM SIGMOD

Novelty detection with Local Outlier Factor

To use *neighbors.LocalOutlierFactor* for novelty detection, i.e. predict labels or compute the score of abnormality of new unseen data, you need to instantiate the estimator with the `novelty` parameter set to `True` before fitting the estimator:

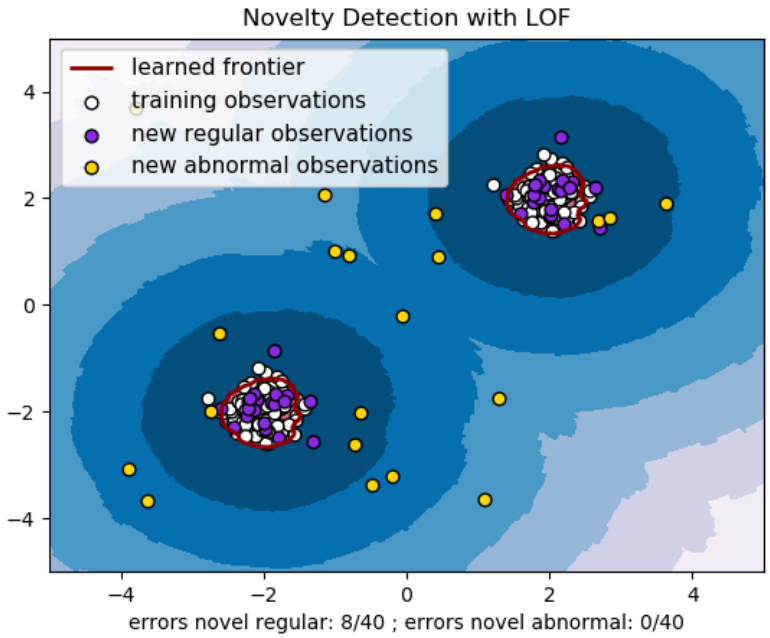
```
lof = LocalOutlierFactor(novelty=True)
lof.fit(X_train)
```

Note that `fit_predict` is not available in this case.

Warning: Novelty detection with Local Outlier Factor‘

When `novelty` is set to `True` be aware that you must only use `predict`, `decision_function` and `score_samples` on new unseen data and not on the training samples as this would lead to wrong results. The scores of abnormality of the training samples are always accessible through the `negative_outlier_factor_` attribute.

Novelty detection with Local Outlier Factor is illustrated below.



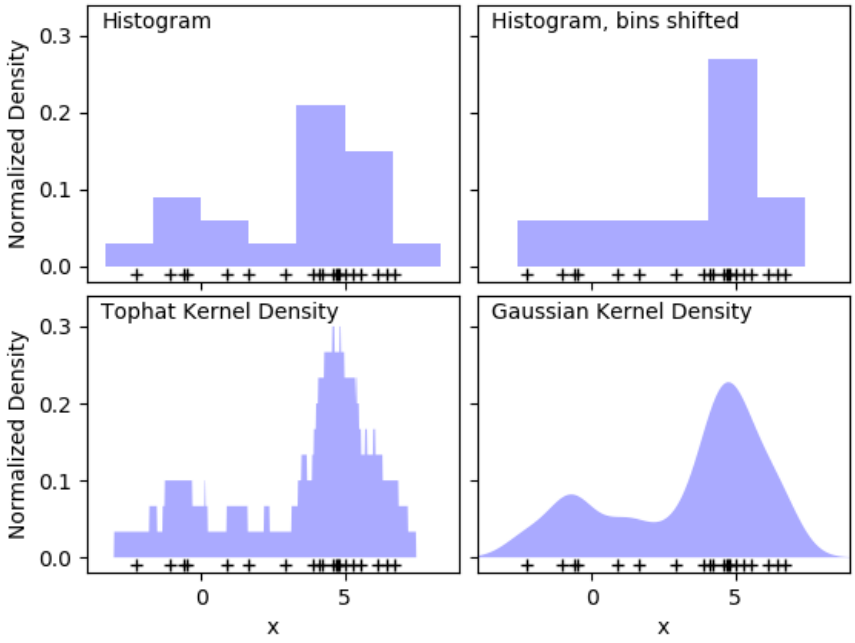
4.2.8 Density Estimation

Density estimation walks the line between unsupervised learning, feature engineering, and data modeling. Some of the most popular and useful density estimation techniques are mixture models such as Gaussian Mixtures (`sklearn.mixture.GaussianMixture`), and neighbor-based approaches such as the kernel density estimate (`sklearn.neighbors.KernelDensity`). Gaussian Mixtures are discussed more fully in the context of *clustering*, because the technique is also useful as an unsupervised clustering scheme.

Density estimation is a very simple concept, and most people are already familiar with one common density estimation technique: the histogram.

Density Estimation: Histograms

A histogram is a simple visualization of data where bins are defined, and the number of data points within each bin is tallied. An example of a histogram can be seen in the upper-left panel of the following figure:



A major problem with histograms, however, is that the choice of binning can have a disproportionate effect on the resulting visualization. Consider the upper-right panel of the above figure. It shows a histogram over the same data, with the bins shifted right. The results of the two visualizations look entirely different, and might lead to different interpretations of the data.

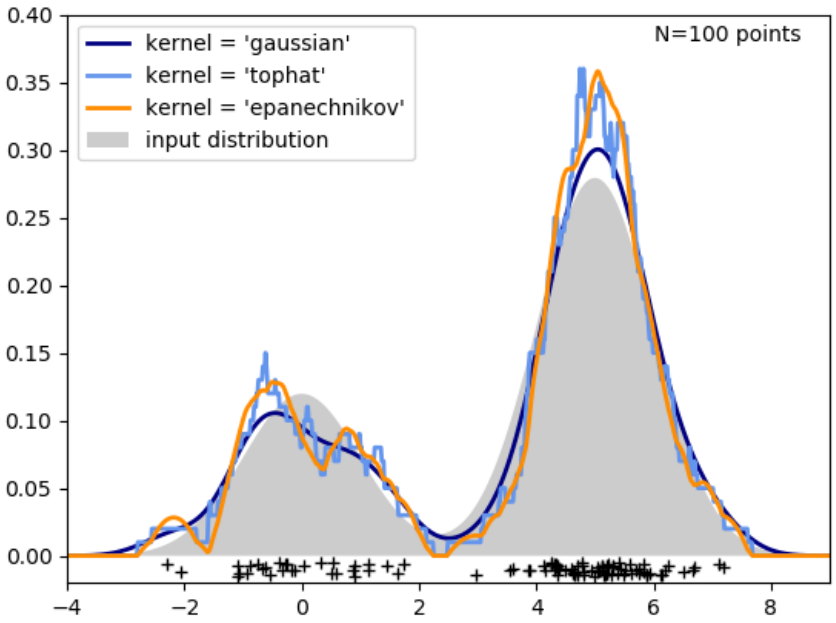
Intuitively, one can also think of a histogram as a stack of blocks, one block per point. By stacking the blocks in the appropriate grid space, we recover the histogram. But what if, instead of stacking the blocks on a regular grid, we center each block on the point it represents, and sum the total height at each location? This idea leads to the lower-left visualization. It is perhaps not as clean as a histogram, but the fact that the data drive the block locations mean that it is a much better representation of the underlying data.

This visualization is an example of a *kernel density estimation*, in this case with a top-hat kernel (i.e. a square block at each point). We can recover a smoother distribution by using a smoother kernel. The bottom-right plot shows a Gaussian kernel density estimate, in which each point contributes a Gaussian curve to the total. The result is a smooth density estimate which is derived from the data, and functions as a powerful non-parametric model of the distribution of points.

Kernel Density Estimation

Kernel density estimation in scikit-learn is implemented in the `sklearn.neighbors.KernelDensity` estimator, which uses the Ball Tree or KD Tree for efficient queries (see *Nearest Neighbors* for a discussion of these). Though the above example uses a 1D data set for simplicity, kernel density estimation can be performed in any number of dimensions, though in practice the curse of dimensionality causes its performance to degrade in high dimensions.

In the following figure, 100 points are drawn from a bimodal distribution, and the kernel density estimates are shown for three choices of kernels:



It's clear how the kernel shape affects the smoothness of the resulting distribution. The scikit-learn kernel density estimator can be used as follows:

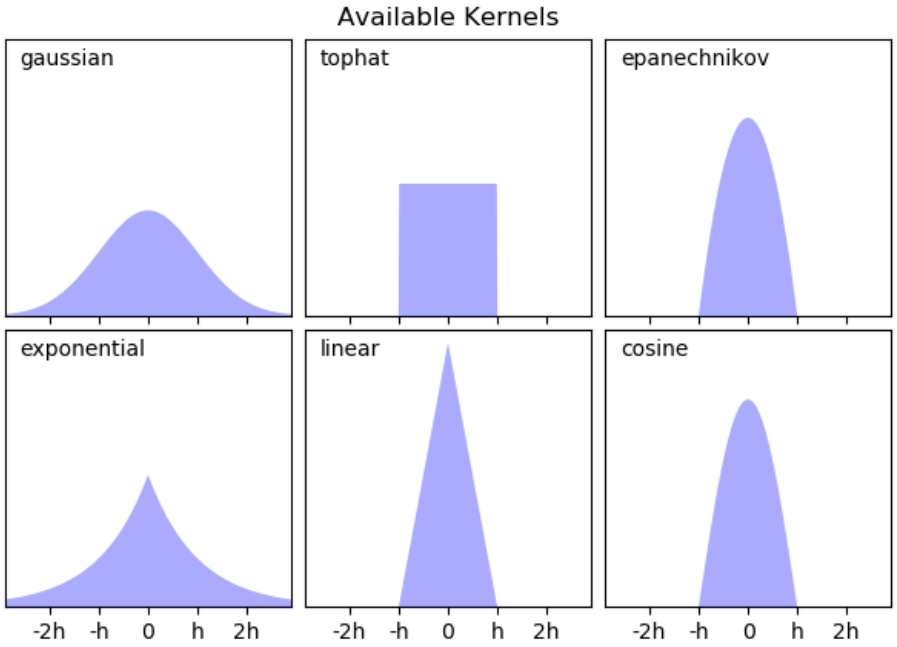
```
>>> from sklearn.neighbors import KernelDensity
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> kde = KernelDensity(kernel='gaussian', bandwidth=0.2).fit(X)
>>> kde.score_samples(X)
array([-0.41075698, -0.41075698, -0.41076071, -0.41075698, -0.41075698,
       -0.41076071])
```

Here we have used `kernel='gaussian'`, as seen above. Mathematically, a kernel is a positive function $K(x; h)$ which is controlled by the bandwidth parameter h. Given this kernel form, the density estimate at a point y within a group of points $x_i; i = 1 \cdots N$ is given by:

$$\rho_K(y) = \sum_{i=1}^{N} K((y - x_i)/h)$$

The bandwidth here acts as a smoothing parameter, controlling the tradeoff between bias and variance in the result. A large bandwidth leads to a very smooth (i.e. high-bias) density distribution. A small bandwidth leads to an unsmooth (i.e. high-variance) density distribution.

sklearn.neighbors.KernelDensity implements several common kernel forms, which are shown in the following figure:



The form of these kernels is as follows:

- Gaussian kernel (`kernel = 'gaussian'`)

  $K(x; h) \propto \exp(-\frac{x^2}{2h^2})$

- Tophat kernel (`kernel = 'tophat'`)

  $K(x; h) \propto 1$ if $x < h$

- Epanechnikov kernel (`kernel = 'epanechnikov'`)

  $K(x; h) \propto 1 - \frac{x^2}{h^2}$

- Exponential kernel (`kernel = 'exponential'`)

  $K(x; h) \propto \exp(-x/h)$

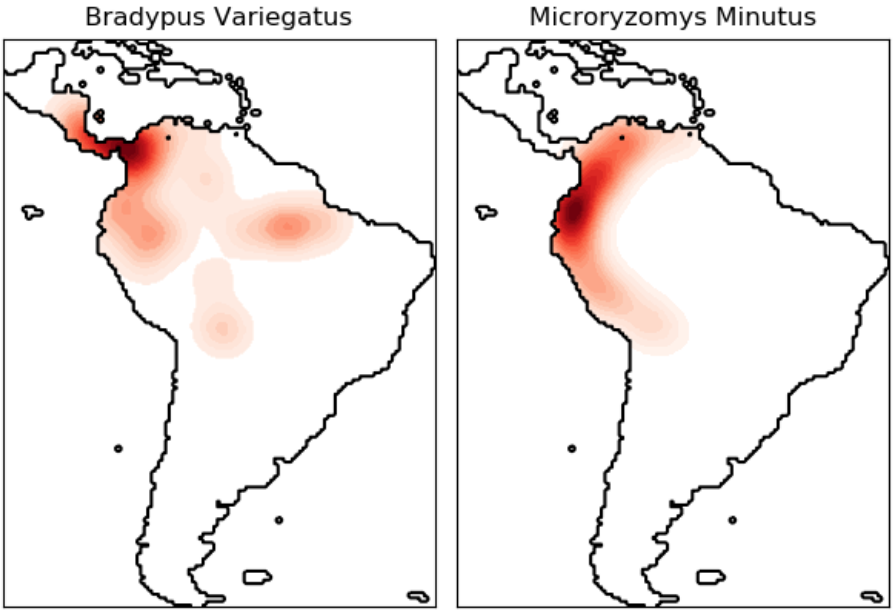
- Linear kernel (`kernel = 'linear'`)

  $K(x; h) \propto 1 - x/h$ if $x < h$

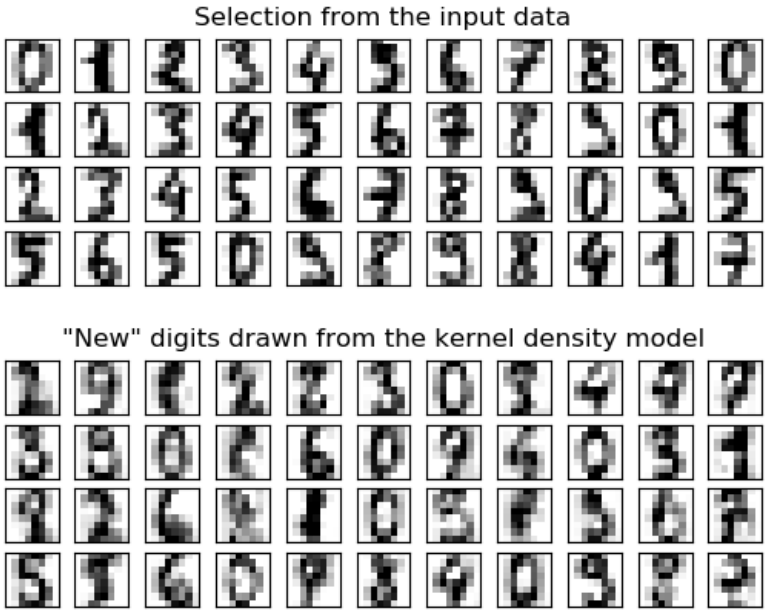
- Cosine kernel (`kernel = 'cosine'`)

  $K(x; h) \propto \cos(\frac{\pi x}{2h})$ if $x < h$

The kernel density estimator can be used with any of the valid distance metrics (see *sklearn.neighbors.DistanceMetric* for a list of available metrics), though the results are properly normalized only for the Euclidean metric. One particularly useful metric is the Haversine distance which measures the angular distance between points on a sphere. Here is an example of using a kernel density estimate for a visualization of geospatial data, in this case the distribution of observations of two different species on the South American continent:



One other useful application of kernel density estimation is to learn a non-parametric generative model of a dataset in order to efficiently draw new samples from this generative model. Here is an example of using this process to create a new set of hand-written digits, using a Gaussian kernel learned on a PCA projection of the data:



The "new" data consists of linear combinations of the input data, with weights probabilistically drawn given the KDE model.

Examples:

- *Simple 1D Kernel Density Estimation*: computation of simple kernel density estimates in one dimension.
- *Kernel Density Estimation*: an example of using Kernel Density estimation to learn a generative model of the hand-written digits data, and drawing new samples from this model.
- *Kernel Density Estimate of Species Distributions*: an example of Kernel Density estimation using the Haversine distance metric to visualize geospatial data

4.2.9 Neural network models (unsupervised)

Restricted Boltzmann machines

Restricted Boltzmann machines (RBM) are unsupervised nonlinear feature learners based on a probabilistic model. The features extracted by an RBM or a hierarchy of RBMs often give good results when fed into a linear classifier such as a linear SVM or a perceptron.

The model makes assumptions regarding the distribution of inputs. At the moment, scikit-learn only provides `BernoulliRBM`, which assumes the inputs are either binary values or values between 0 and 1, each encoding the probability that the specific feature would be turned on.

The RBM tries to maximize the likelihood of the data using a particular graphical model. The parameter learning algorithm used (*Stochastic Maximum Likelihood*) prevents the representations from straying far from the input data, which makes them capture interesting regularities, but makes the model less useful for small datasets, and usually not useful for density estimation.

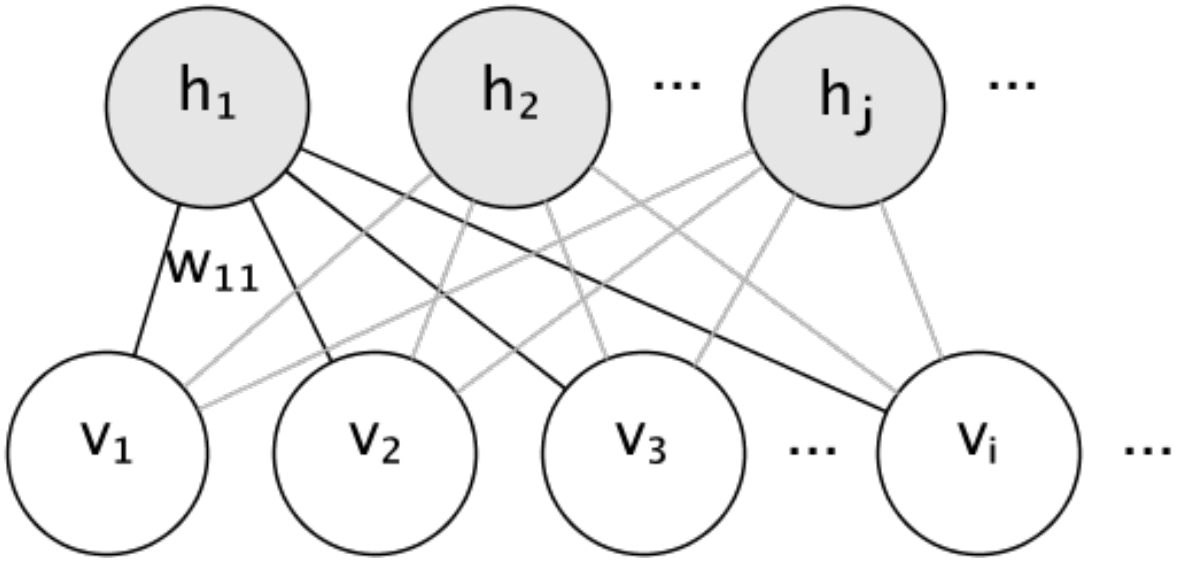
The method gained popularity for initializing deep neural networks with the weights of independent RBMs. This method is known as unsupervised pre-training.

Examples:

- *Restricted Boltzmann Machine features for digit classification*

Graphical model and parametrization

The graphical model of an RBM is a fully-connected bipartite graph.



The nodes are random variables whose states depend on the state of the other nodes they are connected to. The model is therefore parameterized by the weights of the connections, as well as one intercept (bias) term for each visible and hidden unit, omitted from the image for simplicity.

The energy function measures the quality of a joint assignment:

$$E(\mathbf{v}, \mathbf{h}) = -\sum_i \sum_j w_{ij} v_i h_j - \sum_i b_i v_i - \sum_j c_j h_j$$

In the formula above, $\mathbf{b}$ and $\mathbf{c}$ are the intercept vectors for the visible and hidden layers, respectively. The joint

probability of the model is defined in terms of the energy:

$$P(\mathbf{v}, \mathbf{h}) = \frac{e^{-E(\mathbf{v},\mathbf{h})}}{Z}$$

The word *restricted* refers to the bipartite structure of the model, which prohibits direct interaction between hidden units, or between visible units. This means that the following conditional independencies are assumed:

$$h_i \perp h_j | \mathbf{v}$$
$$v_i \perp v_j | \mathbf{h}$$

The bipartite structure allows for the use of efficient block Gibbs sampling for inference.

Bernoulli Restricted Boltzmann machines

In the `BernoulliRBM`, all units are binary stochastic units. This means that the input data should either be binary, or real-valued between 0 and 1 signifying the probability that the visible unit would turn on or off. This is a good model for character recognition, where the interest is on which pixels are active and which aren't. For images of natural scenes it no longer fits because of background, depth and the tendency of neighbouring pixels to take the same values.

The conditional probability distribution of each unit is given by the logistic sigmoid activation function of the input it receives:

$$P(v_i = 1|\mathbf{h}) = \sigma(\sum_j w_{ij} h_j + b_i)$$
$$P(h_i = 1|\mathbf{v}) = \sigma(\sum_i w_{ij} v_i + c_j)$$

where σ is the logistic sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Stochastic Maximum Likelihood learning

The training algorithm implemented in `BernoulliRBM` is known as Stochastic Maximum Likelihood (SML) or Persistent Contrastive Divergence (PCD). Optimizing maximum likelihood directly is infeasible because of the form of the data likelihood:

$$\log P(v) = \log \sum_h e^{-E(v,h)} - \log \sum_{x,y} e^{-E(x,y)}$$

For simplicity the equation above is written for a single training example. The gradient with respect to the weights is formed of two terms corresponding to the ones above. They are usually known as the positive gradient and the negative gradient, because of their respective signs. In this implementation, the gradients are estimated over mini-batches of samples.

In maximizing the log-likelihood, the positive gradient makes the model prefer hidden states that are compatible with the observed training data. Because of the bipartite structure of RBMs, it can be computed efficiently. The negative gradient, however, is intractable. Its goal is to lower the energy of joint states that the model prefers, therefore making it stay true to the data. It can be approximated by Markov chain Monte Carlo using block Gibbs sampling by iteratively sampling each of v and h given the other, until the chain mixes. Samples generated in this way are sometimes referred as fantasy particles. This is inefficient and it is difficult to determine whether the Markov chain mixes.

The Contrastive Divergence method suggests to stop the chain after a small number of iterations, k, usually even 1. This method is fast and has low variance, but the samples are far from the model distribution.

Persistent Contrastive Divergence addresses this. Instead of starting a new chain each time the gradient is needed, and performing only one Gibbs sampling step, in PCD we keep a number of chains (fantasy particles) that are updated k Gibbs steps after each weight update. This allows the particles to explore the space more thoroughly.

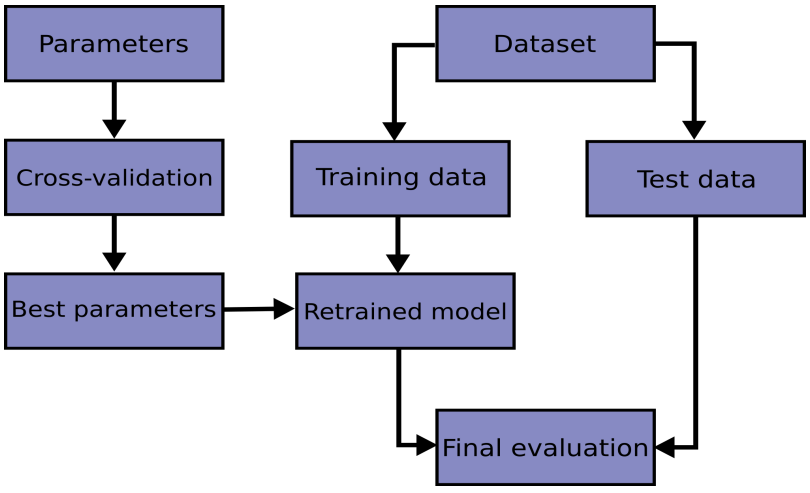
References:

- "A fast learning algorithm for deep belief nets" G. Hinton, S. Osindero, Y.-W. Teh, 2006
- "Training Restricted Boltzmann Machines using Approximations to the Likelihood Gradient" T. Tieleman, 2008

4.3 Model selection and evaluation

4.3.1 Cross-validation: evaluating estimator performance

Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called **overfitting**. To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a **test set** `X_test, y_test`. Note that the word "experiment" is not intended to denote academic use only, because even in commercial settings machine learning usually starts out experimentally. Here is a flowchart of typical cross validation workflow in model training. The best parameters can be determined by *grid search* techniques.



In scikit-learn a random split into training and test sets can be quickly computed with the *train_test_split* helper function. Let's load the iris data set to fit a linear support vector machine on it:

```
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
>>> from sklearn import datasets
>>> from sklearn import svm

>>> X, y = datasets.load_iris(return_X_y=True)
>>> X.shape, y.shape
((150, 4), (150,))
```

We can now quickly sample a training set while holding out 40% of the data for testing (evaluating) our classifier:

```
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.4, random_state=0)

>>> X_train.shape, y_train.shape
((90, 4), (90,))
>>> X_test.shape, y_test.shape
((60, 4), (60,))

>>> clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.96...
```

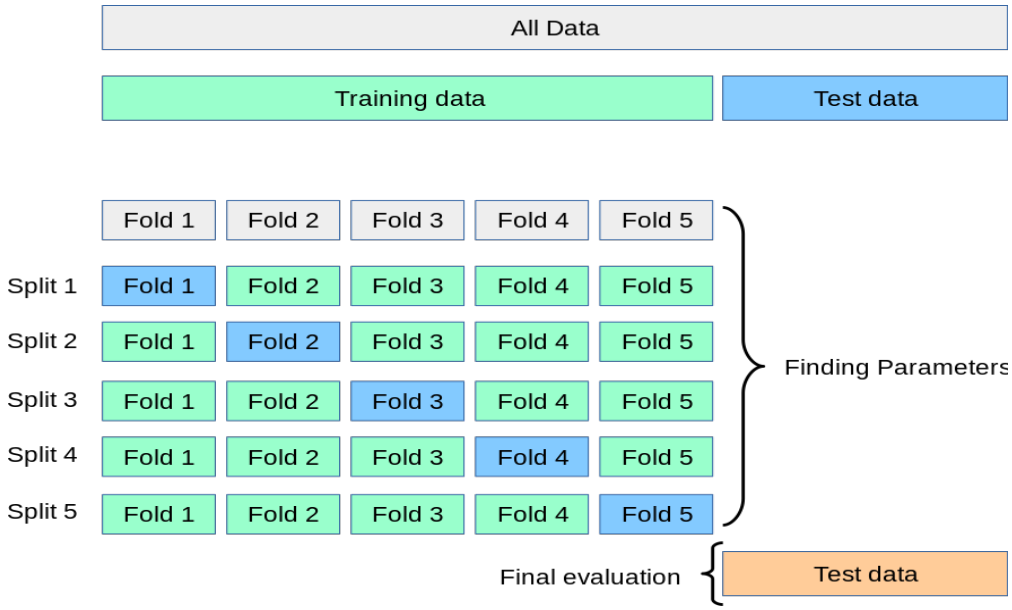
When evaluating different settings ("hyperparameters") for estimators, such as the `C` setting that must be manually set for an SVM, there is still a risk of overfitting *on the test set* because the parameters can be tweaked until the estimator performs optimally. This way, knowledge about the test set can "leak" into the model and evaluation metrics no longer report on generalization performance. To solve this problem, yet another part of the dataset can be held out as a so-called "validation set": training proceeds on the training set, after which evaluation is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.

However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, validation) sets.

A solution to this problem is a procedure called cross-validation (CV for short). A test set should still be held out for final evaluation, but the validation set is no longer needed when doing CV. In the basic approach, called *k*-fold CV, the training set is split into *k* smaller sets (other approaches are described below, but generally follow the same principles). The following procedure is followed for each of the *k* "folds":

- A model is trained using $k - 1$ of the folds as training data;
- the resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure such as accuracy).

The performance measure reported by *k*-fold cross-validation is then the average of the values computed in the loop. This approach can be computationally expensive, but does not waste too much data (as is the case when fixing an arbitrary validation set), which is a major advantage in problems such as inverse inference where the number of samples is very small.



Computing cross-validated metrics

The simplest way to use cross-validation is to call the *cross_val_score* helper function on the estimator and the dataset.

The following example demonstrates how to estimate the accuracy of a linear kernel support vector machine on the iris dataset by splitting the data, fitting a model and computing the score 5 consecutive times (with different splits each time):

```
>>> from sklearn.model_selection import cross_val_score
>>> clf = svm.SVC(kernel='linear', C=1)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.        ])
```

The mean score and the 95% confidence interval of the score estimate are hence given by:

```
>>> print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
Accuracy: 0.98 (+/- 0.03)
```

By default, the score computed at each CV iteration is the `score` method of the estimator. It is possible to change this by using the scoring parameter:

```
>>> from sklearn import metrics
>>> scores = cross_val_score(
...     clf, X, y, cv=5, scoring='f1_macro')
>>> scores
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.        ])
```

See *The scoring parameter: defining model evaluation rules* for details. In the case of the Iris dataset, the samples are balanced across target classes hence the accuracy and the F1-score are almost equal.

When the `cv` argument is an integer, *cross_val_score* uses the *KFold* or *StratifiedKFold* strategies by default, the latter being used if the estimator derives from *ClassifierMixin*.

It is also possible to use other cross validation strategies by passing a cross validation iterator instead, for instance:

```
>>> from sklearn.model_selection import ShuffleSplit
>>> n_samples = X.shape[0]
>>> cv = ShuffleSplit(n_splits=5, test_size=0.3, random_state=0)
>>> cross_val_score(clf, X, y, cv=cv)
array([0.977..., 0.977..., 1.  ..., 0.955..., 1.        ])
```

Another option is to use an iterable yielding (train, test) splits as arrays of indices, for example:

```
>>> def custom_cv_2folds(X):
...     n = X.shape[0]
...     i = 1
...     while i <= 2:
...         idx = np.arange(n * (i - 1) / 2, n * i / 2, dtype=int)
...         yield idx, idx
...         i += 1
...
>>> custom_cv = custom_cv_2folds(X)
>>> cross_val_score(clf, X, y, cv=custom_cv)
array([1.        , 0.973...])
```

Data transformation with held out data

Just as it is important to test a predictor on data held-out from training, preprocessing (such as standardization, feature selection, etc.) and similar *data transformations* similarly should be learnt from a training set and applied to held-out data for prediction:

```
>>> from sklearn import preprocessing
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.4, random_state=0)
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train_transformed = scaler.transform(X_train)
>>> clf = svm.SVC(C=1).fit(X_train_transformed, y_train)
>>> X_test_transformed = scaler.transform(X_test)
>>> clf.score(X_test_transformed, y_test)
0.9333...
```

A *Pipeline* makes it easier to compose estimators, providing this behavior under cross-validation:

```
>>> from sklearn.pipeline import make_pipeline
>>> clf = make_pipeline(preprocessing.StandardScaler(), svm.SVC(C=1))
>>> cross_val_score(clf, X, y, cv=cv)
array([0.977..., 0.933..., 0.955..., 0.933..., 0.977...])
```

See *Pipelines and composite estimators*.

The cross_validate function and multiple metric evaluation

The *cross_validate* function differs from *cross_val_score* in two ways:

- It allows specifying multiple metrics for evaluation.
- It returns a dict containing fit-times, score-times (and optionally training scores as well as fitted estimators) in addition to the test score.

For single metric evaluation, where the scoring parameter is a string, callable or None, the keys will be - `['test_score', 'fit_time', 'score_time']`

And for multiple metric evaluation, the return value is a dict with the following keys - `['test_<scorer1_name>', 'test_<scorer2_name>', 'test_<scorer...>', 'fit_time', 'score_time']`

`return_train_score` is set to `False` by default to save computation time. To evaluate the scores on the training set as well you need to be set to `True`.

You may also retain the estimator fitted on each training set by setting `return_estimator=True`.

The multiple metrics can be specified either as a list, tuple or set of predefined scorer names:

```
>>> from sklearn.model_selection import cross_validate
>>> from sklearn.metrics import recall_score
>>> scoring = ['precision_macro', 'recall_macro']
>>> clf = svm.SVC(kernel='linear', C=1, random_state=0)
>>> scores = cross_validate(clf, X, y, scoring=scoring)
>>> sorted(scores.keys())
['fit_time', 'score_time', 'test_precision_macro', 'test_recall_macro']
>>> scores['test_recall_macro']
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.        ])
```

Or as a dict mapping scorer name to a predefined or custom scoring function:

```
>>> from sklearn.metrics import make_scorer
>>> scoring = {'prec_macro': 'precision_macro',
...            'rec_macro': make_scorer(recall_score, average='macro')}
>>> scores = cross_validate(clf, X, y, scoring=scoring,
...                         cv=5, return_train_score=True)
>>> sorted(scores.keys())
['fit_time', 'score_time', 'test_prec_macro', 'test_rec_macro',
 'train_prec_macro', 'train_rec_macro']
>>> scores['train_rec_macro']
array([0.97..., 0.97..., 0.99..., 0.98..., 0.98...])
```

Here is an example of `cross_validate` using a single metric:

```
>>> scores = cross_validate(clf, X, y,
...                         scoring='precision_macro', cv=5,
...                         return_estimator=True)
>>> sorted(scores.keys())
['estimator', 'fit_time', 'score_time', 'test_score']
```

Obtaining predictions by cross-validation

The function *cross_val_predict* has a similar interface to *cross_val_score*, but returns, for each element in the input, the prediction that was obtained for that element when it was in the test set. Only cross-validation strategies that assign all elements to a test set exactly once can be used (otherwise, an exception is raised).

Warning: Note on inappropriate usage of cross_val_predict

The result of *cross_val_predict* may be different from those obtained using *cross_val_score* as the elements are grouped in different ways. The function *cross_val_score* takes an average over cross-validation folds, whereas *cross_val_predict* simply returns the labels (or probabilities) from several distinct models undistinguished. Thus, *cross_val_predict* is not an appropriate measure of generalisation error.

The function *cross_val_predict* is appropriate for:

- Visualization of predictions obtained from different models.
- Model blending: When predictions of one supervised estimator are used to train another estimator in ensemble methods.

The available cross validation iterators are introduced in the following section.

Examples

- *Receiver Operating Characteristic (ROC) with cross validation*,
- *Recursive feature elimination with cross-validation*,
- *Parameter estimation using grid search with cross-validation*,
- *Sample pipeline for text feature extraction and evaluation*,
- *Plotting Cross-Validated Predictions*,
- *Nested versus non-nested cross-validation*.

Cross validation iterators

The following sections list utilities to generate indices that can be used to generate dataset splits according to different cross validation strategies.

Cross-validation iterators for i.i.d. data

Assuming that some data is Independent and Identically Distributed (i.i.d.) is making the assumption that all samples stem from the same generative process and that the generative process is assumed to have no memory of past generated samples.

The following cross-validators can be used in such cases.

NOTE

While i.i.d. data is a common assumption in machine learning theory, it rarely holds in practice. If one knows that the samples have been generated using a time-dependent process, it's safer to use a *time-series aware cross-validation scheme* Similarly if we know that the generative process has a group structure (samples from collected from different subjects, experiments, measurement devices) it safer to use *group-wise cross-validation*.

K-fold

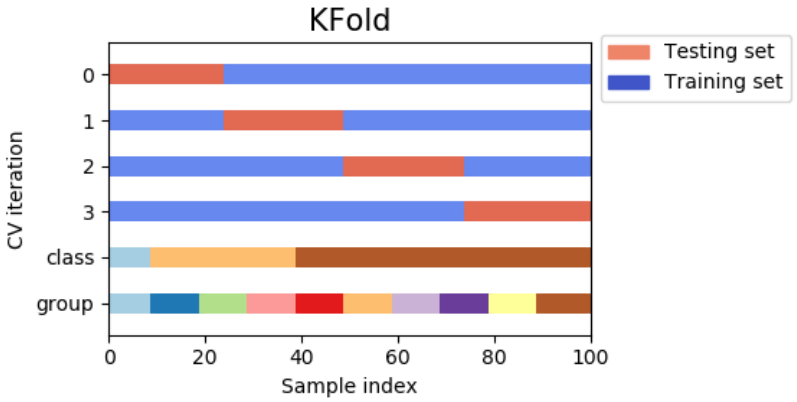
`KFold` divides all the samples in k groups of samples, called folds (if $k = n$, this is equivalent to the *Leave One Out* strategy), of equal sizes (if possible). The prediction function is learned using $k - 1$ folds, and the fold left out is used for test.

Example of 2-fold cross-validation on a dataset with 4 samples:

```
>>> import numpy as np
>>> from sklearn.model_selection import KFold

>>> X = ["a", "b", "c", "d"]
>>> kf = KFold(n_splits=2)
>>> for train, test in kf.split(X):
...     print("%s %s" % (train, test))
[2 3] [0 1]
[0 1] [2 3]
```

Here is a visualization of the cross-validation behavior. Note that `KFold` is not affected by classes or groups.



Each fold is constituted by two arrays: the first one is related to the *training set*, and the second one to the *test set*. Thus, one can create the training/test sets using numpy indexing:

```
>>> X = np.array([[0., 0.], [1., 1.], [-1., -1.], [2., 2.]])
>>> y = np.array([0, 1, 0, 1])
>>> X_train, X_test, y_train, y_test = X[train], X[test], y[train], y[test]
```

Repeated K-Fold

RepeatedKFold repeats K-Fold n times. It can be used when one requires to run *KFold* n times, producing different splits in each repetition.

Example of 2-fold K-Fold repeated 2 times:

```
>>> import numpy as np
>>> from sklearn.model_selection import RepeatedKFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> random_state = 12883823
>>> rkf = RepeatedKFold(n_splits=2, n_repeats=2, random_state=random_state)
>>> for train, test in rkf.split(X):
...     print("%s %s" % (train, test))
...
[2 3] [0 1]
[0 1] [2 3]
[0 2] [1 3]
[1 3] [0 2]
```

Similarly, *RepeatedStratifiedKFold* repeats Stratified K-Fold n times with different randomization in each repetition.

Leave One Out (LOO)

LeaveOneOut (or LOO) is a simple cross-validation. Each learning set is created by taking all the samples except one, the test set being the sample left out. Thus, for n samples, we have n different training sets and n different tests set. This cross-validation procedure does not waste much data as only one sample is removed from the training set:

```
>>> from sklearn.model_selection import LeaveOneOut

>>> X = [1, 2, 3, 4]
>>> loo = LeaveOneOut()
>>> for train, test in loo.split(X):
...     print("%s %s" % (train, test))
[1 2 3] [0]
[0 2 3] [1]
[0 1 3] [2]
[0 1 2] [3]
```

Potential users of LOO for model selection should weigh a few known caveats. When compared with k-fold cross validation, one builds n models from n samples instead of k models, where $n > k$. Moreover, each is trained on $n - 1$ samples rather than $(k - 1)n/k$. In both ways, assuming k is not too large and $k < n$, LOO is more computationally expensive than k-fold cross validation.

In terms of accuracy, LOO often results in high variance as an estimator for the test error. Intuitively, since $n - 1$ of the n samples are used to build each model, models constructed from folds are virtually identical to each other and to the model built from the entire training set.

However, if the learning curve is steep for the training size in question, then 5- or 10- fold cross validation can overestimate the generalization error.

As a general rule, most authors, and empirical evidence, suggest that 5- or 10- fold cross validation should be preferred to LOO.

References:

- http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-12.html;
- T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning, Springer 2009
- L. Breiman, P. Spector Submodel selection and evaluation in regression: The X-random case, International Statistical Review 1992;
- R. Kohavi, A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection, Intl. Jnt. Conf. AI
- R. Bharat Rao, G. Fung, R. Rosales, On the Dangers of Cross-Validation. An Experimental Evaluation, SIAM 2008;
- G. James, D. Witten, T. Hastie, R Tibshirani, An Introduction to Statistical Learning, Springer 2013.

Leave P Out (LPO)

LeavePOut is very similar to *LeaveOneOut* as it creates all the possible training/test sets by removing p samples from the complete set. For n samples, this produces $\binom{n}{p}$ train-test pairs. Unlike *LeaveOneOut* and *KFold*, the test sets will overlap for $p > 1$.

Example of Leave-2-Out on a dataset with 4 samples:

```
>>> from sklearn.model_selection import LeavePOut

>>> X = np.ones(4)
>>> lpo = LeavePOut(p=2)
>>> for train, test in lpo.split(X):
...     print("%s %s" % (train, test))
[2 3] [0 1]
[1 3] [0 2]
[1 2] [0 3]
[0 3] [1 2]
[0 2] [1 3]
[0 1] [2 3]
```

Random permutations cross-validation a.k.a. Shuffle & Split

ShuffleSplit

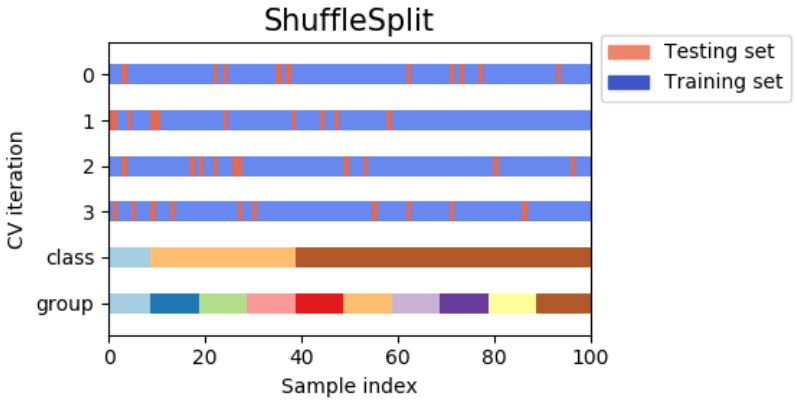
The *ShuffleSplit* iterator will generate a user defined number of independent train / test dataset splits. Samples are first shuffled and then split into a pair of train and test sets.

It is possible to control the randomness for reproducibility of the results by explicitly seeding the `random_state` pseudo random number generator.

Here is a usage example:

```
>>> from sklearn.model_selection import ShuffleSplit
>>> X = np.arange(10)
>>> ss = ShuffleSplit(n_splits=5, test_size=0.25, random_state=0)
>>> for train_index, test_index in ss.split(X):
...     print("%s %s" % (train_index, test_index))
[9 1 6 7 3 0 5] [2 8 4]
[2 9 8 0 6 7 4] [3 5 1]
[4 5 1 0 6 9 7] [2 3 8]
[2 7 5 8 0 3 4] [6 1 9]
[4 1 0 6 8 9 3] [5 2 7]
```

Here is a visualization of the cross-validation behavior. Note that *ShuffleSplit* is not affected by classes or groups.



ShuffleSplit is thus a good alternative to *KFold* cross validation that allows a finer control on the number of iterations and the proportion of samples on each side of the train / test split.

Cross-validation iterators with stratification based on class labels.

Some classification problems can exhibit a large imbalance in the distribution of the target classes: for instance there could be several times more negative samples than positive samples. In such cases it is recommended to use stratified sampling as implemented in *StratifiedKFold* and *StratifiedShuffleSplit* to ensure that relative class frequencies is approximately preserved in each train and validation fold.

Stratified k-fold

StratifiedKFold is a variation of *k-fold* which returns *stratified* folds: each set contains approximately the same percentage of samples of each target class as the complete set.

Here is an example of stratified 3-fold cross-validation on a dataset with 50 samples from two unbalanced classes. We show the number of samples in each class and compare with *KFold*.

```
>>> from sklearn.model_selection import StratifiedKFold, KFold
>>> import numpy as np
>>> X, y = np.ones((50, 1)), np.hstack(([0] * 45, [1] * 5))
>>> skf = StratifiedKFold(n_splits=3)
>>> for train, test in skf.split(X, y):
...     print('train -  {}   |   test -  {}'.format(
```

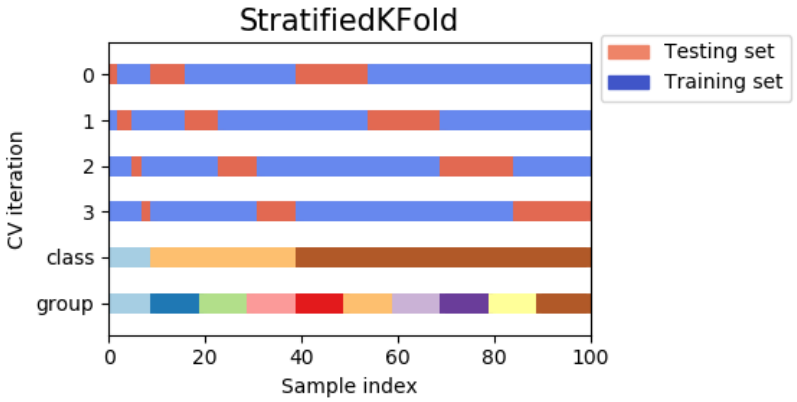
(continues on next page)

(continued from previous page)

```
...          np.bincount(y[train]), np.bincount(y[test])))
train -  [30  3]   |   test -  [15  2]
train -  [30  3]   |   test -  [15  2]
train -  [30  4]   |   test -  [15  1]
>>> kf = KFold(n_splits=3)
>>> for train, test in kf.split(X, y):
...     print('train -  {}   |   test -  {}'.format(
...          np.bincount(y[train]), np.bincount(y[test])))
train -  [28  5]   |   test -  [17]
train -  [28  5]   |   test -  [17]
train -  [34]   |   test -  [11  5]
```

We can see that *StratifiedKFold* preserves the class ratios (approximately 1 / 10) in both train and test dataset.

Here is a visualization of the cross-validation behavior.



RepeatedStratifiedKFold can be used to repeat Stratified K-Fold n times with different randomization in each repetition.

Stratified Shuffle Split

StratifiedShuffleSplit is a variation of *ShuffleSplit*, which returns stratified splits, *i.e* which creates splits by preserving the same percentage for each target class as in the complete set.

Here is a visualization of the cross-validation behavior.

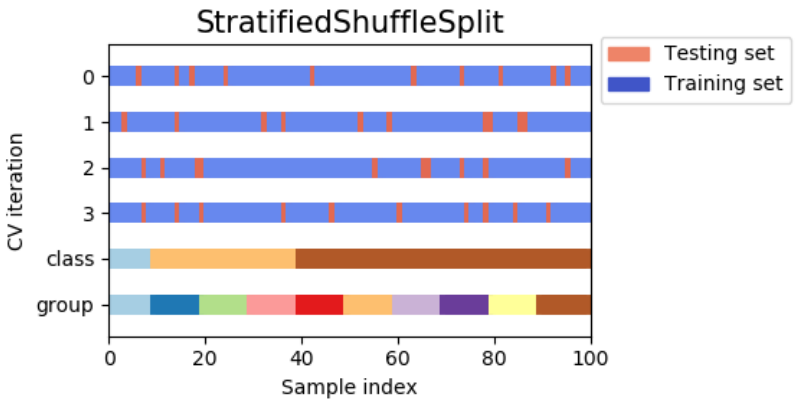
Cross-validation iterators for grouped data.

The i.i.d. assumption is broken if the underlying generative process yield groups of dependent samples.

Such a grouping of data is domain specific. An example would be when there is medical data collected from multiple patients, with multiple samples taken from each patient. And such data is likely to be dependent on the individual group. In our example, the patient id for each sample will be its group identifier.

In this case we would like to know if a model trained on a particular set of groups generalizes well to the unseen groups. To measure this, we need to ensure that all the samples in the validation fold come from groups that are not represented at all in the paired training fold.

The following cross-validation splitters can be used to do that. The grouping identifier for the samples is specified via the `groups` parameter.



Group k-fold

GroupKFold is a variation of k-fold which ensures that the same group is not represented in both testing and training sets. For example if the data is obtained from different subjects with several samples per-subject and if the model is flexible enough to learn from highly person specific features it could fail to generalize to new subjects. *GroupKFold* makes it possible to detect this kind of overfitting situations.

Imagine you have three subjects, each with an associated number from 1 to 3:

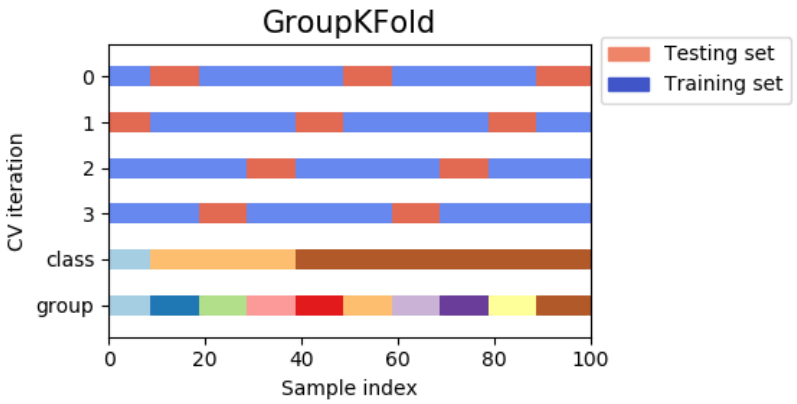
```
>>> from sklearn.model_selection import GroupKFold

>>> X = [0.1, 0.2, 2.2, 2.4, 2.3, 4.55, 5.8, 8.8, 9, 10]
>>> y = ["a", "b", "b", "b", "c", "c", "c", "d", "d", "d"]
>>> groups = [1, 1, 1, 2, 2, 2, 3, 3, 3, 3]

>>> gkf = GroupKFold(n_splits=3)
>>> for train, test in gkf.split(X, y, groups=groups):
...     print("%s %s" % (train, test))
[0 1 2 3 4 5] [6 7 8 9]
[0 1 2 6 7 8 9] [3 4 5]
[3 4 5 6 7 8 9] [0 1 2]
```

Each subject is in a different testing fold, and the same subject is never in both testing and training. Notice that the folds do not have exactly the same size due to the imbalance in the data.

Here is a visualization of the cross-validation behavior.



Leave One Group Out

LeaveOneGroupOut is a cross-validation scheme which holds out the samples according to a third-party provided array of integer groups. This group information can be used to encode arbitrary domain specific pre-defined cross-validation folds.

Each training set is thus constituted by all the samples except the ones related to a specific group.

For example, in the cases of multiple experiments, *LeaveOneGroupOut* can be used to create a cross-validation based on the different experiments: we create a training set using the samples of all the experiments except one:

```
>>> from sklearn.model_selection import LeaveOneGroupOut

>>> X = [1, 5, 10, 50, 60, 70, 80]
>>> y = [0, 1, 1, 2, 2, 2, 2]
>>> groups = [1, 1, 2, 2, 3, 3, 3]
>>> logo = LeaveOneGroupOut()
>>> for train, test in logo.split(X, y, groups=groups):
...     print("%s %s" % (train, test))
[2 3 4 5 6] [0 1]
[0 1 4 5 6] [2 3]
[0 1 2 3] [4 5 6]
```

Another common application is to use time information: for instance the groups could be the year of collection of the samples and thus allow for cross-validation against time-based splits.

Leave P Groups Out

LeavePGroupsOut is similar as *LeaveOneGroupOut*, but removes samples related to P groups for each training/test set.

Example of Leave-2-Group Out:

```
>>> from sklearn.model_selection import LeavePGroupsOut

>>> X = np.arange(6)
>>> y = [1, 1, 1, 2, 2, 2]
>>> groups = [1, 1, 2, 2, 3, 3]
>>> lpgo = LeavePGroupsOut(n_groups=2)
>>> for train, test in lpgo.split(X, y, groups=groups):
...     print("%s %s" % (train, test))
[4 5] [0 1 2 3]
[2 3] [0 1 4 5]
[0 1] [2 3 4 5]
```

Group Shuffle Split

The *GroupShuffleSplit* iterator behaves as a combination of *ShuffleSplit* and *LeavePGroupsOut*, and generates a sequence of randomized partitions in which a subset of groups are held out for each split.

Here is a usage example:

```
>>> from sklearn.model_selection import GroupShuffleSplit

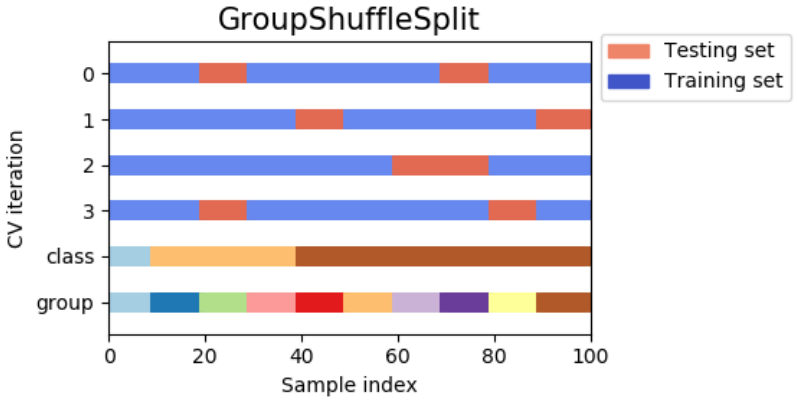
>>> X = [0.1, 0.2, 2.2, 2.4, 2.3, 4.55, 5.8, 0.001]
```

(continues on next page)

(continued from previous page)

```
>>> y = ["a", "b", "b", "b", "c", "c", "c", "a"]
>>> groups = [1, 1, 2, 2, 3, 3, 4, 4]
>>> gss = GroupShuffleSplit(n_splits=4, test_size=0.5, random_state=0)
>>> for train, test in gss.split(X, y, groups=groups):
...     print("%s %s" % (train, test))
...
[0 1 2 3] [4 5 6 7]
[2 3 6 7] [0 1 4 5]
[2 3 4 5] [0 1 6 7]
[4 5 6 7] [0 1 2 3]
```

Here is a visualization of the cross-validation behavior.



This class is useful when the behavior of *LeavePGroupsOut* is desired, but the number of groups is large enough that generating all possible partitions with P groups withheld would be prohibitively expensive. In such a scenario, *GroupShuffleSplit* provides a random sample (with replacement) of the train / test splits generated by *LeavePGroupsOut*.

Predefined Fold-Splits / Validation-Sets

For some datasets, a pre-defined split of the data into training- and validation fold or into several cross-validation folds already exists. Using *PredefinedSplit* it is possible to use these folds e.g. when searching for hyperparameters.

For example, when using a validation set, set the `test_fold` to 0 for all samples that are part of the validation set, and to -1 for all other samples.

Cross validation of time series data

Time series data is characterised by the correlation between observations that are near in time (*autocorrelation*). However, classical cross-validation techniques such as *KFold* and *ShuffleSplit* assume the samples are independent and identically distributed, and would result in unreasonable correlation between training and testing instances (yielding poor estimates of generalisation error) on time series data. Therefore, it is very important to evaluate our model for time series data on the "future" observations least like those that are used to train the model. To achieve this, one solution is provided by *TimeSeriesSplit*.

Time Series Split

TimeSeriesSplit is a variation of *k-fold* which returns first k folds as train set and the $(k+1)$ th fold as test set. Note that unlike standard cross-validation methods, successive training sets are supersets of those that come before them. Also, it adds all surplus data to the first training partition, which is always used to train the model.

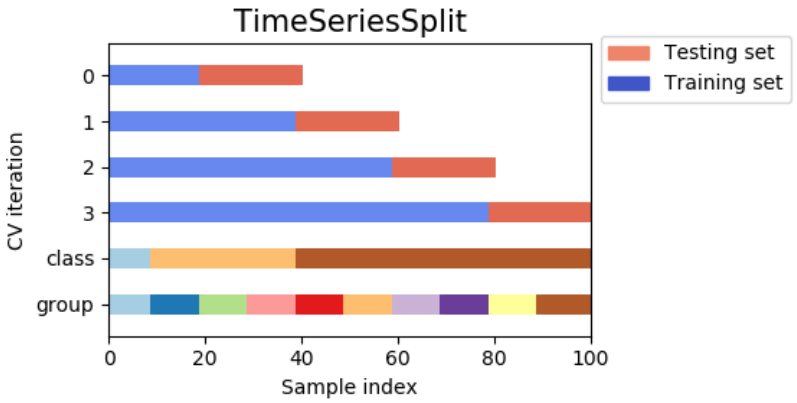
This class can be used to cross-validate time series data samples that are observed at fixed time intervals.

Example of 3-split time series cross-validation on a dataset with 6 samples:

```
>>> from sklearn.model_selection import TimeSeriesSplit

>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> tscv = TimeSeriesSplit(n_splits=3)
>>> print(tscv)
TimeSeriesSplit(max_train_size=None, n_splits=3)
>>> for train, test in tscv.split(X):
...     print("%s %s" % (train, test))
[0 1 2] [3]
[0 1 2 3] [4]
[0 1 2 3 4] [5]
```

Here is a visualization of the cross-validation behavior.



A note on shuffling

If the data ordering is not arbitrary (e.g. samples with the same class label are contiguous), shuffling it first may be essential to get a meaningful cross- validation result. However, the opposite may be true if the samples are not independently and identically distributed. For example, if samples correspond to news articles, and are ordered by their time of publication, then shuffling the data will likely lead to a model that is overfit and an inflated validation score: it will be tested on samples that are artificially similar (close in time) to training samples.

Some cross validation iterators, such as *KFold*, have an inbuilt option to shuffle the data indices before splitting them. Note that:

- This consumes less memory than shuffling the data directly.
- By default no shuffling occurs, including for the (stratified) K fold cross- validation performed by specifying `cv=some_integer` to *cross_val_score*, grid search, etc. Keep in mind that *train_test_split* still returns a random split.

- The `random_state` parameter defaults to `None`, meaning that the shuffling will be different every time `KFold(..., shuffle=True)` is iterated. However, `GridSearchCV` will use the same shuffling for each set of parameters validated by a single call to its `fit` method.
- To get identical results for each split, set `random_state` to an integer.

Cross validation and model selection

Cross validation iterators can also be used to directly perform model selection using Grid Search for the optimal hyperparameters of the model. This is the topic of the next section: *Tuning the hyper-parameters of an estimator*.

4.3.2 Tuning the hyper-parameters of an estimator

Hyper-parameters are parameters that are not directly learnt within estimators. In scikit-learn they are passed as arguments to the constructor of the estimator classes. Typical examples include `C`, `kernel` and `gamma` for Support Vector Classifier, `alpha` for Lasso, etc.

It is possible and recommended to search the hyper-parameter space for the best *cross validation* score.

Any parameter provided when constructing an estimator may be optimized in this manner. Specifically, to find the names and current values for all parameters for a given estimator, use:

```
estimator.get_params()
```

A search consists of:

- an estimator (regressor or classifier such as `sklearn.svm.SVC()`);
- a parameter space;
- a method for searching or sampling candidates;
- a cross-validation scheme; and
- a *score function*.

Some models allow for specialized, efficient parameter search strategies, *outlined below*. Two generic approaches to sampling search candidates are provided in scikit-learn: for given values, *`GridSearchCV`* exhaustively considers all parameter combinations, while *`RandomizedSearchCV`* can sample a given number of candidates from a parameter space with a specified distribution. After describing these tools we detail *best practice* applicable to both approaches.

Note that it is common that a small subset of those parameters can have a large impact on the predictive or computation performance of the model while others can be left to their default values. It is recommended to read the docstring of the estimator class to get a finer understanding of their expected behavior, possibly by reading the enclosed reference to the literature.

Exhaustive Grid Search

The grid search provided by *`GridSearchCV`* exhaustively generates candidates from a grid of parameter values specified with the `param_grid` parameter. For instance, the following `param_grid`:

```
param_grid = [
  {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
  {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel': ['rbf']},
 ]
```

specifies that two grids should be explored: one with a linear kernel and C values in [1, 10, 100, 1000], and the second one with an RBF kernel, and the cross-product of C values ranging in [1, 10, 100, 1000] and gamma values in [0.001, 0.0001].

The *GridSearchCV* instance implements the usual estimator API: when "fitting" it on a dataset all the possible combinations of parameter values are evaluated and the best combination is retained.

Examples:

- See *Parameter estimation using grid search with cross-validation* for an example of Grid Search computation on the digits dataset.
- See *Sample pipeline for text feature extraction and evaluation* for an example of Grid Search coupling parameters from a text documents feature extractor (n-gram count vectorizer and TF-IDF transformer) with a classifier (here a linear SVM trained with SGD with either elastic net or L2 penalty) using a `pipeline.Pipeline` instance.
- See *Nested versus non-nested cross-validation* for an example of Grid Search within a cross validation loop on the iris dataset. This is the best practice for evaluating the performance of a model with grid search.
- See *Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV* for an example of *GridSearchCV* being used to evaluate multiple metrics simultaneously.
- See *Balance model complexity and cross-validated score* for an example of using `refit=callable` interface in *GridSearchCV*. The example shows how this interface adds certain amount of flexibility in identifying the "best" estimator. This interface can also be used in multiple metrics evaluation.

Randomized Parameter Optimization

While using a grid of parameter settings is currently the most widely used method for parameter optimization, other search methods have more favourable properties. *RandomizedSearchCV* implements a randomized search over parameters, where each setting is sampled from a distribution over possible parameter values. This has two main benefits over an exhaustive search:

- A budget can be chosen independent of the number of parameters and possible values.
- Adding parameters that do not influence the performance does not decrease efficiency.

Specifying how parameters should be sampled is done using a dictionary, very similar to specifying parameters for *GridSearchCV*. Additionally, a computation budget, being the number of sampled candidates or sampling iterations, is specified using the `n_iter` parameter. For each parameter, either a distribution over possible values or a list of discrete choices (which will be sampled uniformly) can be specified:

```
{'C': scipy.stats.expon(scale=100), 'gamma': scipy.stats.expon(scale=.1),
  'kernel': ['rbf'], 'class_weight':['balanced', None]}
```

This example uses the `scipy.stats` module, which contains many useful distributions for sampling parameters, such as `expon`, `gamma`, `uniform` or `randint`.

In principle, any function can be passed that provides a `rvs` (random variate sample) method to sample a value. A call to the `rvs` function should provide independent random samples from possible parameter values on consecutive calls.

Warning: The distributions in `scipy.stats` prior to version scipy 0.16 do not allow specifying a random state. Instead, they use the global numpy random state, that can be seeded via `np.random.seed` or set using `np.random.set_state`. However, beginning scikit-learn 0.18, the *`sklearn.model_selection`* module sets the random state provided by the user if scipy >= 0.16 is also available.

For continuous parameters, such as `C` above, it is important to specify a continuous distribution to take full advantage of the randomization. This way, increasing `n_iter` will always lead to a finer search.

A continuous log-uniform random variable is available through `loguniform`. This is a continuous version of log-spaced parameters. For example to specify `C` above, `loguniform(1, 100)` can be used instead of `[1, 10, 100]` or `np.logspace(0, 2, num=1000)`. This is an alias to SciPy's stats.reciprocal.

Mirroring the example above in grid search, we can specify a continuous random variable that is log-uniformly distributed between `1e0` and `1e3`:

```
from sklearn.utils.fixes import loguniform
{'C': loguniform(1e0, 1e3),
 'gamma': loguniform(1e-4, 1e-3),
 'kernel': ['rbf'],
 'class_weight':['balanced', None]}
```

Examples:

- *Comparing randomized search and grid search for hyperparameter estimation* compares the usage and efficiency of randomized search and grid search.

References:

- Bergstra, J. and Bengio, Y., Random search for hyper-parameter optimization, The Journal of Machine Learning Research (2012)

Tips for parameter search

Specifying an objective metric

By default, parameter search uses the `score` function of the estimator to evaluate a parameter setting. These are the *`sklearn.metrics.accuracy_score`* for classification and *`sklearn.metrics.r2_score`* for regression. For some applications, other scoring functions are better suited (for example in unbalanced classification, the accuracy score is often uninformative). An alternative scoring function can be specified via the `scoring` parameter to *`GridSearchCV`*, *`RandomizedSearchCV`* and many of the specialized cross-validation tools described below. See *The scoring parameter: defining model evaluation rules* for more details.

Specifying multiple metrics for evaluation

`GridSearchCV` and `RandomizedSearchCV` allow specifying multiple metrics for the `scoring` parameter.

Multimetric scoring can either be specified as a list of strings of predefined scores names or a dict mapping the scorer name to the scorer function and/or the predefined scorer name(s). See *Using multiple metric evaluation* for more details.

When specifying multiple metrics, the `refit` parameter must be set to the metric (string) for which the `best_params_` will be found and used to build the `best_estimator_` on the whole dataset. If the search should not be refit, set `refit=False`. Leaving refit to the default value `None` will result in an error when using multiple metrics.

See *Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV* for an example usage.

Composite estimators and parameter spaces

GridSearchCV and *RandomizedSearchCV* allow searching over parameters of composite or nested estimators such as *Pipeline*, *ColumnTransformer*, *VotingClassifier* or *CalibratedClassifierCV* using a dedicated `<estimator>__<parameter>` syntax:

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.calibration import CalibratedClassifierCV
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons()
>>> calibrated_forest = CalibratedClassifierCV(
...     base_estimator=RandomForestClassifier(n_estimators=10))
>>> param_grid = {
...     'base_estimator__max_depth': [2, 4, 6, 8]}
>>> search = GridSearchCV(calibrated_forest, param_grid, cv=5)
>>> search.fit(X, y)
GridSearchCV(cv=5,
             estimator=CalibratedClassifierCV(...),
             param_grid={'base_estimator__max_depth': [2, 4, 6, 8]})
```

Here, `<estimator>` is the parameter name of the nested estimator, in this case `base_estimator`. If the meta-estimator is constructed as a collection of estimators as in `pipeline.Pipeline`, then `<estimator>` refers to the name of the estimator, see *Nested parameters*. In practice, there can be several levels of nesting:

```
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.feature_selection import SelectKBest
>>> pipe = Pipeline([
...     ('select', SelectKBest()),
...     ('model', calibrated_forest)])
>>> param_grid = {
...     'select__k': [1, 2],
...     'model__base_estimator__max_depth': [2, 4, 6, 8]}
>>> search = GridSearchCV(pipe, param_grid, cv=5).fit(X, y)
```

Model selection: development and evaluation

Model selection by evaluating various parameter settings can be seen as a way to use the labeled data to "train" the parameters of the grid.

When evaluating the resulting model it is important to do it on held-out samples that were not seen during the grid search process: it is recommended to split the data into a **development set** (to be fed to the `GridSearchCV` instance) and an **evaluation set** to compute performance metrics.

This can be done by using the *train_test_split* utility function.

Parallelism

GridSearchCV and *RandomizedSearchCV* evaluate each parameter setting independently. Computations can be run in parallel if your OS supports it, by using the keyword `n_jobs=-1`. See function signature for more details.

Robustness to failure

Some parameter settings may result in a failure to `fit` one or more folds of the data. By default, this will cause the entire search to fail, even if some parameter settings could be fully evaluated. Setting `error_score=0` (or `=np.NaN`) will make the procedure robust to such failure, issuing a warning and setting the score for that fold to 0 (or `NaN`), but completing the search.

Alternatives to brute force parameter search

Model specific cross-validation

Some models can fit data for a range of values of some parameter almost as efficiently as fitting the estimator for a single value of the parameter. This feature can be leveraged to perform a more efficient cross-validation used for model selection of this parameter.

The most common parameter amenable to this strategy is the parameter encoding the strength of the regularizer. In this case we say that we compute the **regularization path** of the estimator.

Here is the list of such models:

linear_model.ElasticNetCV([l1_ratio, eps, ...])	Elastic Net model with iterative fitting along a regularization path.
linear_model.LarsCV([fit_intercept, ...])	Cross-validated Least Angle Regression model.
linear_model.LassoCV([eps, n_alphas, ...])	Lasso linear model with iterative fitting along a regularization path.
linear_model.LassoLarsCV([fit_intercept, ...])	Cross-validated Lasso, using the LARS algorithm.
linear_model.LogisticRegressionCV([Cs, ...])	Logistic Regression CV (aka logit, MaxEnt) classifier.
linear_model.MultiTaskElasticNetCV([...])	Multi-task L1/L2 ElasticNet with built-in cross-validation.
linear_model.MultiTaskLassoCV([eps, ...])	Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.
linear_model.OrthogonalMatchingPursuitCV([...])	Cross-validated Orthogonal Matching Pursuit model (OMP).
linear_model.RidgeCV([alphas, ...])	Ridge regression with built-in cross-validation.
linear_model.RidgeClassifierCV([alphas, ...])	Ridge classifier with built-in cross-validation.

sklearn.linear_model.ElasticNetCV

class sklearn.linear_model.**ElasticNetCV**(*l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*, *normalize=False*, *precompute='auto'*, *max_iter=1000*, *tol=0.0001*, *cv=None*, *copy_X=True*, *verbose=0*, *n_jobs=None*, *positive=False*, *random_state=None*, *selection='cyclic'*)

Elastic Net model with iterative fitting along a regularization path.

See glossary entry for *cross-validation estimator*.

Read more in the *User Guide*.

Parameters

l1_ratio [float or array of floats, optional] float between 0 and 1 passed to ElasticNet (scaling between l1 and l2 penalties). For `l1_ratio = 0` the penalty is an L2 penalty. For `l1_ratio = 1` it is an L1 penalty. For `0 < l1_ratio < 1`, the penalty is a combination of L1 and L2 This parameter can be a list, in which case the different values are tested by cross-validation and the one giving the best prediction score is used. Note that a good choice of list of values for l1_ratio is often to put more values close to 1 (i.e. Lasso) and less close to 0 (i.e. Ridge), as in `[.1, .5, .7, .9, .95, .99, 1]`

eps [float, optional] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

n_alphas [int, optional] Number of alphas along the regularization path, used for each l1_ratio.

alphas [numpy array, optional] List of alphas where to compute the models. If None alphas are set automatically

fit_intercept [boolean] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

verbose [bool or integer] Amount of verbosity.

n_jobs [int or None, optional (default=None)] Number of CPUs to use during the cross validation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

positive [bool, optional] When set to `True`, forces the coefficients to be positive.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection` == 'random'.

selection [str, default 'cyclic'] If set to 'random', a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to 'random') often leads to significantly faster convergence especially when tol is higher than 1e-4.

Attributes

alpha_ [float] The amount of penalization chosen by cross validation

l1_ratio_ [float] The compromise between l1 and l2 penalization chosen by cross validation

coef_ [array, shape (n_features,) | (n_targets, n_features)] Parameter vector (w in the cost function formula),

intercept_ [float | array, shape (n_targets, n_features)] Independent term in the decision function.

mse_path_ [array, shape (n_l1_ratio, n_alpha, n_folds)] Mean square error for the test set on each fold, varying l1_ratio and alpha.

alphas_ [numpy array, shape (n_alphas,) or (n_l1_ratio, n_alphas)] The grid of alphas used for fitting, for each l1_ratio.

n_iter_ [int] number of iterations run by the coordinate descent solver to reach the specified tolerance for the optimal alpha.

See also:

enet_path

ElasticNet

Notes

For an example, see *examples/linear_model/plot_lasso_model_selection.py*.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

The parameter l1_ratio corresponds to alpha in the glmnet R package while alpha corresponds to the lambda parameter in glmnet. More specifically, the optimization objective is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

If you are interested in controlling the L1 and L2 penalty separately, keep in mind that this is equivalent to:

```
a * L1 + b * L2
```

for:

```
alpha = a + b and l1_ratio = a / (a + b).
```

Examples

```
>>> from sklearn.linear_model import ElasticNetCV
>>> from sklearn.datasets import make_regression
```

```
>>> X, y = make_regression(n_features=2, random_state=0)
>>> regr = ElasticNetCV(cv=5, random_state=0)
>>> regr.fit(X, y)
ElasticNetCV(cv=5, random_state=0)
>>> print(regr.alpha_)
0.199...
>>> print(regr.intercept_)
0.398...
>>> print(regr.predict([[0, 0]]))
[0.398...]
```

Methods

fit(self, X, y)	Fit linear model with coordinate descent
get_params(self[, deep])	Get parameters for this estimator.
path(X, y[, l1_ratio, eps, n_alphas, ...])	Compute elastic net path with coordinate descent.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*, *normalize=False*, *precompute='auto'*, *max_iter=1000*, *tol=0.0001*, *cv=None*, *copy_X=True*, *verbose=0*, *n_jobs=None*, *positive=False*, *random_state=None*, *selection='cyclic'*)

Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)

Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If y is mono-output, X can be sparse.

y [array-like, shape (n_samples,) or (n_samples, n_targets)] Target values

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

static path(*X*, *y*, *l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, *check_input=True*, ***params*)
Compute elastic net path with coordinate descent.

The elastic net optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] Target values.

l1_ratio [float, optional] Number between 0 and 1 passed to elastic net (scaling between l1 and l2 penalties). `l1_ratio=1` corresponds to the Lasso.

eps [float] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

n_alphas [int, optional] Number of alphas along the regularization path.

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [bool, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or int] Amount of verbosity.

return_n_iter [bool] Whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

check_input [bool, default True] Skip input validation checks, including the Gram matrix when provided assuming there are handled by the caller when check_input=False.

****params** [kwargs] Keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha. (Is returned when `return_n_iter` is set to True).

See also:

MultiTaskElasticNet

MultiTaskElasticNetCV

ElasticNet

ElasticNetCV

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

predict (*self*, *X*)

Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

sklearn.linear_model.LarsCV

class sklearn.linear_model.**LarsCV**(*fit_intercept=True*, *verbose=False*, *max_iter=500*, *normalize=True*, *precompute='auto'*, *cv=None*, *max_n_alphas=1000*, *n_jobs=None*, *eps=2.220446049250313e-16*, *copy_X=True*)
Cross-validated Least Angle Regression model.

See glossary entry for *cross-validation estimator*.

Read more in the *User Guide*.

Parameters

fit_intercept [bool, default=True] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

verbose [bool or int, default=False] Sets the verbosity amount

max_iter [int, default=500] Maximum number of iterations to perform.

normalize [bool, default=True] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

precompute [bool, 'auto' or array-like , default='auto'] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix cannot be passed as argument since we will use only subsets of X.

cv [int, cross-validation generator or an iterable, default=None] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

max_n_alphas [int, default=1000] The maximum number of points on the path used to compute the residuals in the cross-validation

n_jobs [int or None, default=None] Number of CPUs to use during the cross validation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. By default, `np.finfo(np.float).eps` is used.

copy_X [bool, default=True] If `True`, X will be copied; else, it may be overwritten.

Attributes

coef_ [array-like of shape (n_features,)] parameter vector (w in the formulation formula)

intercept_ [float] independent term in decision function

coef_path_ [array-like of shape (n_features, n_alphas)] the varying values of the coefficients along the path

alpha_ [float] the estimated regularization parameter alpha

alphas_ [array-like of shape (n_alphas,)] the different values of alpha along the path

cv_alphas_ [array-like of shape (n_cv_alphas,)] all the values of alpha along the path for the different folds

mse_path_ [array-like of shape (n_folds, n_cv_alphas)] the mean square error on left-out for each fold along the path (alpha values given by `cv_alphas`)

n_iter_ [array-like or int] the number of iterations run by Lars with the optimal alpha.

See also:

`lars_path`*, *`LassoLars`*, *`LassoLarsCV`

Examples

```
>>> from sklearn.linear_model import LarsCV
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(n_samples=200, noise=4.0, random_state=0)
>>> reg = LarsCV(cv=5).fit(X, y)
>>> reg.score(X, y)
0.9996...
>>> reg.alpha_
0.0254...
>>> reg.predict(X[:1,])
array([154.0842...])
```

Methods

fit(self, X, y)	Fit the model using X, y as training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *fit_intercept=True*, *verbose=False*, *max_iter=500*, *normalize=True*, *precompute='auto'*, *cv=None*, *max_n_alphas=1000*, *n_jobs=None*, *eps=2.220446049250313e-16*, *copy_X=True*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Fit the model using X, y as training data.

Parameters

X [array-like of shape (n_samples, n_features)] Training data.

y [array-like of shape (n_samples,)] Target values.

Returns

self [object] returns an instance of self.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

sklearn.linear_model.LassoCV

class sklearn.linear_model.**LassoCV**(*eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*, *normalize=False*, *precompute='auto'*, *max_iter=1000*, *tol=0.0001*, *copy_X=True*, *cv=None*, *verbose=False*, *n_jobs=None*, *positive=False*, *random_state=None*, *selection='cyclic'*)

Lasso linear model with iterative fitting along a regularization path.

See glossary entry for *cross-validation estimator*.

The best model is selected by cross-validation.

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Read more in the *User Guide*.

Parameters

eps [float, optional] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

n_alphas [int, optional] Number of alphas along the regularization path

alphas [numpy array, optional] List of alphas where to compute the models. If `None` alphas are set automatically

fit_intercept [boolean, default True] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

verbose [bool or integer] Amount of verbosity.

n_jobs [int or None, optional (default=None)] Number of CPUs to use during the cross validation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

positive [bool, optional] If positive, restrict regression coefficients to be positive

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, random_state is the seed used by the random number generator; If RandomState instance,

random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection` == 'random'.

selection [str, default 'cyclic'] If set to 'random', a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to 'random') often leads to significantly faster convergence especially when tol is higher than 1e-4.

Attributes

alpha_ [float] The amount of penalization chosen by cross validation

coef_ [array, shape (n_features,) | (n_targets, n_features)] parameter vector (w in the cost function formula)

intercept_ [float | array, shape (n_targets,)] independent term in decision function.

mse_path_ [array, shape (n_alphas, n_folds)] mean square error for the test set on each fold, varying alpha

alphas_ [numpy array, shape (n_alphas,)] The grid of alphas used for fitting

dual_gap_ [ndarray, shape ()] The dual gap at the end of the optimization for the optimal alpha (`alpha_`).

n_iter_ [int] number of iterations run by the coordinate descent solver to reach the specified tolerance for the optimal alpha.

See also:

lars_path

lasso_path

LassoLars

Lasso

LassoLarsCV

Notes

For an example, see *examples/linear_model/plot_lasso_model_selection.py*.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn.linear_model import LassoCV
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(noise=4, random_state=0)
>>> reg = LassoCV(cv=5, random_state=0).fit(X, y)
>>> reg.score(X, y)
0.9993...
>>> reg.predict(X[:1,])
array([-78.4951...])
```

Methods

fit(self, X, y)	Fit linear model with coordinate descent
get_params(self[, deep])	Get parameters for this estimator.
path(X, y[, eps, n_alphas, alphas, ...])	Compute Lasso path with coordinate descent
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*, *normalize=False*, *precompute='auto'*, *max_iter=1000*, *tol=0.0001*, *copy_X=True*, *cv=None*, *verbose=False*, *n_jobs=None*, *positive=False*, *random_state=None*, *selection='cyclic'*)

Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)

Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If y is mono-output, X can be sparse.

y [array-like, shape (n_samples,) or (n_samples, n_targets)] Target values

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

static path(*X*, *y*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, ***params*)

Compute Lasso path with coordinate descent

The Lasso optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^2_Fro + alpha * ||W||_21
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,), or (n_samples, n_outputs)] Target values

eps [float, optional] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`

n_alphas [int, optional] Number of alphas along the regularization path

alphas [ndarray, optional] List of alphas where to compute the models. If `None` alphas are set automatically

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or integer] Amount of verbosity.

return_n_iter [bool] whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

****params** [kwargs] keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha.

See also:

lars_path

Lasso

LassoLars

LassoCV

LassoLarsCV

sklearn.decomposition.sparse_encode

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Note that in certain cases, the Lars solver may be significantly faster to implement this functionality. In particular, linear interpolation can be used to retrieve model coefficients between the values output by lars_path

Examples

Comparing lasso_path and lars_path with interpolation:

```
>>> X = np.array([[1, 2, 3.1], [2.3, 5.4, 4.3]]).T
>>> y = np.array([1, 2, 3.1])
>>> # Use lasso_path to compute a coefficient path
>>> _, coef_path, _ = lasso_path(X, y, alphas=[5., 1., .5])
>>> print(coef_path)
[[0.          0.          0.46874778]
 [0.2159048   0.4425765   0.23689075]]
```

```
>>> # Now use lars_path and 1D linear interpolation to compute the
>>> # same path
>>> from sklearn.linear_model import lars_path
>>> alphas, active, coef_path_lars = lars_path(X, y, method='lasso')
>>> from scipy import interpolate
>>> coef_path_continuous = interpolate.interp1d(alphas[::-1],
...                                             coef_path_lars[:, ::-1])
>>> print(coef_path_continuous([5., 1., .5]))
[[0.          0.          0.46915237]
 [0.2159048   0.4425765   0.23668876]]
```

predict (*self*, *X*)

Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.linear_model.LassoCV`

- *Combine predictors using stacking*
- *Feature selection using SelectFromModel and LassoCV*
- *Lasso model selection: Cross-Validation / AIC / BIC*
- *Cross-validation on diabetes Dataset Exercise*

sklearn.linear_model.LassoLarsCV

class sklearn.linear_model.**LassoLarsCV**(*fit_intercept=True*, *verbose=False*, *max_iter=500*, *normalize=True*, *precompute='auto'*, *cv=None*, *max_n_alphas=1000*, *n_jobs=None*, *eps=2.220446049250313e-16*, *copy_X=True*, *positive=False*)

Cross-validated Lasso, using the LARS algorithm.

See glossary entry for *cross-validation estimator*.

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Read more in the *User Guide*.

Parameters

fit_intercept [bool, default=True] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

verbose [bool or int, default=False] Sets the verbosity amount

max_iter [int, default=500] Maximum number of iterations to perform.

normalize [bool, default=True] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

precompute [bool or 'auto' , default='auto'] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix cannot be passed as argument since we will use only subsets of X.

cv [int, cross-validation generator or an iterable, default=None] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

max_n_alphas [int, default=1000] The maximum number of points on the path used to compute the residuals in the cross-validation

n_jobs [int or None, default=None] Number of CPUs to use during the cross validation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. By default, `np.finfo(np.float).eps` is used.

copy_X [bool, default=True] If True, X will be copied; else, it may be overwritten.

positive [bool, default=False] Restrict coefficients to be >= 0. Be aware that you might want to remove fit_intercept which is set True by default. Under the positive restriction the model coefficients do not converge to the ordinary-least-squares solution for small values of alpha. Only coefficients up to the smallest alpha value (`alphas_[alphas_ > 0.].min()` when fit_path=True) reached by the stepwise Lars-Lasso algorithm are typically in congruence with the solution of the coordinate descent Lasso estimator. As a consequence using LassoLarsCV only makes sense for problems where a sparse solution is expected and/or reached.

Attributes

coef_ [array-like of shape (n_features,)] parameter vector (w in the formulation formula)

intercept_ [float] independent term in decision function.

coef_path_ [array-like of shape (n_features, n_alphas)] the varying values of the coefficients along the path

alpha_ [float] the estimated regularization parameter alpha

alphas_ [array-like of shape (n_alphas,)] the different values of alpha along the path

cv_alphas_ [array-like of shape (n_cv_alphas,)] all the values of alpha along the path for the different folds

mse_path_ [array-like of shape (n_folds, n_cv_alphas)] the mean square error on left-out for each fold along the path (alpha values given by `cv_alphas`)

n_iter_ [array-like or int] the number of iterations run by Lars with the optimal alpha.

See also:

lars_path, *LassoLars*, *LarsCV*, *LassoCV*

Notes

The object solves the same problem as the LassoCV object. However, unlike the LassoCV, it find the relevant alphas values by itself. In general, because of this property, it will be more stable. However, it is more fragile to heavily multicollinear datasets.

It is more efficient than the LassoCV if only a small number of features are selected compared to the total number, for instance if there are very few samples compared to the number of features.

Examples

```
>>> from sklearn.linear_model import LassoLarsCV
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(noise=4.0, random_state=0)
>>> reg = LassoLarsCV(cv=5).fit(X, y)
>>> reg.score(X, y)
0.9992...
>>> reg.alpha_
0.0484...
>>> reg.predict(X[:1,])
array([-77.8723...])
```

Methods

fit(self, X, y)	Fit the model using X, y as training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *fit_intercept=True*, *verbose=False*, *max_iter=500*, *normalize=True*, *precompute='auto'*, *cv=None*, *max_n_alphas=1000*, *n_jobs=None*, *eps=2.220446049250313e-16*, *copy_X=True*, *positive=False*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)
Fit the model using X, y as training data.

Parameters

X [array-like of shape (n_samples, n_features)] Training data.

y [array-like of shape (n_samples,)] Target values.

Returns

self [object] returns an instance of self.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.linear_model.LassoLarsCV`

- *Lasso model selection: Cross-Validation / AIC / BIC*

`sklearn.linear_model`.LogisticRegressionCV

class sklearn.linear_model.**LogisticRegressionCV**(*Cs=10*, *fit_intercept=True*, *cv=None*, *dual=False*, *penalty='l2'*, *scoring=None*, *solver='lbfgs'*, *tol=0.0001*, *max_iter=100*, *class_weight=None*, *n_jobs=None*, *verbose=0*, *refit=True*, *intercept_scaling=1.0*, *multi_class='auto'*, *random_state=None*, *l1_ratios=None*)

Logistic Regression CV (aka logit, MaxEnt) classifier.

See glossary entry for *cross-validation estimator*.

This class implements logistic regression using liblinear, newton-cg, sag of lbfgs optimizer. The newton-cg, sag and lbfgs solvers support only L2 regularization with primal formulation. The liblinear solver supports both L1 and L2 regularization, with a dual formulation only for the L2 penalty. Elastic-Net penalty is only supported by the saga solver.

For the grid of `Cs` values and `l1_ratios` values, the best hyperparameter is selected by the cross-validator *`StratifiedKFold`*, but it can be changed using the *cv* parameter. The 'newton-cg', 'sag', 'saga' and 'lbfgs' solvers can warm-start the coefficients (see *Glossary*).

Read more in the *User Guide*.

Parameters

Cs [int or list of floats, default=10] Each of the values in Cs describes the inverse of regularization strength. If Cs is as an int, then a grid of Cs values are chosen in a logarithmic scale between 1e-4 and 1e4. Like in support vector machines, smaller values specify stronger regularization.

fit_intercept [bool, default=True] Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.

cv [int or cross-validation generator, default=None] The default cross-validation generator used is Stratified K-Folds. If an integer is provided, then it is the number of folds used. See the module *`sklearn.model_selection`* module for the list of possible cross-validation objects.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

dual [bool, default=False] Dual or primal formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer dual=False when n_samples > n_features.

penalty [{'l1', 'l2', 'elasticnet'}, default='l2'] Used to specify the norm used in the penalization. The 'newton-cg', 'sag' and 'lbfgs' solvers support only l2 penalties. 'elasticnet' is only supported by the 'saga' solver.

scoring [str or callable, default=None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`. For a list of scoring functions that can be used, look at *`sklearn.metrics`*. The default scoring option used is 'accuracy'.

solver [{'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}, default='lbfgs'] Algorithm to use in the optimization problem.

- For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones.
- For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs' handle multinomial loss; 'liblinear' is limited to one-versus-rest schemes.
- 'newton-cg', 'lbfgs' and 'sag' only handle L2 penalty, whereas 'liblinear' and 'saga' handle L1 penalty.
- 'liblinear' might be slower in LogisticRegressionCV because it does not handle warm-starting.

Note that 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from sklearn.preprocessing.

New in version 0.17: Stochastic Average Gradient descent solver.

New in version 0.19: SAGA solver.

tol [float, default=1e-4] Tolerance for stopping criteria.

max_iter [int, default=100] Maximum number of iterations of the optimization algorithm.

class_weight [dict or 'balanced', default=None] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

New in version 0.17: class_weight == 'balanced'

n_jobs [int, default=None] Number of CPU cores used during the cross-validation loop. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [int, default=0] For the 'liblinear', 'sag' and 'lbfgs' solvers set verbose to any positive number for verbosity.

refit [bool, default=True] If set to True, the scores are averaged across all folds, and the coefs and the C that corresponds to the best score is taken, and a final refit is done using these parameters. Otherwise the coefs, intercepts and C that correspond to the best scores across folds are averaged.

intercept_scaling [float, default=1] Useful only when the solver 'liblinear' is used and self.fit_intercept is set to True. In this case, x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equal to intercept_scaling is appended to the instance vector. The intercept becomes `intercept_scaling * synthetic_feature_weight`.

Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.

multi_class [{'auto, 'ovr', 'multinomial'}, default='auto'] If the option chosen is 'ovr', then a binary problem is fit for each label. For 'multinomial' the loss minimised is the multinomial loss fit across the entire probability distribution, *even when the data is binary*. 'multinomial' is unavailable when solver='liblinear'. 'auto' selects 'ovr' if the data is binary, or if solver='liblinear', and otherwise selects 'multinomial'.

New in version 0.18: Stochastic Average Gradient descent solver for 'multinomial' case.

Changed in version 0.22: Default changed from 'ovr' to 'auto' in 0.22.

random_state [int, RandomState instance, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver='sag'` or `solver='liblinear'`. Note that this only applies to the solver and not the cross-validation generator.

l1_ratios [list of float, default=None] The list of Elastic-Net mixing parameter, with `0 <= l1_ratio <= 1`. Only used if `penalty='elasticnet'`. A value of 0 is equivalent to using `penalty='l2'`, while 1 is equivalent to using `penalty='l1'`. For `0 < l1_ratio <1`, the penalty is a combination of L1 and L2.

Attributes

classes_ [ndarray of shape (n_classes,)] A list of class labels known to the classifier.

coef_ [ndarray of shape (1, n_features) or (n_classes, n_features)] Coefficient of the features in the decision function.

`coef_` is of shape (1, n_features) when the given problem is binary.

intercept_ [ndarray of shape (1,) or (n_classes,)] Intercept (a.k.a. bias) added to the decision function.

If `fit_intercept` is set to False, the intercept is set to zero. `intercept_` is of shape(1,) when the problem is binary.

Cs_ [ndarray of shape (n_cs)] Array of C i.e. inverse of regularization parameter values used for cross-validation.

l1_ratios_ [ndarray of shape (n_l1_ratios)] Array of l1_ratios used for cross-validation. If no l1_ratio is used (i.e. penalty is not 'elasticnet'), this is set to `[None]`

coefs_paths_ [ndarray of shape (n_folds, n_cs, n_features) or (n_folds, n_cs, n_features + 1)] dict with classes as the keys, and the path of coefficients obtained during cross-validating across each fold and then across each Cs after doing an OvR for the corresponding class as values. If the 'multi_class' option is set to 'multinomial', then the coefs_paths are the coefficients corresponding to each class. Each dict value has shape `(n_folds, n_cs, n_features)` or `(n_folds, n_cs, n_features + 1)` depending on whether the intercept is fit or not. If `penalty='elasticnet'`, the shape is `(n_folds, n_cs, n_l1_ratios_, n_features)` or `(n_folds, n_cs, n_l1_ratios_, n_features + 1)`.

scores_ [dict] dict with classes as the keys, and the values as the grid of scores obtained during cross-validating each fold, after doing an OvR for the corresponding class. If the 'multi_class' option given is 'multinomial' then the same scores are repeated across all classes, since this is the multinomial class. Each dict value has shape `(n_folds, n_cs` or `(n_folds, n_cs, n_l1_ratios)` if `penalty='elasticnet'`.

C_ [ndarray of shape (n_classes,) or (n_classes - 1,)] Array of C that maps to the best scores across every class. If refit is set to False, then for each class, the best C is the average of the C's that correspond to the best scores for each fold. `C_` is of shape(n_classes,) when the problem is binary.

l1_ratio_ [ndarray of shape (n_classes,) or (n_classes - 1,)] Array of l1_ratio that maps to the best scores across every class. If refit is set to False, then for each class, the best l1_ratio is the average of the l1_ratio's that correspond to the best scores for each fold. `l1_ratio_` is of shape(n_classes,) when the problem is binary.

n_iter_ [ndarray of shape (n_classes, n_folds, n_cs) or (1, n_folds, n_cs)] Actual number of iterations for all classes, folds and Cs. In the binary or multinomial cases, the first dimension is equal to 1. If `penalty='elasticnet'`, the shape is `(n_classes, n_folds, n_cs, n_l1_ratios)` or `(1, n_folds, n_cs, n_l1_ratios)`.

See also:

LogisticRegression

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import LogisticRegressionCV
>>> X, y = load_iris(return_X_y=True)
>>> clf = LogisticRegressionCV(cv=5, random_state=0).fit(X, y)
>>> clf.predict(X[:2, :])
array([0, 0])
>>> clf.predict_proba(X[:2, :]).shape
(2, 3)
>>> clf.score(X, y)
0.98...
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
densify(self)	Convert coefficient matrix to dense array format.
fit(self, X, y[, sample_weight])	Fit the model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class labels for samples in X.
predict_log_proba(self, X)	Predict logarithm of probability estimates.
predict_proba(self, X)	Probability estimates.
score(self, X, y[, sample_weight])	Returns the score using the `scoring` option on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.
sparsify(self)	Convert coefficient matrix to sparse format.

__init__(*self*, *Cs=10*, *fit_intercept=True*, *cv=None*, *dual=False*, *penalty='l2'*, *scoring=None*, *solver='lbfgs'*, *tol=0.0001*, *max_iter=100*, *class_weight=None*, *n_jobs=None*, *verbose=0*, *refit=True*, *intercept_scaling=1.0*, *multi_class='auto'*, *random_state=None*, *l1_ratios=None*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

densify(*self*)
Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns

self Fitted estimator.

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit the model according to the given training data.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y [array-like of shape (n_samples,)] Target vector relative to X.

sample_weight [array-like of shape (n_samples,) default=None] Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

predict_log_proba (*self*, *X*)

Predict logarithm of probability estimates.

The returned estimates for all classes are ordered by the label of classes.

Parameters

X [array-like of shape (n_samples, n_features)] Vector to be scored, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns

T [array-like of shape (n_samples, n_classes)] Returns the log-probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

predict_proba (*self*, *X*)

Probability estimates.

The returned estimates for all classes are ordered by the label of classes.

For a multi_class problem, if multi_class is set to be "multinomial" the softmax function is used to find the predicted probability of each class. Else use a one-vs-rest approach, i.e calculate the probability of each class assuming it to be positive using the logistic function. and normalize these values across all the classes.

Parameters

X [array-like of shape (n_samples, n_features)] Vector to be scored, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns

T [array-like of shape (n_samples, n_classes)] Returns the probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

score (*self*, *X*, *y*, *sample_weight=None*)

Returns the score using the `scoring` option on the given test data and labels.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Score of self.predict(X) wrt. y.

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

sparsify(*self*)

Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual numpy.ndarray representation.

The `intercept_` member is not converted.

Returns

self Fitted estimator.

Notes

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the partial_fit method (if any) will not work until you call densify.

sklearn.linear_model.MultiTaskElasticNetCV

class sklearn.linear_model.**MultiTaskElasticNetCV**(*l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*, *normalize=False*, *max_iter=1000*, *tol=0.0001*, *cv=None*, *copy_X=True*, *verbose=0*, *n_jobs=None*, *random_state=None*, *selection='cyclic'*)

Multi-task L1/L2 ElasticNet with built-in cross-validation.

See glossary entry for *cross-validation estimator*.

The optimization objective for MultiTaskElasticNet is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

New in version 0.15.

Parameters

l1_ratio [float or array of floats] The ElasticNet mixing parameter, with 0 < l1_ratio <= 1. For l1_ratio = 1 the penalty is an L1/L2 penalty. For l1_ratio = 0 it is an L2 penalty. For `0 < l1_ratio < 1`, the penalty is a combination of L1/L2 and L2. This parameter can be a list, in which case the different values are tested by cross-validation and the one giving the best prediction score is used. Note that a good choice of list of values for l1_ratio is often

to put more values close to 1 (i.e. Lasso) and less close to 0 (i.e. Ridge), as in `[.1, .5, .7, .9, .95, .99, 1]`

eps [float, optional] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

n_alphas [int, optional] Number of alphas along the regularization path

alphas [array-like, optional] List of alphas where to compute the models. If not provided, set automatically.

fit_intercept [boolean] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

verbose [bool or integer] Amount of verbosity.

n_jobs [int or None, optional (default=None)] Number of CPUs to use during the cross validation. Note that this is used only if multiple values for l1_ratio are given. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection` == 'random'.

selection [str, default 'cyclic'] If set to 'random', a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to 'random') often leads to significantly faster convergence especially when tol is higher than 1e-4.

Attributes

intercept_ [array, shape (n_tasks,)] Independent term in decision function.

coef_ [array, shape (n_tasks, n_features)] Parameter vector (W in the cost function formula). Note that `coef_` stores the transpose of `W`, `W.T`.

alpha_ [float] The amount of penalization chosen by cross validation

mse_path_ [array, shape (n_alphas, n_folds) or (n_l1_ratio, n_alphas, n_folds)] mean square error for the test set on each fold, varying alpha

alphas_ [numpy array, shape (n_alphas,) or (n_l1_ratio, n_alphas)] The grid of alphas used for fitting, for each l1_ratio

l1_ratio_ [float] best l1_ratio obtained by cross-validation.

n_iter_ [int] number of iterations run by the coordinate descent solver to reach the specified tolerance for the optimal alpha.

See also:

MultiTaskElasticNet

ElasticNetCV

MultiTaskLassoCV

Notes

The algorithm used to fit the model is coordinate descent.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.MultiTaskElasticNetCV(cv=3)
>>> clf.fit([[0,0], [1, 1], [2, 2]],
...         [[0, 0], [1, 1], [2, 2]])
MultiTaskElasticNetCV(cv=3)
>>> print(clf.coef_)
[[0.52875032 0.46958558]
 [0.52875032 0.46958558]]
>>> print(clf.intercept_)
[0.00166409 0.00166409]
```

Methods

fit(self, X, y)	Fit linear model with coordinate descent
get_params(self[, deep])	Get parameters for this estimator.
path(X, y[, l1_ratio, eps, n_alphas, ...])	Compute elastic net path with coordinate descent.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*, *normalize=False*, *max_iter=1000*, *tol=0.0001*, *cv=None*, *copy_X=True*, *verbose=0*, *n_jobs=None*, *random_state=None*, *selection='cyclic'*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If y is mono-output, X can be sparse.

y [array-like, shape (n_samples,) or (n_samples, n_targets)] Target values

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

static path (*X*, *y*, *l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, *check_input=True*, ***params*)
Compute elastic net path with coordinate descent.

The elastic net optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] Target values.

l1_ratio [float, optional] Number between 0 and 1 passed to elastic net (scaling between l1 and l2 penalties). `l1_ratio=1` corresponds to the Lasso.

eps [float] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

n_alphas [int, optional] Number of alphas along the regularization path.

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [bool, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or int] Amount of verbosity.

return_n_iter [bool] Whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

check_input [bool, default True] Skip input validation checks, including the Gram matrix when provided assuming there are handled by the caller when check_input=False.

****params** [kwargs] Keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha. (Is returned when `return_n_iter` is set to True).

See also:

MultiTaskElasticNet

MultiTaskElasticNetCV

ElasticNet

ElasticNetCV

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

predict(*self*, *X*)

Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

sklearn.linear_model.MultiTaskLassoCV

class sklearn.linear_model.**MultiTaskLassoCV**(*eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*, *normalize=False*, *max_iter=1000*, *tol=0.0001*, *copy_X=True*, *cv=None*, *verbose=False*, *n_jobs=None*, *random_state=None*, *selection='cyclic'*)

Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.

See glossary entry for *cross-validation estimator*.

The optimization objective for MultiTaskLasso is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2 + alpha * ||W||_21
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

New in version 0.15.

Parameters

eps [float, optional] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

n_alphas [int, optional] Number of alphas along the regularization path

alphas [array-like, optional] List of alphas where to compute the models. If not provided, set automatically.

fit_intercept [boolean] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

max_iter [int, optional] The maximum number of iterations.

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

verbose [bool or integer] Amount of verbosity.

n_jobs [int or None, optional (default=None)] Number of CPUs to use during the cross validation. Note that this is used only if multiple values for l1_ratio are given. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection` == 'random'

selection [str, default 'cyclic'] If set to 'random', a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to 'random') often leads to significantly faster convergence especially when tol is higher than 1e-4.

Attributes

intercept_ [array, shape (n_tasks,)] Independent term in decision function.

coef_ [array, shape (n_tasks, n_features)] Parameter vector (W in the cost function formula). Note that `coef_` stores the transpose of `W`, `W.T`.

alpha_ [float] The amount of penalization chosen by cross validation

mse_path_ [array, shape (n_alphas, n_folds)] mean square error for the test set on each fold, varying alpha

alphas_ [numpy array, shape (n_alphas,)] The grid of alphas used for fitting.

n_iter_ [int] number of iterations run by the coordinate descent solver to reach the specified tolerance for the optimal alpha.

See also:

MultiTaskElasticNet

ElasticNetCV

MultiTaskElasticNetCV

Notes

The algorithm used to fit the model is coordinate descent.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn.linear_model import MultiTaskLassoCV
>>> from sklearn.datasets import make_regression
>>> from sklearn.metrics import r2_score
>>> X, y = make_regression(n_targets=2, noise=4, random_state=0)
>>> reg = MultiTaskLassoCV(cv=5, random_state=0).fit(X, y)
>>> r2_score(y, reg.predict(X))
0.9994...
>>> reg.alpha_
0.5713...
>>> reg.predict(X[:1,])
array([[153.7971...,  94.9015...]])
```

Methods

fit(self, X, y)	Fit linear model with coordinate descent
get_params(self[, deep])	Get parameters for this estimator.
path(X, y[, eps, n_alphas, alphas, ...])	Compute Lasso path with coordinate descent
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*, *normalize=False*, *max_iter=1000*, *tol=0.0001*, *copy_X=True*, *cv=None*, *verbose=False*, *n_jobs=None*, *random_state=None*, *selection='cyclic'*)

Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)

Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If y is mono-output, X can be sparse.

y [array-like, shape (n_samples,) or (n_samples, n_targets)] Target values

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

static path(*X*, *y*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, ***params*)

Compute Lasso path with coordinate descent

The Lasso optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^2_Fro + alpha * ||W||_21
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,), or (n_samples, n_outputs)] Target values

eps [float, optional] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`

n_alphas [int, optional] Number of alphas along the regularization path

alphas [ndarray, optional] List of alphas where to compute the models. If `None` alphas are set automatically

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or integer] Amount of verbosity.

return_n_iter [bool] whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

****params** [kwargs] keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha.

See also:

lars_path

Lasso

LassoLars

LassoCV

LassoLarsCV

sklearn.decomposition.sparse_encode

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Note that in certain cases, the Lars solver may be significantly faster to implement this functionality. In particular, linear interpolation can be used to retrieve model coefficients between the values output by lars_path

Examples

Comparing lasso_path and lars_path with interpolation:

```
>>> X = np.array([[1, 2, 3.1], [2.3, 5.4, 4.3]]).T
>>> y = np.array([1, 2, 3.1])
>>> # Use lasso_path to compute a coefficient path
>>> _, coef_path, _ = lasso_path(X, y, alphas=[5., 1., .5])
>>> print(coef_path)
[[0.          0.          0.46874778]
 [0.2159048   0.4425765   0.23689075]]
```

```
>>> # Now use lars_path and 1D linear interpolation to compute the
>>> # same path
>>> from sklearn.linear_model import lars_path
>>> alphas, active, coef_path_lars = lars_path(X, y, method='lasso')
>>> from scipy import interpolate
>>> coef_path_continuous = interpolate.interp1d(alphas[::-1],
...                                             coef_path_lars[:, ::-1])
>>> print(coef_path_continuous([5., 1., .5]))
[[0.          0.          0.46915237]
 [0.2159048   0.4425765   0.23668876]]
```

predict (*self*, *X*)

Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

sklearn.linear_model.OrthogonalMatchingPursuitCV

class sklearn.linear_model.**OrthogonalMatchingPursuitCV**(*copy=True*, *fit_intercept=True*, *normalize=True*, *max_iter=None*, *cv=None*, *n_jobs=None*, *verbose=False*)
Cross-validated Orthogonal Matching Pursuit model (OMP).

See glossary entry for *cross-validation estimator*.

Read more in the *User Guide*.

Parameters

copy [bool, optional] Whether the design matrix X must be copied by the algorithm. A false value is only helpful if X is already Fortran-ordered, otherwise a copy is made anyway.

fit_intercept [boolean, optional] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [boolean, optional, default True] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

max_iter [integer, optional] Maximum numbers of iterations to perform, therefore maximum features to include. 10% of `n_features` but at least 5 if available.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

n_jobs [int or None, optional (default=None)] Number of CPUs to use during the cross validation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [boolean or integer, optional] Sets the verbosity amount

Attributes

intercept_ [float or array, shape (n_targets,)] Independent term in decision function.

coef_ [array, shape (n_features,) or (n_targets, n_features)] Parameter vector (w in the problem formulation).

n_nonzero_coefs_ [int] Estimated number of non-zero coefficients giving the best mean squared error over the cross-validation folds.

n_iter_ [int or array-like] Number of active features across every target for the model refit with the best hyperparameters got by cross-validating across all folds.

See also:

orthogonal_mp

orthogonal_mp_gram

lars_path

Lars

LassoLars

OrthogonalMatchingPursuit

LarsCV

LassoLarsCV

decomposition.sparse_encode

Examples

```
>>> from sklearn.linear_model import OrthogonalMatchingPursuitCV
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(n_features=100, n_informative=10,
...                        noise=4, random_state=0)
>>> reg = OrthogonalMatchingPursuitCV(cv=5).fit(X, y)
>>> reg.score(X, y)
0.9991...
>>> reg.n_nonzero_coefs_
10
>>> reg.predict(X[:1,])
array([-78.3854...])
```

Methods

fit(self, X, y)	Fit the model using X, y as training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *copy=True*, *fit_intercept=True*, *normalize=True*, *max_iter=None*, *cv=None*, *n_jobs=None*, *verbose=False*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)
Fit the model using X, y as training data.

Parameters

X [array-like, shape [n_samples, n_features]] Training data.

y [array-like, shape [n_samples]] Target values. Will be cast to X's dtype if necessary

Returns

self [object] returns an instance of self.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.linear_model.OrthogonalMatchingPursuitCV`

- *Orthogonal Matching Pursuit*

sklearn.linear_model.RidgeCV

class sklearn.linear_model.**RidgeCV**(*alphas=(0.1, 1.0, 10.0)*, *fit_intercept=True*, *normalize=False*, *scoring=None*, *cv=None*, *gcv_mode=None*, *store_cv_values=False*)

Ridge regression with built-in cross-validation.

See glossary entry for *cross-validation estimator*.

By default, it performs Generalized Cross-Validation, which is a form of efficient Leave-One-Out cross-validation.

Read more in the *User Guide*.

Parameters

alphas [ndarray of shape (n_alphas,), default=(0.1, 1.0, 10.0)] Array of alpha values to try. Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to `C^-1` in other linear models such as LogisticRegression or LinearSVC. If using generalized cross-validation, alphas must be positive.

fit_intercept [bool, default=True] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [bool, default=False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

scoring [string, callable, default=None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`. If None, the negative mean squared error if cv is 'auto' or None (i.e. when using generalized cross-validation), and r2 score otherwise.

cv [int, cross-validation generator or an iterable, default=None] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the efficient Leave-One-Out cross-validation (also known as Generalized Cross-Validation).
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if `y` is binary or multiclass, *`sklearn.model_selection.StratifiedKFold`* is used, else, *`sklearn.model_selection.KFold`* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

gcv_mode [{'auto', 'svd', eigen'}, default='auto'] Flag indicating which strategy to use when performing Generalized Cross-Validation. Options are:

```
'auto' : use 'svd' if n_samples > n_features, otherwise use 'eigen'
'svd' : force use of singular value decomposition of X when X is
    dense, eigenvalue decomposition of X^T.X when X is sparse.
'eigen' : force computation via eigendecomposition of X.X^T
```

The 'auto' mode is the default and is intended to pick the cheaper option of the two depending on the shape of the training data.

store_cv_values [bool, default=False] Flag indicating if the cross-validation values corresponding to each alpha should be stored in the `cv_values_` attribute (see below). This flag is only compatible with `cv=None` (i.e. using Generalized Cross-Validation).

Attributes

cv_values_ [ndarray of shape (n_samples, n_alphas) or shape (n_samples, n_targets, n_alphas), optional] Cross-validation values for each alpha (if `store_cv_values=True` and `cv=None`). After `fit()` has been called, this attribute will contain the mean squared errors (by default) or the values of the `{loss,score}_func` function (if provided in the constructor).

coef_ [ndarray of shape (n_features) or (n_targets, n_features)] Weight vector(s).

intercept_ [float or ndarray of shape (n_targets,)] Independent term in decision function. Set to 0.0 if `fit_intercept = False`.

alpha_ [float] Estimated regularization parameter.

See also:

Ridge Ridge regression

RidgeClassifier Ridge classifier

RidgeClassifierCV Ridge classifier with built-in cross validation

Examples

```
>>> from sklearn.datasets import load_diabetes
>>> from sklearn.linear_model import RidgeCV
>>> X, y = load_diabetes(return_X_y=True)
>>> clf = RidgeCV(alphas=[1e-3, 1e-2, 1e-1, 1]).fit(X, y)
>>> clf.score(X, y)
0.5166...
```

Methods

fit(self, X, y[, sample_weight])	Fit Ridge regression model with cv.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *alphas=(0.1, 1.0, 10.0)*, *fit_intercept=True*, *normalize=False*, *scoring=None*, *cv=None*, *gcv_mode=None*, *store_cv_values=False*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit Ridge regression model with cv.

Parameters

X [ndarray of shape (n_samples, n_features)] Training data. If using GCV, will be cast to float64 if necessary.

y [ndarray of shape (n_samples,) or (n_samples, n_targets)] Target values. Will be cast to X's dtype if necessary.

sample_weight [float or ndarray of shape (n_samples,), default=None] Individual weights for each sample. If given a float, every sample will have the same weight.

Returns

self [object]

Notes

When sample_weight is provided, the selected hyperparameter may depend on whether we use generalized cross-validation (cv=None or cv='auto') or another form of cross-validation, because only generalized cross-validation takes the sample weights into account when computing the validation score.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.linear_model.RidgeCV`

- *Face completion with a multi-output estimators*
- *Combine predictors using stacking*
- *Effect of transforming the targets in regression model*

`sklearn.linear_model`.RidgeClassifierCV

class sklearn.linear_model.**RidgeClassifierCV**(*alphas=(0.1, 1.0, 10.0)*, *fit_intercept=True*, *normalize=False*, *scoring=None*, *cv=None*, *class_weight=None*, *store_cv_values=False*)

Ridge classifier with built-in cross-validation.

See glossary entry for *cross-validation estimator*.

By default, it performs Generalized Cross-Validation, which is a form of efficient Leave-One-Out cross-validation. Currently, only the n_features > n_samples case is handled efficiently.

Read more in the *User Guide*.

Parameters

alphas [ndarray of shape (n_alphas,), default=(0.1, 1.0, 10.0)] Array of alpha values to try. Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to `C^-1` in other linear models such as LogisticRegression or LinearSVC.

fit_intercept [bool, default=True] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [bool, default=False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use

sklearn.preprocessing.StandardScaler before calling `fit` on an estimator with `normalize=False`.

scoring [string, callable, default=None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`.

cv [int, cross-validation generator or an iterable, default=None] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the efficient Leave-One-Out cross-validation
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

Refer *User Guide* for the various cross-validation strategies that can be used here.

class_weight [dict or 'balanced', default=None] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

store_cv_values [bool, default=False] Flag indicating if the cross-validation values corresponding to each alpha should be stored in the `cv_values_` attribute (see below). This flag is only compatible with `cv=None` (i.e. using Generalized Cross-Validation).

Attributes

cv_values_ [ndarray of shape (n_samples, n_targets, n_alphas), optional] Cross-validation values for each alpha (if `store_cv_values=True` and `cv=None`). After `fit()` has been called, this attribute will contain the mean squared errors (by default) or the values of the `{loss,score}_func` function (if provided in the constructor). This attribute exists only when `store_cv_values` is True.

coef_ [ndarray of shape (1, n_features) or (n_targets, n_features)] Coefficient of the features in the decision function.

`coef_` is of shape (1, n_features) when the given problem is binary.

intercept_ [float or ndarray of shape (n_targets,)] Independent term in decision function. Set to 0.0 if `fit_intercept = False`.

alpha_ [float] Estimated regularization parameter

classes_ [ndarray of shape (n_classes,)] The classes labels.

See also:

Ridge Ridge regression

RidgeClassifier Ridge classifier

RidgeCV Ridge regression with built-in cross validation

Notes

For multi-class classification, n_class classifiers are trained in a one-versus-all approach. Concretely, this is implemented by taking advantage of the multi-variate response support in Ridge.

Examples

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.linear_model import RidgeClassifierCV
>>> X, y = load_breast_cancer(return_X_y=True)
>>> clf = RidgeClassifierCV(alphas=[1e-3, 1e-2, 1e-1, 1]).fit(X, y)
>>> clf.score(X, y)
0.9630...
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
fit(self, X, y[, sample_weight])	Fit Ridge classifier with cv.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class labels for samples in X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *alphas=(0.1, 1.0, 10.0)*, *fit_intercept=True*, *normalize=False*, *scoring=None*, *cv=None*, *class_weight=None*, *store_cv_values=False*)
Initialize self. See help(type(self)) for accurate signature.

decision_function (*self*, *X*)
Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit Ridge classifier with cv.

Parameters

X [ndarray of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features. When using GCV, will be cast to float64 if necessary.

y [ndarray of shape (n_samples,)] Target values. Will be cast to X's dtype if necessary.

sample_weight [float or ndarray of shape (n_samples,), default=None] Individual weights for each sample. If given a float, every sample will have the same weight.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Information Criterion

Some models can offer an information-theoretic closed-form formula of the optimal estimate of the regularization parameter by computing a single regularization path (instead of several when using cross-validation).

Here is the list of models benefiting from the Akaike Information Criterion (AIC) or the Bayesian Information Criterion (BIC) for automated model selection:

linear_model.LassoLarsIC([criterion, ...])	Lasso model fit with Lars using BIC or AIC for model selection

sklearn.linear_model.LassoLarsIC

class sklearn.linear_model.**LassoLarsIC**(*criterion='aic'*, *fit_intercept=True*, *verbose=False*, *normalize=True*, *precompute='auto'*, *max_iter=500*, *eps=2.220446049250313e-16*, *copy_X=True*, *positive=False*)

Lasso model fit with Lars using BIC or AIC for model selection

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

AIC is the Akaike information criterion and BIC is the Bayes Information criterion. Such criteria are useful to select the value of the regularization parameter by making a trade-off between the goodness of fit and the complexity of the model. A good model should explain well the data while being simple.

Read more in the *User Guide*.

Parameters

criterion [{'bic' , 'aic'}, default='aic'] The type of criterion to use.

fit_intercept [bool, default=True] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

verbose [bool or int, default=False] Sets the verbosity amount

normalize [bool, default=True] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

precompute [bool, 'auto' or array-like, default='auto'] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

max_iter [int, default=500] Maximum number of iterations to perform. Can be used for early stopping.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the `tol` parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization. By default, `np.finfo(np.float).eps` is used

copy_X [bool, default=True] If True, X will be copied; else, it may be overwritten.

positive [bool, default=False] Restrict coefficients to be >= 0. Be aware that you might want to remove fit_intercept which is set True by default. Under the positive restriction the model coefficients do not converge to the ordinary-least-squares solution for small values of alpha. Only coefficients up to the smallest alpha value (`alphas_[alphas_ > 0.].min()` when fit_path=True) reached by the stepwise Lars-Lasso algorithm are typically in congruence with the solution of the coordinate descent Lasso estimator. As a consequence using LassoLarsIC only makes sense for problems where a sparse solution is expected and/or reached.

Attributes

coef_ [array-like of shape (n_features,)] parameter vector (w in the formulation formula)

intercept_ [float] independent term in decision function.

alpha_ [float] the alpha parameter chosen by the information criterion

n_iter_ [int] number of iterations run by lars_path to find the grid of alphas.

criterion_ [array-like of shape (n_alphas,)] The value of the information criteria ('aic', 'bic') across all alphas. The alpha which has the smallest information criterion is chosen. This value is larger by a factor of `n_samples` compared to Eqns. 2.15 and 2.16 in (Zou et al, 2007).

See also:

lars_path, *LassoLars*, *LassoLarsCV*

Notes

The estimation of the number of degrees of freedom is given by:

"On the degrees of freedom of the lasso" Hui Zou, Trevor Hastie, and Robert Tibshirani Ann. Statist. Volume 35, Number 5 (2007), 2173-2192.

https://en.wikipedia.org/wiki/Akaike_information_criterion https://en.wikipedia.org/wiki/Bayesian_information_criterion

Examples

```
>>> from sklearn import linear_model
>>> reg = linear_model.LassoLarsIC(criterion='bic')
>>> reg.fit([[-1, 1], [0, 0], [1, 1]], [-1.1111, 0, -1.1111])
LassoLarsIC(criterion='bic')
>>> print(reg.coef_)
[ 0.  -1.11...]
```

Methods

fit(self, X, y[, copy_X])	Fit the model using X, y as training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *criterion='aic'*, *fit_intercept=True*, *verbose=False*, *normalize=True*, *precompute='auto'*, *max_iter=500*, *eps=2.220446049250313e-16*, *copy_X=True*, *positive=False*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *copy_X=None*)
Fit the model using X, y as training data.

Parameters

X [array-like of shape (n_samples, n_features)] training data.

y [array-like of shape (n_samples,)] target values. Will be cast to X's dtype if necessary

copy_X [bool, default=None] If provided, this parameter will override the choice of copy_X

made at instance creation. If `True`, X will be copied; else, it may be overwritten.

Returns

self [object] returns an instance of self.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.linear_model.LassoLarsIC`

- *Lasso model selection: Cross-Validation / AIC / BIC*

Out of Bag Estimates

When using ensemble methods base upon bagging, i.e. generating new training sets using sampling with replacement, part of the training set remains unused. For each classifier in the ensemble, a different part of the training set is left out.

This left out portion can be used to estimate the generalization error without having to rely on a separate validation set. This estimate comes "for free" as no additional data is needed and can be used for model selection.

This is currently implemented in the following classes:

ensemble.RandomForestClassifier([...])	A random forest classifier.
ensemble.RandomForestRegressor([...])	A random forest regressor.
ensemble.ExtraTreesClassifier([...])	An extra-trees classifier.
ensemble.ExtraTreesRegressor([n_estimators, ...])	An extra-trees regressor.
ensemble.GradientBoostingClassifier([loss, ...])	Gradient Boosting for classification.
ensemble.GradientBoostingRegressor([loss, ...])	Gradient Boosting for regression.

`sklearn.ensemble`.RandomForestClassifier

class sklearn.ensemble.**RandomForestClassifier**(*n_estimators=100*, *criterion='gini'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *bootstrap=True*, *oob_score=False*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*, *class_weight=None*, *ccp_alpha=0.0*, *max_samples=None*)

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if `bootstrap=True` (default).

Read more in the *User Guide*.

Parameters

n_estimators [integer, optional (default=100)] The number of trees in the forest.

Changed in version 0.22: The default value of `n_estimators` changed from 10 to 100 in 0.22.

criterion [string, optional (default="gini")] The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Note: this parameter is tree-specific.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features [int, float, string or None, optional (default="auto")] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=sqrt(n_features)`.
- If "sqrt", then `max_features=sqrt(n_features)` (same as "auto").
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

max_leaf_nodes [int or None, optional (default=None)] Grow trees with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

bootstrap [boolean, optional (default=True)] Whether bootstrap samples are used when building trees. If False, the whole datset is used to build each tree.

oob_score [bool (default=False)] Whether to use out-of-bag samples to estimate the generalization accuracy.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel. *fit*, *predict*, *decision_path* and *apply* are all parallelized over the trees. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional (default=None)] Controls both the randomness of the bootstrapping of the samples used when building trees (if `bootstrap=True`) and the sampling of the features to consider when looking for the best split at each node (if `max_features < n_features`). See *Glossary* for details.

verbose [int, optional (default=0)] Controls the verbosity when fitting and predicting.

warm_start [bool, optional (default=False)] When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See *the Glossary*.

class_weight [dict, list of dicts, "balanced", "balanced_subsample" or None, optional (default=None)] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

The "balanced_subsample" mode is the same as "balanced" except that weights are computed based on the bootstrap sample for every tree grown.

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

ccp_alpha [non-negative float, optional (default=0.0)] Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than `ccp_alpha` will be chosen. By default, no pruning is performed. See *Minimal Cost-Complexity Pruning* for details.

New in version 0.22.

max_samples [int or float, default=None] If bootstrap is True, the number of samples to draw from X to train each base estimator.

- If None (default), then draw `X.shape[0]` samples.
- If int, then draw `max_samples` samples.
- If float, then draw `max_samples * X.shape[0]` samples. Thus, `max_samples` should be in the interval `(0, 1)`.

New in version 0.22.

Attributes

base_estimator_ [DecisionTreeClassifier] The child estimator template used to create the collection of fitted sub-estimators.

estimators_ [list of DecisionTreeClassifier] The collection of fitted sub-estimators.

classes_ [array of shape (n_classes,) or a list of such arrays] The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

n_classes_ [int or list] The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).

n_features_ [int] The number of features when `fit` is performed.

n_outputs_ [int] The number of outputs when `fit` is performed.

feature_importances_ [ndarray of shape (n_features,)] Return the feature importances (the higher, the more important the feature).

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate. This attribute exists only when `oob_score` is True.

oob_decision_function_ [array of shape (n_samples, n_classes)] Decision function computed with out-of-bag estimate on the training set. If n_estimators is small it might be possible that a data point was never left out during the bootstrap. In this case, `oob_decision_function_` might contain NaN. This attribute exists only when `oob_score` is True.

See also:

DecisionTreeClassifier, *ExtraTreesClassifier*

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data, `max_features=n_features` and `bootstrap=False`, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed.

References

[R45f14345c000-1]

Examples

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.datasets import make_classification
```

```
>>> X, y = make_classification(n_samples=1000, n_features=4,
...                            n_informative=2, n_redundant=0,
...                            random_state=0, shuffle=False)
>>> clf = RandomForestClassifier(max_depth=2, random_state=0)
>>> clf.fit(X, y)
RandomForestClassifier(max_depth=2, random_state=0)
>>> print(clf.feature_importances_)
[0.14205973 0.76664038 0.0282433  0.06305659]
>>> print(clf.predict([[0, 0, 0, 0]]))
[1]
```

Methods

apply(self, X)	Apply trees in the forest to X, return leaf indices.
decision_path(self, X)	Return the decision path in the forest.
fit(self, X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class for X.
predict_log_proba(self, X)	Predict class log-probabilities for X.
predict_proba(self, X)	Predict class probabilities for X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_estimators=100*, *criterion='gini'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *bootstrap=True*, *oob_score=False*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*, *class_weight=None*, *ccp_alpha=0.0*, *max_samples=None*)

Initialize self. See help(type(self)) for accurate signature.

apply(*self*, *X*)
Apply trees in the forest to X, return leaf indices.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

X_leaves [array_like, shape = [n_samples, n_estimators]] For each datapoint x in X and for each tree in the forest, return the index of the leaf x ends up in.

decision_path(*self*, *X*)
Return the decision path in the forest.

New in version 0.18.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

indicator [sparse csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

n_nodes_ptr [array of size (n_estimators + 1,)] The columns from indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]] gives the indicator value for the i-th estimator.

property feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [array, shape = [n_features]] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

fit(*self*, *X*, *y*, *sample_weight=None*)
Build a forest of trees from the training set (X, y).

Parameters

X [array-like or sparse matrix of shape (n_samples, n_features)] The training input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels in classification, real numbers in regression).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict class for X.

The predicted class of an input sample is a vote by the trees in the forest, weighted by their probability estimates. That is, the predicted class is the one with highest mean probability estimate across the trees.

Parameters

X [array-like or sparse matrix of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The predicted classes.

predict_log_proba(*self*, *X*)
Predict class log-probabilities for X.

The predicted class log-probabilities of an input sample is computed as the log of the mean predicted class probabilities of the trees in the forest.

Parameters

X [array-like or sparse matrix of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

p [array of shape (n_samples, n_classes), or a list of n_outputs] such arrays if n_outputs > 1. The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

predict_proba(*self*, *X*)
Predict class probabilities for X.

The predicted class probabilities of an input sample are computed as the mean predicted class probabilities of the trees in the forest. The class probability of a single tree is the fraction of samples of the same class in a leaf.

Parameters

X [array-like or sparse matrix of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

p [array of shape (n_samples, n_classes), or a list of n_outputs] such arrays if n_outputs > 1. The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.ensemble.RandomForestClassifier`

- *ROC Curve with Visualization API*
- *Comparison of Calibration of Classifiers*
- *Probability Calibration for 3-class classification*
- *Classifier comparison*
- *Inductive Clustering*
- *Plot class probabilities calculated by the VotingClassifier*
- *OOB Errors for Random Forests*
- *Feature transformations with ensembles of trees*
- *Plot the decision surfaces of ensembles of trees on the iris dataset*
- *Permutation Importance with Multicollinear or Correlated Features*
- *Permutation Importance vs Random Forest Feature Importance (MDI)*
- *Release Highlights for scikit-learn 0.22*
- *Classification of text documents using sparse features*

sklearn.ensemble.RandomForestRegressor

class sklearn.ensemble.**RandomForestRegressor**(*n_estimators=100*, *criterion='mse'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *bootstrap=True*, *oob_score=False*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*, *ccp_alpha=0.0*, *max_samples=None*)

A random forest regressor.

A random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if `bootstrap=True` (default).

Read more in the *User Guide*.

Parameters

n_estimators [integer, optional (default=10)] The number of trees in the forest.

Changed in version 0.22: The default value of `n_estimators` changed from 10 to 100 in 0.22.

criterion [string, optional (default="mse")] The function to measure the quality of a split. Supported criteria are "mse" for the mean squared error, which is equal to variance reduction as feature selection criterion, and "mae" for the mean absolute error.

New in version 0.18: Mean Absolute Error (MAE) criterion.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features [int, float, string or None, optional (default="auto")] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=n_features`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

max_leaf_nodes [int or None, optional (default=None)] Grow trees with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

bootstrap [boolean, optional (default=True)] Whether bootstrap samples are used when building trees. If False, the whole datset is used to build each tree.

oob_score [bool, optional (default=False)] whether to use out-of-bag samples to estimate the R^2 on unseen data.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel. *fit*, *predict*, *decision_path* and *apply* are all parallelized over the trees. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional (default=None)] Controls both the randomness of the bootstrapping of the samples used when building trees (if

bootstrap=True) and the sampling of the features to consider when looking for the best split at each node (if max_features < n_features). See *Glossary* for details.

verbose [int, optional (default=0)] Controls the verbosity when fitting and predicting.

warm_start [bool, optional (default=False)] When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See *the Glossary*.

ccp_alpha [non-negative float, optional (default=0.0)] Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp_alpha will be chosen. By default, no pruning is performed. See *Minimal Cost-Complexity Pruning* for details.

New in version 0.22.

max_samples [int or float, default=None] If bootstrap is True, the number of samples to draw from X to train each base estimator.

- If None (default), then draw X.shape[0] samples.
- If int, then draw max_samples samples.
- If float, then draw max_samples * X.shape[0] samples. Thus, max_samples should be in the interval (0, 1).

New in version 0.22.

Attributes

base_estimator_ [DecisionTreeRegressor] The child estimator template used to create the collection of fitted sub-estimators.

estimators_ [list of DecisionTreeRegressor] The collection of fitted sub-estimators.

feature_importances_ [ndarray of shape (n_features,)] Return the feature importances (the higher, the more important the feature).

n_features_ [int] The number of features when fit is performed.

n_outputs_ [int] The number of outputs when fit is performed.

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate. This attribute exists only when oob_score is True.

oob_prediction_ [ndarray of shape (n_samples,)] Prediction computed with out-of-bag estimate on the training set. This attribute exists only when oob_score is True.

See also:

DecisionTreeRegressor, *ExtraTreesRegressor*

Notes

The default values for the parameters controlling the size of the trees (e.g. max_depth, min_samples_leaf, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data, max_features=n_features and bootstrap=False, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, random_state has to be fixed.

The default value `max_features="auto"` uses `n_features` rather than `n_features / 3`. The latter was originally suggested in [1], whereas the former was more recently justified empirically in [2].

References

[Rf91cab2dc427-1], [Rf91cab2dc427-2]

Examples

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> from sklearn.datasets import make_regression
```

```
>>> X, y = make_regression(n_features=4, n_informative=2,
...                        random_state=0, shuffle=False)
>>> regr = RandomForestRegressor(max_depth=2, random_state=0)
>>> regr.fit(X, y)
RandomForestRegressor(max_depth=2, random_state=0)
>>> print(regr.feature_importances_)
[0.18146984 0.81473937 0.00145312 0.00233767]
>>> print(regr.predict([[0, 0, 0, 0]]))
[-8.32987858]
```

Methods

apply(self, X)	Apply trees in the forest to X, return leaf indices.
decision_path(self, X)	Return the decision path in the forest.
fit(self, X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict regression target for X.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_estimators=100*, *criterion='mse'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *bootstrap=True*, *oob_score=False*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*, *ccp_alpha=0.0*, *max_samples=None*)

Initialize self. See help(type(self)) for accurate signature.

apply(*self*, *X*)

Apply trees in the forest to X, return leaf indices.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

X_leaves [array_like, shape = [n_samples, n_estimators]] For each datapoint x in X and for each tree in the forest, return the index of the leaf x ends up in.

decision_path (*self*, *X*)
Return the decision path in the forest.

New in version 0.18.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

indicator [sparse csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

n_nodes_ptr [array of size (n_estimators + 1,)] The columns from indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]] gives the indicator value for the i-th estimator.

property feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [array, shape = [n_features]] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

fit (*self*, *X*, *y*, *sample_weight=None*)
Build a forest of trees from the training set (X, y).

Parameters

X [array-like or sparse matrix of shape (n_samples, n_features)] The training input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels in classification, real numbers in regression).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict regression target for X.

The predicted regression target of an input sample is computed as the mean predicted regression targets of the trees in the forest.

Parameters

X [array-like or sparse matrix of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.ensemble.RandomForestRegressor`

- *Plot individual and voting regression predictions*
- *Comparing random forests and the multi-output meta estimator*
- *Combine predictors using stacking*
- *Prediction Latency*
- *Imputing missing values with variants of IterativeImputer*
- *Imputing missing values before building an estimator*

`sklearn.ensemble.ExtraTreesClassifier`

class sklearn.ensemble.**ExtraTreesClassifier**(*n_estimators=100*, *criterion='gini'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *bootstrap=False*, *oob_score=False*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*, *class_weight=None*, *ccp_alpha=0.0*, *max_samples=None*)

An extra-trees classifier.

This class implements a meta estimator that fits a number of randomized decision trees (a.k.a. extra-trees) on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

Read more in the *User Guide*.

Parameters

n_estimators [integer, optional (default=10)] The number of trees in the forest.

Changed in version 0.22: The default value of `n_estimators` changed from 10 to 100 in 0.22.

criterion [string, optional (default="gini")] The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least

min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider min_samples_leaf as the minimum number.
- If float, then min_samples_leaf is a fraction and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features [int, float, string or None, optional (default="auto")] The number of features to consider when looking for the best split:

- If int, then consider max_features features at each split.
- If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
- If "auto", then max_features=sqrt(n_features).
- If "sqrt", then max_features=sqrt(n_features).
- If "log2", then max_features=log2(n_features).
- If None, then max_features=n_features.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.

max_leaf_nodes [int or None, optional (default=None)] Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where N is the total number of samples, N_t is the number of samples at the current node, N_t_L is the number of samples in the left child, and N_t_R is the number of samples in the right child.

N, N_t, N_t_R and N_t_L all refer to the weighted sum, if sample_weight is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: min_impurity_split has been deprecated in favor of min_impurity_decrease in 0.19. The default value of min_impurity_split will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use min_impurity_decrease instead.

bootstrap [boolean, optional (default=False)] Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

oob_score [bool, optional (default=False)] Whether to use out-of-bag samples to estimate the generalization accuracy.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel. *fit*, *predict*, *decision_path* and *apply* are all parallelized over the trees. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional (default=None)] Controls 3 sources of randomness:

- the bootstrapping of the samples used when building trees (if `bootstrap=True`)
- the sampling of the features to consider when looking for the best split at each node (if `max_features < n_features`)
- the draw of the splits for each of the `max_features`

See *Glossary* for details.

verbose [int, optional (default=0)] Controls the verbosity when fitting and predicting.

warm_start [bool, optional (default=False)] When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See *the Glossary*.

class_weight [dict, list of dicts, "balanced", "balanced_subsample" or None, optional (default=None)] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

The "balanced_subsample" mode is the same as "balanced" except that weights are computed based on the bootstrap sample for every tree grown.

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

ccp_alpha [non-negative float, optional (default=0.0)] Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than `ccp_alpha` will be chosen. By default, no pruning is performed. See *Minimal Cost-Complexity Pruning* for details.

New in version 0.22.

max_samples [int or float, default=None] If bootstrap is True, the number of samples to draw from X to train each base estimator.

- If None (default), then draw `X.shape[0]` samples.
- If int, then draw `max_samples` samples.
- If float, then draw `max_samples * X.shape[0]` samples. Thus, `max_samples` should be in the interval `(0, 1)`.

New in version 0.22.

Attributes

base_estimator_ [ExtraTreeClassifier] The child estimator template used to create the collection of fitted sub-estimators.

estimators_ [list of DecisionTreeClassifier] The collection of fitted sub-estimators.

classes_ [array of shape (n_classes,) or a list of such arrays] The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

n_classes_ [int or list] The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).

feature_importances_ [ndarray of shape (n_features,)] Return the feature importances (the higher, the more important the feature).

n_features_ [int] The number of features when `fit` is performed.

n_outputs_ [int] The number of outputs when `fit` is performed.

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate. This attribute exists only when `oob_score` is True.

oob_decision_function_ [array of shape (n_samples, n_classes)] Decision function computed with out-of-bag estimate on the training set. If n_estimators is small it might be possible that a data point was never left out during the bootstrap. In this case, `oob_decision_function_` might contain NaN. This attribute exists only when `oob_score` is True.

See also:

sklearn.tree.ExtraTreeClassifier Base classifier for this ensemble.

RandomForestClassifier Ensemble Classifier based on trees with optimal splits.

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

References

[Rc8f28bfad63f-1]

Examples

```
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_features=4, random_state=0)
>>> clf = ExtraTreesClassifier(n_estimators=100, random_state=0)
>>> clf.fit(X, y)
ExtraTreesClassifier(random_state=0)
>>> clf.predict([[0, 0, 0, 0]])
array([1])
```

Methods

apply(self, X)	Apply trees in the forest to X, return leaf indices.
decision_path(self, X)	Return the decision path in the forest.
fit(self, X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class for X.
predict_log_proba(self, X)	Predict class log-probabilities for X.
predict_proba(self, X)	Predict class probabilities for X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_estimators=100*, *criterion='gini'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *bootstrap=False*, *oob_score=False*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*, *class_weight=None*, *ccp_alpha=0.0*, *max_samples=None*)

Initialize self. See help(type(self)) for accurate signature.

apply(*self*, *X*)

Apply trees in the forest to X, return leaf indices.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

X_leaves [array_like, shape = [n_samples, n_estimators]] For each datapoint x in X and for each tree in the forest, return the index of the leaf x ends up in.

decision_path(*self*, *X*)

Return the decision path in the forest.

New in version 0.18.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

indicator [sparse csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

n_nodes_ptr [array of size (n_estimators + 1,)] The columns from indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]] gives the indicator value for the i-th estimator.

property feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [array, shape = [n_features]] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

fit (*self*, *X*, *y*, *sample_weight=None*)

Build a forest of trees from the training set (X, y).

Parameters

X [array-like or sparse matrix of shape (n_samples, n_features)] The training input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels in classification, real numbers in regression).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

self [object]

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)

Predict class for X.

The predicted class of an input sample is a vote by the trees in the forest, weighted by their probability estimates. That is, the predicted class is the one with highest mean probability estimate across the trees.

Parameters

X [array-like or sparse matrix of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The predicted classes.

predict_log_proba (*self*, *X*)

Predict class log-probabilities for X.

The predicted class log-probabilities of an input sample is computed as the log of the mean predicted class probabilities of the trees in the forest.

Parameters

X [array-like or sparse matrix of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

p [array of shape (n_samples, n_classes), or a list of n_outputs] such arrays if n_outputs > 1. The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

predict_proba(*self*, *X*)

Predict class probabilities for X.

The predicted class probabilities of an input sample are computed as the mean predicted class probabilities of the trees in the forest. The class probability of a single tree is the fraction of samples of the same class in a leaf.

Parameters

X [array-like or sparse matrix of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

p [array of shape (n_samples, n_classes), or a list of n_outputs] such arrays if n_outputs > 1. The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.ensemble.ExtraTreesClassifier`

- *Pixel importances with a parallel forest of trees*
- *Feature importances with forests of trees*
- *Hashing feature transformation using Totally Random Trees*
- *Plot the decision surfaces of ensembles of trees on the iris dataset*

`sklearn.ensemble.ExtraTreesRegressor`

class sklearn.ensemble.**ExtraTreesRegressor**(*n_estimators=100*, *criterion='mse'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *bootstrap=False*, *oob_score=False*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*, *ccp_alpha=0.0*, *max_samples=None*)

An extra-trees regressor.

This class implements a meta estimator that fits a number of randomized decision trees (a.k.a. extra-trees) on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

Read more in the *User Guide*.

Parameters

n_estimators [integer, optional (default=10)] The number of trees in the forest.

Changed in version 0.22: The default value of `n_estimators` changed from 10 to 100 in 0.22.

criterion [string, optional (default="mse")] The function to measure the quality of a split. Supported criteria are "mse" for the mean squared error, which is equal to variance reduction as feature selection criterion, and "mae" for the mean absolute error.

New in version 0.18: Mean Absolute Error (MAE) criterion.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features [int, float, string or None, optional (default="auto")] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=n_features`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

max_leaf_nodes [int or None, optional (default=None)] Grow trees with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

bootstrap [boolean, optional (default=False)] Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

oob_score [bool, optional (default=False)] Whether to use out-of-bag samples to estimate the R^2 on unseen data.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel. *fit*, *predict*, *decision_path* and *apply* are all parallelized over the trees. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional (default=None)] Controls 3 sources of randomness:

- the bootstrapping of the samples used when building trees (if `bootstrap=True`)
- the sampling of the features to consider when looking for the best split at each node (if `max_features < n_features`)
- the draw of the splits for each of the `max_features`

See *Glossary* for details.

verbose [int, optional (default=0)] Controls the verbosity when fitting and predicting.

warm_start [bool, optional (default=False)] When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See *the Glossary*.

ccp_alpha [non-negative float, optional (default=0.0)] Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than `ccp_alpha` will be chosen. By default, no pruning is performed. See *Minimal Cost-Complexity Pruning* for details.

New in version 0.22.

max_samples [int or float, default=None] If bootstrap is True, the number of samples to draw from X to train each base estimator.

- If None (default), then draw `X.shape[0]` samples.
- If int, then draw `max_samples` samples.
- If float, then draw `max_samples * X.shape[0]` samples. Thus, `max_samples` should be in the interval `(0, 1)`.

New in version 0.22.

Attributes

base_estimator_ [ExtraTreeRegressor] The child estimator template used to create the collection of fitted sub-estimators.

estimators_ [list of DecisionTreeRegressor] The collection of fitted sub-estimators.

feature_importances_ [ndarray of shape (n_features,)] Return the feature importances (the higher, the more important the feature).

n_features_ [int] The number of features.

n_outputs_ [int] The number of outputs.

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate. This attribute exists only when `oob_score` is True.

oob_prediction_ [ndarray of shape (n_samples,)] Prediction computed with out-of-bag estimate on the training set. This attribute exists only when `oob_score` is True.

See also:

sklearn.tree.ExtraTreeRegressor Base estimator for this ensemble.

RandomForestRegressor Ensemble regressor using trees with optimal splits.

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

References

[Ra7d0c8995fbc-1]

Methods

apply(self, X)	Apply trees in the forest to X, return leaf indices.
decision_path(self, X)	Return the decision path in the forest.
fit(self, X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict regression target for X.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_estimators=100*, *criterion='mse'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *bootstrap=False*, *oob_score=False*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*, *ccp_alpha=0.0*, *max_samples=None*)

Initialize self. See help(type(self)) for accurate signature.

apply(*self*, *X*)

Apply trees in the forest to X, return leaf indices.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

X_leaves [array_like, shape = [n_samples, n_estimators]] For each datapoint x in X and for each tree in the forest, return the index of the leaf x ends up in.

decision_path(*self*, *X*)

Return the decision path in the forest.

New in version 0.18.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

indicator [sparse csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

n_nodes_ptr [array of size (n_estimators + 1,)] The columns from indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]] gives the indicator value for the i-th estimator.

property feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [array, shape = [n_features]] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

fit (*self*, *X*, *y*, *sample_weight=None*)

Build a forest of trees from the training set (X, y).

Parameters

X [array-like or sparse matrix of shape (n_samples, n_features)] The training input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels in classification, real numbers in regression).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

self [object]

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)

Predict regression target for X.

The predicted regression target of an input sample is computed as the mean predicted regression targets of the trees in the forest.

Parameters

X [array-like or sparse matrix of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.ensemble.ExtraTreesRegressor`

- *Face completion with a multi-output estimators*
- *Imputing missing values with variants of IterativeImputer*

sklearn.ensemble.GradientBoostingClassifier

class sklearn.ensemble.**GradientBoostingClassifier**(*loss='deviance'*, *learning_rate=0.1*, *n_estimators=100*, *subsample=1.0*, *criterion='friedman_mse'*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_depth=3*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *init=None*, *random_state=None*, *max_features=None*, *verbose=0*, *max_leaf_nodes=None*, *warm_start=False*, *presort='deprecated'*, *validation_fraction=0.1*, *n_iter_no_change=None*, *tol=0.0001*, *ccp_alpha=0.0*)

Gradient Boosting for classification.

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage `n_classes_` regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced.

Read more in the *User Guide*.

Parameters

loss [{'deviance', 'exponential'}, optional (default='deviance')] loss function to be optimized. 'deviance' refers to deviance (= logistic regression) for classification with probabilistic outputs. For loss 'exponential' gradient boosting recovers the AdaBoost algorithm.

learning_rate [float, optional (default=0.1)] learning rate shrinks the contribution of each tree by `learning_rate`. There is a trade-off between learning_rate and n_estimators.

n_estimators [int (default=100)] The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance.

subsample [float, optional (default=1.0)] The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting. `subsample` interacts with the parameter `n_estimators`. Choosing `subsample < 1.0` leads to a reduction of variance and an increase in bias.

criterion [string, optional (default="friedman_mse")] The function to measure the quality of a split. Supported criteria are "friedman_mse" for the mean squared error with improvement score by Friedman, "mse" for mean squared error, and "mae" for the mean absolute error. The default value of "friedman_mse" is generally the best as it can provide a better approximation in some cases.

New in version 0.18.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.

- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_depth [integer, optional (default=3)] maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. Tune this parameter for best performance; the best value depends on the interaction of the input variables.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

init [estimator or 'zero', optional (default=None)] An estimator object that is used to compute the initial predictions. `init` has to provide *fit* and *predict_proba*. If 'zero', the initial raw predictions are set to zero. By default, a `DummyEstimator` predicting the classes priors is used.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

max_features [int, float, string or None, optional (default=None)] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.

- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=sqrt(n_features)`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Choosing `max_features < n_features` leads to a reduction of variance and an increase in bias.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

verbose [int, default: 0] Enable verbose output. If 1 then it prints progress and performance once in a while (the more trees the lower the frequency). If greater than 1 then it prints progress and performance for every tree.

max_leaf_nodes [int or None, optional (default=None)] Grow trees with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

warm_start [bool, default: False] When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just erase the previous solution. See *the Glossary*.

presort [deprecated, default='deprecated'] This parameter is deprecated and will be removed in v0.24.

Deprecated since version 0.22.

validation_fraction [float, optional, default 0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if `n_iter_no_change` is set to an integer.

New in version 0.20.

n_iter_no_change [int, default None] `n_iter_no_change` is used to decide if early stopping will be used to terminate training when validation score is not improving. By default it is set to None to disable early stopping. If set to a number, it will set aside `validation_fraction` size of the training data as validation and terminate training when validation score is not improving in all of the previous `n_iter_no_change` numbers of iterations. The split is stratified.

New in version 0.20.

tol [float, optional, default 1e-4] Tolerance for the early stopping. When the loss is not improving by at least tol for `n_iter_no_change` iterations (if set to a number), the training stops.

New in version 0.20.

ccp_alpha [non-negative float, optional (default=0.0)] Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than `ccp_alpha` will be chosen. By default, no pruning is performed. See *Minimal Cost-Complexity Pruning* for details.

New in version 0.22.

Attributes

n_estimators_ [int] The number of estimators as selected by early stopping (if `n_iter_no_change` is specified). Otherwise it is set to `n_estimators`.

New in version 0.20.

feature_importances_ [array, shape (n_features,)] Return the feature importances (the higher, the more important the feature).

oob_improvement_ [array, shape (n_estimators,)] The improvement in loss (= deviance) on the out-of-bag samples relative to the previous iteration. `oob_improvement_[0]` is the improvement in loss of the first stage over the `init` estimator. Only available if `subsample < 1.0`

train_score_ [array, shape (n_estimators,)] The i-th score `train_score_[i]` is the deviance (= loss) of the model at iteration `i` on the in-bag sample. If `subsample == 1` this is the deviance on the training data.

loss_ [LossFunction] The concrete `LossFunction` object.

init_ [estimator] The estimator that provides the initial predictions. Set via the `init` argument or `loss.init_estimator`.

estimators_ [ndarray of DecisionTreeRegressor,shape (n_estimators, `loss_.K`)] The collection of fitted sub-estimators. `loss_.K` is 1 for binary classification, otherwise n_classes.

classes_ [array of shape (n_classes,)] The classes labels.

See also:

sklearn.ensemble.HistGradientBoostingClassifier

sklearn.tree.DecisionTreeClassifier, ***RandomForestClassifier***

AdaBoostClassifier

Notes

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data and `max_features=n_features`, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed.

References

J. Friedman, Greedy Function Approximation: A Gradient Boosting Machine, The Annals of Statistics, Vol. 29, No. 5, 2001.

J. Friedman, Stochastic Gradient Boosting, 1999

T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning Ed. 2, Springer, 2009.

Methods

apply(self, X)	Apply trees in the ensemble to X, return leaf indices.
decision_function(self, X)	Compute the decision function of `X`.
fit(self, X, y[, sample_weight, monitor])	Fit the gradient boosting model.

Continued on next page

Table 19 – continued from previous page

get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class for X.
predict_log_proba(self, X)	Predict class log-probabilities for X.
predict_proba(self, X)	Predict class probabilities for X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.
staged_decision_function(self, X)	Compute decision function of X for each iteration.
staged_predict(self, X)	Predict class at each stage for X.
staged_predict_proba(self, X)	Predict class probabilities at each stage for X.

__init__ (*self*, *loss='deviance'*, *learning_rate=0.1*, *n_estimators=100*, *subsample=1.0*, *criterion='friedman_mse'*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_depth=3*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *init=None*, *random_state=None*, *max_features=None*, *verbose=0*, *max_leaf_nodes=None*, *warm_start=False*, *presort='deprecated'*, *validation_fraction=0.1*, *n_iter_no_change=None*, *tol=0.0001*, *ccp_alpha=0.0*)

Initialize self. See help(type(self)) for accurate signature.

apply (*self*, *X*)

Apply trees in the ensemble to X, return leaf indices.

New in version 0.17.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted to a sparse `csr_matrix`.

Returns

X_leaves [array-like, shape (n_samples, n_estimators, n_classes)] For each datapoint x in X and for each tree in the ensemble, return the index of the leaf x ends up in each estimator. In the case of binary classification n_classes is 1.

decision_function (*self*, *X*)

Compute the decision function of `X`.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

score [array, shape (n_samples, n_classes) or (n_samples,)] The decision function of the input samples, which corresponds to the raw values predicted from the trees of the ensemble . The order of the classes corresponds to that in the attribute *classes_*. Regression and binary classification produce an array of shape [n_samples].

property feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [array, shape (n_features,)] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

fit (*self*, *X*, *y*, *sample_weight=None*, *monitor=None*)
Fit the gradient boosting model.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

y [array-like, shape (n_samples,)] Target values (strings or integers in classification, real numbers in regression) For classification, labels must correspond to classes.

sample_weight [array-like, shape (n_samples,) or None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

monitor [callable, optional] The monitor is called after each iteration with the current iteration, a reference to the estimator and the local variables of `_fit_stages` as keyword arguments `callable(i, self, locals())`. If the callable returns `True` the fitting procedure is stopped. The monitor can be used for various things such as computing held-out estimates, early stopping, model introspect, and snapshoting.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict class for X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

y [array, shape (n_samples,)] The predicted values.

predict_log_proba (*self*, *X*)
Predict class log-probabilities for X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

p [array, shape (n_samples, n_classes)] The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

Raises

AttributeError If the `loss` does not support probabilities.

predict_proba(*self*, *X*)

Predict class probabilities for X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

p [array, shape (n_samples, n_classes)] The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

Raises

AttributeError If the `loss` does not support probabilities.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

staged_decision_function(*self*, *X*)

Compute decision function of `X` for each iteration.

This method allows monitoring (i.e. determine error on testing set) after each stage.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

score [generator of array, shape (n_samples, k)] The decision function of the input samples, which corresponds to the raw values predicted from the trees of the ensemble . The classes corresponds to that in the attribute *classes_*. Regression and binary classification are special cases with `k == 1`, otherwise `k==n_classes`.

staged_predict (*self*, *X*)

Predict class at each stage for X.

This method allows monitoring (i.e. determine error on testing set) after each stage.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

y [generator of array of shape (n_samples,)] The predicted value of the input samples.

staged_predict_proba (*self*, *X*)

Predict class probabilities at each stage for X.

This method allows monitoring (i.e. determine error on testing set) after each stage.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

y [generator of array of shape (n_samples,)] The predicted value of the input samples.

Examples using `sklearn.ensemble.GradientBoostingClassifier`

- *Gradient Boosting regularization*
- *Early stopping of Gradient Boosting*
- *Feature transformations with ensembles of trees*
- *Gradient Boosting Out-of-Bag estimates*
- *Feature discretization*

sklearn.ensemble.GradientBoostingRegressor

class sklearn.ensemble.**GradientBoostingRegressor**(*loss='ls'*, *learning_rate=0.1*, *n_estimators=100*, *subsample=1.0*, *criterion='friedman_mse'*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_depth=3*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *init=None*, *random_state=None*, *max_features=None*, *alpha=0.9*, *verbose=0*, *max_leaf_nodes=None*, *warm_start=False*, *presort='deprecated'*, *validation_fraction=0.1*, *n_iter_no_change=None*, *tol=0.0001*, *ccp_alpha=0.0*)

Gradient Boosting for regression.

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage a regression tree is fit on the negative gradient of the given loss function.

Read more in the *User Guide*.

Parameters

loss [{'ls', 'lad', 'huber', 'quantile'}, optional (default='ls')] loss function to be optimized. 'ls' refers to least squares regression. 'lad' (least absolute deviation) is a highly robust loss function solely based on order information of the input variables. 'huber' is a combination of the two. 'quantile' allows quantile regression (use `alpha` to specify the quantile).

learning_rate [float, optional (default=0.1)] learning rate shrinks the contribution of each tree by `learning_rate`. There is a trade-off between learning_rate and n_estimators.

n_estimators [int (default=100)] The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance.

subsample [float, optional (default=1.0)] The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting. `subsample` interacts with the parameter `n_estimators`. Choosing `subsample < 1.0` leads to a reduction of variance and an increase in bias.

criterion [string, optional (default="friedman_mse")] The function to measure the quality of a split. Supported criteria are "friedman_mse" for the mean squared error with improvement score by Friedman, "mse" for mean squared error, and "mae" for the mean absolute error. The default value of "friedman_mse" is generally the best as it can provide a better approximation in some cases.

New in version 0.18.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_depth [integer, optional (default=3)] maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. Tune this parameter for best performance; the best value depends on the interaction of the input variables.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

init [estimator or 'zero', optional (default=None)] An estimator object that is used to compute the initial predictions. `init` has to provide *fit* and *predict*. If 'zero', the initial raw predictions are set to zero. By default a `DummyEstimator` is used, predicting either the average target value (for loss='ls'), or a quantile for the other losses.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

max_features [int, float, string or None, optional (default=None)] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.

- If "auto", then `max_features=n_features`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Choosing `max_features < n_features` leads to a reduction of variance and an increase in bias.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

alpha [float (default=0.9)] The alpha-quantile of the huber loss function and the quantile loss function. Only if `loss='huber'` or `loss='quantile'`.

verbose [int, default: 0] Enable verbose output. If 1 then it prints progress and performance once in a while (the more trees the lower the frequency). If greater than 1 then it prints progress and performance for every tree.

max_leaf_nodes [int or None, optional (default=None)] Grow trees with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

warm_start [bool, default: False] When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just erase the previous solution. See *the Glossary*.

presort [deprecated, default='deprecated'] This parameter is deprecated and will be removed in v0.24.

Deprecated since version 0.22.

validation_fraction [float, optional, default 0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if `n_iter_no_change` is set to an integer.

New in version 0.20.

n_iter_no_change [int, default None] `n_iter_no_change` is used to decide if early stopping will be used to terminate training when validation score is not improving. By default it is set to None to disable early stopping. If set to a number, it will set aside `validation_fraction` size of the training data as validation and terminate training when validation score is not improving in all of the previous `n_iter_no_change` numbers of iterations.

New in version 0.20.

tol [float, optional, default 1e-4] Tolerance for the early stopping. When the loss is not improving by at least tol for `n_iter_no_change` iterations (if set to a number), the training stops.

New in version 0.20.

ccp_alpha [non-negative float, optional (default=0.0)] Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than `ccp_alpha` will be chosen. By default, no pruning is performed. See *Minimal Cost-Complexity Pruning* for details.

New in version 0.22.

Attributes

*feature_importances_* [array, shape (n_features,)] Return the feature importances (the higher, the more important the feature).

oob_improvement_ [array, shape (n_estimators,)] The improvement in loss (= deviance) on the out-of-bag samples relative to the previous iteration. `oob_improvement_[0]` is the improvement in loss of the first stage over the `init` estimator. Only available if `subsample < 1.0`

train_score_ [array, shape (n_estimators,)] The i-th score `train_score_[i]` is the deviance (= loss) of the model at iteration `i` on the in-bag sample. If `subsample == 1` this is the deviance on the training data.

loss_ [LossFunction] The concrete `LossFunction` object.

init_ [estimator] The estimator that provides the initial predictions. Set via the `init` argument or `loss.init_estimator`.

estimators_ [array of DecisionTreeRegressor, shape (n_estimators, 1)] The collection of fitted sub-estimators.

See also:

sklearn.ensemble.HistGradientBoostingRegressor

sklearn.tree.DecisionTreeRegressor, *RandomForestRegressor*

Notes

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data and `max_features=n_features`, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed.

References

J. Friedman, Greedy Function Approximation: A Gradient Boosting Machine, The Annals of Statistics, Vol. 29, No. 5, 2001.

J. Friedman, Stochastic Gradient Boosting, 1999

T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning Ed. 2, Springer, 2009.

Methods

apply(self, X)	Apply trees in the ensemble to X, return leaf indices.
fit(self, X, y[, sample_weight, monitor])	Fit the gradient boosting model.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict regression target for X.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.
staged_predict(self, X)	Predict regression target at each stage for X.

__init__ (*self*, *loss='ls'*, *learning_rate=0.1*, *n_estimators=100*, *subsample=1.0*, *criterion='friedman_mse'*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_depth=3*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *init=None*, *random_state=None*, *max_features=None*, *alpha=0.9*, *verbose=0*, *max_leaf_nodes=None*, *warm_start=False*, *presort='deprecated'*, *validation_fraction=0.1*, *n_iter_no_change=None*, *tol=0.0001*, *ccp_alpha=0.0*)
Initialize self. See help(type(self)) for accurate signature.

apply (*self*, *X*)
Apply trees in the ensemble to X, return leaf indices.

New in version 0.17.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted to a sparse `csr_matrix`.

Returns

X_leaves [array-like, shape (n_samples, n_estimators)] For each datapoint x in X and for each tree in the ensemble, return the index of the leaf x ends up in each estimator.

property feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [array, shape (n_features,)] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

fit (*self*, *X*, *y*, *sample_weight=None*, *monitor=None*)
Fit the gradient boosting model.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

y [array-like, shape (n_samples,)] Target values (strings or integers in classification, real numbers in regression) For classification, labels must correspond to classes.

sample_weight [array-like, shape (n_samples,) or None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

monitor [callable, optional] The monitor is called after each iteration with the current iteration, a reference to the estimator and the local variables of `_fit_stages` as keyword arguments `callable(i, self, locals())`. If the callable returns `True` the fitting procedure is stopped. The monitor can be used for various things such as computing held-out estimates, early stopping, model introspect, and snapshoting.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict regression target for X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

y [array, shape (n_samples,)] The predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

staged_predict (*self*, *X*)

Predict regression target at each stage for X.

This method allows monitoring (i.e. determine error on testing set) after each stage.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

y [generator of array of shape (n_samples,)] The predicted value of the input samples.

Examples using `sklearn.ensemble.GradientBoostingRegressor`

- *Plot individual and voting regression predictions*
- *Prediction Intervals for Gradient Boosting Regression*
- *Gradient Boosting regression*
- *Model Complexity Influence*

4.3.3 Metrics and scoring: quantifying the quality of predictions

There are 3 different APIs for evaluating the quality of a model's predictions:

- **Estimator score method**: Estimators have a `score` method providing a default evaluation criterion for the problem they are designed to solve. This is not discussed on this page, but in each estimator's documentation.
- **Scoring parameter**: Model-evaluation tools using *cross-validation* (such as *model_selection.cross_val_score* and *model_selection.GridSearchCV*) rely on an internal *scoring* strategy. This is discussed in the section *The scoring parameter: defining model evaluation rules*.
- **Metric functions**: The `metrics` module implements functions assessing prediction error for specific purposes. These metrics are detailed in sections on *Classification metrics*, *Multilabel ranking metrics*, *Regression metrics* and *Clustering metrics*.

Finally, *Dummy estimators* are useful to get a baseline value of those metrics for random predictions.

See also:

For "pairwise" metrics, between *samples* and not estimators or predictions, see the *Pairwise metrics, Affinities and Kernels* section.

The `scoring` parameter: defining model evaluation rules

Model selection and evaluation using tools, such as *model_selection.GridSearchCV* and *model_selection.cross_val_score*, take a `scoring` parameter that controls what metric they apply to the estimators evaluated.

Common cases: predefined values

For the most common use cases, you can designate a scorer object with the `scoring` parameter; the table below shows all possible values. All scorer objects follow the convention that **higher return values are better than lower return values**. Thus metrics which measure the distance between the model and the data, like *metrics.mean_squared_error*, are available as neg_mean_squared_error which return the negated value of the metric.

Scoring	Function	Comment
Classification		
'accuracy'	*metrics.accuracy_score*	
'balanced_accuracy'	*metrics.balanced_accuracy_score*	
'average_precision'	*metrics.average_precision_score*	
'neg_brier_score'	*metrics.brier_score_loss*	
'f1'	*metrics.f1_score*	for binary targets
'f1_micro'	*metrics.f1_score*	micro-averaged
'f1_macro'	*metrics.f1_score*	macro-averaged
'f1_weighted'	*metrics.f1_score*	weighted average
'f1_samples'	*metrics.f1_score*	by multilabel sample
'neg_log_loss'	*metrics.log_loss*	requires `predict_proba` support
'precision' etc.	*metrics.precision_score*	suffixes apply as with 'f1'
'recall' etc.	*metrics.recall_score*	suffixes apply as with 'f1'
'jaccard' etc.	*metrics.jaccard_score*	suffixes apply as with 'f1'
'roc_auc'	*metrics.roc_auc_score*	
'roc_auc_ovr'	*metrics.roc_auc_score*	
'roc_auc_ovo'	*metrics.roc_auc_score*	
'roc_auc_ovr_weighted'	*metrics.roc_auc_score*	
'roc_auc_ovo_weighted'	*metrics.roc_auc_score*	
Clustering		
'adjusted_mutual_info_score'	*metrics.adjusted_mutual_info_score*	
'adjusted_rand_score'	*metrics.adjusted_rand_score*	
'completeness_score'	*metrics.completeness_score*	
'fowlkes_mallows_score'	*metrics.fowlkes_mallows_score*	
'homogeneity_score'	*metrics.homogeneity_score*	
'mutual_info_score'	*metrics.mutual_info_score*	
'normalized_mutual_info_score'	*metrics.normalized_mutual_info_score*	
'v_measure_score'	*metrics.v_measure_score*	
Regression		
'explained_variance'	*metrics.explained_variance_score*	
'max_error'	*metrics.max_error*	
'neg_mean_absolute_error'	*metrics.mean_absolute_error*	
'neg_mean_squared_error'	*metrics.mean_squared_error*	
'neg_root_mean_squared_error'	*metrics.mean_squared_error*	
'neg_mean_squared_log_error'	*metrics.mean_squared_log_error*	
'neg_median_absolute_error'	*metrics.median_absolute_error*	
'r2'	*metrics.r2_score*	
'neg_mean_poisson_deviance'	*metrics.mean_poisson_deviance*	
'neg_mean_gamma_deviance'	*metrics.mean_gamma_deviance*	

Usage examples:

```
>>> from sklearn import svm, datasets
```

(continues on next page)

(continued from previous page)

```
>>> from sklearn.model_selection import cross_val_score
>>> X, y = datasets.load_iris(return_X_y=True)
>>> clf = svm.SVC(random_state=0)
>>> cross_val_score(clf, X, y, cv=5, scoring='recall_macro')
array([0.96..., 0.96..., 0.96..., 0.93..., 1.        ])
>>> model = svm.SVC()
>>> cross_val_score(model, X, y, cv=5, scoring='wrong_choice')
Traceback (most recent call last):
ValueError: 'wrong_choice' is not a valid scoring value. Use sorted(sklearn.metrics.
↪SCORERS.keys()) to get valid options.
```

Note: The values listed by the ValueError exception correspond to the functions measuring prediction accuracy described in the following sections. The scorer objects for those functions are stored in the dictionary `sklearn.metrics.SCORERS`.

Defining your scoring strategy from metric functions

The module *sklearn.metrics* also exposes a set of simple functions measuring a prediction error given ground truth and prediction:

- functions ending with `_score` return a value to maximize, the higher the better.
- functions ending with `_error` or `_loss` return a value to minimize, the lower the better. When converting into a scorer object using *make_scorer*, set the `greater_is_better` parameter to False (True by default; see the parameter description below).

Metrics available for various machine learning tasks are detailed in sections below.

Many metrics are not given names to be used as `scoring` values, sometimes because they require additional parameters, such as *fbeta_score*. In such cases, you need to generate an appropriate scoring object. The simplest way to generate a callable object for scoring is by using *make_scorer*. That function converts metrics into callables that can be used for model evaluation.

One typical use case is to wrap an existing metric function from the library with non-default values for its parameters, such as the `beta` parameter for the *fbeta_score* function:

```
>>> from sklearn.metrics import fbeta_score, make_scorer
>>> ftwo_scorer = make_scorer(fbeta_score, beta=2)
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.svm import LinearSVC
>>> grid = GridSearchCV(LinearSVC(), param_grid={'C': [1, 10]},
...                     scoring=ftwo_scorer, cv=5)
```

The second use case is to build a completely custom scorer object from a simple python function using *make_scorer*, which can take several parameters:

- the python function you want to use (`my_custom_loss_func` in the example below)
- whether the python function returns a score (`greater_is_better=True`, the default) or a loss (`greater_is_better=False`). If a loss, the output of the python function is negated by the scorer object, conforming to the cross validation convention that scorers return higher values for better models.
- for classification metrics only: whether the python function you provided requires continuous decision certainties (`needs_threshold=True`). The default value is False.
- any additional parameters, such as `beta` or `labels` in *f1_score*.

Here is an example of building custom scorers, and of using the `greater_is_better` parameter:

```
>>> import numpy as np
>>> def my_custom_loss_func(y_true, y_pred):
...     diff = np.abs(y_true - y_pred).max()
...     return np.log1p(diff)
...
>>> # score will negate the return value of my_custom_loss_func,
>>> # which will be np.log(2), 0.693, given the values for X
>>> # and y defined below.
>>> score = make_scorer(my_custom_loss_func, greater_is_better=False)
>>> X = [[1], [1]]
>>> y = [0, 1]
>>> from sklearn.dummy import DummyClassifier
>>> clf = DummyClassifier(strategy='most_frequent', random_state=0)
>>> clf = clf.fit(X, y)
>>> my_custom_loss_func(clf.predict(X), y)
0.69...
>>> score(clf, X, y)
-0.69...
```

Implementing your own scoring object

You can generate even more flexible model scorers by constructing your own scoring object from scratch, without using the *make_scorer* factory. For a callable to be a scorer, it needs to meet the protocol specified by the following two rules:

- It can be called with parameters `(estimator, X, y)`, where `estimator` is the model that should be evaluated, `X` is validation data, and `y` is the ground truth target for `X` (in the supervised case) or `None` (in the unsupervised case).
- It returns a floating point number that quantifies the `estimator` prediction quality on `X`, with reference to `y`. Again, by convention higher numbers are better, so if your scorer returns loss, that value should be negated.

Note: Using custom scorers in functions where n_jobs > 1

While defining the custom scoring function alongside the calling function should work out of the box with the default joblib backend (loky), importing it from another module will be a more robust approach and work independently of the joblib backend.

For example, to use `n_jobs` greater than 1 in the example below, `custom_scoring_function` function is saved in a user-created module (`custom_scorer_module.py`) and imported:

```
>>> from custom_scorer_module import custom_scoring_function
>>> cross_val_score(model,
...  X_train,
...  y_train,
...  scoring=make_scorer(custom_scoring_function, greater_is_better=False),
...  cv=5,
...  n_jobs=-1)
```

Using multiple metric evaluation

Scikit-learn also permits evaluation of multiple metrics in `GridSearchCV`, `RandomizedSearchCV` and `cross_validate`.

There are two ways to specify multiple scoring metrics for the `scoring` parameter:

- **As an iterable of string metrics::**

```
>>> scoring = ['accuracy', 'precision']
```

- **As a `dict` mapping the scorer name to the scoring function::**

```
>>> from sklearn.metrics import accuracy_score
>>> from sklearn.metrics import make_scorer
>>> scoring = {'accuracy': make_scorer(accuracy_score),
...            'prec': 'precision'}
```

Note that the dict values can either be scorer functions or one of the predefined metric strings.

Currently only those scorer functions that return a single score can be passed inside the dict. Scorer functions that return multiple values are not permitted and will require a wrapper to return a single metric:

```
>>> from sklearn.model_selection import cross_validate
>>> from sklearn.metrics import confusion_matrix
>>> # A sample toy binary classification dataset
>>> X, y = datasets.make_classification(n_classes=2, random_state=0)
>>> svm = LinearSVC(random_state=0)
>>> def tn(y_true, y_pred): return confusion_matrix(y_true, y_pred)[0, 0]
>>> def fp(y_true, y_pred): return confusion_matrix(y_true, y_pred)[0, 1]
>>> def fn(y_true, y_pred): return confusion_matrix(y_true, y_pred)[1, 0]
>>> def tp(y_true, y_pred): return confusion_matrix(y_true, y_pred)[1, 1]
>>> scoring = {'tp': make_scorer(tp), 'tn': make_scorer(tn),
...            'fp': make_scorer(fp), 'fn': make_scorer(fn)}
>>> cv_results = cross_validate(svm.fit(X, y), X, y, cv=5, scoring=scoring)
>>> # Getting the test set true positive scores
>>> print(cv_results['test_tp'])
[10  9  8  7  8]
>>> # Getting the test set false negative scores
>>> print(cv_results['test_fn'])
[0 1 2 3 2]
```

Classification metrics

The *sklearn.metrics* module implements several loss, score, and utility functions to measure classification performance. Some metrics might require probability estimates of the positive class, confidence values, or binary decisions values. Most implementations allow each sample to provide a weighted contribution to the overall score, through the `sample_weight` parameter.

Some of these are restricted to the binary classification case:

precision_recall_curve(y_true, probas_pred)	Compute precision-recall pairs for different probability thresholds
roc_curve(y_true, y_score[, pos_label, ...])	Compute Receiver operating characteristic (ROC)

Others also work in the multiclass case:

balanced_accuracy_score(y_true, y_pred[, ...])	Compute the balanced accuracy
cohen_kappa_score(y1, y2[, labels, weights, ...])	Cohen's kappa: a statistic that measures inter-annotator agreement.
confusion_matrix(y_true, y_pred[, labels, ...])	Compute confusion matrix to evaluate the accuracy of a classification.
hinge_loss(y_true, pred_decision[, labels, ...])	Average hinge loss (non-regularized)
matthews_corrcoef(y_true, y_pred[, ...])	Compute the Matthews correlation coefficient (MCC)
roc_auc_score(y_true, y_score[, average, ...])	Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.

Some also work in the multilabel case:

accuracy_score(y_true, y_pred[, normalize, ...])	Accuracy classification score.
classification_report(y_true, y_pred[, ...])	Build a text report showing the main classification metrics
f1_score(y_true, y_pred[, labels, ...])	Compute the F1 score, also known as balanced F-score or F-measure
fbeta_score(y_true, y_pred, beta[, labels, ...])	Compute the F-beta score
hamming_loss(y_true, y_pred[, labels, ...])	Compute the average Hamming loss.
jaccard_score(y_true, y_pred[, labels, ...])	Jaccard similarity coefficient score
log_loss(y_true, y_pred[, eps, normalize, ...])	Log loss, aka logistic loss or cross-entropy loss.
multilabel_confusion_matrix(y_true, y_pred)	Compute a confusion matrix for each class or sample
precision_recall_fscore_support(y_true, y_pred)	Compute precision, recall, F-measure and support for each class
precision_score(y_true, y_pred[, labels, ...])	Compute the precision
recall_score(y_true, y_pred[, labels, ...])	Compute the recall
roc_auc_score(y_true, y_score[, average, ...])	Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.
zero_one_loss(y_true, y_pred[, normalize, ...])	Zero-one classification loss.

And some work with binary and multilabel (but not multiclass) problems:

average_precision_score(y_true, y_score[, ...])	Compute average precision (AP) from prediction scores

In the following sub-sections, we will describe each of those functions, preceded by some notes on common API and metric definition.

From binary to multiclass and multilabel

Some metrics are essentially defined for binary classification tasks (e.g. f1_score, roc_auc_score). In these cases, by default only the positive label is evaluated, assuming by default that the positive class is labelled 1 (though this may be configurable through the pos_label parameter).

In extending a binary metric to multiclass or multilabel problems, the data is treated as a collection of binary problems, one for each class. There are then a number of ways to average binary metric calculations across the set of classes, each of which may be useful in some scenario. Where available, you should select among these using the average parameter.

- `"macro"` simply calculates the mean of the binary metrics, giving equal weight to each class. In problems where infrequent classes are nonetheless important, macro-averaging may be a means of highlighting their performance. On the other hand, the assumption that all classes are equally important is often untrue, such that macro-averaging will over-emphasize the typically low performance on an infrequent class.
- `"weighted"` accounts for class imbalance by computing the average of binary metrics in which each class's score is weighted by its presence in the true data sample.
- `"micro"` gives each sample-class pair an equal contribution to the overall metric (except as a result of sample-weight). Rather than summing the metric per class, this sums the dividends and divisors that make up the per-class metrics to calculate an overall quotient. Micro-averaging may be preferred in multilabel settings, including multiclass classification where a majority class is to be ignored.
- `"samples"` applies only to multilabel problems. It does not calculate a per-class measure, instead calculating the metric over the true and predicted classes for each sample in the evaluation data, and returning their (`sample_weight`-weighted) average.
- Selecting `average=None` will return an array with the score for each class.

While multiclass data is provided to the metric, like binary targets, as an array of class labels, multilabel data is specified as an indicator matrix, in which cell `[i, j]` has value 1 if sample `i` has label `j` and value 0 otherwise.

Accuracy score

The *accuracy_score* function computes the accuracy, either the fraction (default) or the count (normalize=False) of correct predictions.

In multilabel classification, the function returns the subset accuracy. If the entire set of predicted labels for a sample strictly match with the true set of labels, then the subset accuracy is 1.0; otherwise it is 0.0.

If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value, then the fraction of correct predictions over n_{samples} is defined as

$$\texttt{accuracy}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1(\hat{y}_i = y_i)$$

where $1(x)$ is the indicator function.

```
>>> import numpy as np
>>> from sklearn.metrics import accuracy_score
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]
>>> accuracy_score(y_true, y_pred)
0.5
>>> accuracy_score(y_true, y_pred, normalize=False)
2
```

In the multilabel case with binary label indicators:

```
>>> accuracy_score(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5
```

Example:

- See *Test with permutations the significance of a classification score* for an example of accuracy score usage using permutations of the dataset.

Balanced accuracy score

The `balanced_accuracy_score` function computes the balanced accuracy, which avoids inflated performance estimates on imbalanced datasets. It is the macro-average of recall scores per class or, equivalently, raw accuracy where each sample is weighted according to the inverse prevalence of its true class. Thus for balanced datasets, the score is equal to accuracy.

In the binary case, balanced accuracy is equal to the arithmetic mean of sensitivity (true positive rate) and specificity (true negative rate), or the area under the ROC curve with binary predictions rather than scores.

If the classifier performs equally well on either class, this term reduces to the conventional accuracy (i.e., the number of correct predictions divided by the total number of predictions).

In contrast, if the conventional accuracy is above chance only because the classifier takes advantage of an imbalanced test set, then the balanced accuracy, as appropriate, will drop to $\frac{1}{n\_classes}$.

The score ranges from 0 to 1, or when `adjusted=True` is used, it rescaled to the range $\frac{1}{1-n\_classes}$ to 1, inclusive, with performance at random scoring 0.

If y_i is the true value of the i-th sample, and w_i is the corresponding sample weight, then we adjust the sample weight to:

$$\hat{w}_i = \frac{w_i}{\sum_j 1(y_j = y_i) w_j}$$

where $1(x)$ is the indicator function. Given predicted $\hat{y}_i$ for sample i, balanced accuracy is defined as:

$$\texttt{balanced-accuracy}(y, \hat{y}, w) = \frac{1}{\sum \hat{w}_i} \sum_i 1(\hat{y}_i = y_i) \hat{w}_i$$

With `adjusted=True`, balanced accuracy reports the relative increase from $\texttt{balanced-accuracy}(y, \mathbf{0}, w) = \frac{1}{n\_classes}$. In the binary case, this is also known as *Youden's J statistic*, or *informedness*.

Note: The multiclass definition here seems the most reasonable extension of the metric used in binary classification, though there is no certain consensus in the literature:

- Our definition: [Mosley2013], [Kelleher2015] and [Guyon2015], where [Guyon2015] adopt the adjusted version to ensure that random predictions have a score of 0 and perfect predictions have a score of 1..
- Class balanced accuracy as described in [Mosley2013]: the minimum between the precision and the recall for each class is computed. Those values are then averaged over the total number of classes to get the balanced accuracy.
- Balanced Accuracy as described in [Urbanowicz2015]: the average of sensitivity and specificity is computed for each class and then averaged over total number of classes.

References:

Cohen's kappa

The function `cohen_kappa_score` computes Cohen's kappa statistic. This measure is intended to compare labelings by different human annotators, not a classifier versus a ground truth.

The kappa score (see docstring) is a number between -1 and 1. Scores above .8 are generally considered good agreement; zero or lower means no agreement (practically random labels).

Kappa scores can be computed for binary or multiclass problems, but not for multilabel problems (except by manually computing a per-label score) and not for more than two annotators.

```
>>> from sklearn.metrics import cohen_kappa_score
>>> y_true = [2, 0, 2, 2, 0, 1]
>>> y_pred = [0, 0, 2, 2, 0, 2]
>>> cohen_kappa_score(y_true, y_pred)
0.4285714285714286
```

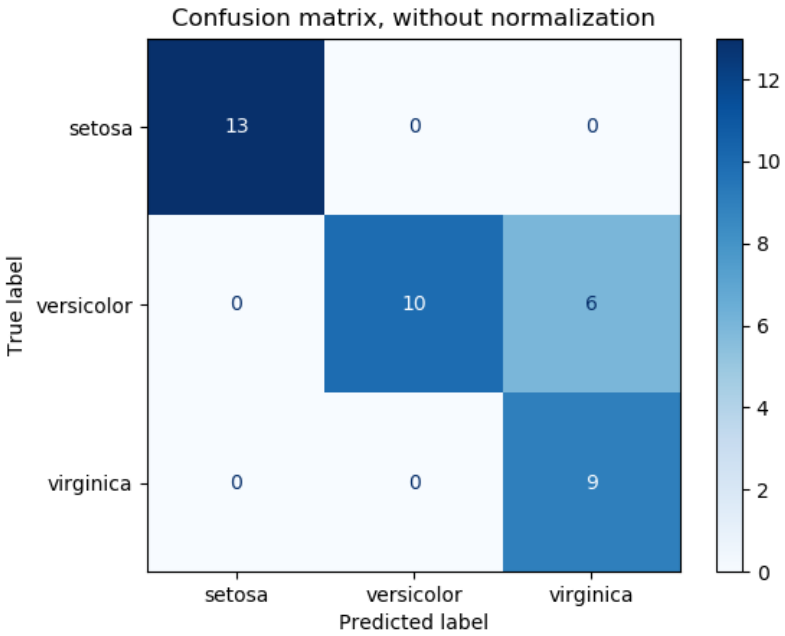
Confusion matrix

The *confusion_matrix* function evaluates classification accuracy by computing the confusion matrix with each row corresponding to the true class <https://en.wikipedia.org/wiki/Confusion_matrix>‘_. (Wikipedia and other references may use different convention for axes.)

By definition, entry i, j in a confusion matrix is the number of observations actually in group i, but predicted to be in group j. Here is an example:

```
>>> from sklearn.metrics import confusion_matrix
>>> y_true = [2, 0, 2, 2, 0, 1]
>>> y_pred = [0, 0, 2, 2, 0, 2]
>>> confusion_matrix(y_true, y_pred)
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])
```

plot_confusion_matrix can be used to visually represent a confusion matrix as shown in the *Confusion matrix* example, which creates the following figure:



The parameter `normalize` allows to report ratios instead of counts. The confusion matrix can be normalized in 3 different ways: `'pred'`, `'true'`, and `'all'` which will divide the counts by the sum of each columns, rows, or the entire matrix, respectively.

```
>>> y_true = [0, 0, 0, 1, 1, 1, 1, 1]
>>> y_pred = [0, 1, 0, 1, 0, 1, 0, 1]
>>> confusion_matrix(y_true, y_pred, normalize='all')
array([[0.25 , 0.125],
       [0.25 , 0.375]])
```

For binary problems, we can get counts of true negatives, false positives, false negatives and true positives as follows:

```
>>> y_true = [0, 0, 0, 1, 1, 1, 1, 1]
>>> y_pred = [0, 1, 0, 1, 0, 1, 0, 1]
>>> tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
>>> tn, fp, fn, tp
(2, 1, 2, 3)
```

Example:

- See *Confusion matrix* for an example of using a confusion matrix to evaluate classifier output quality.
- See *Recognizing hand-written digits* for an example of using a confusion matrix to classify hand-written digits.
- See *Classification of text documents using sparse features* for an example of using a confusion matrix to classify text documents.

Classification report

The `classification_report` function builds a text report showing the main classification metrics. Here is a small example with custom `target_names` and inferred labels:

```
>>> from sklearn.metrics import classification_report
>>> y_true = [0, 1, 2, 2, 0]
>>> y_pred = [0, 0, 2, 1, 0]
>>> target_names = ['class 0', 'class 1', 'class 2']
>>> print(classification_report(y_true, y_pred, target_names=target_names))
              precision    recall  f1-score   support

     class 0       0.67      1.00      0.80         2
     class 1       0.00      0.00      0.00         1
     class 2       1.00      0.50      0.67         2

    accuracy                           0.60         5
   macro avg       0.56      0.50      0.49         5
weighted avg       0.67      0.60      0.59         5
```

Example:

- See *Recognizing hand-written digits* for an example of classification report usage for hand-written digits.
- See *Classification of text documents using sparse features* for an example of classification report usage for text documents.
- See *Parameter estimation using grid search with cross-validation* for an example of classification report usage for grid search with nested cross-validation.

Hamming loss

The *hamming_loss* computes the average Hamming loss or Hamming distance between two sets of samples.

If $\hat{y}_j$ is the predicted value for the j-th label of a given sample, y_j is the corresponding true value, and n_{labels} is the number of classes or labels, then the Hamming loss $L_{Hamming}$ between two samples is defined as:

$$L_{Hamming}(y, \hat{y}) = \frac{1}{n_{\text{labels}}} \sum_{j=0}^{n_{\text{labels}}-1} 1(\hat{y}_j \neq y_j)$$

where $1(x)$ is the indicator function.

```
>>> from sklearn.metrics import hamming_loss
>>> y_pred = [1, 2, 3, 4]
>>> y_true = [2, 2, 3, 4]
>>> hamming_loss(y_true, y_pred)
0.25
```

In the multilabel case with binary label indicators:

```
>>> hamming_loss(np.array([[0, 1], [1, 1]]), np.zeros((2, 2)))
0.75
```

Note: In multiclass classification, the Hamming loss corresponds to the Hamming distance between `y_true` and `y_pred` which is similar to the *Zero one loss* function. However, while zero-one loss penalizes prediction sets that do not strictly match true sets, the Hamming loss penalizes individual labels. Thus the Hamming loss, upper bounded by the zero-one loss, is always between zero and one, inclusive; and predicting a proper subset or superset of the true labels will give a Hamming loss between zero and one, exclusive.

Precision, recall and F-measures

Intuitively, precision is the ability of the classifier not to label as positive a sample that is negative, and recall is the ability of the classifier to find all the positive samples.

The F-measure (F_β and F_1 measures) can be interpreted as a weighted harmonic mean of the precision and recall. A F_β measure reaches its best value at 1 and its worst score at 0. With $\beta = 1$, F_β and F_1 are equivalent, and the recall and the precision are equally important.

The *precision_recall_curve* computes a precision-recall curve from the ground truth label and a score given by the classifier by varying a decision threshold.

The *average_precision_score* function computes the average precision (AP) from prediction scores. The value is between 0 and 1 and higher is better. AP is defined as

$$\text{AP} = \sum_n (R_n - R_{n-1}) P_n$$

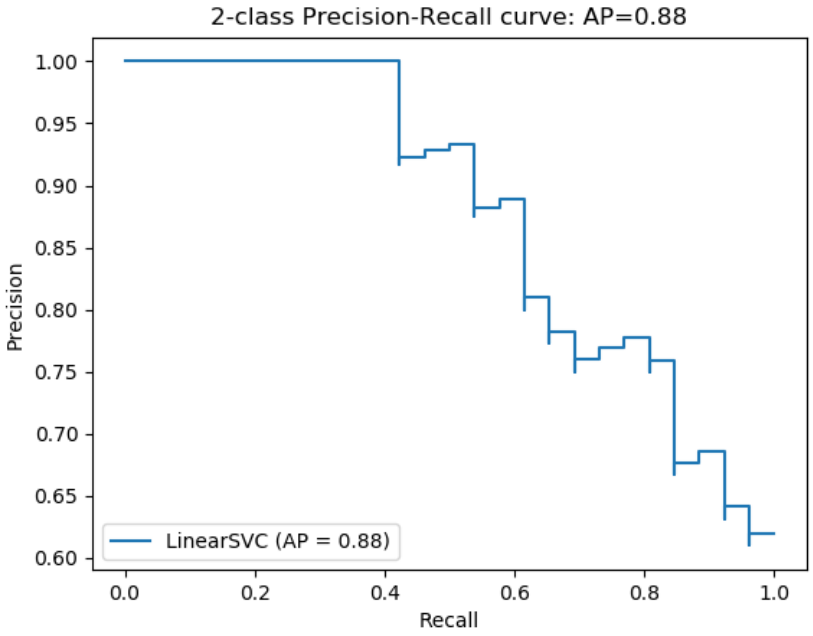
where P_n and R_n are the precision and recall at the nth threshold. With random predictions, the AP is the fraction of positive samples.

References [Manning2008] and [Everingham2010] present alternative variants of AP that interpolate the precision-recall curve. Currently, *average_precision_score* does not implement any interpolated variant. References [Davis2006] and [Flach2015] describe why a linear interpolation of points on the precision-recall curve provides an overly-optimistic measure of classifier performance. This linear interpolation is used when computing area under the curve with the trapezoidal rule in *auc*.

Several functions allow you to analyze the precision, recall and F-measures score:

average_precision_score(y_true, y_score[, ...])	Compute average precision (AP) from prediction scores
f1_score(y_true, y_pred[, labels, ...])	Compute the F1 score, also known as balanced F-score or F-measure
fbeta_score(y_true, y_pred, beta[, labels, ...])	Compute the F-beta score
precision_recall_curve(y_true, probas_pred)	Compute precision-recall pairs for different probability thresholds
precision_recall_fscore_support(y_true, y_pred)	Compute precision, recall, F-measure and support for each class
precision_score(y_true, y_pred[, labels, ...])	Compute the precision
recall_score(y_true, y_pred[, labels, ...])	Compute the recall

Note that the precision_recall_curve function is restricted to the binary case. The average_precision_score function works only in binary classification and multilabel indicator format. The plot_precision_recall_curve function plots the precision recall as follows.



Examples:

- See *Classification of text documents using sparse features* for an example of f1_score usage to classify text documents.
- See *Parameter estimation using grid search with cross-validation* for an example of precision_score and recall_score usage to estimate parameters using grid search with nested cross-validation.
- See *Precision-Recall* for an example of precision_recall_curve usage to evaluate classifier output quality.

References:

Binary classification

In a binary classification task, the terms ''positive'' and ''negative'' refer to the classifier's prediction, and the terms ''true'' and ''false'' refer to whether that prediction corresponds to the external judgment (sometimes known as the ''observation''). Given these definitions, we can formulate the following table:

	Actual class (observation)	
Predicted class (expectation)	tp (true positive) Correct result	fp (false positive) Unexpected result
	fn (false negative) Missing result	tn (true negative) Correct absence of result

In this context, we can define the notions of precision, recall and F-measure:

$$\text{precision} = \frac{tp}{tp + fp},$$

$$\text{recall} = \frac{tp}{tp + fn},$$

$$F_\beta = (1 + \beta^2)\frac{\text{precision} \times \text{recall}}{\beta^2 \text{precision} + \text{recall}}.$$

Here are some small examples in binary classification:

```
>>> from sklearn import metrics
>>> y_pred = [0, 1, 0, 0]
>>> y_true = [0, 1, 0, 1]
>>> metrics.precision_score(y_true, y_pred)
1.0
>>> metrics.recall_score(y_true, y_pred)
0.5
>>> metrics.f1_score(y_true, y_pred)
0.66...
>>> metrics.fbeta_score(y_true, y_pred, beta=0.5)
0.83...
>>> metrics.fbeta_score(y_true, y_pred, beta=1)
0.66...
>>> metrics.fbeta_score(y_true, y_pred, beta=2)
0.55...
>>> metrics.precision_recall_fscore_support(y_true, y_pred, beta=0.5)
(array([0.66..., 1.        ]), array([1. , 0.5]), array([0.71..., 0.83...]), array([2,
↪ 2]))


>>> import numpy as np
>>> from sklearn.metrics import precision_recall_curve
>>> from sklearn.metrics import average_precision_score
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> precision, recall, threshold = precision_recall_curve(y_true, y_scores)
>>> precision
array([0.66..., 0.5       , 1.        , 1.        ])
>>> recall
array([1. , 0.5, 0.5, 0. ])
```

(continues on next page)

(continued from previous page)

```
>>> threshold
array([0.35, 0.4 , 0.8 ])
>>> average_precision_score(y_true, y_scores)
0.83...
```

Multiclass and multilabel classification

In multiclass and multilabel classification task, the notions of precision, recall, and F-measures can be applied to each label independently. There are a few ways to combine results across labels, specified by the `average` argument to the *average_precision_score* (multilabel only), *f1_score*, *fbeta_score*, *precision_recall_fscore_support*, *precision_score* and *recall_score* functions, as described *above*. Note that if all labels are included, "micro"-averaging in a multiclass setting will produce precision, recall and F that are all identical to accuracy. Also note that "weighted" averaging may produce an F-score that is not between precision and recall.

To make this more explicit, consider the following notation:

- y the set of *predicted* $(sample, label)$ pairs
- $\hat{y}$ the set of *true* $(sample, label)$ pairs
- L the set of labels
- S the set of samples
- y_s the subset of y with sample s, i.e. $y_s := \{(s', l) \in y | s' = s\}$
- y_l the subset of y with label l
- similarly, $\hat{y}_s$ and $\hat{y}_l$ are subsets of $\hat{y}$
- $P(A, B) := \frac{|A \cap B|}{|A|}$ for some sets A and B
- $R(A, B) := \frac{|A \cap B|}{|B|}$ (Conventions vary on handling $B = \emptyset$; this implementation uses $R(A, B) := 0$, and similar for P.)
- $F_\beta(A, B) := \left(1 + \beta^2\right) \frac{P(A,B) \times R(A,B)}{\beta^2 P(A,B) + R(A,B)}$

Then the metrics are defined as:

average	Precision	Recall	F_beta
"micro"	$P(y, \hat{y})$	$R(y, \hat{y})$	$F_\beta(y, \hat{y})$
"samples"	$\frac{1}{\lvert S \rvert} \sum_{s \in S} P(y_s, \hat{y}_s)$	$\frac{1}{\lvert S \rvert} \sum_{s \in S} R(y_s, \hat{y}_s)$	$\frac{1}{\lvert S \rvert} \sum_{s \in S} F_\beta(y_s, \hat{y}_s)$
"macro"	$\frac{1}{\lvert L \rvert} \sum_{l \in L} P(y_l, \hat{y}_l)$	$\frac{1}{\lvert L \rvert} \sum_{l \in L} R(y_l, \hat{y}_l)$	$\frac{1}{\lvert L \rvert} \sum_{l \in L} F_\beta(y_l, \hat{y}_l)$
"weighted"	$\frac{1}{\sum_{l \in L} \lvert \hat{y}_l \rvert} \sum_{l \in L} \lvert \hat{y}_l \rvert P(y_l, \hat{y}_l)$	$\frac{1}{\sum_{l \in L} \lvert \hat{y}_l \rvert} \sum_{l \in L} \lvert \hat{y}_l \rvert R(y_l, \hat{y}_l)$	$\frac{1}{\sum_{l \in L} \lvert \hat{y}_l \rvert} \sum_{l \in L} \lvert \hat{y}_l \rvert F_\beta(y_l, \hat{y}_l)$
None	$\langle P(y_l, \hat{y}_l) \vert l \in L \rangle$	$\langle R(y_l, \hat{y}_l) \vert l \in L \rangle$	$\langle F_\beta(y_l, \hat{y}_l) \vert l \in L \rangle$

```
>>> from sklearn import metrics
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> metrics.precision_score(y_true, y_pred, average='macro')
0.22...
>>> metrics.recall_score(y_true, y_pred, average='micro')
0.33...
>>> metrics.f1_score(y_true, y_pred, average='weighted')
```

(continues on next page)

(continued from previous page)

```
0.26...
>>> metrics.fbeta_score(y_true, y_pred, average='macro', beta=0.5)
0.23...
>>> metrics.precision_recall_fscore_support(y_true, y_pred, beta=0.5, average=None)
(array([0.66..., 0.        , 0.        ]), array([1., 0., 0.]), array([0.71..., 0.
↪    , 0.        ]), array([2, 2, 2]...))
```

For multiclass classification with a "negative class", it is possible to exclude some labels:

```
>>> metrics.recall_score(y_true, y_pred, labels=[1, 2], average='micro')
... # excluding 0, no labels were correctly recalled
0.0
```

Similarly, labels not present in the data sample may be accounted for in macro-averaging.

```
>>> metrics.precision_score(y_true, y_pred, labels=[0, 1, 2, 3], average='macro')
0.166...
```

Jaccard similarity coefficient score

The *jaccard_score* function computes the average of Jaccard similarity coefficients, also called the Jaccard index, between pairs of label sets.

The Jaccard similarity coefficient of the i-th samples, with a ground truth label set y_i and predicted label set $\hat{y}_i$, is defined as

$$J(y_i, \hat{y}_i) = \frac{|y_i \cap \hat{y}_i|}{|y_i \cup \hat{y}_i|}.$$

jaccard_score works like *precision_recall_fscore_support* as a naively set-wise measure applying natively to binary targets, and extended to apply to multilabel and multiclass through the use of `average` (see *above*).

In the binary case:

```
>>> import numpy as np
>>> from sklearn.metrics import jaccard_score
>>> y_true = np.array([[0, 1, 1],
...                    [1, 1, 0]])
>>> y_pred = np.array([[1, 1, 1],
...                    [1, 0, 0]])
>>> jaccard_score(y_true[0], y_pred[0])
0.6666...
```

In the multilabel case with binary label indicators:

```
>>> jaccard_score(y_true, y_pred, average='samples')
0.5833...
>>> jaccard_score(y_true, y_pred, average='macro')
0.6666...
>>> jaccard_score(y_true, y_pred, average=None)
array([0.5, 0.5, 1. ])
```

Multiclass problems are binarized and treated like the corresponding multilabel problem:

```
>>> y_pred = [0, 2, 1, 2]
>>> y_true = [0, 1, 2, 2]
>>> jaccard_score(y_true, y_pred, average=None)
array([1. , 0. , 0.33...])
>>> jaccard_score(y_true, y_pred, average='macro')
0.44...
>>> jaccard_score(y_true, y_pred, average='micro')
0.33...
```

Hinge loss

The *hinge_loss* function computes the average distance between the model and the data using hinge loss, a one-sided metric that considers only prediction errors. (Hinge loss is used in maximal margin classifiers such as support vector machines.)

If the labels are encoded with +1 and -1, y: is the true value, and w is the predicted decisions as output by `decision_function`, then the hinge loss is defined as:

$$L_{\text{Hinge}}(y, w) = \max\{1 - wy, 0\} = |1 - wy|_+$$

If there are more than two labels, *hinge_loss* uses a multiclass variant due to Crammer & Singer. Here is the paper describing it.

If y_w is the predicted decision for true label and y_t is the maximum of the predicted decisions for all other labels, where predicted decisions are output by decision function, then multiclass hinge loss is defined by:

$$L_{\text{Hinge}}(y_w, y_t) = \max\{1 + y_t - y_w, 0\}$$

Here a small example demonstrating the use of the *hinge_loss* function with a svm classifier in a binary class problem:

```
>>> from sklearn import svm
>>> from sklearn.metrics import hinge_loss
>>> X = [[0], [1]]
>>> y = [-1, 1]
>>> est = svm.LinearSVC(random_state=0)
>>> est.fit(X, y)
LinearSVC(random_state=0)
>>> pred_decision = est.decision_function([[-2], [3], [0.5]])
>>> pred_decision
array([-2.18...,  2.36...,  0.09...])
>>> hinge_loss([-1, 1, 1], pred_decision)
0.3...
```

Here is an example demonstrating the use of the *hinge_loss* function with a svm classifier in a multiclass problem:

```
>>> X = np.array([[0], [1], [2], [3]])
>>> Y = np.array([0, 1, 2, 3])
>>> labels = np.array([0, 1, 2, 3])
>>> est = svm.LinearSVC()
>>> est.fit(X, Y)
LinearSVC()
>>> pred_decision = est.decision_function([[-1], [2], [3]])
>>> y_true = [0, 2, 3]
>>> hinge_loss(y_true, pred_decision, labels)
0.56...
```

Log loss

Log loss, also called logistic regression loss or cross-entropy loss, is defined on probability estimates. It is commonly used in (multinomial) logistic regression and neural networks, as well as in some variants of expectation-maximization, and can be used to evaluate the probability outputs (`predict_proba`) of a classifier instead of its discrete predictions.

For binary classification with a true label $y \in \{0, 1\}$ and a probability estimate $p = \Pr(y = 1)$, the log loss per sample is the negative log-likelihood of the classifier given the true label:

$$L_{\log}(y, p) = -\log \Pr(y|p) = -(y \log(p) + (1 - y) \log(1 - p))$$

This extends to the multiclass case as follows. Let the true labels for a set of samples be encoded as a 1-of-K binary indicator matrix Y, i.e., $y_{i,k} = 1$ if sample i has label k taken from a set of K labels. Let P be a matrix of probability estimates, with $p_{i,k} = \Pr(t_{i,k} = 1)$. Then the log loss of the whole set is

$$L_{\log}(Y, P) = -\log \Pr(Y|P) = -\frac{1}{N} \sum_{i=0}^{N-1} \sum_{k=0}^{K-1} y_{i,k} \log p_{i,k}$$

To see how this generalizes the binary log loss given above, note that in the binary case, $p_{i,0} = 1 - p_{i,1}$ and $y_{i,0} = 1 - y_{i,1}$, so expanding the inner sum over $y_{i,k} \in \{0, 1\}$ gives the binary log loss.

The *log_loss* function computes log loss given a list of ground-truth labels and a probability matrix, as returned by an estimator's `predict_proba` method.

```
>>> from sklearn.metrics import log_loss
>>> y_true = [0, 0, 1, 1]
>>> y_pred = [[.9, .1], [.8, .2], [.3, .7], [.01, .99]]
>>> log_loss(y_true, y_pred)
0.1738...
```

The first `[.9, .1]` in `y_pred` denotes 90% probability that the first sample has label 0. The log loss is non-negative.

Matthews correlation coefficient

The *matthews_corrcoef* function computes the Matthew's correlation coefficient (MCC) for binary classes. Quoting Wikipedia:

> "The Matthews correlation coefficient is used in machine learning as a measure of the quality of binary (two-class) classifications. It takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used even if the classes are of very different sizes. The MCC is in essence a correlation coefficient value between -1 and +1. A coefficient of +1 represents a perfect prediction, 0 an average random prediction and -1 an inverse prediction. The statistic is also known as the phi coefficient."

In the binary (two-class) case, tp, tn, fp and fn are respectively the number of true positives, true negatives, false positives and false negatives, the MCC is defined as

$$MCC = \frac{tp \times tn - fp \times fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}}.$$

In the multiclass case, the Matthews correlation coefficient can be defined in terms of a *confusion_matrix* C for K classes. To simplify the definition consider the following intermediate variables:

- $t_k = \sum_i^K C_{ik}$ the number of times class k truly occurred,
- $p_k = \sum_i^K C_{ki}$ the number of times class k was predicted,

- $c = \sum_k^K C_{kk}$ the total number of samples correctly predicted,
- $s = \sum_i^K \sum_j^K C_{ij}$ the total number of samples.

Then the multiclass MCC is defined as:

$$MCC = \frac{c \times s - \sum_k^K p_k \times t_k}{\sqrt{(s^2 - \sum_k^K p_k^2) \times (s^2 - \sum_k^K t_k^2)}}$$

When there are more than two labels, the value of the MCC will no longer range between -1 and +1. Instead the minimum value will be somewhere between -1 and 0 depending on the number and distribution of ground true labels. The maximum value is always +1.

Here is a small example illustrating the usage of the *matthews_corrcoef* function:

```
>>> from sklearn.metrics import matthews_corrcoef
>>> y_true = [+1, +1, +1, -1]
>>> y_pred = [+1, -1, +1, +1]
>>> matthews_corrcoef(y_true, y_pred)
-0.33...
```

Multi-label confusion matrix

The *multilabel_confusion_matrix* function computes class-wise (default) or sample-wise (samplewise=True) multilabel confusion matrix to evaluate the accuracy of a classification. multilabel_confusion_matrix also treats multiclass data as if it were multilabel, as this is a transformation commonly applied to evaluate multiclass problems with binary classification metrics (such as precision, recall, etc.).

When calculating class-wise multilabel confusion matrix C, the count of true negatives for class i is $C_{i,0,0}$, false negatives is $C_{i,1,0}$, true positives is $C_{i,1,1}$ and false positives is $C_{i,0,1}$.

Here is an example demonstrating the use of the *multilabel_confusion_matrix* function with *multilabel indicator matrix* input:

```
>>> import numpy as np
>>> from sklearn.metrics import multilabel_confusion_matrix
>>> y_true = np.array([[1, 0, 1],
...                    [0, 1, 0]])
>>> y_pred = np.array([[1, 0, 0],
...                    [0, 1, 1]])
>>> multilabel_confusion_matrix(y_true, y_pred)
array([[[1, 0],
        [0, 1]],

       [[1, 0],
        [0, 1]],

       [[0, 1],
        [1, 0]]])
```

Or a confusion matrix can be constructed for each sample's labels:

```
>>> multilabel_confusion_matrix(y_true, y_pred, samplewise=True)
array([[[1, 0],
        [1, 1]],
<BLANKLINE>
       [[1, 1],
        [0, 1]]])
```

Here is an example demonstrating the use of the *multilabel_confusion_matrix* function with *multiclass* input:

```
>>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
>>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
>>> multilabel_confusion_matrix(y_true, y_pred,
...                             labels=["ant", "bird", "cat"])
array([[[3, 1],
        [0, 2]],

       [[5, 0],
        [1, 0]],

       [[2, 1],
        [1, 2]]])
```

Here are some examples demonstrating the use of the *multilabel_confusion_matrix* function to calculate recall (or sensitivity), specificity, fall out and miss rate for each class in a problem with multilabel indicator matrix input.

Calculating recall (also called the true positive rate or the sensitivity) for each class:

```
>>> y_true = np.array([[0, 0, 1],
...                    [0, 1, 0],
...                    [1, 1, 0]])
>>> y_pred = np.array([[0, 1, 0],
...                    [0, 0, 1],
...                    [1, 1, 0]])
>>> mcm = multilabel_confusion_matrix(y_true, y_pred)
>>> tn = mcm[:, 0, 0]
>>> tp = mcm[:, 1, 1]
>>> fn = mcm[:, 1, 0]
>>> fp = mcm[:, 0, 1]
>>> tp / (tp + fn)
array([1. , 0.5, 0. ])
```

Calculating specificity (also called the true negative rate) for each class:

```
>>> tn / (tn + fp)
array([1. , 0. , 0.5])
```

Calculating fall out (also called the false positive rate) for each class:

```
>>> fp / (fp + tn)
array([0. , 1. , 0.5])
```

Calculating miss rate (also called the false negative rate) for each class:

```
>>> fn / (fn + tp)
array([0. , 0.5, 1. ])
```

Receiver operating characteristic (ROC)

The function *roc_curve* computes the receiver operating characteristic curve, or ROC curve. Quoting Wikipedia :

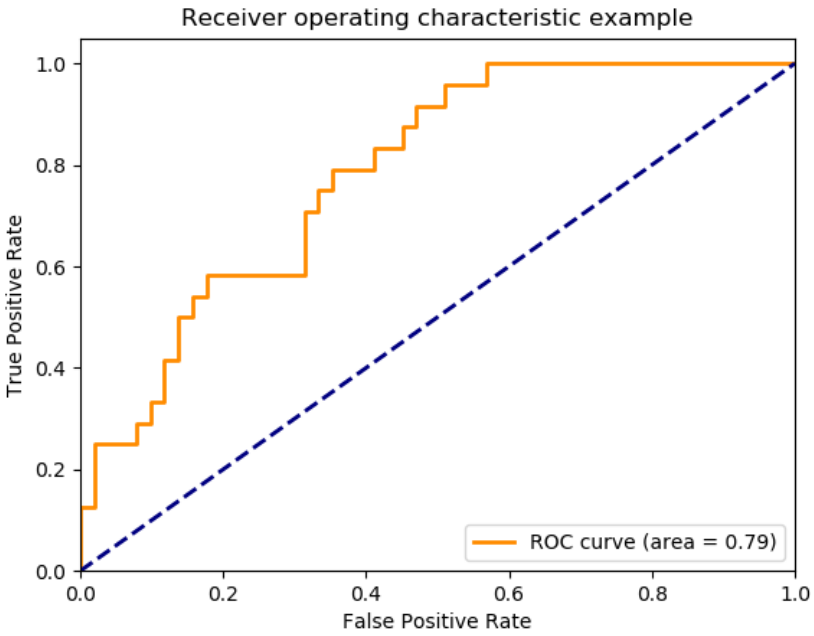
> "A receiver operating characteristic (ROC), or simply ROC curve, is a graphical plot which illustrates the performance of a binary classifier system as its discrimination threshold is varied. It is created by

plotting the fraction of true positives out of the positives (TPR = true positive rate) vs. the fraction of false positives out of the negatives (FPR = false positive rate), at various threshold settings. TPR is also known as sensitivity, and FPR is one minus the specificity or true negative rate."

This function requires the true binary value and the target scores, which can either be probability estimates of the positive class, confidence values, or binary decisions. Here is a small example of how to use the *roc_curve* function:

```
>>> import numpy as np
>>> from sklearn.metrics import roc_curve
>>> y = np.array([1, 1, 2, 2])
>>> scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> fpr, tpr, thresholds = roc_curve(y, scores, pos_label=2)
>>> fpr
array([0. , 0. , 0.5, 0.5, 1. ])
>>> tpr
array([0. , 0.5, 0.5, 1. , 1. ])
>>> thresholds
array([1.8 , 0.8 , 0.4 , 0.35, 0.1 ])
```

This figure shows an example of such an ROC curve:



The *roc_auc_score* function computes the area under the receiver operating characteristic (ROC) curve, which is also denoted by AUC or AUROC. By computing the area under the roc curve, the curve information is summarized in one number. For more information see the Wikipedia article on AUC.

```
>>> import numpy as np
>>> from sklearn.metrics import roc_auc_score
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> roc_auc_score(y_true, y_scores)
0.75
```

In multi-label classification, the *roc_auc_score* function is extended by averaging over the labels as *above*.

Compared to metrics such as the subset accuracy, the Hamming loss, or the F1 score, ROC doesn't require optimizing

a threshold for each label.

The *roc_auc_score* function can also be used in multi-class classification. Two averaging strategies are currently supported: the one-vs-one algorithm computes the average of the pairwise ROC AUC scores, and the one-vs-rest algorithm computes the average of the ROC AUC scores for each class against all other classes. In both cases, the predicted labels are provided in an array with values from 0 to `n_classes`, and the scores correspond to the probability estimates that a sample belongs to a particular class. The OvO and OvR algorithms support weighting uniformly (`average='macro'`) and by prevalence (`average='weighted'`).

One-vs-one Algorithm: Computes the average AUC of all possible pairwise combinations of classes. [HT2001] defines a multiclass AUC metric weighted uniformly:

$$\frac{2}{c(c-1)}\sum_{j=1}^{c}\sum_{k>j}^{c}(\text{AUC}(j|k)+\text{AUC}(k|j))$$

where c is the number of classes and $\text{AUC}(j|k)$ is the AUC with class j as the positive class and class k as the negative class. In general, $\text{AUC}(j|k) \neq \text{AUC}(k|j))$ in the multiclass case. This algorithm is used by setting the keyword argument `multiclass` to `'ovo'` and `average` to `'macro'`.

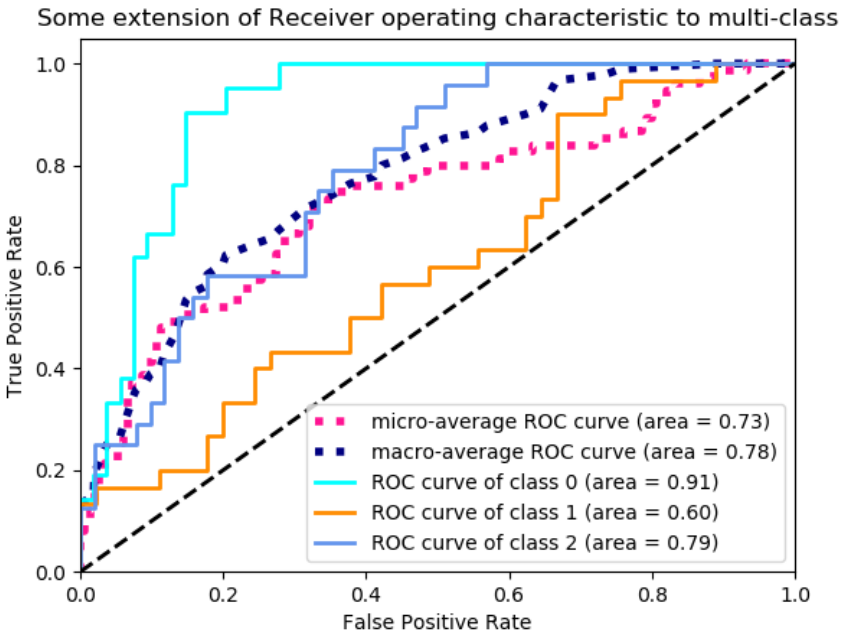
The [HT2001] multiclass AUC metric can be extended to be weighted by the prevalence:

$$\frac{2}{c(c-1)}\sum_{j=1}^{c}\sum_{k>j}^{c}p(j\cup k)(\text{AUC}(j|k)+\text{AUC}(k|j))$$

where c is the number of classes. This algorithm is used by setting the keyword argument `multiclass` to `'ovo'` and `average` to `'weighted'`. The `'weighted'` option returns a prevalence-weighted average as described in [FC2009].

One-vs-rest Algorithm: Computes the AUC of each class against the rest [PD2000]. The algorithm is functionally the same as the multilabel case. To enable this algorithm set the keyword argument `multiclass` to `'ovr'`. Like OvO, OvR supports two types of averaging: `'macro'` [F2006] and `'weighted'` [F2001].

In applications where a high false positive rate is not tolerable the parameter `max_fpr` of *roc_auc_score* can be used to summarize the ROC curve up to the given limit.



Examples:

- See *Receiver Operating Characteristic (ROC)* for an example of using ROC to evaluate the quality of the output of a classifier.
- See *Receiver Operating Characteristic (ROC) with cross validation* for an example of using ROC to evaluate classifier output quality, using cross-validation.
- See *Species distribution modeling* for an example of using ROC to model species distribution.

References:

Zero one loss

The `zero_one_loss` function computes the sum or the average of the 0-1 classification loss (L_{0-1}) over n_{samples}. By default, the function normalizes over the sample. To get the sum of the L_{0-1}, set `normalize` to `False`.

In multilabel classification, the `zero_one_loss` scores a subset as one if its labels strictly match the predictions, and as a zero if there are any errors. By default, the function returns the percentage of imperfectly predicted subsets. To get the count of such subsets instead, set `normalize` to `False`

If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value, then the 0-1 loss L_{0-1} is defined as:

$$L_{0-1}(y_i, \hat{y}_i) = 1(\hat{y}_i \neq y_i)$$

where $1(x)$ is the indicator function.

```
>>> from sklearn.metrics import zero_one_loss
>>> y_pred = [1, 2, 3, 4]
>>> y_true = [2, 2, 3, 4]
>>> zero_one_loss(y_true, y_pred)
0.25
>>> zero_one_loss(y_true, y_pred, normalize=False)
1
```

In the multilabel case with binary label indicators, where the first label set [0,1] has an error:

```
>>> zero_one_loss(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5

>>> zero_one_loss(np.array([[0, 1], [1, 1]]), np.ones((2, 2)),  normalize=False)
1
```

Example:

- See *Recursive feature elimination with cross-validation* for an example of zero one loss usage to perform recursive feature elimination with cross-validation.

Brier score loss

The *brier_score_loss* function computes the Brier score for binary classes. Quoting Wikipedia:

> "The Brier score is a proper score function that measures the accuracy of probabilistic predictions. It is applicable to tasks in which predictions must assign probabilities to a set of mutually exclusive discrete outcomes."

This function returns a score of the mean square difference between the actual outcome and the predicted probability of the possible outcome. The actual outcome has to be 1 or 0 (true or false), while the predicted probability of the actual outcome can be a value between 0 and 1.

The brier score loss is also between 0 to 1 and the lower the score (the mean square difference is smaller), the more accurate the prediction is. It can be thought of as a measure of the "calibration" of a set of probabilistic predictions.

$$BS = \frac{1}{N}\sum_{t=1}^{N}(f_t - o_t)^2$$

where : N is the total number of predictions, f_t is the predicted probability of the actual outcome o_t.

Here is a small example of usage of this function::

```
>>> import numpy as np
>>> from sklearn.metrics import brier_score_loss
>>> y_true = np.array([0, 1, 1, 0])
>>> y_true_categorical = np.array(["spam", "ham", "ham", "spam"])
>>> y_prob = np.array([0.1, 0.9, 0.8, 0.4])
>>> y_pred = np.array([0, 1, 1, 0])
>>> brier_score_loss(y_true, y_prob)
0.055
>>> brier_score_loss(y_true, 1 - y_prob, pos_label=0)
0.055
>>> brier_score_loss(y_true_categorical, y_prob, pos_label="ham")
0.055
>>> brier_score_loss(y_true, y_prob > 0.5)
0.0
```

Example:

- See *Probability calibration of classifiers* for an example of Brier score loss usage to perform probability calibration of classifiers.

References:

- G. Brier, Verification of forecasts expressed in terms of probability, Monthly weather review 78.1 (1950)

Multilabel ranking metrics

In multilabel learning, each sample can have any number of ground truth labels associated with it. The goal is to give high scores and better rank to the ground truth labels.

Coverage error

The *coverage_error* function computes the average number of labels that have to be included in the final prediction such that all true labels are predicted. This is useful if you want to know how many top-scored-labels you have to predict in average without missing any true one. The best value of this metrics is thus the average number of true labels.

Note: Our implementation's score is 1 greater than the one given in Tsoumakas et al., 2010. This extends it to handle the degenerate case in which an instance has 0 true labels.

Formally, given a binary indicator matrix of the ground truth labels $y \in \{0,1\}^{n_{\text{samples}} \times n_{\text{labels}}}$ and the score associated with each label $\hat{f} \in \mathbb{R}^{n_{\text{samples}} \times n_{\text{labels}}}$, the coverage is defined as

$$coverage(y, \hat{f}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \max_{j:y_{ij}=1} \text{rank}_{ij}$$

with $\text{rank}_{ij} = \left|\left\{k : \hat{f}_{ik} \geq \hat{f}_{ij}\right\}\right|$. Given the rank definition, ties in `y_scores` are broken by giving the maximal rank that would have been assigned to all tied values.

Here is a small example of usage of this function:

```
>>> import numpy as np
>>> from sklearn.metrics import coverage_error
>>> y_true = np.array([[1, 0, 0], [0, 0, 1]])
>>> y_score = np.array([[0.75, 0.5, 1], [1, 0.2, 0.1]])
>>> coverage_error(y_true, y_score)
2.5
```

Label ranking average precision

The *label_ranking_average_precision_score* function implements label ranking average precision (LRAP). This metric is linked to the *average_precision_score* function, but is based on the notion of label ranking instead of precision and recall.

Label ranking average precision (LRAP) averages over the samples the answer to the following question: for each ground truth label, what fraction of higher-ranked labels were true labels? This performance measure will be higher if you are able to give better rank to the labels associated with each sample. The obtained score is always strictly greater than 0, and the best value is 1. If there is exactly one relevant label per sample, label ranking average precision is equivalent to the mean reciprocal rank.

Formally, given a binary indicator matrix of the ground truth labels $y \in \{0,1\}^{n_{\text{samples}} \times n_{\text{labels}}}$ and the score associated with each label $\hat{f} \in \mathbb{R}^{n_{\text{samples}} \times n_{\text{labels}}}$, the average precision is defined as

$$LRAP(y, \hat{f}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \frac{1}{||y_i||_0} \sum_{j:y_{ij}=1} \frac{|\mathcal{L}_{ij}|}{\text{rank}_{ij}}$$

where $\mathcal{L}_{ij} = \left\{k : y_{ik} = 1, \hat{f}_{ik} \geq \hat{f}_{ij}\right\}$, $\text{rank}_{ij} = \left|\left\{k : \hat{f}_{ik} \geq \hat{f}_{ij}\right\}\right|$, $|\cdot|$ computes the cardinality of the set (i.e., the number of elements in the set), and $||\cdot||_0$ is the ℓ_0 "norm" (which computes the number of nonzero elements in a vector).

Here is a small example of usage of this function:

```
>>> import numpy as np
>>> from sklearn.metrics import label_ranking_average_precision_score
>>> y_true = np.array([[1, 0, 0], [0, 0, 1]])
>>> y_score = np.array([[0.75, 0.5, 1], [1, 0.2, 0.1]])
>>> label_ranking_average_precision_score(y_true, y_score)
0.416...
```

Ranking loss

The *label_ranking_loss* function computes the ranking loss which averages over the samples the number of label pairs that are incorrectly ordered, i.e. true labels have a lower score than false labels, weighted by the inverse of the number of ordered pairs of false and true labels. The lowest achievable ranking loss is zero.

Formally, given a binary indicator matrix of the ground truth labels $y \in \{0, 1\}^{n_{\text{samples}} \times n_{\text{labels}}}$ and the score associated with each label $\hat{f} \in \mathbb{R}^{n_{\text{samples}} \times n_{\text{labels}}}$, the ranking loss is defined as

$$ranking\_loss(y, \hat{f}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \frac{1}{||y_i||_0(n_{\text{labels}} - ||y_i||_0)} \left|\left\{(k, l) : \hat{f}_{ik} \leq \hat{f}_{il}, y_{ik} = 1, y_{il} = 0\right\}\right|$$

where $|\cdot|$ computes the cardinality of the set (i.e., the number of elements in the set) and $||\cdot||_0$ is the ℓ_0 "norm" (which computes the number of nonzero elements in a vector).

Here is a small example of usage of this function:

```
>>> import numpy as np
>>> from sklearn.metrics import label_ranking_loss
>>> y_true = np.array([[1, 0, 0], [0, 0, 1]])
>>> y_score = np.array([[0.75, 0.5, 1], [1, 0.2, 0.1]])
>>> label_ranking_loss(y_true, y_score)
0.75...
>>> # With the following prediction, we have perfect and minimal loss
>>> y_score = np.array([[1.0, 0.1, 0.2], [0.1, 0.2, 0.9]])
>>> label_ranking_loss(y_true, y_score)
0.0
```

References:

- Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

Normalized Discounted Cumulative Gain

Discounted Cumulative Gain (DCG) and Normalized Discounted Cumulative Gain (NDCG) are ranking metrics; they compare a predicted order to ground-truth scores, such as the relevance of answers to a query.

from the Wikipedia page for Discounted Cumulative Gain:

"Discounted cumulative gain (DCG) is a measure of ranking quality. In information retrieval, it is often used to measure effectiveness of web search engine algorithms or related applications. Using a graded relevance scale of documents in a search-engine result set, DCG measures the usefulness, or gain, of a document based on its position in the result list. The gain is accumulated from the top of the result list to the bottom, with the gain of each result discounted at lower ranks"

DCG orders the true targets (e.g. relevance of query answers) in the predicted order, then multiplies them by a logarithmic decay and sums the result. The sum can be truncated after the first K results, in which case we call it DCG@K. NDCG, or NDCG@K is DCG divided by the DCG obtained by a perfect prediction, so that it is always between 0 and 1. Usually, NDCG is preferred to DCG.

Compared with the ranking loss, NDCG can take into account relevance scores, rather than a ground-truth ranking. So if the ground-truth consists only of an ordering, the ranking loss should be preferred; if the ground-truth consists of actual usefulness scores (e.g. 0 for irrelevant, 1 for relevant, 2 for very relevant), NDCG can be used.

For one sample, given the vector of continuous ground-truth values for each target $y \in \mathbb{R}^M$, where M is the number of outputs, and the prediction $\hat{y}$, which induces the ranking function f, the DCG score is

$$\sum_{r=1}^{\min(K,M)} \frac{y_{f(r)}}{\log(1+r)}$$

and the NDCG score is the DCG score divided by the DCG score obtained for y.

References:

- Wikipedia entry for Discounted Cumulative Gain: https://en.wikipedia.org/wiki/Discounted_cumulative_gain
- Jarvelin, K., & Kekalainen, J. (2002). Cumulated gain-based evaluation of IR techniques. ACM Transactions on Information Systems (TOIS), 20(4), 422-446.
- Wang, Y., Wang, L., Li, Y., He, D., Chen, W., & Liu, T. Y. (2013, May). A theoretical analysis of NDCG ranking measures. In Proceedings of the 26th Annual Conference on Learning Theory (COLT 2013)
- McSherry, F., & Najork, M. (2008, March). Computing information retrieval performance measures efficiently in the presence of tied scores. In European conference on information retrieval (pp. 414-421). Springer, Berlin, Heidelberg.

Regression metrics

The *sklearn.metrics* module implements several loss, score, and utility functions to measure regression performance. Some of those have been enhanced to handle the multioutput case: *mean_squared_error*, *mean_absolute_error*, *explained_variance_score* and *r2_score*.

These functions have an `multioutput` keyword argument which specifies the way the scores or losses for each individual target should be averaged. The default is `'uniform_average'`, which specifies a uniformly weighted mean over outputs. If an `ndarray` of shape `(n_outputs,)` is passed, then its entries are interpreted as weights and an according weighted average is returned. If `multioutput` is `'raw_values'` is specified, then all unaltered individual scores or losses will be returned in an array of shape `(n_outputs,)`.

The *r2_score* and *explained_variance_score* accept an additional value `'variance_weighted'` for the `multioutput` parameter. This option leads to a weighting of each individual score by the variance of the corresponding target variable. This setting quantifies the globally captured unscaled variance. If the target variables are of different scale, then this score puts more importance on well explaining the higher variance variables. `multioutput='variance_weighted'` is the default value for *r2_score* for backward compatibility. This will be changed to `uniform_average` in the future.

Explained variance score

The *explained_variance_score* computes the explained variance regression score.

If $\hat{y}$ is the estimated target output, y the corresponding (correct) target output, and Var is Variance, the square of the standard deviation, then the explained variance is estimated as follow:

$$explained\_variance(y, \hat{y}) = 1 - \frac{Var\{y - \hat{y}\}}{Var\{y\}}$$

The best possible score is 1.0, lower values are worse.

Here is a small example of usage of the *explained_variance_score* function:

```
>>> from sklearn.metrics import explained_variance_score
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> explained_variance_score(y_true, y_pred)
0.957...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> explained_variance_score(y_true, y_pred, multioutput='raw_values')
array([0.967..., 1.        ])
>>> explained_variance_score(y_true, y_pred, multioutput=[0.3, 0.7])
0.990...
```

Max error

The *max_error* function computes the maximum residual error , a metric that captures the worst case error between the predicted value and the true value. In a perfectly fitted single output regression model, `max_error` would be `0` on the training set and though this would be highly unlikely in the real world, this metric shows the extent of error that the model had when it was fitted.

If $\hat{y}_i$ is the predicted value of the i-th sample, and y_i is the corresponding true value, then the max error is defined as

$$\text{Max Error}(y, \hat{y}) = max(|y_i - \hat{y}_i|)$$

Here is a small example of usage of the *max_error* function:

```
>>> from sklearn.metrics import max_error
>>> y_true = [3, 2, 7, 1]
>>> y_pred = [9, 2, 7, 1]
>>> max_error(y_true, y_pred)
6
```

The *max_error* does not support multioutput.

Mean absolute error

The *mean_absolute_error* function computes mean absolute error, a risk metric corresponding to the expected value of the absolute error loss or $l1$-norm loss.

If $\hat{y}_i$ is the predicted value of the i-th sample, and y_i is the corresponding true value, then the mean absolute error (MAE) estimated over n_{samples} is defined as

$$\text{MAE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} |y_i - \hat{y}_i| .$$

Here is a small example of usage of the *mean_absolute_error* function:

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_absolute_error(y_true, y_pred)
0.5
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> mean_absolute_error(y_true, y_pred)
0.75
>>> mean_absolute_error(y_true, y_pred, multioutput='raw_values')
array([0.5, 1. ])
>>> mean_absolute_error(y_true, y_pred, multioutput=[0.3, 0.7])
0.85...
```

Mean squared error

The *mean_squared_error* function computes mean square error, a risk metric corresponding to the expected value of the squared (quadratic) error or loss.

If $\hat{y}_i$ is the predicted value of the i-th sample, and y_i is the corresponding true value, then the mean squared error (MSE) estimated over n_{samples} is defined as

$$\text{MSE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} (y_i - \hat{y}_i)^2.$$

Here is a small example of usage of the *mean_squared_error* function:

```
>>> from sklearn.metrics import mean_squared_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_squared_error(y_true, y_pred)
0.375
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> mean_squared_error(y_true, y_pred)
0.7083...
```

Examples:

- See *Gradient Boosting regression* for an example of mean squared error usage to evaluate gradient boosting regression.

Mean squared logarithmic error

The *mean_squared_log_error* function computes a risk metric corresponding to the expected value of the squared logarithmic (quadratic) error or loss.

If $\hat{y}_i$ is the predicted value of the i-th sample, and y_i is the corresponding true value, then the mean squared logarithmic error (MSLE) estimated over n_{samples} is defined as

$$\text{MSLE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} (\log_e(1 + y_i) - \log_e(1 + \hat{y}_i))^2.$$

Where $\log_e(x)$ means the natural logarithm of x. This metric is best to use when targets having exponential growth, such as population counts, average sales of a commodity over a span of years etc. Note that this metric penalizes an under-predicted estimate greater than an over-predicted estimate.

Here is a small example of usage of the *mean_squared_log_error* function:

```
>>> from sklearn.metrics import mean_squared_log_error
>>> y_true = [3, 5, 2.5, 7]
>>> y_pred = [2.5, 5, 4, 8]
>>> mean_squared_log_error(y_true, y_pred)
0.039...
>>> y_true = [[0.5, 1], [1, 2], [7, 6]]
>>> y_pred = [[0.5, 2], [1, 2.5], [8, 8]]
>>> mean_squared_log_error(y_true, y_pred)
0.044...
```

Median absolute error

The *median_absolute_error* is particularly interesting because it is robust to outliers. The loss is calculated by taking the median of all absolute differences between the target and the prediction.

If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value, then the median absolute error (MedAE) estimated over n_{samples} is defined as

$$\text{MedAE}(y, \hat{y}) = \text{median}(\mid y_1 - \hat{y}_1 \mid, \ldots, \mid y_n - \hat{y}_n \mid).$$

The *median_absolute_error* does not support multioutput.

Here is a small example of usage of the *median_absolute_error* function:

```
>>> from sklearn.metrics import median_absolute_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> median_absolute_error(y_true, y_pred)
0.5
```

R^2 score, the coefficient of determination

The *r2_score* function computes the coefficient of determination, usually denoted as R^2.

It represents the proportion of variance (of y) that has been explained by the independent variables in the model. It provides an indication of goodness of fit and therefore a measure of how well unseen samples are likely to be predicted by the model, through the proportion of explained variance.

As such variance is dataset dependent, R^2 may not be meaningfully comparable across different datasets. Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value for total n samples, the estimated R^2 is defined as:

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

where $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ and $\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} \epsilon_i^2$.

Note that *r2_score* calculates unadjusted R^2 without correcting for bias in sample variance of y.

Here is a small example of usage of the *r2_score* function:

```
>>> from sklearn.metrics import r2_score
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> r2_score(y_true, y_pred)
0.948...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> r2_score(y_true, y_pred, multioutput='variance_weighted')
0.938...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> r2_score(y_true, y_pred, multioutput='uniform_average')
0.936...
>>> r2_score(y_true, y_pred, multioutput='raw_values')
array([0.965..., 0.908...])
>>> r2_score(y_true, y_pred, multioutput=[0.3, 0.7])
0.925...
```

Example:

- See *Lasso and Elastic Net for Sparse Signals* for an example of R^2 score usage to evaluate Lasso and Elastic Net on sparse signals.

Mean Poisson, Gamma, and Tweedie deviances

The *mean_tweedie_deviance* function computes the mean Tweedie deviance error with a `power` parameter (p). This is a metric that elicits predicted expectation values of regression targets.

Following special cases exist,

- when `power=0` it is equivalent to *mean_squared_error*.
- when `power=1` it is equivalent to *mean_poisson_deviance*.
- when `power=2` it is equivalent to *mean_gamma_deviance*.

If $\hat{y}_i$ is the predicted value of the i-th sample, and y_i is the corresponding true value, then the mean Tweedie deviance error (D) for power p, estimated over n_{samples} is defined as

$$\mathrm{D}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \begin{cases} (y_i - \hat{y}_i)^2, & \text{for } p = 0 \text{ (Normal)} \\ 2(y_i \log(y/\hat{y}_i) + \hat{y}_i - y_i), & \text{for} p = 1 \text{ (Poisson)} \\ 2(\log(\hat{y}_i/y_i) + y_i/\hat{y}_i - 1), & \text{for} p = 2 \text{ (Gamma)} \\ 2\left(\frac{\max(y_i,0)^{2-p}}{(1-p)(2-p)} - \frac{y\,\hat{y}_i^{1-p}}{1-p} + \frac{\hat{y}_i^{2-p}}{2-p}\right), & \text{otherwise} \end{cases}$$

Tweedie deviance is a homogeneous function of degree `2-power`. Thus, Gamma distribution with `power=2` means that simultaneously scaling `y_true` and `y_pred` has no effect on the deviance. For Poisson distribution `power=1` the deviance scales linearly, and for Normal distribution (`power=0`), quadratically. In general, the higher `power` the less weight is given to extreme deviations between true and predicted targets.

For instance, let's compare the two predictions 1.0 and 100 that are both 50% of their corresponding true value.

The mean squared error (`power=0`) is very sensitive to the prediction difference of the second point,:

```
>>> from sklearn.metrics import mean_tweedie_deviance
>>> mean_tweedie_deviance([1.0], [1.5], power=0)
0.25
>>> mean_tweedie_deviance([100.], [150.], power=0)
2500.0
```

If we increase `power` to 1,:

```
>>> mean_tweedie_deviance([1.0], [1.5], power=1)
0.18...
>>> mean_tweedie_deviance([100.], [150.], power=1)
18.9...
```

the difference in errors decreases. Finally, by setting, `power=2`:

```
>>> mean_tweedie_deviance([1.0], [1.5], power=2)
0.14...
>>> mean_tweedie_deviance([100.], [150.], power=2)
0.14...
```

we would get identical errors. The deviance when `power=2` is thus only sensitive to relative errors.

Clustering metrics

The *`sklearn.metrics`* module implements several loss, score, and utility functions. For more information see the *Clustering performance evaluation* section for instance clustering, and *Biclustering evaluation* for biclustering.

Dummy estimators

When doing supervised learning, a simple sanity check consists of comparing one's estimator against simple rules of thumb. *`DummyClassifier`* implements several such simple strategies for classification:

- `stratified` generates random predictions by respecting the training set class distribution.
- `most_frequent` always predicts the most frequent label in the training set.
- `prior` always predicts the class that maximizes the class prior (like `most_frequent`) and `predict_proba` returns the class prior.
- `uniform` generates predictions uniformly at random.
- **`constant` always predicts a constant label that is provided by the user.** A major motivation of this method is F1-scoring, when the positive class is in the minority.

Note that with all these strategies, the `predict` method completely ignores the input data!

To illustrate *`DummyClassifier`*, first let's create an imbalanced dataset:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> X, y = load_iris(return_X_y=True)
>>> y[y != 1] = -1
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Next, let's compare the accuracy of `SVC` and `most_frequent`:

```
>>> from sklearn.dummy import DummyClassifier
>>> from sklearn.svm import SVC
>>> clf = SVC(kernel='linear', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.63...
>>> clf = DummyClassifier(strategy='most_frequent', random_state=0)
>>> clf.fit(X_train, y_train)
DummyClassifier(random_state=0, strategy='most_frequent')
>>> clf.score(X_test, y_test)
0.57...
```

We see that SVC doesn't do much better than a dummy classifier. Now, let's change the kernel:

```
>>> clf = SVC(kernel='rbf', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.94...
```

We see that the accuracy was boosted to almost 100%. A cross validation strategy is recommended for a better estimate of the accuracy, if it is not too CPU costly. For more information see the *Cross-validation: evaluating estimator performance* section. Moreover if you want to optimize over the parameter space, it is highly recommended to use an appropriate methodology; see the *Tuning the hyper-parameters of an estimator* section for details.

More generally, when the accuracy of a classifier is too close to random, it probably means that something went wrong: features are not helpful, a hyperparameter is not correctly tuned, the classifier is suffering from class imbalance, etc...

DummyRegressor also implements four simple rules of thumb for regression:

- `mean` always predicts the mean of the training targets.
- `median` always predicts the median of the training targets.
- `quantile` always predicts a user provided quantile of the training targets.
- `constant` always predicts a constant value that is provided by the user.

In all these strategies, the `predict` method completely ignores the input data.

4.3.4 Model persistence

After training a scikit-learn model, it is desirable to have a way to persist the model for future use without having to retrain. The following section gives you an example of how to persist a model with pickle. We'll also review a few security and maintainability issues when working with pickle serialization.

An alternative to pickling is to export the model to another format using one of the model export tools listed under *Related Projects*. Unlike pickling, once exported you cannot recover the full Scikit-learn estimator object, but you can deploy the model for prediction, usually by using tools supporting open model interchange formats such as ONNX or PMML.

Persistence example

It is possible to save a model in scikit-learn by using Python's built-in persistence model, namely pickle:

```
>>> from sklearn import svm
>>> from sklearn import datasets
>>> clf = svm.SVC()
>>> X, y= datasets.load_iris(return_X_y=True)
>>> clf.fit(X, y)
```

(continues on next page)

(continued from previous page)

```
SVC()

>>> import pickle
>>> s = pickle.dumps(clf)
>>> clf2 = pickle.loads(s)
>>> clf2.predict(X[0:1])
array([0])
>>> y[0]
0
```

In the specific case of scikit-learn, it may be better to use joblib's replacement of pickle (`dump` & `load`), which is more efficient on objects that carry large numpy arrays internally as is often the case for fitted scikit-learn estimators, but can only pickle to the disk and not to a string:

```
>>> from joblib import dump, load
>>> dump(clf, 'filename.joblib')
```

Later you can load back the pickled model (possibly in another Python process) with:

```
>>> clf = load('filename.joblib')
```

Note: `dump` and `load` functions also accept file-like object instead of filenames. More information on data persistence with Joblib is available here.

Security & maintainability limitations

pickle (and joblib by extension), has some issues regarding maintainability and security. Because of this,

- Never unpickle untrusted data as it could lead to malicious code being executed upon loading.
- While models saved using one version of scikit-learn might load in other versions, this is entirely unsupported and inadvisable. It should also be kept in mind that operations performed on such data could give different and unexpected results.

In order to rebuild a similar model with future versions of scikit-learn, additional metadata should be saved along the pickled model:

- The training data, e.g. a reference to an immutable snapshot
- The python source code used to generate the model
- The versions of scikit-learn and its dependencies
- The cross validation score obtained on the training data

This should make it possible to check that the cross-validation score is in the same range as before.

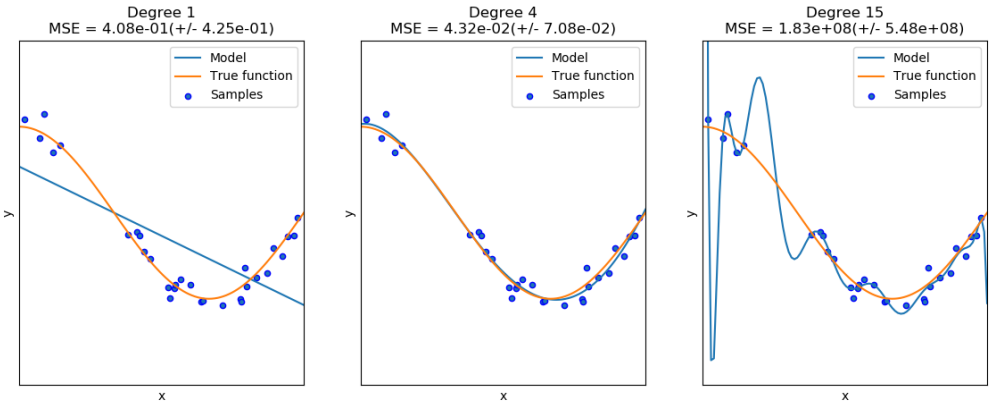
Since a model internal representation may be different on two different architectures, dumping a model on one architecture and loading it on another architecture is not supported.

If you want to know more about these issues and explore other possible serialization methods, please refer to this talk by Alex Gaynor.

4.3.5 Validation curves: plotting scores to evaluate models

Every estimator has its advantages and drawbacks. Its generalization error can be decomposed in terms of bias, variance and noise. The **bias** of an estimator is its average error for different training sets. The **variance** of an estimator indicates how sensitive it is to varying training sets. Noise is a property of the data.

In the following plot, we see a function $f(x) = \cos(\frac{3}{2}\pi x)$ and some noisy samples from that function. We use three different estimators to fit the function: linear regression with polynomial features of degree 1, 4 and 15. We see that the first estimator can at best provide only a poor fit to the samples and the true function because it is too simple (high bias), the second estimator approximates it almost perfectly and the last estimator approximates the training data perfectly but does not fit the true function very well, i.e. it is very sensitive to varying training data (high variance).



Bias and variance are inherent properties of estimators and we usually have to select learning algorithms and hyperparameters so that both bias and variance are as low as possible (see Bias-variance dilemma). Another way to reduce the variance of a model is to use more training data. However, you should only collect more training data if the true function is too complex to be approximated by an estimator with a lower variance.

In the simple one-dimensional problem that we have seen in the example it is easy to see whether the estimator suffers from bias or variance. However, in high-dimensional spaces, models can become very difficult to visualize. For this reason, it is often helpful to use the tools described below.

Examples:

- *Underfitting vs. Overfitting*
- *Plotting Validation Curves*
- *Plotting Learning Curves*

Validation curve

To validate a model we need a scoring function (see *Metrics and scoring: quantifying the quality of predictions*), for example accuracy for classifiers. The proper way of choosing multiple hyperparameters of an estimator are of course grid search or similar methods (see *Tuning the hyper-parameters of an estimator*) that select the hyperparameter with the maximum score on a validation set or multiple validation sets. Note that if we optimized the hyperparameters based on a validation score the validation score is biased and not a good estimate of the generalization any longer. To get a proper estimate of the generalization we have to compute the score on another test set.

However, it is sometimes helpful to plot the influence of a single hyperparameter on the training score and the validation score to find out whether the estimator is overfitting or underfitting for some hyperparameter values.

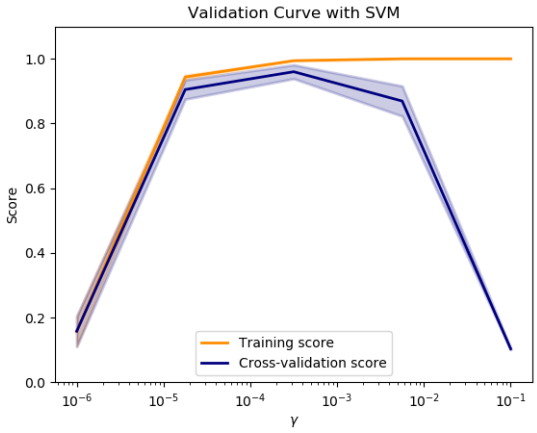
The function *validation_curve* can help in this case:

```
>>> import numpy as np
>>> from sklearn.model_selection import validation_curve
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import Ridge

>>> np.random.seed(0)
>>> X, y = load_iris(return_X_y=True)
>>> indices = np.arange(y.shape[0])
>>> np.random.shuffle(indices)
>>> X, y = X[indices], y[indices]

>>> train_scores, valid_scores = validation_curve(Ridge(), X, y, "alpha",
...                                               np.logspace(-7, 3, 3),
...                                               cv=5)
>>> train_scores
array([[0.93..., 0.94..., 0.92..., 0.91..., 0.92...],
       [0.93..., 0.94..., 0.92..., 0.91..., 0.92...],
       [0.51..., 0.52..., 0.49..., 0.47..., 0.49...]])
>>> valid_scores
array([[0.90..., 0.84..., 0.94..., 0.96..., 0.93...],
       [0.90..., 0.84..., 0.94..., 0.96..., 0.93...],
       [0.46..., 0.25..., 0.50..., 0.49..., 0.52...]])
```

If the training score and the validation score are both low, the estimator will be underfitting. If the training score is high and the validation score is low, the estimator is overfitting and otherwise it is working very well. A low training score and a high validation score is usually not possible. All three cases can be found in the plot below where we vary the parameter γ of an SVM on the digits dataset.



Learning curve

A learning curve shows the validation and training score of an estimator for varying numbers of training samples. It is a tool to find out how much we benefit from adding more training data and whether the estimator suffers more from a variance error or a bias error. Consider the following example where we plot the learning curve of a naive Bayes classifier and an SVM.

For the naive Bayes, both the validation score and the training score converge to a value that is quite low with increasing size of the training set. Thus, we will probably not benefit much from more training data.

In contrast, for small amounts of data, the training score of the SVM is much greater than the validation score. Adding more training samples will most likely increase generalization.

We can use the function *learning_curve* to generate the values that are required to plot such a learning curve (number of samples that have been used, the average scores on the training sets and the average scores on the validation sets):

```
>>> from sklearn.model_selection import learning_curve
>>> from sklearn.svm import SVC

>>> train_sizes, train_scores, valid_scores = learning_curve(
...     SVC(kernel='linear'), X, y, train_sizes=[50, 80, 110], cv=5)
>>> train_sizes
array([ 50, 80, 110])
>>> train_scores
array([[0.98..., 0.98 , 0.98..., 0.98..., 0.98...],
       [0.98..., 1.   , 0.98..., 0.98..., 0.98...],
       [0.98..., 1.   , 0.98..., 0.98..., 0.99...]])
>>> valid_scores
array([[1. ,  0.93...,  1. ,  1. ,  0.96...],
       [1. ,  0.96...,  1. ,  1. ,  0.96...],
       [1. ,  0.96...,  1. ,  1. ,  0.96...]])
```

4.4 Inspection

Predictive performance is often the main goal of developing machine learning models. Yet summarising performance with an evaluation metric is often insufficient: it assumes that the evaluation metric and test dataset perfectly reflect the target domain, which is rarely true. In certain domains, a model needs a certain level of interpretability before it can be deployed. A model that is exhibiting performance issues needs to be debugged for one to understand the model's underlying issue. The *sklearn.inspection* module provides tools to help understand the predictions from a model and what affects them. This can be used to evaluate assumptions and biases of a model, design a better model, or to diagnose issues with model performance.

4.4.1 Partial dependence plots

Partial dependence plots (PDP) show the dependence between the target response[1] and a set of 'target' features, marginalizing over the values of all other features (the 'complement' features). Intuitively, we can interpret the partial dependence as the expected target response as a function of the 'target' features.

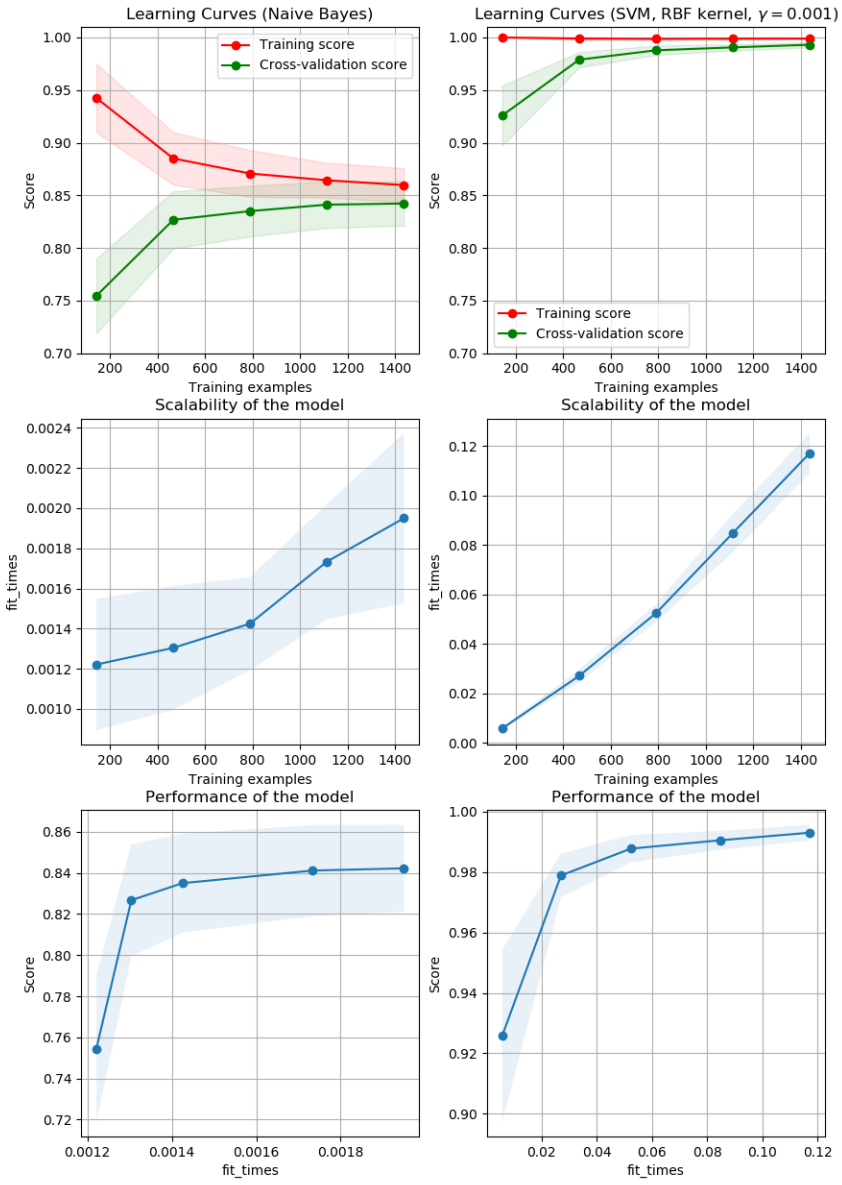
Due to the limits of human perception the size of the target feature set must be small (usually, one or two) thus the target features are usually chosen among the most important features.

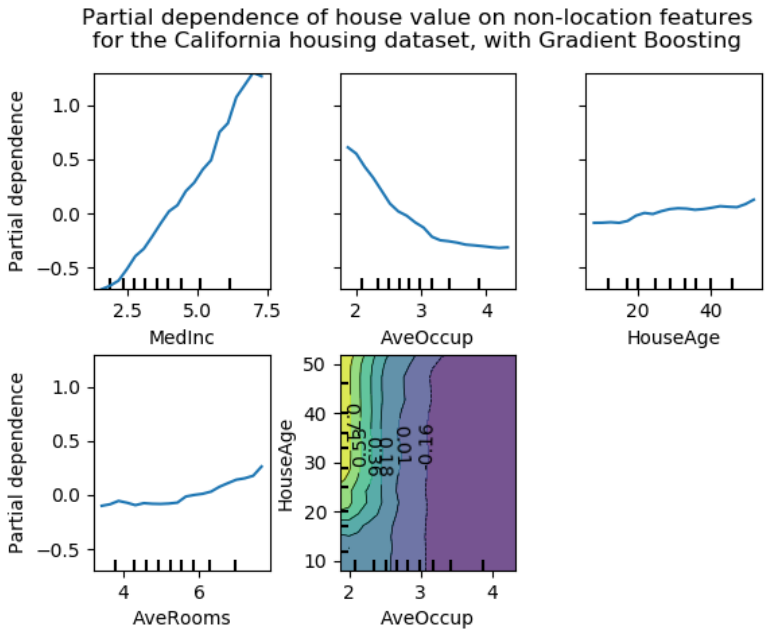
The figure below shows four one-way and one two-way partial dependence plots for the California housing dataset, with a *GradientBoostingRegressor*:

One-way PDPs tell us about the interaction between the target response and the target feature (e.g. linear, non-linear). The upper left plot in the above figure shows the effect of the median income in a district on the median house price; we can clearly see a linear relationship among them. Note that PDPs assume that the target features are independent from the complement features, and this assumption is often violated in practice.

PDPs with two target features show the interactions among the two features. For example, the two-variable PDP in the above figure shows the dependence of median house price on joint values of house age and average occupants per

[1] For classification, the target response may be the probability of a class (the positive class for binary classification), or the decision function.


Learning Curves (Naive Bayes)
Training score
Cross-validation score
Score
Training examples
Learning Curves (SVM, RBF kernel, $\gamma = 0.001$)
Training score
Cross-validation score
Score
Training examples
Scalability of the model
fit_times
Training examples
Scalability of the model
fit_times
Training examples
Performance of the model
Score
fit_times
Performance of the model
Score
fit_times



household. We can clearly see an interaction between the two features: for an average occupancy greater than two, the house price is nearly independent of the house age, whereas for values less than 2 there is a strong dependence on age.

The *sklearn.inspection* module provides a convenience function *plot_partial_dependence* to create one-way and two-way partial dependence plots. In the below example we show how to create a grid of partial dependence plots: two one-way PDPs for the features `0` and `1` and a two-way PDP between the two features:

```
>>> from sklearn.datasets import make_hastie_10_2
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> from sklearn.inspection import plot_partial_dependence

>>> X, y = make_hastie_10_2(random_state=0)
>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X, y)
>>> features = [0, 1, (0, 1)]
>>> plot_partial_dependence(clf, X, features)
```

You can access the newly created figure and Axes objects using `plt.gcf()` and `plt.gca()`.

For multi-class classification, you need to set the class label for which the PDPs should be created via the `target` argument:

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> mc_clf = GradientBoostingClassifier(n_estimators=10,
...     max_depth=1).fit(iris.data, iris.target)
>>> features = [3, 2, (3, 2)]
>>> plot_partial_dependence(mc_clf, X, features, target=0)
```

The same parameter `target` is used to specify the target in multi-output regression settings.

If you need the raw values of the partial dependence function rather than the plots, you can use the *sklearn.inspection.partial_dependence* function:

```
>>> from sklearn.inspection import partial_dependence

>>> pdp, axes = partial_dependence(clf, X, [0])
>>> pdp
array([[ 2.466...,  2.466..., ...
>>> axes
[array([-1.624..., -1.592..., ...
```

The values at which the partial dependence should be evaluated are directly generated from `X`. For 2-way partial dependence, a 2D-grid of values is generated. The `values` field returned by *`sklearn.inspection.partial_dependence`* gives the actual values used in the grid for each target feature. They also correspond to the axis of the plots.

For each value of the 'target' features in the `grid` the partial dependence function needs to marginalize the predictions of the estimator over all possible values of the 'complement' features. With the `'brute'` method, this is done by replacing every target feature value of `X` by those in the grid, and computing the average prediction.

In decision trees this can be evaluated efficiently without reference to the training data (`'recursion'` method). For each grid point a weighted tree traversal is performed: if a split node involves a 'target' feature, the corresponding left or right branch is followed, otherwise both branches are followed, each branch is weighted by the fraction of training samples that entered that branch. Finally, the partial dependence is given by a weighted average of all visited leaves. Note that with the `'recursion'` method, `X` is only used to generate the grid, not to compute the averaged predictions. The averaged predictions will always be computed on the data with which the trees were trained.

Examples:

- *Partial Dependence Plots*

References

T. Hastie, R. Tibshirani and J. Friedman, The Elements of Statistical Learning, Second Edition, Section 10.13.2, Springer, 2009.

C. Molnar, Interpretable Machine Learning, Section 5.1, 2019.

4.4.2 Permutation feature importance

Permutation feature importance is a model inspection technique that can be used for any *fitted estimator* when the data is rectangular. This is especially useful for non-linear or opaque *estimators*. The permutation feature importance is defined to be the decrease in a model score when a single feature value is randomly shuffled[1]. This procedure breaks the relationship between the feature and the target, thus the drop in the model score is indicative of how much the model depends on the feature. This technique benefits from being model agnostic and can be calculated many times with different permutations of the feature.

The *`permutation_importance`* function calculates the feature importance of *estimators* for a given dataset. The `n_repeats` parameter sets the number of times a feature is randomly shuffled and returns a sample of feature importances. Permutation importances can either be computed on the training set or an held-out testing or validation set. Using a held-out set makes it possible to highlight which features contribute the most to the generalization power of the inspected model. Features that are important on the training set but not on the held-out set might cause the model to overfit.

[1] L. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001. https://doi.org/10.1023/A:1010933404324

Note that features that are deemed non-important for some model with a low predictive performance could be highly predictive for a model that generalizes better. The conclusions should always be drawn in the context of the specific model under inspection and cannot be automatically generalized to the intrinsic predictive value of the features by them-selves. Therefore it is always important to evaluate the predictive power of a model using a held-out set (or better with cross-validation) prior to computing importances.

Relation to impurity-based importance in trees

Tree based models provides a different measure of feature importances based on the mean decrease in impurity (MDI, the splitting criterion). This gives importance to features that may not be predictive on unseen data. The permutation feature importance avoids this issue, since it can be applied to unseen data. Furthermore, impurity-based feature importance for trees are strongly biased and favor high cardinality features (typically numerical features). Permutation-based feature importances do not exhibit such a bias. Additionally, the permutation feature importance may use an arbitrary metric on the tree's predictions. These two methods of obtaining feature importance are explored in: *Permutation Importance vs Random Forest Feature Importance (MDI)*.

Strongly correlated features

When two features are correlated and one of the features is permuted, the model will still have access to the feature through its correlated feature. This will result in a lower importance for both features, where they might *actually* be important. One way to handle this is to cluster features that are correlated and only keep one feature from each cluster. This use case is explored in: *Permutation Importance with Multicollinear or Correlated Features*.

Examples:

- *Permutation Importance vs Random Forest Feature Importance (MDI)*
- *Permutation Importance with Multicollinear or Correlated Features*

References:

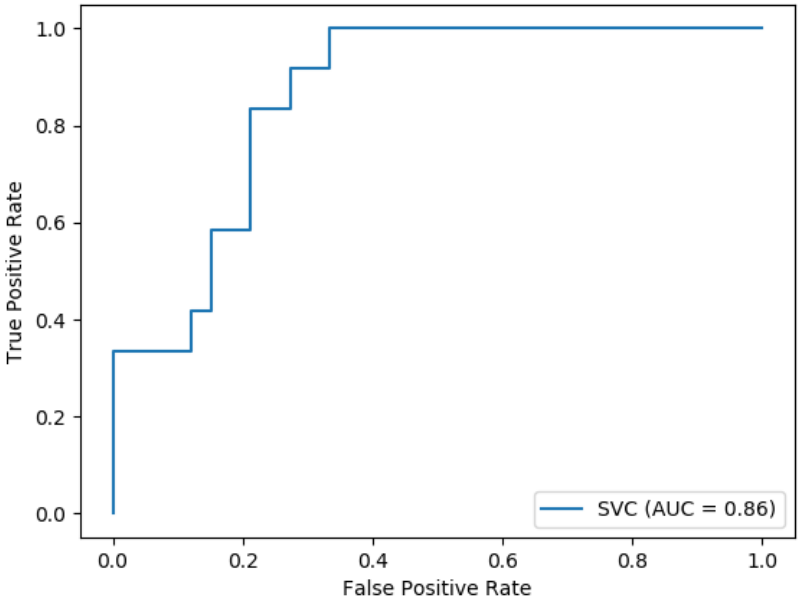
4.5 Visualizations

Scikit-learn defines a simple API for creating visualizations for machine learning. The key feature of this API is to allow for quick plotting and visual adjustments without recalculation. In the following example, we plot a ROC curve for a fitted support vector machine:

```
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import plot_roc_curve
from sklearn.datasets import load_wine

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
svc = SVC(random_state=42)
svc.fit(X_train, y_train)

svc_disp = plot_roc_curve(svc, X_test, y_test)
```



The returned `svc_disp` object allows us to continue using the already computed ROC curve for SVC in future plots. In this case, the `svc_disp` is a *`RocCurveDisplay`* that stores the computed values as attributes called `roc_auc`, `fpr`, and `tpr`. Next, we train a random forest classifier and plot the previously computed roc curve again by using the `plot` method of the `Display` object.

```
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier

rfc = RandomForestClassifier(random_state=42)
rfc.fit(X_train, y_train)

ax = plt.gca()
rfc_disp = plot_roc_curve(rfc, X_test, y_test, ax=ax, alpha=0.8)
svc_disp.plot(ax=ax, alpha=0.8)
```

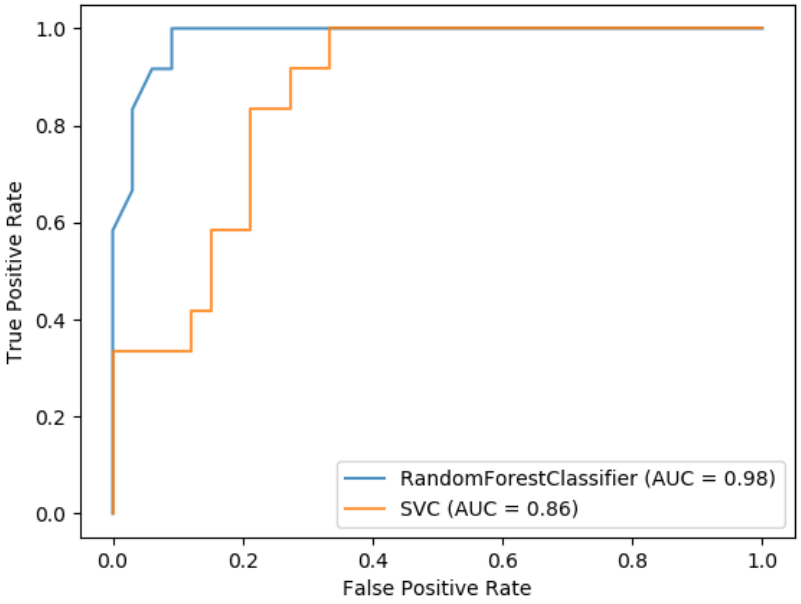
Notice that we pass `alpha=0.8` to the plot functions to adjust the alpha values of the curves.

Examples:

- *ROC Curve with Visualization API*
- *Advanced Plotting With Partial Dependence*

4.5.1 Available Plotting Utilities

Functions



inspection.plot_partial_dependence(...[, ...])	Partial dependence plots.
metrics.plot_confusion_matrix(estimator, X, ...)	Plot Confusion Matrix.
metrics.plot_precision_recall_curve(...[, ...])	Plot Precision Recall Curve for binary classifiers.
metrics.plot_roc_curve(estimator, X, y[, ...])	Plot Receiver operating characteristic (ROC) curve.

Display Objects

inspection.PartialDependenceDisplay(...)	Partial Dependence Plot (PDP) visualization.
metrics.ConfusionMatrixDisplay(...)	Confusion Matrix visualization.
metrics.PrecisionRecallDisplay(precision, ...)	Precision Recall visualization.
metrics.RocCurveDisplay(fpr, tpr, roc_auc, ...)	ROC Curve visualization.

4.6 Dataset transformations

scikit-learn provides a library of transformers, which may clean (see *Preprocessing data*), reduce (see *Unsupervised dimensionality reduction*), expand (see *Kernel Approximation*) or generate (see *Feature extraction*) feature representations.

Like other estimators, these are represented by classes with a `fit` method, which learns model parameters (e.g. mean and standard deviation for normalization) from a training set, and a `transform` method which applies this transformation model to unseen data. `fit_transform` may be more convenient and efficient for modelling and transforming the training data simultaneously.

Combining such transformers, either in parallel or series is covered in *Pipelines and composite estimators*. *Pairwise metrics, Affinities and Kernels* covers transforming feature spaces into affinity matrices, while *Transforming the prediction target (y)* considers transformations of the target space (e.g. categorical labels) for use in scikit-learn.

4.6.1 Pipelines and composite estimators

All estimators in a pipeline, except the last one, must be transformers (i.e. must have a *transform* method). The last estimator may be any type (transformer, classifier, etc.).

Usage

Construction

The *Pipeline* is built using a list of `(key, value)` pairs, where the `key` is a string containing the name you want to give this step and `value` is an estimator object:

```
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.svm import SVC
>>> from sklearn.decomposition import PCA
>>> estimators = [('reduce_dim', PCA()), ('clf', SVC())]
>>> pipe = Pipeline(estimators)
>>> pipe
Pipeline(steps=[('reduce_dim', PCA()), ('clf', SVC())])
```

The utility function *make_pipeline* is a shorthand for constructing pipelines; it takes a variable number of estimators and returns a pipeline, filling in the names automatically:

```
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.naive_bayes import MultinomialNB
>>> from sklearn.preprocessing import Binarizer
>>> make_pipeline(Binarizer(), MultinomialNB())
Pipeline(steps=[('binarizer', Binarizer()), ('multinomialnb', MultinomialNB())])
```

Accessing steps

The estimators of a pipeline are stored as a list in the `steps` attribute, but can be accessed by index or name by indexing (with `[idx]`) the Pipeline:

```
>>> pipe.steps[0]
('reduce_dim', PCA())
>>> pipe[0]
PCA()
>>> pipe['reduce_dim']
PCA()
```

Pipeline's `named_steps` attribute allows accessing steps by name with tab completion in interactive environments:

```
>>> pipe.named_steps.reduce_dim is pipe['reduce_dim']
True
```

A sub-pipeline can also be extracted using the slicing notation commonly used for Python Sequences such as lists or strings (although only a step of 1 is permitted). This is convenient for performing only some of the transformations (or their inverse):

```
>>> pipe[:1]
Pipeline(steps=[('reduce_dim', PCA())])
>>> pipe[-1:]
Pipeline(steps=[('clf', SVC())])
```

Nested parameters

Parameters of the estimators in the pipeline can be accessed using the `<estimator>__<parameter>` syntax:

```
>>> pipe.set_params(clf__C=10)
Pipeline(steps=[('reduce_dim', PCA()), ('clf', SVC(C=10))])
```

This is particularly important for doing grid searches:

```
>>> from sklearn.model_selection import GridSearchCV
>>> param_grid = dict(reduce_dim__n_components=[2, 5, 10],
...                   clf__C=[0.1, 10, 100])
>>> grid_search = GridSearchCV(pipe, param_grid=param_grid)
```

Individual steps may also be replaced as parameters, and non-final steps may be ignored by setting them to `'passthrough'`:

```
>>> from sklearn.linear_model import LogisticRegression
>>> param_grid = dict(reduce_dim=['passthrough', PCA(5), PCA(10)],
...                   clf=[SVC(), LogisticRegression()],
...                   clf__C=[0.1, 10, 100])
>>> grid_search = GridSearchCV(pipe, param_grid=param_grid)
```

The estimators of the pipeline can be retrieved by index:

```
>>> pipe[0]
PCA()
```

or by name:

```
>>> pipe['reduce_dim']
PCA()
```

Examples:

- *Pipeline Anova SVM*
- *Sample pipeline for text feature extraction and evaluation*
- *Pipelining: chaining a PCA and a logistic regression*
- *Explicit feature map approximation for RBF kernels*
- *SVM-Anova: SVM with univariate feature selection*
- *Selecting dimensionality reduction with Pipeline and GridSearchCV*

See also:

- *Composite estimators and parameter spaces*

Notes

Calling `fit` on the pipeline is the same as calling `fit` on each estimator in turn, `transform` the input and pass it on to the next step. The pipeline has all the methods that the last estimator in the pipeline has, i.e. if the last estimator is a classifier, the *Pipeline* can be used as a classifier. If the last estimator is a transformer, again, so is the pipeline.

Caching transformers: avoid repeated computation

Fitting transformers may be computationally expensive. With its `memory` parameter set, *Pipeline* will cache each transformer after calling `fit`. This feature is used to avoid computing the fit transformers within a pipeline if the parameters and input data are identical. A typical example is the case of a grid search in which the transformers can be fitted only once and reused for each configuration.

The parameter `memory` is needed in order to cache the transformers. `memory` can be either a string containing the directory where to cache the transformers or a joblib.Memory object:

```
>>> from tempfile import mkdtemp
>>> from shutil import rmtree
>>> from sklearn.decomposition import PCA
>>> from sklearn.svm import SVC
>>> from sklearn.pipeline import Pipeline
>>> estimators = [('reduce_dim', PCA()), ('clf', SVC())]
>>> cachedir = mkdtemp()
>>> pipe = Pipeline(estimators, memory=cachedir)
>>> pipe
Pipeline(memory=...,
         steps=[('reduce_dim', PCA()), ('clf', SVC())])
>>> # Clear the cache directory when you don't need it anymore
>>> rmtree(cachedir)
```

Warning: Side effect of caching transformers

Using a *Pipeline* without cache enabled, it is possible to inspect the original instance such as:

```
>>> from sklearn.datasets import load_digits
>>> X_digits, y_digits = load_digits(return_X_y=True)
>>> pca1 = PCA()
>>> svm1 = SVC()
>>> pipe = Pipeline([('reduce_dim', pca1), ('clf', svm1)])
>>> pipe.fit(X_digits, y_digits)
Pipeline(steps=[('reduce_dim', PCA()), ('clf', SVC())])
>>> # The pca instance can be inspected directly
>>> print(pca1.components_)
    [[-1.77484909e-19  ... 4.07058917e-18]]
```

Enabling caching triggers a clone of the transformers before fitting. Therefore, the transformer instance given to the pipeline cannot be inspected directly. In following example, accessing the `PCA` instance `pca2` will raise an `AttributeError` since `pca2` will be an unfitted transformer. Instead, use the attribute `named_steps` to inspect estimators within the pipeline:

```
>>> cachedir = mkdtemp()
>>> pca2 = PCA()
>>> svm2 = SVC()
>>> cached_pipe = Pipeline([('reduce_dim', pca2), ('clf', svm2)],
...                        memory=cachedir)
>>> cached_pipe.fit(X_digits, y_digits)
Pipeline(memory=...,
         steps=[('reduce_dim', PCA()), ('clf', SVC())])
>>> print(cached_pipe.named_steps['reduce_dim'].components_)
    [[-1.77484909e-19  ... 4.07058917e-18]]
>>> # Remove the cache directory
>>> rmtree(cachedir)
```

Examples:

- *Selecting dimensionality reduction with Pipeline and GridSearchCV*

Transforming target in regression

`TransformedTargetRegressor` transforms the targets `y` before fitting a regression model. The predictions are mapped back to the original space via an inverse transform. It takes as an argument the regressor that will be used for prediction, and the transformer that will be applied to the target variable:

```
>>> import numpy as np
>>> from sklearn.datasets import load_boston
>>> from sklearn.compose import TransformedTargetRegressor
>>> from sklearn.preprocessing import QuantileTransformer
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.model_selection import train_test_split
>>> X, y = load_boston(return_X_y=True)
>>> transformer = QuantileTransformer(output_distribution='normal')
>>> regressor = LinearRegression()
>>> regr = TransformedTargetRegressor(regressor=regressor,
...                                   transformer=transformer)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
>>> regr.fit(X_train, y_train)
TransformedTargetRegressor(...)
>>> print('R2 score: {0:.2f}'.format(regr.score(X_test, y_test)))
R2 score: 0.67
>>> raw_target_regr = LinearRegression().fit(X_train, y_train)
>>> print('R2 score: {0:.2f}'.format(raw_target_regr.score(X_test, y_test)))
R2 score: 0.64
```

For simple transformations, instead of a Transformer object, a pair of functions can be passed, defining the transformation and its inverse mapping:

```
>>> def func(x):
...     return np.log(x)
>>> def inverse_func(x):
...     return np.exp(x)
```

Subsequently, the object is created as:

```
>>> regr = TransformedTargetRegressor(regressor=regressor,
...                                   func=func,
...                                   inverse_func=inverse_func)
>>> regr.fit(X_train, y_train)
TransformedTargetRegressor(...)
>>> print('R2 score: {0:.2f}'.format(regr.score(X_test, y_test)))
R2 score: 0.65
```

By default, the provided functions are checked at each fit to be the inverse of each other. However, it is possible to bypass this checking by setting `check_inverse` to `False`:

```
>>> def inverse_func(x):
...     return x
>>> regr = TransformedTargetRegressor(regressor=regressor,
...                                   func=func,
...                                   inverse_func=inverse_func,
...                                   check_inverse=False)
>>> regr.fit(X_train, y_train)
TransformedTargetRegressor(...)
>>> print('R2 score: {0:.2f}'.format(regr.score(X_test, y_test)))
R2 score: -4.50
```

Note: The transformation can be triggered by setting either `transformer` or the pair of functions `func` and `inverse_func`. However, setting both options will raise an error.

Examples:

- *Effect of transforming the targets in regression model*

FeatureUnion: composite feature spaces

FeatureUnion combines several transformer objects into a new transformer that combines their output. A *FeatureUnion* takes a list of transformer objects. During fitting, each of these is fit to the data independently. The transformers are applied in parallel, and the feature matrices they output are concatenated side-by-side into a larger matrix.

When you want to apply different transformations to each field of the data, see the related class *sklearn.compose.ColumnTransformer* (see *user guide*).

FeatureUnion serves the same purposes as *Pipeline* - convenience and joint parameter estimation and validation.

FeatureUnion and *Pipeline* can be combined to create complex models.

(A *FeatureUnion* has no way of checking whether two transformers might produce identical features. It only produces a union when the feature sets are disjoint, and making sure they are the caller's responsibility.)

Usage

A *FeatureUnion* is built using a list of `(key, value)` pairs, where the `key` is the name you want to give to a given transformation (an arbitrary string; it only serves as an identifier) and `value` is an estimator object:

```
>>> from sklearn.pipeline import FeatureUnion
>>> from sklearn.decomposition import PCA
>>> from sklearn.decomposition import KernelPCA
>>> estimators = [('linear_pca', PCA()), ('kernel_pca', KernelPCA())]
>>> combined = FeatureUnion(estimators)
>>> combined
FeatureUnion(transformer_list=[('linear_pca', PCA()),
                               ('kernel_pca', KernelPCA())])
```

Like pipelines, feature unions have a shorthand constructor called *make_union* that does not require explicit naming of the components.

Like `Pipeline`, individual steps may be replaced using `set_params`, and ignored by setting to `'drop'`:

```
>>> combined.set_params(kernel_pca='drop')
FeatureUnion(transformer_list=[('linear_pca', PCA()),
                               ('kernel_pca', 'drop')])
```

Examples:

- *Concatenating multiple feature extraction methods*

ColumnTransformer for heterogeneous data

Warning: The *compose.ColumnTransformer* class is experimental and the API is subject to change.

Many datasets contain features of different types, say text, floats, and dates, where each type of feature requires separate preprocessing or feature extraction steps. Often it is easiest to preprocess data before applying scikit-learn methods, for example using pandas. Processing your data before passing it to scikit-learn might be problematic for one of the following reasons:

1. Incorporating statistics from test data into the preprocessors makes cross-validation scores unreliable (known as *data leakage*), for example in the case of scalers or imputing missing values.
2. You may want to include the parameters of the preprocessors in a *parameter search*.

The *ColumnTransformer* helps performing different transformations for different columns of the data, within a *Pipeline* that is safe from data leakage and that can be parametrized. *ColumnTransformer* works on arrays, sparse matrices, and pandas DataFrames.

To each column, a different transformation can be applied, such as preprocessing or a specific feature extraction method:

```
>>> import pandas as pd
>>> X = pd.DataFrame(
...     {'city': ['London', 'London', 'Paris', 'Sallisaw'],
...      'title': ["His Last Bow", "How Watson Learned the Trick",
...                "A Moveable Feast", "The Grapes of Wrath"],
...      'expert_rating': [5, 3, 4, 5],
...      'user_rating': [4, 5, 4, 3]})
```

For this data, we might want to encode the `'city'` column as a categorical variable using *preprocessing.OneHotEncoder* but apply a *feature_extraction.text.CountVectorizer* to the `'title'` column.

As we might use multiple feature extraction methods on the same column, we give each transformer a unique name, say `'city_category'` and `'title_bow'`. By default, the remaining rating columns are ignored (`remainder='drop'`):

```
>>> from sklearn.compose import ColumnTransformer
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> from sklearn.preprocessing import OneHotEncoder
>>> column_trans = ColumnTransformer(
...     [('city_category', OneHotEncoder(dtype='int'),['city']),
...      ('title_bow', CountVectorizer(), 'title')],
...     remainder='drop')

>>> column_trans.fit(X)
ColumnTransformer(transformers=[('city_category', OneHotEncoder(dtype='int'),
                                 ['city']),
                                ('title_bow', CountVectorizer(), 'title')])

>>> column_trans.get_feature_names()
['city_category__x0_London', 'city_category__x0_Paris', 'city_category__x0_Sallisaw',
'title_bow__bow', 'title_bow__feast', 'title_bow__grapes', 'title_bow__his',
'title_bow__how', 'title_bow__last', 'title_bow__learned', 'title_bow__moveable',
'title_bow__of', 'title_bow__the', 'title_bow__trick', 'title_bow__watson',
'title_bow__wrath']

>>> column_trans.transform(X).toarray()
array([[1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0],
       [0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1]]...)
```

In the above example, the *CountVectorizer* expects a 1D array as input and therefore the columns were specified as a string (`'title'`). However, *preprocessing.OneHotEncoder* as most of other transformers expects 2D data, therefore in that case you need to specify the column as a list of strings (`['city']`).

Apart from a scalar or a single item list, the column selection can be specified as a list of multiple items, an integer array, a slice, a boolean mask, or with a *make_column_selector*. The *make_column_selector* is used to select columns based on data type or column name:

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.compose import make_column_selector
>>> ct = ColumnTransformer([
...       ('scale', StandardScaler(),
...       make_column_selector(dtype_include=np.number)),
...       ('onehot',
...       OneHotEncoder(),
...       make_column_selector(pattern='city', dtype_include=object))])
>>> ct.fit_transform(X)
array([[ 0.904...,  0.      ,  1. ,  0. ,  0. ],
       [-1.507...,  1.414...,  1. ,  0. ,  0. ],
       [-0.301...,  0.      ,  0. ,  1. ,  0. ],
       [ 0.904..., -1.414...,  0. ,  0. ,  1. ]])
```

Strings can reference columns if the input is a DataFrame, integers are always interpreted as the positional columns.

We can keep the remaining rating columns by setting `remainder='passthrough'`. The values are appended to the end of the transformation:

```
>>> column_trans = ColumnTransformer(
...     [('city_category', OneHotEncoder(dtype='int'),['city']),
...      ('title_bow', CountVectorizer(), 'title')],
...     remainder='passthrough')

>>> column_trans.fit_transform(X)
array([[1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 5, 4],
       [1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 3, 5],
       [0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 4, 4],
       [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 5, 3]]...)
```

The `remainder` parameter can be set to an estimator to transform the remaining rating columns. The transformed values are appended to the end of the transformation:

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> column_trans = ColumnTransformer(
...     [('city_category', OneHotEncoder(), ['city']),
...      ('title_bow', CountVectorizer(), 'title')],
...     remainder=MinMaxScaler())

>>> column_trans.fit_transform(X)[:, -2:]
array([[1. , 0.5],
       [0. , 1. ],
       [0.5, 0.5],
       [1. , 0. ]])
```

The *make_column_transformer* function is available to more easily create a *ColumnTransformer* object. Specifically, the names will be given automatically. The equivalent for the above example would be:

```
>>> from sklearn.compose import make_column_transformer
>>> column_trans = make_column_transformer(
...     (OneHotEncoder(), ['city']),
...     (CountVectorizer(), 'title'),
...     remainder=MinMaxScaler())
>>> column_trans
ColumnTransformer(remainder=MinMaxScaler(),
                  transformers=[('onehotencoder', OneHotEncoder(), ['city']),
                                ('countvectorizer', CountVectorizer(),
                                 'title')])
```

Examples:

- *Column Transformer with Heterogeneous Data Sources*
- *Column Transformer with Mixed Types*

4.6.2 Feature extraction

The *sklearn.feature_extraction* module can be used to extract features in a format supported by machine learning algorithms from datasets consisting of formats such as text and image.

Note: Feature extraction is very different from *Feature selection*: the former consists in transforming arbitrary data, such as text or images, into numerical features usable for machine learning. The latter is a machine learning technique

applied on these features.

Loading features from dicts

The class *DictVectorizer* can be used to convert feature arrays represented as lists of standard Python `dict` objects to the NumPy/SciPy representation used by scikit-learn estimators.

While not particularly fast to process, Python's `dict` has the advantages of being convenient to use, being sparse (absent features need not be stored) and storing feature names in addition to values.

DictVectorizer implements what is called one-of-K or "one-hot" coding for categorical (aka nominal, discrete) features. Categorical features are "attribute-value" pairs where the value is restricted to a list of discrete of possibilities without ordering (e.g. topic identifiers, types of objects, tags, names...).

In the following, "city" is a categorical attribute while "temperature" is a traditional numerical feature:

```
>>> measurements = [
...     {'city': 'Dubai', 'temperature': 33.},
...     {'city': 'London', 'temperature': 12.},
...     {'city': 'San Francisco', 'temperature': 18.},
... ]

>>> from sklearn.feature_extraction import DictVectorizer
>>> vec = DictVectorizer()

>>> vec.fit_transform(measurements).toarray()
array([[ 1.,  0.,  0., 33.],
       [ 0.,  1.,  0., 12.],
       [ 0.,  0.,  1., 18.]])

>>> vec.get_feature_names()
['city=Dubai', 'city=London', 'city=San Francisco', 'temperature']
```

DictVectorizer is also a useful representation transformation for training sequence classifiers in Natural Language Processing models that typically work by extracting feature windows around a particular word of interest.

For example, suppose that we have a first algorithm that extracts Part of Speech (PoS) tags that we want to use as complementary tags for training a sequence classifier (e.g. a chunker). The following dict could be such a window of features extracted around the word 'sat' in the sentence 'The cat sat on the mat.':

```
>>> pos_window = [
...     {
...         'word-2': 'the',
...         'pos-2': 'DT',
...         'word-1': 'cat',
...         'pos-1': 'NN',
...         'word+1': 'on',
...         'pos+1': 'PP',
...     },
...     # in a real application one would extract many such dictionaries
... ]
```

This description can be vectorized into a sparse two-dimensional matrix suitable for feeding into a classifier (maybe after being piped into a *text.TfidfTransformer* for normalization):

```
>>> vec = DictVectorizer()
>>> pos_vectorized = vec.fit_transform(pos_window)
>>> pos_vectorized
<1x6 sparse matrix of type '<... 'numpy.float64'>'
    with 6 stored elements in Compressed Sparse ... format>
>>> pos_vectorized.toarray()
array([[1., 1., 1., 1., 1., 1.]])
>>> vec.get_feature_names()
['pos+1=PP', 'pos-1=NN', 'pos-2=DT', 'word+1=on', 'word-1=cat', 'word-2=the']
```

As you can imagine, if one extracts such a context around each individual word of a corpus of documents the resulting matrix will be very wide (many one-hot-features) with most of them being valued to zero most of the time. So as to make the resulting data structure able to fit in memory the `DictVectorizer` class uses a `scipy.sparse` matrix by default instead of a `numpy.ndarray`.

Feature hashing

The class *`FeatureHasher`* is a high-speed, low-memory vectorizer that uses a technique known as feature hashing, or the "hashing trick". Instead of building a hash table of the features encountered in training, as the vectorizers do, instances of *`FeatureHasher`* apply a hash function to the features to determine their column index in sample matrices directly. The result is increased speed and reduced memory usage, at the expense of inspectability; the hasher does not remember what the input features looked like and has no `inverse_transform` method.

Since the hash function might cause collisions between (unrelated) features, a signed hash function is used and the sign of the hash value determines the sign of the value stored in the output matrix for a feature. This way, collisions are likely to cancel out rather than accumulate error, and the expected mean of any output feature's value is zero. This mechanism is enabled by default with `alternate_sign=True` and is particularly useful for small hash table sizes (`n_features < 10000`). For large hash table sizes, it can be disabled, to allow the output to be passed to estimators like *`sklearn.naive_bayes.MultinomialNB`* or *`sklearn.feature_selection.chi2`* feature selectors that expect non-negative inputs.

`FeatureHasher` accepts either mappings (like Python's `dict` and its variants in the `collections` module), `(feature, value)` pairs, or strings, depending on the constructor parameter `input_type`. Mapping are treated as lists of `(feature, value)` pairs, while single strings have an implicit value of 1, so `['feat1', 'feat2', 'feat3']` is interpreted as `[('feat1', 1), ('feat2', 1), ('feat3', 1)]`. If a single feature occurs multiple times in a sample, the associated values will be summed (so `('feat', 2)` and `('feat', 3.5)` become `('feat', 5.5)`). The output from *`FeatureHasher`* is always a `scipy.sparse` matrix in the CSR format.

Feature hashing can be employed in document classification, but unlike *`text.CountVectorizer`*, *`FeatureHasher`* does not do word splitting or any other preprocessing except Unicode-to-UTF-8 encoding; see *Vectorizing a large text corpus with the hashing trick*, below, for a combined tokenizer/hasher.

As an example, consider a word-level natural language processing task that needs features extracted from `(token, part_of_speech)` pairs. One could use a Python generator function to extract features:

```
def token_features(token, part_of_speech):
    if token.isdigit():
        yield "numeric"
    else:
        yield "token={}".format(token.lower())
        yield "token,pos={},{}".format(token, part_of_speech)
    if token[0].isupper():
        yield "uppercase_initial"
    if token.isupper():
        yield "all_uppercase"
    yield "pos={}".format(part_of_speech)
```

Then, the `raw_X` to be fed to `FeatureHasher.transform` can be constructed using:

```
raw_X = (token_features(tok, pos_tagger(tok)) for tok in corpus)
```

and fed to a hasher with:

```
hasher = FeatureHasher(input_type='string')
X = hasher.transform(raw_X)
```

to get a `scipy.sparse` matrix `X`.

Note the use of a generator comprehension, which introduces laziness into the feature extraction: tokens are only processed on demand from the hasher.

Implementation details

FeatureHasher uses the signed 32-bit variant of MurmurHash3. As a result (and because of limitations in `scipy.sparse`), the maximum number of features supported is currently $2^{31} - 1$.

The original formulation of the hashing trick by Weinberger et al. used two separate hash functions h and ξ to determine the column index and sign of a feature, respectively. The present implementation works under the assumption that the sign bit of MurmurHash3 is independent of its other bits.

Since a simple modulo is used to transform the hash function to a column index, it is advisable to use a power of two as the `n_features` parameter; otherwise the features will not be mapped evenly to the columns.

References:

- Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola and Josh Attenberg (2009). Feature hashing for large scale multitask learning. Proc. ICML.
- MurmurHash3.

Text feature extraction

The Bag of Words representation

Text Analysis is a major application field for machine learning algorithms. However the raw data, a sequence of symbols cannot be fed directly to the algorithms themselves as most of them expect numerical feature vectors with a fixed size rather than the raw text documents with variable length.

In order to address this, scikit-learn provides utilities for the most common ways to extract numerical features from text content, namely:

- **tokenizing** strings and giving an integer id for each possible token, for instance by using white-spaces and punctuation as token separators.
- **counting** the occurrences of tokens in each document.
- **normalizing** and weighting with diminishing importance tokens that occur in the majority of samples / documents.

In this scheme, features and samples are defined as follows:

- each **individual token occurrence frequency** (normalized or not) is treated as a **feature**.
- the vector of all the token frequencies for a given **document** is considered a multivariate **sample**.

A corpus of documents can thus be represented by a matrix with one row per document and one column per token (e.g. word) occurring in the corpus.

We call **vectorization** the general process of turning a collection of text documents into numerical feature vectors. This specific strategy (tokenization, counting and normalization) is called the **Bag of Words** or "Bag of n-grams" representation. Documents are described by word occurrences while completely ignoring the relative position information of the words in the document.

Sparsity

As most documents will typically use a very small subset of the words used in the corpus, the resulting matrix will have many feature values that are zeros (typically more than 99% of them).

For instance a collection of 10,000 short text documents (such as emails) will use a vocabulary with a size in the order of 100,000 unique words in total while each document will use 100 to 1000 unique words individually.

In order to be able to store such a matrix in memory but also to speed up algebraic operations matrix / vector, implementations will typically use a sparse representation such as the implementations available in the `scipy.sparse` package.

Common Vectorizer usage

`CountVectorizer` implements both tokenization and occurrence counting in a single class:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
```

This model has many parameters, however the default values are quite reasonable (please see the *reference documentation* for the details):

```
>>> vectorizer = CountVectorizer()
>>> vectorizer
CountVectorizer()
```

Let's use it to tokenize and count the word occurrences of a minimalistic corpus of text documents:

```
>>> corpus = [
...     'This is the first document.',
...     'This is the second second document.',
...     'And the third one.',
...     'Is this the first document?',
... ]
>>> X = vectorizer.fit_transform(corpus)
>>> X
<4x9 sparse matrix of type '<... 'numpy.int64'>'
    with 19 stored elements in Compressed Sparse ... format>
```

The default configuration tokenizes the string by extracting words of at least 2 letters. The specific function that does this step can be requested explicitly:

```
>>> analyze = vectorizer.build_analyzer()
>>> analyze("This is a text document to analyze.") == (
...     ['this', 'is', 'text', 'document', 'to', 'analyze'])
True
```

Each term found by the analyzer during the fit is assigned a unique integer index corresponding to a column in the resulting matrix. This interpretation of the columns can be retrieved as follows:

```
>>> vectorizer.get_feature_names() == (
...     ['and', 'document', 'first', 'is', 'one',
...      'second', 'the', 'third', 'this'])
True

>>> X.toarray()
array([[0, 1, 1, 1, 0, 0, 1, 0, 1],
       [0, 1, 0, 1, 0, 2, 1, 0, 1],
       [1, 0, 0, 0, 1, 0, 1, 1, 0],
       [0, 1, 1, 1, 0, 0, 1, 0, 1]]...)
```

The converse mapping from feature name to column index is stored in the `vocabulary_` attribute of the vectorizer:

```
>>> vectorizer.vocabulary_.get('document')
1
```

Hence words that were not seen in the training corpus will be completely ignored in future calls to the transform method:

```
>>> vectorizer.transform(['Something completely new.']).toarray()
array([[0, 0, 0, 0, 0, 0, 0, 0]]...)
```

Note that in the previous corpus, the first and the last documents have exactly the same words hence are encoded in equal vectors. In particular we lose the information that the last document is an interrogative form. To preserve some of the local ordering information we can extract 2-grams of words in addition to the 1-grams (individual words):

```
>>> bigram_vectorizer = CountVectorizer(ngram_range=(1, 2),
...                                     token_pattern=r'\b\w+\b', min_df=1)
>>> analyze = bigram_vectorizer.build_analyzer()
>>> analyze('Bi-grams are cool!') == (
...     ['bi', 'grams', 'are', 'cool', 'bi grams', 'grams are', 'are cool'])
True
```

The vocabulary extracted by this vectorizer is hence much bigger and can now resolve ambiguities encoded in local positioning patterns:

```
>>> X_2 = bigram_vectorizer.fit_transform(corpus).toarray()
>>> X_2
array([[0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0],
       [0, 0, 1, 0, 0, 1, 1, 0, 0, 2, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0],
       [1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1]]...)
```

In particular the interrogative form “Is this” is only present in the last document:

```
>>> feature_index = bigram_vectorizer.vocabulary_.get('is this')
>>> X_2[:, feature_index]
array([0, 0, 0, 1]...)
```

Using stop words

Stop words are words like “and”, “the”, “him”, which are presumed to be uninformative in representing the content of a text, and which may be removed to avoid them being construed as signal for prediction. Sometimes, however, similar words are useful for prediction, such as in classifying writing style or personality.

There are several known issues in our provided 'english' stop word list. It does not aim to be a general, 'one-size-fits-all' solution as some tasks may require a more custom solution. See [NQY18] for more details.

Please take care in choosing a stop word list. Popular stop word lists may include words that are highly informative to some tasks, such as *computer*.

You should also make sure that the stop word list has had the same preprocessing and tokenization applied as the one used in the vectorizer. The word *we've* is split into *we* and *ve* by CountVectorizer's default tokenizer, so if *we've* is in `stop_words`, but *ve* is not, *ve* will be retained from *we've* in transformed text. Our vectorizers will try to identify and warn about some kinds of inconsistencies.

References

Tf–idf term weighting

In a large text corpus, some words will be very present (e.g. "the", "a", "is" in English) hence carrying very little meaningful information about the actual contents of the document. If we were to feed the direct count data directly to a classifier those very frequent terms would shadow the frequencies of rarer yet more interesting terms.

In order to re-weight the count features into floating point values suitable for usage by a classifier it is very common to use the tf–idf transform.

Tf means **term-frequency** while tf–idf means term-frequency times **inverse document-frequency**: tf-idf(t,d) $=$ tf(t,d) $\times$ idf(t).

Using the `TfidfTransformer`'s default settings, `TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True, sublinear_tf=False)` the term frequency, the number of times a term occurs in a given document, is multiplied with idf component, which is computed as

$\text{idf}(t) = \log \frac{1+n}{1+\text{df}(t)} + 1,$

where n is the total number of documents in the document set, and $\text{df}(t)$ is the number of documents in the document set that contain term t. The resulting tf-idf vectors are then normalized by the Euclidean norm:

$v_{norm} = \frac{v}{||v||_2} = \frac{v}{\sqrt{{v_1}^2 + {v_2}^2 + \cdots + {v_n}^2}}.$

This was originally a term weighting scheme developed for information retrieval (as a ranking function for search engines results) that has also found good use in document classification and clustering.

The following sections contain further explanations and examples that illustrate how the tf-idfs are computed exactly and how the tf-idfs computed in scikit-learn's *TfidfTransformer* and *TfidfVectorizer* differ slightly from the standard textbook notation that defines the idf as

$\text{idf}(t) = \log \frac{n}{1+\text{df}(t)}.$

In the *TfidfTransformer* and *TfidfVectorizer* with `smooth_idf=False`, the "1" count is added to the idf instead of the idf's denominator:

$\text{idf}(t) = \log \frac{n}{\text{df}(t)} + 1$

This normalization is implemented by the *TfidfTransformer* class:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> transformer = TfidfTransformer(smooth_idf=False)
>>> transformer
TfidfTransformer(smooth_idf=False)
```

Again please see the *reference documentation* for the details on all the parameters.

Let's take an example with the following counts. The first term is present 100% of the time hence not very interesting. The two other features only in less than 50% of the time hence probably more representative of the content of the documents:

```
>>> counts = [[3, 0, 1],
...           [2, 0, 0],
...           [3, 0, 0],
...           [4, 0, 0],
...           [3, 2, 0],
...           [3, 0, 2]]
...
>>> tfidf = transformer.fit_transform(counts)
>>> tfidf
<6x3 sparse matrix of type '<... 'numpy.float64'>'
    with 9 stored elements in Compressed Sparse ... format>

>>> tfidf.toarray()
array([[0.81940995, 0.        , 0.57320793],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [0.47330339, 0.88089948, 0.        ],
       [0.58149261, 0.        , 0.81355169]])
```

Each row is normalized to have unit Euclidean norm:

$v_{norm} = \frac{v}{||v||_2} = \frac{v}{\sqrt{v_1{}^2 + v_2{}^2 + \cdots + v_n{}^2}}$

For example, we can compute the tf-idf of the first term in the first document in the `counts` array as follows:

$n = 6$

$\text{df}(t)_{\text{term1}} = 6$

$\text{idf}(t)_{\text{term1}} = \log \frac{n}{\text{df}(t)} + 1 = \log(1) + 1 = 1$

$\text{tf-idf}_{\text{term1}} = \text{tf} \times \text{idf} = 3 \times 1 = 3$

Now, if we repeat this computation for the remaining 2 terms in the document, we get

$\text{tf-idf}_{\text{term2}} = 0 \times (\log(6/1) + 1) = 0$

$\text{tf-idf}_{\text{term3}} = 1 \times (\log(6/2) + 1) \approx 2.0986$

and the vector of raw tf-idfs:

$\text{tf-idf}_{\text{raw}} = [3, 0, 2.0986]$.

Then, applying the Euclidean (L2) norm, we obtain the following tf-idfs for document 1:

$\frac{[3, 0, 2.0986]}{\sqrt{\left(3^2 + 0^2 + 2.0986^2\right)}} = [0.819, 0, 0.573]$.

Furthermore, the default parameter `smooth_idf=True` adds "1" to the numerator and denominator as if an extra document was seen containing every term in the collection exactly once, which prevents zero divisions:

$\text{idf}(t) = \log \frac{1+n}{1+\text{df}(t)} + 1$

Using this modification, the tf-idf of the third term in document 1 changes to 1.8473:

$\text{tf-idf}_{\text{term3}} = 1 \times \log(7/3) + 1 \approx 1.8473$

And the L2-normalized tf-idf changes to

$\frac{[3,0,1.8473]}{\sqrt{\left(3^2+0^2+1.8473^2\right)}} = [0.8515, 0, 0.5243]$:

```
>>> transformer = TfidfTransformer()
>>> transformer.fit_transform(counts).toarray()
array([[0.85151335, 0.        , 0.52433293],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [0.55422893, 0.83236428, 0.        ],
       [0.63035731, 0.        , 0.77630514]])
```

The weights of each feature computed by the `fit` method call are stored in a model attribute:

```
>>> transformer.idf_
array([1. ..., 2.25..., 1.84...])
```

As tf–idf is very often used for text features, there is also another class called *`TfidfVectorizer`* that combines all the options of *`CountVectorizer`* and *`TfidfTransformer`* in a single model:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> vectorizer = TfidfVectorizer()
>>> vectorizer.fit_transform(corpus)
<4x9 sparse matrix of type '<... 'numpy.float64'>'
    with 19 stored elements in Compressed Sparse ... format>
```

While the tf–idf normalization is often very useful, there might be cases where the binary occurrence markers might offer better features. This can be achieved by using the `binary` parameter of *`CountVectorizer`*. In particular, some estimators such as *Bernoulli Naive Bayes* explicitly model discrete boolean random variables. Also, very short texts are likely to have noisy tf–idf values while the binary occurrence info is more stable.

As usual the best way to adjust the feature extraction parameters is to use a cross-validated grid search, for instance by pipelining the feature extractor with a classifier:

- *Sample pipeline for text feature extraction and evaluation*

Decoding text files

Text is made of characters, but files are made of bytes. These bytes represent characters according to some *encoding*. To work with text files in Python, their bytes must be *decoded* to a character set called Unicode. Common encodings are ASCII, Latin-1 (Western Europe), KOI8-R (Russian) and the universal encodings UTF-8 and UTF-16. Many others exist.

Note: An encoding can also be called a 'character set', but this term is less accurate: several encodings can exist for a single character set.

The text feature extractors in scikit-learn know how to decode text files, but only if you tell them what encoding the files are in. The *`CountVectorizer`* takes an `encoding` parameter for this purpose. For modern text files, the correct encoding is probably UTF-8, which is therefore the default (`encoding="utf-8"`).

If the text you are loading is not actually encoded with UTF-8, however, you will get a `UnicodeDecodeError`. The vectorizers can be told to be silent about decoding errors by setting the `decode_error` parameter to either `"ignore"` or `"replace"`. See the documentation for the Python function `bytes.decode` for more details (type `help(bytes.decode)` at the Python prompt).

If you are having trouble decoding text, here are some things to try:

- Find out what the actual encoding of the text is. The file might come with a header or README that tells you the encoding, or there might be some standard encoding you can assume based on where the text comes from.
- You may be able to find out what kind of encoding it is in general using the UNIX command `file`. The Python `chardet` module comes with a script called `chardetect.py` that will guess the specific encoding, though you cannot rely on its guess being correct.
- You could try UTF-8 and disregard the errors. You can decode byte strings with `bytes.decode(errors='replace')` to replace all decoding errors with a meaningless character, or set `decode_error='replace'` in the vectorizer. This may damage the usefulness of your features.
- Real text may come from a variety of sources that may have used different encodings, or even be sloppily decoded in a different encoding than the one it was encoded with. This is common in text retrieved from the Web. The Python package ftfy can automatically sort out some classes of decoding errors, so you could try decoding the unknown text as `latin-1` and then using `ftfy` to fix errors.
- If the text is in a mish-mash of encodings that is simply too hard to sort out (which is the case for the 20 Newsgroups dataset), you can fall back on a simple single-byte encoding such as `latin-1`. Some text may display incorrectly, but at least the same sequence of bytes will always represent the same feature.

For example, the following snippet uses `chardet` (not shipped with scikit-learn, must be installed separately) to figure out the encoding of three texts. It then vectorizes the texts and prints the learned vocabulary. The output is not shown here.

```
>>> import chardet    # doctest: +SKIP
>>> text1 = b"Sei mir gegr\xc3\xbc\xc3\x9ft mein Sauerkraut"
>>> text2 = b"holdselig sind deine Ger\xfcche"
>>> text3 = b"\xff\xfeA\x00u\x00f\x00 \x00F\x00l\x00\xfc\x00g\x00e\x00l\x00n\x00␣
↪\x00d\x00e\x00s\x00 \x00G\x00e\x00s\x00a\x00n\x00g\x00e\x00s\x00,\x00␣
↪\x00H\x00e\x00r\x00z\x00l\x00i\x00e\x00b\x00c\x00h\x00e\x00n\x00,\x00␣
↪\x00t\x00r\x00a\x00g\x00 \x00i\x00c\x00h\x00 \x00d\x00i\x00c\x00h\x00␣
↪\x00f\x00o\x00r\x00t\x00"
>>> decoded = [x.decode(chardet.detect(x)['encoding'])
...            for x in (text1, text2, text3)]        # doctest: +SKIP
>>> v = CountVectorizer().fit(decoded).vocabulary_    # doctest: +SKIP
>>> for term in v: print(v)                           # doctest: +SKIP
```

(Depending on the version of `chardet`, it might get the first one wrong.)

For an introduction to Unicode and character encodings in general, see Joel Spolsky's Absolute Minimum Every Software Developer Must Know About Unicode.

Applications and examples

The bag of words representation is quite simplistic but surprisingly useful in practice.

In particular in a **supervised setting** it can be successfully combined with fast and scalable linear models to train **document classifiers**, for instance:

- *Classification of text documents using sparse features*

In an **unsupervised setting** it can be used to group similar documents together by applying clustering algorithms such as *K-means*:

- *Clustering text documents using k-means*

Finally it is possible to discover the main topics of a corpus by relaxing the hard assignment constraint of clustering, for instance by using *Non-negative matrix factorization (NMF or NNMF)*:

- *Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation*

Limitations of the Bag of Words representation

A collection of unigrams (what bag of words is) cannot capture phrases and multi-word expressions, effectively disregarding any word order dependence. Additionally, the bag of words model doesn't account for potential misspellings or word derivations.

N-grams to the rescue! Instead of building a simple collection of unigrams (n=1), one might prefer a collection of bigrams (n=2), where occurrences of pairs of consecutive words are counted.

One might alternatively consider a collection of character n-grams, a representation resilient against misspellings and derivations.

For example, let's say we're dealing with a corpus of two documents: `['words', 'wprds']`. The second document contains a misspelling of the word 'words'. A simple bag of words representation would consider these two as very distinct documents, differing in both of the two possible features. A character 2-gram representation, however, would find the documents matching in 4 out of 8 features, which may help the preferred classifier decide better:

```
>>> ngram_vectorizer = CountVectorizer(analyzer='char_wb', ngram_range=(2, 2))
>>> counts = ngram_vectorizer.fit_transform(['words', 'wprds'])
>>> ngram_vectorizer.get_feature_names() == (
...     [' w', 'ds', 'or', 'pr', 'rd', 's ', 'wo', 'wp'])
True
>>> counts.toarray().astype(int)
array([[1, 1, 1, 0, 1, 1, 1, 0],
       [1, 1, 0, 1, 1, 1, 0, 1]])
```

In the above example, `char_wb` analyzer is used, which creates n-grams only from characters inside word boundaries (padded with space on each side). The `char` analyzer, alternatively, creates n-grams that span across words:

```
>>> ngram_vectorizer = CountVectorizer(analyzer='char_wb', ngram_range=(5, 5))
>>> ngram_vectorizer.fit_transform(['jumpy fox'])
<1x4 sparse matrix of type '<... 'numpy.int64'>'
   with 4 stored elements in Compressed Sparse ... format>
>>> ngram_vectorizer.get_feature_names() == (
...     [' fox ', ' jump', 'jumpy', 'umpy '])
True

>>> ngram_vectorizer = CountVectorizer(analyzer='char', ngram_range=(5, 5))
>>> ngram_vectorizer.fit_transform(['jumpy fox'])
<1x5 sparse matrix of type '<... 'numpy.int64'>'
    with 5 stored elements in Compressed Sparse ... format>
>>> ngram_vectorizer.get_feature_names() == (
...     ['jumpy', 'mpy f', 'py fo', 'umpy ', 'y fox'])
True
```

The word boundaries-aware variant `char_wb` is especially interesting for languages that use white-spaces for word separation as it generates significantly less noisy features than the raw `char` variant in that case. For such languages it can increase both the predictive accuracy and convergence speed of classifiers trained using such features while retaining the robustness with regards to misspellings and word derivations.

While some local positioning information can be preserved by extracting n-grams instead of individual words, bag of words and bag of n-grams destroy most of the inner structure of the document and hence most of the meaning carried by that internal structure.

In order to address the wider task of Natural Language Understanding, the local structure of sentences and paragraphs should thus be taken into account. Many such models will thus be casted as "Structured output" problems which are currently outside of the scope of scikit-learn.

Vectorizing a large text corpus with the hashing trick

The above vectorization scheme is simple but the fact that it holds an **in- memory mapping from the string tokens to the integer feature indices** (the `vocabulary_` attribute) causes several **problems when dealing with large datasets**:

- the larger the corpus, the larger the vocabulary will grow and hence the memory use too,
- fitting requires the allocation of intermediate data structures of size proportional to that of the original dataset.
- building the word-mapping requires a full pass over the dataset hence it is not possible to fit text classifiers in a strictly online manner.
- pickling and un-pickling vectorizers with a large `vocabulary_` can be very slow (typically much slower than pickling / un-pickling flat data structures such as a NumPy array of the same size),
- it is not easily possible to split the vectorization work into concurrent sub tasks as the `vocabulary_` attribute would have to be a shared state with a fine grained synchronization barrier: the mapping from token string to feature index is dependent on ordering of the first occurrence of each token hence would have to be shared, potentially harming the concurrent workers' performance to the point of making them slower than the sequential variant.

It is possible to overcome those limitations by combining the "hashing trick" (*Feature hashing*) implemented by the *`sklearn.feature_extraction.FeatureHasher`* class and the text preprocessing and tokenization features of the *`CountVectorizer`*.

This combination is implementing in *`HashingVectorizer`*, a transformer class that is mostly API compatible with *`CountVectorizer`*. *`HashingVectorizer`* is stateless, meaning that you don't have to call `fit` on it:

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> hv = HashingVectorizer(n_features=10)
>>> hv.transform(corpus)
<4x10 sparse matrix of type '<... 'numpy.float64'>'
    with 16 stored elements in Compressed Sparse ... format>
```

You can see that 16 non-zero feature tokens were extracted in the vector output: this is less than the 19 non-zeros extracted previously by the *`CountVectorizer`* on the same toy corpus. The discrepancy comes from hash function collisions because of the low value of the `n_features` parameter.

In a real world setting, the `n_features` parameter can be left to its default value of `2 ** 20` (roughly one million possible features). If memory or downstream models size is an issue selecting a lower value such as `2 ** 18` might help without introducing too many additional collisions on typical text classification tasks.

Note that the dimensionality does not affect the CPU training time of algorithms which operate on CSR matrices (`LinearSVC(dual=True)`, `Perceptron`, `SGDClassifier`, `PassiveAggressive`) but it does for algorithms that work with CSC matrices (`LinearSVC(dual=False)`, `Lasso()`, etc).

Let's try again with the default setting:

```
>>> hv = HashingVectorizer()
>>> hv.transform(corpus)
<4x1048576 sparse matrix of type '<... 'numpy.float64'>'
    with 19 stored elements in Compressed Sparse ... format>
```

We no longer get the collisions, but this comes at the expense of a much larger dimensionality of the output space. Of course, other terms than the 19 used here might still collide with each other.

The *`HashingVectorizer`* also comes with the following limitations:

- it is not possible to invert the model (no `inverse_transform` method), nor to access the original string representation of the features, because of the one-way nature of the hash function that performs the mapping.

- it does not provide IDF weighting as that would introduce statefulness in the model. A *TfidfTransformer* can be appended to it in a pipeline if required.

Performing out-of-core scaling with HashingVectorizer

An interesting development of using a *HashingVectorizer* is the ability to perform out-of-core scaling. This means that we can learn from data that does not fit into the computer's main memory.

A strategy to implement out-of-core scaling is to stream data to the estimator in mini-batches. Each mini-batch is vectorized using *HashingVectorizer* so as to guarantee that the input space of the estimator has always the same dimensionality. The amount of memory used at any time is thus bounded by the size of a mini-batch. Although there is no limit to the amount of data that can be ingested using such an approach, from a practical point of view the learning time is often limited by the CPU time one wants to spend on the task.

For a full-fledged example of out-of-core scaling in a text classification task see *Out-of-core classification of text documents*.

Customizing the vectorizer classes

It is possible to customize the behavior by passing a callable to the vectorizer constructor:

```
>>> def my_tokenizer(s):
...     return s.split()
...
>>> vectorizer = CountVectorizer(tokenizer=my_tokenizer)
>>> vectorizer.build_analyzer()(u"Some... punctuation!") == (
...     ['some...', 'punctuation!'])
True
```

In particular we name:

- `preprocessor`: a callable that takes an entire document as input (as a single string), and returns a possibly transformed version of the document, still as an entire string. This can be used to remove HTML tags, lowercase the entire document, etc.
- `tokenizer`: a callable that takes the output from the preprocessor and splits it into tokens, then returns a list of these.
- `analyzer`: a callable that replaces the preprocessor and tokenizer. The default analyzers all call the preprocessor and tokenizer, but custom analyzers will skip this. N-gram extraction and stop word filtering take place at the analyzer level, so a custom analyzer may have to reproduce these steps.

(Lucene users might recognize these names, but be aware that scikit-learn concepts may not map one-to-one onto Lucene concepts.)

To make the preprocessor, tokenizer and analyzers aware of the model parameters it is possible to derive from the class and override the `build_preprocessor`, `build_tokenizer` and `build_analyzer` factory methods instead of passing custom functions.

Some tips and tricks:

- If documents are pre-tokenized by an external package, then store them in files (or strings) with the tokens separated by whitespace and pass `analyzer=str.split`
- Fancy token-level analysis such as stemming, lemmatizing, compound splitting, filtering based on part-of-speech, etc. are not included in the scikit-learn codebase, but can be added by customizing either the tokenizer or the analyzer. Here's a `CountVectorizer` with a tokenizer and lemmatizer using NLTK:

```
>>> from nltk import word_tokenize
>>> from nltk.stem import WordNetLemmatizer
>>> class LemmaTokenizer:
...     def __init__(self):
...         self.wnl = WordNetLemmatizer()
...     def __call__(self, doc):
...         return [self.wnl.lemmatize(t) for t in word_tokenize(doc)]
...
>>> vect = CountVectorizer(tokenizer=LemmaTokenizer())
```

(Note that this will not filter out punctuation.)

The following example will, for instance, transform some British spelling to American spelling:

```
>>> import re
>>> def to_british(tokens):
...     for t in tokens:
...         t = re.sub(r"(...)our$", r"\1or", t)
...         t = re.sub(r"([bt])re$", r"\1er", t)
...         t = re.sub(r"([iy])s(e$|ing|ation)", r"\1z\2", t)
...         t = re.sub(r"ogue$", "og", t)
...         yield t
...
>>> class CustomVectorizer(CountVectorizer):
...     def build_tokenizer(self):
...         tokenize = super().build_tokenizer()
...         return lambda doc: list(to_british(tokenize(doc)))
...
>>> print(CustomVectorizer().build_analyzer()(u"color colour"))
[...'color', ...'color']
```

for other styles of preprocessing; examples include stemming, lemmatization, or normalizing numerical tokens, with the latter illustrated in:

- *Biclustering documents with the Spectral Co-clustering algorithm*

Customizing the vectorizer can also be useful when handling Asian languages that do not use an explicit word separator such as whitespace.

Image feature extraction

Patch extraction

The *extract_patches_2d* function extracts patches from an image stored as a two-dimensional array, or three-dimensional with color information along the third axis. For rebuilding an image from all its patches, use *reconstruct_from_patches_2d*. For example let use generate a 4x4 pixel picture with 3 color channels (e.g. in RGB format):

```
>>> import numpy as np
>>> from sklearn.feature_extraction import image

>>> one_image = np.arange(4 * 4 * 3).reshape((4, 4, 3))
>>> one_image[:, :, 0]  # R channel of a fake RGB picture
array([[ 0,  3,  6,  9],
       [12, 15, 18, 21],
       [24, 27, 30, 33],
```

(continues on next page)

(continued from previous page)

```
       [36, 39, 42, 45]])

>>> patches = image.extract_patches_2d(one_image, (2, 2), max_patches=2,
...     random_state=0)
>>> patches.shape
(2, 2, 2, 3)
>>> patches[:, :, :, 0]
array([[[ 0,  3],
        [12, 15]],

       [[15, 18],
        [27, 30]]])
>>> patches = image.extract_patches_2d(one_image, (2, 2))
>>> patches.shape
(9, 2, 2, 3)
>>> patches[4, :, :, 0]
array([[15, 18],
       [27, 30]])
```

Let us now try to reconstruct the original image from the patches by averaging on overlapping areas:

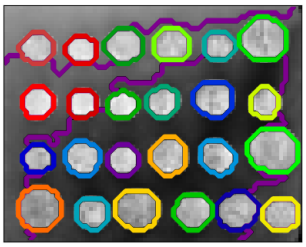
```
>>> reconstructed = image.reconstruct_from_patches_2d(patches, (4, 4, 3))
>>> np.testing.assert_array_equal(one_image, reconstructed)
```

The *PatchExtractor* class works in the same way as *extract_patches_2d*, only it supports multiple images as input. It is implemented as an estimator, so it can be used in pipelines. See:

```
>>> five_images = np.arange(5 * 4 * 4 * 3).reshape(5, 4, 4, 3)
>>> patches = image.PatchExtractor((2, 2)).transform(five_images)
>>> patches.shape
(45, 2, 2, 3)
```

Connectivity graph of an image

Several estimators in the scikit-learn can use connectivity information between features or samples. For instance Ward clustering (*Hierarchical clustering*) can cluster together only neighboring pixels of an image, thus forming contiguous patches:



For this purpose, the estimators use a 'connectivity' matrix, giving which samples are connected.

The function *img_to_graph* returns such a matrix from a 2D or 3D image. Similarly, *grid_to_graph* build a connectivity matrix for images given the shape of these image.

These matrices can be used to impose connectivity in estimators that use connectivity information, such as Ward clustering (*Hierarchical clustering*), but also to build precomputed kernels, or similarity matrices.

Note: Examples

- *A demo of structured Ward hierarchical clustering on an image of coins*
- *Spectral clustering for image segmentation*
- *Feature agglomeration vs. univariate selection*

4.6.3 Preprocessing data

The `sklearn.preprocessing` package provides several common utility functions and transformer classes to change raw feature vectors into a representation that is more suitable for the downstream estimators.

In general, learning algorithms benefit from standardization of the data set. If some outliers are present in the set, robust scalers or transformers are more appropriate. The behaviors of the different scalers, transformers, and normalizers on a dataset containing marginal outliers is highlighted in *Compare the effect of different scalers on data with outliers*.

Standardization, or mean removal and variance scaling

Standardization of datasets is a **common requirement for many machine learning estimators** implemented in scikit-learn; they might behave badly if the individual features do not more or less look like standard normally distributed data: Gaussian with **zero mean and unit variance**.

In practice we often ignore the shape of the distribution and just transform the data to center it by removing the mean value of each feature, then scale it by dividing non-constant features by their standard deviation.

For instance, many elements used in the objective function of a learning algorithm (such as the RBF kernel of Support Vector Machines or the l1 and l2 regularizers of linear models) assume that all features are centered around zero and have variance in the same order. If a feature has a variance that is orders of magnitude larger than others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

The function *scale* provides a quick and easy way to perform this operation on a single array-like dataset:

```
>>> from sklearn import preprocessing
>>> import numpy as np
>>> X_train = np.array([[ 1., -1.,  2.],
...                     [ 2.,  0.,  0.],
...                     [ 0.,  1., -1.]])
>>> X_scaled = preprocessing.scale(X_train)

>>> X_scaled
array([[ 0.  ..., -1.22...,  1.33...],
       [ 1.22...,  0.  ..., -0.26...],
       [-1.22...,  1.22..., -1.06...]])
```

Scaled data has zero mean and unit variance:

```
>>> X_scaled.mean(axis=0)
array([0., 0., 0.])

>>> X_scaled.std(axis=0)
array([1., 1., 1.])
```

The `preprocessing` module further provides a utility class *StandardScaler* that implements the `Transformer` API to compute the mean and standard deviation on a training set so as to be able to later reapply the same transformation on the testing set. This class is hence suitable for use in the early steps of a *sklearn.pipeline.Pipeline*:

```
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> scaler
StandardScaler()

>>> scaler.mean_
array([1. ..., 0. ..., 0.33...])

>>> scaler.scale_
array([0.81..., 0.81..., 1.24...])

>>> scaler.transform(X_train)
array([[ 0.  ..., -1.22...,  1.33...],
       [ 1.22...,  0.  ..., -0.26...],
       [-1.22...,  1.22..., -1.06...]])
```

The scaler instance can then be used on new data to transform it the same way it did on the training set:

```
>>> X_test = [[-1., 1., 0.]]
>>> scaler.transform(X_test)
array([[-2.44...,  1.22..., -0.26...]])
```

It is possible to disable either centering or scaling by either passing `with_mean=False` or `with_std=False` to the constructor of *StandardScaler*.

Scaling features to a range

An alternative standardization is scaling features to lie between a given minimum and maximum value, often between zero and one, or so that the maximum absolute value of each feature is scaled to unit size. This can be achieved using *MinMaxScaler* or *MaxAbsScaler*, respectively.

The motivation to use this scaling include robustness to very small standard deviations of features and preserving zero entries in sparse data.

Here is an example to scale a toy data matrix to the `[0, 1]` range:

```
>>> X_train = np.array([[ 1., -1.,  2.],
...                     [ 2.,  0.,  0.],
...                     [ 0.,  1., -1.]])
...
>>> min_max_scaler = preprocessing.MinMaxScaler()
>>> X_train_minmax = min_max_scaler.fit_transform(X_train)
>>> X_train_minmax
array([[0.5       , 0.        , 1.        ],
       [1.        , 0.5       , 0.33333333],
       [0.        , 1.        , 0.        ]])
```

The same instance of the transformer can then be applied to some new test data unseen during the fit call: the same scaling and shifting operations will be applied to be consistent with the transformation performed on the train data:

```
>>> X_test = np.array([[-3., -1.,  4.]])
>>> X_test_minmax = min_max_scaler.transform(X_test)
```

(continues on next page)

(continued from previous page)

```
>>> X_test_minmax
array([[-1.5       ,  0.        ,  1.66666667]])
```

It is possible to introspect the scaler attributes to find about the exact nature of the transformation learned on the training data:

```
>>> min_max_scaler.scale_
array([0.5       , 0.5       , 0.33...])

>>> min_max_scaler.min_
array([0.        , 0.5       , 0.33...])
```

If *MinMaxScaler* is given an explicit `feature_range=(min, max)` the full formula is:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))

X_scaled = X_std * (max - min) + min
```

MaxAbsScaler works in a very similar fashion, but scales in a way that the training data lies within the range `[-1, 1]` by dividing through the largest maximum value in each feature. It is meant for data that is already centered at zero or sparse data.

Here is how to use the toy data from the previous example with this scaler:

```
>>> X_train = np.array([[ 1., -1.,  2.],
...                     [ 2.,  0.,  0.],
...                     [ 0.,  1., -1.]])
...
>>> max_abs_scaler = preprocessing.MaxAbsScaler()
>>> X_train_maxabs = max_abs_scaler.fit_transform(X_train)
>>> X_train_maxabs
array([[ 0.5, -1. ,  1. ],
       [ 1. ,  0. ,  0. ],
       [ 0. ,  1. , -0.5]])
>>> X_test = np.array([[ -3., -1.,  4.]])
>>> X_test_maxabs = max_abs_scaler.transform(X_test)
>>> X_test_maxabs
array([[-1.5, -1. ,  2. ]])
>>> max_abs_scaler.scale_
array([2.,  1.,  2.])
```

As with *scale*, the module further provides convenience functions *minmax_scale* and *maxabs_scale* if you don't want to create an object.

Scaling sparse data

Centering sparse data would destroy the sparseness structure in the data, and thus rarely is a sensible thing to do. However, it can make sense to scale sparse inputs, especially if features are on different scales.

MaxAbsScaler and *maxabs_scale* were specifically designed for scaling sparse data, and are the recommended way to go about this. However, *scale* and *StandardScaler* can accept `scipy.sparse` matrices as input, as long as `with_mean=False` is explicitly passed to the constructor. Otherwise a `ValueError` will be raised as silently centering would break the sparsity and would often crash the execution by allocating excessive amounts of memory unintentionally. *RobustScaler* cannot be fitted to sparse inputs, but you can use the `transform` method on sparse inputs.

Note that the scalers accept both Compressed Sparse Rows and Compressed Sparse Columns format (see `scipy.sparse.csr_matrix` and `scipy.sparse.csc_matrix`). Any other sparse input will be **converted to the Compressed Sparse Rows representation**. To avoid unnecessary memory copies, it is recommended to choose the CSR or CSC representation upstream.

Finally, if the centered data is expected to be small enough, explicitly converting the input to an array using the `toarray` method of sparse matrices is another option.

Scaling data with outliers

If your data contains many outliers, scaling using the mean and variance of the data is likely to not work very well. In these cases, you can use *`robust_scale`* and *`RobustScaler`* as drop-in replacements instead. They use more robust estimates for the center and range of your data.

References:

Further discussion on the importance of centering and scaling data is available on this FAQ: Should I normalize/standardize/rescale the data?

Scaling vs Whitening

It is sometimes not enough to center and scale the features independently, since a downstream model can further make some assumption on the linear independence of the features.

To address this issue you can use *`sklearn.decomposition.PCA`* with `whiten=True` to further remove the linear correlation across features.

Scaling a 1D array

All above functions (i.e. *`scale`*, *`minmax_scale`*, *`maxabs_scale`*, and *`robust_scale`*) accept 1D array which can be useful in some specific case.

Centering kernel matrices

If you have a kernel matrix of a kernel K that computes a dot product in a feature space defined by function ϕ, a *`KernelCenterer`* can transform the kernel matrix so that it contains inner products in the feature space defined by ϕ followed by removal of the mean in that space.

Non-linear transformation

Two types of transformations are available: quantile transforms and power transforms. Both quantile and power transforms are based on monotonic transformations of the features and thus preserve the rank of the values along each feature.

Quantile transforms put all features into the same desired distribution based on the formula $G^{-1}(F(X))$ where F is the cumulative distribution function of the feature and G^{-1} the quantile function of the desired output distribution G. This formula is using the two following facts: (i) if X is a random variable with a continuous cumulative distribution function F then $F(X)$ is uniformly distributed on $[0, 1]$; (ii) if U is a random variable with uniform distribution on $[0, 1]$ then $G^{-1}(U)$ has distribution G. By performing a rank transformation, a quantile transform smooths out unusual

distributions and is less influenced by outliers than scaling methods. It does, however, distort correlations and distances within and across features.

Power transforms are a family of parametric transformations that aim to map data from any distribution to as close to a Gaussian distribution.

Mapping to a Uniform distribution

QuantileTransformer and *quantile_transform* provide a non-parametric transformation to map the data to a uniform distribution with values between 0 and 1:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
>>> quantile_transformer = preprocessing.QuantileTransformer(random_state=0)
>>> X_train_trans = quantile_transformer.fit_transform(X_train)
>>> X_test_trans = quantile_transformer.transform(X_test)
>>> np.percentile(X_train[:, 0], [0, 25, 50, 75, 100])
array([ 4.3,  5.1,  5.8,  6.5,  7.9])
```

This feature corresponds to the sepal length in cm. Once the quantile transformation applied, those landmarks approach closely the percentiles previously defined:

```
>>> np.percentile(X_train_trans[:, 0], [0, 25, 50, 75, 100])
...
array([ 0.00... ,  0.24...,  0.49...,  0.73...,  0.99... ])
```

This can be confirmed on a independent testing set with similar remarks:

```
>>> np.percentile(X_test[:, 0], [0, 25, 50, 75, 100])
...
array([ 4.4  ,  5.125,  5.75 ,  6.175,  7.3  ])
>>> np.percentile(X_test_trans[:, 0], [0, 25, 50, 75, 100])
...
array([ 0.01...,  0.25...,  0.46...,  0.60... ,  0.94...])
```

Mapping to a Gaussian distribution

In many modeling scenarios, normality of the features in a dataset is desirable. Power transforms are a family of parametric, monotonic transformations that aim to map data from any distribution to as close to a Gaussian distribution as possible in order to stabilize variance and minimize skewness.

PowerTransformer currently provides two such power transformations, the Yeo-Johnson transform and the Box-Cox transform.

The Yeo-Johnson transform is given by:

$$x_i^{(\lambda)} = \begin{cases} [(x_i + 1)^\lambda - 1]/\lambda & \text{if } \lambda \neq 0, x_i \geq 0, \\ \ln(x_i + 1) & \text{if } \lambda = 0, x_i \geq 0 \\ -[(-x_i + 1)^{2-\lambda} - 1]/(2 - \lambda) & \text{if } \lambda \neq 2, x_i < 0, \\ -\ln(-x_i + 1) & \text{if } \lambda = 2, x_i < 0 \end{cases}$$

while the Box-Cox transform is given by:

$$x_i^{(\lambda)} = \begin{cases} \dfrac{x_i^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0, \\ \ln(x_i) & \text{if } \lambda = 0, \end{cases}$$

Box-Cox can only be applied to strictly positive data. In both methods, the transformation is parameterized by λ, which is determined through maximum likelihood estimation. Here is an example of using Box-Cox to map samples drawn from a lognormal distribution to a normal distribution:

```
>>> pt = preprocessing.PowerTransformer(method='box-cox', standardize=False)
>>> X_lognormal = np.random.RandomState(616).lognormal(size=(3, 3))
>>> X_lognormal
array([[1.28..., 1.18..., 0.84...],
       [0.94..., 1.60..., 0.38...],
       [1.35..., 0.21..., 1.09...]])
>>> pt.fit_transform(X_lognormal)
array([[ 0.49...,  0.17..., -0.15...],
       [-0.05...,  0.58..., -0.57...],
       [ 0.69..., -0.84...,  0.10...]])
```

While the above example sets the `standardize` option to `False`, *`PowerTransformer`* will apply zero-mean, unit-variance normalization to the transformed output by default.

Below are examples of Box-Cox and Yeo-Johnson applied to various probability distributions. Note that when applied to certain distributions, the power transforms achieve very Gaussian-like results, but with others, they are ineffective. This highlights the importance of visualizing the data before and after transformation.

It is also possible to map data to a normal distribution using *`QuantileTransformer`* by setting `output_distribution='normal'`. Using the earlier example with the iris dataset:

```
>>> quantile_transformer = preprocessing.QuantileTransformer(
...     output_distribution='normal', random_state=0)
>>> X_trans = quantile_transformer.fit_transform(X)
>>> quantile_transformer.quantiles_
array([[4.3, 2. , 1. , 0.1],
       [4.4, 2.2, 1.1, 0.1],
       [4.4, 2.2, 1.2, 0.1],
       ...,
       [7.7, 4.1, 6.7, 2.5],
       [7.7, 4.2, 6.7, 2.5],
       [7.9, 4.4, 6.9, 2.5]])
```

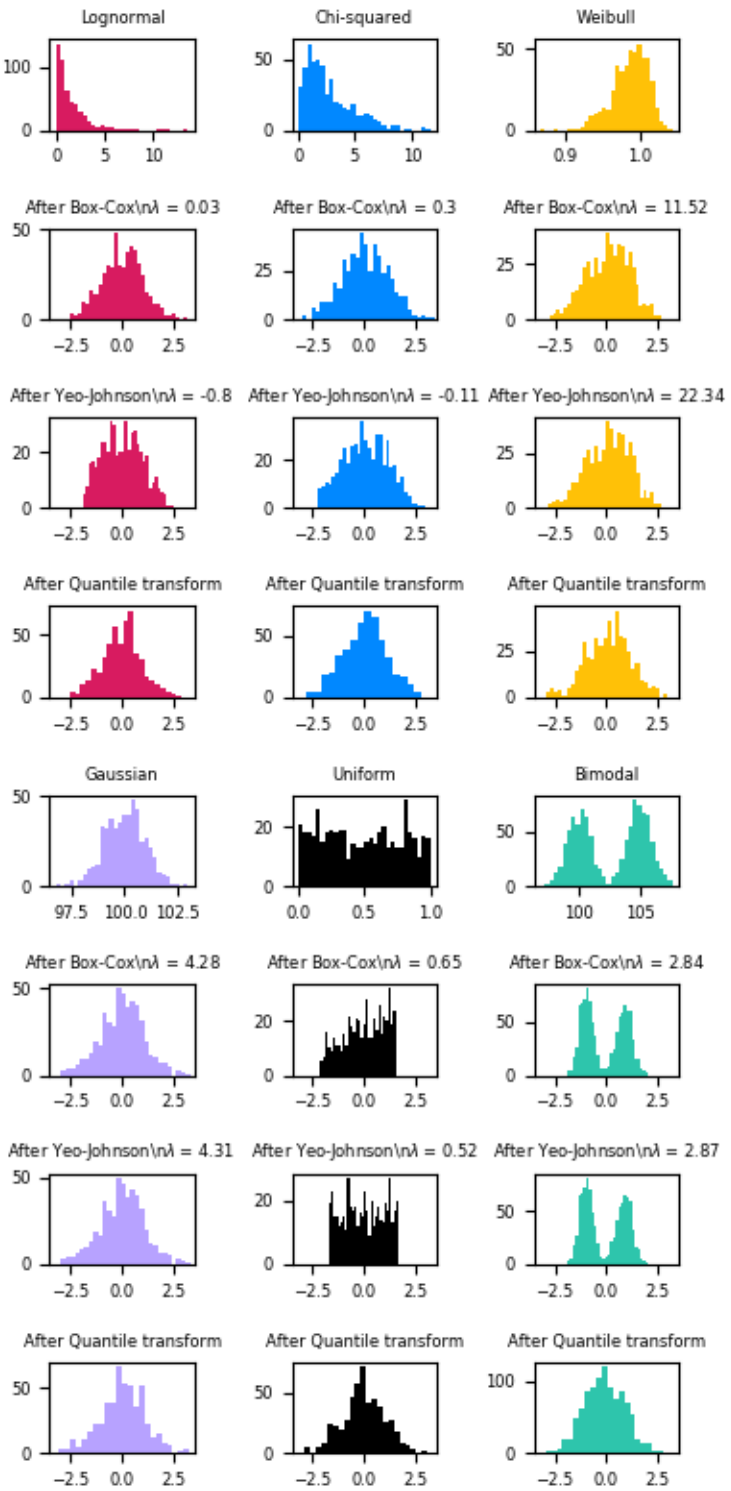
Thus the median of the input becomes the mean of the output, centered at 0. The normal output is clipped so that the input's minimum and maximum — corresponding to the 1e-7 and 1 - 1e-7 quantiles respectively — do not become infinite under the transformation.

Normalization

Normalization is the process of **scaling individual samples to have unit norm**. This process can be useful if you plan to use a quadratic form such as the dot-product or any other kernel to quantify the similarity of any pair of samples.

This assumption is the base of the Vector Space Model often used in text classification and clustering contexts.

The function *`normalize`* provides a quick and easy way to perform this operation on a single array-like dataset, either using the `l1` or `l2` norms:


Lognormal
Chi-squared
Weibull
After Box-Cox\nλ = 0.03
After Box-Cox\nλ = 0.3
After Box-Cox\nλ = 11.52
After Yeo-Johnson\nλ = -0.8
After Yeo-Johnson\nλ = -0.11
After Yeo-Johnson\nλ = 22.34
After Quantile transform
After Quantile transform
After Quantile transform
Gaussian
Uniform
Bimodal
After Box-Cox\nλ = 4.28
After Box-Cox\nλ = 0.65
After Box-Cox\nλ = 2.84
After Yeo-Johnson\nλ = 4.31
After Yeo-Johnson\nλ = 0.52
After Yeo-Johnson\nλ = 2.87
After Quantile transform
After Quantile transform
After Quantile transform

```
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]
>>> X_normalized = preprocessing.normalize(X, norm='l2')

>>> X_normalized
array([[ 0.40..., -0.40...,  0.81...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 0.  ...,  0.70..., -0.70...]])
```

The `preprocessing` module further provides a utility class *`Normalizer`* that implements the same operation using the `Transformer` API (even though the `fit` method is useless in this case: the class is stateless as this operation treats samples independently).

This class is hence suitable for use in the early steps of a *`sklearn.pipeline.Pipeline`*:

```
>>> normalizer = preprocessing.Normalizer().fit(X)  # fit does nothing
>>> normalizer
Normalizer()
```

The normalizer instance can then be used on sample vectors as any transformer:

```
>>> normalizer.transform(X)
array([[ 0.40..., -0.40...,  0.81...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 0.  ...,  0.70..., -0.70...]])

>>> normalizer.transform([[-1.,  1., 0.]])
array([[-0.70...,  0.70...,  0.  ...]])
```

Note: L2 normalization is also known as spatial sign preprocessing.

Sparse input

`normalize` and *`Normalizer`* accept **both dense array-like and sparse matrices from scipy.sparse as input**.

For sparse input the data is **converted to the Compressed Sparse Rows representation** (see `scipy.sparse.csr_matrix`) before being fed to efficient Cython routines. To avoid unnecessary memory copies, it is recommended to choose the CSR representation upstream.

Encoding categorical features

Often features are not given as continuous values but categorical. For example a person could have features `["male", "female"]`, `["from Europe", "from US", "from Asia"]`, `["uses Firefox", "uses Chrome", "uses Safari", "uses Internet Explorer"]`. Such features can be efficiently coded as integers, for instance `["male", "from US", "uses Internet Explorer"]` could be expressed as `[0, 1, 3]` while `["female", "from Asia", "uses Chrome"]` would be `[1, 2, 1]`.

To convert categorical features to such integer codes, we can use the *`OrdinalEncoder`*. This estimator transforms each categorical feature to one new feature of integers (0 to n_categories - 1):

```
>>> enc = preprocessing.OrdinalEncoder()
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox
↪']]
```

(continues on next page)

(continued from previous page)

```
>>> enc.fit(X)
OrdinalEncoder()
>>> enc.transform([['female', 'from US', 'uses Safari']])
array([[0., 1., 1.]])
```

Such integer representation can, however, not be used directly with all scikit-learn estimators, as these expect continuous input, and would interpret the categories as being ordered, which is often not desired (i.e. the set of browsers was ordered arbitrarily).

Another possibility to convert categorical features to features that can be used with scikit-learn estimators is to use a one-of-K, also known as one-hot or dummy encoding. This type of encoding can be obtained with the *OneHotEncoder*, which transforms each categorical feature with `n_categories` possible values into `n_categories` binary features, with one of them 1, and all others 0.

Continuing the example above:

```
>>> enc = preprocessing.OneHotEncoder()
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox
↪']]
>>> enc.fit(X)
OneHotEncoder()
>>> enc.transform([['female', 'from US', 'uses Safari'],
...                ['male', 'from Europe', 'uses Safari']]).toarray()
array([[1., 0., 0., 1., 0., 1.],
       [0., 1., 1., 0., 0., 1.]])
```

By default, the values each feature can take is inferred automatically from the dataset and can be found in the `categories_` attribute:

```
>>> enc.categories_
[array(['female', 'male'], dtype=object), array(['from Europe', 'from US'],␣
↪dtype=object), array(['uses Firefox', 'uses Safari'], dtype=object)]
```

It is possible to specify this explicitly using the parameter `categories`. There are two genders, four possible continents and four web browsers in our dataset:

```
>>> genders = ['female', 'male']
>>> locations = ['from Africa', 'from Asia', 'from Europe', 'from US']
>>> browsers = ['uses Chrome', 'uses Firefox', 'uses IE', 'uses Safari']
>>> enc = preprocessing.OneHotEncoder(categories=[genders, locations, browsers])
>>> # Note that for there are missing categorical values for the 2nd and 3rd
>>> # feature
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox
↪']]
>>> enc.fit(X)
OneHotEncoder(categories=[['female', 'male'],
                          ['from Africa', 'from Asia', 'from Europe',
                           'from US'],
                          ['uses Chrome', 'uses Firefox', 'uses IE',
                           'uses Safari']])
>>> enc.transform([['female', 'from Asia', 'uses Chrome']]).toarray()
array([[1., 0., 0., 1., 0., 0., 1., 0., 0., 0.]])
```

If there is a possibility that the training data might have missing categorical features, it can often be better to specify `handle_unknown='ignore'` instead of setting the `categories` manually as above. When `handle_unknown='ignore'` is specified and unknown categories are encountered during transform, no error will be raised but the resulting one-hot encoded columns for this feature will be all zeros

(handle_unknown='ignore' is only supported for one-hot encoding):

```
>>> enc = preprocessing.OneHotEncoder(handle_unknown='ignore')
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox
↪']]
>>> enc.fit(X)
OneHotEncoder(handle_unknown='ignore')
>>> enc.transform([['female', 'from Asia', 'uses Chrome']]).toarray()
array([[1., 0., 0., 0., 0., 0.]])
```

It is also possible to encode each column into n_categories - 1 columns instead of n_categories columns by using the drop parameter. This parameter allows the user to specify a category for each feature to be dropped. This is useful to avoid co-linearity in the input matrix in some classifiers. Such functionality is useful, for example, when using non-regularized regression (*LinearRegression*), since co-linearity would cause the covariance matrix to be non-invertible. When this paramenter is not None, handle_unknown must be set to error:

```
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox
↪']]
>>> drop_enc = preprocessing.OneHotEncoder(drop='first').fit(X)
>>> drop_enc.categories_
[array(['female', 'male'], dtype=object), array(['from Europe', 'from US'],␣
↪dtype=object), array(['uses Firefox', 'uses Safari'], dtype=object)]
>>> drop_enc.transform(X).toarray()
array([[1., 1., 1.],
       [0., 0., 0.]])
```

See *Loading features from dicts* for categorical features that are represented as a dict, not as scalars.

Discretization

Discretization (otherwise known as quantization or binning) provides a way to partition continuous features into discrete values. Certain datasets with continuous features may benefit from discretization, because discretization can transform the dataset of continuous attributes to one with only nominal attributes.

One-hot encoded discretized features can make a model more expressive, while maintaining interpretability. For instance, pre-processing with a discretizer can introduce nonlinearity to linear models.

K-bins discretization

KBinsDiscretizer discretizes features into k bins:

```
>>> X = np.array([[ -3., 5., 15 ],
...               [  0., 6., 14 ],
...               [  6., 3., 11 ]])
>>> est = preprocessing.KBinsDiscretizer(n_bins=[3, 2, 2], encode='ordinal').fit(X)
```

By default the output is one-hot encoded into a sparse matrix (See *Encoding categorical features*) and this can be configured with the encode parameter. For each feature, the bin edges are computed during fit and together with the number of bins, they will define the intervals. Therefore, for the current example, these intervals are defined as:

- feature 1: $[-\infty, -1), [-1, 2), [2, \infty)$
- feature 2: $[-\infty, 5), [5, \infty)$
- feature 3: $[-\infty, 14), [14, \infty)$

Based on these bin intervals, X is transformed as follows:

```
>>> est.transform(X)
array([[ 0., 1., 1.],
       [ 1., 1., 1.],
       [ 2., 0., 0.]])
```

The resulting dataset contains ordinal attributes which can be further used in a *sklearn.pipeline.Pipeline*.

Discretization is similar to constructing histograms for continuous data. However, histograms focus on counting features which fall into particular bins, whereas discretization focuses on assigning feature values to these bins.

KBinsDiscretizer implements different binning strategies, which can be selected with the `strategy` parameter. The 'uniform' strategy uses constant-width bins. The 'quantile' strategy uses the quantiles values to have equally populated bins in each feature. The 'kmeans' strategy defines bins based on a k-means clustering procedure performed on each feature independently.

Examples:

- *Using KBinsDiscretizer to discretize continuous features*
- *Feature discretization*
- *Demonstrating the different strategies of KBinsDiscretizer*

Feature binarization

Feature binarization is the process of **thresholding numerical features to get boolean values**. This can be useful for downstream probabilistic estimators that make assumption that the input data is distributed according to a multi-variate Bernoulli distribution. For instance, this is the case for the *sklearn.neural_network.BernoulliRBM*.

It is also common among the text processing community to use binary feature values (probably to simplify the probabilistic reasoning) even if normalized counts (a.k.a. term frequencies) or TF-IDF valued features often perform slightly better in practice.

As for the *Normalizer*, the utility class *Binarizer* is meant to be used in the early stages of *sklearn.pipeline.Pipeline*. The `fit` method does nothing as each sample is treated independently of others:

```
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]

>>> binarizer = preprocessing.Binarizer().fit(X)  # fit does nothing
>>> binarizer
Binarizer()

>>> binarizer.transform(X)
array([[1., 0., 1.],
       [1., 0., 0.],
       [0., 1., 0.]])
```

It is possible to adjust the threshold of the binarizer:

```
>>> binarizer = preprocessing.Binarizer(threshold=1.1)
>>> binarizer.transform(X)
array([[0., 0., 1.],
       [1., 0., 0.],
       [0., 0., 0.]])
```

As for the *StandardScaler* and *Normalizer* classes, the preprocessing module provides a companion function *binarize* to be used when the transformer API is not necessary.

Note that the *Binarizer* is similar to the *KBinsDiscretizer* when `k = 2`, and when the bin edge is at the value `threshold`.

Sparse input

binarize and *Binarizer* accept **both dense array-like and sparse matrices from scipy.sparse as input**.

For sparse input the data is **converted to the Compressed Sparse Rows representation** (see `scipy.sparse.csr_matrix`). To avoid unnecessary memory copies, it is recommended to choose the CSR representation upstream.

Imputation of missing values

Tools for imputing missing values are discussed at *Imputation of missing values*.

Generating polynomial features

Often it's useful to add complexity to the model by considering nonlinear features of the input data. A simple and common method to use is polynomial features, which can get features' high-order and interaction terms. It is implemented in *PolynomialFeatures*:

```
>>> import numpy as np
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> poly = PolynomialFeatures(2)
>>> poly.fit_transform(X)
array([[ 1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  2.,  3.,  4.,  6.,  9.],
       [ 1.,  4.,  5., 16., 20., 25.]])
```

The features of X have been transformed from (X_1, X_2) to $(1, X_1, X_2, X_1^2, X_1X_2, X_2^2)$.

In some cases, only interaction terms among features are required, and it can be gotten with the setting `interaction_only=True`:

```
>>> X = np.arange(9).reshape(3, 3)
>>> X
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> poly = PolynomialFeatures(degree=3, interaction_only=True)
>>> poly.fit_transform(X)
array([[  1.,   0.,   1.,   2.,   0.,   0.,   2.,   0.],
       [  1.,   3.,   4.,   5.,  12.,  15.,  20.,  60.],
       [  1.,   6.,   7.,   8.,  42.,  48.,  56., 336.]])
```

The features of X have been transformed from (X_1, X_2, X_3) to $(1, X_1, X_2, X_3, X_1X_2, X_1X_3, X_2X_3, X_1X_2X_3)$.

Note that polynomial features are used implicitly in kernel methods (e.g., *sklearn.svm.SVC*, *sklearn.decomposition.KernelPCA*) when using polynomial *Kernel functions*.

See *Polynomial interpolation* for Ridge regression using created polynomial features.

Custom transformers

Often, you will want to convert an existing Python function into a transformer to assist in data cleaning or processing. You can implement a transformer from an arbitrary function with *FunctionTransformer*. For example, to build a transformer that applies a log transformation in a pipeline, do:

```
>>> import numpy as np
>>> from sklearn.preprocessing import FunctionTransformer
>>> transformer = FunctionTransformer(np.log1p, validate=True)
>>> X = np.array([[0, 1], [2, 3]])
>>> transformer.transform(X)
array([[0.        , 0.69314718],
       [1.09861229, 1.38629436]])
```

You can ensure that `func` and `inverse_func` are the inverse of each other by setting `check_inverse=True` and calling `fit` before `transform`. Please note that a warning is raised and can be turned into an error with a `filterwarnings`:

```
>>> import warnings
>>> warnings.filterwarnings("error", message=".*check_inverse*.",
...                         category=UserWarning, append=False)
```

For a full code example that demonstrates using a *FunctionTransformer* to do custom feature selection, see *Using FunctionTransformer to select columns*

4.6.4 Imputation of missing values

For various reasons, many real world datasets contain missing values, often encoded as blanks, NaNs or other placeholders. Such datasets however are incompatible with scikit-learn estimators which assume that all values in an array are numerical, and that all have and hold meaning. A basic strategy to use incomplete datasets is to discard entire rows and/or columns containing missing values. However, this comes at the price of losing data which may be valuable (even though incomplete). A better strategy is to impute the missing values, i.e., to infer them from the known part of the data. See the *Glossary of Common Terms and API Elements* entry on imputation.

Univariate vs. Multivariate Imputation

One type of imputation algorithm is univariate, which imputes values in the i-th feature dimension using only non-missing values in that feature dimension (e.g. `impute.SimpleImputer`). By contrast, multivariate imputation algorithms use the entire set of available feature dimensions to estimate the missing values (e.g. `impute.IterativeImputer`).

Univariate feature imputation

The *SimpleImputer* class provides basic strategies for imputing missing values. Missing values can be imputed with a provided constant value, or using the statistics (mean, median or most frequent) of each column in which the missing values are located. This class also allows for different missing values encodings.

The following snippet demonstrates how to replace missing values, encoded as `np.nan`, using the mean value of the columns (axis 0) that contain the missing values:

```
>>> import numpy as np
>>> from sklearn.impute import SimpleImputer
>>> imp = SimpleImputer(missing_values=np.nan, strategy='mean')
>>> imp.fit([[1, 2], [np.nan, 3], [7, 6]])
SimpleImputer()
>>> X = [[np.nan, 2], [6, np.nan], [7, 6]]
>>> print(imp.transform(X))
[[4.          2.        ]
 [6.          3.666...]
 [7.          6.        ]]
```

The *SimpleImputer* class also supports sparse matrices:

```
>>> import scipy.sparse as sp
>>> X = sp.csc_matrix([[1, 2], [0, -1], [8, 4]])
>>> imp = SimpleImputer(missing_values=-1, strategy='mean')
>>> imp.fit(X)
SimpleImputer(missing_values=-1)
>>> X_test = sp.csc_matrix([[-1, 2], [6, -1], [7, 6]])
>>> print(imp.transform(X_test).toarray())
[[3. 2.]
 [6. 3.]
 [7. 6.]]
```

Note that this format is not meant to be used to implicitly store missing values in the matrix because it would densify it at transform time. Missing values encoded by 0 must be used with dense input.

The *SimpleImputer* class also supports categorical data represented as string values or pandas categoricals when using the `'most_frequent'` or `'constant'` strategy:

```
>>> import pandas as pd
>>> df = pd.DataFrame([["a", "x"],
...                    [np.nan, "y"],
...                    ["a", np.nan],
...                    ["b", "y"]], dtype="category")
...
>>> imp = SimpleImputer(strategy="most_frequent")
>>> print(imp.fit_transform(df))
[['a' 'x']
 ['a' 'y']
 ['a' 'y']
 ['b' 'y']]
```

Multivariate feature imputation

A more sophisticated approach is to use the *IterativeImputer* class, which models each feature with missing values as a function of other features, and uses that estimate for imputation. It does so in an iterated round-robin fashion: at each step, a feature column is designated as output `y` and the other feature columns are treated as inputs `X`. A regressor is fit on `(X, y)` for known `y`. Then, the regressor is used to predict the missing values of `y`. This is done for each feature in an iterative fashion, and then is repeated for `max_iter` imputation rounds. The results of the final imputation round are returned.

Note: This estimator is still **experimental** for now: the predictions and the API might change without any deprecation

cycle. To use it, you need to explicitly import `enable_iterative_imputer`.

```
>>> import numpy as np
>>> from sklearn.experimental import enable_iterative_imputer
>>> from sklearn.impute import IterativeImputer
>>> imp = IterativeImputer(max_iter=10, random_state=0)
>>> imp.fit([[1, 2], [3, 6], [4, 8], [np.nan, 3], [7, np.nan]])
IterativeImputer(random_state=0)
>>> X_test = [[np.nan, 2], [6, np.nan], [np.nan, 6]]
>>> # the model learns that the second feature is double the first
>>> print(np.round(imp.transform(X_test)))
[[ 1.  2.]
 [ 6. 12.]
 [ 3.  6.]]
```

Both *SimpleImputer* and *IterativeImputer* can be used in a Pipeline as a way to build a composite estimator that supports imputation. See *Imputing missing values before building an estimator*.

Flexibility of IterativeImputer

There are many well-established imputation packages in the R data science ecosystem: Amelia, mi, mice, missForest, etc. missForest is popular, and turns out to be a particular instance of different sequential imputation algorithms that can all be implemented with *IterativeImputer* by passing in different regressors to be used for predicting missing feature values. In the case of missForest, this regressor is a Random Forest. See *Imputing missing values with variants of IterativeImputer*.

Multiple vs. Single Imputation

In the statistics community, it is common practice to perform multiple imputations, generating, for example, `m` separate imputations for a single feature matrix. Each of these `m` imputations is then put through the subsequent analysis pipeline (e.g. feature engineering, clustering, regression, classification). The `m` final analysis results (e.g. held-out validation errors) allow the data scientist to obtain understanding of how analytic results may differ as a consequence of the inherent uncertainty caused by the missing values. The above practice is called multiple imputation.

Our implementation of *IterativeImputer* was inspired by the R MICE package (Multivariate Imputation by Chained Equations)[1], but differs from it by returning a single imputation instead of multiple imputations. However, *IterativeImputer* can also be used for multiple imputations by applying it repeatedly to the same dataset with different random seeds when `sample_posterior=True`. See[2], chapter 4 for more discussion on multiple vs. single imputations.

It is still an open problem as to how useful single vs. multiple imputation is in the context of prediction and classification when the user is not interested in measuring uncertainty due to missing values.

Note that a call to the `transform` method of *IterativeImputer* is not allowed to change the number of samples. Therefore multiple imputations cannot be achieved by a single call to `transform`.

References

[1] Stef van Buuren, Karin Groothuis-Oudshoorn (2011). "mice: Multivariate Imputation by Chained Equations in R". Journal of Statistical Software 45: 1-67.

[2] Roderick J A Little and Donald B Rubin (1986). "Statistical Analysis with Missing Data". John Wiley & Sons, Inc., New York, NY, USA.

Nearest neighbors imputation

The *KNNImputer* class provides imputation for filling in missing values using the k-Nearest Neighbors approach. By default, a euclidean distance metric that supports missing values, `nan_euclidean_distances`, is used to find the nearest neighbors. Each missing feature is imputed using values from `n_neighbors` nearest neighbors that have a value for the feature. The feature of the neighbors are averaged uniformly or weighted by distance to each neighbor. If a sample has more than one feature missing, then the neighbors for that sample can be different depending on the particular feature being imputed. When the number of available neighbors is less than `n_neighbors` and there are no defined distances to the training set, the training set average for that feature is used during imputation. If there is at least one neighbor with a defined distance, the weighted or unweighted average of the remaining neighbors will be used during imputation. If a feature is always missing in training, it is removed during `transform`. For more information on the methodology, see ref. [OL2001].

The following snippet demonstrates how to replace missing values, encoded as `np.nan`, using the mean feature value of the two nearest neighbors of samples with missing values:

```
>>> import numpy as np
>>> from sklearn.impute import KNNImputer
>>> nan = np.nan
>>> X = [[1, 2, nan], [3, 4, 3], [nan, 6, 5], [8, 8, 7]]
>>> imputer = KNNImputer(n_neighbors=2, weights="uniform")
>>> imputer.fit_transform(X)
array([[1. , 2. , 4. ],
       [3. , 4. , 3. ],
       [5.5, 6. , 5. ],
       [8. , 8. , 7. ]])
```

Marking imputed values

The *MissingIndicator* transformer is useful to transform a dataset into corresponding binary matrix indicating the presence of missing values in the dataset. This transformation is useful in conjunction with imputation. When using imputation, preserving the information about which values had been missing can be informative. Note that both the *SimpleImputer* and *IterativeImputer* have the boolean parameter `add_indicator` (`False` by default) which when set to `True` provides a convenient way of stacking the output of the *MissingIndicator* transformer with the output of the imputer.

`NaN` is usually used as the placeholder for missing values. However, it enforces the data type to be float. The parameter `missing_values` allows to specify other placeholder such as integer. In the following example, we will use `-1` as missing values:

```
>>> from sklearn.impute import MissingIndicator
>>> X = np.array([[-1, -1, 1, 3],
...               [4, -1, 0, -1],
...               [8, -1, 1, 0]])
>>> indicator = MissingIndicator(missing_values=-1)
>>> mask_missing_values_only = indicator.fit_transform(X)
>>> mask_missing_values_only
array([[ True,  True, False],
       [False,  True,  True],
       [False,  True, False]])
```

The `features` parameter is used to choose the features for which the mask is constructed. By default, it is `'missing-only'` which returns the imputer mask of the features containing missing values at `fit` time:

```
>>> indicator.features_
array([0, 1, 3])
```

The `features` parameter can be set to `'all'` to return all features whether or not they contain missing values:

```
>>> indicator = MissingIndicator(missing_values=-1, features="all")
>>> mask_all = indicator.fit_transform(X)
>>> mask_all
array([[ True,  True, False, False],
       [False,  True, False,  True],
       [False,  True, False, False]])
>>> indicator.features_
array([0, 1, 2, 3])
```

When using the *MissingIndicator* in a `Pipeline`, be sure to use the `FeatureUnion` or `ColumnTransformer` to add the indicator features to the regular features. First we obtain the `iris` dataset, and add some missing values to it.

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.impute import SimpleImputer, MissingIndicator
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.pipeline import FeatureUnion, make_pipeline
>>> from sklearn.tree import DecisionTreeClassifier
>>> X, y = load_iris(return_X_y=True)
>>> mask = np.random.randint(0, 2, size=X.shape).astype(np.bool)
>>> X[mask] = np.nan
>>> X_train, X_test, y_train, _ = train_test_split(X, y, test_size=100,
...                                                random_state=0)
```

Now we create a `FeatureUnion`. All features will be imputed using *SimpleImputer*, in order to enable classifiers to work with this data. Additionally, it adds the the indicator variables from *MissingIndicator*.

```
>>> transformer = FeatureUnion(
...     transformer_list=[
...         ('features', SimpleImputer(strategy='mean')),
...         ('indicators', MissingIndicator())])
>>> transformer = transformer.fit(X_train, y_train)
>>> results = transformer.transform(X_test)
>>> results.shape
(100, 8)
```

Of course, we cannot use the transformer to make any predictions. We should wrap this in a `Pipeline` with a classifier (e.g., a `DecisionTreeClassifier`) to be able to make predictions.

```
>>> clf = make_pipeline(transformer, DecisionTreeClassifier())
>>> clf = clf.fit(X_train, y_train)
>>> results = clf.predict(X_test)
>>> results.shape
(100,)
```

4.6.5 Unsupervised dimensionality reduction

If your number of features is high, it may be useful to reduce it with an unsupervised step prior to supervised steps. Many of the *Unsupervised learning* methods implement a `transform` method that can be used to reduce the dimensionality. Below we discuss two specific example of this pattern that are heavily used.

Pipelining

The unsupervised data reduction and the supervised estimator can be chained in one step. See *Pipeline: chaining estimators*.

PCA: principal component analysis

`decomposition.PCA` looks for a combination of features that capture well the variance of the original features. See *Decomposing signals in components (matrix factorization problems)*.

Examples

- *Faces recognition example using eigenfaces and SVMs*

Random projections

The module: `random_projection` provides several tools for data reduction by random projections. See the relevant section of the documentation: *Random Projection*.

Examples

- *The Johnson-Lindenstrauss bound for embedding with random projections*

Feature agglomeration

`cluster.FeatureAgglomeration` applies *Hierarchical clustering* to group together features that behave similarly.

Examples

- *Feature agglomeration vs. univariate selection*
- *Feature agglomeration*

Feature scaling

Note that if features have very different scaling or statistical properties, `cluster.FeatureAgglomeration` may not be able to capture the links between related features. Using a `preprocessing.StandardScaler` can be useful in these settings.

4.6.6 Random Projection

The `sklearn.random_projection` module implements a simple and computationally efficient way to reduce the dimensionality of the data by trading a controlled amount of accuracy (as additional variance) for faster processing

times and smaller model sizes. This module implements two types of unstructured random matrix: *Gaussian random matrix* and *sparse random matrix*.

The dimensions and distribution of random projections matrices are controlled so as to preserve the pairwise distances between any two samples of the dataset. Thus random projection is a suitable approximation technique for distance based method.

References:

- Sanjoy Dasgupta. 2000. Experiments with random projection. In Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence (UAI'00), Craig Boutilier and Moisés Goldszmidt (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 143-151.
- Ella Bingham and Heikki Mannila. 2001. Random projection in dimensionality reduction: applications to image and text data. In Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '01). ACM, New York, NY, USA, 245-250.

The Johnson-Lindenstrauss lemma

The main theoretical result behind the efficiency of random projection is the Johnson-Lindenstrauss lemma (quoting Wikipedia):

> In mathematics, the Johnson-Lindenstrauss lemma is a result concerning low-distortion embeddings of points from high-dimensional into low-dimensional Euclidean space. The lemma states that a small set of points in a high-dimensional space can be embedded into a space of much lower dimension in such a way that distances between the points are nearly preserved. The map used for the embedding is at least Lipschitz, and can even be taken to be an orthogonal projection.

Knowing only the number of samples, the *sklearn.random_projection.johnson_lindenstrauss_min_dim* estimates conservatively the minimal size of the random subspace to guarantee a bounded distortion introduced by the random projection:

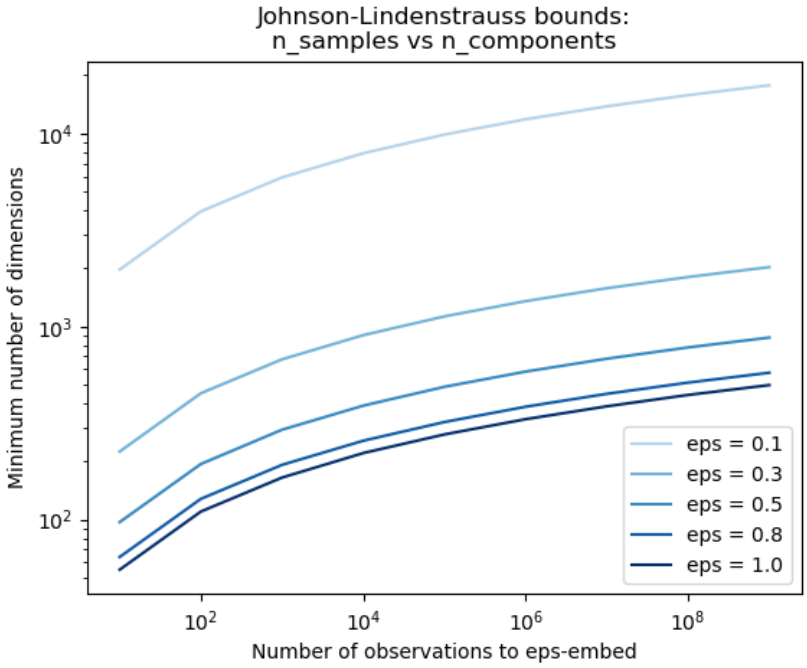
```
>>> from sklearn.random_projection import johnson_lindenstrauss_min_dim
>>> johnson_lindenstrauss_min_dim(n_samples=1e6, eps=0.5)
663
>>> johnson_lindenstrauss_min_dim(n_samples=1e6, eps=[0.5, 0.1, 0.01])
array([    663,   11841, 1112658])
>>> johnson_lindenstrauss_min_dim(n_samples=[1e4, 1e5, 1e6], eps=0.1)
array([ 7894,  9868, 11841])
```

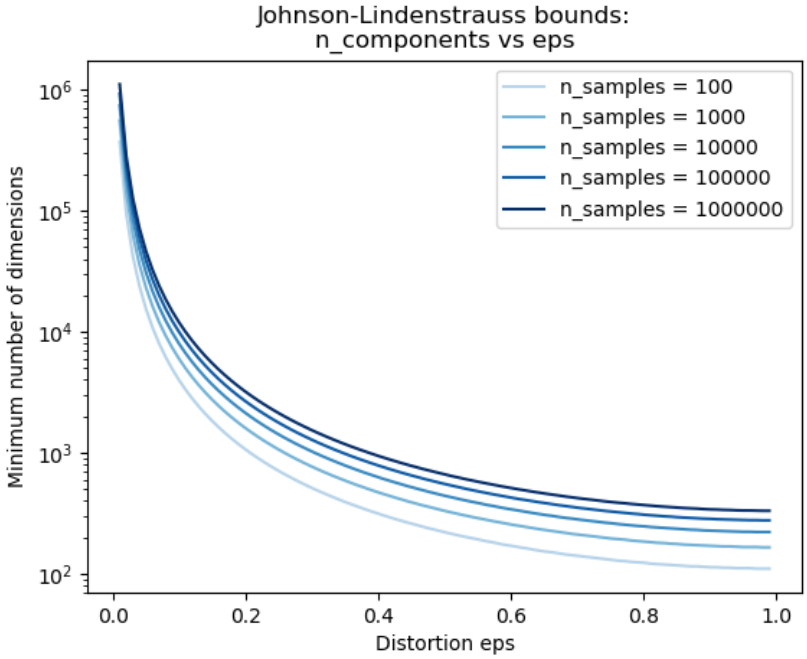
Example:

- See *The Johnson-Lindenstrauss bound for embedding with random projections* for a theoretical explication on the Johnson-Lindenstrauss lemma and an empirical validation using sparse random matrices.

References:

- Sanjoy Dasgupta and Anupam Gupta, 1999. An elementary proof of the Johnson-Lindenstrauss Lemma.


Johnson-Lindenstrauss bounds:
n_samples vs n_components
Minimum number of dimensions
Number of observations to eps-embed
eps = 0.1
eps = 0.3
eps = 0.5
eps = 0.8
eps = 1.0


Johnson-Lindenstrauss bounds:
n_components vs eps
Minimum number of dimensions
Distortion eps
n_samples = 100
n_samples = 1000
n_samples = 10000
n_samples = 100000
n_samples = 1000000

Gaussian random projection

The *sklearn.random_projection.GaussianRandomProjection* reduces the dimensionality by projecting the original input space on a randomly generated matrix where components are drawn from the following distribution $N(0, \frac{1}{n_{components}})$.

Here a small excerpt which illustrates how to use the Gaussian random projection transformer:

```
>>> import numpy as np
>>> from sklearn import random_projection
>>> X = np.random.rand(100, 10000)
>>> transformer = random_projection.GaussianRandomProjection()
>>> X_new = transformer.fit_transform(X)
>>> X_new.shape
(100, 3947)
```

Sparse random projection

The *sklearn.random_projection.SparseRandomProjection* reduces the dimensionality by projecting the original input space using a sparse random matrix.

Sparse random matrices are an alternative to dense Gaussian random projection matrix that guarantees similar embedding quality while being much more memory efficient and allowing faster computation of the projected data.

If we define `s = 1 / density`, the elements of the random matrix are drawn from

$$\begin{cases} -\sqrt{\frac{s}{n_{\text{components}}}} & \\ 0 & \text{with probability} \\ +\sqrt{\frac{s}{n_{\text{components}}}} & \end{cases} \begin{matrix} \\ 1/2s \\ 1 - 1/s \\ 1/2s \end{matrix}$$

where $n_{\text{components}}$ is the size of the projected subspace. By default the density of non zero elements is set to the minimum density as recommended by Ping Li et al.: $1/\sqrt{n_{\text{features}}}$.

Here a small excerpt which illustrates how to use the sparse random projection transformer:

```
>>> import numpy as np
>>> from sklearn import random_projection
>>> X = np.random.rand(100, 10000)
>>> transformer = random_projection.SparseRandomProjection()
>>> X_new = transformer.fit_transform(X)
>>> X_new.shape
(100, 3947)
```

References:

- D. Achlioptas. 2003. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. Journal of Computer and System Sciences 66 (2003) 671–687
- Ping Li, Trevor J. Hastie, and Kenneth W. Church. 2006. Very sparse random projections. In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '06). ACM, New York, NY, USA, 287-296.

4.6.7 Kernel Approximation

This submodule contains functions that approximate the feature mappings that correspond to certain kernels, as they are used for example in support vector machines (see *Support Vector Machines*). The following feature functions perform non-linear transformations of the input, which can serve as a basis for linear classification or other algorithms.

The advantage of using approximate explicit feature maps compared to the kernel trick, which makes use of feature maps implicitly, is that explicit mappings can be better suited for online learning and can significantly reduce the cost of learning with very large datasets. Standard kernelized SVMs do not scale well to large datasets, but using an approximate kernel map it is possible to use much more efficient linear SVMs. In particular, the combination of kernel map approximations with `SGDClassifier` can make non-linear learning on large datasets possible.

Since there has not been much empirical work using approximate embeddings, it is advisable to compare results against exact kernel methods when possible.

See also:

Polynomial regression: extending linear models with basis functions for an exact polynomial transformation.

Nystroem Method for Kernel Approximation

The Nystroem method, as implemented in `Nystroem` is a general method for low-rank approximations of kernels. It achieves this by essentially subsampling the data on which the kernel is evaluated. By default `Nystroem` uses the `rbf` kernel, but it can use any kernel function or a precomputed kernel matrix. The number of samples used - which is also the dimensionality of the features computed - is given by the parameter `n_components`.

Radial Basis Function Kernel

The `RBFSampler` constructs an approximate mapping for the radial basis function kernel, also known as *Random Kitchen Sinks* [RR2007]. This transformation can be used to explicitly model a kernel map, prior to applying a linear algorithm, for example a linear SVM:

```
>>> from sklearn.kernel_approximation import RBFSampler
>>> from sklearn.linear_model import SGDClassifier
>>> X = [[0, 0], [1, 1], [1, 0], [0, 1]]
>>> y = [0, 0, 1, 1]
>>> rbf_feature = RBFSampler(gamma=1, random_state=1)
>>> X_features = rbf_feature.fit_transform(X)
>>> clf = SGDClassifier(max_iter=5)
>>> clf.fit(X_features, y)
SGDClassifier(max_iter=5)
>>> clf.score(X_features, y)
1.0
```

The mapping relies on a Monte Carlo approximation to the kernel values. The `fit` function performs the Monte Carlo sampling, whereas the `transform` method performs the mapping of the data. Because of the inherent randomness of the process, results may vary between different calls to the `fit` function.

The `fit` function takes two arguments: `n_components`, which is the target dimensionality of the feature transform, and `gamma`, the parameter of the RBF-kernel. A higher `n_components` will result in a better approximation of the kernel and will yield results more similar to those produced by a kernel SVM. Note that "fitting" the feature function does not actually depend on the data given to the `fit` function. Only the dimensionality of the data is used. Details on the method can be found in [RR2007].

For a given value of `n_components` `RBFSampler` is often less accurate as `Nystroem`. `RBFSampler` is cheaper to compute, though, making use of larger feature spaces more efficient.

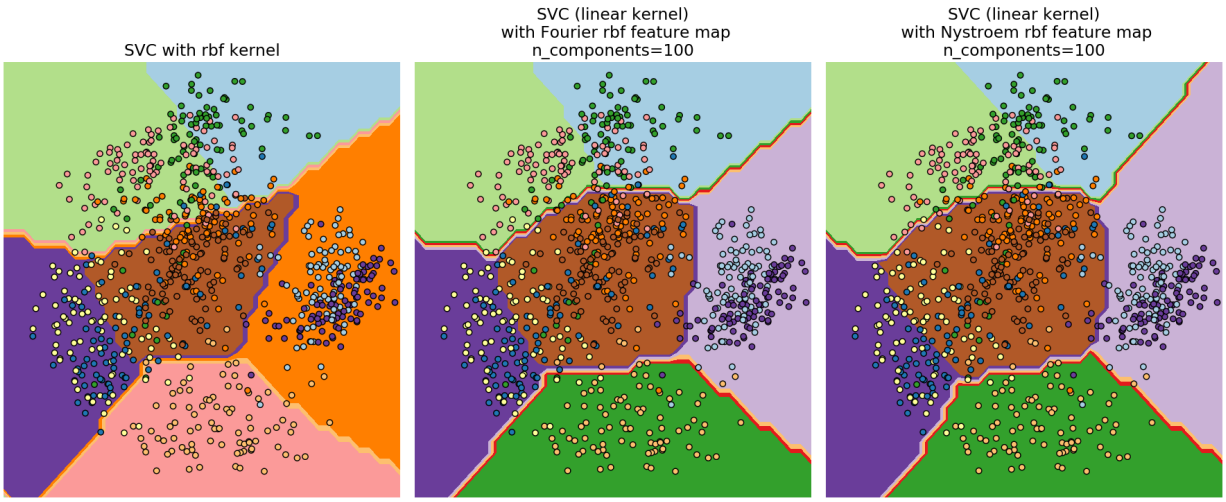


Fig. 9: Comparing an exact RBF kernel (left) with the approximation (right)

Examples:

- *Explicit feature map approximation for RBF kernels*

Additive Chi Squared Kernel

The additive chi squared kernel is a kernel on histograms, often used in computer vision.

The additive chi squared kernel as used here is given by

$$k(x, y) = \sum_i \frac{2x_i y_i}{x_i + y_i}$$

This is not exactly the same as `sklearn.metrics.additive_chi2_kernel`. The authors of [VZ2010] prefer the version above as it is always positive definite. Since the kernel is additive, it is possible to treat all components x_i separately for embedding. This makes it possible to sample the Fourier transform in regular intervals, instead of approximating using Monte Carlo sampling.

The class *`AdditiveChi2Sampler`* implements this component wise deterministic sampling. Each component is sampled n times, yielding $2n + 1$ dimensions per input dimension (the multiple of two stems from the real and complex part of the Fourier transform). In the literature, n is usually chosen to be 1 or 2, transforming the dataset to size `n_samples * 5 * n_features` (in the case of $n = 2$).

The approximate feature map provided by *`AdditiveChi2Sampler`* can be combined with the approximate feature map provided by *`RBFSampler`* to yield an approximate feature map for the exponentiated chi squared kernel. See the [VZ2010] for details and [VVZ2010] for combination with the *`RBFSampler`*.

Skewed Chi Squared Kernel

The skewed chi squared kernel is given by:

$$k(x, y) = \prod_i \frac{2\sqrt{x_i + c}\sqrt{y_i + c}}{x_i + y_i + 2c}$$

It has properties that are similar to the exponentiated chi squared kernel often used in computer vision, but allows for a simple Monte Carlo approximation of the feature map.

The usage of the *SkewedChi2Sampler* is the same as the usage described above for the *RBFSampler*. The only difference is in the free parameter, that is called c. For a motivation for this mapping and the mathematical details see [LS2010].

Mathematical Details

Kernel methods like support vector machines or kernelized PCA rely on a property of reproducing kernel Hilbert spaces. For any positive definite kernel function k (a so called Mercer kernel), it is guaranteed that there exists a mapping ϕ into a Hilbert space $\mathcal{H}$, such that

$$k(x, y) = \langle \phi(x), \phi(y) \rangle$$

Where $\langle \cdot, \cdot \rangle$ denotes the inner product in the Hilbert space.

If an algorithm, such as a linear support vector machine or PCA, relies only on the scalar product of data points x_i, one may use the value of $k(x_i, x_j)$, which corresponds to applying the algorithm to the mapped data points $\phi(x_i)$. The advantage of using k is that the mapping ϕ never has to be calculated explicitly, allowing for arbitrary large features (even infinite).

One drawback of kernel methods is, that it might be necessary to store many kernel values $k(x_i, x_j)$ during optimization. If a kernelized classifier is applied to new data y_j, $k(x_i, y_j)$ needs to be computed to make predictions, possibly for many different x_i in the training set.

The classes in this submodule allow to approximate the embedding ϕ, thereby working explicitly with the representations $\phi(x_i)$, which obviates the need to apply the kernel or store training examples.

References:

4.6.8 Pairwise metrics, Affinities and Kernels

The *sklearn.metrics.pairwise* submodule implements utilities to evaluate pairwise distances or affinity of sets of samples.

This module contains both distance metrics and kernels. A brief summary is given on the two here.

Distance metrics are functions `d(a, b)` such that `d(a, b) < d(a, c)` if objects `a` and `b` are considered "more similar" than objects `a` and `c`. Two objects exactly alike would have a distance of zero. One of the most popular examples is Euclidean distance. To be a 'true' metric, it must obey the following four conditions:

```
1. d(a, b) >= 0, for all a and b
2. d(a, b) == 0, if and only if a = b, positive definiteness
3. d(a, b) == d(b, a), symmetry
4. d(a, c) <= d(a, b) + d(b, c), the triangle inequality
```

Kernels are measures of similarity, i.e. `s(a, b) > s(a, c)` if objects `a` and `b` are considered "more similar" than objects `a` and `c`. A kernel must also be positive semi-definite.

There are a number of ways to convert between a distance metric and a similarity measure, such as a kernel. Let `D` be the distance, and `S` be the kernel:

1. `S = np.exp(-D * gamma)`, where one heuristic for choosing `gamma` is `1 / num_features`
2. `S = 1. / (D / np.max(D))`

The distances between the row vectors of X and the row vectors of Y can be evaluated using *pairwise_distances*. If Y is omitted the pairwise distances of the row vectors of X are calculated. Similarly, *pairwise.pairwise_kernels* can be used to calculate the kernel between X and Y using different kernel functions. See the API reference for more details.

```
>>> import numpy as np
>>> from sklearn.metrics import pairwise_distances
>>> from sklearn.metrics.pairwise import pairwise_kernels
>>> X = np.array([[2, 3], [3, 5], [5, 8]])
>>> Y = np.array([[1, 0], [2, 1]])
>>> pairwise_distances(X, Y, metric='manhattan')
array([[ 4.,  2.],
       [ 7.,  5.],
       [12., 10.]])
>>> pairwise_distances(X, metric='manhattan')
array([[0., 3., 8.],
       [3., 0., 5.],
       [8., 5., 0.]])
>>> pairwise_kernels(X, Y, metric='linear')
array([[ 2.,  7.],
       [ 3., 11.],
       [ 5., 18.]])
```

Cosine similarity

cosine_similarity computes the L2-normalized dot product of vectors. That is, if x and y are row vectors, their cosine similarity k is defined as:

$$k(x, y) = \frac{xy^\top}{\|x\|\|y\|}$$

This is called cosine similarity, because Euclidean (L2) normalization projects the vectors onto the unit sphere, and their dot product is then the cosine of the angle between the points denoted by the vectors.

This kernel is a popular choice for computing the similarity of documents represented as tf-idf vectors. *cosine_similarity* accepts scipy.sparse matrices. (Note that the tf-idf functionality in sklearn.feature_extraction.text can produce normalized vectors, in which case *cosine_similarity* is equivalent to *linear_kernel*, only slower.)

References:

- C.D. Manning, P. Raghavan and H. Schütze (2008). Introduction to Information Retrieval. Cambridge University Press. https://nlp.stanford.edu/IR-book/html/htmledition/the-vector-space-model-for-scoring-1.html

Linear kernel

The function *linear_kernel* computes the linear kernel, that is, a special case of *polynomial_kernel* with degree=1 and coef0=0 (homogeneous). If x and y are column vectors, their linear kernel is:

$$k(x, y) = x^\top y$$

Polynomial kernel

The function *polynomial_kernel* computes the degree-d polynomial kernel between two vectors. The polynomial kernel represents the similarity between two vectors. Conceptually, the polynomial kernels considers not only the similarity between vectors under the same dimension, but also across dimensions. When used in machine learning algorithms, this allows to account for feature interaction.

The polynomial kernel is defined as:

$$k(x, y) = (\gamma x^\top y + c_0)^d$$

where:

- x, y are the input vectors
- d is the kernel degree

If $c_0 = 0$ the kernel is said to be homogeneous.

Sigmoid kernel

The function *sigmoid_kernel* computes the sigmoid kernel between two vectors. The sigmoid kernel is also known as hyperbolic tangent, or Multilayer Perceptron (because, in the neural network field, it is often used as neuron activation function). It is defined as:

$$k(x, y) = \tanh(\gamma x^\top y + c_0)$$

where:

- x, y are the input vectors
- γ is known as slope
- c_0 is known as intercept

RBF kernel

The function *rbf_kernel* computes the radial basis function (RBF) kernel between two vectors. This kernel is defined as:

$$k(x, y) = \exp(-\gamma \|x - y\|^2)$$

where x and y are the input vectors. If $\gamma = \sigma^{-2}$ the kernel is known as the Gaussian kernel of variance σ^2.

Laplacian kernel

The function *laplacian_kernel* is a variant on the radial basis function kernel defined as:

$$k(x, y) = \exp(-\gamma \|x - y\|_1)$$

where x and y are the input vectors and $\|x - y\|_1$ is the Manhattan distance between the input vectors.

It has proven useful in ML applied to noiseless data. See e.g. Machine learning for quantum mechanics in a nutshell.

Chi-squared kernel

The chi-squared kernel is a very popular choice for training non-linear SVMs in computer vision applications. It can be computed using *chi2_kernel* and then passed to an *sklearn.svm.SVC* with `kernel="precomputed"`:

```
>>> from sklearn.svm import SVC
>>> from sklearn.metrics.pairwise import chi2_kernel
>>> X = [[0, 1], [1, 0], [.2, .8], [.7, .3]]
>>> y = [0, 1, 0, 1]
>>> K = chi2_kernel(X, gamma=.5)
>>> K
array([[1.        , 0.36787944, 0.89483932, 0.58364548],
       [0.36787944, 1.        , 0.51341712, 0.83822343],
       [0.89483932, 0.51341712, 1.        , 0.7768366 ],
       [0.58364548, 0.83822343, 0.7768366 , 1.        ]])

>>> svm = SVC(kernel='precomputed').fit(K, y)
>>> svm.predict(K)
array([0, 1, 0, 1])
```

It can also be directly used as the `kernel` argument:

```
>>> svm = SVC(kernel=chi2_kernel).fit(X, y)
>>> svm.predict(X)
array([0, 1, 0, 1])
```

The chi squared kernel is given by

$$k(x, y) = \exp\left(-\gamma \sum_i \frac{(x[i] - y[i])^2}{x[i] + y[i]}\right)$$

The data is assumed to be non-negative, and is often normalized to have an L1-norm of one. The normalization is rationalized with the connection to the chi squared distance, which is a distance between discrete probability distributions.

The chi squared kernel is most commonly used on histograms (bags) of visual words.

References:

- Zhang, J. and Marszalek, M. and Lazebnik, S. and Schmid, C. Local features and kernels for classification of texture and object categories: A comprehensive study International Journal of Computer Vision 2007 https://research.microsoft.com/en-us/um/people/manik/projects/trade-off/papers/ZhangIJCV06.pdf

4.6.9 Transforming the prediction target (`y`)

These are transformers that are not intended to be used on features, only on supervised learning targets. See also *Transforming target in regression* if you want to transform the prediction target for learning, but evaluate the model in the original (untransformed) space.

Label binarization

LabelBinarizer is a utility class to help create a label indicator matrix from a list of multi-class labels:

```
>>> from sklearn import preprocessing
>>> lb = preprocessing.LabelBinarizer()
>>> lb.fit([1, 2, 6, 4, 2])
LabelBinarizer()
>>> lb.classes_
array([1, 2, 4, 6])
>>> lb.transform([1, 6])
array([[1, 0, 0, 0],
       [0, 0, 0, 1]])
```

For multiple labels per instance, use *MultiLabelBinarizer*:

```
>>> lb = preprocessing.MultiLabelBinarizer()
>>> lb.fit_transform([(1, 2), (3,)])
array([[1, 1, 0],
       [0, 0, 1]])
>>> lb.classes_
array([1, 2, 3])
```

Label encoding

LabelEncoder is a utility class to help normalize labels such that they contain only values between 0 and n_classes-1. This is sometimes useful for writing efficient Cython routines. *LabelEncoder* can be used as follows:

```
>>> from sklearn import preprocessing
>>> le = preprocessing.LabelEncoder()
>>> le.fit([1, 2, 2, 6])
LabelEncoder()
>>> le.classes_
array([1, 2, 6])
>>> le.transform([1, 1, 2, 6])
array([0, 0, 1, 2])
>>> le.inverse_transform([0, 0, 1, 2])
array([1, 1, 2, 6])
```

It can also be used to transform non-numerical labels (as long as they are hashable and comparable) to numerical labels:

```
>>> le = preprocessing.LabelEncoder()
>>> le.fit(["paris", "paris", "tokyo", "amsterdam"])
LabelEncoder()
>>> list(le.classes_)
['amsterdam', 'paris', 'tokyo']
>>> le.transform(["tokyo", "tokyo", "paris"])
array([2, 2, 1])
>>> list(le.inverse_transform([2, 2, 1]))
['tokyo', 'tokyo', 'paris']
```

4.7 Dataset loading utilities

The `sklearn.datasets` package embeds some small toy datasets as introduced in the *Getting Started* section.

This package also features helpers to fetch larger datasets commonly used by the machine learning community to benchmark algorithms on data that comes from the 'real world'.

To evaluate the impact of the scale of the dataset (`n_samples` and `n_features`) while controlling the statistical properties of the data (typically the correlation and informativeness of the features), it is also possible to generate synthetic data.

4.7.1 General dataset API

There are three main kinds of dataset interfaces that can be used to get datasets depending on the desired type of dataset.

The dataset loaders. They can be used to load small standard datasets, described in the *Toy datasets* section.

The dataset fetchers. They can be used to download and load larger datasets, described in the *Real world datasets* section.

Both loaders and fetchers functions return a dictionary-like object holding at least two items: an array of shape `n_samples` * `n_features` with key `data` (except for 20newsgroups) and a numpy array of length `n_samples`, containing the target values, with key `target`.

It's also possible for almost all of these function to constrain the output to be a tuple containing only the data and the target, by setting the `return_X_y` parameter to `True`.

The datasets also contain a full description in their `DESCR` attribute and some contain `feature_names` and `target_names`. See the dataset descriptions below for details.

The dataset generation functions. They can be used to generate controlled synthetic datasets, described in the *Generated datasets* section.

These functions return a tuple `(X, y)` consisting of a `n_samples` * `n_features` numpy array `X` and an array of length `n_samples` containing the targets `y`.

In addition, there are also miscellaneous tools to load datasets of other formats or from other locations, described in the *Loading other datasets* section.

4.7.2 Toy datasets

scikit-learn comes with a few small standard datasets that do not require to download any file from some external website.

They can be loaded using the following functions:

load_boston([return_X_y])	Load and return the boston house-prices dataset (regression).
load_iris([return_X_y])	Load and return the iris dataset (classification).
load_diabetes([return_X_y])	Load and return the diabetes dataset (regression).
load_digits([n_class, return_X_y])	Load and return the digits dataset (classification).
load_linnerud([return_X_y])	Load and return the linnerud dataset (multivariate regression).
load_wine([return_X_y])	Load and return the wine dataset (classification).
load_breast_cancer([return_X_y])	Load and return the breast cancer wisconsin dataset (classification).

These datasets are useful to quickly illustrate the behavior of the various algorithms implemented in scikit-learn. They are however often too small to be representative of real world machine learning tasks.

Boston house prices dataset

Data Set Characteristics:

Number of Instances 506

Number of Attributes 13 numeric/categorical predictive. Median Value (attribute 14) is usually the target.

Attribute Information (in order)

- CRIM per capita crime rate by town
- ZN proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS proportion of non-retail business acres per town
- CHAS Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- NOX nitric oxides concentration (parts per 10 million)
- RM average number of rooms per dwelling
- AGE proportion of owner-occupied units built prior to 1940
- DIS weighted distances to five Boston employment centres
- RAD index of accessibility to radial highways
- TAX full-value property-tax rate per $10,000
- PTRATIO pupil-teacher ratio by town
- B 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- LSTAT % lower status of the population
- MEDV Median value of owner-occupied homes in $1000's

Missing Attribute Values None

Creator Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset. https://archive.ics.uci.edu/ml/machine-learning-databases/housing/

This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic prices and the demand for clean air', J. Environ. Economics & Management, vol.5, 81-102, 1978. Used in Belsley, Kuh & Welsch, 'Regression diagnostics ...', Wiley, 1980. N.B. Various transformations are used in the table on pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that address regression problems.

References

- Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Collinearity', Wiley, 1980. 244-261.
- Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth International Conference of Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.

Iris plants dataset

Data Set Characteristics:

Number of Instances 150 (50 in each of three classes)

Number of Attributes 4 numeric, predictive attributes and the class

Attribute Information

- sepal length in cm
- sepal width in cm
- petal length in cm
- petal width in cm
- **class:**
  - Iris-Setosa
  - Iris-Versicolour
  - Iris-Virginica

Summary Statistics

sepal length:	4.3	7.9	5.84	0.83	0.7826
sepal width:	2.0	4.4	3.05	0.43	-0.4194
petal length:	1.0	6.9	3.76	1.76	0.9490 (high!)
petal width:	0.1	2.5	1.20	0.76	0.9565 (high!)

Missing Attribute Values None

Class Distribution 33.3% for each of 3 classes.

Creator R.A. Fisher

Donor Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)

Date July, 1988

The famous Iris database, first used by Sir R.A. Fisher. The dataset is taken from Fisher's paper. Note that it's the same as in R, but not as in the UCI Machine Learning Repository, which has two wrong data points.

This is perhaps the best known database to be found in the pattern recognition literature. Fisher's paper is a classic in the field and is referenced frequently to this day. (See Duda & Hart, for example.) The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2; the latter are NOT linearly separable from each other.

References

- Fisher, R.A. "The use of multiple measurements in taxonomic problems" Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to Mathematical Statistics" (John Wiley, NY, 1950).
- Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analysis. (Q327.D83) John Wiley & Sons. ISBN 0-471-22361-1. See page 218.
- Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System Structure and Classification Rule for Recognition in Partially Exposed Environments". IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-2, No. 1, 67-71.

- Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule". IEEE Transactions on Information Theory, May 1972, 431-433.
- See also: 1988 MLC Proceedings, 54-64. Cheeseman et al"s AUTOCLASS II conceptual clustering system finds 3 classes in the data.
- Many, many more ...

Diabetes dataset

Ten baseline variables, age, sex, body mass index, average blood pressure, and six blood serum measurements were obtained for each of n = 442 diabetes patients, as well as the response of interest, a quantitative measure of disease progression one year after baseline.

Data Set Characteristics:

Number of Instances 442

Number of Attributes First 10 columns are numeric predictive values

Target Column 11 is a quantitative measure of disease progression one year after baseline

Attribute Information

- Age
- Sex
- Body mass index
- Average blood pressure
- S1
- S2
- S3
- S4
- S5
- S6

Note: Each of these 10 feature variables have been mean centered and scaled by the standard deviation times `n_samples` (i.e. the sum of squares of each column totals 1).

Source URL: https://www4.stat.ncsu.edu/~boos/var.select/diabetes.html

For more information see: Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani (2004) "Least Angle Regression," Annals of Statistics (with discussion), 407-499. (https://web.stanford.edu/~hastie/Papers/LARS/LeastAngle_2002.pdf)

Optical recognition of handwritten digits dataset

Data Set Characteristics:

Number of Instances 5620

Number of Attributes 64

Attribute Information 8x8 image of integer pixels in the range 0..16.

Missing Attribute Values None

Creator

E. Alpaydin (alpaydin '@' boun.edu.tr)

Date July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

The data set contains images of hand-written digits: 10 classes where each class refers to a digit.

Preprocessing programs made available by NIST were used to extract normalized bitmaps of handwritten digits from a preprinted form. From a total of 43 people, 30 contributed to the training set and different 13 to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of 4x4 and the number of on pixels are counted in each block. This generates an input matrix of 8x8 where each element is an integer in the range 0..16. This reduces dimensionality and gives invariance to small distortions.

For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G. T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C. L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469, 1994.

References

- C. Kaynak (1995) Methods of Combining Multiple Classifiers and Their Applications to Handwritten Digit Recognition, MSc Thesis, Institute of Graduate Studies in Science and Engineering, Bogazici University.
- E. Alpaydin, C. Kaynak (1998) Cascading Classifiers, Kybernetika.
- Ken Tang and Ponnuthurai N. Suganthan and Xi Yao and A. Kai Qin. Linear dimensionalityreduction using relevance weighted LDA. School of Electrical and Electronic Engineering Nanyang Technological University. 2005.
- Claudio Gentile. A New Approximate Maximal Margin Classification Algorithm. NIPS. 2000.

Linnerrud dataset

Data Set Characteristics:

Number of Instances 20

Number of Attributes 3

Missing Attribute Values None

The Linnerud dataset constains two small dataset:

- ***physiological* - CSV containing 20 observations on 3 exercise variables:** Weight, Waist and Pulse.
- ***exercise* - CSV containing 20 observations on 3 physiological variables:** Chins, Situps and Jumps.

References

- Tenenhaus, M. (1998). La regression PLS: theorie et pratique. Paris: Editions Technic.

Wine recognition dataset

Data Set Characteristics:

Number of Instances 178 (50 in each of three classes)

Number of Attributes 13 numeric, predictive attributes and the class

Attribute Information

- Alcohol
- Malic acid
- Ash
- Alcalinity of ash
- Magnesium
- Total phenols
- Flavanoids
- Nonflavanoid phenols
- Proanthocyanins
- Color intensity
- Hue
- OD280/OD315 of diluted wines
- Proline

- **class:**
  - class_0
  - class_1
  - class_2

Summary Statistics

Alcohol:	11.0	14.8	13.0	0.8
Malic Acid:	0.74	5.80	2.34	1.12
Ash:	1.36	3.23	2.36	0.27
Alcalinity of Ash:	10.6	30.0	19.5	3.3
Magnesium:	70.0	162.0	99.7	14.3
Total Phenols:	0.98	3.88	2.29	0.63
Flavanoids:	0.34	5.08	2.03	1.00
Nonflavanoid Phenols:	0.13	0.66	0.36	0.12
Proanthocyanins:	0.41	3.58	1.59	0.57
Colour Intensity:	1.3	13.0	5.1	2.3
Hue:	0.48	1.71	0.96	0.23
OD280/OD315 of diluted wines:	1.27	4.00	2.61	0.71
Proline:	278	1680	746	315

Missing Attribute Values None

Class Distribution class_0 (59), class_1 (71), class_2 (48)

Creator R.A. Fisher

Donor Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)

Date July, 1988

This is a copy of UCI ML Wine recognition datasets. https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data

The data is the results of a chemical analysis of wines grown in the same region in Italy by three different cultivators. There are thirteen different measurements taken for different constituents found in the three types of wine.

Original Owners:

Forina, M. et al, PARVUS - An Extendible Package for Data Exploration, Classification and Correlation. Institute of Pharmaceutical and Food Analysis and Technologies, Via Brigata Salerno, 16147 Genoa, Italy.

Citation:

Lichman, M. (2013). UCI Machine Learning Repository [https://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

References

(1) S. Aeberhard, D. Coomans and O. de Vel, Comparison of Classifiers in High Dimensional Settings, Tech. Rep. no. 92-02, (1992), Dept. of Computer Science and Dept. of Mathematics and Statistics, James Cook University of North Queensland. (Also submitted to Technometrics).

The data was used with many others for comparing various classifiers. The classes are separable, though only RDA has achieved 100% correct classification. (RDA : 100%, QDA 99.4%, LDA 98.9%, 1NN 96.1% (z-transformed data)) (All results using the leave-one-out technique)

(2) S. Aeberhard, D. Coomans and O. de Vel, "THE CLASSIFICATION PERFORMANCE OF RDA" Tech. Rep. no. 92-01, (1992), Dept. of Computer Science and Dept. of Mathematics and Statistics, James Cook University of North Queensland. (Also submitted to Journal of Chemometrics).

Breast cancer wisconsin (diagnostic) dataset

Data Set Characteristics:

Number of Instances 569

Number of Attributes 30 numeric, predictive attributes and the class

Attribute Information

- radius (mean of distances from center to points on the perimeter)
- texture (standard deviation of gray-scale values)
- perimeter
- area
- smoothness (local variation in radius lengths)
- compactness (perimeter^2 / area - 1.0)
- concavity (severity of concave portions of the contour)
- concave points (number of concave portions of the contour)
- symmetry
- fractal dimension ("coastline approximation" - 1)

The mean, standard error, and "worst" or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features. For instance, field 3 is Mean Radius, field 13 is Radius SE, field 23 is Worst Radius.

- **class:**
  - WDBC-Malignant
  - WDBC-Benign

Summary Statistics

radius (mean):	6.981	28.11
texture (mean):	9.71	39.28
perimeter (mean):	43.79	188.5
area (mean):	143.5	2501.0
smoothness (mean):	0.053	0.163
compactness (mean):	0.019	0.345
concavity (mean):	0.0	0.427
concave points (mean):	0.0	0.201
symmetry (mean):	0.106	0.304
fractal dimension (mean):	0.05	0.097
radius (standard error):	0.112	2.873
texture (standard error):	0.36	4.885
perimeter (standard error):	0.757	21.98
area (standard error):	6.802	542.2
smoothness (standard error):	0.002	0.031
compactness (standard error):	0.002	0.135
concavity (standard error):	0.0	0.396
concave points (standard error):	0.0	0.053
symmetry (standard error):	0.008	0.079
fractal dimension (standard error):	0.001	0.03
radius (worst):	7.93	36.04
texture (worst):	12.02	49.54
perimeter (worst):	50.41	251.2
area (worst):	185.2	4254.0
smoothness (worst):	0.071	0.223
compactness (worst):	0.027	1.058
concavity (worst):	0.0	1.252
concave points (worst):	0.0	0.291
symmetry (worst):	0.156	0.664
fractal dimension (worst):	0.055	0.208

Missing Attribute Values None

Class Distribution 212 - Malignant, 357 - Benign

Creator Dr. William H. Wolberg, W. Nick Street, Olvi L. Mangasarian

Donor Nick Street

Date November, 1995

This is a copy of UCI ML Breast Cancer Wisconsin (Diagnostic) datasets. https://goo.gl/U2Uwz2

Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image.

Separating plane described above was obtained using Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree Construction Via Linear Programming." Proceedings of the 4th Midwest Artificial Intelligence and Cognitive Science Society, pp. 97-101, 1992], a classification method which uses linear programming to construct a decision tree. Relevant features were selected using an exhaustive search in the space of 1-4 features and 1-3 separating planes.

The actual linear program used to obtain the separating plane in the 3-dimensional space is that described in: [K. P. Bennett and O. L. Mangasarian: "Robust Linear Programming Discrimination of Two Linearly Inseparable Sets", Optimization Methods and Software 1, 1992, 23-34].

This database is also available through the UW CS ftp server:

ftp ftp.cs.wisc.edu cd math-prog/cpo-dataset/machine-learn/WDBC/

References

- W.N. Street, W.H. Wolberg and O.L. Mangasarian. Nuclear feature extraction for breast tumor diagnosis. IS&T/SPIE 1993 International Symposium on Electronic Imaging: Science and Technology, volume 1905, pages 861-870, San Jose, CA, 1993.
- O.L. Mangasarian, W.N. Street and W.H. Wolberg. Breast cancer diagnosis and prognosis via linear programming. Operations Research, 43(4), pages 570-577, July-August 1995.
- W.H. Wolberg, W.N. Street, and O.L. Mangasarian. Machine learning techniques to diagnose breast cancer from fine-needle aspirates. Cancer Letters 77 (1994) 163-171.

4.7.3 Real world datasets

scikit-learn provides tools to load larger datasets, downloading them if necessary.

They can be loaded using the following functions:

fetch_olivetti_faces([data_home, shuffle, ...])	Load the Olivetti faces data-set from AT&T (classification).
fetch_20newsgroups([data_home, subset, ...])	Load the filenames and data from the 20 newsgroups dataset (classification).
fetch_20newsgroups_vectorized([subset, ...])	Load the 20 newsgroups dataset and vectorize it into token counts (classification).
fetch_lfw_people([data_home, funneled, ...])	Load the Labeled Faces in the Wild (LFW) people dataset (classification).
fetch_lfw_pairs([subset, data_home, ...])	Load the Labeled Faces in the Wild (LFW) pairs dataset (classification).
fetch_covtype([data_home, ...])	Load the covertype dataset (classification).
fetch_rcv1([data_home, subset, ...])	Load the RCV1 multilabel dataset (classification).
fetch_kddcup99([subset, data_home, shuffle, ...])	Load the kddcup99 dataset (classification).
fetch_california_housing([data_home, ...])	Load the California housing dataset (regression).

The Olivetti faces dataset

This dataset contains a set of face images taken between April 1992 and April 1994 at AT&T Laboratories Cambridge. The *sklearn.datasets.fetch_olivetti_faces* function is the data fetching / caching function that downloads the data archive from AT&T.

As described on the original website:

There are ten different images of each of 40 distinct subjects. For some subjects, the images were taken at different times, varying the lighting, facial expressions (open / closed eyes, smiling / not smiling) and facial details (glasses / no glasses). All the images were taken against a dark homogeneous background with the subjects in an upright, frontal position (with tolerance for some side movement).

Data Set Characteristics:

Classes	40
Samples total	400
Dimensionality	4096
Features	real, between 0 and 1

The image is quantized to 256 grey levels and stored as unsigned 8-bit integers; the loader will convert these to floating point values on the interval [0, 1], which are easier to work with for many algorithms.

The "target" for this database is an integer from 0 to 39 indicating the identity of the person pictured; however, with only 10 examples per class, this relatively small dataset is more interesting from an unsupervised or semi-supervised perspective.

The original dataset consisted of 92 x 112, while the version available here consists of 64x64 images.

When using these images, please give credit to AT&T Laboratories Cambridge.

The 20 newsgroups text dataset

The 20 newsgroups dataset comprises around 18000 newsgroups posts on 20 topics split in two subsets: one for training (or development) and the other one for testing (or for performance evaluation). The split between the train and test set is based upon a messages posted before and after a specific date.

This module contains two loaders. The first one, *sklearn.datasets.fetch_20newsgroups*, returns a list of the raw texts that can be fed to text feature extractors such as *sklearn.feature_extraction.text.CountVectorizer* with custom parameters so as to extract feature vectors. The second one, *sklearn.datasets.fetch_20newsgroups_vectorized*, returns ready-to-use features, i.e., it is not necessary to use a feature extractor.

Data Set Characteristics:

Classes	20
Samples total	18846
Dimensionality	1
Features	text

Usage

The *sklearn.datasets.fetch_20newsgroups* function is a data fetching / caching functions that downloads the data archive from the original 20 newsgroups website, extracts the archive contents in the `~/scikit_learn_data/20news_home` folder and calls the *sklearn.datasets.load_files* on either the training or testing set folder, or both of them:

```
>>> from sklearn.datasets import fetch_20newsgroups
>>> newsgroups_train = fetch_20newsgroups(subset='train')

>>> from pprint import pprint
>>> pprint(list(newsgroups_train.target_names))
```

(continues on next page)

(continued from previous page)

```
['alt.atheism',
 'comp.graphics',
 'comp.os.ms-windows.misc',
 'comp.sys.ibm.pc.hardware',
 'comp.sys.mac.hardware',
 'comp.windows.x',
 'misc.forsale',
 'rec.autos',
 'rec.motorcycles',
 'rec.sport.baseball',
 'rec.sport.hockey',
 'sci.crypt',
 'sci.electronics',
 'sci.med',
 'sci.space',
 'soc.religion.christian',
 'talk.politics.guns',
 'talk.politics.mideast',
 'talk.politics.misc',
 'talk.religion.misc']
```

The real data lies in the `filenames` and `target` attributes. The target attribute is the integer index of the category:

```
>>> newsgroups_train.filenames.shape
(11314,)
>>> newsgroups_train.target.shape
(11314,)
>>> newsgroups_train.target[:10]
array([ 7,  4,  4,  1, 14, 16, 13,  3,  2,  4])
```

It is possible to load only a sub-selection of the categories by passing the list of the categories to load to the *sklearn.datasets.fetch_20newsgroups* function:

```
>>> cats = ['alt.atheism', 'sci.space']
>>> newsgroups_train = fetch_20newsgroups(subset='train', categories=cats)

>>> list(newsgroups_train.target_names)
['alt.atheism', 'sci.space']
>>> newsgroups_train.filenames.shape
(1073,)
>>> newsgroups_train.target.shape
(1073,)
>>> newsgroups_train.target[:10]
array([0, 1, 1, 1, 0, 1, 1, 0, 0, 0])
```

Converting text to vectors

In order to feed predictive or clustering models with the text data, one first need to turn the text into vectors of numerical values suitable for statistical analysis. This can be achieved with the utilities of the `sklearn.feature_extraction.text` as demonstrated in the following example that extract TF-IDF vectors of unigram tokens from a subset of 20news:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> categories = ['alt.atheism', 'talk.religion.misc',
```

(continues on next page)

(continued from previous page)

```
...                'comp.graphics', 'sci.space']
>>> newsgroups_train = fetch_20newsgroups(subset='train',
...                                       categories=categories)
>>> vectorizer = TfidfVectorizer()
>>> vectors = vectorizer.fit_transform(newsgroups_train.data)
>>> vectors.shape
(2034, 34118)
```

The extracted TF-IDF vectors are very sparse, with an average of 159 non-zero components by sample in a more than 30000-dimensional space (less than .5% non-zero features):

```
>>> vectors.nnz / float(vectors.shape[0])
159.01327...
```

sklearn.datasets.fetch_20newsgroups_vectorized is a function which returns ready-to-use token counts features instead of file names.

Filtering text for more realistic training

It is easy for a classifier to overfit on particular things that appear in the 20 Newsgroups data, such as newsgroup headers. Many classifiers achieve very high F-scores, but their results would not generalize to other documents that aren't from this window of time.

For example, let's look at the results of a multinomial Naive Bayes classifier, which is fast to train and achieves a decent F-score:

```
>>> from sklearn.naive_bayes import MultinomialNB
>>> from sklearn import metrics
>>> newsgroups_test = fetch_20newsgroups(subset='test',
...                                      categories=categories)
>>> vectors_test = vectorizer.transform(newsgroups_test.data)
>>> clf = MultinomialNB(alpha=.01)
>>> clf.fit(vectors, newsgroups_train.target)
MultinomialNB(alpha=0.01, class_prior=None, fit_prior=True)

>>> pred = clf.predict(vectors_test)
>>> metrics.f1_score(newsgroups_test.target, pred, average='macro')
0.88213...
```

(The example *Classification of text documents using sparse features* shuffles the training and test data, instead of segmenting by time, and in that case multinomial Naive Bayes gets a much higher F-score of 0.88. Are you suspicious yet of what's going on inside this classifier?)

Let's take a look at what the most informative features are:

```
>>> import numpy as np
>>> def show_top10(classifier, vectorizer, categories):
...     feature_names = np.asarray(vectorizer.get_feature_names())
...     for i, category in enumerate(categories):
...         top10 = np.argsort(classifier.coef_[i])[-10:]
...         print("%s: %s" % (category, " ".join(feature_names[top10])))
...
>>> show_top10(clf, vectorizer, newsgroups_train.target_names)
alt.atheism: edu it and in you that is of to the
comp.graphics: edu in graphics it is for and of to the
```

(continues on next page)

(continued from previous page)

```
sci.space: edu it that is in and space to of the
talk.religion.misc: not it you in is that and to of the
```

You can now see many things that these features have overfit to:

- Almost every group is distinguished by whether headers such as `NNTP-Posting-Host:` and `Distribution:` appear more or less often.
- Another significant feature involves whether the sender is affiliated with a university, as indicated either by their headers or their signature.
- The word "article" is a significant feature, based on how often people quote previous posts like this: "In article [article ID], [name] <[e-mail address]> wrote:"
- Other features match the names and e-mail addresses of particular people who were posting at the time.

With such an abundance of clues that distinguish newsgroups, the classifiers barely have to identify topics from text at all, and they all perform at the same high level.

For this reason, the functions that load 20 Newsgroups data provide a parameter called **remove**, telling it what kinds of information to strip out of each file. **remove** should be a tuple containing any subset of `('headers', 'footers', 'quotes')`, telling it to remove headers, signature blocks, and quotation blocks respectively.

```
>>> newsgroups_test = fetch_20newsgroups(subset='test',
...                                      remove=('headers', 'footers', 'quotes'),
...                                      categories=categories)
>>> vectors_test = vectorizer.transform(newsgroups_test.data)
>>> pred = clf.predict(vectors_test)
>>> metrics.f1_score(pred, newsgroups_test.target, average='macro')
0.77310...
```

This classifier lost over a lot of its F-score, just because we removed metadata that has little to do with topic classification. It loses even more if we also strip this metadata from the training data:

```
>>> newsgroups_train = fetch_20newsgroups(subset='train',
...                                       remove=('headers', 'footers', 'quotes'),
...                                       categories=categories)
>>> vectors = vectorizer.fit_transform(newsgroups_train.data)
>>> clf = MultinomialNB(alpha=.01)
>>> clf.fit(vectors, newsgroups_train.target)
MultinomialNB(alpha=0.01, class_prior=None, fit_prior=True)
```

```
>>> vectors_test = vectorizer.transform(newsgroups_test.data)
>>> pred = clf.predict(vectors_test)
>>> metrics.f1_score(newsgroups_test.target, pred, average='macro')
0.76995...
```

Some other classifiers cope better with this harder version of the task. Try running *Sample pipeline for text feature extraction and evaluation* with and without the `--filter` option to compare the results.

Recommendation

When evaluating text classifiers on the 20 Newsgroups data, you should strip newsgroup-related metadata. In scikit-learn, you can do this by setting `remove=('headers', 'footers', 'quotes')`. The F-score will be lower because it is more realistic.

Examples

- *Sample pipeline for text feature extraction and evaluation*
- *Classification of text documents using sparse features*

The Labeled Faces in the Wild face recognition dataset

This dataset is a collection of JPEG pictures of famous people collected over the internet, all details are available on the official website:

http://vis-www.cs.umass.edu/lfw/

Each picture is centered on a single face. The typical task is called Face Verification: given a pair of two pictures, a binary classifier must predict whether the two images are from the same person.

An alternative task, Face Recognition or Face Identification is: given the picture of the face of an unknown person, identify the name of the person by referring to a gallery of previously seen pictures of identified persons.

Both Face Verification and Face Recognition are tasks that are typically performed on the output of a model trained to perform Face Detection. The most popular model for Face Detection is called Viola-Jones and is implemented in the OpenCV library. The LFW faces were extracted by this face detector from various online websites.

Data Set Characteristics:

Classes	5749
Samples total	13233
Dimensionality	5828
Features	real, between 0 and 255

Usage

`scikit-learn` provides two loaders that will automatically download, cache, parse the metadata files, decode the jpeg and convert the interesting slices into memmapped numpy arrays. This dataset size is more than 200 MB. The first load typically takes more than a couple of minutes to fully decode the relevant part of the JPEG files into numpy arrays. If the dataset has been loaded once, the following times the loading times less than 200ms by using a memmapped version memoized on the disk in the `~/scikit_learn_data/lfw_home/` folder using `joblib`.

The first loader is used for the Face Identification task: a multi-class classification task (hence supervised learning):

```
>>> from sklearn.datasets import fetch_lfw_people
>>> lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)

>>> for name in lfw_people.target_names:
...     print(name)
...
Ariel Sharon
Colin Powell
Donald Rumsfeld
George W Bush
Gerhard Schroeder
Hugo Chavez
Tony Blair
```

The default slice is a rectangular shape around the face, removing most of the background:

```
>>> lfw_people.data.dtype
dtype('float32')

>>> lfw_people.data.shape
(1288, 1850)

>>> lfw_people.images.shape
(1288, 50, 37)
```

Each of the `1140` faces is assigned to a single person id in the `target` array:

```
>>> lfw_people.target.shape
(1288,)

>>> list(lfw_people.target[:10])
[5, 6, 3, 1, 0, 1, 3, 4, 3, 0]
```

The second loader is typically used for the face verification task: each sample is a pair of two picture belonging or not to the same person:

```
>>> from sklearn.datasets import fetch_lfw_pairs
>>> lfw_pairs_train = fetch_lfw_pairs(subset='train')

>>> list(lfw_pairs_train.target_names)
['Different persons', 'Same person']

>>> lfw_pairs_train.pairs.shape
(2200, 2, 62, 47)

>>> lfw_pairs_train.data.shape
(2200, 5828)

>>> lfw_pairs_train.target.shape
(2200,)
```

Both for the *`sklearn.datasets.fetch_lfw_people`* and *`sklearn.datasets.fetch_lfw_pairs`* function it is possible to get an additional dimension with the RGB color channels by passing `color=True`, in that case the shape will be `(2200, 2, 62, 47, 3)`.

The *`sklearn.datasets.fetch_lfw_pairs`* datasets is subdivided into 3 subsets: the development `train` set, the development `test` set and an evaluation `10_folds` set meant to compute performance metrics using a 10-folds cross validation scheme.

References:

- Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments. Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. University of Massachusetts, Amherst, Technical Report 07-49, October, 2007.

Examples

Faces recognition example using eigenfaces and SVMs

Forest covertypes

The samples in this dataset correspond to 30×30m patches of forest in the US, collected for the task of predicting each patch's cover type, i.e. the dominant species of tree. There are seven covertypes, making this a multiclass classification problem. Each sample has 54 features, described on the dataset's homepage. Some of the features are boolean indicators, while others are discrete or continuous measurements.

Data Set Characteristics:

Classes	7
Samples total	581012
Dimensionality	54
Features	int

sklearn.datasets.fetch_covtype will load the covertype dataset; it returns a dictionary-like object with the feature matrix in the `data` member and the target values in `target`. The dataset will be downloaded from the web if necessary.

RCV1 dataset

Reuters Corpus Volume I (RCV1) is an archive of over 800,000 manually categorized newswire stories made available by Reuters, Ltd. for research purposes. The dataset is extensively described in[1].

Data Set Characteristics:

Classes	103
Samples total	804414
Dimensionality	47236
Features	real, between 0 and 1

sklearn.datasets.fetch_rcv1 will load the following version: RCV1-v2, vectors, full sets, topics multilabels:

```
>>> from sklearn.datasets import fetch_rcv1
>>> rcv1 = fetch_rcv1()
```

It returns a dictionary-like object, with the following attributes:

`data`: The feature matrix is a scipy CSR sparse matrix, with 804414 samples and 47236 features. Non-zero values contains cosine-normalized, log TF-IDF vectors. A nearly chronological split is proposed in[1]: The first 23149 samples are the training set. The last 781265 samples are the testing set. This follows the official LYRL2004 chronological split. The array has 0.16% of non zero values:

```
>>> rcv1.data.shape
(804414, 47236)
```

`target`: The target values are stored in a scipy CSR sparse matrix, with 804414 samples and 103 categories. Each sample has a value of 1 in its categories, and 0 in others. The array has 3.15% of non zero values:

```
>>> rcv1.target.shape
(804414, 103)
```

[1] Lewis, D. D., Yang, Y., Rose, T. G., & Li, F. (2004). RCV1: A new benchmark collection for text categorization research. The Journal of Machine Learning Research, 5, 361-397.

`sample_id`: Each sample can be identified by its ID, ranging (with gaps) from 2286 to 810596:

```
>>> rcv1.sample_id[:3]
array([2286, 2287, 2288], dtype=uint32)
```

`target_names`: The target values are the topics of each sample. Each sample belongs to at least one topic, and to up to 17 topics. There are 103 topics, each represented by a string. Their corpus frequencies span five orders of magnitude, from 5 occurrences for 'GMIL', to 381327 for 'CCAT':

```
>>> rcv1.target_names[:3].tolist()
['E11', 'ECAT', 'M11']
```

The dataset will be downloaded from the rcv1 homepage if necessary. The compressed size is about 656 MB.

References

Kddcup 99 dataset

The KDD Cup '99 dataset was created by processing the tcpdump portions of the 1998 DARPA Intrusion Detection System (IDS) Evaluation dataset, created by MIT Lincoln Lab [1]. The artificial data (described on the dataset's homepage) was generated using a closed network and hand-injected attacks to produce a large number of different types of attack with normal activity in the background. As the initial goal was to produce a large training set for supervised learning algorithms, there is a large proportion (80.1%) of abnormal data which is unrealistic in real world, and inappropriate for unsupervised anomaly detection which aims at detecting 'abnormal' data, ie

1) qualitatively different from normal data

2) in large minority among the observations.

We thus transform the KDD Data set into two different data sets: SA and SF.

-SA is obtained by simply selecting all the normal data, and a small proportion of abnormal data to gives an anomaly proportion of 1%.

-SF is obtained as in [2] by simply picking up the data whose attribute logged_in is positive, thus focusing on the intrusion attack, which gives a proportion of 0.3% of attack.

-http and smtp are two subsets of SF corresponding with third feature equal to 'http' (resp. to 'smtp')

General KDD structure :

Samples total	4898431
Dimensionality	41
Features	discrete (int) or continuous (float)
Targets	str, 'normal.' or name of the anomaly type

SA structure :

Samples total	976158
Dimensionality	41
Features	discrete (int) or continuous (float)
Targets	str, 'normal.' or name of the anomaly type

SF structure :

Samples total	699691
Dimensionality	4
Features	discrete (int) or continuous (float)
Targets	str, 'normal.' or name of the anomaly type

http structure :

Samples total	619052
Dimensionality	3
Features	discrete (int) or continuous (float)
Targets	str, 'normal.' or name of the anomaly type

smtp structure :

Samples total	95373
Dimensionality	3
Features	discrete (int) or continuous (float)
Targets	str, 'normal.' or name of the anomaly type

sklearn.datasets.fetch_kddcup99 will load the kddcup99 dataset; it returns a dictionary-like object with the feature matrix in the `data` member and the target values in `target`. The dataset will be downloaded from the web if necessary.

California Housing dataset

Data Set Characteristics:

Number of Instances 20640

Number of Attributes 8 numeric, predictive attributes and the target

Attribute Information

- MedInc median income in block
- HouseAge median house age in block
- AveRooms average number of rooms
- AveBedrms average number of bedrooms
- Population block population
- AveOccup average house occupancy
- Latitude house block latitude
- Longitude house block longitude

Missing Attribute Values None

This dataset was obtained from the StatLib repository. http://lib.stat.cmu.edu/datasets/

The target variable is the median house value for California districts.

This dataset was derived from the 1990 U.S. census, using one row per census block group. A block group is the smallest geographical unit for which the U.S. Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people).

It can be downloaded/loaded using the `sklearn.datasets.fetch_california_housing` function.

References

- Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions, Statistics and Probability Letters, 33 (1997) 291-297

4.7.4 Generated datasets

In addition, scikit-learn includes various random sample generators that can be used to build artificial datasets of controlled size and complexity.

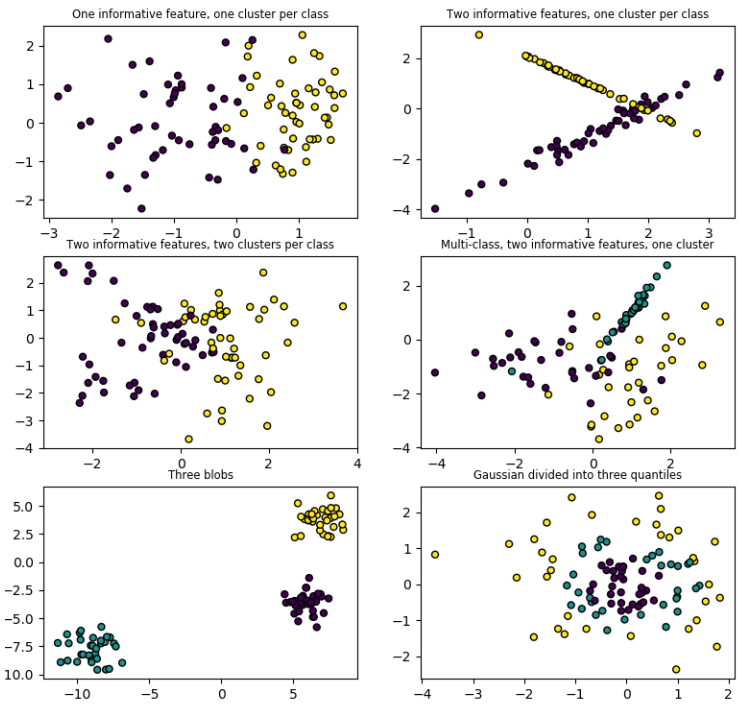
Generators for classification and clustering

These generators produce a matrix of features and corresponding discrete targets.

Single label

Both `make_blobs` and `make_classification` create multiclass datasets by allocating each class one or more normally-distributed clusters of points. `make_blobs` provides greater control regarding the centers and standard deviations of each cluster, and is used to demonstrate clustering. `make_classification` specialises in introducing noise by way of: correlated, redundant and uninformative features; multiple Gaussian clusters per class; and linear transformations of the feature space.

`make_gaussian_quantiles` divides a single Gaussian cluster into near-equal-size classes separated by concentric hyperspheres. `make_hastie_10_2` generates a similar binary, 10-dimensional problem.

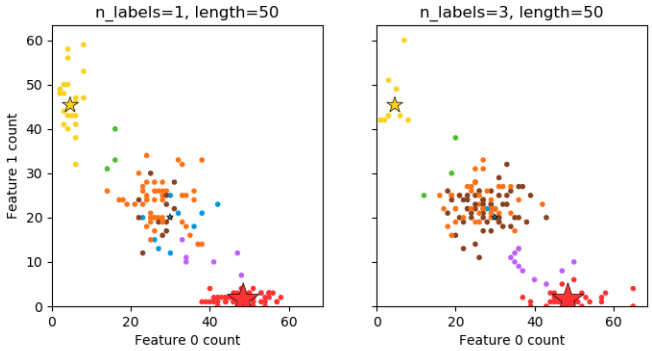


make_circles and *make_moons* generate 2d binary classification datasets that are challenging to certain algorithms (e.g. centroid-based clustering or linear classification), including optional Gaussian noise. They are useful for visualisation. *make_circles* produces Gaussian data with a spherical decision boundary for binary classification, while *make_moons* produces two interleaving half circles.

Multilabel

make_multilabel_classification generates random samples with multiple labels, reflecting a bag of words drawn from a mixture of topics. The number of topics for each document is drawn from a Poisson distribution, and the topics themselves are drawn from a fixed random distribution. Similarly, the number of words is drawn from Poisson, with words drawn from a multinomial, where each topic defines a probability distribution over words. Simplifications with respect to true bag-of-words mixtures include:

- Per-topic word distributions are independently drawn, where in reality all would be affected by a sparse base distribution, and would be correlated.
- For a document generated from multiple topics, all topics are weighted equally in generating its bag of words.
- Documents without labels words at random, rather than from a base distribution.



Biclustering

make_biclusters(shape, n_clusters[, noise, ...])	Generate an array with constant block diagonal structure for biclustering.
make_checkerboard(shape, n_clusters[, ...])	Generate an array with block checkerboard structure for biclustering.

Generators for regression

make_regression produces regression targets as an optionally-sparse random linear combination of random features, with noise. Its informative features may be uncorrelated, or low rank (few features account for most of the variance).

Other regression generators generate functions deterministically from randomized features. *make_sparse_uncorrelated* produces a target as a linear combination of four features with fixed coefficients. Others encode explicitly non-linear relations: *make_friedman1* is related by polynomial and sine transforms; *make_friedman2* includes feature multiplication and reciprocation; and *make_friedman3* is similar with an arctan transformation on the target.

Generators for manifold learning

make_s_curve([n_samples, noise, random_state])	Generate an S curve dataset.
make_swiss_roll([n_samples, noise, random_state])	Generate a swiss roll dataset.

Generators for decomposition

make_low_rank_matrix([n_samples, ...])	Generate a mostly low rank matrix with bell-shaped singular values
make_sparse_coded_signal(n_samples, ...[, ...])	Generate a signal as a sparse combination of dictionary elements.
make_spd_matrix(n_dim[, random_state])	Generate a random symmetric, positive-definite matrix.
make_sparse_spd_matrix([dim, alpha, ...])	Generate a sparse symmetric definite positive matrix.

4.7.5 Loading other datasets

Sample images

Scikit-learn also embed a couple of sample JPEG images published under Creative Commons license by their authors. Those images can be useful to test algorithms and pipeline on 2D data.

load_sample_images()	Load sample images for image manipulation.
load_sample_image(image_name)	Load the numpy array of a single sample image

Warning: The default coding of images is based on the `uint8` dtype to spare memory. Often machine learning algorithms work best if the input is converted to a floating point representation first. Also, if you plan to use `matplotlib.pyplpt.imshow` don't forget to scale to the range 0 - 1 as done in the following example.

Examples:

- *Color Quantization using K-Means*

Datasets in svmlight / libsvm format

scikit-learn includes utility functions for loading datasets in the svmlight / libsvm format. In this format, each line takes the form `<label> <feature-id>:<feature-value> <feature-id>:<feature-value> ..`

.. This format is especially suitable for sparse datasets. In this module, scipy sparse CSR matrices are used for `X` and numpy arrays are used for `y`.

You may load a dataset like as follows:

```
>>> from sklearn.datasets import load_svmlight_file
>>> X_train, y_train = load_svmlight_file("/path/to/train_dataset.txt")
...
```

You may also load two (or more) datasets at once:

```
>>> X_train, y_train, X_test, y_test = load_svmlight_files(
...     ("/path/to/train_dataset.txt", "/path/to/test_dataset.txt"))
...
```

In this case, `X_train` and `X_test` are guaranteed to have the same number of features. Another way to achieve the same result is to fix the number of features:

```
>>> X_test, y_test = load_svmlight_file(
...     "/path/to/test_dataset.txt", n_features=X_train.shape[1])
...
```

Related links:

Public datasets in svmlight / libsvm format: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets

Faster API-compatible implementation: https://github.com/mblondel/svmlight-loader

Downloading datasets from the openml.org repository

openml.org is a public repository for machine learning data and experiments, that allows everybody to upload open datasets.

The `sklearn.datasets` package is able to download datasets from the repository using the function *`sklearn.datasets.fetch_openml`*.

For example, to download a dataset of gene expressions in mice brains:

```
>>> from sklearn.datasets import fetch_openml
>>> mice = fetch_openml(name='miceprotein', version=4)
```

To fully specify a dataset, you need to provide a name and a version, though the version is optional, see *Dataset Versions* below. The dataset contains a total of 1080 examples belonging to 8 different classes:

```
>>> mice.data.shape
(1080, 77)
>>> mice.target.shape
(1080,)
>>> np.unique(mice.target)
array(['c-CS-m', 'c-CS-s', 'c-SC-m', 'c-SC-s', 't-CS-m', 't-CS-s', 't-SC-m', 't-SC-s
↪'], dtype=object)
```

You can get more information on the dataset by looking at the `DESCR` and `details` attributes:

```
>>> print(mice.DESCR)
**Author**: Clara Higuera, Katheleen J. Gardiner, Krzysztof J. Cios
**Source**: [UCI](https://archive.ics.uci.edu/ml/datasets/Mice+Protein+Expression) -␣
↪2015
**Please cite**: Higuera C, Gardiner KJ, Cios KJ (2015) Self-Organizing
Feature Maps Identify Proteins Critical to Learning in a Mouse Model of Down
Syndrome. PLoS ONE 10(6): e0129126...

>>> mice.details
{'id': '40966', 'name': 'MiceProtein', 'version': '4', 'format': 'ARFF',
'upload_date': '2017-11-08T16:00:15', 'licence': 'Public',
'url': 'https://www.openml.org/data/v1/download/17928620/MiceProtein.arff',
'file_id': '17928620', 'default_target_attribute': 'class',
'row_id_attribute': 'MouseID',
'ignore_attribute': ['Genotype', 'Treatment', 'Behavior'],
'tag': ['OpenML-CC18', 'study_135', 'study_98', 'study_99'],
'visibility': 'public', 'status': 'active',
'md5_checksum': '3c479a6885bfa0438971388283a1ce32'}
```

The `DESCR` contains a free-text description of the data, while `details` contains a dictionary of meta-data stored by openml, like the dataset id. For more details, see the OpenML documentation The `data_id` of the mice protein dataset is 40966, and you can use this (or the name) to get more information on the dataset on the openml website:

```
>>> mice.url
'https://www.openml.org/d/40966'
```

The `data_id` also uniquely identifies a dataset from OpenML:

```
>>> mice = fetch_openml(data_id=40966)
>>> mice.details
{'id': '4550', 'name': 'MiceProtein', 'version': '1', 'format': 'ARFF',
'creator': ...,
'upload_date': '2016-02-17T14:32:49', 'licence': 'Public', 'url':
'https://www.openml.org/data/v1/download/1804243/MiceProtein.ARFF', 'file_id':
'1804243', 'default_target_attribute': 'class', 'citation': 'Higuera C,
Gardiner KJ, Cios KJ (2015) Self-Organizing Feature Maps Identify Proteins
Critical to Learning in a Mouse Model of Down Syndrome. PLoS ONE 10(6):
e0129126. [Web Link] journal.pone.0129126', 'tag': ['OpenML100', 'study_14',
'study_34'], 'visibility': 'public', 'status': 'active', 'md5_checksum':
'3c479a6885bfa0438971388283a1ce32'}
```

Dataset Versions

A dataset is uniquely specified by its `data_id`, but not necessarily by its name. Several different "versions" of a dataset with the same name can exist which can contain entirely different datasets. If a particular version of a dataset has been found to contain significant issues, it might be deactivated. Using a name to specify a dataset will yield the earliest version of a dataset that is still active. That means that `fetch_openml(name="miceprotein")` can yield different results at different times if earlier versions become inactive. You can see that the dataset with `data_id` 40966 that we fetched above is the version 1 of the "miceprotein" dataset:

```
>>> mice.details['version']
'1'
```

In fact, this dataset only has one version. The iris dataset on the other hand has multiple versions:

```
>>> iris = fetch_openml(name="iris")
>>> iris.details['version']
'1'
>>> iris.details['id']
'61'

>>> iris_61 = fetch_openml(data_id=61)
>>> iris_61.details['version']
'1'
>>> iris_61.details['id']
'61'

>>> iris_969 = fetch_openml(data_id=969)
>>> iris_969.details['version']
'3'
>>> iris_969.details['id']
'969'
```

Specifying the dataset by the name "iris" yields the lowest version, version 1, with the `data_id` 61. To make sure you always get this exact dataset, it is safest to specify it by the dataset `data_id`. The other dataset, with `data_id` 969, is version 3 (version 2 has become inactive), and contains a binarized version of the data:

```
>>> np.unique(iris_969.target)
array(['N', 'P'], dtype=object)
```

You can also specify both the name and the version, which also uniquely identifies the dataset:

```
>>> iris_version_3 = fetch_openml(name="iris", version=3)
>>> iris_version_3.details['version']
'3'
>>> iris_version_3.details['id']
'969'
```

References:

- Vanschoren, van Rijn, Bischl and Torgo "OpenML: networked science in machine learning", ACM SIGKDD Explorations Newsletter, 15(2), 49-60, 2014.

Loading from external datasets

scikit-learn works on any numeric data stored as numpy arrays or scipy sparse matrices. Other types that are convertible to numeric arrays such as pandas DataFrame are also acceptable.

Here are some recommended ways to load standard columnar data into a format usable by scikit-learn:

- pandas.io provides tools to read data from common formats including CSV, Excel, JSON and SQL. DataFrames may also be constructed from lists of tuples or dicts. Pandas handles heterogeneous data smoothly and provides tools for manipulation and conversion into a numeric array suitable for scikit-learn.
- scipy.io specializes in binary formats often used in scientific computing context such as .mat and .arff
- numpy/routines.io for standard loading of columnar data into numpy arrays
- scikit-learn's `datasets.load_svmlight_file` for the svmlight or libSVM sparse format

- scikit-learn's `datasets.load_files` for directories of text files where the name of each directory is the name of each category and each file inside of each directory corresponds to one sample from that category

For some miscellaneous data such as images, videos, and audio, you may wish to refer to:

- skimage.io or Imageio for loading images and videos into numpy arrays
- scipy.io.wavfile.read for reading WAV files into a numpy array

Categorical (or nominal) features stored as strings (common in pandas DataFrames) will need converting to numerical features using *`sklearn.preprocessing.OneHotEncoder`* or *`sklearn.preprocessing.OrdinalEncoder`* or similar. See *Preprocessing data*.

Note: if you manage your own numerical data it is recommended to use an optimized file format such as HDF5 to reduce data load times. Various libraries such as H5Py, PyTables and pandas provides a Python interface for reading and writing data in that format.

4.8 Computing with scikit-learn

4.8.1 Strategies to scale computationally: bigger data

For some applications the amount of examples, features (or both) and/or the speed at which they need to be processed are challenging for traditional approaches. In these cases scikit-learn has a number of options you can consider to make your system scale.

Scaling with instances using out-of-core learning

Out-of-core (or "external memory") learning is a technique used to learn from data that cannot fit in a computer's main memory (RAM).

Here is a sketch of a system designed to achieve this goal:

1. a way to stream instances
2. a way to extract features from instances
3. an incremental algorithm

Streaming instances

Basically, 1. may be a reader that yields instances from files on a hard drive, a database, from a network stream etc. However, details on how to achieve this are beyond the scope of this documentation.

Extracting features

2. could be any relevant way to extract features among the different *feature extraction* methods supported by scikit-learn. However, when working with data that needs vectorization and where the set of features or values is not known in advance one should take explicit care. A good example is text classification where unknown terms are likely to be found during training. It is possible to use a stateful vectorizer if making multiple passes over the data is reasonable from an application point of view. Otherwise, one can turn up the difficulty by using a stateless feature extractor. Currently the preferred way to do this is to use the so-called *hashing trick* as implemented by *`sklearn.feature_extraction.FeatureHasher`* for datasets with categorical variables represented as list of Python dicts or *`sklearn.feature_extraction.text.HashingVectorizer`* for text documents.

Incremental learning

Finally, for 3. we have a number of options inside scikit-learn. Although not all algorithms can learn incrementally (i.e. without seeing all the instances at once), all estimators implementing the `partial_fit` API are candidates. Actually, the ability to learn incrementally from a mini-batch of instances (sometimes called "online learning") is key to out-of-core learning as it guarantees that at any given time there will be only a small amount of instances in the main memory. Choosing a good size for the mini-batch that balances relevancy and memory footprint could involve some tuning[1].

Here is a list of incremental estimators for different tasks:

- **Classification**
  - *sklearn.naive_bayes.MultinomialNB*
  - *sklearn.naive_bayes.BernoulliNB*
  - *sklearn.linear_model.Perceptron*
  - *sklearn.linear_model.SGDClassifier*
  - *sklearn.linear_model.PassiveAggressiveClassifier*
  - *sklearn.neural_network.MLPClassifier*
- **Regression**
  - *sklearn.linear_model.SGDRegressor*
  - *sklearn.linear_model.PassiveAggressiveRegressor*
  - *sklearn.neural_network.MLPRegressor*
- **Clustering**
  - *sklearn.cluster.MiniBatchKMeans*
  - *sklearn.cluster.Birch*
- **Decomposition / feature Extraction**
  - *sklearn.decomposition.MiniBatchDictionaryLearning*
  - *sklearn.decomposition.IncrementalPCA*
  - *sklearn.decomposition.LatentDirichletAllocation*
- **Preprocessing**
  - *sklearn.preprocessing.StandardScaler*
  - *sklearn.preprocessing.MinMaxScaler*
  - *sklearn.preprocessing.MaxAbsScaler*

For classification, a somewhat important thing to note is that although a stateless feature extraction routine may be able to cope with new/unseen attributes, the incremental learner itself may be unable to cope with new/unseen targets classes. In this case you have to pass all the possible classes to the first `partial_fit` call using the `classes=` parameter.

Another aspect to consider when choosing a proper algorithm is that not all of them put the same importance on each example over time. Namely, the `Perceptron` is still sensitive to badly labeled examples even after many examples whereas the `SGD*` and `PassiveAggressive*` families are more robust to this kind of artifacts. Conversely, the

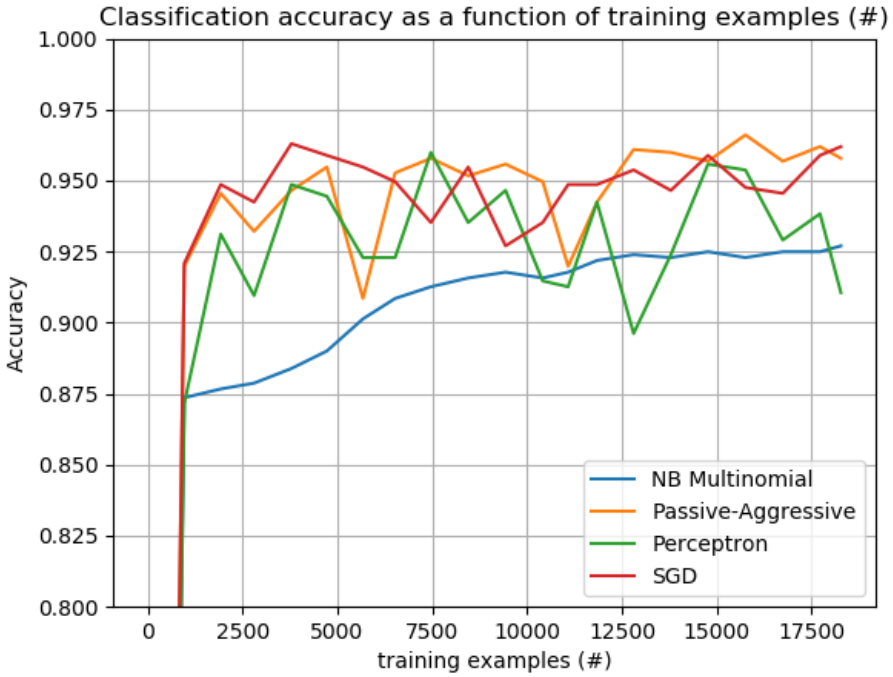
[1] Depending on the algorithm the mini-batch size can influence results or not. SGD*, PassiveAggressive*, and discrete NaiveBayes are truly online and are not affected by batch size. Conversely, MiniBatchKMeans convergence rate is affected by the batch size. Also, its memory footprint can vary dramatically with batch size.

latter also tend to give less importance to remarkably different, yet properly labeled examples when they come late in the stream as their learning rate decreases over time.

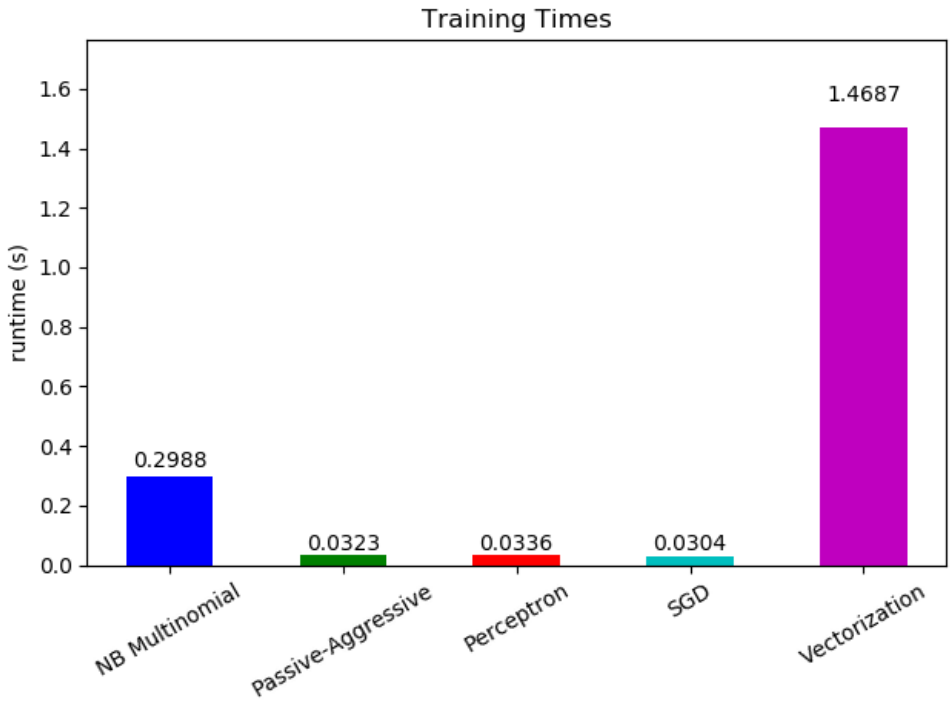
Examples

Finally, we have a full-fledged example of *Out-of-core classification of text documents*. It is aimed at providing a starting point for people wanting to build out-of-core learning systems and demonstrates most of the notions discussed above.

Furthermore, it also shows the evolution of the performance of different algorithms with the number of processed examples.



Now looking at the computation time of the different parts, we see that the vectorization is much more expensive than learning itself. From the different algorithms, `MultinomialNB` is the most expensive, but its overhead can be mitigated by increasing the size of the mini-batches (exercise: change `minibatch_size` to 100 and 10000 in the program and compare).



Notes

4.8.2 Computational Performance

For some applications the performance (mainly latency and throughput at prediction time) of estimators is crucial. It may also be of interest to consider the training throughput but this is often less important in a production setup (where it often takes place offline).

We will review here the orders of magnitude you can expect from a number of scikit-learn estimators in different contexts and provide some tips and tricks for overcoming performance bottlenecks.

Prediction latency is measured as the elapsed time necessary to make a prediction (e.g. in micro-seconds). Latency is often viewed as a distribution and operations engineers often focus on the latency at a given percentile of this distribution (e.g. the 90 percentile).

Prediction throughput is defined as the number of predictions the software can deliver in a given amount of time (e.g. in predictions per second).

An important aspect of performance optimization is also that it can hurt prediction accuracy. Indeed, simpler models (e.g. linear instead of non-linear, or with fewer parameters) often run faster but are not always able to take into account the same exact properties of the data as more complex ones.

Prediction Latency

One of the most straight-forward concerns one may have when using/choosing a machine learning toolkit is the latency at which predictions can be made in a production environment.

The main factors that influence the prediction latency are

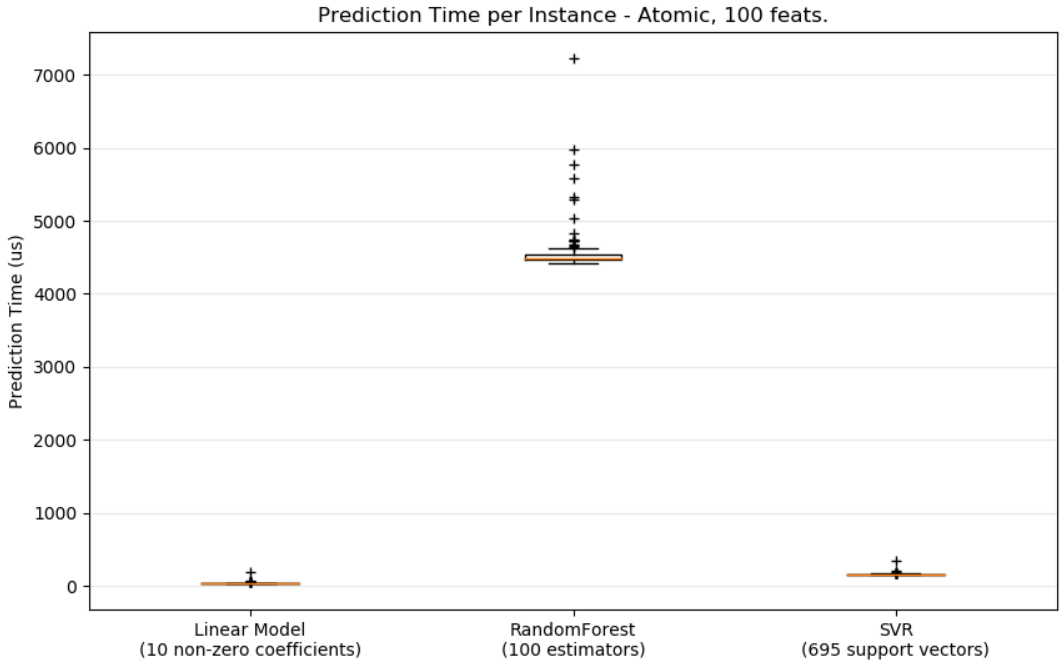
1. Number of features

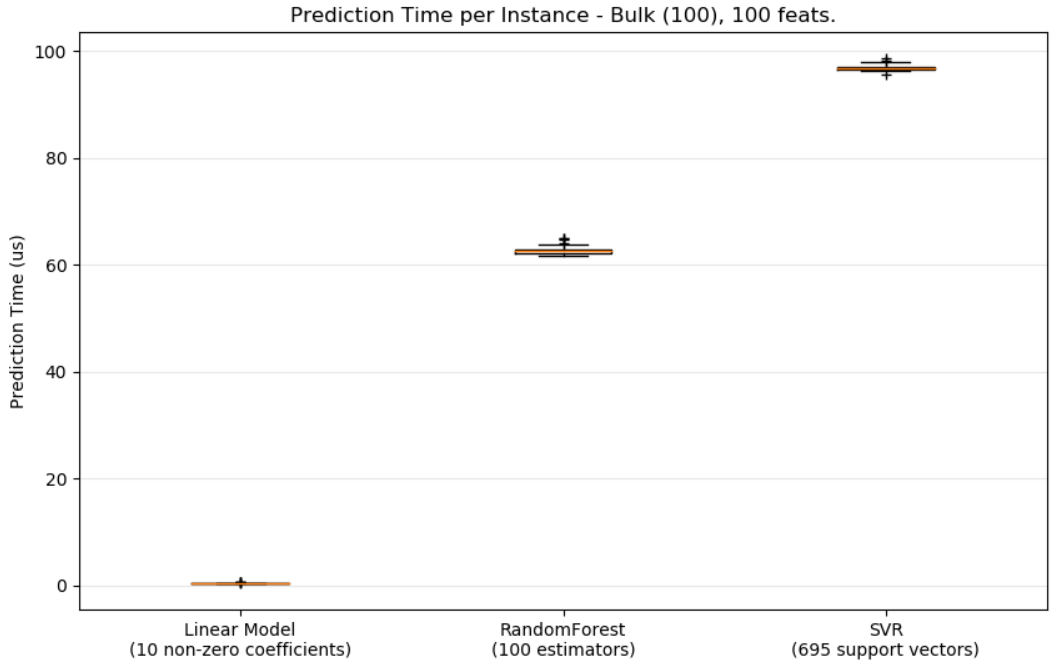
2. Input data representation and sparsity
3. Model complexity
4. Feature extraction

A last major parameter is also the possibility to do predictions in bulk or one-at-a-time mode.

Bulk versus Atomic mode

In general doing predictions in bulk (many instances at the same time) is more efficient for a number of reasons (branching predictability, CPU cache, linear algebra libraries optimizations etc.). Here we see on a setting with few features that independently of estimator choice the bulk mode is always faster, and for some of them by 1 to 2 orders of magnitude:





To benchmark different estimators for your case you can simply change the `n_features` parameter in this example: *Prediction Latency*. This should give you an estimate of the order of magnitude of the prediction latency.

Configuring Scikit-learn for reduced validation overhead

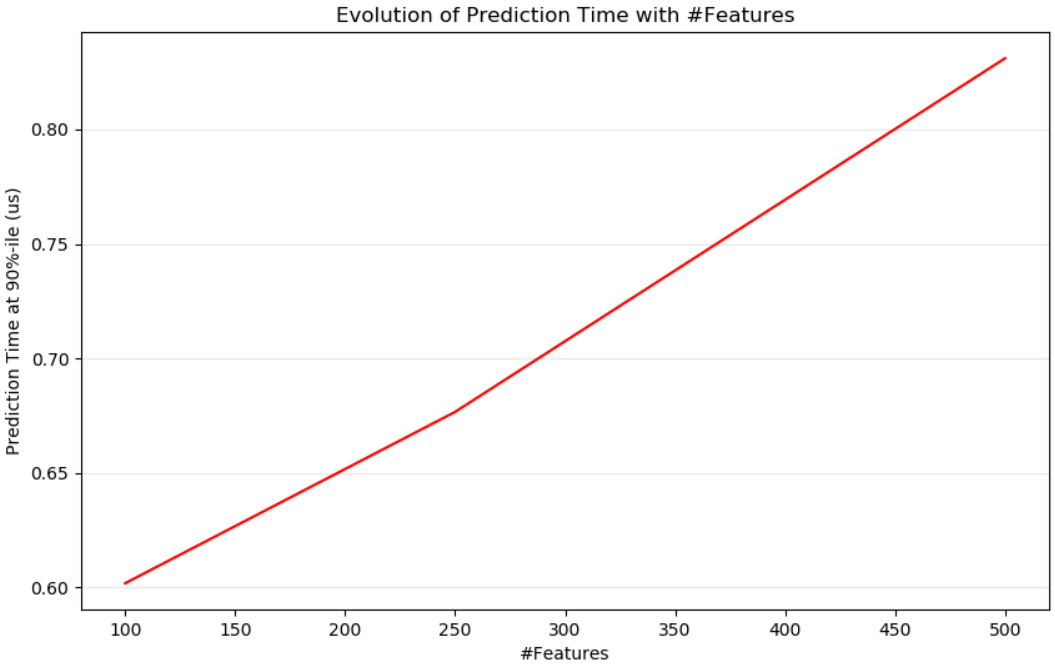
Scikit-learn does some validation on data that increases the overhead per call to `predict` and similar functions. In particular, checking that features are finite (not NaN or infinite) involves a full pass over the data. If you ensure that your data is acceptable, you may suppress checking for finiteness by setting the environment variable `SKLEARN_ASSUME_FINITE` to a non-empty string before importing scikit-learn, or configure it in Python with *`sklearn.set_config`*. For more control than these global settings, a `config_context` allows you to set this configuration within a specified context:

```
>>> import sklearn
>>> with sklearn.config_context(assume_finite=True):
...     pass  # do learning/prediction here with reduced validation
```

Note that this will affect all uses of *`sklearn.utils.assert_all_finite`* within the context.

Influence of the Number of Features

Obviously when the number of features increases so does the memory consumption of each example. Indeed, for a matrix of M instances with N features, the space complexity is in $O(NM)$. From a computing perspective it also means that the number of basic operations (e.g., multiplications for vector-matrix products in linear models) increases too. Here is a graph of the evolution of the prediction latency with the number of features:



Overall you can expect the prediction time to increase at least linearly with the number of features (non-linear cases can happen depending on the global memory footprint and estimator).

Influence of the Input Data Representation

Scipy provides sparse matrix data structures which are optimized for storing sparse data. The main feature of sparse formats is that you don't store zeros so if your data is sparse then you use much less memory. A non-zero value in a sparse (CSR or CSC) representation will only take on average one 32bit integer position + the 64 bit floating point value + an additional 32bit per row or column in the matrix. Using sparse input on a dense (or sparse) linear model can speedup prediction by quite a bit as only the non zero valued features impact the dot product and thus the model predictions. Hence if you have 100 non zeros in 1e6 dimensional space, you only need 100 multiply and add operation instead of 1e6.

Calculation over a dense representation, however, may leverage highly optimised vector operations and multithreading in BLAS, and tends to result in fewer CPU cache misses. So the sparsity should typically be quite high (10% non-zeros max, to be checked depending on the hardware) for the sparse input representation to be faster than the dense input representation on a machine with many CPUs and an optimized BLAS implementation.

Here is sample code to test the sparsity of your input:

```
def sparsity_ratio(X):
    return 1.0 - np.count_nonzero(X) / float(X.shape[0] * X.shape[1])
print("input sparsity ratio:", sparsity_ratio(X))
```

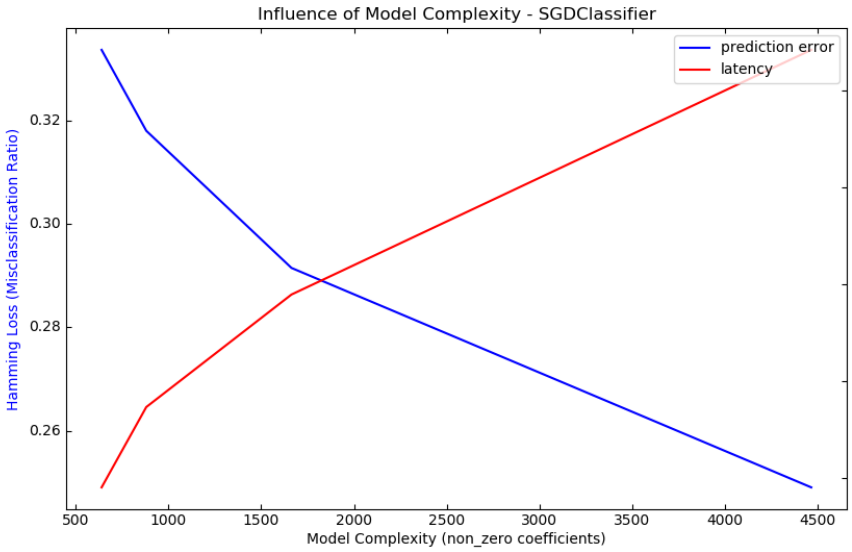
As a rule of thumb you can consider that if the sparsity ratio is greater than 90% you can probably benefit from sparse formats. Check Scipy's sparse matrix formats documentation for more information on how to build (or convert your data to) sparse matrix formats. Most of the time the `CSR` and `CSC` formats work best.

Influence of the Model Complexity

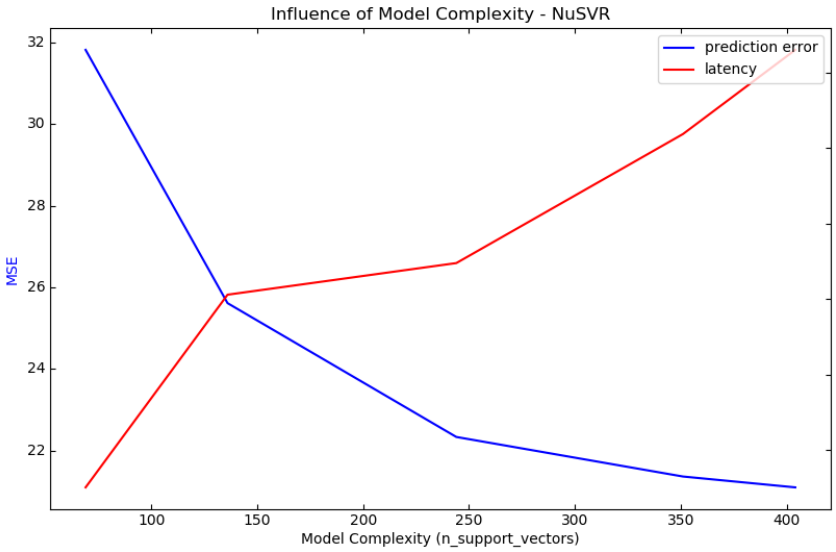
Generally speaking, when model complexity increases, predictive power and latency are supposed to increase. Increasing predictive power is usually interesting, but for many applications we would better not increase prediction latency too much. We will now review this idea for different families of supervised models.

For *sklearn.linear_model* (e.g. Lasso, ElasticNet, SGDClassifier/Regressor, Ridge & RidgeClassifier, PassiveAggressiveClassifier/Regressor, LinearSVC, LogisticRegression...) the decision function that is applied at prediction time is the same (a dot product) , so latency should be equivalent.

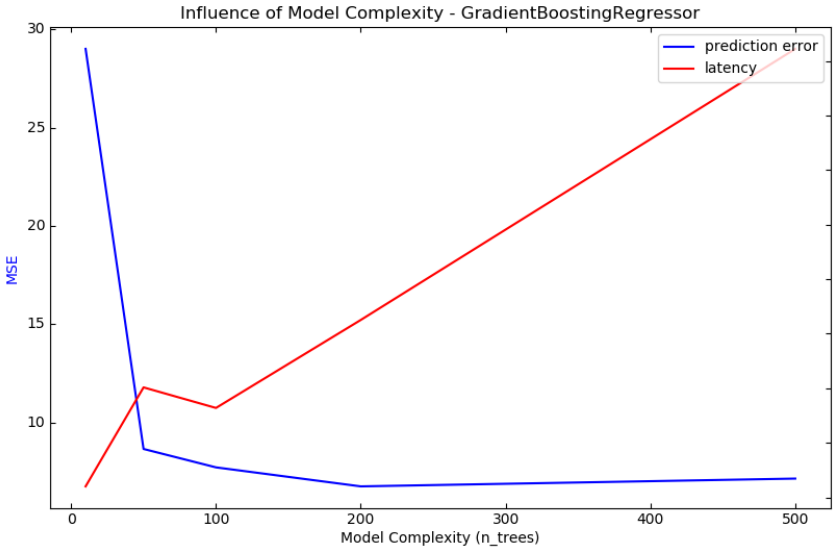
Here is an example using *sklearn.linear_model.SGDClassifier* with the `elasticnet` penalty. The regularization strength is globally controlled by the `alpha` parameter. With a sufficiently high `alpha`, one can then increase the `l1_ratio` parameter of `elasticnet` to enforce various levels of sparsity in the model coefficients. Higher sparsity here is interpreted as less model complexity as we need fewer coefficients to describe it fully. Of course sparsity influences in turn the prediction time as the sparse dot-product takes time roughly proportional to the number of non-zero coefficients.



For the *sklearn.svm* family of algorithms with a non-linear kernel, the latency is tied to the number of support vectors (the fewer the faster). Latency and throughput should (asymptotically) grow linearly with the number of support vectors in a SVC or SVR model. The kernel will also influence the latency as it is used to compute the projection of the input vector once per support vector. In the following graph the `nu` parameter of *sklearn.svm.NuSVR* was used to influence the number of support vectors.



For *sklearn.ensemble* of trees (e.g. RandomForest, GBT, ExtraTrees etc) the number of trees and their depth play the most important role. Latency and throughput should scale linearly with the number of trees. In this case we used directly the `n_estimators` parameter of `sklearn.ensemble.gradient_boosting.GradientBoostingRegressor`.

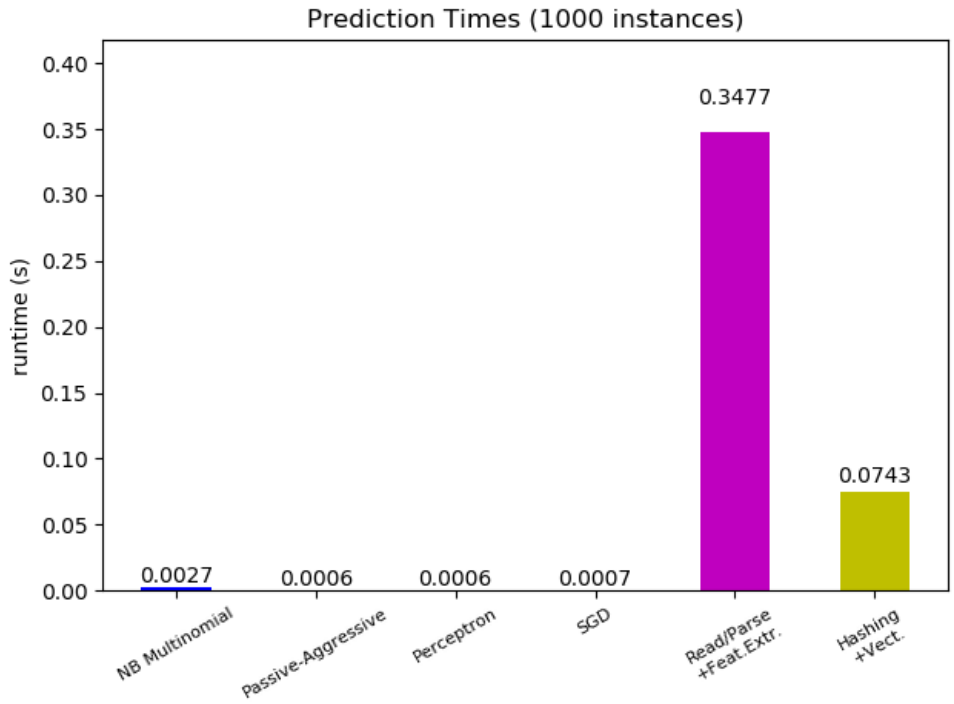


In any case be warned that decreasing model complexity can hurt accuracy as mentioned above. For instance a non-linearly separable problem can be handled with a speedy linear model but prediction power will very likely suffer in the process.

Feature Extraction Latency

Most scikit-learn models are usually pretty fast as they are implemented either with compiled Cython extensions or optimized computing libraries. On the other hand, in many real world applications the feature extraction process (i.e.

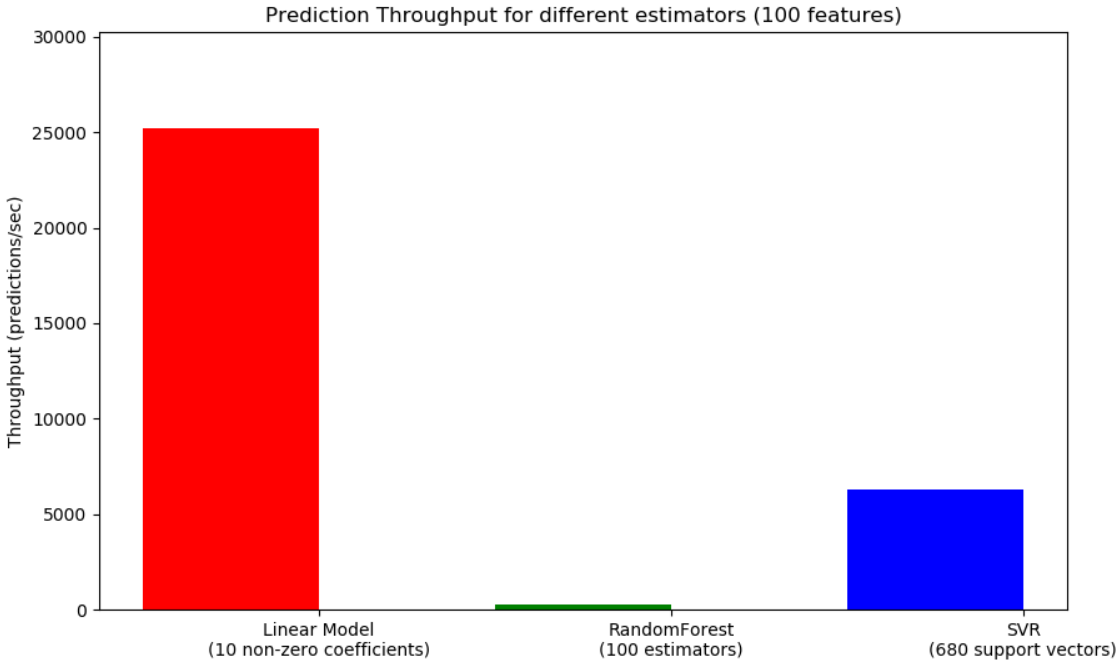
turning raw data like database rows or network packets into numpy arrays) governs the overall prediction time. For example on the Reuters text classification task the whole preparation (reading and parsing SGML files, tokenizing the text and hashing it into a common vector space) is taking 100 to 500 times more time than the actual prediction code, depending on the chosen model.



In many cases it is thus recommended to carefully time and profile your feature extraction code as it may be a good place to start optimizing when your overall latency is too slow for your application.

Prediction Throughput

Another important metric to care about when sizing production systems is the throughput i.e. the number of predictions you can make in a given amount of time. Here is a benchmark from the *Prediction Latency* example that measures this quantity for a number of estimators on synthetic data:



These throughputs are achieved on a single process. An obvious way to increase the throughput of your application is to spawn additional instances (usually processes in Python because of the GIL) that share the same model. One might also add machines to spread the load. A detailed explanation on how to achieve this is beyond the scope of this documentation though.

Tips and Tricks

Linear algebra libraries

As scikit-learn relies heavily on Numpy/Scipy and linear algebra in general it makes sense to take explicit care of the versions of these libraries. Basically, you ought to make sure that Numpy is built using an optimized BLAS / LAPACK library.

Not all models benefit from optimized BLAS and Lapack implementations. For instance models based on (randomized) decision trees typically do not rely on BLAS calls in their inner loops, nor do kernel SVMs (`SVC`, `SVR`, `NuSVC`, `NuSVR`). On the other hand a linear model implemented with a BLAS DGEMM call (via `numpy.dot`) will typically benefit hugely from a tuned BLAS implementation and lead to orders of magnitude speedup over a non-optimized BLAS.

You can display the BLAS / LAPACK implementation used by your NumPy / SciPy / scikit-learn install with the following commands:

```
from numpy.distutils.system_info import get_info
print(get_info('blas_opt'))
print(get_info('lapack_opt'))
```

Optimized BLAS / LAPACK implementations include:

- Atlas (need hardware specific tuning by rebuilding on the target machine)
- OpenBLAS

- MKL
- Apple Accelerate and vecLib frameworks (OSX only)

More information can be found on the Scipy install page and in this blog post from Daniel Nouri which has some nice step by step install instructions for Debian / Ubuntu.

Limiting Working Memory

Some calculations when implemented using standard numpy vectorized operations involve using a large amount of temporary memory. This may potentially exhaust system memory. Where computations can be performed in fixed-memory chunks, we attempt to do so, and allow the user to hint at the maximum size of this working memory (defaulting to 1GB) using *sklearn.set_config* or `config_context`. The following suggests to limit temporary working memory to 128 MiB:

```
>>> import sklearn
>>> with sklearn.config_context(working_memory=128):
...     pass  # do chunked work here
```

An example of a chunked operation adhering to this setting is `metric.pairwise_distances_chunked`, which facilitates computing row-wise reductions of a pairwise distance matrix.

Model Compression

Model compression in scikit-learn only concerns linear models for the moment. In this context it means that we want to control the model sparsity (i.e. the number of non-zero coordinates in the model vectors). It is generally a good idea to combine model sparsity with sparse input data representation.

Here is sample code that illustrates the use of the `sparsify()` method:

```
clf = SGDRegressor(penalty='elasticnet', l1_ratio=0.25)
clf.fit(X_train, y_train).sparsify()
clf.predict(X_test)
```

In this example we prefer the `elasticnet` penalty as it is often a good compromise between model compactness and prediction power. One can also further tune the `l1_ratio` parameter (in combination with the regularization strength `alpha`) to control this tradeoff.

A typical benchmark on synthetic data yields a >30% decrease in latency when both the model and input are sparse (with 0.000024 and 0.027400 non-zero coefficients ratio respectively). Your mileage may vary depending on the sparsity and size of your data and model. Furthermore, sparsifying can be very useful to reduce the memory usage of predictive models deployed on production servers.

Model Reshaping

Model reshaping consists in selecting only a portion of the available features to fit a model. In other words, if a model discards features during the learning phase we can then strip those from the input. This has several benefits. Firstly it reduces memory (and therefore time) overhead of the model itself. It also allows to discard explicit feature selection components in a pipeline once we know which features to keep from a previous run. Finally, it can help reduce processing time and I/O usage upstream in the data access and feature extraction layers by not collecting and building features that are discarded by the model. For instance if the raw data come from a database, it can make it possible to write simpler and faster queries or reduce I/O usage by making the queries return lighter records. At the moment, reshaping needs to be performed manually in scikit-learn. In the case of sparse input (particularly in `CSR` format), it is generally sufficient to not generate the relevant features, leaving their columns empty.

Links

- *scikit-learn developer performance documentation*
- Scipy sparse matrix formats documentation

4.8.3 Parallelism, resource management, and configuration

Parallelism

Some scikit-learn estimators and utilities can parallelize costly operations using multiple CPU cores, thanks to the following components:

- via the joblib library. In this case the number of threads or processes can be controlled with the `n_jobs` parameter.
- via OpenMP, used in C or Cython code.

In addition, some of the numpy routines that are used internally by scikit-learn may also be parallelized if numpy is installed with specific numerical libraries such as MKL, OpenBLAS, or BLIS.

We describe these 3 scenarios in the following subsections.

Joblib-based parallelism

When the underlying implementation uses joblib, the number of workers (threads or processes) that are spawned in parallel can be controlled via the `n_jobs` parameter.

Note: Where (and how) parallelization happens in the estimators is currently poorly documented. Please help us by improving our docs and tackle issue 14228!

Joblib is able to support both multi-processing and multi-threading. Whether joblib chooses to spawn a thread or a process depends on the **backend** that it's using.

Scikit-learn generally relies on the `loky` backend, which is joblib's default backend. Loky is a multi-processing backend. When doing multi-processing, in order to avoid duplicating the memory in each process (which isn't reasonable with big datasets), joblib will create a memmap that all processes can share, when the data is bigger than 1MB.

In some specific cases (when the code that is run in parallel releases the GIL), scikit-learn will indicate to `joblib` that a multi-threading backend is preferable.

As a user, you may control the backend that joblib will use (regardless of what scikit-learn recommends) by using a context manager:

```
from joblib import parallel_backend

with parallel_backend('threading', n_jobs=2):
    # Your scikit-learn code here
```

Please refer to the joblib's docs for more details.

In practice, whether parallelism is helpful at improving runtime depends on many factors. It is usually a good idea to experiment rather than assuming that increasing the number of workers is always a good thing. In some cases it can be highly detrimental to performance to run multiple copies of some estimators or functions in parallel (see oversubscription below).

OpenMP-based parallelism

OpenMP is used to parallelize code written in Cython or C, relying on multi-threading exclusively. By default (and unless joblib is trying to avoid oversubscription), the implementation will use as many threads as possible.

You can control the exact number of threads that are used via the `OMP_NUM_THREADS` environment variable:

```
OMP_NUM_THREADS=4 python my_script.py
```

Parallel Numpy routines from numerical libraries

Scikit-learn relies heavily on NumPy and SciPy, which internally call multi-threaded linear algebra routines implemented in libraries such as MKL, OpenBLAS or BLIS.

The number of threads used by the OpenBLAS, MKL or BLIS libraries can be set via the `MKL_NUM_THREADS`, `OPENBLAS_NUM_THREADS`, and `BLIS_NUM_THREADS` environment variables.

Please note that scikit-learn has no direct control over these implementations. Scikit-learn solely relies on Numpy and Scipy.

Note: At the time of writing (2019), NumPy and SciPy packages distributed on pypi.org (used by `pip`) and on the conda-forge channel are linked with OpenBLAS, while conda packages shipped on the "defaults" channel from anaconda.org are linked by default with MKL.

Oversubscription: spawning too many threads

It is generally recommended to avoid using significantly more processes or threads than the number of CPUs on a machine. Over-subscription happens when a program is running too many threads at the same time.

Suppose you have a machine with 8 CPUs. Consider a case where you're running a `GridSearchCV` (parallelized with joblib) with `n_jobs=8` over a `HistGradientBoostingClassifier` (parallelized with OpenMP). Each instance of `HistGradientBoostingClassifier` will spawn 8 threads (since you have 8 CPUs). That's a total of `8 * 8 = 64` threads, which leads to oversubscription of physical CPU resources and to scheduling overhead.

Oversubscription can arise in the exact same fashion with parallelized routines from MKL, OpenBLAS or BLIS that are nested in joblib calls.

Starting from `joblib >= 0.14`, when the `loky` backend is used (which is the default), joblib will tell its child **processes** to limit the number of threads they can use, so as to avoid oversubscription. In practice the heuristic that joblib uses is to tell the processes to use `max_threads = n_cpus // n_jobs`, via their corresponding environment variable. Back to our example from above, since the joblib backend of `GridSearchCV` is `loky`, each process will only be able to use 1 thread instead of 8, thus mitigating the oversubscription issue.

Note that:

- Manually setting one of the environment variables (`OMP_NUM_THREADS`, `MKL_NUM_THREADS`, `OPENBLAS_NUM_THREADS`, or `BLIS_NUM_THREADS`) will take precedence over what joblib tries to do. The total number of threads will be `n_jobs * <LIB>_NUM_THREADS`. Note that setting this limit will also impact your computations in the main process, which will only use `<LIB>_NUM_THREADS`. Joblib exposes a context manager for finer control over the number of threads in its workers (see joblib docs linked below).
- Joblib is currently unable to avoid oversubscription in a multi-threading context. It can only do so with the `loky` backend (which spawns processes).

You will find additional details about joblib mitigation of oversubscription in joblib documentation.

Configuration switches

Python runtime

sklearn.set_config controls the following behaviors:

assume_finite used to skip validation, which enables faster computations but may lead to segmentation faults if the data contains NaNs.

working_memory the optimal size of temporary arrays used by some algorithms.

Environment variables

These environment variables should be set before importing scikit-learn.

SKLEARN_SITE_JOBLIB When this environment variable is set to a non zero value, scikit-learn uses the site joblib rather than its vendored version. Consequently, joblib must be installed for scikit-learn to run. Note that using the site joblib is at your own risks: the versions of scikit-learn and joblib need to be compatible. Currently, joblib 0.11+ is supported. In addition, dumps from joblib.Memory might be incompatible, and you might loose some caches and have to redownload some datasets.

Deprecated since version 0.21: As of version 0.21 this parameter has no effect, vendored joblib was removed and site joblib is always used.

SKLEARN_ASSUME_FINITE Sets the default value for the `assume_finite` argument of *sklearn.set_config*.

SKLEARN_WORKING_MEMORY Sets the default value for the `working_memory` argument of *sklearn.set_config*.

SKLEARN_SEED Sets the seed of the global random generator when running the tests, for reproducibility.

SKLEARN_SKIP_NETWORK_TESTS When this environment variable is set to a non zero value, the tests that need network access are skipped.

CHAPTER

FIVE

GLOSSARY OF COMMON TERMS AND API ELEMENTS

This glossary hopes to definitively represent the tacit and explicit conventions applied in Scikit-learn and its API, while providing a reference for users and contributors. It aims to describe the concepts and either detail their corresponding API or link to other relevant parts of the documentation which do so. By linking to glossary entries from the API Reference and User Guide, we may minimize redundancy and inconsistency.

We begin by listing general concepts (and any that didn't fit elsewhere), but more specific sets of related terms are listed below: *Class APIs and Estimator Types*, *Target Types*, *Methods*, *Parameters*, *Attributes*, *Data and sample properties*.

5.1 General Concepts

1d

1d array One-dimensional array. A NumPy array whose `.shape` has length 1. A vector.

2d

2d array Two-dimensional array. A NumPy array whose `.shape` has length 2. Often represents a matrix.

API Refers to both the *specific* interfaces for estimators implemented in Scikit-learn and the *generalized* conventions across types of estimators as described in this glossary and *overviewed in the contributor documentation*.

The specific interfaces that constitute Scikit-learn's public API are largely documented in *API Reference*. However we less formally consider anything as public API if none of the identifiers required to access it begins with _. We generally try to maintain *backwards compatibility* for all objects in the public API.

Private API, including functions, modules and methods beginning _ are not assured to be stable.

array-like The most common data format for *input* to Scikit-learn estimators and functions, array-like is any type object for which `numpy.asarray` will produce an array of appropriate shape (usually 1 or 2-dimensional) of appropriate dtype (usually numeric).

This includes:

- a numpy array
- a list of numbers
- a list of length-k lists of numbers for some fixed length k
- a `pandas.DataFrame` with all columns numeric
- a numeric `pandas.Series`

It excludes:

- a *sparse matrix*
- an iterator

- a generator

Note that *output* from scikit-learn estimators and functions (e.g. predictions) should generally be arrays or sparse matrices, or lists thereof (as in multi-output `tree.DecisionTreeClassifier`'s `predict_proba`). An estimator where `predict()` returns a list or a `pandas.Series` is not valid.

attribute

attributes We mostly use attribute to refer to how model information is stored on an estimator during fitting. Any public attribute stored on an estimator instance is required to begin with an alphabetic character and end in a single underscore if it is set in *fit* or *partial_fit*. These are what is documented under an estimator's *Attributes* documentation. The information stored in attributes is usually either: sufficient statistics used for prediction or transformation; *transductive* outputs such as *labels_* or *embedding_*; or diagnostic data, such as *feature_importances_*. Common attributes are listed *below*.

A public attribute may have the same name as a constructor *parameter*, with a _ appended. This is used to store a validated or estimated version of the user's input. For example, `decomposition.PCA` is constructed with an `n_components` parameter. From this, together with other parameters and the data, PCA estimates the attribute `n_components_`.

Further private attributes used in prediction/transformation/etc. may also be set when fitting. These begin with a single underscore and are not assured to be stable for public access.

A public attribute on an estimator instance that does not end in an underscore should be the stored, unmodified value of an `__init__` *parameter* of the same name. Because of this equivalence, these are documented under an estimator's *Parameters* documentation.

backwards compatibility We generally try to maintain backwards compatibility (i.e. interfaces and behaviors may be extended but not changed or removed) from release to release but this comes with some exceptions:

Public API only The behaviour of objects accessed through private identifiers (those beginning _) may be changed arbitrarily between versions.

As documented We will generally assume that the users have adhered to the documented parameter types and ranges. If the documentation asks for a list and the user gives a tuple, we do not assure consistent behavior from version to version.

Deprecation Behaviors may change following a *deprecation* period (usually two releases long). Warnings are issued using Python's `warnings` module.

Keyword arguments We may sometimes assume that all optional parameters (other than X and y to *fit* and similar methods) are passed as keyword arguments only and may be positionally reordered.

Bug fixes and enhancements Bug fixes and – less often – enhancements may change the behavior of estimators, including the predictions of an estimator trained on the same data and *random_state*. When this happens, we attempt to note it clearly in the changelog.

Serialization We make no assurances that pickling an estimator in one version will allow it to be unpickled to an equivalent model in the subsequent version. (For estimators in the sklearn package, we issue a warning when this unpickling is attempted, even if it may happen to work.) See *Security & maintainability limitations*.

`utils.estimator_checks.check_estimator` We provide limited backwards compatibility assurances for the estimator checks: we may add extra requirements on estimators tested with this function, usually when these were informally assumed but not formally tested.

Despite this informal contract with our users, the software is provided as is, as stated in the licence. When a release inadvertently introduces changes that are not backwards compatible, these are known as software regressions.

callable A function, class or an object which implements the `__call__` method; anything that returns True when the argument of callable().

categorical feature A categorical or nominal *feature* is one that has a finite set of discrete values across the population of data. These are commonly represented as columns of integers or strings. Strings will be rejected by most scikit-learn estimators, and integers will be treated as ordinal or count-valued. For the use with most estimators, categorical variables should be one-hot encoded. Notable exceptions include tree-based models such as random forests and gradient boosting models that often work better and faster with integer-coded categorical variables. `OrdinalEncoder` helps encoding string-valued categorical features as ordinal integers, and `OneHotEncoder` can be used to one-hot encode categorical features. See also *Encoding categorical features* and the categorical-encoding package for tools related to encoding categorical features.

clone

cloned To copy an *estimator instance* and create a new one with identical *parameters*, but without any fitted *attributes*, using `clone`.

When `fit` is called, a *meta-estimator* usually clones a wrapped estimator instance before fitting the cloned instance. (Exceptions, for legacy reasons, include `Pipeline` and `FeatureUnion`.)

common tests This refers to the tests run on almost every estimator class in Scikit-learn to check they comply with basic API conventions. They are available for external use through `utils.estimator_checks.check_estimator`, with most of the implementation in `sklearn/utils/estimator_checks.py`.

Note: Some exceptions to the common testing regime are currently hard-coded into the library, but we hope to replace this by marking exceptional behaviours on the estimator using semantic *estimator tags*.

deprecation We use deprecation to slowly violate our *backwards compatibility* assurances, usually to to:

- change the default value of a parameter; or
- remove a parameter, attribute, method, class, etc.

We will ordinarily issue a warning when a deprecated element is used, although there may be limitations to this. For instance, we will raise a warning when someone sets a parameter that has been deprecated, but may not when they access that parameter's attribute on the estimator instance.

See the *Contributors' Guide*.

dimensionality May be used to refer to the number of *features* (i.e. *n_features*), or columns in a 2d feature matrix. Dimensions are, however, also used to refer to the length of a NumPy array's shape, distinguishing a 1d array from a 2d matrix.

docstring The embedded documentation for a module, class, function, etc., usually in code as a string at the beginning of the object's definition, and accessible as the object's `__doc__` attribute.

We try to adhere to PEP257, and follow NumpyDoc conventions.

double underscore

double underscore notation When specifying parameter names for nested estimators, `__` may be used to separate between parent and child in some contexts. The most common use is when setting parameters through a meta-estimator with *set_params* and hence in specifying a search grid in *parameter search*. See *parameter*. It is also used in `pipeline.Pipeline.fit` for passing *sample properties* to the `fit` methods of estimators in the pipeline.

dtype

data type NumPy arrays assume a homogeneous data type throughout, available in the `.dtype` attribute of an array (or sparse matrix). We generally assume simple data types for scikit-learn data: float or integer. We may support object or string data types for arrays before encoding or vectorizing. Our estimators do not work with struct arrays, for instance.

TODO: Mention efficiency and precision issues; casting policy.

duck typing We try to apply duck typing to determine how to handle some input values (e.g. checking whether a given estimator is a classifier). That is, we avoid using `isinstance` where possible, and rely on the presence or absence of attributes to determine an object's behaviour. Some nuance is required when following this approach:

- For some estimators, an attribute may only be available once it is *fitted*. For instance, we cannot a priori determine if *predict_proba* is available in a grid search where the grid includes alternating between a probabilistic and a non-probabilistic predictor in the final step of the pipeline. In the following, we can only determine if `clf` is probabilistic after fitting it on some data:

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.linear_model import SGDClassifier
>>> clf = GridSearchCV(SGDClassifier(),
...                    param_grid={'loss': ['log', 'hinge']})
```

  This means that we can only check for duck-typed attributes after fitting, and that we must be careful to make *meta-estimators* only present attributes according to the state of the underlying estimator after fitting.

- Checking if an attribute is present (using `hasattr`) is in general just as expensive as getting the attribute (`getattr` or dot notation). In some cases, getting the attribute may indeed be expensive (e.g. for some implementations of *feature_importances_*, which may suggest this is an API design flaw). So code which does `hasattr` followed by `getattr` should be avoided; `getattr` within a try-except block is preferred.

- For determining some aspects of an estimator's expectations or support for some feature, we use *estimator tags* instead of duck typing.

early stopping This consists in stopping an iterative optimization method before the convergence of the training loss, to avoid over-fitting. This is generally done by monitoring the generalization score on a validation set. When available, it is activated through the parameter `early_stopping` or by setting a positive *n_iter_no_change*.

estimator instance We sometimes use this terminology to distinguish an *estimator* class from a constructed instance. For example, in the following, `cls` is an estimator class, while `est1` and `est2` are instances:

```
cls = RandomForestClassifier
est1 = cls()
est2 = RandomForestClassifier()
```

examples We try to give examples of basic usage for most functions and classes in the API:

- as doctests in their docstrings (i.e. within the `sklearn/` library code itself).

- as examples in the *example gallery* rendered (using sphinx-gallery) from scripts in the `examples/` directory, exemplifying key features or parameters of the estimator/function. These should also be referenced from the User Guide.

- sometimes in the *User Guide* (built from `doc/`) alongside a technical description of the estimator.

evaluation metric

evaluation metrics Evaluation metrics give a measure of how well a model performs. We may use this term specifically to refer to the functions in `metrics` (disregarding `metrics.pairwise`), as distinct from the *score* method and the *scoring* API used in cross validation. See *Metrics and scoring: quantifying the quality of predictions*.

These functions usually accept a ground truth (or the raw data where the metric evaluates clustering without a ground truth) and a prediction, be it the output of *predict* (`y_pred`), of *predict_proba* (`y_proba`), or of an arbitrary score function including *decision_function* (`y_score`). Functions are usually named to end with `_score` if a greater score indicates a better model, and `_loss` if a lesser score indicates a better model. This diversity of interface motivates the scoring API.

Note that some estimators can calculate metrics that are not included in `metrics` and are estimator-specific, notably model likelihoods.

estimator tags A proposed feature (e.g. #8022) by which the capabilities of an estimator are described through a set of semantic tags. This would enable some runtime behaviors based on estimator inspection, but it also allows each estimator to be tested for appropriate invariances while being excepted from other *common tests*.

Some aspects of estimator tags are currently determined through the *duck typing* of methods like `predict_proba` and through some special attributes on estimator objects:

`_estimator_type` This string-valued attribute identifies an estimator as being a classifier, regressor, etc. It is set by mixins such as *`base.ClassifierMixin`*, but needs to be more explicitly adopted on a *meta-estimator*. Its value should usually be checked by way of a helper such as *`base.is_classifier`*.

`_pairwise` This boolean attribute indicates whether the data (`X`) passed to `fit` and similar methods consists of pairwise measures over samples rather than a feature representation for each sample. It is usually `True` where an estimator has a `metric` or `affinity` or `kernel` parameter with value 'precomputed'. Its primary purpose is that when a *meta-estimator* extracts a sub-sample of data intended for a pairwise estimator, the data needs to be indexed on both axes, while other data is indexed only on the first axis.

feature

features

feature vector In the abstract, a feature is a function (in its mathematical sense) mapping a sampled object to a numeric or categorical quantity. "Feature" is also commonly used to refer to these quantities, being the individual elements of a vector representing a sample. In a data matrix, features are represented as columns: each column contains the result of applying a feature function to a set of samples.

Elsewhere features are known as attributes, predictors, regressors, or independent variables.

Nearly all estimators in scikit-learn assume that features are numeric, finite and not missing, even when they have semantically distinct domains and distributions (categorical, ordinal, count-valued, real-valued, interval). See also *categorical feature* and *missing values*.

`n_features` indicates the number of features in a dataset.

fitting Calling *fit* (or *fit_transform*, *fit_predict*, etc.) on an estimator.

fitted The state of an estimator after *fitting*.

There is no conventional procedure for checking if an estimator is fitted. However, an estimator that is not fitted:

- should raise *`exceptions.NotFittedError`* when a prediction method (*predict*, *transform*, etc.) is called. (*`utils.validation.check_is_fitted`* is used internally for this purpose.)
- should not have any *attributes* beginning with an alphabetic character and ending with an underscore. (Note that a descriptor for the attribute may still be present on the class, but hasattr should return False)

function We provide ad hoc function interfaces for many algorithms, while *estimator* classes provide a more consistent interface.

In particular, Scikit-learn may provide a function interface that fits a model to some data and returns the learnt model parameters, as in *`linear_model.enet_path`*. For transductive models, this also returns the embedding or cluster labels, as in *`manifold.spectral_embedding`* or *`cluster.dbscan`*. Many preprocessing transformers also provide a function interface, akin to calling *fit_transform*, as in *`preprocessing.maxabs_scale`*. Users should be careful to avoid *data leakage* when making use of these `fit_transform`-equivalent functions.

We do not have a strict policy about when to or when not to provide function forms of estimators, but maintainers should consider consistency with existing interfaces, and whether providing a function would lead users astray from best practices (as regards data leakage, etc.)

gallery See *examples*.

hyperparameter

hyper-parameter See *parameter*.

impute

imputation Most machine learning algorithms require that their inputs have no *missing values*, and will not work if this requirement is violated. Algorithms that attempt to fill in (or impute) missing values are referred to as imputation algorithms.

indexable An *array-like*, *sparse matrix*, pandas DataFrame or sequence (usually a list).

induction

inductive Inductive (contrasted with *transductive*) machine learning builds a model of some data that can then be applied to new instances. Most estimators in Scikit-learn are inductive, having *predict* and/or *transform* methods.

joblib A Python library (https://joblib.readthedocs.io) used in Scikit-learn to facilite simple parallelism and caching. Joblib is oriented towards efficiently working with numpy arrays, such as through use of *memory mapping*. See *Parallelism* for more information.

label indicator matrix

multilabel indicator matrix

multilabel indicator matrices The format used to represent multilabel data, where each row of a 2d array or sparse matrix corresponds to a sample, each column corresponds to a class, and each element is 1 if the sample is labeled with the class and 0 if not.

leakage

data leakage A problem in cross validation where generalization performance can be over-estimated since knowledge of the test data was inadvertently included in training a model. This is a risk, for instance, when applying a *transformer* to the entirety of a dataset rather than each training portion in a cross validation split.

We aim to provide interfaces (such as `pipeline` and `model_selection`) that shield the user from data leakage.

memmapping

memory map

memory mapping A memory efficiency strategy that keeps data on disk rather than copying it into main memory. Memory maps can be created for arrays that can be read, written, or both, using `numpy.memmap`. When using *joblib* to parallelize operations in Scikit-learn, it may automatically memmap large arrays to reduce memory duplication overhead in multiprocessing.

missing values Most Scikit-learn estimators do not work with missing values. When they do (e.g. in *`impute.SimpleImputer`*), NaN is the preferred representation of missing values in float arrays. If the array has integer dtype, NaN cannot be represented. For this reason, we support specifying another `missing_values` value when *imputation* or learning can be performed in integer space. *Unlabeled data* is a special case of missing values in the *target*.

`n_features` The number of *features*.

`n_outputs` The number of *outputs* in the *target*.

`n_samples` The number of *samples*.

`n_targets` Synonym for *n_outputs*.

narrative docs

narrative documentation An alias for *User Guide*, i.e. documentation written in `doc/modules/`. Unlike the *API reference* provided through docstrings, the User Guide aims to:

- group tools provided by Scikit-learn together thematically or in terms of usage;
- motivate why someone would use each particular tool, often through comparison;
- provide both intuitive and technical descriptions of tools;
- provide or link to *examples* of using key features of a tool.

np A shorthand for Numpy due to the conventional import statement:

```
import numpy as np
```

online learning Where a model is iteratively updated by receiving each batch of ground truth *targets* soon after making predictions on corresponding batch of data. Intrinsically, the model must be usable for prediction after each batch. See *partial_fit*.

out-of-core An efficiency strategy where not all the data is stored in main memory at once, usually by performing learning on batches of data. See *partial_fit*.

outputs Individual scalar/categorical variables per sample in the *target*. For example, in multilabel classification each possible label corresponds to a binary output. Also called *responses*, *tasks* or *targets*. See *multiclass multioutput* and *continuous multioutput*.

pair A tuple of length two.

parameter

parameters

param

params We mostly use *parameter* to refer to the aspects of an estimator that can be specified in its construction. For example, `max_depth` and `random_state` are parameters of `RandomForestClassifier`. Parameters to an estimator's constructor are stored unmodified as attributes on the estimator instance, and conventionally start with an alphabetic character and end with an alphanumeric character. Each estimator's constructor parameters are described in the estimator's docstring.

We do not use parameters in the statistical sense, where parameters are values that specify a model and can be estimated from data. What we call parameters might be what statisticians call hyperparameters to the model: aspects for configuring model structure that are often not directly learnt from data. However, our parameters are also used to prescribe modeling operations that do not affect the learnt model, such as *n_jobs* for controlling parallelism.

When talking about the parameters of a *meta-estimator*, we may also be including the parameters of the estimators wrapped by the meta-estimator. Ordinarily, these nested parameters are denoted by using a *double underscore* (`__`) to separate between the estimator-as-parameter and its parameter. Thus `clf = BaggingClassifier(base_estimator=DecisionTreeClassifier(max_depth=3))` has a deep parameter `base_estimator__max_depth` with value `3`, which is accessible with `clf.base_estimator.max_depth` or `clf.get_params()['base_estimator__max_depth']`.

The list of parameters and their current values can be retrieved from an *estimator instance* using its *get_params* method.

Between construction and fitting, parameters may be modified using *set_params*. To enable this, parameters are not ordinarily validated or altered when the estimator is constructed, or when each parameter is set. Parameter validation is performed when *fit* is called.

Common parameters are listed *below*.

pairwise metric

pairwise metrics In its broad sense, a pairwise metric defines a function for measuring similarity or dissimilarity between two samples (with each ordinarily represented as a *feature vector*). We particularly provide implementations of distance metrics (as well as improper metrics like Cosine Distance) through *metrics.pairwise_distances*, and of kernel functions (a constrained class of similarity functions) in `metrics.pairwise_kernels`. These can compute pairwise distance matrices that are symmetric and hence store data redundantly.

See also *precomputed* and *metric*.

Note that for most distance metrics, we rely on implementations from `scipy.spatial.distance`, but may reimplement for efficiency in our context. The `neighbors` module also duplicates some metric implementations for integration with efficient binary tree search data structures.

pd A shorthand for Pandas due to the conventional import statement:

```
import pandas as pd
```

precomputed Where algorithms rely on *pairwise metrics*, and can be computed from pairwise metrics alone, we often allow the user to specify that the *X* provided is already in the pairwise (dis)similarity space, rather than in a feature space. That is, when passed to *fit*, it is a square, symmetric matrix, with each vector indicating (dis)similarity to every sample, and when passed to prediction/transformation methods, each row corresponds to a testing sample and each column to a training sample.

Use of precomputed X is usually indicated by setting a `metric`, `affinity` or `kernel` parameter to the string 'precomputed'. An estimator should mark itself as being *_pairwise* if this is the case.

rectangular Data that can be represented as a matrix with *samples* on the first axis and a fixed, finite set of *features* on the second is called rectangular.

This term excludes samples with non-vectorial structure, such as text, an image of arbitrary size, a time series of arbitrary length, a set of vectors, etc. The purpose of a *vectorizer* is to produce rectangular forms of such data.

sample

samples We usually use this term as a noun to indicate a single feature vector. Elsewhere a sample is called an instance, data point, or observation. `n_samples` indicates the number of samples in a dataset, being the number of rows in a data array *X*.

sample property

sample properties A sample property is data for each sample (e.g. an array of length n_samples) passed to an estimator method or a similar function, alongside but distinct from the *features* (`X`) and *target* (`y`). The most prominent example is *sample_weight*; see others at *Data and sample properties*.

As of version 0.19 we do not have a consistent approach to handling sample properties and their routing in *meta-estimators*, though a `fit_params` parameter is often used.

scikit-learn-contrib A venue for publishing Scikit-learn-compatible libraries that are broadly authorized by the core developers and the contrib community, but not maintained by the core developer team. See https://scikit-learn-contrib.github.io.

scikit-learn enhancement proposals

SLEP

SLEPs Changes to the API principles and changes to dependencies or supported versions happen via a *SLEP* and follows the decision-making process outlined in *Scikit-learn governance and decision-making*. For all votes, a proposal must have been made public and discussed before the vote. Such proposal must be a consolidated document, in the form of a 'Scikit-Learn Enhancement Proposal' (SLEP), rather than a long discussion on an issue. A SLEP must be submitted as a pull-request to enhancement proposals using the SLEP template.

semi-supervised

semi-supervised learning

semisupervised Learning where the expected prediction (label or ground truth) is only available for some samples provided as training data when *fitting* the model. We conventionally apply the label `-1` to *unlabeled* samples in semi-supervised classification.

sparse matrix

sparse graph A representation of two-dimensional numeric data that is more memory efficient the corresponding dense numpy array where almost all elements are zero. We use the `scipy.sparse` framework, which provides several underlying sparse data representations, or *formats*. Some formats are more efficient than others for particular tasks, and when a particular format provides especial benefit, we try to document this fact in Scikit-learn parameter descriptions.

Some sparse matrix formats (notably CSR, CSC, COO and LIL) distinguish between *implicit* and *explicit* zeros. Explicit zeros are stored (i.e. they consume memory in a `data` array) in the data structure, while implicit zeros correspond to every element not otherwise defined in explicit storage.

Two semantics for sparse matrices are used in Scikit-learn:

matrix semantics The sparse matrix is interpreted as an array with implicit and explicit zeros being interpreted as the number 0. This is the interpretation most often adopted, e.g. when sparse matrices are used for feature matrices or *multilabel indicator matrices*.

graph semantics As with `scipy.sparse.csgraph`, explicit zeros are interpreted as the number 0, but implicit zeros indicate a masked or absent value, such as the absence of an edge between two vertices of a graph, where an explicit value indicates an edge's weight. This interpretation is adopted to represent connectivity in clustering, in representations of nearest neighborhoods (e.g. *`neighbors.kneighbors_graph`*), and for precomputed distance representation where only distances in the neighborhood of each point are required.

When working with sparse matrices, we assume that it is sparse for a good reason, and avoid writing code that densifies a user-provided sparse matrix, instead maintaining sparsity or raising an error if not possible (i.e. if an estimator does not / cannot support sparse matrices).

supervised

supervised learning Learning where the expected prediction (label or ground truth) is available for each sample when *fitting* the model, provided as *y*. This is the approach taken in a *classifier* or *regressor* among other estimators.

target

targets The *dependent variable* in *supervised* (and *semisupervised*) learning, passed as *y* to an estimator's *fit* method. Also known as *dependent variable*, *outcome variable*, *response variable*, *ground truth* or *label*. Scikit-learn works with targets that have minimal structure: a class from a finite set, a finite real-valued number, multiple classes, or multiple numbers. See *Target Types*.

transduction

transductive A transductive (contrasted with *inductive*) machine learning method is designed to model a specific dataset, but not to apply that model to unseen data. Examples include *`manifold.TSNE`*, *`cluster.AgglomerativeClustering`* and *`neighbors.LocalOutlierFactor`*.

unlabeled

unlabeled data Samples with an unknown ground truth when fitting; equivalently, *missing values* in the *target*. See also *semisupervised* and *unsupervised* learning.

unsupervised

unsupervised learning Learning where the expected prediction (label or ground truth) is not available for each sample when *fitting* the model, as in *clusterers* and *outlier detectors*. Unsupervised estimators ignore any *y* passed to *fit*.

5.2 Class APIs and Estimator Types

classifier

classifiers A *supervised* (or *semi-supervised*) *predictor* with a finite set of discrete possible output values.

A classifier supports modeling some of *binary*, *multiclass*, *multilabel*, or *multiclass multioutput* targets. Within scikit-learn, all classifiers support multi-class classification, defaulting to using a one-vs-rest strategy over the binary classification problem.

Classifiers must store a *classes_* attribute after fitting, and usually inherit from `base.ClassifierMixin`, which sets their *_estimator_type* attribute.

A classifier can be distinguished from other estimators with `is_classifier`.

A classifier must implement:

- *fit*
- *predict*
- *score*

It may also be appropriate to implement *decision_function*, *predict_proba* and *predict_log_proba*.

clusterer

clusterers A *unsupervised predictor* with a finite set of discrete output values.

A clusterer usually stores *labels_* after fitting, and must do so if it is *transductive*.

A clusterer must implement:

- *fit*
- *fit_predict* if *transductive*
- *predict* if *inductive*

density estimator TODO

estimator

estimators An object which manages the estimation and decoding of a model. The model is estimated as a deterministic function of:

- *parameters* provided in object construction or with *set_params*;
- the global `numpy.random` random state if the estimator's *random_state* parameter is set to None; and
- any data or *sample properties* passed to the most recent call to *fit*, *fit_transform* or *fit_predict*, or data similarly passed in a sequence of calls to *partial_fit*.

The estimated model is stored in public and private *attributes* on the estimator instance, facilitating decoding through prediction and transformation methods.

Estimators must provide a *fit* method, and should provide *set_params* and *get_params*, although these are usually provided by inheritance from `base.BaseEstimator`.

The core functionality of some estimators may also be available as a *function*.

feature extractor

feature extractors A *transformer* which takes input where each sample is not represented as an *array-like* object of fixed length, and produces an *array-like* object of *features* for each sample (and thus a 2-dimensional array-like for a set of samples). In other words, it (lossily) maps a non-rectangular data representation into *rectangular* data.

Feature extractors must implement at least:

- *fit*
- *transform*
- *get_feature_names*

meta-estimator

meta-estimators

metaestimator

metaestimators An *estimator* which takes another estimator as a parameter. Examples include `pipeline.Pipeline`, `model_selection.GridSearchCV`, `feature_selection.SelectFromModel` and `ensemble.BaggingClassifier`.

In a meta-estimator's *fit* method, any contained estimators should be *cloned* before they are fit (although FIXME: Pipeline and FeatureUnion do not do this currently). An exception to this is that an estimator may explicitly document that it accepts a prefitted estimator (e.g. using `prefit=True` in `feature_selection.SelectFromModel`). One known issue with this is that the prefitted estimator will lose its model if the meta-estimator is cloned. A meta-estimator should have `fit` called before prediction, even if all contained estimators are prefitted.

In cases where a meta-estimator's primary behaviors (e.g. *predict* or *transform* implementation) are functions of prediction/transformation methods of the provided *base estimator* (or multiple base estimators), a meta-estimator should provide at least the standard methods provided by the base estimator. It may not be possible to identify which methods are provided by the underlying estimator until the meta-estimator has been *fitted* (see also *duck typing*), for which `utils.metaestimators.if_delegate_has_method` may help. It should also provide (or modify) the *estimator tags* and *classes_* attribute provided by the base estimator.

Meta-estimators should be careful to validate data as minimally as possible before passing it to an underlying estimator. This saves computation time, and may, for instance, allow the underlying estimator to easily work with data that is not *rectangular*.

outlier detector

outlier detectors An *unsupervised* binary *predictor* which models the distinction between core and outlying samples.

Outlier detectors must implement:

- *fit*
- *fit_predict* if *transductive*
- *predict* if *inductive*

Inductive outlier detectors may also implement *decision_function* to give a normalized inlier score where outliers have score below 0. *score_samples* may provide an unnormalized score per sample.

predictor

predictors An *estimator* supporting *predict* and/or *fit_predict*. This encompasses *classifier*, *regressor*, *outlier detector* and *clusterer*.

In statistics, "predictors" refers to *features*.

regressor

regressors A *supervised* (or *semi-supervised*) *predictor* with *continuous* output values.

Regressors usually inherit from `base.RegressorMixin`, which sets their *_estimator_type* attribute.

A regressor can be distinguished from other estimators with `is_regressor`.

A regressor must implement:

- *fit*
- *predict*
- *score*

transformer

transformers An estimator supporting *transform* and/or *fit_transform*. A purely *transductive* transformer, such as `manifold.TSNE`, may not implement `transform`.

vectorizer

vectorizers See *feature extractor*.

There are further APIs specifically related to a small family of estimators, such as:

cross-validation splitter

CV splitter

cross-validation generator A non-estimator family of classes used to split a dataset into a sequence of train and test portions (see *Cross-validation: evaluating estimator performance*), by providing *split* and *get_n_splits* methods. Note that unlike estimators, these do not have *fit* methods and do not provide *set_params* or *get_params*. Parameter validation may be performed in `__init__`.

cross-validation estimator An estimator that has built-in cross-validation capabilities to automatically select the best hyper-parameters (see the *User Guide*). Some example of cross-validation estimators are `ElasticNetCV` and `LogisticRegressionCV`. Cross-validation estimators are named `EstimatorCV` and tend to be roughly equivalent to `GridSearchCV(Estimator(), ...)`. The advantage of using a cross-validation estimator over the canonical *Estimator* class along with *grid search* is that they can take advantage of warm-starting by reusing precomputed results in the previous steps of the cross-validation process. This generally leads to speed improvements. An exception is the `RidgeCV` class, which can instead perform efficient Leave-One-Out CV.

scorer A non-estimator callable object which evaluates an estimator on given test data, returning a number. Unlike *evaluation metrics*, a greater returned number must correspond with a *better* score. See *The scoring parameter: defining model evaluation rules*.

Further examples:

- `neighbors.DistanceMetric`
- `gaussian_process.kernels.Kernel`
- `tree.Criterion`

5.3 Target Types

binary A classification problem consisting of two classes. A binary target may represented as for a *multiclass* problem but with only two labels. A binary decision function is represented as a 1d array.

Semantically, one class is often considered the "positive" class. Unless otherwise specified (e.g. using *pos_label* in *evaluation metrics*), we consider the class label with the greater value (numerically or lexicographically) as the positive class: of labels [0, 1], 1 is the positive class; of [1, 2], 2 is the positive class; of ['no', 'yes'], 'yes' is the positive class; of ['no', 'YES'], 'no' is the positive class. This affects the output of *decision_function*, for instance.

Note that a dataset sampled from a multiclass `y` or a continuous `y` may appear to be binary.

`type_of_target` will return 'binary' for binary input, or a similar array with only a single class present.

continuous A regression problem where each sample's target is a finite floating point number, represented as a 1-dimensional array of floats (or sometimes ints).

type_of_target will return 'continuous' for continuous input, but if the data is all integers, it will be identified as 'multiclass'.

continuous multioutput

multioutput continuous A regression problem where each sample's target consists of `n_outputs` *outputs*, each one a finite floating point number, for a fixed int `n_outputs > 1` in a particular dataset.

Continuous multioutput targets are represented as multiple *continuous* targets, horizontally stacked into an array of shape `(n_samples, n_outputs)`.

type_of_target will return 'continuous-multioutput' for continuous multioutput input, but if the data is all integers, it will be identified as 'multiclass-multioutput'.

multiclass A classification problem consisting of more than two classes. A multiclass target may be represented as a 1-dimensional array of strings or integers. A 2d column vector of integers (i.e. a single output in *multioutput* terms) is also accepted.

We do not officially support other orderable, hashable objects as class labels, even if estimators may happen to work when given classification targets of such type.

For semi-supervised classification, *unlabeled* samples should have the special label -1 in `y`.

Within sckit-learn, all estimators supporting binary classification also support multiclass classification, using One-vs-Rest by default.

A *preprocessing.LabelEncoder* helps to canonicalize multiclass targets as integers.

type_of_target will return 'multiclass' for multiclass input. The user may also want to handle 'binary' input identically to 'multiclass'.

multiclass multioutput

multioutput multiclass A classification problem where each sample's target consists of `n_outputs` *outputs*, each a class label, for a fixed int `n_outputs > 1` in a particular dataset. Each output has a fixed set of available classes, and each sample is labelled with a class for each output. An output may be binary or multiclass, and in the case where all outputs are binary, the target is *multilabel*.

Multiclass multioutput targets are represented as multiple *multiclass* targets, horizontally stacked into an array of shape `(n_samples, n_outputs)`.

XXX: For simplicity, we may not always support string class labels for multiclass multioutput, and integer class labels should be used.

`multioutput` provides estimators which estimate multi-output problems using multiple single-output estimators. This may not fully account for dependencies among the different outputs, which methods natively handling the multioutput case (e.g. decision trees, nearest neighbors, neural networks) may do better.

type_of_target will return 'multiclass-multioutput' for multiclass multioutput input.

multilabel A *multiclass multioutput* target where each output is *binary*. This may be represented as a 2d (dense) array or sparse matrix of integers, such that each column is a separate binary target, where positive labels are indicated with 1 and negative labels are usually -1 or 0. Sparse multilabel targets are not supported everywhere that dense multilabel targets are supported.

Semantically, a multilabel target can be thought of as a set of labels for each sample. While not used internally, *preprocessing.MultiLabelBinarizer* is provided as a utility to convert from a list of sets representation to a 2d array or sparse matrix. One-hot encoding a multiclass target with *preprocessing.LabelBinarizer* turns it into a multilabel problem.

type_of_target will return 'multilabel-indicator' for multilabel input, whether sparse or dense.

multioutput

multi-output A target where each sample has multiple classification/regression labels. See *multiclass multioutput* and *continuous multioutput*. We do not currently support modelling mixed classification and regression targets.

5.4 Methods

`decision_function` In a fitted *classifier* or *outlier detector*, predicts a "soft" score for each sample in relation to each class, rather than the "hard" categorical prediction produced by *predict*. Its input is usually only some observed data, *X*.

If the estimator was not already *fitted*, calling this method should raise a `exceptions.NotFittedError`.

Output conventions:

binary classification A 1-dimensional array, where values strictly greater than zero indicate the positive class (i.e. the last class in *classes_*).

multiclass classification A 2-dimensional array, where the row-wise arg-maximum is the predicted class. Columns are ordered according to *classes_*.

multilabel classification Scikit-learn is inconsistent in its representation of multilabel decision functions. Some estimators represent it like multiclass multioutput, i.e. a list of 2d arrays, each with two columns. Others represent it with a single 2d array, whose columns correspond to the individual binary classification decisions. The latter representation is ambiguously identical to the multiclass classification format, though its semantics differ: it should be interpreted, like in the binary case, by thresholding at 0.

TODO: This gist highlights the use of the different formats for multilabel.

multioutput classification A list of 2d arrays, corresponding to each multiclass decision function.

outlier detection A 1-dimensional array, where a value greater than or equal to zero indicates an inlier.

`fit` The `fit` method is provided on every estimator. It usually takes some *samples* `X`, *targets* `y` if the model is supervised, and potentially other *sample properties* such as *sample_weight*. It should:

- clear any prior *attributes* stored on the estimator, unless *warm_start* is used;
- validate and interpret any *parameters*, ideally raising an error if invalid;
- validate the input data;
- estimate and store model attributes from the estimated parameters and provided data; and
- return the now *fitted* estimator to facilitate method chaining.

Target Types describes possible formats for `y`.

`fit_predict` Used especially for *unsupervised*, *transductive* estimators, this fits the model and returns the predictions (similar to *predict*) on the training data. In clusterers, these predictions are also stored in the *labels_* attribute, and the output of `.fit_predict(X)` is usually equivalent to `.fit(X).predict(X)`. The parameters to `fit_predict` are the same as those to `fit`.

`fit_transform` A method on *transformers* which fits the estimator and returns the transformed training data. It takes parameters as in *fit* and its output should have the same shape as calling `.fit(X, ...).transform(X)`. There are nonetheless rare cases where `.fit_transform(X, ...)` and `.fit(X, ...).transform(X)` do not return the same value, wherein training data needs to be handled differently (due to model blending in stacked ensembles, for instance; such cases should be clearly documented). *Transductive* transformers may also provide `fit_transform` but not *transform*.

One reason to implement `fit_transform` is that performing `fit` and `transform` separately would be less efficient than together. *base.TransformerMixin* provides a default implementation, providing a consistent interface across transformers where `fit_transform` is or is not specialised.

In *inductive* learning – where the goal is to learn a generalised model that can be applied to new data – users should be careful not to apply `fit_transform` to the entirety of a dataset (i.e. training and test data together) before further modelling, as this results in *data leakage*.

get_feature_names Primarily for *feature extractors*, but also used for other transformers to provide string names for each column in the output of the estimator's *transform* method. It outputs a list of strings, and may take a list of strings as input, corresponding to the names of input columns from which output column names can be generated. By default input features are named x0, x1, ...

get_n_splits On a *CV splitter* (not an estimator), returns the number of elements one would get if iterating through the return value of *split* given the same parameters. Takes the same parameters as split.

get_params Gets all *parameters*, and their values, that can be set using *set_params*. A parameter `deep` can be used, when set to False to only return those parameters not including __, i.e. not due to indirection via contained estimators.

Most estimators adopt the definition from *base.BaseEstimator*, which simply adopts the parameters defined for `__init__`. *pipeline.Pipeline*, among others, reimplements `get_params` to declare the estimators named in its `steps` parameters as themselves being parameters.

partial_fit Facilitates fitting an estimator in an online fashion. Unlike `fit`, repeatedly calling `partial_fit` does not clear the model, but updates it with respect to the data provided. The portion of data provided to `partial_fit` may be called a mini-batch. Each mini-batch must be of consistent shape, etc. In iterative estimators, `partial_fit` often only performs a single iteration.

`partial_fit` may also be used for *out-of-core* learning, although usually limited to the case where learning can be performed online, i.e. the model is usable after each `partial_fit` and there is no separate processing needed to finalize the model. *cluster.Birch* introduces the convention that calling `partial_fit(X)` will produce a model that is not finalized, but the model can be finalized by calling `partial_fit()` i.e. without passing a further mini-batch.

Generally, estimator parameters should not be modified between calls to `partial_fit`, although `partial_fit` should validate them as well as the new mini-batch of data. In contrast, `warm_start` is used to repeatedly fit the same estimator with the same data but varying parameters.

Like `fit`, `partial_fit` should return the estimator object.

To clear the model, a new estimator should be constructed, for instance with *base.clone*.

NOTE: Using `partial_fit` after `fit` results in undefined behavior.

predict Makes a prediction for each sample, usually only taking *X* as input (but see under regressor output conventions below). In a *classifier* or *regressor*, this prediction is in the same target space used in fitting (e.g. one of {'red', 'amber', 'green'} if the `y` in fitting consisted of these strings). Despite this, even when `y` passed to *fit* is a list or other array-like, the output of `predict` should always be an array or sparse matrix. In a *clusterer* or *outlier detector* the prediction is an integer.

If the estimator was not already *fitted*, calling this method should raise a *exceptions.NotFittedError*.

Output conventions:

classifier An array of shape `(n_samples,)` `(n_samples, n_outputs)`. *Multilabel* data may be represented as a sparse matrix if a sparse matrix was used in fitting. Each element should be one of the values in the classifier's *classes_* attribute.

clusterer An array of shape `(n_samples,)` where each value is from 0 to `n_clusters - 1` if the corresponding sample is clustered, and -1 if the sample is not clustered, as in *cluster.dbscan*.

outlier detector An array of shape `(n_samples,)` where each value is -1 for an outlier and 1 otherwise.

regressor A numeric array of shape `(n_samples,)`, usually float64. Some regressors have extra options in their `predict` method, allowing them to return standard deviation (`return_std=True`) or covariance (`return_cov=True`) relative to the predicted value. In this case, the return value is a tuple of arrays corresponding to (prediction mean, std, cov) as required.

predict_log_proba The natural logarithm of the output of *predict_proba*, provided to facilitate numerical stability.

predict_proba A method in *classifiers* and *clusterers* that are able to return probability estimates for each class/cluster. Its input is usually only some observed data, *X*.

If the estimator was not already *fitted*, calling this method should raise a `exceptions.NotFittedError`.

Output conventions are like those for *decision_function* except in the *binary* classification case, where one column is output for each class (while `decision_function` outputs a 1d array). For binary and multiclass predictions, each row should add to 1.

Like other methods, `predict_proba` should only be present when the estimator can make probabilistic predictions (see *duck typing*). This means that the presence of the method may depend on estimator parameters (e.g. in `linear_model.SGDClassifier`) or training data (e.g. in `model_selection.GridSearchCV`) and may only appear after fitting.

score A method on an estimator, usually a *predictor*, which evaluates its predictions on a given dataset, and returns a single numerical score. A greater return value should indicate better predictions; accuracy is used for classifiers and R^2 for regressors by default.

If the estimator was not already *fitted*, calling this method should raise a `exceptions.NotFittedError`.

Some estimators implement a custom, estimator-specific score function, often the likelihood of the data under the model.

score_samples TODO

If the estimator was not already *fitted*, calling this method should raise a `exceptions.NotFittedError`.

set_params Available in any estimator, takes keyword arguments corresponding to keys in *get_params*. Each is provided a new value to assign such that calling `get_params` after `set_params` will reflect the changed *parameters*. Most estimators use the implementation in `base.BaseEstimator`, which handles nested parameters and otherwise sets the parameter as an attribute on the estimator. The method is overridden in `pipeline.Pipeline` and related estimators.

split On a *CV splitter* (not an estimator), this method accepts parameters (*X*, *y*, *groups*), where all may be optional, and returns an iterator over `(train_idx, test_idx)` pairs. Each of {train,test}_idx is a 1d integer array, with values from 0 from `X.shape[0] - 1` of any length, such that no values appear in both some `train_idx` and its corresponding `test_idx`.

transform In a *transformer*, transforms the input, usually only *X*, into some transformed space (conventionally notated as *Xt*). Output is an array or sparse matrix of length *n_samples* and with number of columns fixed after *fitting*.

If the estimator was not already *fitted*, calling this method should raise a `exceptions.NotFittedError`.

5.5 Parameters

These common parameter names, specifically used in estimator construction (see concept *parameter*), sometimes also appear as parameters of functions or non-estimator constructors.

class_weight Used to specify sample weights when fitting classifiers as a function of the *target* class. Where *sample_weight* is also supported and given, it is multiplied by the `class_weight` contribution. Similarly, where `class_weight` is used in a *multioutput* (including *multilabel*) tasks, the weights are multiplied across outputs (i.e. columns of `y`).

By default all samples have equal weight such that classes are effectively weighted by their their prevalence in the training data. This could be achieved explicitly with `class_weight={label1: 1, label2: 1, ...}` for all class labels.

More generally, `class_weight` is specified as a dict mapping class labels to weights (`{class_label: weight}`), such that each sample of the named class is given that weight.

`class_weight='balanced'` can be used to give all classes equal weight by giving each sample a weight inversely related to its class's prevalence in the training data: `n_samples / (n_classes * np.bincount(y))`. Class weights will be used differently depending on the algorithm: for linear models (such as linear SVM or logistic regression), the class weights will alter the loss function by weighting the loss of each sample by its class weight. For tree-based algorithms, the class weights will be used for reweighting the splitting criterion. **Note** however that this rebalancing does not take the weight of samples in each class into account.

For multioutput classification, a list of dicts is used to specify weights for each output. For example, for four-class multilabel classification weights should be `[{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}]` instead of `[{1:1}, {2:5}, {3:1}, {4:1}]`.

The `class_weight` parameter is validated and interpreted with `utils.compute_class_weight`.

cv Determines a cross validation splitting strategy, as used in cross-validation based routines. `cv` is also available in estimators such as *`multioutput.ClassifierChain`* or *`calibration.CalibratedClassifierCV`* which use the predictions of one estimator as training data for another, to not overfit the training supervision.

Possible inputs for `cv` are usually:

- An integer, specifying the number of folds in K-fold cross validation. K-fold will be stratified over classes if the estimator is a classifier (determined by *`base.is_classifier`*) and the *targets* may represent a binary or multiclass (but not multioutput) classification problem (determined by *`utils.multiclass.type_of_target`*).
- A *cross-validation splitter* instance. Refer to the *User Guide* for splitters available within Scikit-learn.
- An iterable yielding train/test splits.

With some exceptions (especially where not using cross validation at all is an option), the default is 5-fold.

`cv` values are validated and interpreted with `utils.check_cv`.

kernel TODO

max_iter For estimators involving iterative optimization, this determines the maximum number of iterations to be performed in *fit*. If `max_iter` iterations are run without convergence, a *`exceptions.ConvergenceWarning`* should be raised. Note that the interpretation of "a single iteration" is inconsistent across estimators: some, but not all, use it to mean a single epoch (i.e. a pass over every sample in the data).

FIXME perhaps we should have some common tests about the relationship between ConvergenceWarning and max_iter.

memory Some estimators make use of `joblib.Memory` to store partial solutions during fitting. Thus when `fit` is called again, those partial solutions have been memoized and can be reused.

A `memory` parameter can be specified as a string with a path to a directory, or a `joblib.Memory` instance (or an object with a similar interface, i.e. a `cache` method) can be used.

`memory` values are validated and interpreted with *`utils.validation.check_memory`*.

metric As a parameter, this is the scheme for determining the distance between two data points. See *metrics.pairwise_distances*. In practice, for some algorithms, an improper distance metric (one that does not obey the triangle inequality, such as Cosine Distance) may be used.

XXX: hierarchical clustering uses `affinity` with this meaning.

We also use *metric* to refer to *evaluation metrics*, but avoid using this sense as a parameter name.

n_components The number of features which a *transformer* should transform the input into. See *components_* for the special case of affine projection.

n_iter_no_change Number of iterations with no improvement to wait before stopping the iterative procedure. This is also known as a *patience* parameter. It is typically used with *early stopping* to avoid stopping too early.

n_jobs This parameter is used to specify how many concurrent processes or threads should be used for routines that are parallelized with *joblib*.

`n_jobs` is an integer, specifying the maximum number of concurrently running workers. If 1 is given, no joblib parallelism is used at all, which is useful for debugging. If set to -1, all CPUs are used. For `n_jobs` below -1, (n_cpus + 1 + n_jobs) are used. For example with `n_jobs=-2`, all CPUs but one are used.

`n_jobs` is `None` by default, which means *unset*; it will generally be interpreted as `n_jobs=1`, unless the current `joblib.Parallel` backend context specifies otherwise.

For more details on the use of `joblib` and its interactions with scikit-learn, please refer to our *parallelism notes*.

pos_label Value with which positive labels must be encoded in binary classification problems in which the positive class is not assumed. This value is typically required to compute asymmetric evaluation metrics such as precision and recall.

random_state Whenever randomization is part of a Scikit-learn algorithm, a `random_state` parameter may be provided to control the random number generator used. Note that the mere presence of `random_state` doesn't mean that randomization is always used, as it may be dependent on another parameter, e.g. `shuffle`, being set.

`random_state`'s value may be:

None (default) Use the global random state from `numpy.random`.

An integer Use a new random number generator seeded by the given integer. To make a randomized algorithm deterministic (i.e. running it multiple times will produce the same result), an arbitrary integer `random_state` can be used. However, it may be worthwhile checking that your results are stable across a number of different distinct random seeds. Popular integer random seeds are 0 and 42.

A `numpy.random.RandomState` instance Use the provided random state, only affecting other users of the same random state instance. Calling fit multiple times will reuse the same instance, and will produce different results.

utils.check_random_state is used internally to validate the input `random_state` and return a `RandomState` instance.

scoring Specifies the score function to be maximized (usually by *cross validation*), or – in some cases – multiple score functions to be reported. The score function can be a string accepted by *metrics.get_scorer* or a callable *scorer*, not to be confused with an *evaluation metric*, as the latter have a more diverse API. `scoring` may also be set to None, in which case the estimator's *score* method is used. See *The scoring parameter: defining model evaluation rules* in the User Guide.

Where multiple metrics can be evaluated, `scoring` may be given either as a list of unique strings or a dict with names as keys and callables as values. Note that this does *not* specify which score function is to be maximised, and another parameter such as `refit` may be used for this purpose.

The `scoring` parameter is validated and interpreted using *metrics.check_scoring*.

verbose Logging is not handled very consistently in Scikit-learn at present, but when it is provided as an option, the `verbose` parameter is usually available to choose no logging (set to False). Any True value should enable some logging, but larger integers (e.g. above 10) may be needed for full verbosity. Verbose logs are usually printed to Standard Output. Estimators should not produce any output on Standard Output with the default `verbose` setting.

warm_start When fitting an estimator repeatedly on the same dataset, but for multiple parameter values (such as to find the value maximizing performance as in *grid search*), it may be possible to reuse aspects of the model learnt from the previous parameter value, saving time. When `warm_start` is true, the existing *fitted* model *attributes* are used to initialise the new model in a subsequent call to *fit*.

Note that this is only applicable for some models and some parameters, and even some orders of parameter values. For example, `warm_start` may be used when building random forests to add more trees to the forest (increasing `n_estimators`) but not to reduce their number.

partial_fit also retains the model between calls, but differs: with `warm_start` the parameters change and the data is (more-or-less) constant across calls to `fit`; with `partial_fit`, the mini-batch of data changes and model parameters stay fixed.

There are cases where you want to use `warm_start` to fit on different, but closely related data. For example, one may initially fit to a subset of the data, then fine-tune the parameter search on the full dataset. For classification, all data in a sequence of `warm_start` calls to `fit` must include samples from each class.

5.6 Attributes

See concept *attribute*.

classes_ A list of class labels known to the *classifier*, mapping each label to a numerical index used in the model representation our output. For instance, the array output from *predict_proba* has columns aligned with `classes_`. For *multi-output* classifiers, `classes_` should be a list of lists, with one class listing for each output. For each output, the classes should be sorted (numerically, or lexicographically for strings).

`classes_` and the mapping to indices is often managed with *`preprocessing.LabelEncoder`*.

components_ An affine transformation matrix of shape `(n_components, n_features)` used in many linear *transformers* where *n_components* is the number of output features and *n_features* is the number of input features.

See also *components_* which is a similar attribute for linear predictors.

coef_ The weight/coefficient matrix of a generalised linear model *predictor*, of shape `(n_features,)` for binary classification and single-output regression, `(n_classes, n_features)` for multiclass classification and `(n_targets, n_features)` for multi-output regression. Note this does not include the intercept (or bias) term, which is stored in `intercept_`.

When available, `feature_importances_` is not usually provided as well, but can be calculated as the norm of each feature's entry in `coef_`.

See also *components_* which is a similar attribute for linear transformers.

embedding_ An embedding of the training data in *manifold learning* estimators, with shape `(n_samples, n_components)`, identical to the output of *fit_transform*. See also *labels_*.

n_iter_ The number of iterations actually performed when fitting an iterative estimator that may stop upon convergence. See also *max_iter*.

feature_importances_ A vector of shape `(n_features,)` available in some *predictors* to provide a relative measure of the importance of each feature in the predictions of the model.

labels_ A vector containing a cluster label for each sample of the training data in *clusterers*, identical to the output of *fit_predict*. See also *embedding_*.

5.7 Data and sample properties

See concept *sample property*.

groups Used in cross validation routines to identify samples which are correlated. Each value is an identifier such that, in a supporting *CV splitter*, samples from some `groups` value may not appear in both a training set and its corresponding test set. See *Cross-validation iterators for grouped data.*.

sample_weight A relative weight for each sample. Intuitively, if all weights are integers, a weighted model or score should be equivalent to that calculated when repeating the sample the number of times specified in the weight. Weights may be specified as floats, so that sample weights are usually equivalent up to a constant positive scaling factor.

FIXME Is this interpretation always the case in practice? We have no common tests.

Some estimators, such as decision trees, support negative weights. FIXME: This feature or its absence may not be tested or documented in many estimators.

This is not entirely the case where other parameters of the model consider the number of samples in a region, as with `min_samples` in *`cluster.DBSCAN`*. In this case, a count of samples becomes to a sum of their weights.

In classification, sample weights can also be specified as a function of class with the *class_weight* estimator *parameter*.

X Denotes data that is observed at training and prediction time, used as independent variables in learning. The notation is uppercase to denote that it is ordinarily a matrix (see *rectangular*). When a matrix, each sample may be represented by a *feature* vector, or a vector of *precomputed* (dis)similarity with each training sample. `X` may also not be a matrix, and may require a *feature extractor* or a *pairwise metric* to turn it into one before learning a model.

Xt Shorthand for "transformed *X*".

y

Y Denotes data that may be observed at training time as the dependent variable in learning, but which is unavailable at prediction time, and is usually the *target* of prediction. The notation may be uppercase to denote that it is a matrix, representing *multi-output* targets, for instance; but usually we use `y` and sometimes do so even when multiple outputs are assumed.

CHAPTER

SIX

EXAMPLES

6.1 Miscellaneous examples

Miscellaneous and introductory examples for scikit-learn.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.1.1 Compact estimator representations

This example illustrates the use of the print_changed_only global parameter.

Setting print_changed_only to True will alternate the representation of estimators to only show the parameters that have been set to non-default values. This can be used to have more compact representations.

Out:

```
Default representation:
LogisticRegression(penalty='l1')

With changed_only option:
LogisticRegression(penalty='l1')
```

```
print(__doc__)

from sklearn.linear_model import LogisticRegression
from sklearn import set_config


lr = LogisticRegression(penalty='l1')
print('Default representation:')
print(lr)
# LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
#                    intercept_scaling=1, l1_ratio=None, max_iter=100,
#                    multi_class='auto', n_jobs=None, penalty='l1',
#                    random_state=None, solver='warn', tol=0.0001, verbose=0,
#                    warm_start=False)
```

(continues on next page)

(continued from previous page)

```
set_config(print_changed_only=True)
print('\nWith changed_only option:')
print(lr)
# LogisticRegression(penalty='l1')
```

Total running time of the script: (0 minutes 0.097 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.1.2 ROC Curve with Visualization API

Scikit-learn defines a simple API for creating visualizations for machine learning. The key features of this API is to allow for quick plotting and visual adjustments without recalculation. In this example, we will demonstrate how to use the visualization API by comparing ROC curves.

```
print(__doc__)
```

Load Data and Train a SVC

First, we load the wine dataset and convert it to a binary classification problem. Then, we train a support vector classifier on a training dataset.

```
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import plot_roc_curve
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split

X, y = load_wine(return_X_y=True)
y = y == 2

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
svc = SVC(random_state=42)
svc.fit(X_train, y_train)
```

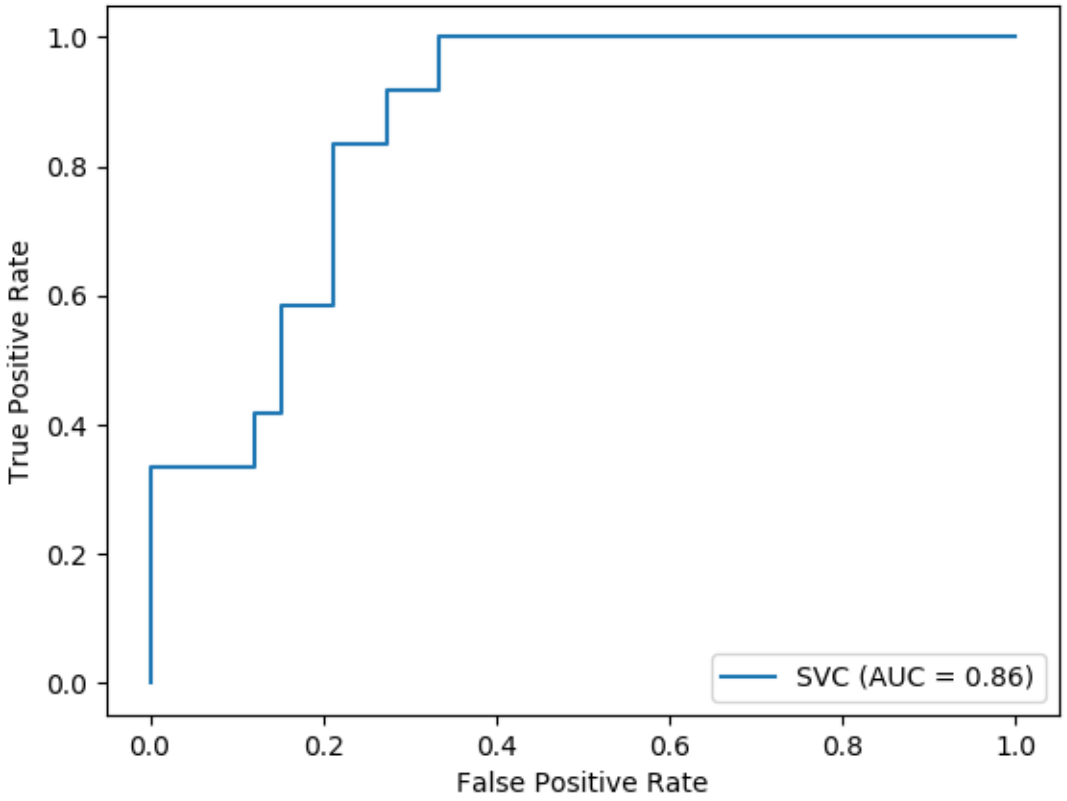
Out:

```
SVC(random_state=42)
```

Plotting the ROC Curve

Next, we plot the ROC curve with a single call to *sklearn.metrics.plot_roc_curve*. The returned `svc_disp` object allows us to continue using the already computed ROC curve for the SVC in future plots.

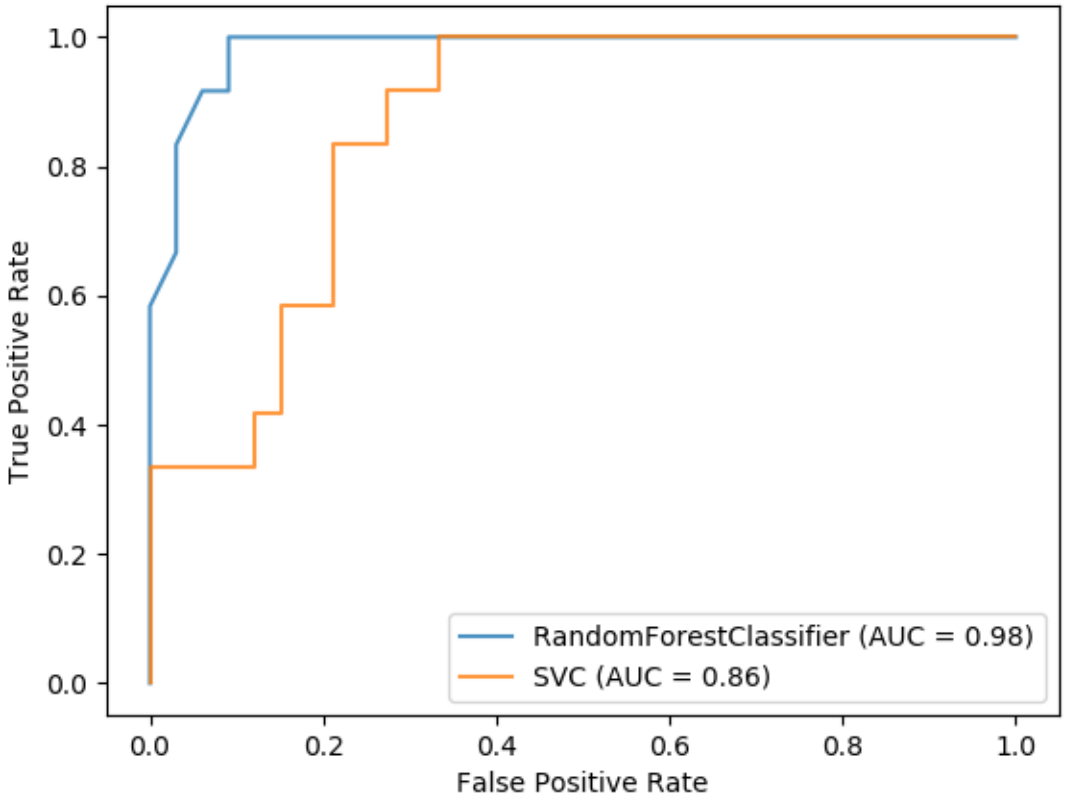
```
svc_disp = plot_roc_curve(svc, X_test, y_test)
plt.show()
```



Training a Random Forest and Plotting the ROC Curve

We train a random forest classifier and create a plot comparing it to the SVC ROC curve. Notice how `svc_disp` uses *`plot`* to plot the SVC ROC curve without recomputing the values of the roc curve itself. Furthermore, we pass `alpha=0.8` to the plot functions to adjust the alpha values of the curves.

```
rfc = RandomForestClassifier(n_estimators=10, random_state=42)
rfc.fit(X_train, y_train)
ax = plt.gca()
rfc_disp = plot_roc_curve(rfc, X_test, y_test, ax=ax, alpha=0.8)
svc_disp.plot(ax=ax, alpha=0.8)
plt.show()
```



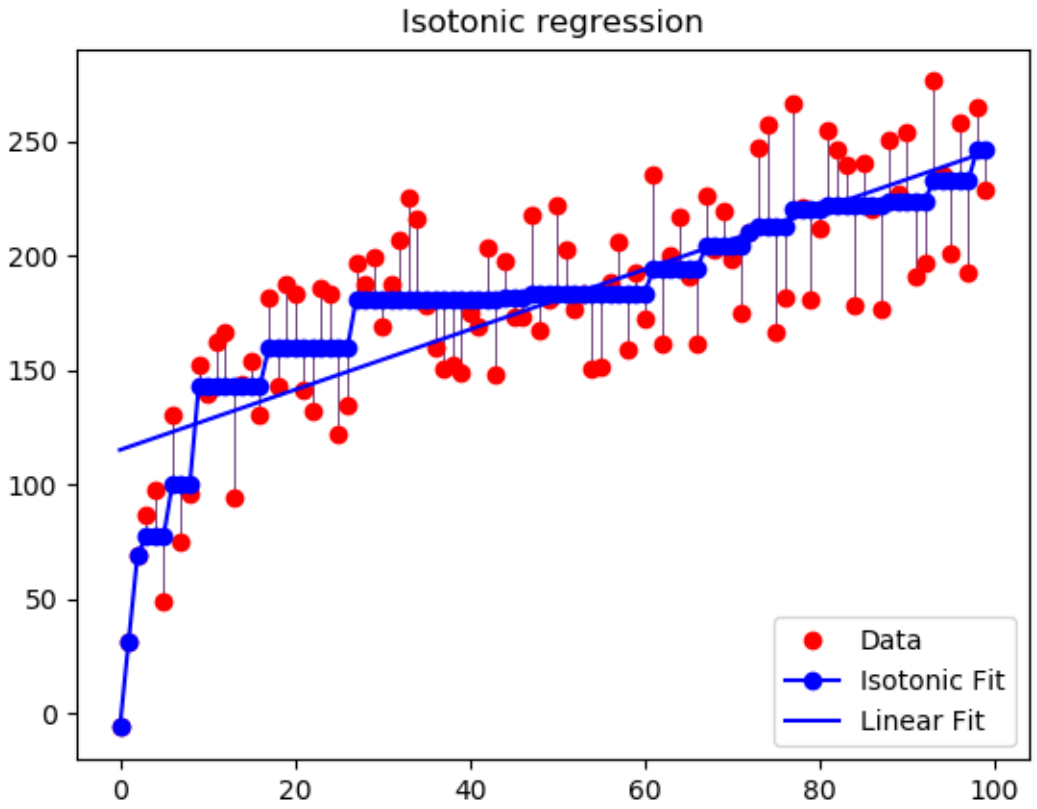
Total running time of the script: (0 minutes 0.567 seconds)

Estimated memory usage: 12 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.1.3 Isotonic Regression

An illustration of the isotonic regression on generated data. The isotonic regression finds a non-decreasing approximation of a function while minimizing the mean squared error on the training data. The benefit of such a model is that it does not assume any form for the target function such as linearity. For comparison a linear regression is also presented.




```
print(__doc__)

# Author: Nelle Varoquaux <nelle.varoquaux@gmail.com>
#         Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection

from sklearn.linear_model import LinearRegression
from sklearn.isotonic import IsotonicRegression
from sklearn.utils import check_random_state

n = 100
x = np.arange(n)
rs = check_random_state(0)
y = rs.randint(-50, 50, size=(n,)) + 50. * np.log1p(np.arange(n))

# ##############################################################################
# Fit IsotonicRegression and LinearRegression models

ir = IsotonicRegression()

y_ = ir.fit_transform(x, y)
```

(continues on next page)

(continued from previous page)

```
lr = LinearRegression()
lr.fit(x[:, np.newaxis], y)  # x needs to be 2d for LinearRegression

# #############################################################################
# Plot result

segments = [[[i, y[i]], [i, y_[i]]] for i in range(n)]
lc = LineCollection(segments, zorder=0)
lc.set_array(np.ones(len(y)))
lc.set_linewidths(np.full(n, 0.5))

fig = plt.figure()
plt.plot(x, y, 'r.', markersize=12)
plt.plot(x, y_, 'b.-', markersize=12)
plt.plot(x, lr.predict(x[:, np.newaxis]), 'b-')
plt.gca().add_collection(lc)
plt.legend(('Data', 'Isotonic Fit', 'Linear Fit'), loc='lower right')
plt.title('Isotonic regression')
plt.show()
```

Total running time of the script: (0 minutes 0.192 seconds)

Estimated memory usage: 11 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.1.4 Advanced Plotting With Partial Dependence

The *plot_partial_dependence* function returns a *PartialDependenceDisplay* object that can be used for plotting without needing to recalculate the partial dependence. In this example, we show how to plot partial dependence plots and how to quickly customize the plot with the visualization API.

Note: See also *ROC Curve with Visualization API*

```
print(__doc__)

import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.tree import DecisionTreeRegressor
from sklearn.inspection import plot_partial_dependence
```

Train models on the boston housing price dataset

First, we train a decision tree and a multi-layer perceptron on the boston housing price dataset.

```
boston = load_boston()
X = pd.DataFrame(boston.data, columns=boston.feature_names)
y = boston.target

tree = DecisionTreeRegressor()
mlp = make_pipeline(StandardScaler(),
                    MLPRegressor(hidden_layer_sizes=(100, 100),
                                 tol=1e-2, max_iter=500, random_state=0))
tree.fit(X, y)
mlp.fit(X, y)
```

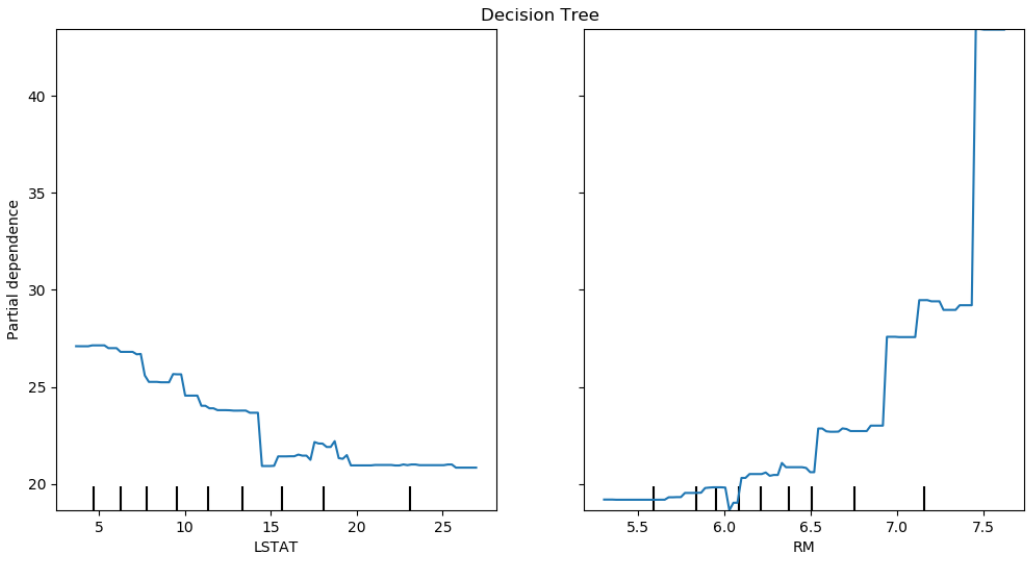
Out:

```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('mlpregressor',
                 MLPRegressor(hidden_layer_sizes=(100, 100), max_iter=500,
                              random_state=0, tol=0.01))])
```

Plotting partial dependence for two features

We plot partial dependence curves for features "LSTAT" and "RM" for the decision tree. With two features, *plot_partial_dependence* expects to plot two curves. Here the plot function place a grid of two plots using the space defined by `ax` .

```
fig, ax = plt.subplots(figsize=(12, 6))
ax.set_title("Decision Tree")
tree_disp = plot_partial_dependence(tree, X, ["LSTAT", "RM"], ax=ax)
```

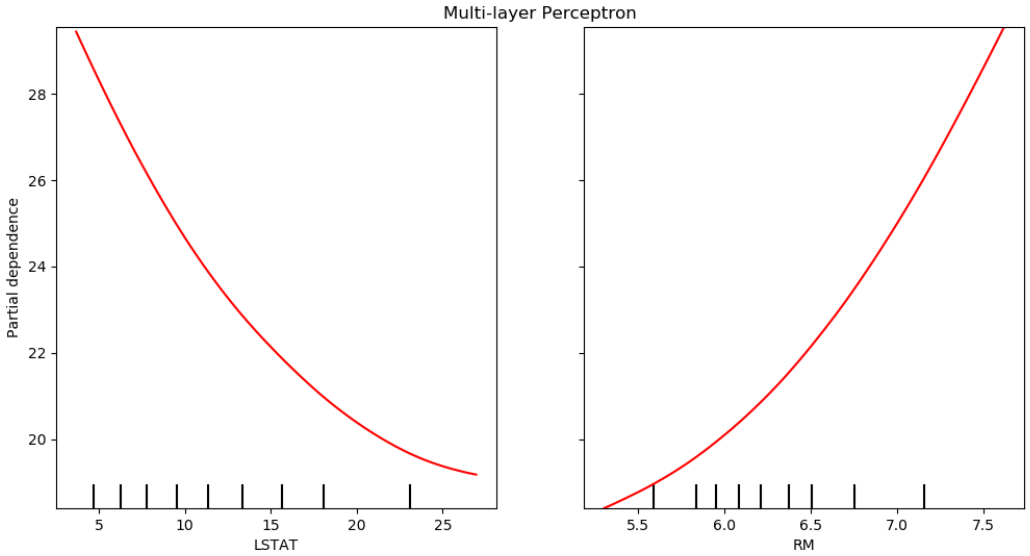


Out:

```
/home/circleci/project/sklearn/tree/_classes.py:1233: FutureWarning: the classes_␣
↪attribute is to be deprecated from version 0.22 and will be removed in 0.24.
  warnings.warn(msg, FutureWarning)
```

The partial depdendence curves can be plotted for the multi-layer perceptron. In this case, `line_kw` is passed to *`plot_partial_dependence`* to change the color of the curve.

```
fig, ax = plt.subplots(figsize=(12, 6))
ax.set_title("Multi-layer Perceptron")
mlp_disp = plot_partial_dependence(mlp, X, ["LSTAT", "RM"], ax=ax,
                                   line_kw={"c": "red"})
```

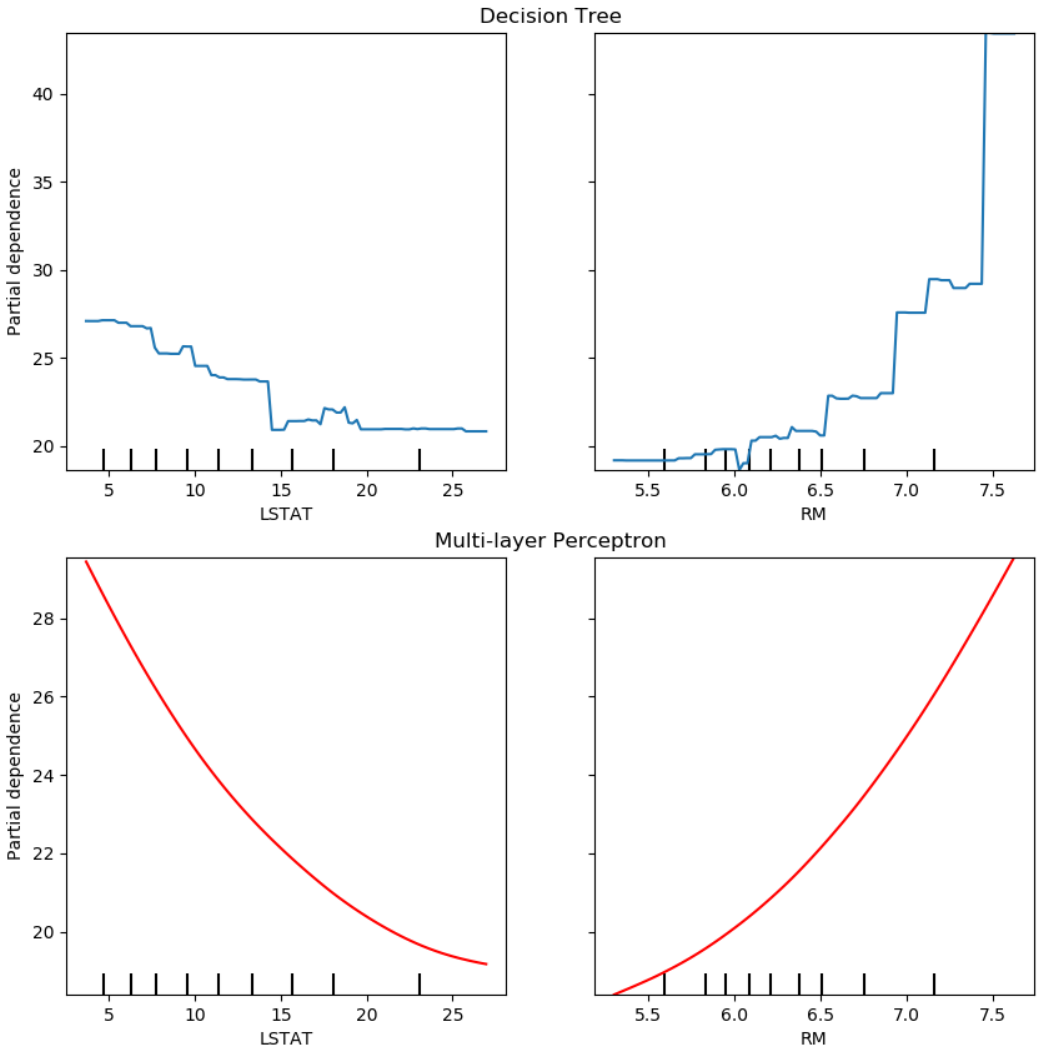


Plotting partial dependence of the two models together

The `tree_disp` and `mlp_disp` *`PartialDependenceDisplay`* objects contain all the computed information needed to recreate the partial dependence curves. This means we can easily create additional plots without needing to recompute the curves.

One way to plot the curves is to place them in the same figure, with the curves of each model on each row. First, we create a figure with two axes within two rows and one column. The two axes are passed to the *`plot`* functions of `tree_disp` and `mlp_disp`. The given axes will be used by the plotting function to draw the partial dependence. The resulting plot places the decision tree partial dependence curves in the first row of the multi-layer perceptron in the second row.

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 10))
tree_disp.plot(ax=ax1)
ax1.set_title("Decision Tree")
mlp_disp.plot(ax=ax2, line_kw={"c": "red"})
ax2.set_title("Multi-layer Perceptron")
```



Out:

```
Text(0.5, 1.0, 'Multi-layer Perceptron')
```

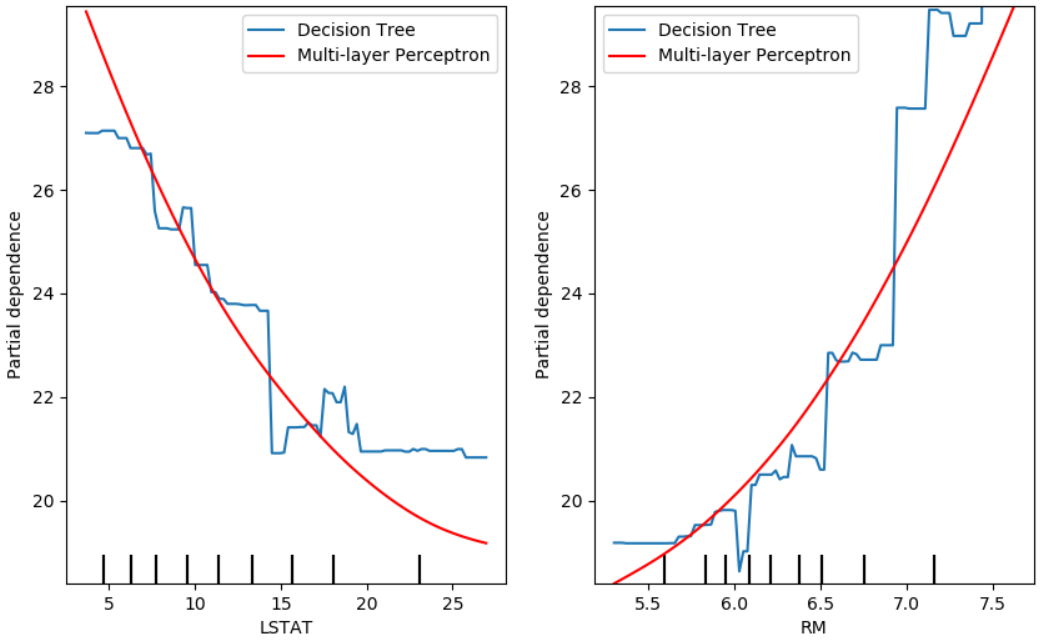
Another way to compare the curves is to plot them on top of each other. Here, we create a figure with one row and two columns. The axes are passed into the *plot* function as a list, which will plot the partial dependence curves of each model on the same axes. The length of the axes list must be equal to the number of plots drawn.

```
# Sets this image as the thumbnail for sphinx gallery
# sphinx_gallery_thumbnail_number = 4
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 6))
tree_disp.plot(ax=[ax1, ax2], line_kw={"label": "Decision Tree"})
mlp_disp.plot(ax=[ax1, ax2], line_kw={"label": "Multi-layer Perceptron",
                                      "c": "red"})
```

(continues on next page)

(continued from previous page)

```
ax1.legend()
ax2.legend()
```

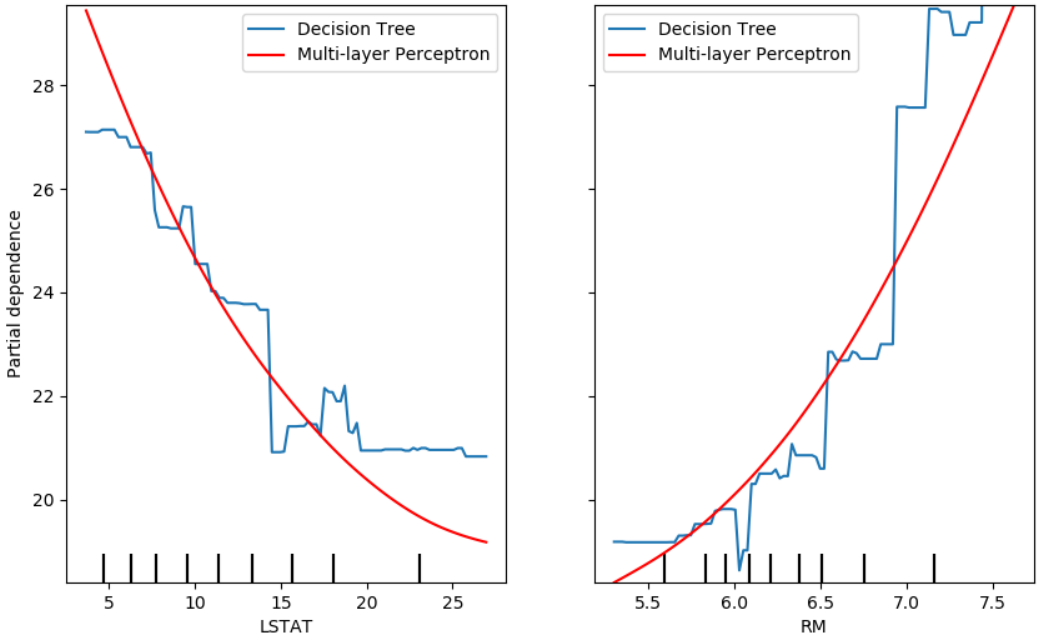


Out:

```
<matplotlib.legend.Legend object at 0x7ffa2778ec70>
```

`tree_disp.axes_` is a numpy array container the axes used to draw the partial dependence plots. This can be passed to `mlp_disp` to have the same affect of drawing the plots on top of each other. Furthermore, the `mlp_disp.figure_` stores the figure, which allows for resizing the figure after calling `plot`. In this case `tree_disp.axes_` has two dimensions, thus `plot` will only show the y label and y ticks on the left most plot.

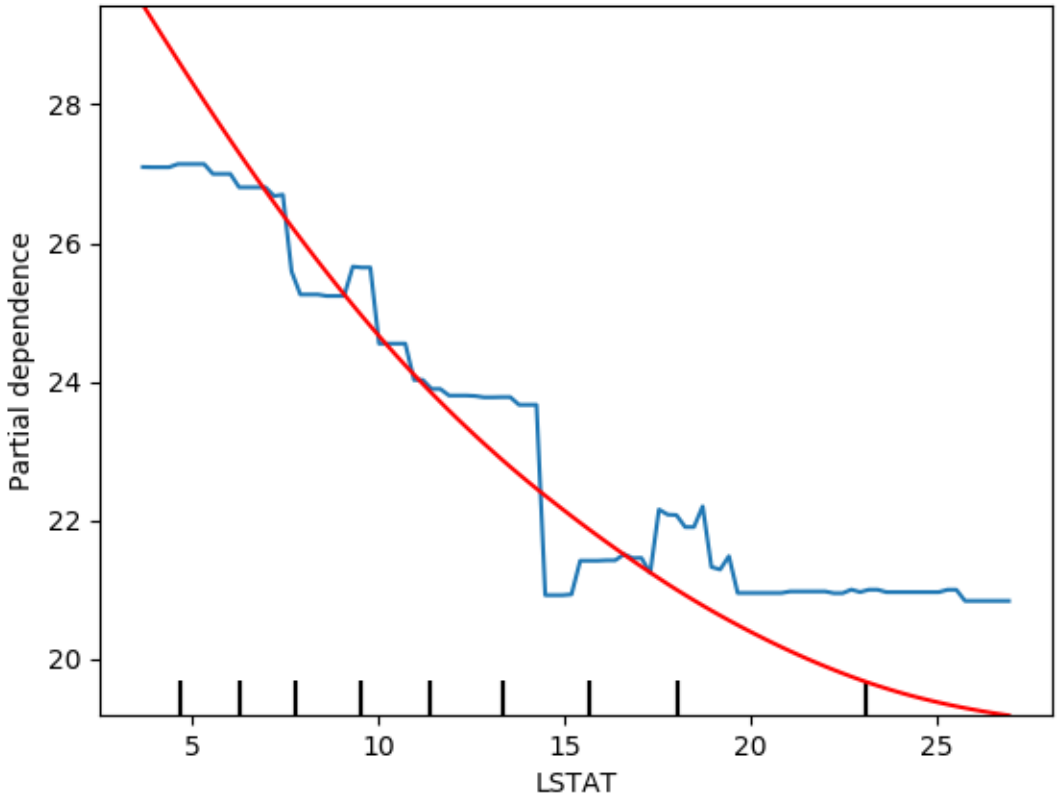
```
tree_disp.plot(line_kw={"label": "Decision Tree"})
mlp_disp.plot(line_kw={"label": "Multi-layer Perceptron", "c": "red"},
              ax=tree_disp.axes_)
tree_disp.figure_.set_size_inches(10, 6)
tree_disp.axes_[0, 0].legend()
tree_disp.axes_[0, 1].legend()
plt.show()
```



Plotting partial dependence for one feature

Here, we plot the partial dependence curves for a single feature, "LSTAT", on the same axes. In this case, `tree_disp.axes_` is passed into the second plot function.

```
tree_disp = plot_partial_dependence(tree, X, ["LSTAT"])
mlp_disp = plot_partial_dependence(mlp, X, ["LSTAT"],
                                   ax=tree_disp.axes_, line_kw={"c": "red"})
```



Out:

```
/home/circleci/project/sklearn/tree/_classes.py:1233: FutureWarning: the classes_
↪attribute is to be deprecated from version 0.22 and will be removed in 0.24.
  warnings.warn(msg, FutureWarning)
```

Total running time of the script: (0 minutes 3.668 seconds)

Estimated memory usage: 47 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.1.5 Face completion with a multi-output estimators

This example shows the use of multi-output estimator to complete images. The goal is to predict the lower half of a face given its upper half.

The first column of images shows true faces. The next columns illustrate how extremely randomized trees, k nearest neighbors, linear regression and ridge regression complete the lower half of those faces.


```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_olivetti_faces
from sklearn.utils.validation import check_random_state

from sklearn.ensemble import ExtraTreesRegressor
```

(continues on next page)

(continued from previous page)

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import RidgeCV

# Load the faces datasets
data, targets = fetch_olivetti_faces(return_X_y=True)

train = data[targets < 30]
test = data[targets >= 30]  # Test on independent people

# Test on a subset of people
n_faces = 5
rng = check_random_state(4)
face_ids = rng.randint(test.shape[0], size=(n_faces, ))
test = test[face_ids, :]

n_pixels = data.shape[1]
# Upper half of the faces
X_train = train[:, :(n_pixels + 1) // 2]
# Lower half of the faces
y_train = train[:, n_pixels // 2:]
X_test = test[:, :(n_pixels + 1) // 2]
y_test = test[:, n_pixels // 2:]

# Fit estimators
ESTIMATORS = {
    "Extra trees": ExtraTreesRegressor(n_estimators=10, max_features=32,
                                       random_state=0),
    "K-nn": KNeighborsRegressor(),
    "Linear regression": LinearRegression(),
    "Ridge": RidgeCV(),
}

y_test_predict = dict()
for name, estimator in ESTIMATORS.items():
    estimator.fit(X_train, y_train)
    y_test_predict[name] = estimator.predict(X_test)

# Plot the completed faces
image_shape = (64, 64)

n_cols = 1 + len(ESTIMATORS)
plt.figure(figsize=(2. * n_cols, 2.26 * n_faces))
plt.suptitle("Face completion with multi-output estimators", size=16)

for i in range(n_faces):
    true_face = np.hstack((X_test[i], y_test[i]))

    if i:
        sub = plt.subplot(n_faces, n_cols, i * n_cols + 1)
    else:
        sub = plt.subplot(n_faces, n_cols, i * n_cols + 1,
                          title="true faces")

    sub.axis("off")
    sub.imshow(true_face.reshape(image_shape),
               cmap=plt.cm.gray,
```

(continues on next page)

(continued from previous page)

```
                interpolation="nearest")

    for j, est in enumerate(sorted(ESTIMATORS)):
        completed_face = np.hstack((X_test[i], y_test_predict[est][i]))

        if i:
            sub = plt.subplot(n_faces, n_cols, i * n_cols + 2 + j)

        else:
            sub = plt.subplot(n_faces, n_cols, i * n_cols + 2 + j,
                              title=est)

        sub.axis("off")
        sub.imshow(completed_face.reshape(image_shape),
                   cmap=plt.cm.gray,
                   interpolation="nearest")

plt.show()
```

Total running time of the script: (0 minutes 2.393 seconds)

Estimated memory usage: 263 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.1.6 Multilabel classification

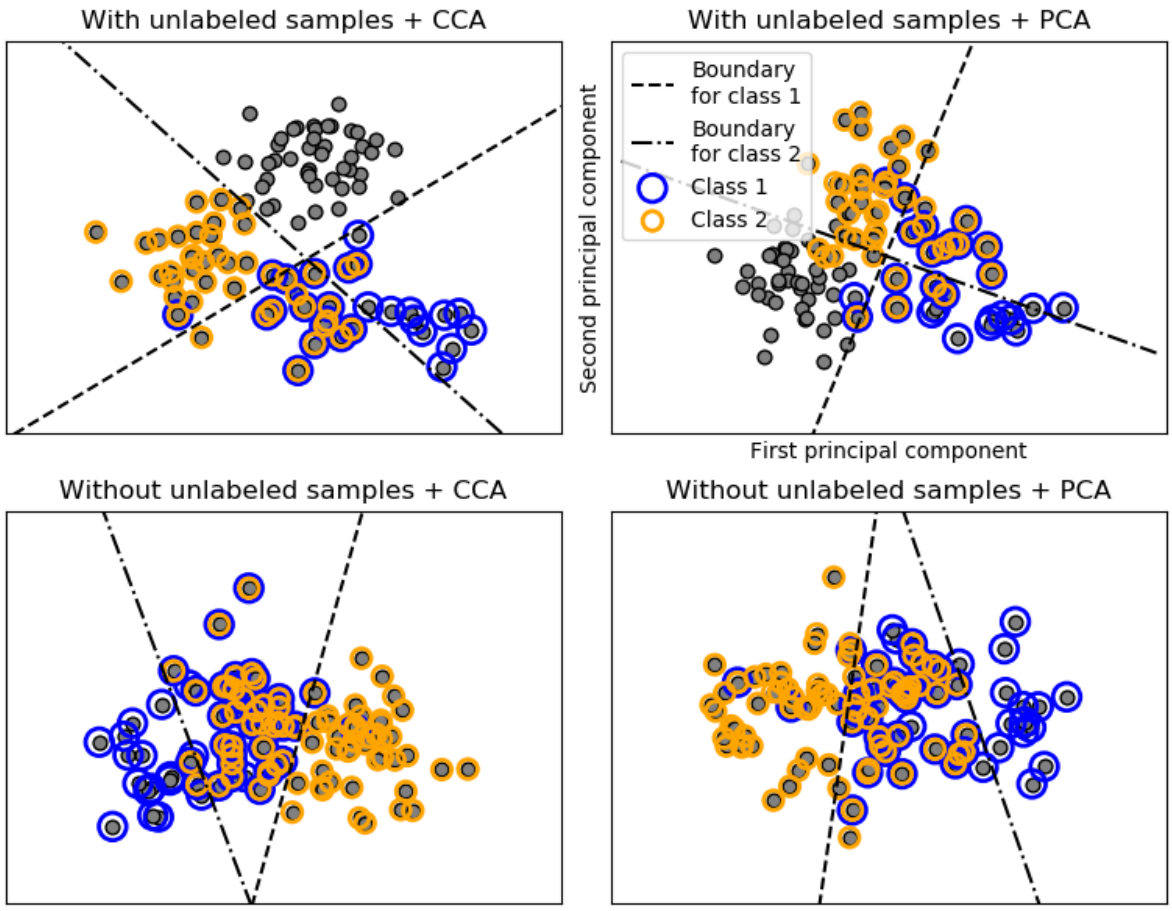
This example simulates a multi-label document classification problem. The dataset is generated randomly based on the following process:

- pick the number of labels: n ~ Poisson(n_labels)
- n times, choose a class c: c ~ Multinomial(theta)
- pick the document length: k ~ Poisson(length)
- k times, choose a word: w ~ Multinomial(theta_c)

In the above process, rejection sampling is used to make sure that n is more than 2, and that the document length is never zero. Likewise, we reject classes which have already been chosen. The documents that are assigned to both classes are plotted surrounded by two colored circles.

The classification is performed by projecting to the first two principal components found by PCA and CCA for visualisation purposes, followed by using the *sklearn.multiclass.OneVsRestClassifier* metaclassifier using two SVCs with linear kernels to learn a discriminative model for each class. Note that PCA is used to perform an unsupervised dimensionality reduction, while CCA is used to perform a supervised one.

Note: in the plot, "unlabeled samples" does not mean that we don't know the labels (as in semi-supervised learning) but that the samples simply do *not* have a label.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_multilabel_classification
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
from sklearn.decomposition import PCA
from sklearn.cross_decomposition import CCA


def plot_hyperplane(clf, min_x, max_x, linestyle, label):
    # get the separating hyperplane
    w = clf.coef_[0]
    a = -w[0] / w[1]
    xx = np.linspace(min_x - 5, max_x + 5)  # make sure the line is long enough
    yy = a * xx - (clf.intercept_[0]) / w[1]
    plt.plot(xx, yy, linestyle, label=label)


def plot_subfigure(X, Y, subplot, title, transform):
    if transform == "pca":
        X = PCA(n_components=2).fit_transform(X)
    elif transform == "cca":
```

(continues on next page)

(continued from previous page)

```
        X = CCA(n_components=2).fit(X, Y).transform(X)
    else:
        raise ValueError

    min_x = np.min(X[:, 0])
    max_x = np.max(X[:, 0])

    min_y = np.min(X[:, 1])
    max_y = np.max(X[:, 1])

    classif = OneVsRestClassifier(SVC(kernel='linear'))
    classif.fit(X, Y)

    plt.subplot(2, 2, subplot)
    plt.title(title)

    zero_class = np.where(Y[:, 0])
    one_class = np.where(Y[:, 1])
    plt.scatter(X[:, 0], X[:, 1], s=40, c='gray', edgecolors=(0, 0, 0))
    plt.scatter(X[zero_class, 0], X[zero_class, 1], s=160, edgecolors='b',
                facecolors='none', linewidths=2, label='Class 1')
    plt.scatter(X[one_class, 0], X[one_class, 1], s=80, edgecolors='orange',
                facecolors='none', linewidths=2, label='Class 2')

    plot_hyperplane(classif.estimators_[0], min_x, max_x, 'k--',
                    'Boundary\nfor class 1')
    plot_hyperplane(classif.estimators_[1], min_x, max_x, 'k-.',
                    'Boundary\nfor class 2')
    plt.xticks(())
    plt.yticks(())

    plt.xlim(min_x - .5 * max_x, max_x + .5 * max_x)
    plt.ylim(min_y - .5 * max_y, max_y + .5 * max_y)
    if subplot == 2:
        plt.xlabel('First principal component')
        plt.ylabel('Second principal component')
        plt.legend(loc="upper left")


plt.figure(figsize=(8, 6))

X, Y = make_multilabel_classification(n_classes=2, n_labels=1,
                                      allow_unlabeled=True,
                                      random_state=1)

plot_subfigure(X, Y, 1, "With unlabeled samples + CCA", "cca")
plot_subfigure(X, Y, 2, "With unlabeled samples + PCA", "pca")

X, Y = make_multilabel_classification(n_classes=2, n_labels=1,
                                      allow_unlabeled=False,
                                      random_state=1)

plot_subfigure(X, Y, 3, "Without unlabeled samples + CCA", "cca")
plot_subfigure(X, Y, 4, "Without unlabeled samples + PCA", "pca")

plt.subplots_adjust(.04, .02, .97, .94, .09, .2)
plt.show()
```

Total running time of the script: (0 minutes 0.318 seconds)

Estimated memory usage: 10 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.1.7 Comparing anomaly detection algorithms for outlier detection on toy datasets

This example shows characteristics of different anomaly detection algorithms on 2D datasets. Datasets contain one or two modes (regions of high density) to illustrate the ability of algorithms to cope with multimodal data.

For each dataset, 15% of samples are generated as random uniform noise. This proportion is the value given to the nu parameter of the OneClassSVM and the contamination parameter of the other outlier detection algorithms. Decision boundaries between inliers and outliers are displayed in black except for Local Outlier Factor (LOF) as it has no predict method to be applied on new data when it is used for outlier detection.

The `sklearn.svm.OneClassSVM` is known to be sensitive to outliers and thus does not perform very well for outlier detection. This estimator is best suited for novelty detection when the training set is not contaminated by outliers. That said, outlier detection in high-dimension, or without any assumptions on the distribution of the inlying data is very challenging, and a One-class SVM might give useful results in these situations depending on the value of its hyperparameters.

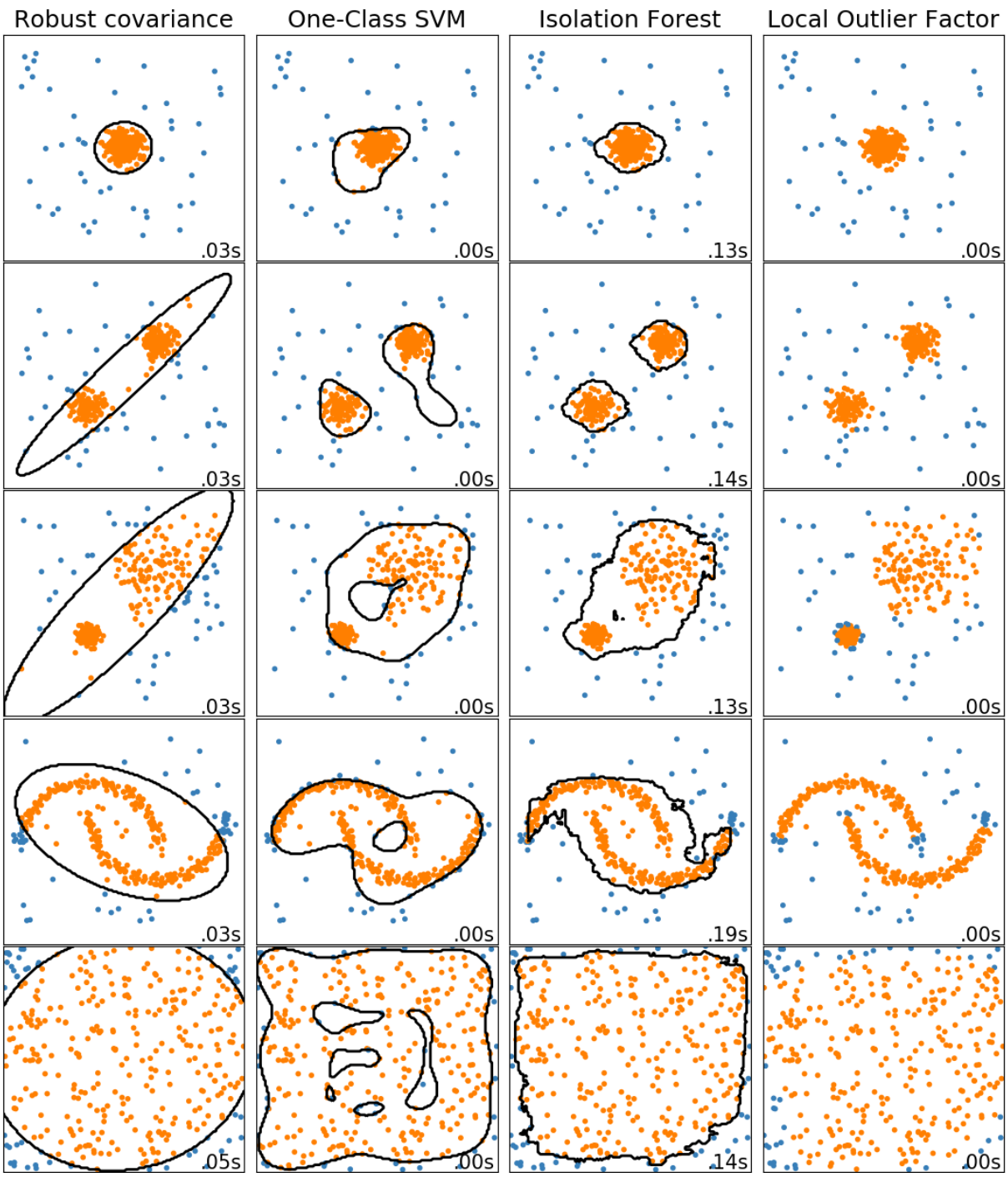
`sklearn.covariance.EllipticEnvelope` assumes the data is Gaussian and learns an ellipse. It thus degrades when the data is not unimodal. Notice however that this estimator is robust to outliers.

`sklearn.ensemble.IsolationForest` and `sklearn.neighbors.LocalOutlierFactor` seem to perform reasonably well for multi-modal data sets. The advantage of `sklearn.neighbors.LocalOutlierFactor` over the other estimators is shown for the third data set, where the two modes have different densities. This advantage is explained by the local aspect of LOF, meaning that it only compares the score of abnormality of one sample with the scores of its neighbors.

Finally, for the last data set, it is hard to say that one sample is more abnormal than another sample as they are uniformly distributed in a hypercube. Except for the `sklearn.svm.OneClassSVM` which overfits a little, all estimators present decent solutions for this situation. In such a case, it would be wise to look more closely at the scores of abnormality of the samples as a good estimator should assign similar scores to all the samples.

While these examples give some intuition about the algorithms, this intuition might not apply to very high dimensional data.

Finally, note that parameters of the models have been here handpicked but that in practice they need to be adjusted. In the absence of labelled data, the problem is completely unsupervised so model selection can be a challenge.




```
# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#         Albert Thomas <albert.thomas@telecom-paristech.fr>
# License: BSD 3 clause

import time

import numpy as np
import matplotlib
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt

from sklearn import svm
from sklearn.datasets import make_moons, make_blobs
from sklearn.covariance import EllipticEnvelope
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor

print(__doc__)

matplotlib.rcParams['contour.negative_linestyle'] = 'solid'

# Example settings
n_samples = 300
outliers_fraction = 0.15
n_outliers = int(outliers_fraction * n_samples)
n_inliers = n_samples - n_outliers

# define outlier/anomaly detection methods to be compared
anomaly_algorithms = [
    ("Robust covariance", EllipticEnvelope(contamination=outliers_fraction)),
    ("One-Class SVM", svm.OneClassSVM(nu=outliers_fraction, kernel="rbf",
                                      gamma=0.1)),
    ("Isolation Forest", IsolationForest(contamination=outliers_fraction,
                                         random_state=42)),
    ("Local Outlier Factor", LocalOutlierFactor(
        n_neighbors=35, contamination=outliers_fraction))]

# Define datasets
blobs_params = dict(random_state=0, n_samples=n_inliers, n_features=2)
datasets = [
    make_blobs(centers=[[0, 0], [0, 0]], cluster_std=0.5,
               **blobs_params)[0],
    make_blobs(centers=[[2, 2], [-2, -2]], cluster_std=[0.5, 0.5],
               **blobs_params)[0],
    make_blobs(centers=[[2, 2], [-2, -2]], cluster_std=[1.5, .3],
               **blobs_params)[0],
    4. * (make_moons(n_samples=n_samples, noise=.05, random_state=0)[0] -
          np.array([0.5, 0.25])),
    14. * (np.random.RandomState(42).rand(n_samples, 2) - 0.5)]

# Compare given classifiers under given settings
xx, yy = np.meshgrid(np.linspace(-7, 7, 150),
                     np.linspace(-7, 7, 150))

plt.figure(figsize=(len(anomaly_algorithms) * 2 + 3, 12.5))
plt.subplots_adjust(left=.02, right=.98, bottom=.001, top=.96, wspace=.05,
                    hspace=.01)

plot_num = 1
rng = np.random.RandomState(42)

for i_dataset, X in enumerate(datasets):
    # Add outliers
    X = np.concatenate([X, rng.uniform(low=-6, high=6,
                       size=(n_outliers, 2))], axis=0)
```

(continues on next page)

(continued from previous page)

```
    for name, algorithm in anomaly_algorithms:
        t0 = time.time()
        algorithm.fit(X)
        t1 = time.time()
        plt.subplot(len(datasets), len(anomaly_algorithms), plot_num)
        if i_dataset == 0:
            plt.title(name, size=18)

        # fit the data and tag outliers
        if name == "Local Outlier Factor":
            y_pred = algorithm.fit_predict(X)
        else:
            y_pred = algorithm.fit(X).predict(X)

        # plot the levels lines and the points
        if name != "Local Outlier Factor":  # LOF does not implement predict
            Z = algorithm.predict(np.c_[xx.ravel(), yy.ravel()])
            Z = Z.reshape(xx.shape)
            plt.contour(xx, yy, Z, levels=[0], linewidths=2, colors='black')

        colors = np.array(['#377eb8', '#ff7f00'])
        plt.scatter(X[:, 0], X[:, 1], s=10, color=colors[(y_pred + 1) // 2])

        plt.xlim(-7, 7)
        plt.ylim(-7, 7)
        plt.xticks(())
        plt.yticks(())
        plt.text(.99, .01, ('%.2fs' % (t1 - t0)).lstrip('0'),
                 transform=plt.gca().transAxes, size=15,
                 horizontalalignment='right')
        plot_num += 1

plt.show()
```

Total running time of the script: (0 minutes 4.223 seconds)

Estimated memory usage: 9 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.1.8 The Johnson-Lindenstrauss bound for embedding with random projections

The Johnson-Lindenstrauss lemma states that any high dimensional dataset can be randomly projected into a lower dimensional Euclidean space while controlling the distortion in the pairwise distances.

```
print(__doc__)

import sys
from time import time
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from distutils.version import LooseVersion
from sklearn.random_projection import johnson_lindenstrauss_min_dim
```

(continues on next page)

(continued from previous page)

```
from sklearn.random_projection import SparseRandomProjection
from sklearn.datasets import fetch_20newsgroups_vectorized
from sklearn.datasets import load_digits
from sklearn.metrics.pairwise import euclidean_distances

# `normed` is being deprecated in favor of `density` in histograms
if LooseVersion(matplotlib.__version__) >= '2.1':
    density_param = {'density': True}
else:
    density_param = {'normed': True}
```

Theoretical bounds

The distortion introduced by a random projection `p` is asserted by the fact that `p` is defining an eps-embedding with good probability as defined by:

$$(1 - eps)\|u - v\|^2 < \|p(u) - p(v)\|^2 < (1 + eps)\|u - v\|^2$$

Where u and v are any rows taken from a dataset of shape [n_samples, n_features] and p is a projection by a random Gaussian N(0, 1) matrix with shape [n_components, n_features] (or a sparse Achlioptas matrix).

The minimum number of components to guarantees the eps-embedding is given by:

$$n\_components >= 4log(n\_samples)/(eps^2/2 - eps^3/3)$$

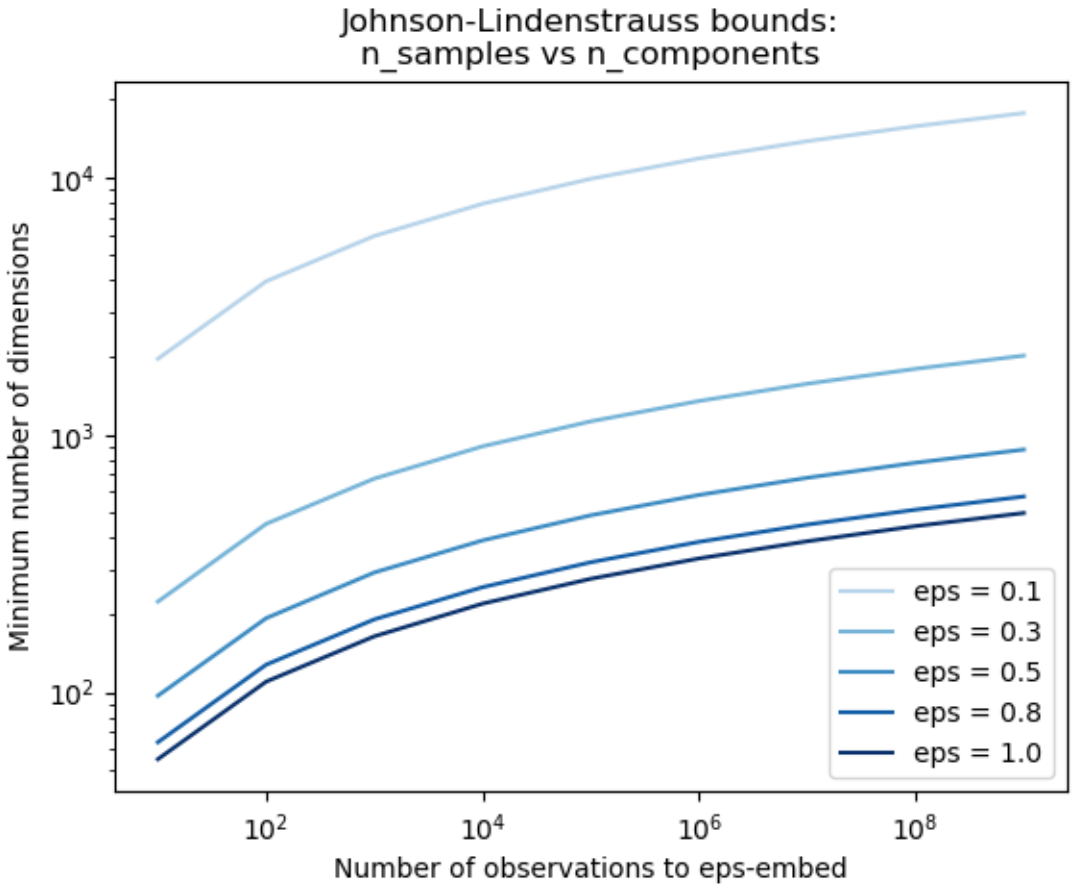
The first plot shows that with an increasing number of samples `n_samples`, the minimal number of dimensions `n_components` increased logarithmically in order to guarantee an `eps`-embedding.

```
# range of admissible distortions
eps_range = np.linspace(0.1, 0.99, 5)
colors = plt.cm.Blues(np.linspace(0.3, 1.0, len(eps_range)))

# range of number of samples (observation) to embed
n_samples_range = np.logspace(1, 9, 9)

plt.figure()
for eps, color in zip(eps_range, colors):
    min_n_components = johnson_lindenstrauss_min_dim(n_samples_range, eps=eps)
    plt.loglog(n_samples_range, min_n_components, color=color)

plt.legend(["eps = %0.1f" % eps for eps in eps_range], loc="lower right")
plt.xlabel("Number of observations to eps-embed")
plt.ylabel("Minimum number of dimensions")
plt.title("Johnson-Lindenstrauss bounds:\nn_samples vs n_components")
plt.show()
```



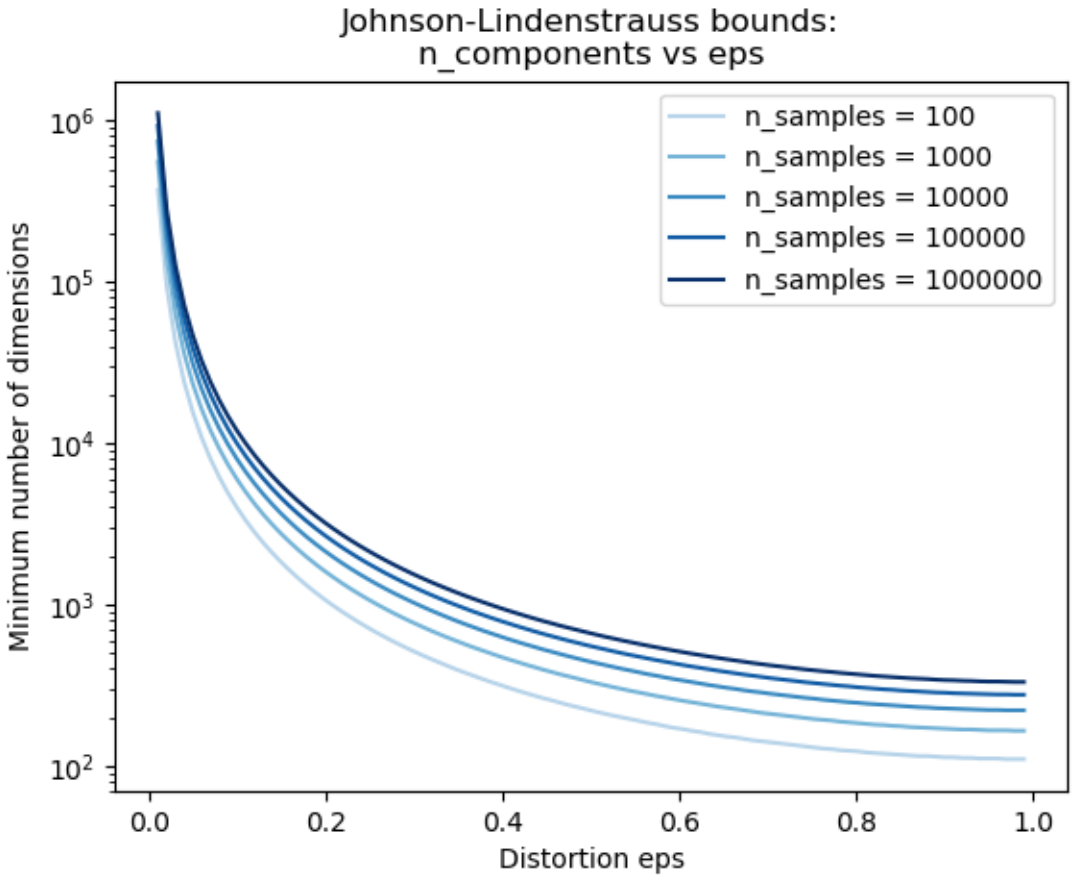
The second plot shows that an increase of the admissible distortion `eps` allows to reduce drastically the minimal number of dimensions `n_components` for a given number of samples `n_samples`

```
# range of admissible distortions
eps_range = np.linspace(0.01, 0.99, 100)

# range of number of samples (observation) to embed
n_samples_range = np.logspace(2, 6, 5)
colors = plt.cm.Blues(np.linspace(0.3, 1.0, len(n_samples_range)))

plt.figure()
for n_samples, color in zip(n_samples_range, colors):
    min_n_components = johnson_lindenstrauss_min_dim(n_samples, eps=eps_range)
    plt.semilogy(eps_range, min_n_components, color=color)

plt.legend(["n_samples = %d" % n for n in n_samples_range], loc="upper right")
plt.xlabel("Distortion eps")
plt.ylabel("Minimum number of dimensions")
plt.title("Johnson-Lindenstrauss bounds:\nn_components vs eps")
plt.show()
```



Empirical validation

We validate the above bounds on the 20 newsgroups text document (TF-IDF word frequencies) dataset or on the digits dataset:

- for the 20 newsgroups dataset some 500 documents with 100k features in total are projected using a sparse random matrix to smaller euclidean spaces with various values for the target number of dimensions `n_components`.
- for the digits dataset, some 8x8 gray level pixels data for 500 handwritten digits pictures are randomly projected to spaces for various larger number of dimensions `n_components`.

The default dataset is the 20 newsgroups dataset. To run the example on the digits dataset, pass the `--use-digits-dataset` command line argument to this script.

```
if '--use-digits-dataset' in sys.argv:
    data = load_digits().data[:500]
else:
    data = fetch_20newsgroups_vectorized().data[:500]
```

For each value of `n_components`, we plot:

- 2D distribution of sample pairs with pairwise distances in original and projected spaces as x and y axis respectively.
- 1D histogram of the ratio of those distances (projected / original).

```
n_samples, n_features = data.shape
print("Embedding %d samples with dim %d using various random projections"
      % (n_samples, n_features))

n_components_range = np.array([300, 1000, 10000])
dists = euclidean_distances(data, squared=True).ravel()

# select only non-identical samples pairs
nonzero = dists != 0
dists = dists[nonzero]

for n_components in n_components_range:
    t0 = time()
    rp = SparseRandomProjection(n_components=n_components)
    projected_data = rp.fit_transform(data)
    print("Projected %d samples from %d to %d in %0.3fs"
          % (n_samples, n_features, n_components, time() - t0))
    if hasattr(rp, 'components_'):
        n_bytes = rp.components_.data.nbytes
        n_bytes += rp.components_.indices.nbytes
        print("Random matrix with size: %0.3fMB" % (n_bytes / 1e6))

    projected_dists = euclidean_distances(
        projected_data, squared=True).ravel()[nonzero]

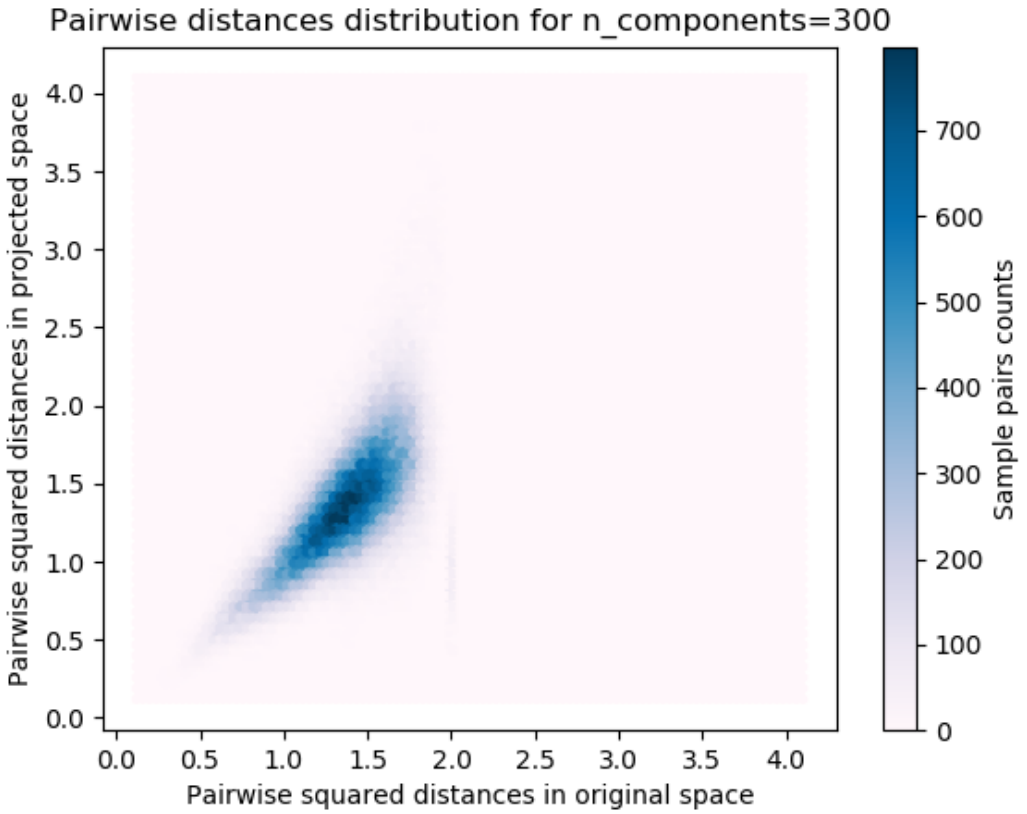
    plt.figure()
    min_dist = min(projected_dists.min(), dists.min())
    max_dist = max(projected_dists.max(), dists.max())
    plt.hexbin(dists, projected_dists, gridsize=100, cmap=plt.cm.PuBu,
               extent=[min_dist, max_dist, min_dist, max_dist])
    plt.xlabel("Pairwise squared distances in original space")
    plt.ylabel("Pairwise squared distances in projected space")
    plt.title("Pairwise distances distribution for n_components=%d" %
              n_components)
    cb = plt.colorbar()
    cb.set_label('Sample pairs counts')

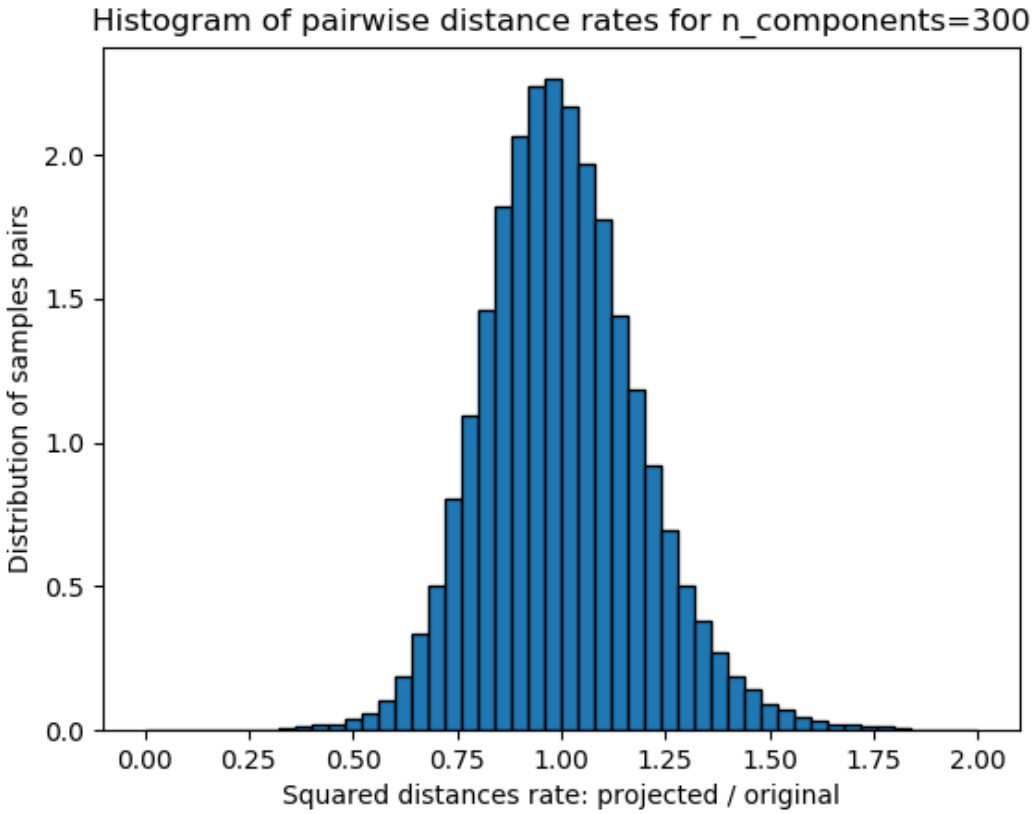
    rates = projected_dists / dists
    print("Mean distances rate: %0.2f (%0.2f)"
          % (np.mean(rates), np.std(rates)))

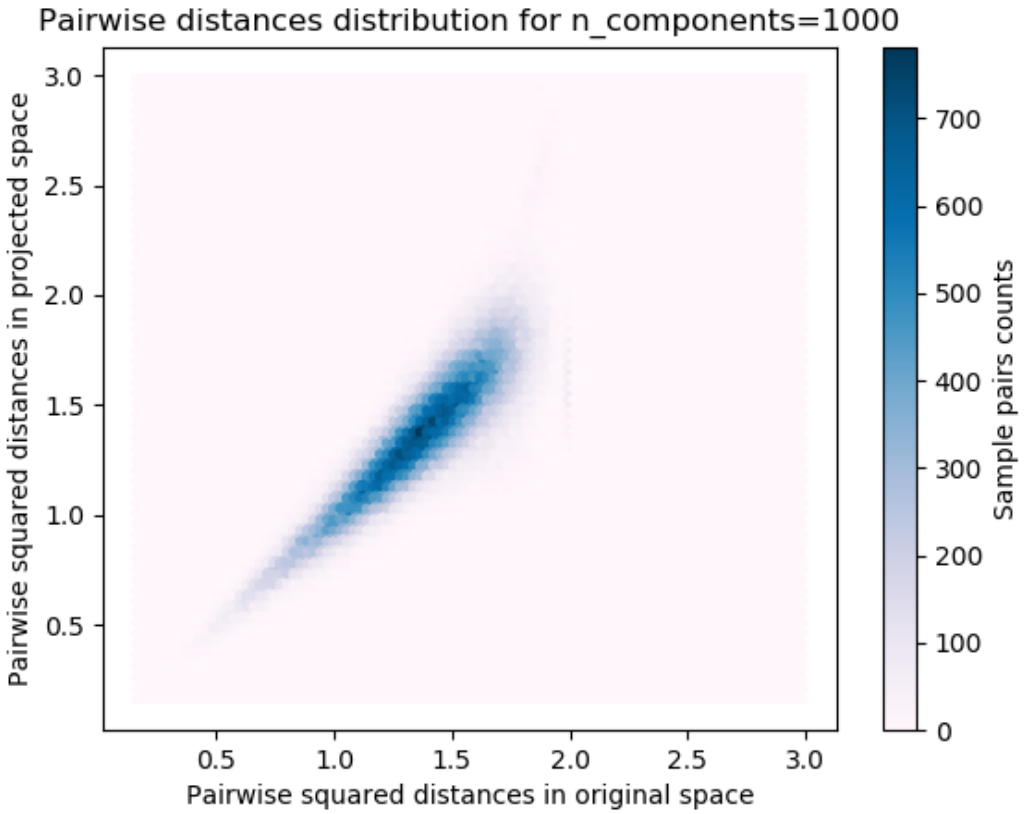
    plt.figure()
    plt.hist(rates, bins=50, range=(0., 2.), edgecolor='k', **density_param)
    plt.xlabel("Squared distances rate: projected / original")
    plt.ylabel("Distribution of samples pairs")
    plt.title("Histogram of pairwise distance rates for n_components=%d" %
              n_components)

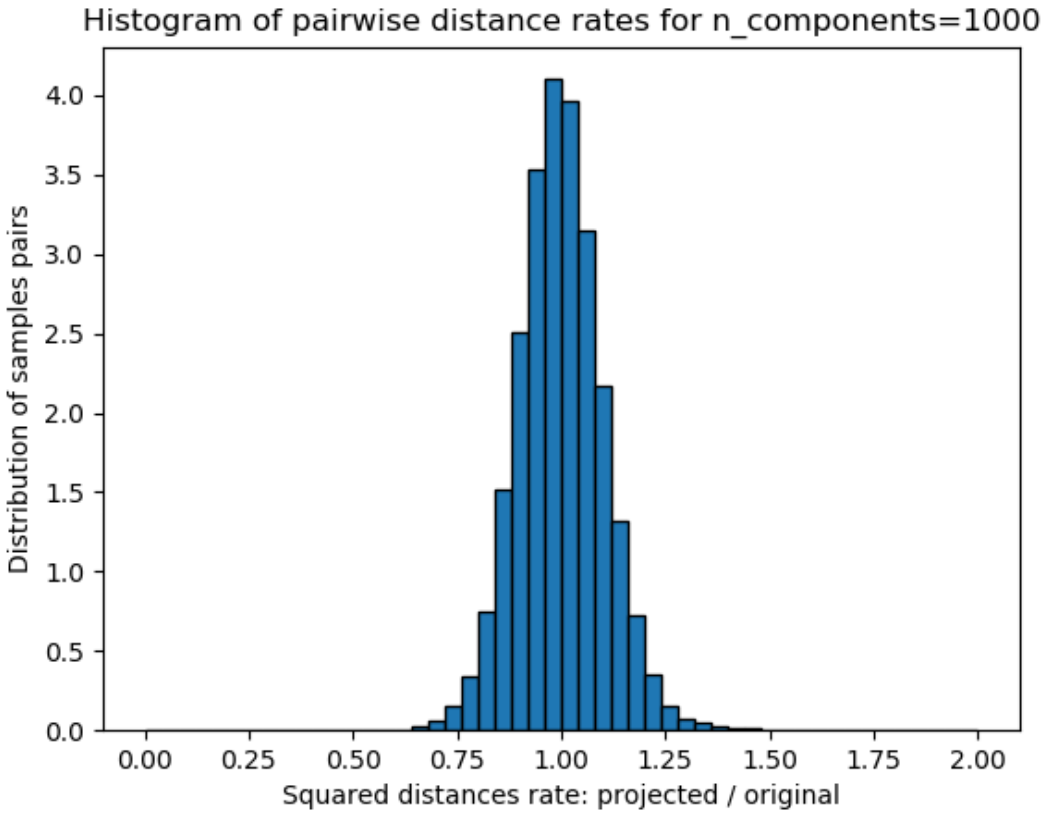
    # TODO: compute the expected value of eps and add them to the previous plot
    # as vertical lines / region

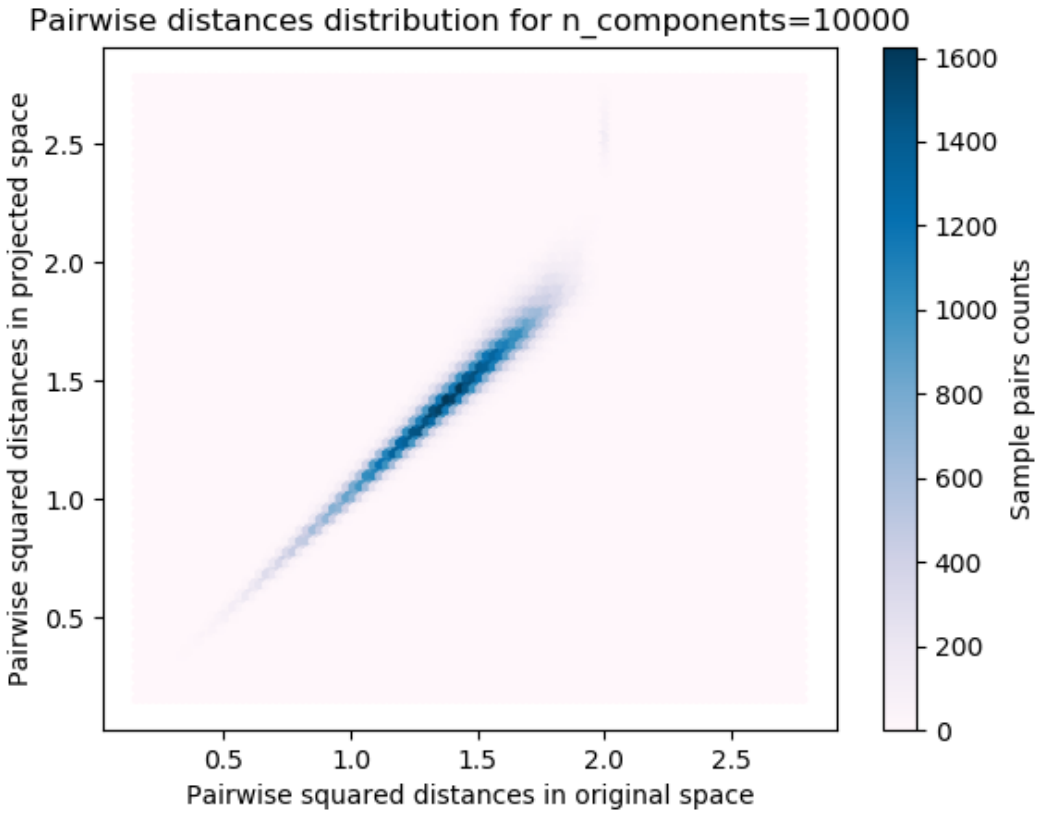
plt.show()
```

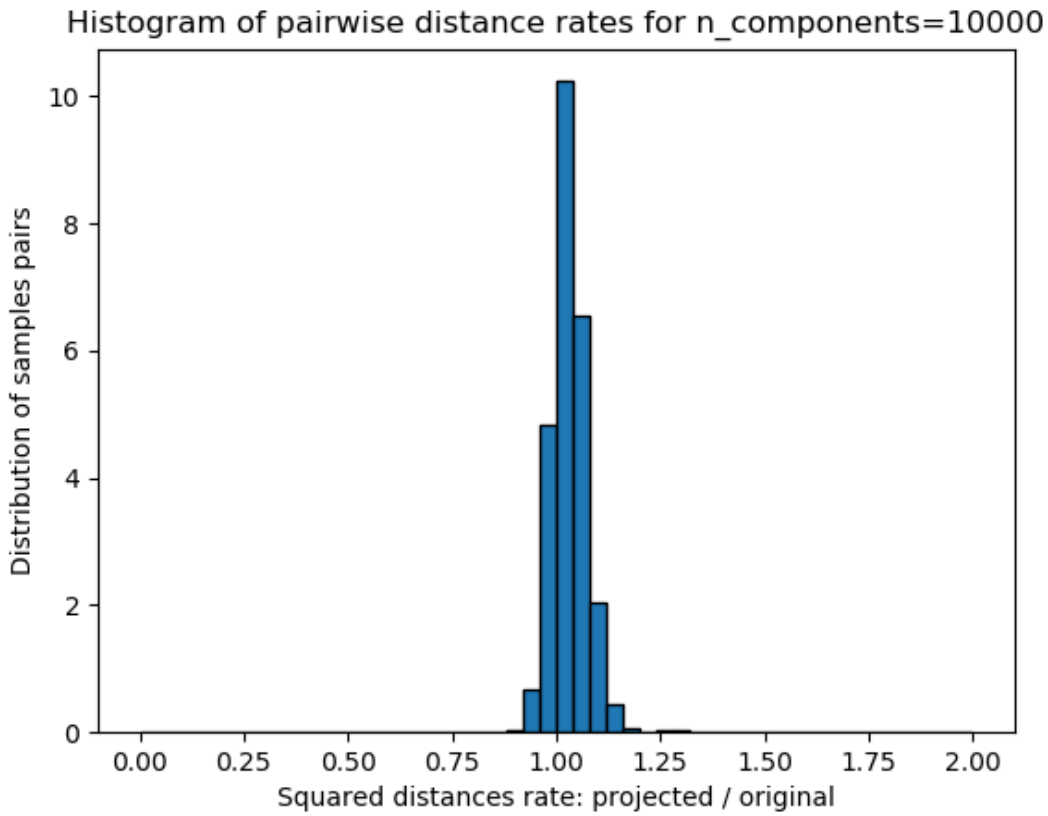

Pairwise distances distribution for n_components=300
Pairwise squared distances in projected space
Pairwise squared distances in original space
Sample pairs counts
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
100
200
300
400
500
600
700


Histogram of pairwise distance rates for n_components=300
Distribution of samples pairs
0.0
0.5
1.0
1.5
2.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
Squared distances rate: projected / original


Pairwise distances distribution for n_components=1000
Pairwise squared distances in projected space
Pairwise squared distances in original space
Sample pairs counts
3.0
2.5
2.0
1.5
1.0
0.5
0.5
1.0
1.5
2.0
2.5
3.0
700
600
500
400
300
200
100
0


Histogram of pairwise distance rates for n_components=1000
Distribution of samples pairs
Squared distances rate: projected / original
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


Pairwise distances distribution for n_components=10000
Pairwise squared distances in projected space
Pairwise squared distances in original space
Sample pairs counts
0.5
1.0
1.5
2.0
2.5
0
200
400
600
800
1000
1200
1400
1600



Out:

```
Embedding 500 samples with dim 130107 using various random projections
Projected 500 samples from 130107 to 300 in 0.632s
Random matrix with size: 1.296MB
Mean distances rate: 1.00 (0.19)
Projected 500 samples from 130107 to 1000 in 2.118s
Random matrix with size: 4.328MB
Mean distances rate: 1.00 (0.10)
Projected 500 samples from 130107 to 10000 in 22.211s
Random matrix with size: 43.315MB
Mean distances rate: 1.03 (0.04)
```

We can see that for low values of `n_components` the distribution is wide with many distorted pairs and a skewed distribution (due to the hard limit of zero ratio on the left as distances are always positives) while for larger values of n_components the distortion is controlled and the distances are well preserved by the random projection.

Remarks

According to the JL lemma, projecting 500 samples without too much distortion will require at least several thousands dimensions, irrespective of the number of features of the original dataset.

Hence using random projections on the digits dataset which only has 64 features in the input space does not make sense: it does not allow for dimensionality reduction in this case.

On the twenty newsgroups on the other hand the dimensionality can be decreased from 56436 down to 10000 while reasonably preserving pairwise distances.

Total running time of the script: (0 minutes 28.844 seconds)

Estimated memory usage: 96 MB

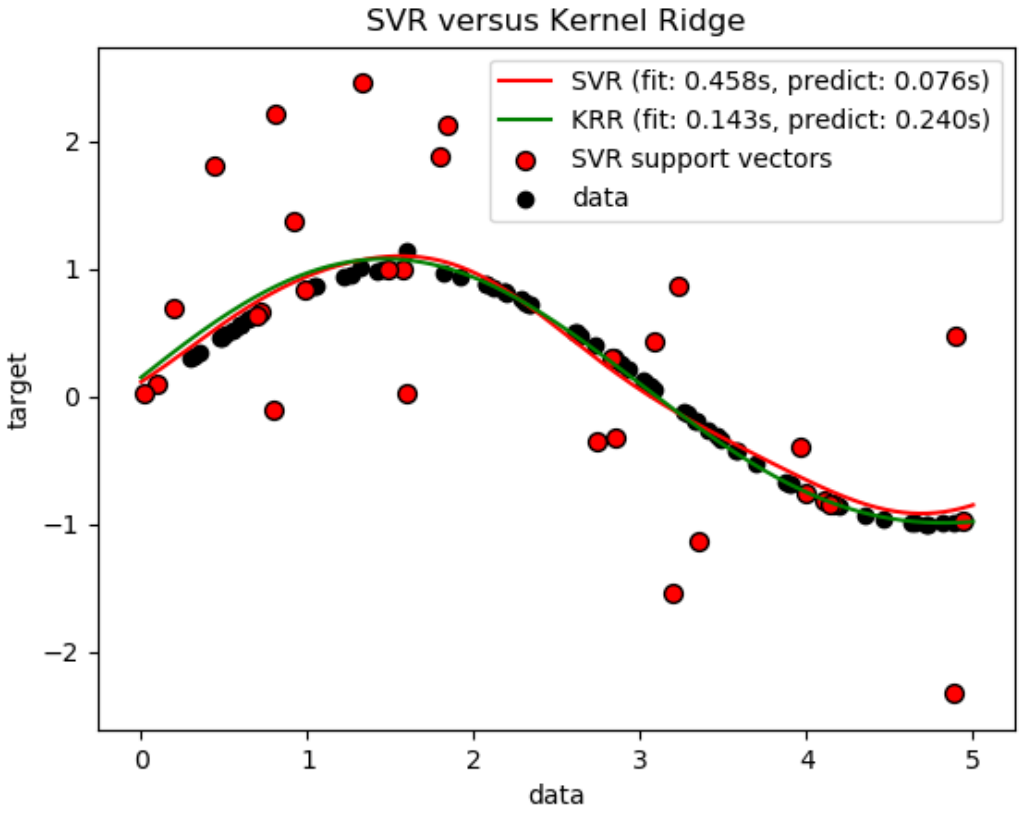
Note: Click *here* to download the full example code or to run this example in your browser via Binder

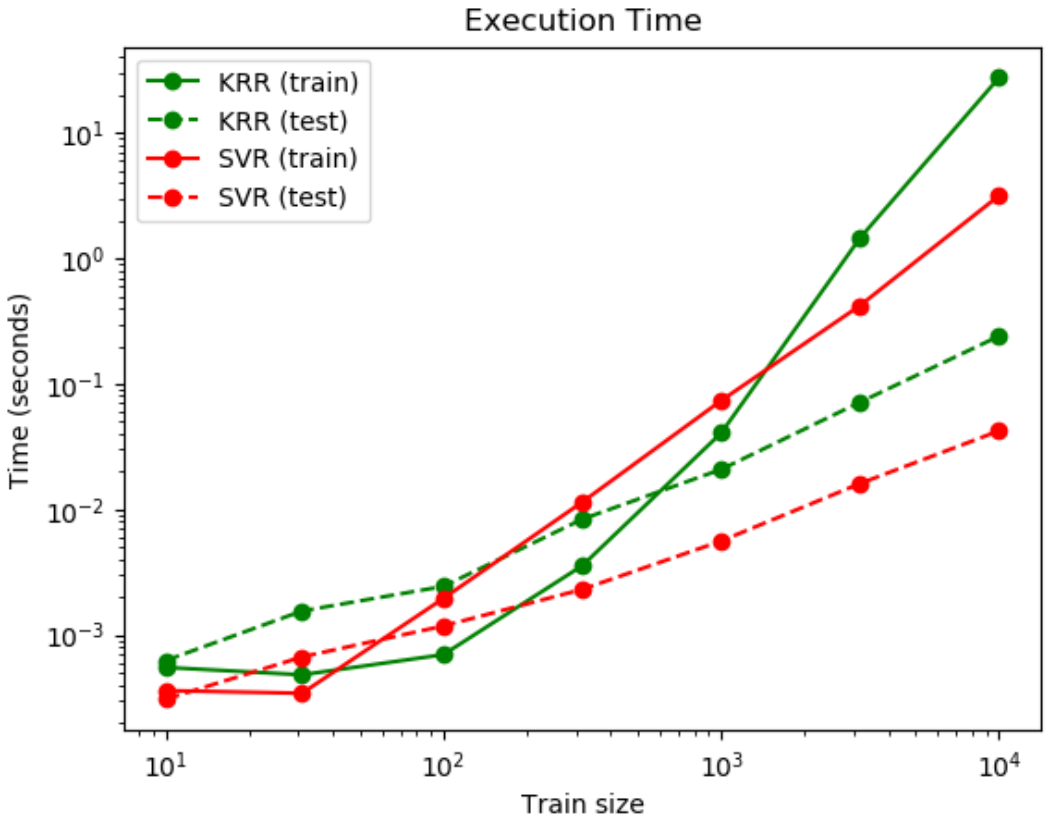
6.1.9 Comparison of kernel ridge regression and SVR

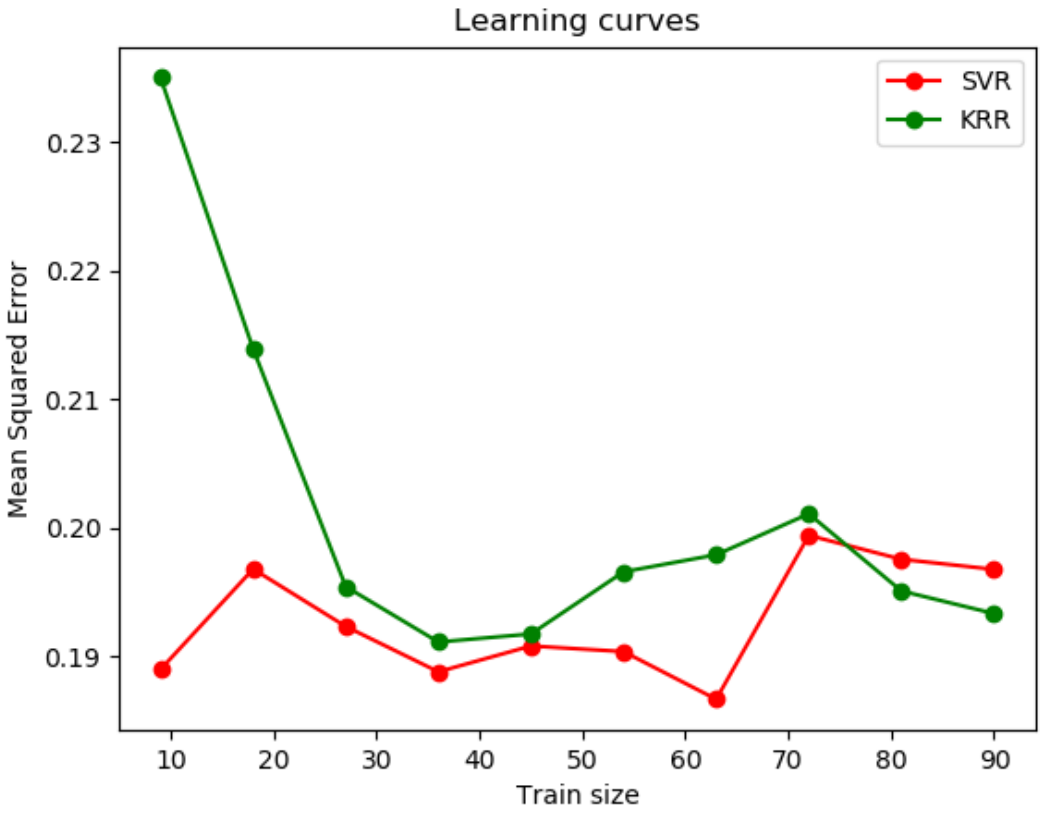
Both kernel ridge regression (KRR) and SVR learn a non-linear function by employing the kernel trick, i.e., they learn a linear function in the space induced by the respective kernel which corresponds to a non-linear function in the original space. They differ in the loss functions (ridge versus epsilon-insensitive loss). In contrast to SVR, fitting a KRR can be done in closed-form and is typically faster for medium-sized datasets. On the other hand, the learned model is non-sparse and thus slower than SVR at prediction-time.

This example illustrates both methods on an artificial dataset, which consists of a sinusoidal target function and strong noise added to every fifth datapoint. The first figure compares the learned model of KRR and SVR when both complexity/regularization and bandwidth of the RBF kernel are optimized using grid-search. The learned functions are very similar; however, fitting KRR is approx. seven times faster than fitting SVR (both with grid-search). However, prediction of 100000 target values is more than tree times faster with SVR since it has learned a sparse model using only approx. 1/3 of the 100 training datapoints as support vectors.

The next figure compares the time for fitting and prediction of KRR and SVR for different sizes of the training set. Fitting KRR is faster than SVR for medium- sized training sets (less than 1000 samples); however, for larger training sets SVR scales better. With regard to prediction time, SVR is faster than KRR for all sizes of the training set because of the learned sparse solution. Note that the degree of sparsity and thus the prediction time depends on the parameters epsilon and C of the SVR.


SVR versus Kernel Ridge
SVR (fit: 0.458s, predict: 0.076s)
KRR (fit: 0.143s, predict: 0.240s)
SVR support vectors
data
target
data
2
1
0
−1
−2
0
1
2
3
4
5


Execution Time
KRR (train)
KRR (test)
SVR (train)
SVR (test)
Time (seconds)
Train size
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{1}
10^{2}
10^{3}
10^{4}



Out:

```
SVR complexity and bandwidth selected and model fitted in 0.458 s
KRR complexity and bandwidth selected and model fitted in 0.143 s
Support vector ratio: 0.320
SVR prediction for 100000 inputs in 0.076 s
KRR prediction for 100000 inputs in 0.240 s
```

```
# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD 3 clause


import time

import numpy as np

from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import learning_curve
from sklearn.kernel_ridge import KernelRidge
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
rng = np.random.RandomState(0)

# ##############################################################################
# Generate sample data
X = 5 * rng.rand(10000, 1)
y = np.sin(X).ravel()

# Add noise to targets
y[::5] += 3 * (0.5 - rng.rand(X.shape[0] // 5))

X_plot = np.linspace(0, 5, 100000)[:, None]

# ##############################################################################
# Fit regression model
train_size = 100
svr = GridSearchCV(SVR(kernel='rbf', gamma=0.1),
                   param_grid={"C": [1e0, 1e1, 1e2, 1e3],
                               "gamma": np.logspace(-2, 2, 5)})

kr = GridSearchCV(KernelRidge(kernel='rbf', gamma=0.1),
                  param_grid={"alpha": [1e0, 0.1, 1e-2, 1e-3],
                              "gamma": np.logspace(-2, 2, 5)})

t0 = time.time()
svr.fit(X[:train_size], y[:train_size])
svr_fit = time.time() - t0
print("SVR complexity and bandwidth selected and model fitted in %.3f s"
      % svr_fit)

t0 = time.time()
kr.fit(X[:train_size], y[:train_size])
kr_fit = time.time() - t0
print("KRR complexity and bandwidth selected and model fitted in %.3f s"
      % kr_fit)

sv_ratio = svr.best_estimator_.support_.shape[0] / train_size
print("Support vector ratio: %.3f" % sv_ratio)

t0 = time.time()
y_svr = svr.predict(X_plot)
svr_predict = time.time() - t0
print("SVR prediction for %d inputs in %.3f s"
      % (X_plot.shape[0], svr_predict))

t0 = time.time()
y_kr = kr.predict(X_plot)
kr_predict = time.time() - t0
print("KRR prediction for %d inputs in %.3f s"
      % (X_plot.shape[0], kr_predict))


# ##############################################################################
# Look at the results
sv_ind = svr.best_estimator_.support_
plt.scatter(X[sv_ind], y[sv_ind], c='r', s=50, label='SVR support vectors',
            zorder=2, edgecolors=(0, 0, 0))
plt.scatter(X[:100], y[:100], c='k', label='data', zorder=1,
```

(continues on next page)

(continued from previous page)

```
            edgecolors=(0, 0, 0))
plt.plot(X_plot, y_svr, c='r',
         label='SVR (fit: %.3fs, predict: %.3fs)' % (svr_fit, svr_predict))
plt.plot(X_plot, y_kr, c='g',
         label='KRR (fit: %.3fs, predict: %.3fs)' % (kr_fit, kr_predict))
plt.xlabel('data')
plt.ylabel('target')
plt.title('SVR versus Kernel Ridge')
plt.legend()

# Visualize training and prediction time
plt.figure()

# Generate sample data
X = 5 * rng.rand(10000, 1)
y = np.sin(X).ravel()
y[::5] += 3 * (0.5 - rng.rand(X.shape[0] // 5))
sizes = np.logspace(1, 4, 7).astype(np.int)
for name, estimator in {"KRR": KernelRidge(kernel='rbf', alpha=0.1,
                                           gamma=10),
                        "SVR": SVR(kernel='rbf', C=1e1, gamma=10)}.items():
    train_time = []
    test_time = []
    for train_test_size in sizes:
        t0 = time.time()
        estimator.fit(X[:train_test_size], y[:train_test_size])
        train_time.append(time.time() - t0)

        t0 = time.time()
        estimator.predict(X_plot[:1000])
        test_time.append(time.time() - t0)

    plt.plot(sizes, train_time, 'o-', color="r" if name == "SVR" else "g",
             label="%s (train)" % name)
    plt.plot(sizes, test_time, 'o--', color="r" if name == "SVR" else "g",
             label="%s (test)" % name)

plt.xscale("log")
plt.yscale("log")
plt.xlabel("Train size")
plt.ylabel("Time (seconds)")
plt.title('Execution Time')
plt.legend(loc="best")

# Visualize learning curves
plt.figure()

svr = SVR(kernel='rbf', C=1e1, gamma=0.1)
kr = KernelRidge(kernel='rbf', alpha=0.1, gamma=0.1)
train_sizes, train_scores_svr, test_scores_svr = \
    learning_curve(svr, X[:100], y[:100], train_sizes=np.linspace(0.1, 1, 10),
                   scoring="neg_mean_squared_error", cv=10)
train_sizes_abs, train_scores_kr, test_scores_kr = \
    learning_curve(kr, X[:100], y[:100], train_sizes=np.linspace(0.1, 1, 10),
                   scoring="neg_mean_squared_error", cv=10)

plt.plot(train_sizes, -test_scores_svr.mean(1), 'o-', color="r",
```

(continues on next page)

(continued from previous page)

```
         label="SVR")
plt.plot(train_sizes, -test_scores_kr.mean(1), 'o-', color="g",
         label="KRR")
plt.xlabel("Train size")
plt.ylabel("Mean Squared Error")
plt.title('Learning curves')
plt.legend(loc="best")

plt.show()
```

Total running time of the script: (0 minutes 34.921 seconds)

Estimated memory usage: 1561 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.1.10 Explicit feature map approximation for RBF kernels

An example illustrating the approximation of the feature map of an RBF kernel.

It shows how to use *RBFSampler* and *Nystroem* to approximate the feature map of an RBF kernel for classification with an SVM on the digits dataset. Results using a linear SVM in the original space, a linear SVM using the approximate mappings and using a kernelized SVM are compared. Timings and accuracy for varying amounts of Monte Carlo samplings (in the case of *RBFSampler*, which uses random Fourier features) and different sized subsets of the training set (for *Nystroem*) for the approximate mapping are shown.

Please note that the dataset here is not large enough to show the benefits of kernel approximation, as the exact SVM is still reasonably fast.

Sampling more dimensions clearly leads to better classification results, but comes at a greater cost. This means there is a tradeoff between runtime and accuracy, given by the parameter n_components. Note that solving the Linear SVM and also the approximate kernel SVM could be greatly accelerated by using stochastic gradient descent via *sklearn.linear_model.SGDClassifier*. This is not easily possible for the case of the kernelized SVM.

Python package and dataset imports, load dataset

```
# Author: Gael Varoquaux <gael dot varoquaux at normalesup dot org>
#         Andreas Mueller <amueller@ais.uni-bonn.de>
# License: BSD 3 clause

print(__doc__)

# Standard scientific Python imports
import matplotlib.pyplot as plt
import numpy as np
from time import time

# Import datasets, classifiers and performance metrics
from sklearn import datasets, svm, pipeline
from sklearn.kernel_approximation import (RBFSampler,
                                          Nystroem)
from sklearn.decomposition import PCA
```

(continues on next page)

(continued from previous page)

```
# The digits dataset
digits = datasets.load_digits(n_class=9)
```

Timing and accuracy plots

To apply an classifier on this data, we need to flatten the image, to turn the data in a (samples, feature) matrix:

```
n_samples = len(digits.data)
data = digits.data / 16.
data -= data.mean(axis=0)

# We learn the digits on the first half of the digits
data_train, targets_train = (data[:n_samples // 2],
                             digits.target[:n_samples // 2])


# Now predict the value of the digit on the second half:
data_test, targets_test = (data[n_samples // 2:],
                           digits.target[n_samples // 2:])
# data_test = scaler.transform(data_test)

# Create a classifier: a support vector classifier
kernel_svm = svm.SVC(gamma=.2)
linear_svm = svm.LinearSVC()

# create pipeline from kernel approximation
# and linear svm
feature_map_fourier = RBFSampler(gamma=.2, random_state=1)
feature_map_nystroem = Nystroem(gamma=.2, random_state=1)
fourier_approx_svm = pipeline.Pipeline([("feature_map", feature_map_fourier),
                                        ("svm", svm.LinearSVC())])

nystroem_approx_svm = pipeline.Pipeline([("feature_map", feature_map_nystroem),
                                         ("svm", svm.LinearSVC())])

# fit and predict using linear and kernel svm:

kernel_svm_time = time()
kernel_svm.fit(data_train, targets_train)
kernel_svm_score = kernel_svm.score(data_test, targets_test)
kernel_svm_time = time() - kernel_svm_time

linear_svm_time = time()
linear_svm.fit(data_train, targets_train)
linear_svm_score = linear_svm.score(data_test, targets_test)
linear_svm_time = time() - linear_svm_time

sample_sizes = 30 * np.arange(1, 10)
fourier_scores = []
nystroem_scores = []
fourier_times = []
nystroem_times = []

for D in sample_sizes:
    fourier_approx_svm.set_params(feature_map__n_components=D)
```

(continues on next page)

(continued from previous page)

```
    nystroem_approx_svm.set_params(feature_map__n_components=D)
    start = time()
    nystroem_approx_svm.fit(data_train, targets_train)
    nystroem_times.append(time() - start)

    start = time()
    fourier_approx_svm.fit(data_train, targets_train)
    fourier_times.append(time() - start)

    fourier_score = fourier_approx_svm.score(data_test, targets_test)
    nystroem_score = nystroem_approx_svm.score(data_test, targets_test)
    nystroem_scores.append(nystroem_score)
    fourier_scores.append(fourier_score)

# plot the results:
plt.figure(figsize=(16, 4))
accuracy = plt.subplot(121)
# second y axis for timings
timescale = plt.subplot(122)

accuracy.plot(sample_sizes, nystroem_scores, label="Nystroem approx. kernel")
timescale.plot(sample_sizes, nystroem_times, '--',
               label='Nystroem approx. kernel')

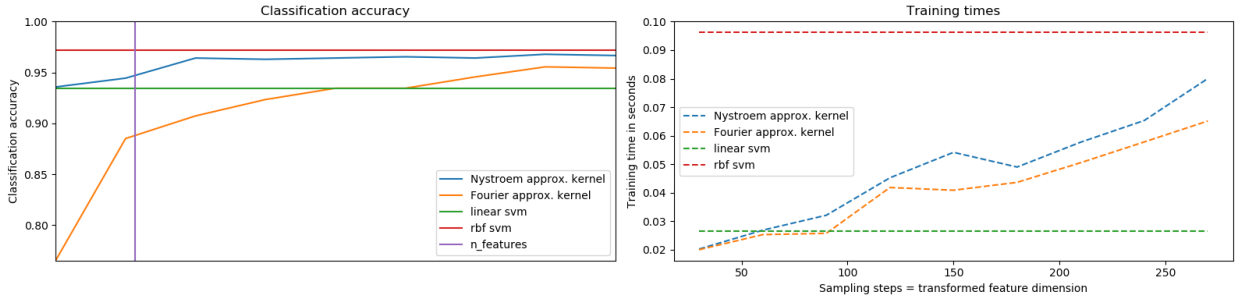
accuracy.plot(sample_sizes, fourier_scores, label="Fourier approx. kernel")
timescale.plot(sample_sizes, fourier_times, '--',
               label='Fourier approx. kernel')

# horizontal lines for exact rbf and linear kernels:
accuracy.plot([sample_sizes[0], sample_sizes[-1]],
              [linear_svm_score, linear_svm_score], label="linear svm")
timescale.plot([sample_sizes[0], sample_sizes[-1]],
               [linear_svm_time, linear_svm_time], '--', label='linear svm')

accuracy.plot([sample_sizes[0], sample_sizes[-1]],
              [kernel_svm_score, kernel_svm_score], label="rbf svm")
timescale.plot([sample_sizes[0], sample_sizes[-1]],
               [kernel_svm_time, kernel_svm_time], '--', label='rbf svm')

# vertical line for dataset dimensionality = 64
accuracy.plot([64, 64], [0.7, 1], label="n_features")

# legends and labels
accuracy.set_title("Classification accuracy")
timescale.set_title("Training times")
accuracy.set_xlim(sample_sizes[0], sample_sizes[-1])
accuracy.set_xticks(())
accuracy.set_ylim(np.min(fourier_scores), 1)
timescale.set_xlabel("Sampling steps = transformed feature dimension")
accuracy.set_ylabel("Classification accuracy")
timescale.set_ylabel("Training time in seconds")
accuracy.legend(loc='best')
timescale.legend(loc='best')
plt.tight_layout()
plt.show()
```



Decision Surfaces of RBF Kernel SVM and Linear SVM

The second plot visualized the decision surfaces of the RBF kernel SVM and the linear SVM with approximate kernel maps. The plot shows decision surfaces of the classifiers projected onto the first two principal components of the data. This visualization should be taken with a grain of salt since it is just an interesting slice through the decision surface in 64 dimensions. In particular note that a datapoint (represented as a dot) does not necessarily be classified into the region it is lying in, since it will not lie on the plane that the first two principal components span. The usage of *RBFSampler* and *Nystroem* is described in detail in *Kernel Approximation*.

```
# visualize the decision surface, projected down to the first
# two principal components of the dataset
pca = PCA(n_components=8).fit(data_train)

X = pca.transform(data_train)

# Generate grid along first two principal components
multiples = np.arange(-2, 2, 0.1)
# steps along first component
first = multiples[:, np.newaxis] * pca.components_[0, :]
# steps along second component
second = multiples[:, np.newaxis] * pca.components_[1, :]
# combine
grid = first[np.newaxis, :, :] + second[:, np.newaxis, :]
flat_grid = grid.reshape(-1, data.shape[1])

# title for the plots
titles = ['SVC with rbf kernel',
          'SVC (linear kernel)\n with Fourier rbf feature map\n'
          'n_components=100',
          'SVC (linear kernel)\n with Nystroem rbf feature map\n'
          'n_components=100']

plt.figure(figsize=(18, 7.5))
plt.rcParams.update({'font.size': 14})
# predict and plot
for i, clf in enumerate((kernel_svm, nystroem_approx_svm,
                         fourier_approx_svm)):
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    plt.subplot(1, 3, i + 1)
    Z = clf.predict(flat_grid)

    # Put the result into a color plot
    Z = Z.reshape(grid.shape[:-1])
    plt.contourf(multiples, multiples, Z, cmap=plt.cm.Paired)
```

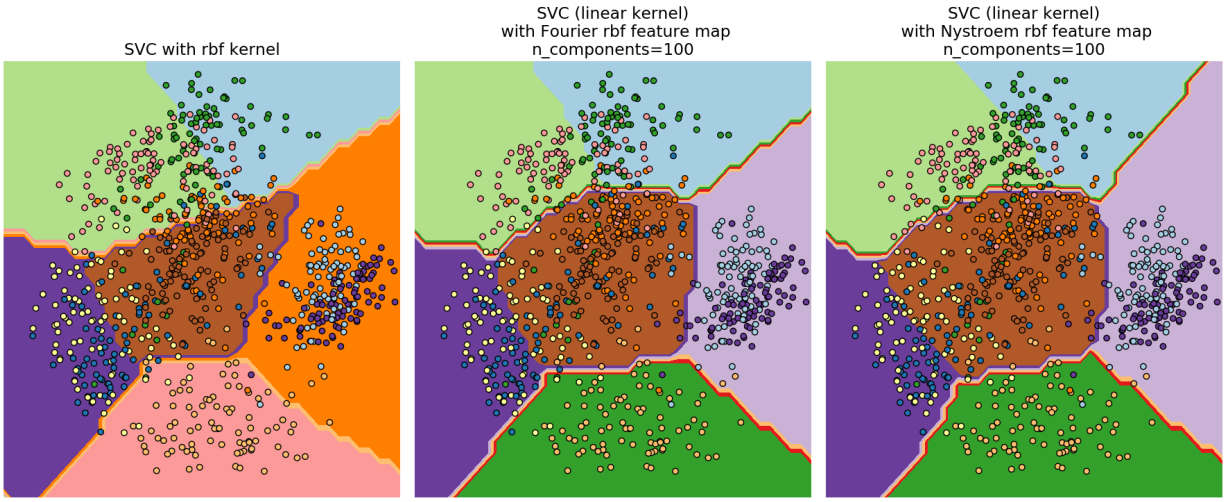
(continues on next page)

(continued from previous page)

```
    plt.axis('off')

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=targets_train, cmap=plt.cm.Paired,
                edgecolors=(0, 0, 0))

    plt.title(titles[i])
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 2.019 seconds)

Estimated memory usage: 10 MB

6.2 Biclustering

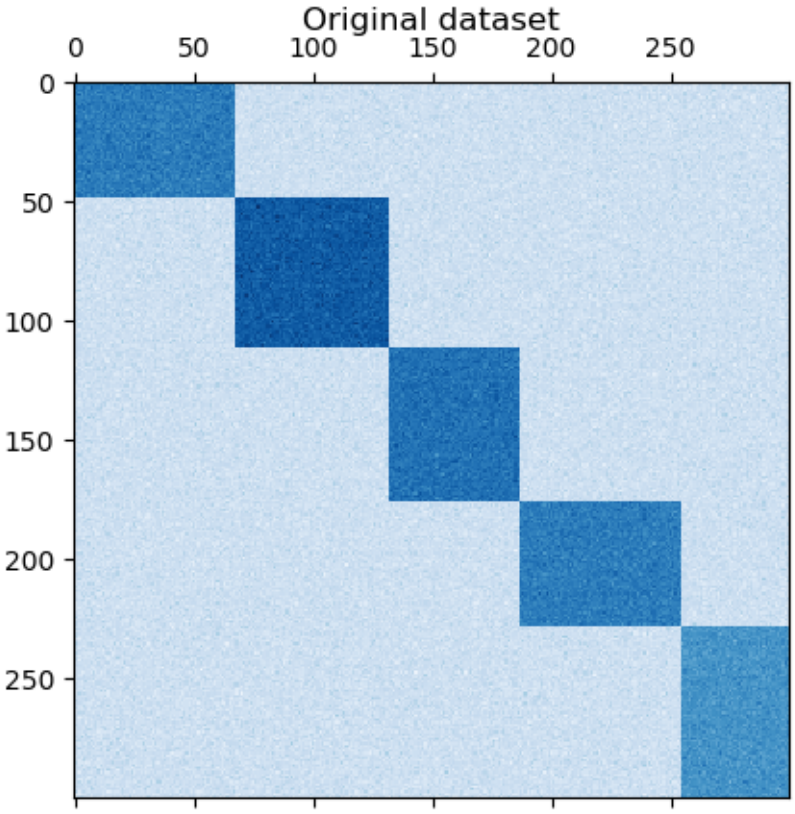
Examples concerning the `sklearn.cluster.bicluster` module.

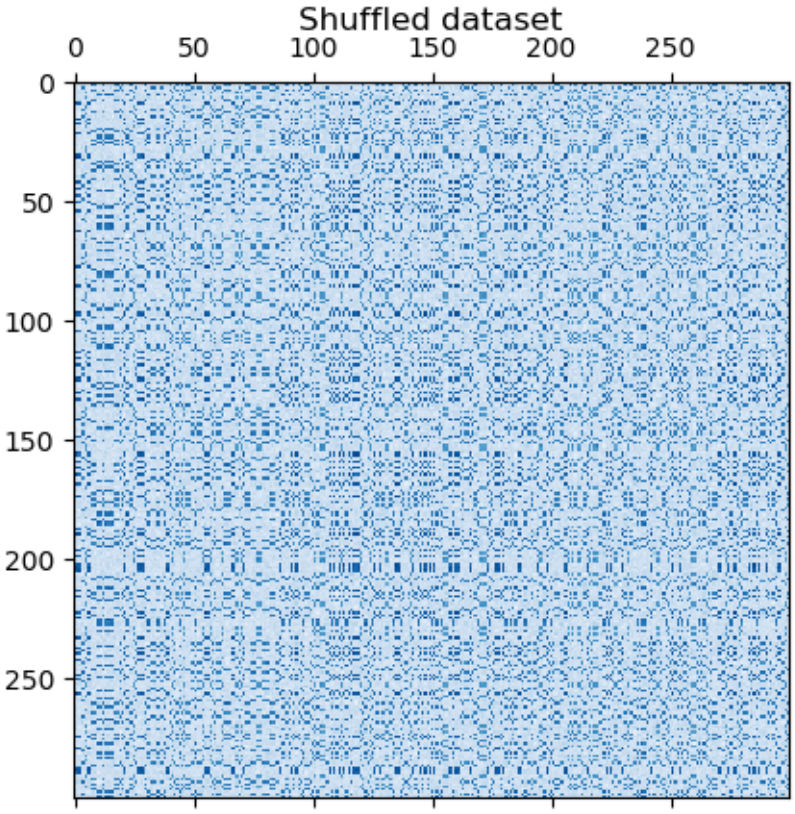
Note: Click *here* to download the full example code or to run this example in your browser via Binder

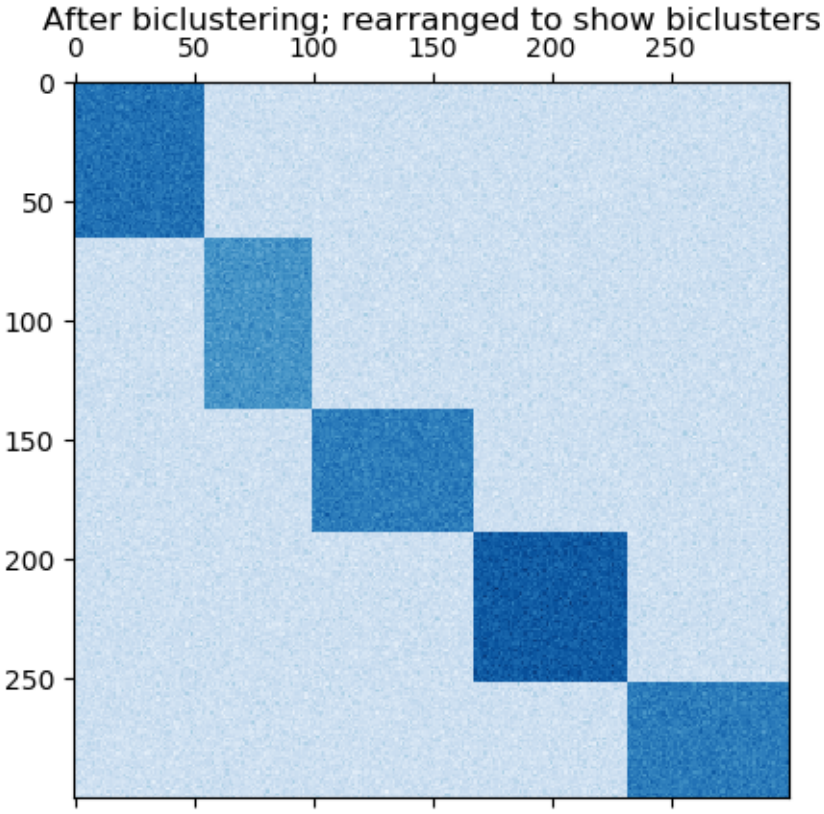
6.2.1 A demo of the Spectral Co-Clustering algorithm

This example demonstrates how to generate a dataset and bicluster it using the Spectral Co-Clustering algorithm.

The dataset is generated using the `make_biclusters` function, which creates a matrix of small values and implants bicluster with large values. The rows and columns are then shuffled and passed to the Spectral Co-Clustering algorithm. Rearranging the shuffled matrix to make biclusters contiguous shows how accurately the algorithm found the biclusters.


Original dataset
0
50
100
150
200
250
0
50
100
150
200
250


Shuffled dataset
0
50
100
150
200
250
0
50
100
150
200
250



-

Out:

```
consensus score: 1.000
```

```
print(__doc__)

# Author: Kemal Eren <kemal@kemaleren.com>
# License: BSD 3 clause

import numpy as np
from matplotlib import pyplot as plt

from sklearn.datasets import make_biclusters
from sklearn.cluster import SpectralCoclustering
from sklearn.metrics import consensus_score

data, rows, columns = make_biclusters(
    shape=(300, 300), n_clusters=5, noise=5,
    shuffle=False, random_state=0)
```

(continues on next page)

(continued from previous page)

```
plt.matshow(data, cmap=plt.cm.Blues)
plt.title("Original dataset")

# shuffle clusters
rng = np.random.RandomState(0)
row_idx = rng.permutation(data.shape[0])
col_idx = rng.permutation(data.shape[1])
data = data[row_idx][:, col_idx]

plt.matshow(data, cmap=plt.cm.Blues)
plt.title("Shuffled dataset")

model = SpectralCoclustering(n_clusters=5, random_state=0)
model.fit(data)
score = consensus_score(model.biclusters_,
                        (rows[:, row_idx], columns[:, col_idx]))

print("consensus score: {:.3f}".format(score))

fit_data = data[np.argsort(model.row_labels_)]
fit_data = fit_data[:, np.argsort(model.column_labels_)]

plt.matshow(fit_data, cmap=plt.cm.Blues)
plt.title("After biclustering; rearranged to show biclusters")

plt.show()
```

Total running time of the script: (0 minutes 0.531 seconds)

Estimated memory usage: 9 MB

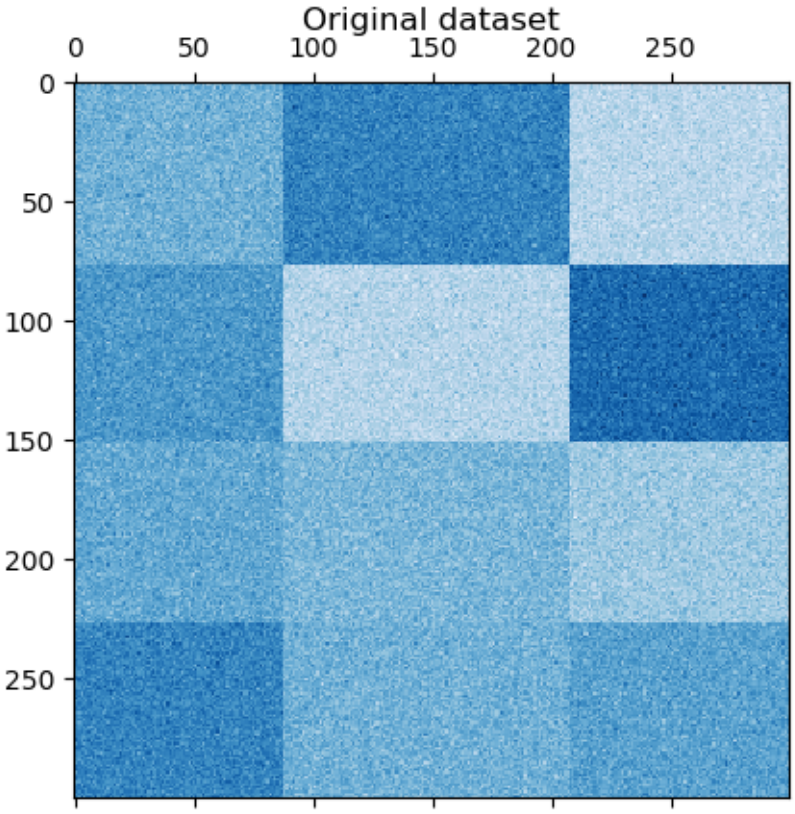
Note: Click *here* to download the full example code or to run this example in your browser via Binder

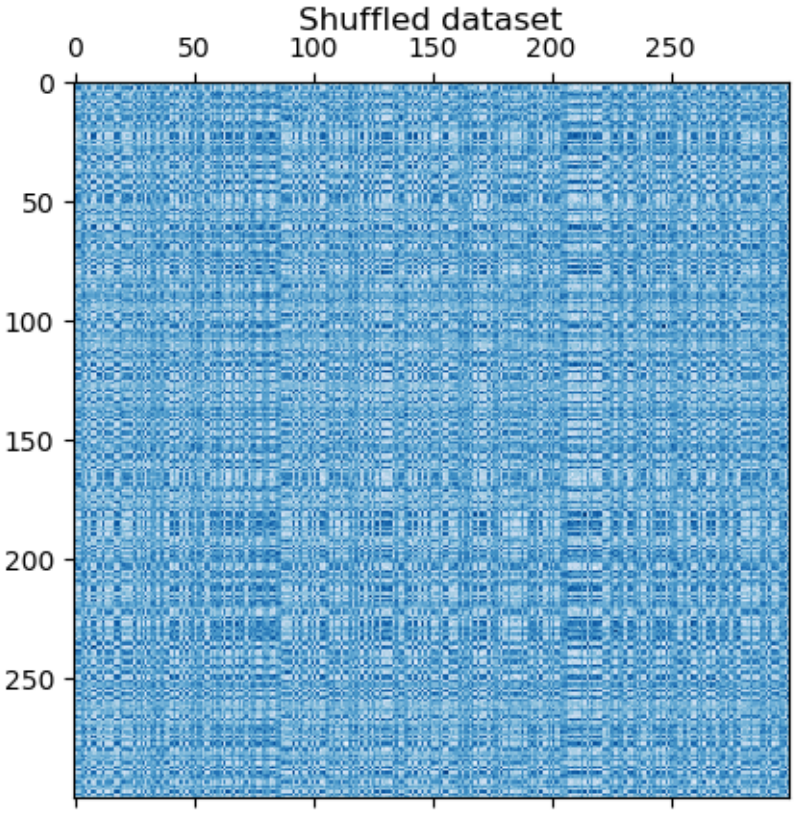
6.2.2 A demo of the Spectral Biclustering algorithm

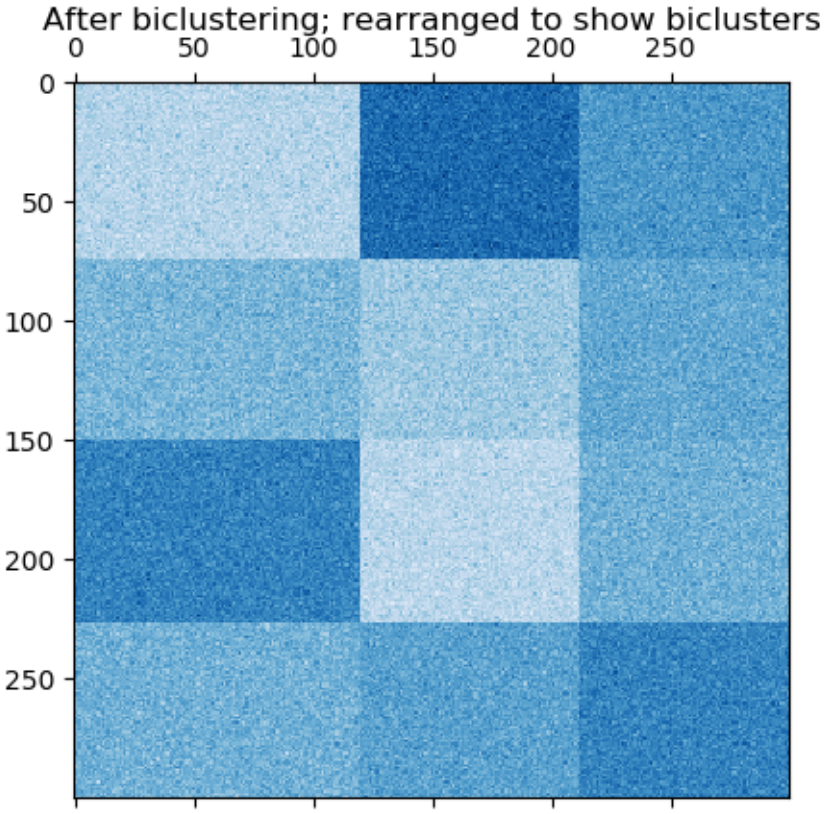
This example demonstrates how to generate a checkerboard dataset and bicluster it using the Spectral Biclustering algorithm.

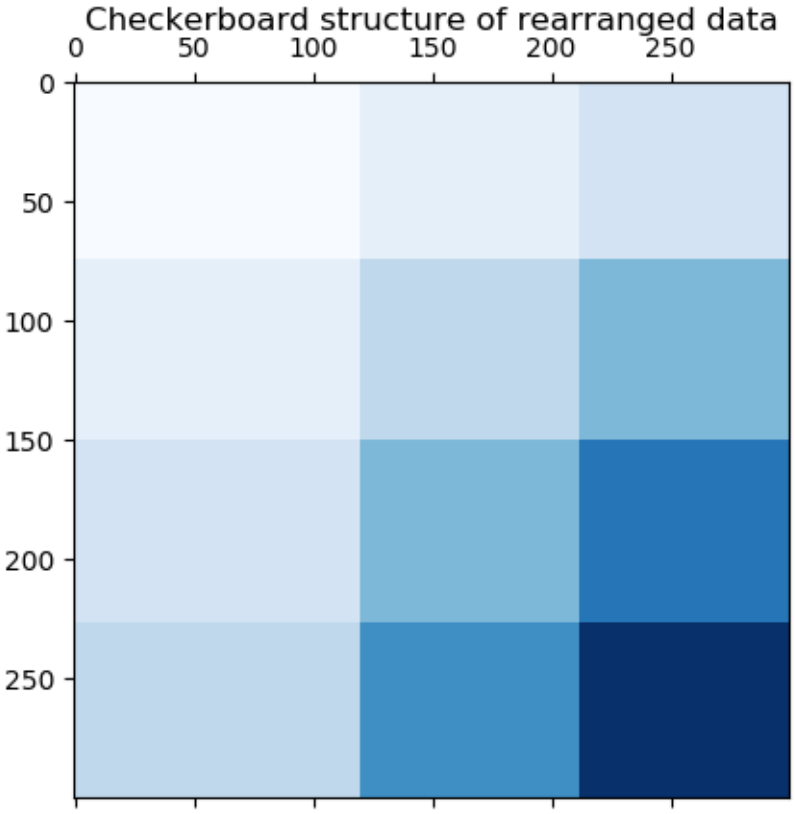
The data is generated with the `make_checkerboard` function, then shuffled and passed to the Spectral Biclustering algorithm. The rows and columns of the shuffled matrix are rearranged to show the biclusters found by the algorithm.

The outer product of the row and column label vectors shows a representation of the checkerboard structure.


Original dataset
0
50
100
150
200
250
0
50
100
150
200
250


Shuffled dataset
0
50
100
150
200
250
0
50
100
150
200
250


After biclustering; rearranged to show biclusters
0
50
100
150
200
250
0
50
100
150
200
250



-

Out:

```
consensus score: 1.0
```

```
print(__doc__)

# Author: Kemal Eren <kemal@kemaleren.com>
# License: BSD 3 clause

import numpy as np
from matplotlib import pyplot as plt

from sklearn.datasets import make_checkerboard
from sklearn.cluster import SpectralBiclustering
from sklearn.metrics import consensus_score


n_clusters = (4, 3)
data, rows, columns = make_checkerboard(
    shape=(300, 300), n_clusters=n_clusters, noise=10,
```

(continues on next page)

(continued from previous page)

```
    shuffle=False, random_state=0)

plt.matshow(data, cmap=plt.cm.Blues)
plt.title("Original dataset")

# shuffle clusters
rng = np.random.RandomState(0)
row_idx = rng.permutation(data.shape[0])
col_idx = rng.permutation(data.shape[1])
data = data[row_idx][:, col_idx]

plt.matshow(data, cmap=plt.cm.Blues)
plt.title("Shuffled dataset")

model = SpectralBiclustering(n_clusters=n_clusters, method='log',
                             random_state=0)
model.fit(data)
score = consensus_score(model.biclusters_,
                        (rows[:, row_idx], columns[:, col_idx]))

print("consensus score: {:.1f}".format(score))

fit_data = data[np.argsort(model.row_labels_)]
fit_data = fit_data[:, np.argsort(model.column_labels_)]

plt.matshow(fit_data, cmap=plt.cm.Blues)
plt.title("After biclustering; rearranged to show biclusters")

plt.matshow(np.outer(np.sort(model.row_labels_) + 1,
                     np.sort(model.column_labels_) + 1),
            cmap=plt.cm.Blues)
plt.title("Checkerboard structure of rearranged data")

plt.show()
```

Total running time of the script: (0 minutes 0.800 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.2.3 Biclustering documents with the Spectral Co-clustering algorithm

This example demonstrates the Spectral Co-clustering algorithm on the twenty newsgroups dataset. The 'comp.os.ms-windows.misc' category is excluded because it contains many posts containing nothing but data.

The TF-IDF vectorized posts form a word frequency matrix, which is then biclustered using Dhillon's Spectral Co-Clustering algorithm. The resulting document-word biclusters indicate subsets words used more often in those subsets documents.

For a few of the best biclusters, its most common document categories and its ten most important words get printed. The best biclusters are determined by their normalized cut. The best words are determined by comparing their sums inside and outside the bicluster.

For comparison, the documents are also clustered using MiniBatchKMeans. The document clusters derived from the biclusters achieve a better V-measure than clusters found by MiniBatchKMeans.

Out:

```
Vectorizing...
Coclustering...
Done in 2.75s. V-measure: 0.4387
MiniBatchKMeans...
Done in 5.69s. V-measure: 0.3344

Best biclusters:
----------------
bicluster 0 : 1829 documents, 2524 words
categories   : 22% comp.sys.ibm.pc.hardware, 19% comp.sys.mac.hardware, 18% comp.
↪graphics
words        : card, pc, ram, drive, bus, mac, motherboard, port, windows, floppy

bicluster 1 : 2391 documents, 3275 words
categories   : 18% rec.motorcycles, 17% rec.autos, 15% sci.electronics
words        : bike, engine, car, dod, bmw, honda, oil, motorcycle, behanna, ysu

bicluster 2 : 1887 documents, 4232 words
categories   : 23% talk.politics.guns, 19% talk.politics.misc, 13% sci.med
words        : gun, guns, firearms, geb, drugs, banks, dyer, amendment, clinton, cdt

bicluster 3 : 1146 documents, 3263 words
categories   : 29% talk.politics.mideast, 26% soc.religion.christian, 25% alt.atheism
words        : god, jesus, christians, atheists, kent, sin, morality, belief,␣
↪resurrection, marriage

bicluster 4 : 1732 documents, 3967 words
categories   : 26% sci.crypt, 23% sci.space, 17% sci.med
words        : clipper, encryption, key, escrow, nsa, crypto, keys, intercon, secure,␣
↪wiretap
```

```
from collections import defaultdict
import operator
from time import time

import numpy as np

from sklearn.cluster import SpectralCoclustering
from sklearn.cluster import MiniBatchKMeans
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.cluster import v_measure_score

print(__doc__)


def number_normalizer(tokens):
    """ Map all numeric tokens to a placeholder.
```

(continues on next page)

(continued from previous page)

```
    For many applications, tokens that begin with a number are not directly
    useful, but the fact that such a token exists can be relevant.  By applying
    this form of dimensionality reduction, some methods may perform better.
    """
    return ("#NUMBER" if token[0].isdigit() else token for token in tokens)


class NumberNormalizingVectorizer(TfidfVectorizer):
    def build_tokenizer(self):
        tokenize = super().build_tokenizer()
        return lambda doc: list(number_normalizer(tokenize(doc)))


# exclude 'comp.os.ms-windows.misc'
categories = ['alt.atheism', 'comp.graphics',
              'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware',
              'comp.windows.x', 'misc.forsale', 'rec.autos',
              'rec.motorcycles', 'rec.sport.baseball',
              'rec.sport.hockey', 'sci.crypt', 'sci.electronics',
              'sci.med', 'sci.space', 'soc.religion.christian',
              'talk.politics.guns', 'talk.politics.mideast',
              'talk.politics.misc', 'talk.religion.misc']
newsgroups = fetch_20newsgroups(categories=categories)
y_true = newsgroups.target

vectorizer = NumberNormalizingVectorizer(stop_words='english', min_df=5)
cocluster = SpectralCoclustering(n_clusters=len(categories),
                                 svd_method='arpack', random_state=0)
kmeans = MiniBatchKMeans(n_clusters=len(categories), batch_size=20000,
                         random_state=0)

print("Vectorizing...")
X = vectorizer.fit_transform(newsgroups.data)

print("Coclustering...")
start_time = time()
cocluster.fit(X)
y_cocluster = cocluster.row_labels_
print("Done in {:.2f}s. V-measure: {:.4f}".format(
    time() - start_time,
    v_measure_score(y_cocluster, y_true)))

print("MiniBatchKMeans...")
start_time = time()
y_kmeans = kmeans.fit_predict(X)
print("Done in {:.2f}s. V-measure: {:.4f}".format(
    time() - start_time,
    v_measure_score(y_kmeans, y_true)))

feature_names = vectorizer.get_feature_names()
document_names = list(newsgroups.target_names[i] for i in newsgroups.target)


def bicluster_ncut(i):
    rows, cols = cocluster.get_indices(i)
    if not (np.any(rows) and np.any(cols)):
```

(continues on next page)

(continued from previous page)

```
        import sys
        return sys.float_info.max
    row_complement = np.nonzero(np.logical_not(cocluster.rows_[i]))[0]
    col_complement = np.nonzero(np.logical_not(cocluster.columns_[i]))[0]
    # Note: the following is identical to X[rows[:, np.newaxis],
    # cols].sum() but much faster in scipy <= 0.16
    weight = X[rows][:, cols].sum()
    cut = (X[row_complement][:, cols].sum() +
           X[rows][:, col_complement].sum())
    return cut / weight


def most_common(d):
    """Items of a defaultdict(int) with the highest values.

    Like Counter.most_common in Python >=2.7.
    """
    return sorted(d.items(), key=operator.itemgetter(1), reverse=True)


bicluster_ncuts = list(bicluster_ncut(i)
                       for i in range(len(newsgroups.target_names)))
best_idx = np.argsort(bicluster_ncuts)[:5]

print()
print("Best biclusters:")
print("----------------")
for idx, cluster in enumerate(best_idx):
    n_rows, n_cols = cocluster.get_shape(cluster)
    cluster_docs, cluster_words = cocluster.get_indices(cluster)
    if not len(cluster_docs) or not len(cluster_words):
        continue

    # categories
    counter = defaultdict(int)
    for i in cluster_docs:
        counter[document_names[i]] += 1
    cat_string = ", ".join("{:.0f}% {}".format(float(c) / n_rows * 100, name)
                           for name, c in most_common(counter)[:3])

    # words
    out_of_cluster_docs = cocluster.row_labels_ != cluster
    out_of_cluster_docs = np.where(out_of_cluster_docs)[0]
    word_col = X[:, cluster_words]
    word_scores = np.array(word_col[cluster_docs, :].sum(axis=0) -
                           word_col[out_of_cluster_docs, :].sum(axis=0))
    word_scores = word_scores.ravel()
    important_words = list(feature_names[cluster_words[i]]
                           for i in word_scores.argsort()[:-11:-1])

    print("bicluster {} : {} documents, {} words".format(
        idx, n_rows, n_cols))
    print("categories   : {}".format(cat_string))
    print("words        : {}\n".format(', '.join(important_words)))
```

Total running time of the script: (0 minutes 11.526 seconds)

Estimated memory usage: 72 MB

6.3 Calibration

Examples illustrating the calibration of predicted probabilities of classifiers.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.3.1 Comparison of Calibration of Classifiers

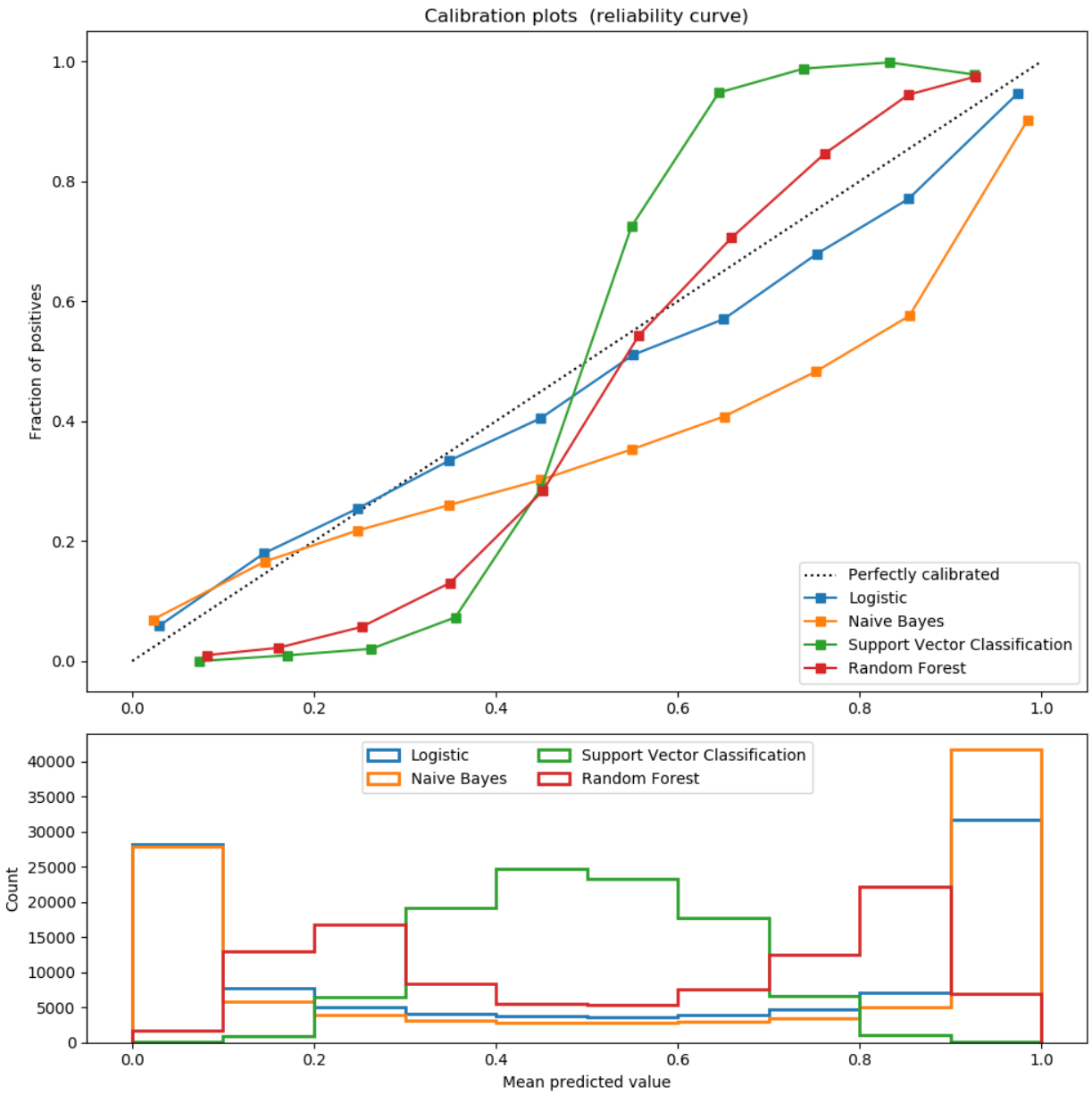
Well calibrated classifiers are probabilistic classifiers for which the output of the predict_proba method can be directly interpreted as a confidence level. For instance a well calibrated (binary) classifier should classify the samples such that among the samples to which it gave a predict_proba value close to 0.8, approx. 80% actually belong to the positive class.

LogisticRegression returns well calibrated predictions as it directly optimizes log-loss. In contrast, the other methods return biased probabilities, with different biases per method:

- GaussianNaiveBayes tends to push probabilities to 0 or 1 (note the counts in the histograms). This is mainly because it makes the assumption that features are conditionally independent given the class, which is not the case in this dataset which contains 2 redundant features.
- RandomForestClassifier shows the opposite behavior: the histograms show peaks at approx. 0.2 and 0.9 probability, while probabilities close to 0 or 1 are very rare. An explanation for this is given by Niculescu-Mizil and Caruana[1]: "Methods such as bagging and random forests that average predictions from a base set of models can have difficulty making predictions near 0 and 1 because variance in the underlying base models will bias predictions that should be near zero or one away from these values. Because predictions are restricted to the interval [0,1], errors caused by variance tend to be one- sided near zero and one. For example, if a model should predict p = 0 for a case, the only way bagging can achieve this is if all bagged trees predict zero. If we add noise to the trees that bagging is averaging over, this noise will cause some trees to predict values larger than 0 for this case, thus moving the average prediction of the bagged ensemble away from 0. We observe this effect most strongly with random forests because the base-level trees trained with random forests have relatively high variance due to feature subsetting." As a result, the calibration curve shows a characteristic sigmoid shape, indicating that the classifier could trust its "intuition" more and return probabilities closer to 0 or 1 typically.
- Support Vector Classification (SVC) shows an even more sigmoid curve as the RandomForestClassifier, which is typical for maximum-margin methods (compare Niculescu-Mizil and Caruana[1]), which focus on hard samples that are close to the decision boundary (the support vectors).

References:

[1] Predicting Good Probabilities with Supervised Learning, A. Niculescu-Mizil & R. Caruana, ICML 2005




```
print(__doc__)

# Author: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD Style.

import numpy as np
np.random.seed(0)

import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
```

(continues on next page)

(continued from previous page)

```
from sklearn.svm import LinearSVC
from sklearn.calibration import calibration_curve

X, y = datasets.make_classification(n_samples=100000, n_features=20,
                                    n_informative=2, n_redundant=2)

train_samples = 100  # Samples used for training the models

X_train = X[:train_samples]
X_test = X[train_samples:]
y_train = y[:train_samples]
y_test = y[train_samples:]

# Create classifiers
lr = LogisticRegression()
gnb = GaussianNB()
svc = LinearSVC(C=1.0)
rfc = RandomForestClassifier()


# #############################################################################
# Plot calibration plots

plt.figure(figsize=(10, 10))
ax1 = plt.subplot2grid((3, 1), (0, 0), rowspan=2)
ax2 = plt.subplot2grid((3, 1), (2, 0))

ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")
for clf, name in [(lr, 'Logistic'),
                  (gnb, 'Naive Bayes'),
                  (svc, 'Support Vector Classification'),
                  (rfc, 'Random Forest')]:
    clf.fit(X_train, y_train)
    if hasattr(clf, "predict_proba"):
        prob_pos = clf.predict_proba(X_test)[:, 1]
    else:  # use decision function
        prob_pos = clf.decision_function(X_test)
        prob_pos = \
            (prob_pos - prob_pos.min()) / (prob_pos.max() - prob_pos.min())
    fraction_of_positives, mean_predicted_value = \
        calibration_curve(y_test, prob_pos, n_bins=10)

    ax1.plot(mean_predicted_value, fraction_of_positives, "s-",
             label="%s" % (name, ))

    ax2.hist(prob_pos, range=(0, 1), bins=10, label=name,
             histtype="step", lw=2)

ax1.set_ylabel("Fraction of positives")
ax1.set_ylim([-0.05, 1.05])
ax1.legend(loc="lower right")
ax1.set_title('Calibration plots  (reliability curve)')

ax2.set_xlabel("Mean predicted value")
ax2.set_ylabel("Count")
ax2.legend(loc="upper center", ncol=2)
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 1.301 seconds)

Estimated memory usage: 54 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.3.2 Probability Calibration curves

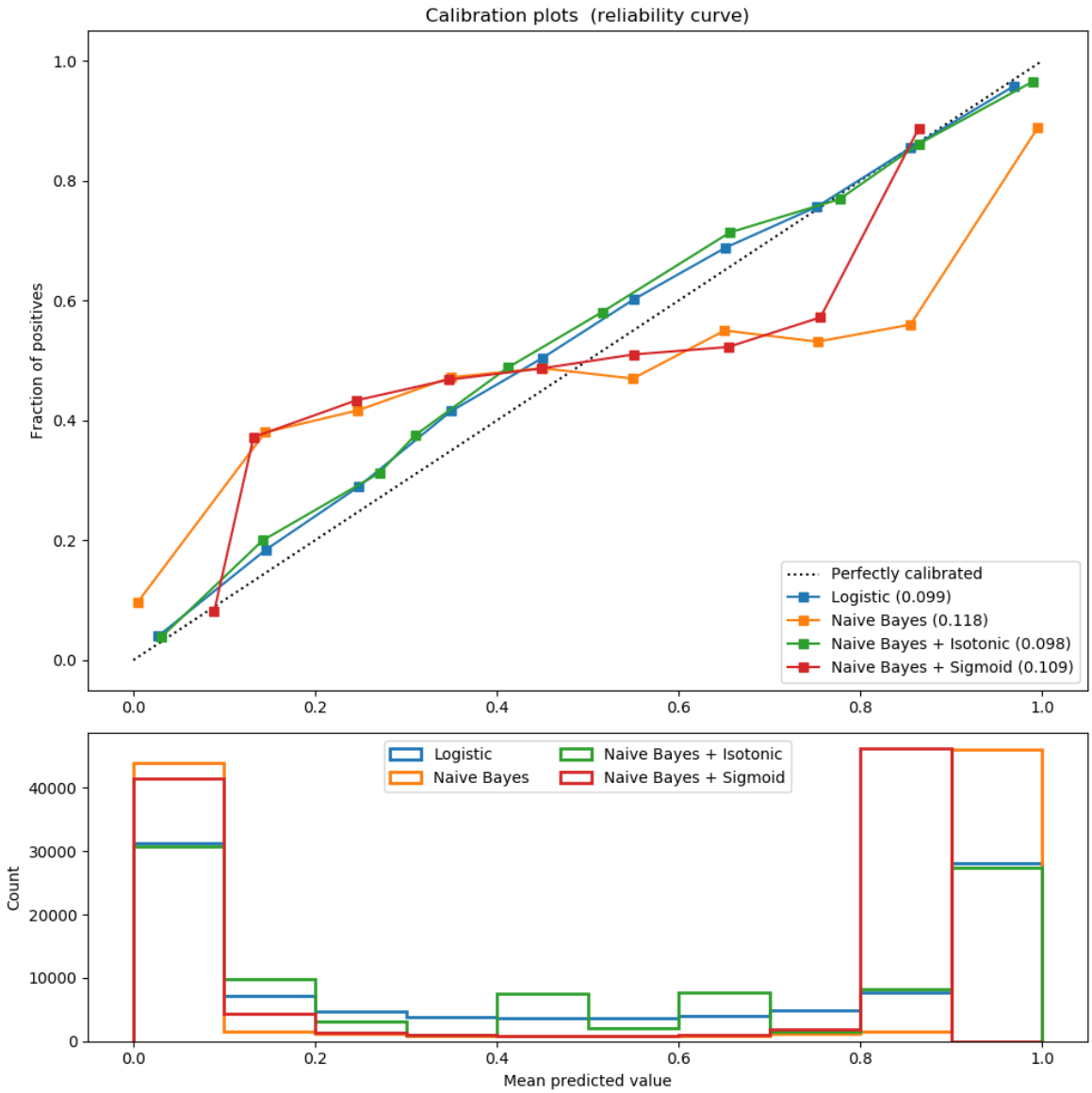
When performing classification one often wants to predict not only the class label, but also the associated probability. This probability gives some kind of confidence on the prediction. This example demonstrates how to display how well calibrated the predicted probabilities are and how to calibrate an uncalibrated classifier.

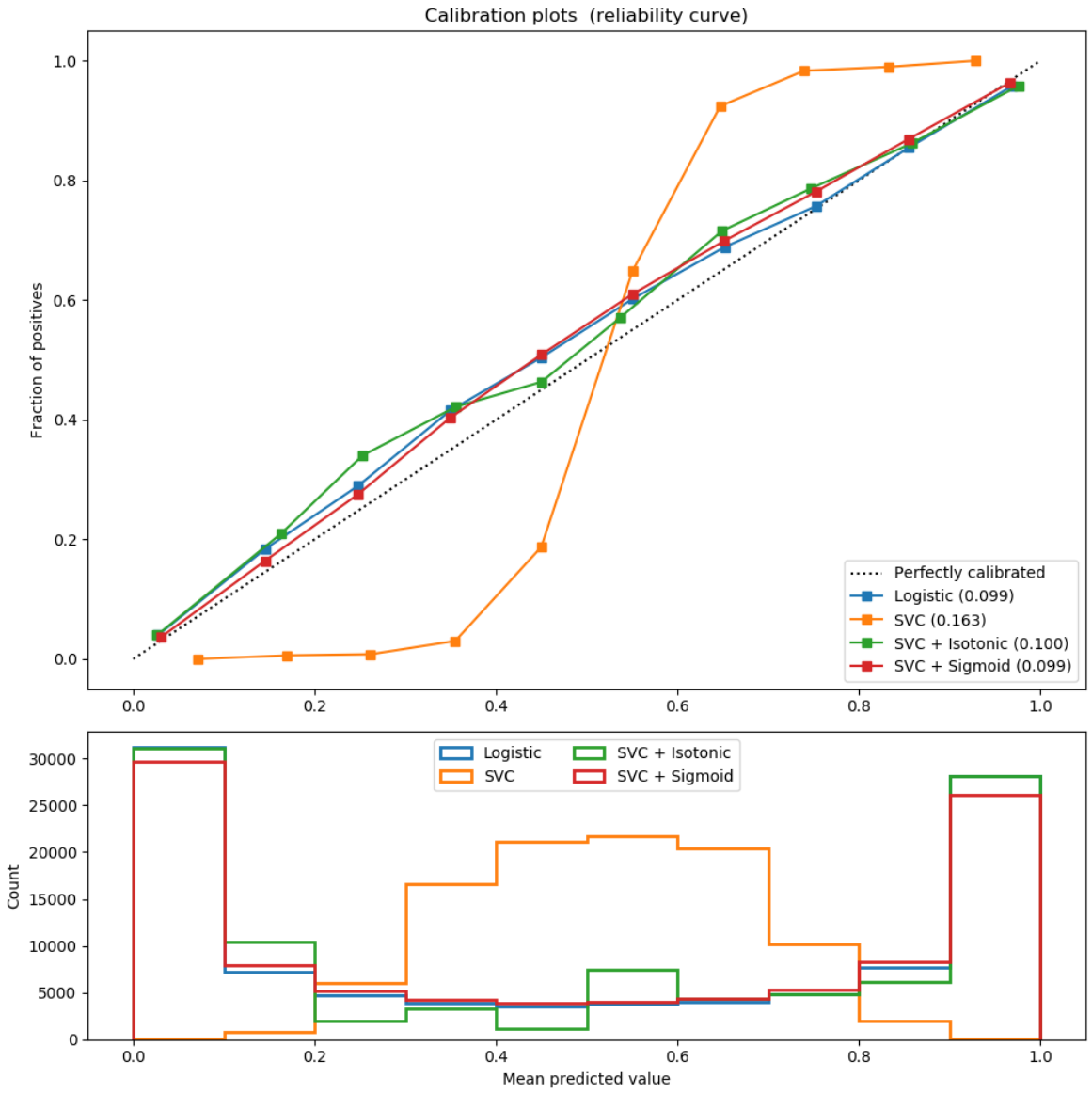
The experiment is performed on an artificial dataset for binary classification with 100,000 samples (1,000 of them are used for model fitting) with 20 features. Of the 20 features, only 2 are informative and 10 are redundant. The first figure shows the estimated probabilities obtained with logistic regression, Gaussian naive Bayes, and Gaussian naive Bayes with both isotonic calibration and sigmoid calibration. The calibration performance is evaluated with Brier score, reported in the legend (the smaller the better). One can observe here that logistic regression is well calibrated while raw Gaussian naive Bayes performs very badly. This is because of the redundant features which violate the assumption of feature-independence and result in an overly confident classifier, which is indicated by the typical transposed-sigmoid curve.

Calibration of the probabilities of Gaussian naive Bayes with isotonic regression can fix this issue as can be seen from the nearly diagonal calibration curve. Sigmoid calibration also improves the brier score slightly, albeit not as strongly as the non-parametric isotonic regression. This can be attributed to the fact that we have plenty of calibration data such that the greater flexibility of the non-parametric model can be exploited.

The second figure shows the calibration curve of a linear support-vector classifier (LinearSVC). LinearSVC shows the opposite behavior as Gaussian naive Bayes: the calibration curve has a sigmoid curve, which is typical for an under-confident classifier. In the case of LinearSVC, this is caused by the margin property of the hinge loss, which lets the model focus on hard samples that are close to the decision boundary (the support vectors).

Both kinds of calibration can fix this issue and yield nearly identical results. This shows that sigmoid calibration can deal with situations where the calibration curve of the base classifier is sigmoid (e.g., for LinearSVC) but not where it is transposed-sigmoid (e.g., Gaussian naive Bayes).


Calibration plots (reliability curve)
Fraction of positives
Perfectly calibrated
Logistic (0.099)
Naive Bayes (0.118)
Naive Bayes + Isotonic (0.098)
Naive Bayes + Sigmoid (0.109)
Logistic
Naive Bayes
Naive Bayes + Isotonic
Naive Bayes + Sigmoid
Count
Mean predicted value



-

Out:

```
Logistic:
        Brier: 0.099
        Precision: 0.872
        Recall: 0.851
        F1: 0.862

Naive Bayes:
        Brier: 0.118
        Precision: 0.857
        Recall: 0.876
        F1: 0.867

Naive Bayes + Isotonic:
        Brier: 0.098
        Precision: 0.883
```

(continues on next page)

(continued from previous page)

```
        Recall: 0.836
        F1: 0.859

Naive Bayes + Sigmoid:
        Brier: 0.109
        Precision: 0.861
        Recall: 0.871
        F1: 0.866

Logistic:
        Brier: 0.099
        Precision: 0.872
        Recall: 0.851
        F1: 0.862

SVC:
        Brier: 0.163
        Precision: 0.872
        Recall: 0.852
        F1: 0.862

SVC + Isotonic:
        Brier: 0.100
        Precision: 0.853
        Recall: 0.878
        F1: 0.865

SVC + Sigmoid:
        Brier: 0.099
        Precision: 0.874
        Recall: 0.849
        F1: 0.861
```

```
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@telecom-paristech.fr>
#         Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD Style.

import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import LinearSVC
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (brier_score_loss, precision_score, recall_score,
                             f1_score)
from sklearn.calibration import CalibratedClassifierCV, calibration_curve
from sklearn.model_selection import train_test_split
```

(continues on next page)

(continued from previous page)

```
# Create dataset of classification task with many redundant and few
# informative features
X, y = datasets.make_classification(n_samples=100000, n_features=20,
                                    n_informative=2, n_redundant=10,
                                    random_state=42)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.99,
                                                    random_state=42)


def plot_calibration_curve(est, name, fig_index):
    """Plot calibration curve for est w/o and with calibration. """
    # Calibrated with isotonic calibration
    isotonic = CalibratedClassifierCV(est, cv=2, method='isotonic')

    # Calibrated with sigmoid calibration
    sigmoid = CalibratedClassifierCV(est, cv=2, method='sigmoid')

    # Logistic regression with no calibration as baseline
    lr = LogisticRegression(C=1.)

    fig = plt.figure(fig_index, figsize=(10, 10))
    ax1 = plt.subplot2grid((3, 1), (0, 0), rowspan=2)
    ax2 = plt.subplot2grid((3, 1), (2, 0))

    ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")
    for clf, name in [(lr, 'Logistic'),
                      (est, name),
                      (isotonic, name + ' + Isotonic'),
                      (sigmoid, name + ' + Sigmoid')]:
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        if hasattr(clf, "predict_proba"):
            prob_pos = clf.predict_proba(X_test)[:, 1]
        else:  # use decision function
            prob_pos = clf.decision_function(X_test)
            prob_pos = \
                (prob_pos - prob_pos.min()) / (prob_pos.max() - prob_pos.min())

        clf_score = brier_score_loss(y_test, prob_pos, pos_label=y.max())
        print("%s:" % name)
        print("\tBrier: %1.3f" % (clf_score))
        print("\tPrecision: %1.3f" % precision_score(y_test, y_pred))
        print("\tRecall: %1.3f" % recall_score(y_test, y_pred))
        print("\tF1: %1.3f\n" % f1_score(y_test, y_pred))

        fraction_of_positives, mean_predicted_value = \
            calibration_curve(y_test, prob_pos, n_bins=10)

        ax1.plot(mean_predicted_value, fraction_of_positives, "s-",
                 label="%s (%1.3f)" % (name, clf_score))

        ax2.hist(prob_pos, range=(0, 1), bins=10, label=name,
                 histtype="step", lw=2)

    ax1.set_ylabel("Fraction of positives")
    ax1.set_ylim([-0.05, 1.05])
```

(continues on next page)

(continued from previous page)

```
    ax1.legend(loc="lower right")
    ax1.set_title('Calibration plots  (reliability curve)')

    ax2.set_xlabel("Mean predicted value")
    ax2.set_ylabel("Count")
    ax2.legend(loc="upper center", ncol=2)

    plt.tight_layout()

# Plot calibration curve for Gaussian Naive Bayes
plot_calibration_curve(GaussianNB(), "Naive Bayes", 1)

# Plot calibration curve for Linear SVC
plot_calibration_curve(LinearSVC(max_iter=10000), "SVC", 2)

plt.show()
```

Total running time of the script: (0 minutes 2.474 seconds)

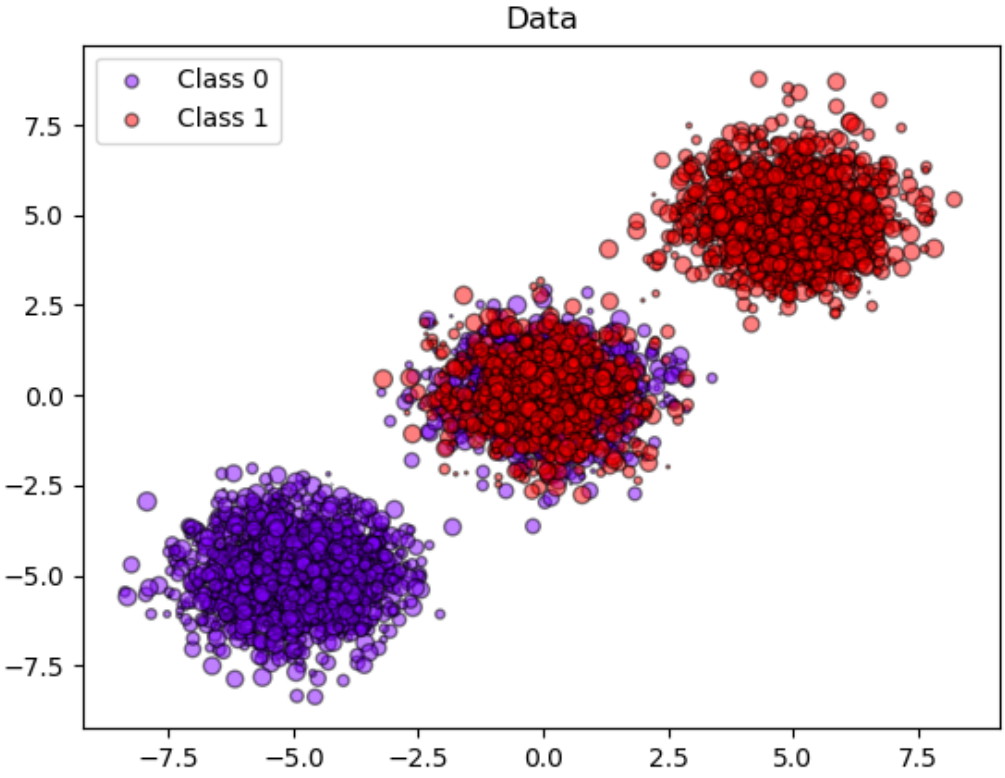
Estimated memory usage: 28 MB

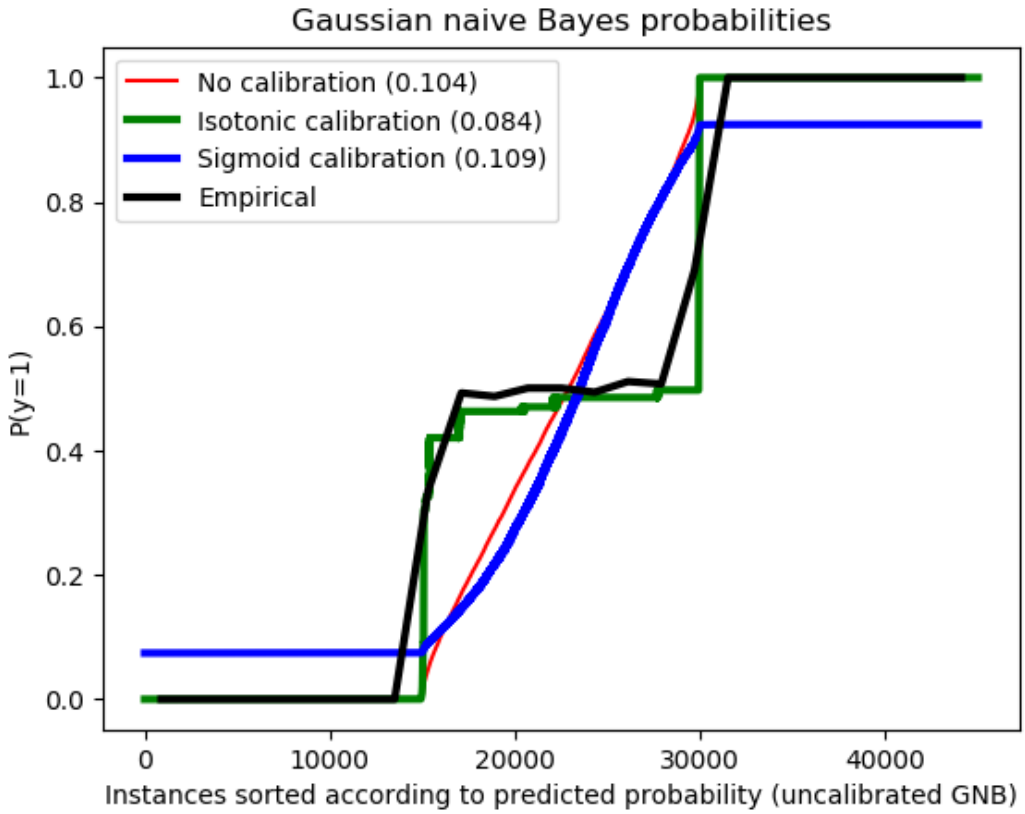
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.3.3 Probability calibration of classifiers

When performing classification you often want to predict not only the class label, but also the associated probability. This probability gives you some kind of confidence on the prediction. However, not all classifiers provide well-calibrated probabilities, some being over-confident while others being under-confident. Thus, a separate calibration of predicted probabilities is often desirable as a postprocessing. This example illustrates two different methods for this calibration and evaluates the quality of the returned probabilities using Brier's score (see https://en.wikipedia.org/wiki/Brier_score).

Compared are the estimated probability using a Gaussian naive Bayes classifier without calibration, with a sigmoid calibration, and with a non-parametric isotonic calibration. One can observe that only the non-parametric model is able to provide a probability calibration that returns probabilities close to the expected 0.5 for most of the samples belonging to the middle cluster with heterogeneous labels. This results in a significantly improved Brier score.


Data
Class 0
Class 1
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5



Out:

```
Brier scores: (the smaller the better)
No calibration: 0.104
With isotonic calibration: 0.084
With sigmoid calibration: 0.109
```

```
print(__doc__)

# Author: Mathieu Blondel <mathieu@mblondel.org>
#         Alexandre Gramfort <alexandre.gramfort@telecom-paristech.fr>
#         Balazs Kegl <balazs.kegl@gmail.com>
#         Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD Style.

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm

from sklearn.datasets import make_blobs
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import brier_score_loss
```

(continues on next page)

(continued from previous page)

```
from sklearn.calibration import CalibratedClassifierCV
from sklearn.model_selection import train_test_split


n_samples = 50000
n_bins = 3  # use 3 bins for calibration_curve as we have 3 clusters here

# Generate 3 blobs with 2 classes where the second blob contains
# half positive samples and half negative samples. Probability in this
# blob is therefore 0.5.
centers = [(-5, -5), (0, 0), (5, 5)]
X, y = make_blobs(n_samples=n_samples, centers=centers, shuffle=False,
                  random_state=42)

y[:n_samples // 2] = 0
y[n_samples // 2:] = 1
sample_weight = np.random.RandomState(42).rand(y.shape[0])

# split train, test for calibration
X_train, X_test, y_train, y_test, sw_train, sw_test = \
    train_test_split(X, y, sample_weight, test_size=0.9, random_state=42)

# Gaussian Naive-Bayes with no calibration
clf = GaussianNB()
clf.fit(X_train, y_train)  # GaussianNB itself does not support sample-weights
prob_pos_clf = clf.predict_proba(X_test)[:, 1]

# Gaussian Naive-Bayes with isotonic calibration
clf_isotonic = CalibratedClassifierCV(clf, cv=2, method='isotonic')
clf_isotonic.fit(X_train, y_train, sw_train)
prob_pos_isotonic = clf_isotonic.predict_proba(X_test)[:, 1]

# Gaussian Naive-Bayes with sigmoid calibration
clf_sigmoid = CalibratedClassifierCV(clf, cv=2, method='sigmoid')
clf_sigmoid.fit(X_train, y_train, sw_train)
prob_pos_sigmoid = clf_sigmoid.predict_proba(X_test)[:, 1]

print("Brier scores: (the smaller the better)")

clf_score = brier_score_loss(y_test, prob_pos_clf, sw_test)
print("No calibration: %1.3f" % clf_score)

clf_isotonic_score = brier_score_loss(y_test, prob_pos_isotonic, sw_test)
print("With isotonic calibration: %1.3f" % clf_isotonic_score)

clf_sigmoid_score = brier_score_loss(y_test, prob_pos_sigmoid, sw_test)
print("With sigmoid calibration: %1.3f" % clf_sigmoid_score)

# ##############################################################################
# Plot the data and the predicted probabilities
plt.figure()
y_unique = np.unique(y)
colors = cm.rainbow(np.linspace(0.0, 1.0, y_unique.size))
for this_y, color in zip(y_unique, colors):
    this_X = X_train[y_train == this_y]
    this_sw = sw_train[y_train == this_y]
    plt.scatter(this_X[:, 0], this_X[:, 1], s=this_sw * 50,
```

(continues on next page)

(continued from previous page)

```
                c=color[np.newaxis, :],
                alpha=0.5, edgecolor='k',
                label="Class %s" % this_y)
plt.legend(loc="best")
plt.title("Data")

plt.figure()
order = np.lexsort((prob_pos_clf, ))
plt.plot(prob_pos_clf[order], 'r', label='No calibration (%1.3f)' % clf_score)
plt.plot(prob_pos_isotonic[order], 'g', linewidth=3,
         label='Isotonic calibration (%1.3f)' % clf_isotonic_score)
plt.plot(prob_pos_sigmoid[order], 'b', linewidth=3,
         label='Sigmoid calibration (%1.3f)' % clf_sigmoid_score)
plt.plot(np.linspace(0, y_test.size, 51)[1::2],
         y_test[order].reshape(25, -1).mean(1),
         'k', linewidth=3, label=r'Empirical')
plt.ylim([-0.05, 1.05])
plt.xlabel("Instances sorted according to predicted probability "
           "(uncalibrated GNB)")
plt.ylabel("P(y=1)")
plt.legend(loc="upper left")
plt.title("Gaussian naive Bayes probabilities")

plt.show()
```

Total running time of the script: (0 minutes 0.617 seconds)

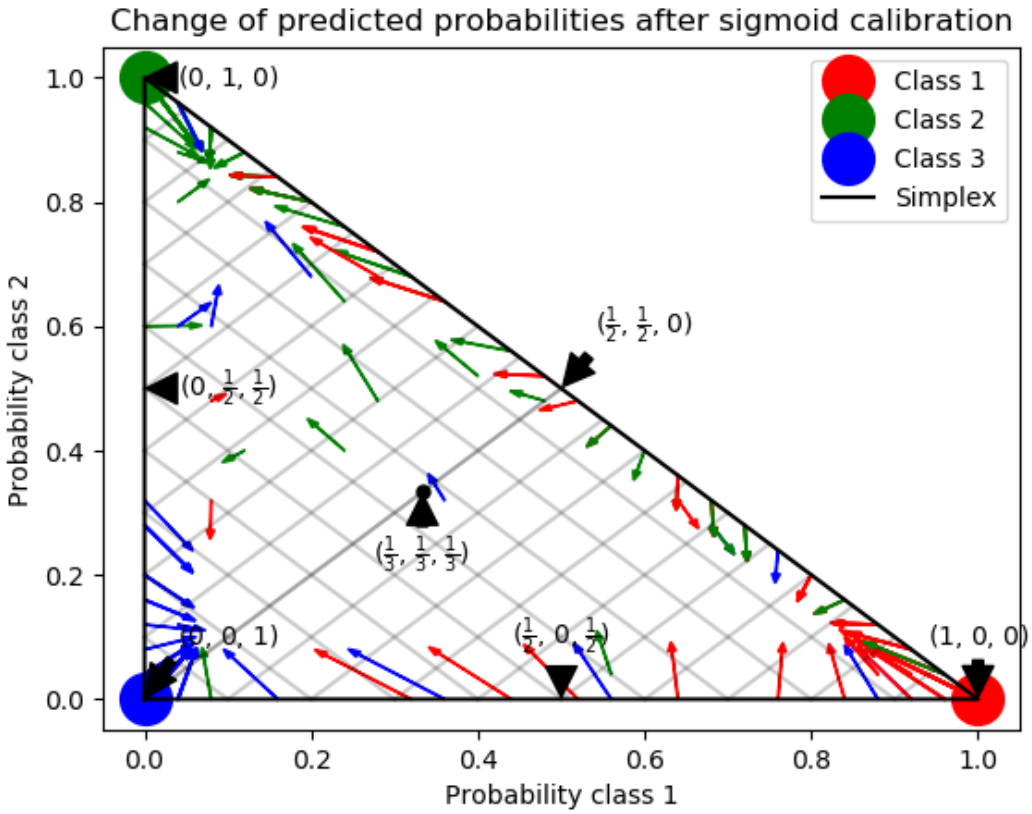
Estimated memory usage: 10 MB

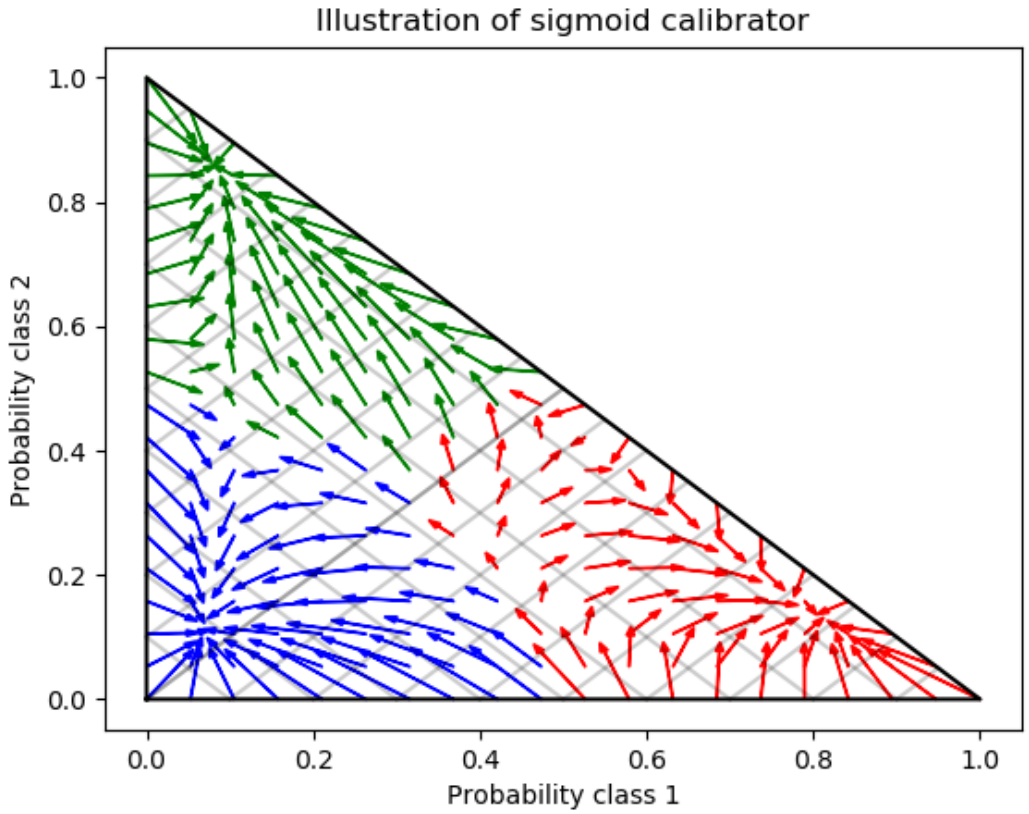
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.3.4 Probability Calibration for 3-class classification

This example illustrates how sigmoid calibration changes predicted probabilities for a 3-class classification problem. Illustrated is the standard 2-simplex, where the three corners correspond to the three classes. Arrows point from the probability vectors predicted by an uncalibrated classifier to the probability vectors predicted by the same classifier after sigmoid calibration on a hold-out validation set. Colors indicate the true class of an instance (red: class 1, green: class 2, blue: class 3).

The base classifier is a random forest classifier with 25 base estimators (trees). If this classifier is trained on all 800 training datapoints, it is overly confident in its predictions and thus incurs a large log-loss. Calibrating an identical classifier, which was trained on 600 datapoints, with method='sigmoid' on the remaining 200 datapoints reduces the confidence of the predictions, i.e., moves the probability vectors from the edges of the simplex towards the center. This calibration results in a lower log-loss. Note that an alternative would have been to increase the number of base estimators which would have resulted in a similar decrease in log-loss.


Change of predicted probabilities after sigmoid calibration
Class 1
Class 2
Class 3
Simplex
(0, 1, 0)
(1/2, 1/2, 0)
(0, 1/2, 1/2)
(1/3, 1/3, 1/3)
(0, 0, 1)
(1/2, 0, 1/2)
(1, 0, 0)
Probability class 2
Probability class 1
0.0
0.2
0.4
0.6
0.8
1.0



Out:

```
Log-loss of
 * uncalibrated classifier trained on 800 datapoints: 1.280
 * classifier trained on 600 datapoints and calibrated on 200 datapoint: 0.534
```

```
print(__doc__)

# Author: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD Style.


import matplotlib.pyplot as plt

import numpy as np

from sklearn.datasets import make_blobs
from sklearn.ensemble import RandomForestClassifier
from sklearn.calibration import CalibratedClassifierCV
from sklearn.metrics import log_loss

np.random.seed(0)
```

(continues on next page)

(continued from previous page)

```
# Generate data
X, y = make_blobs(n_samples=1000, random_state=42, cluster_std=5.0)
X_train, y_train = X[:600], y[:600]
X_valid, y_valid = X[600:800], y[600:800]
X_train_valid, y_train_valid = X[:800], y[:800]
X_test, y_test = X[800:], y[800:]

# Train uncalibrated random forest classifier on whole train and validation
# data and evaluate on test data
clf = RandomForestClassifier(n_estimators=25)
clf.fit(X_train_valid, y_train_valid)
clf_probs = clf.predict_proba(X_test)
score = log_loss(y_test, clf_probs)

# Train random forest classifier, calibrate on validation data and evaluate
# on test data
clf = RandomForestClassifier(n_estimators=25)
clf.fit(X_train, y_train)
clf_probs = clf.predict_proba(X_test)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid", cv="prefit")
sig_clf.fit(X_valid, y_valid)
sig_clf_probs = sig_clf.predict_proba(X_test)
sig_score = log_loss(y_test, sig_clf_probs)

# Plot changes in predicted probabilities via arrows
plt.figure()
colors = ["r", "g", "b"]
for i in range(clf_probs.shape[0]):
    plt.arrow(clf_probs[i, 0], clf_probs[i, 1],
              sig_clf_probs[i, 0] - clf_probs[i, 0],
              sig_clf_probs[i, 1] - clf_probs[i, 1],
              color=colors[y_test[i]], head_width=1e-2)

# Plot perfect predictions
plt.plot([1.0], [0.0], 'ro', ms=20, label="Class 1")
plt.plot([0.0], [1.0], 'go', ms=20, label="Class 2")
plt.plot([0.0], [0.0], 'bo', ms=20, label="Class 3")

# Plot boundaries of unit simplex
plt.plot([0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0], 'k', label="Simplex")

# Annotate points on the simplex
plt.annotate(r'($\frac{1}{3}$, $\frac{1}{3}$, $\frac{1}{3}$)',
             xy=(1.0/3, 1.0/3), xytext=(1.0/3, .23), xycoords='data',
             arrowprops=dict(facecolor='black', shrink=0.05),
             horizontalalignment='center', verticalalignment='center')
plt.plot([1.0/3], [1.0/3], 'ko', ms=5)
plt.annotate(r'($\frac{1}{2}$, $0$, $\frac{1}{2}$)',
             xy=(.5, .0), xytext=(.5, .1), xycoords='data',
             arrowprops=dict(facecolor='black', shrink=0.05),
             horizontalalignment='center', verticalalignment='center')
plt.annotate(r'($0$, $\frac{1}{2}$, $\frac{1}{2}$)',
             xy=(.0, .5), xytext=(.1, .5), xycoords='data',
             arrowprops=dict(facecolor='black', shrink=0.05),
             horizontalalignment='center', verticalalignment='center')
plt.annotate(r'($\frac{1}{2}$, $\frac{1}{2}$, $0$)',
```

(continues on next page)

(continued from previous page)

```
             xy=(.5, .5), xytext=(.6, .6), xycoords='data',
             arrowprops=dict(facecolor='black', shrink=0.05),
             horizontalalignment='center', verticalalignment='center')
plt.annotate(r'($0$, $0$, $1$)',
             xy=(0, 0), xytext=(.1, .1), xycoords='data',
             arrowprops=dict(facecolor='black', shrink=0.05),
             horizontalalignment='center', verticalalignment='center')
plt.annotate(r'($1$, $0$, $0$)',
             xy=(1, 0), xytext=(1, .1), xycoords='data',
             arrowprops=dict(facecolor='black', shrink=0.05),
             horizontalalignment='center', verticalalignment='center')
plt.annotate(r'($0$, $1$, $0$)',
             xy=(0, 1), xytext=(.1, 1), xycoords='data',
             arrowprops=dict(facecolor='black', shrink=0.05),
             horizontalalignment='center', verticalalignment='center')
# Add grid
plt.grid(False)
for x in [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]:
    plt.plot([0, x], [x, 0], 'k', alpha=0.2)
    plt.plot([0, 0 + (1-x)/2], [x, x + (1-x)/2], 'k', alpha=0.2)
    plt.plot([x, x + (1-x)/2], [0, 0 + (1-x)/2], 'k', alpha=0.2)

plt.title("Change of predicted probabilities after sigmoid calibration")
plt.xlabel("Probability class 1")
plt.ylabel("Probability class 2")
plt.xlim(-0.05, 1.05)
plt.ylim(-0.05, 1.05)
plt.legend(loc="best")

print("Log-loss of")
print(" * uncalibrated classifier trained on 800 datapoints: %.3f "
      % score)
print(" * classifier trained on 600 datapoints and calibrated on "
      "200 datapoint: %.3f" % sig_score)

# Illustrate calibrator
plt.figure()
# generate grid over 2-simplex
p1d = np.linspace(0, 1, 20)
p0, p1 = np.meshgrid(p1d, p1d)
p2 = 1 - p0 - p1
p = np.c_[p0.ravel(), p1.ravel(), p2.ravel()]
p = p[p[:, 2] >= 0]

calibrated_classifier = sig_clf.calibrated_classifiers_[0]
prediction = np.vstack([calibrator.predict(this_p)
                        for calibrator, this_p in
                        zip(calibrated_classifier.calibrators_, p.T)]).T
prediction /= prediction.sum(axis=1)[:, None]

# Plot modifications of calibrator
for i in range(prediction.shape[0]):
    plt.arrow(p[i, 0], p[i, 1],
              prediction[i, 0] - p[i, 0], prediction[i, 1] - p[i, 1],
              head_width=1e-2, color=colors[np.argmax(p[i])])
# Plot boundaries of unit simplex
plt.plot([0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0], 'k', label="Simplex")
```

(continues on next page)

(continued from previous page)

```
plt.grid(False)
for x in [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]:
    plt.plot([0, x], [x, 0], 'k', alpha=0.2)
    plt.plot([0, 0 + (1-x)/2], [x, x + (1-x)/2], 'k', alpha=0.2)
    plt.plot([x, x + (1-x)/2], [0, 0 + (1-x)/2], 'k', alpha=0.2)

plt.title("Illustration of sigmoid calibrator")
plt.xlabel("Probability class 1")
plt.ylabel("Probability class 2")
plt.xlim(-0.05, 1.05)
plt.ylim(-0.05, 1.05)

plt.show()
```

Total running time of the script: (0 minutes 0.670 seconds)

Estimated memory usage: 8 MB

6.4 Classification

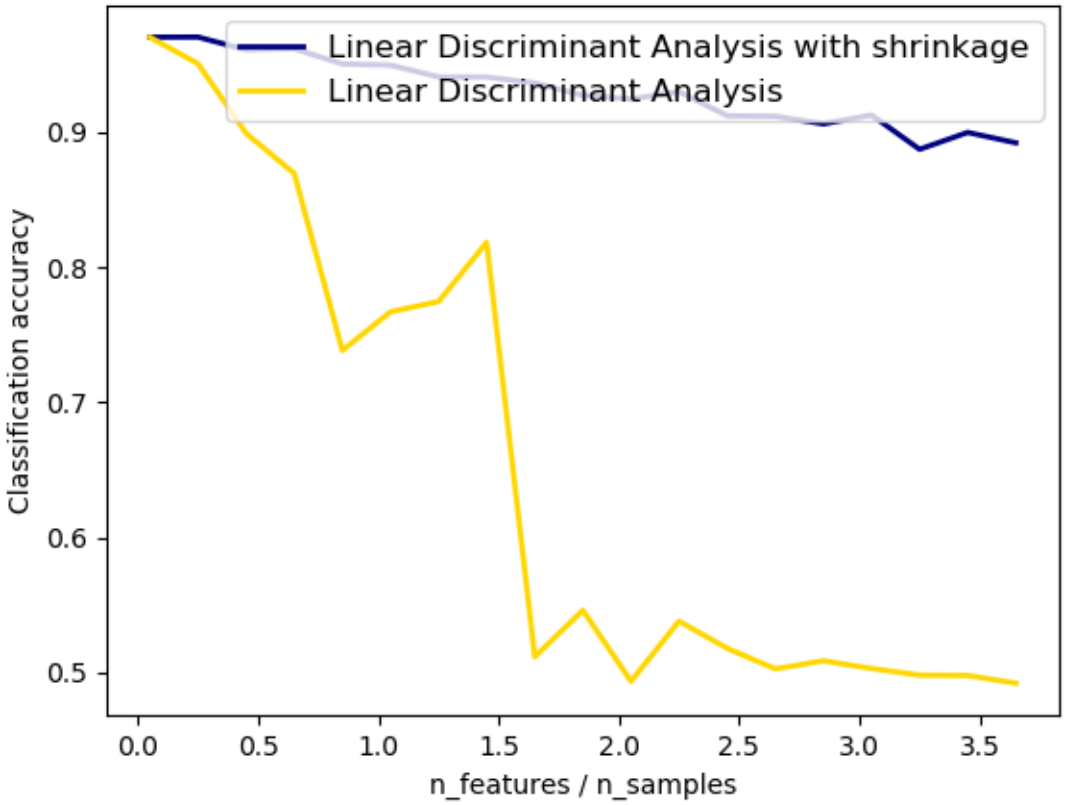
General examples about classification algorithms.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.4.1 Normal and Shrinkage Linear Discriminant Analysis for classification

Shows how shrinkage improves classification.

criminant Analysis vs. shrinkage Linear Discriminant Analysis (1 discriminati




```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis


n_train = 20  # samples for training
n_test = 200  # samples for testing
n_averages = 50  # how often to repeat classification
n_features_max = 75  # maximum number of features
step = 4  # step size for the calculation


def generate_data(n_samples, n_features):
    """Generate random blob-ish data with noisy features.

    This returns an array of input data with shape `(n_samples, n_features)`
    and an array of `n_samples` target labels.

    Only one feature contains discriminative information, the other features
    contain only noise.
    """
    X, y = make_blobs(n_samples=n_samples, n_features=1, centers=[[-2], [2]])
```

(continues on next page)

(continued from previous page)

```
    # add non-discriminative features
    if n_features > 1:
        X = np.hstack([X, np.random.randn(n_samples, n_features - 1)])
    return X, y

acc_clf1, acc_clf2 = [], []
n_features_range = range(1, n_features_max + 1, step)
for n_features in n_features_range:
    score_clf1, score_clf2 = 0, 0
    for _ in range(n_averages):
        X, y = generate_data(n_train, n_features)

        clf1 = LinearDiscriminantAnalysis(solver='lsqr', shrinkage='auto').fit(X, y)
        clf2 = LinearDiscriminantAnalysis(solver='lsqr', shrinkage=None).fit(X, y)

        X, y = generate_data(n_test, n_features)
        score_clf1 += clf1.score(X, y)
        score_clf2 += clf2.score(X, y)

    acc_clf1.append(score_clf1 / n_averages)
    acc_clf2.append(score_clf2 / n_averages)

features_samples_ratio = np.array(n_features_range) / n_train

plt.plot(features_samples_ratio, acc_clf1, linewidth=2,
         label="Linear Discriminant Analysis with shrinkage", color='navy')
plt.plot(features_samples_ratio, acc_clf2, linewidth=2,
         label="Linear Discriminant Analysis", color='gold')

plt.xlabel('n_features / n_samples')
plt.ylabel('Classification accuracy')

plt.legend(loc=1, prop={'size': 12})
plt.suptitle('Linear Discriminant Analysis vs. \
shrinkage Linear Discriminant Analysis (1 discriminative feature)')
plt.show()
```

Total running time of the script: (0 minutes 3.857 seconds)

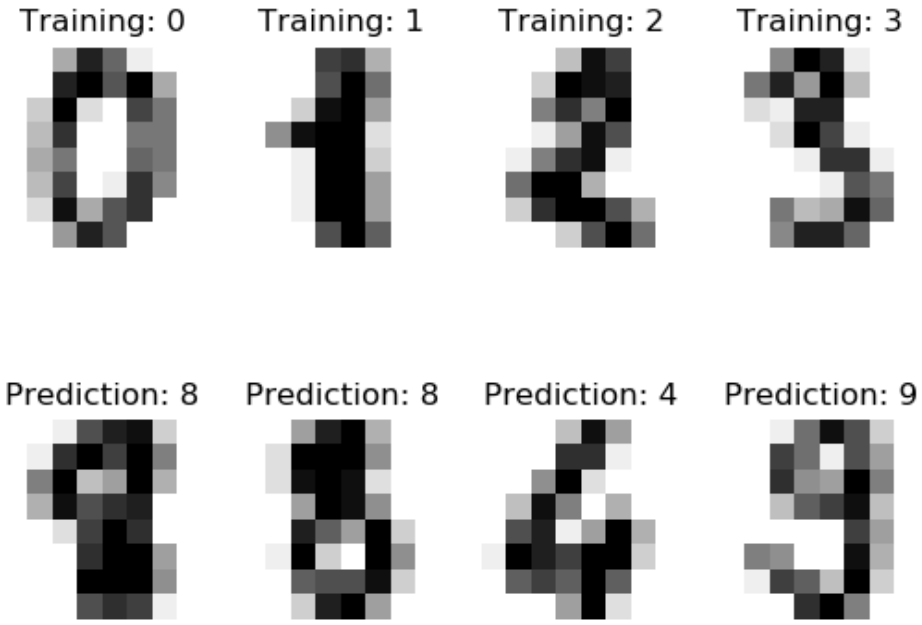
Estimated memory usage: 8 MB

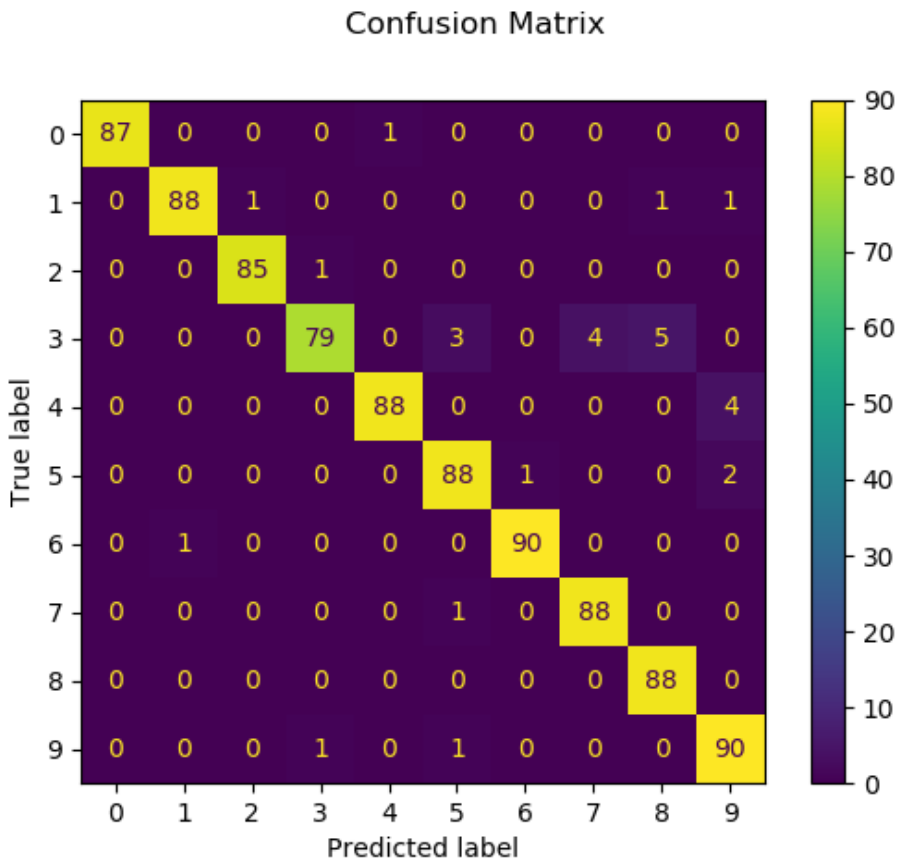
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.4.2 Recognizing hand-written digits

An example showing how the scikit-learn can be used to recognize images of hand-written digits.

This example is commented in the *tutorial section of the user manual*.


Training: 0
Training: 1
Training: 2
Training: 3
Prediction: 8
Prediction: 8
Prediction: 4
Prediction: 9



-

Out:

```
Classification report for classifier SVC(gamma=0.001):
              precision    recall  f1-score   support

           0       1.00      0.99      0.99        88
           1       0.99      0.97      0.98        91
           2       0.99      0.99      0.99        86
           3       0.98      0.87      0.92        91
           4       0.99      0.96      0.97        92
           5       0.95      0.97      0.96        91
           6       0.99      0.99      0.99        91
           7       0.96      0.99      0.97        89
           8       0.94      1.00      0.97        88
           9       0.93      0.98      0.95        92

    accuracy                           0.97       899
   macro avg       0.97      0.97      0.97       899
weighted avg       0.97      0.97      0.97       899


Confusion matrix:
[[87  0  0  0  1  0  0  0  0  0]
 [ 0 88  1  0  0  0  0  0  1  1]
 [ 0  0 85  1  0  0  0  0  0  0]
 [ 0  0  0 79  0  3  0  4  5  0]
 [ 0  0  0  0 88  0  0  0  0  4]
```

(continues on next page)

(continued from previous page)

```
 [ 0  0  0  0  0 88  1  0  0  2]
 [ 0  1  0  0  0  0 90  0  0  0]
 [ 0  0  0  0  0  1  0 88  0  0]
 [ 0  0  0  0  0  0  0  0 88  0]
 [ 0  0  0  1  0  1  0  0  0 90]]
```

```
print(__doc__)

# Author: Gael Varoquaux <gael dot varoquaux at normalesup dot org>
# License: BSD 3 clause

# Standard scientific Python imports
import matplotlib.pyplot as plt

# Import datasets, classifiers and performance metrics
from sklearn import datasets, svm, metrics
from sklearn.model_selection import train_test_split

# The digits dataset
digits = datasets.load_digits()

# The data that we are interested in is made of 8x8 images of digits, let's
# have a look at the first 4 images, stored in the `images` attribute of the
# dataset.  If we were working from image files, we could load them using
# matplotlib.pyplot.imread.  Note that each image must have the same size. For these
# images, we know which digit they represent: it is given in the 'target' of
# the dataset.
_, axes = plt.subplots(2, 4)
images_and_labels = list(zip(digits.images, digits.target))
for ax, (image, label) in zip(axes[0, :], images_and_labels[:4]):
    ax.set_axis_off()
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    ax.set_title('Training: %i' % label)

# To apply a classifier on this data, we need to flatten the image, to
# turn the data in a (samples, feature) matrix:
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))

# Create a classifier: a support vector classifier
classifier = svm.SVC(gamma=0.001)

# Split data into train and test subsets
X_train, X_test, y_train, y_test = train_test_split(
    data, digits.target, test_size=0.5, shuffle=False)

# We learn the digits on the first half of the digits
classifier.fit(X_train, y_train)

# Now predict the value of the digit on the second half:
predicted = classifier.predict(X_test)
```

(continues on next page)

(continued from previous page)

```
images_and_predictions = list(zip(digits.images[n_samples // 2:], predicted))
for ax, (image, prediction) in zip(axes[1, :], images_and_predictions[:4]):
    ax.set_axis_off()
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    ax.set_title('Prediction: %i' % prediction)

print("Classification report for classifier %s:\n%s\n"
      % (classifier, metrics.classification_report(y_test, predicted)))
disp = metrics.plot_confusion_matrix(classifier, X_test, y_test)
disp.figure_.suptitle("Confusion Matrix")
print("Confusion matrix:\n%s" % disp.confusion_matrix)

plt.show()
```

Total running time of the script: (0 minutes 0.654 seconds)

Estimated memory usage: 8 MB

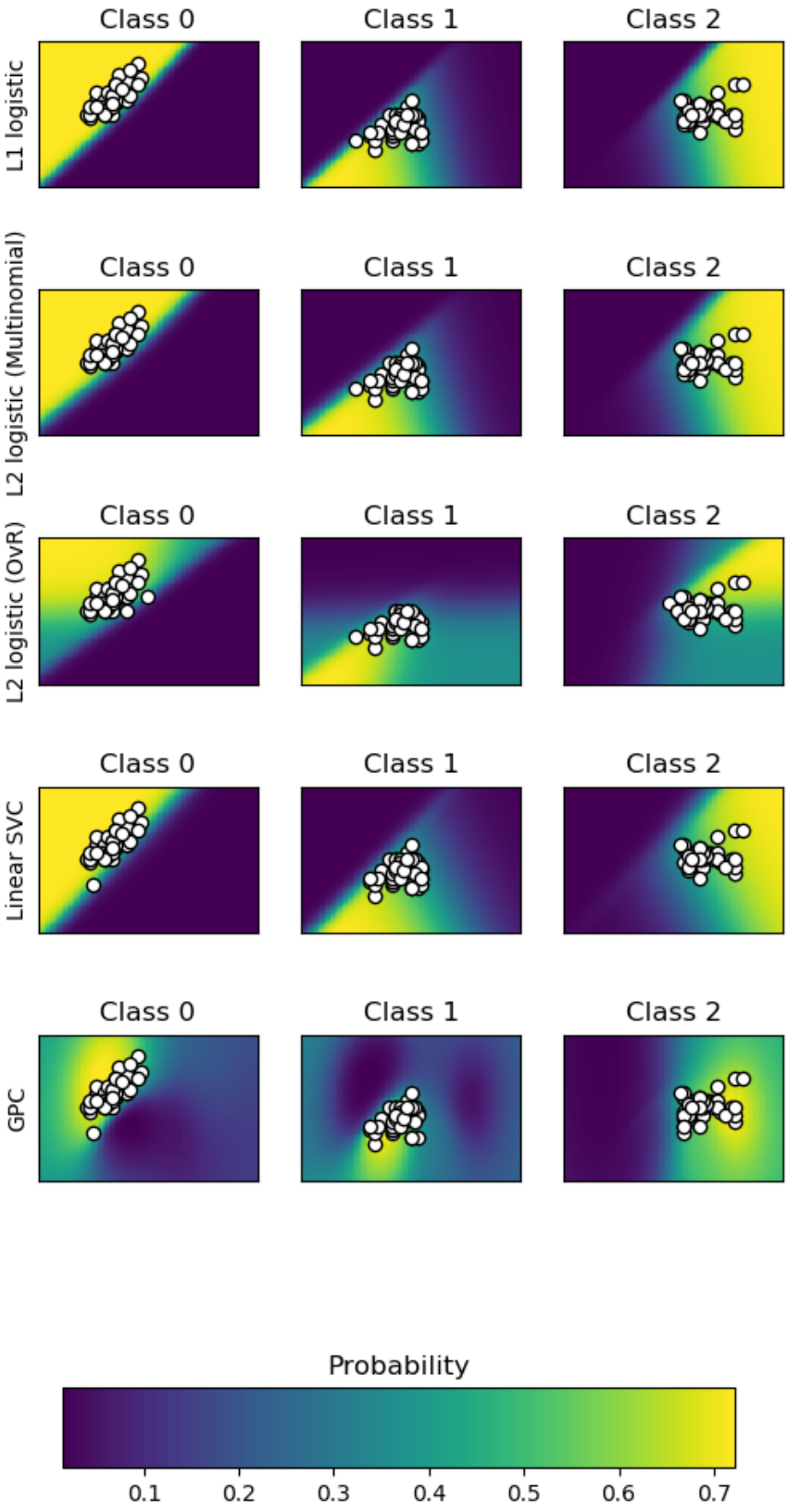
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.4.3 Plot classification probability

Plot the classification probability for different classifiers. We use a 3 class dataset, and we classify it with a Support Vector classifier, L1 and L2 penalized logistic regression with either a One-Vs-Rest or multinomial setting, and Gaussian process classification.

Linear SVC is not a probabilistic classifier by default but it has a built-in calibration option enabled in this example (`probability=True`).

The logistic regression with One-Vs-Rest is not a multiclass classifier out of the box. As a result it has more trouble in separating class 2 and 3 than the other estimators.


Class 0
Class 1
Class 2
L1 logistic
L2 logistic (Multinomial)
L2 logistic (OvR)
Linear SVC
GPC
Probability
0.1
0.2
0.3
0.4
0.5
0.6
0.7

Out:

```
Accuracy (train) for L1 logistic: 82.7%
Accuracy (train) for L2 logistic (Multinomial): 82.7%
Accuracy (train) for L2 logistic (OvR): 79.3%
Accuracy (train) for Linear SVC: 82.0%
Accuracy (train) for GPC: 82.7%
```

```
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

import matplotlib.pyplot as plt
import numpy as np

from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn import datasets

iris = datasets.load_iris()
X = iris.data[:, 0:2]  # we only take the first two features for visualization
y = iris.target

n_features = X.shape[1]

C = 10
kernel = 1.0 * RBF([1.0, 1.0])  # for GPC

# Create different classifiers.
classifiers = {
    'L1 logistic': LogisticRegression(C=C, penalty='l1',
                                      solver='saga',
                                      multi_class='multinomial',
                                      max_iter=10000),
    'L2 logistic (Multinomial)': LogisticRegression(C=C, penalty='l2',
                                                    solver='saga',
                                                    multi_class='multinomial',
                                                    max_iter=10000),
    'L2 logistic (OvR)': LogisticRegression(C=C, penalty='l2',
                                            solver='saga',
                                            multi_class='ovr',
                                            max_iter=10000),
    'Linear SVC': SVC(kernel='linear', C=C, probability=True,
                      random_state=0),
    'GPC': GaussianProcessClassifier(kernel)
}

n_classifiers = len(classifiers)
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(3 * 2, n_classifiers * 2))
plt.subplots_adjust(bottom=.2, top=.95)

xx = np.linspace(3, 9, 100)
yy = np.linspace(1, 5, 100).T
xx, yy = np.meshgrid(xx, yy)
Xfull = np.c_[xx.ravel(), yy.ravel()]

for index, (name, classifier) in enumerate(classifiers.items()):
    classifier.fit(X, y)

    y_pred = classifier.predict(X)
    accuracy = accuracy_score(y, y_pred)
    print("Accuracy (train) for %s: %0.1f%% " % (name, accuracy * 100))

    # View probabilities:
    probas = classifier.predict_proba(Xfull)
    n_classes = np.unique(y_pred).size
    for k in range(n_classes):
        plt.subplot(n_classifiers, n_classes, index * n_classes + k + 1)
        plt.title("Class %d" % k)
        if k == 0:
            plt.ylabel(name)
        imshow_handle = plt.imshow(probas[:, k].reshape((100, 100)),
                                   extent=(3, 9, 1, 5), origin='lower')
        plt.xticks(())
        plt.yticks(())
        idx = (y_pred == k)
        if idx.any():
            plt.scatter(X[idx, 0], X[idx, 1], marker='o', c='w', edgecolor='k')

ax = plt.axes([0.15, 0.04, 0.7, 0.05])
plt.title("Probability")
plt.colorbar(imshow_handle, cax=ax, orientation='horizontal')

plt.show()
```

Total running time of the script: (0 minutes 1.188 seconds)

Estimated memory usage: 45 MB

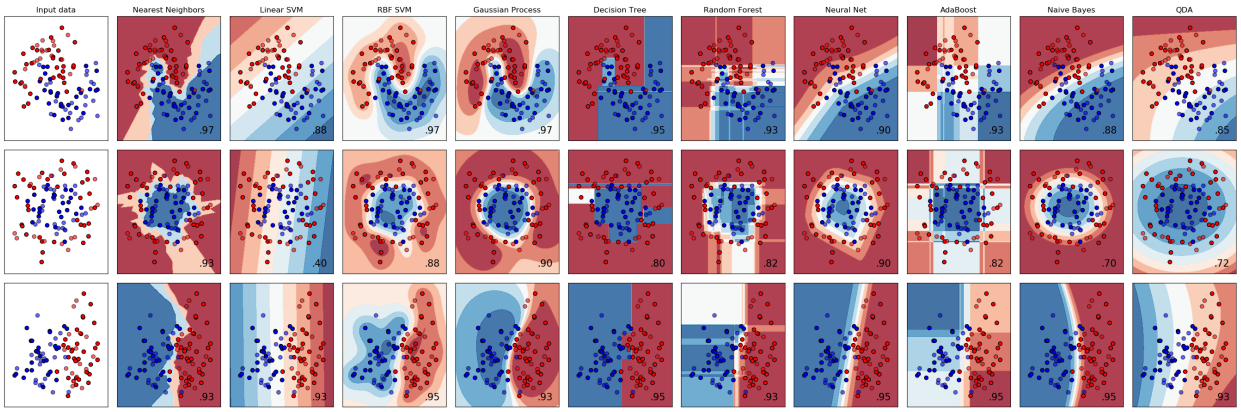
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.4.4 Classifier comparison

A comparison of a several classifiers in scikit-learn on synthetic datasets. The point of this example is to illustrate the nature of decision boundaries of different classifiers. This should be taken with a grain of salt, as the intuition conveyed by these examples does not necessarily carry over to real datasets.

Particularly in high-dimensional spaces, data can more easily be separated linearly and the simplicity of classifiers such as naive Bayes and linear SVMs might lead to better generalization than is achieved by other classifiers.

The plots show training points in solid colors and testing points semi-transparent. The lower right shows the classification accuracy on the test set.




```
print(__doc__)


# Code source: Gaël Varoquaux
#              Andreas Müller
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

h = .02  # step size in the mesh

names = ["Nearest Neighbors", "Linear SVM", "RBF SVM", "Gaussian Process",
         "Decision Tree", "Random Forest", "Neural Net", "AdaBoost",
         "Naive Bayes", "QDA"]

classifiers = [
    KNeighborsClassifier(3),
    SVC(kernel="linear", C=0.025),
    SVC(gamma=2, C=1),
    GaussianProcessClassifier(1.0 * RBF(1.0)),
    DecisionTreeClassifier(max_depth=5),
    RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1),
    MLPClassifier(alpha=1, max_iter=1000),
    AdaBoostClassifier(),
    GaussianNB(),
    QuadraticDiscriminantAnalysis()]

X, y = make_classification(n_features=2, n_redundant=0, n_informative=2,
```

(continues on next page)

(continued from previous page)

```
                           random_state=1, n_clusters_per_class=1)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [make_moons(noise=0.3, random_state=0),
            make_circles(noise=0.2, factor=0.5, random_state=1),
            linearly_separable
            ]

figure = plt.figure(figsize=(27, 9))
i = 1
# iterate over datasets
for ds_cnt, ds in enumerate(datasets):
    # preprocess dataset, split into training and test part
    X, y = ds
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=.4, random_state=42)

    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # just plot the dataset first
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])
    ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
    if ds_cnt == 0:
        ax.set_title("Input data")
    # Plot the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
               edgecolors='k')
    # Plot the testing points
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.6,
               edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    i += 1

    # iterate over classifiers
    for name, clf in zip(names, classifiers):
        ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
        clf.fit(X_train, y_train)
        score = clf.score(X_test, y_test)

        # Plot the decision boundary. For that, we will assign a color to each
        # point in the mesh [x_min, x_max]x[y_min, y_max].
        if hasattr(clf, "decision_function"):
            Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
        else:
            Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

        # Put the result into a color plot
```

(continues on next page)

(continued from previous page)

```
        Z = Z.reshape(xx.shape)
        ax.contourf(xx, yy, Z, cmap=cm, alpha=.8)

        # Plot the training points
        ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
                   edgecolors='k')
        # Plot the testing points
        ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
                   edgecolors='k', alpha=0.6)

        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
        ax.set_xticks(())
        ax.set_yticks(())
        if ds_cnt == 0:
            ax.set_title(name)
        ax.text(xx.max() - .3, yy.min() + .3, ('%.2f' % score).lstrip('0'),
                size=15, horizontalalignment='right')
        i += 1

plt.tight_layout()
plt.show()
```

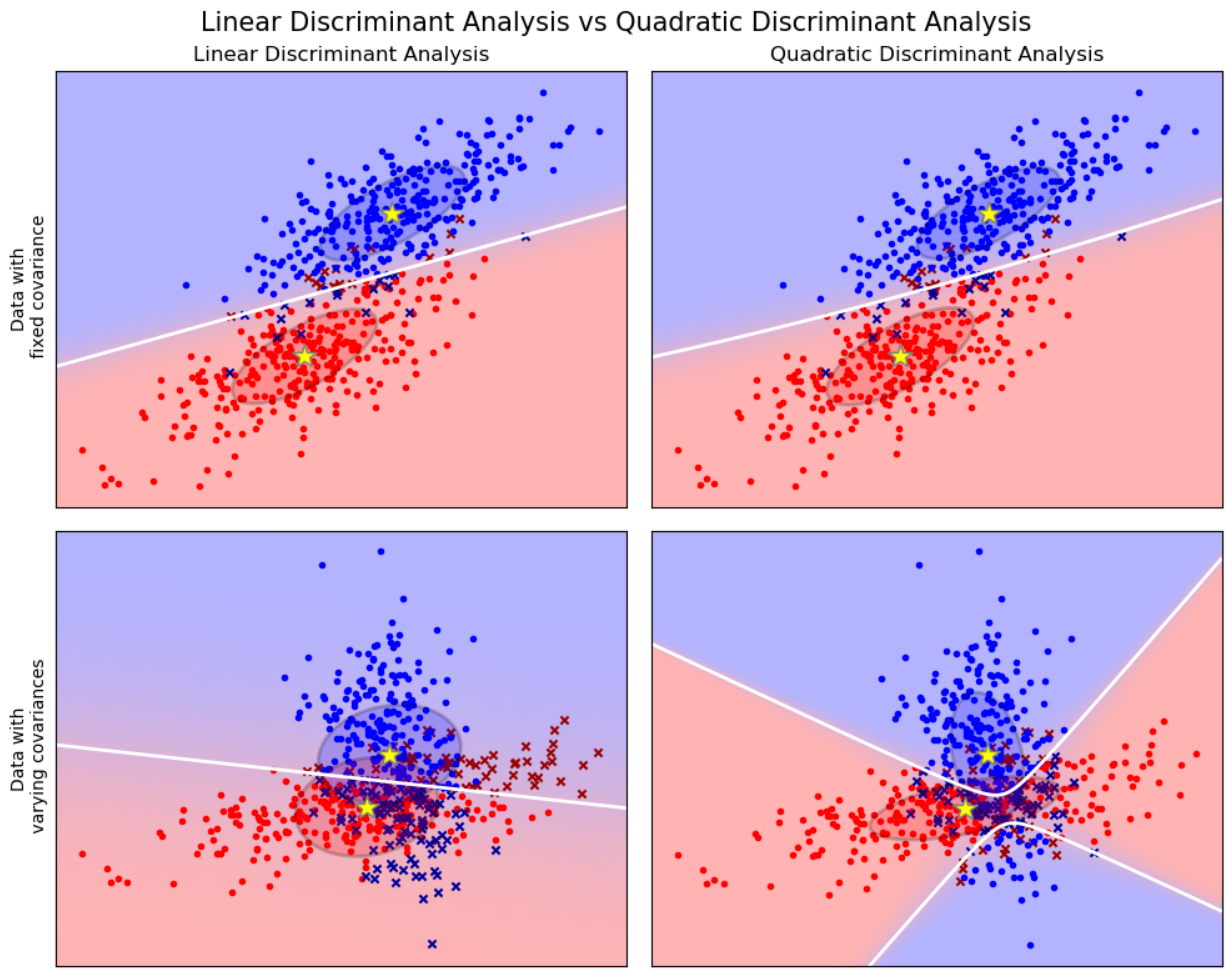
Total running time of the script: (0 minutes 4.667 seconds)

Estimated memory usage: 97 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.4.5 Linear and Quadratic Discriminant Analysis with covariance ellipsoid

This example plots the covariance ellipsoids of each class and decision boundary learned by LDA and QDA. The ellipsoids display the double standard deviation for each class. With LDA, the standard deviation is the same for all the classes, while each class has its own standard deviation with QDA.




```
print(__doc__)

from scipy import linalg
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib import colors

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

# #############################################################################
# Colormap
cmap = colors.LinearSegmentedColormap(
    'red_blue_classes',
    {'red': [(0, 1, 1), (1, 0.7, 0.7)],
     'green': [(0, 0.7, 0.7), (1, 0.7, 0.7)],
     'blue': [(0, 0.7, 0.7), (1, 1, 1)]})
plt.cm.register_cmap(cmap=cmap)


# #############################################################################
# Generate datasets
```

(continues on next page)

(continued from previous page)

```
def dataset_fixed_cov():
    '''Generate 2 Gaussians samples with the same covariance matrix'''
    n, dim = 300, 2
    np.random.seed(0)
    C = np.array([[0., -0.23], [0.83, .23]])
    X = np.r_[np.dot(np.random.randn(n, dim), C),
              np.dot(np.random.randn(n, dim), C) + np.array([1, 1])]
    y = np.hstack((np.zeros(n), np.ones(n)))
    return X, y


def dataset_cov():
    '''Generate 2 Gaussians samples with different covariance matrices'''
    n, dim = 300, 2
    np.random.seed(0)
    C = np.array([[0., -1.], [2.5, .7]]) * 2.
    X = np.r_[np.dot(np.random.randn(n, dim), C),
              np.dot(np.random.randn(n, dim), C.T) + np.array([1, 4])]
    y = np.hstack((np.zeros(n), np.ones(n)))
    return X, y


# ##############################################################################
# Plot functions
def plot_data(lda, X, y, y_pred, fig_index):
    splot = plt.subplot(2, 2, fig_index)
    if fig_index == 1:
        plt.title('Linear Discriminant Analysis')
        plt.ylabel('Data with\n fixed covariance')
    elif fig_index == 2:
        plt.title('Quadratic Discriminant Analysis')
    elif fig_index == 3:
        plt.ylabel('Data with\n varying covariances')

    tp = (y == y_pred)  # True Positive
    tp0, tp1 = tp[y == 0], tp[y == 1]
    X0, X1 = X[y == 0], X[y == 1]
    X0_tp, X0_fp = X0[tp0], X0[~tp0]
    X1_tp, X1_fp = X1[tp1], X1[~tp1]

    # class 0: dots
    plt.scatter(X0_tp[:, 0], X0_tp[:, 1], marker='.', color='red')
    plt.scatter(X0_fp[:, 0], X0_fp[:, 1], marker='x',
                s=20, color='#990000')  # dark red

    # class 1: dots
    plt.scatter(X1_tp[:, 0], X1_tp[:, 1], marker='.', color='blue')
    plt.scatter(X1_fp[:, 0], X1_fp[:, 1], marker='x',
                s=20, color='#000099')  # dark blue

    # class 0 and 1 : areas
    nx, ny = 200, 100
    x_min, x_max = plt.xlim()
    y_min, y_max = plt.ylim()
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, nx),
                         np.linspace(y_min, y_max, ny))
    Z = lda.predict_proba(np.c_[xx.ravel(), yy.ravel()])
```

(continues on next page)

(continued from previous page)

```
    Z = Z[:, 1].reshape(xx.shape)
    plt.pcolormesh(xx, yy, Z, cmap='red_blue_classes',
                   norm=colors.Normalize(0., 1.), zorder=0)
    plt.contour(xx, yy, Z, [0.5], linewidths=2., colors='white')

    # means
    plt.plot(lda.means_[0][0], lda.means_[0][1],
             '*', color='yellow', markersize=15, markeredgecolor='grey')
    plt.plot(lda.means_[1][0], lda.means_[1][1],
             '*', color='yellow', markersize=15, markeredgecolor='grey')

    return splot


def plot_ellipse(splot, mean, cov, color):
    v, w = linalg.eigh(cov)
    u = w[0] / linalg.norm(w[0])
    angle = np.arctan(u[1] / u[0])
    angle = 180 * angle / np.pi  # convert to degrees
    # filled Gaussian at 2 standard deviation
    ell = mpl.patches.Ellipse(mean, 2 * v[0] ** 0.5, 2 * v[1] ** 0.5,
                              180 + angle, facecolor=color,
                              edgecolor='black', linewidth=2)
    ell.set_clip_box(splot.bbox)
    ell.set_alpha(0.2)
    splot.add_artist(ell)
    splot.set_xticks(())
    splot.set_yticks(())


def plot_lda_cov(lda, splot):
    plot_ellipse(splot, lda.means_[0], lda.covariance_, 'red')
    plot_ellipse(splot, lda.means_[1], lda.covariance_, 'blue')


def plot_qda_cov(qda, splot):
    plot_ellipse(splot, qda.means_[0], qda.covariance_[0], 'red')
    plot_ellipse(splot, qda.means_[1], qda.covariance_[1], 'blue')


plt.figure(figsize=(10, 8), facecolor='white')
plt.suptitle('Linear Discriminant Analysis vs Quadratic Discriminant Analysis',
             y=0.98, fontsize=15)
for i, (X, y) in enumerate([dataset_fixed_cov(), dataset_cov()]):
    # Linear Discriminant Analysis
    lda = LinearDiscriminantAnalysis(solver="svd", store_covariance=True)
    y_pred = lda.fit(X, y).predict(X)
    splot = plot_data(lda, X, y, y_pred, fig_index=2 * i + 1)
    plot_lda_cov(lda, splot)
    plt.axis('tight')

    # Quadratic Discriminant Analysis
    qda = QuadraticDiscriminantAnalysis(store_covariance=True)
    y_pred = qda.fit(X, y).predict(X)
    splot = plot_data(qda, X, y, y_pred, fig_index=2 * i + 2)
    plot_qda_cov(qda, splot)
    plt.axis('tight')
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.subplots_adjust(top=0.92)
plt.show()
```

Total running time of the script: (0 minutes 0.567 seconds)

Estimated memory usage: 8 MB

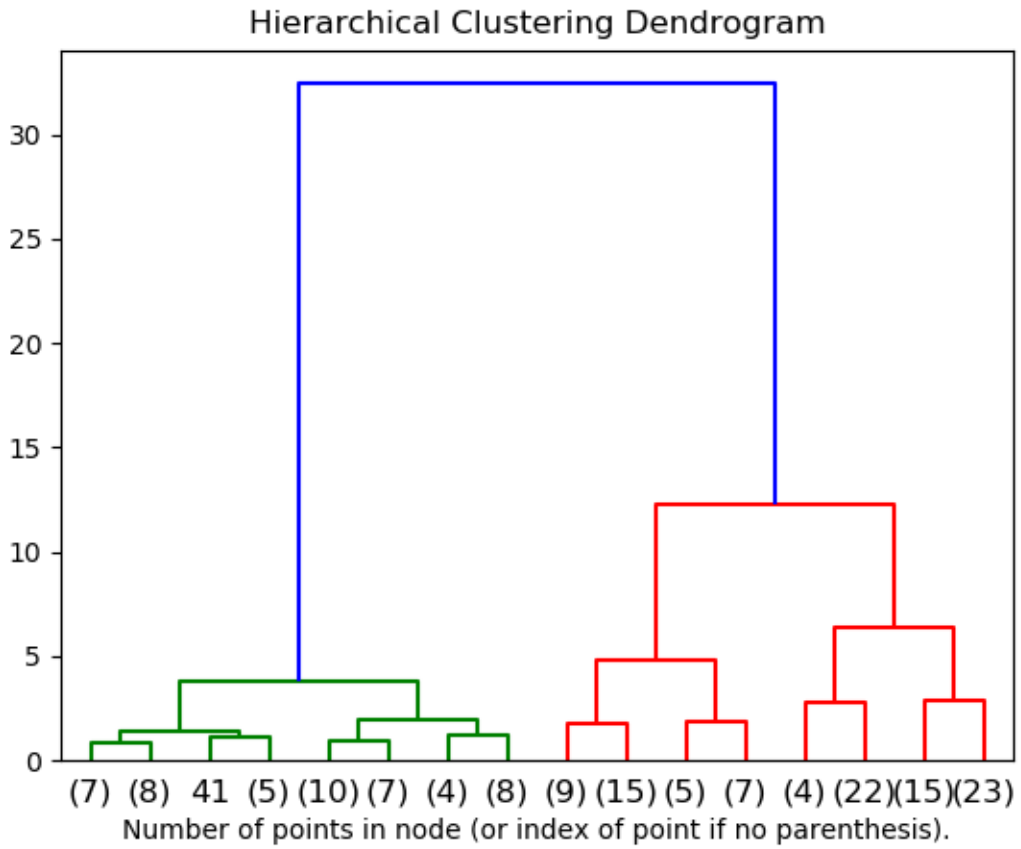
6.5 Clustering

Examples concerning the *sklearn.cluster* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.1 Plot Hierarchical Clustering Dendrogram

This example plots the corresponding dendrogram of a hierarchical clustering using AgglomerativeClustering and the dendrogram method available in scipy.



```
import numpy as np

from matplotlib import pyplot as plt
from scipy.cluster.hierarchy import dendrogram
from sklearn.datasets import load_iris
from sklearn.cluster import AgglomerativeClustering


def plot_dendrogram(model, **kwargs):
    # Create linkage matrix and then plot the dendrogram

    # create the counts of samples under each node
    counts = np.zeros(model.children_.shape[0])
    n_samples = len(model.labels_)
    for i, merge in enumerate(model.children_):
        current_count = 0
        for child_idx in merge:
            if child_idx < n_samples:
                current_count += 1  # leaf node
            else:
                current_count += counts[child_idx - n_samples]
        counts[i] = current_count

    linkage_matrix = np.column_stack([model.children_, model.distances_,
                                      counts]).astype(float)

    # Plot the corresponding dendrogram
    dendrogram(linkage_matrix, **kwargs)


iris = load_iris()
X = iris.data

# setting distance_threshold=0 ensures we compute the full tree.
model = AgglomerativeClustering(distance_threshold=0, n_clusters=None)

model = model.fit(X)
plt.title('Hierarchical Clustering Dendrogram')
# plot the top three levels of the dendrogram
plot_dendrogram(model, truncate_mode='level', p=3)
plt.xlabel("Number of points in node (or index of point if no parenthesis).")
plt.show()
```

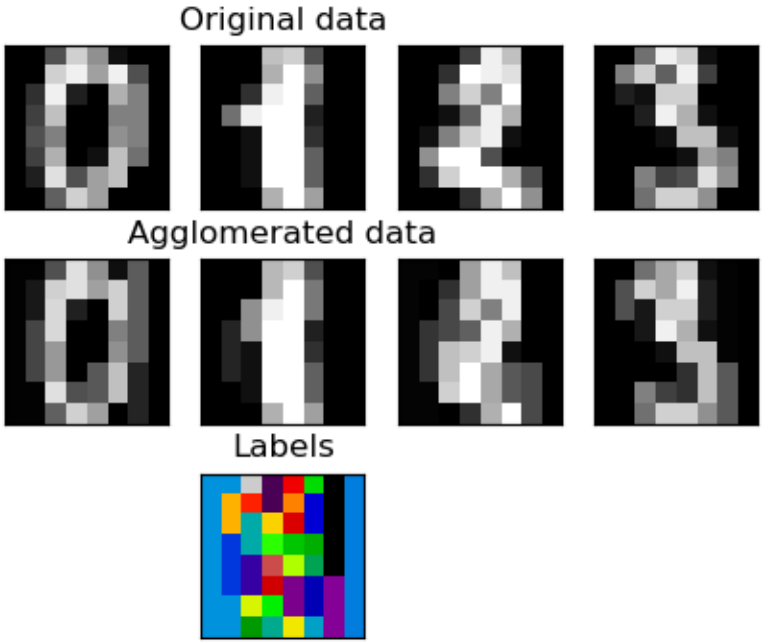
Total running time of the script: (0 minutes 0.324 seconds)

Estimated memory usage: 9 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.2 Feature agglomeration

These images how similar features are merged together using feature agglomeration.




```
print(__doc__)

# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn import datasets, cluster
from sklearn.feature_extraction.image import grid_to_graph

digits = datasets.load_digits()
images = digits.images
X = np.reshape(images, (len(images), -1))
connectivity = grid_to_graph(*images[0].shape)

agglo = cluster.FeatureAgglomeration(connectivity=connectivity,
                                     n_clusters=32)

agglo.fit(X)
X_reduced = agglo.transform(X)

X_restored = agglo.inverse_transform(X_reduced)
images_restored = np.reshape(X_restored, images.shape)
plt.figure(1, figsize=(4, 3.5))
plt.clf()
plt.subplots_adjust(left=.01, right=.99, bottom=.01, top=.91)
for i in range(4):
    plt.subplot(3, 4, i + 1)
    plt.imshow(images[i], cmap=plt.cm.gray, vmax=16, interpolation='nearest')
    plt.xticks(())
    plt.yticks(())
    if i == 1:
```

(continues on next page)

(continued from previous page)

```
        plt.title('Original data')
    plt.subplot(3, 4, 4 + i + 1)
    plt.imshow(images_restored[i], cmap=plt.cm.gray, vmax=16,
               interpolation='nearest')
    if i == 1:
        plt.title('Agglomerated data')
    plt.xticks(())
    plt.yticks(())

plt.subplot(3, 4, 10)
plt.imshow(np.reshape(agglo.labels_, images[0].shape),
           interpolation='nearest', cmap=plt.cm.nipy_spectral)
plt.xticks(())
plt.yticks(())
plt.title('Labels')
plt.show()
```

Total running time of the script: (0 minutes 0.465 seconds)

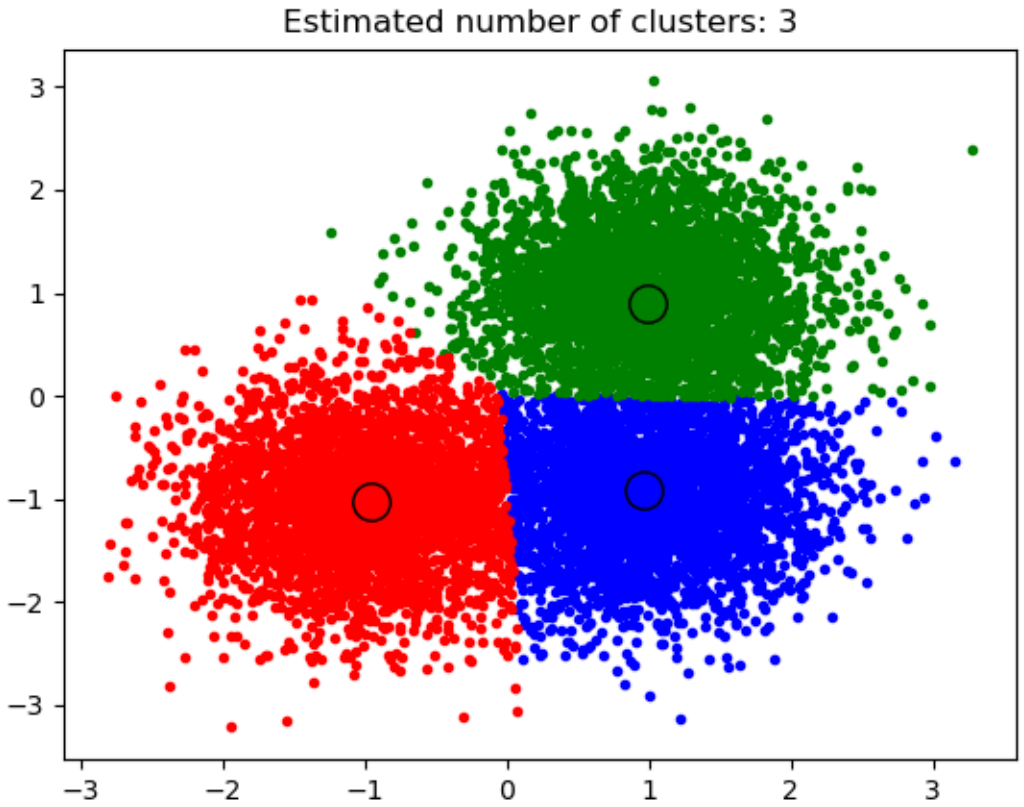
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.3 A demo of the mean-shift clustering algorithm

Reference:

Dorin Comaniciu and Peter Meer, “Mean Shift: A robust approach toward feature space analysis”. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2002. pp. 603-619.



Out:

```
number of estimated clusters : 3
```

```
print(__doc__)

import numpy as np
from sklearn.cluster import MeanShift, estimate_bandwidth
from sklearn.datasets import make_blobs

# #############################################################################
# Generate sample data
centers = [[1, 1], [-1, -1], [1, -1]]
X, _ = make_blobs(n_samples=10000, centers=centers, cluster_std=0.6)

# #############################################################################
# Compute clustering with MeanShift

# The following bandwidth can be automatically detected using
bandwidth = estimate_bandwidth(X, quantile=0.2, n_samples=500)
```

(continues on next page)

(continued from previous page)

```
ms = MeanShift(bandwidth=bandwidth, bin_seeding=True)
ms.fit(X)
labels = ms.labels_
cluster_centers = ms.cluster_centers_

labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)

print("number of estimated clusters : %d" % n_clusters_)

# #############################################################################
# Plot result
import matplotlib.pyplot as plt
from itertools import cycle

plt.figure(1)
plt.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    my_members = labels == k
    cluster_center = cluster_centers[k]
    plt.plot(X[my_members, 0], X[my_members, 1], col + '.')
    plt.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
             markeredgecolor='k', markersize=14)
plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

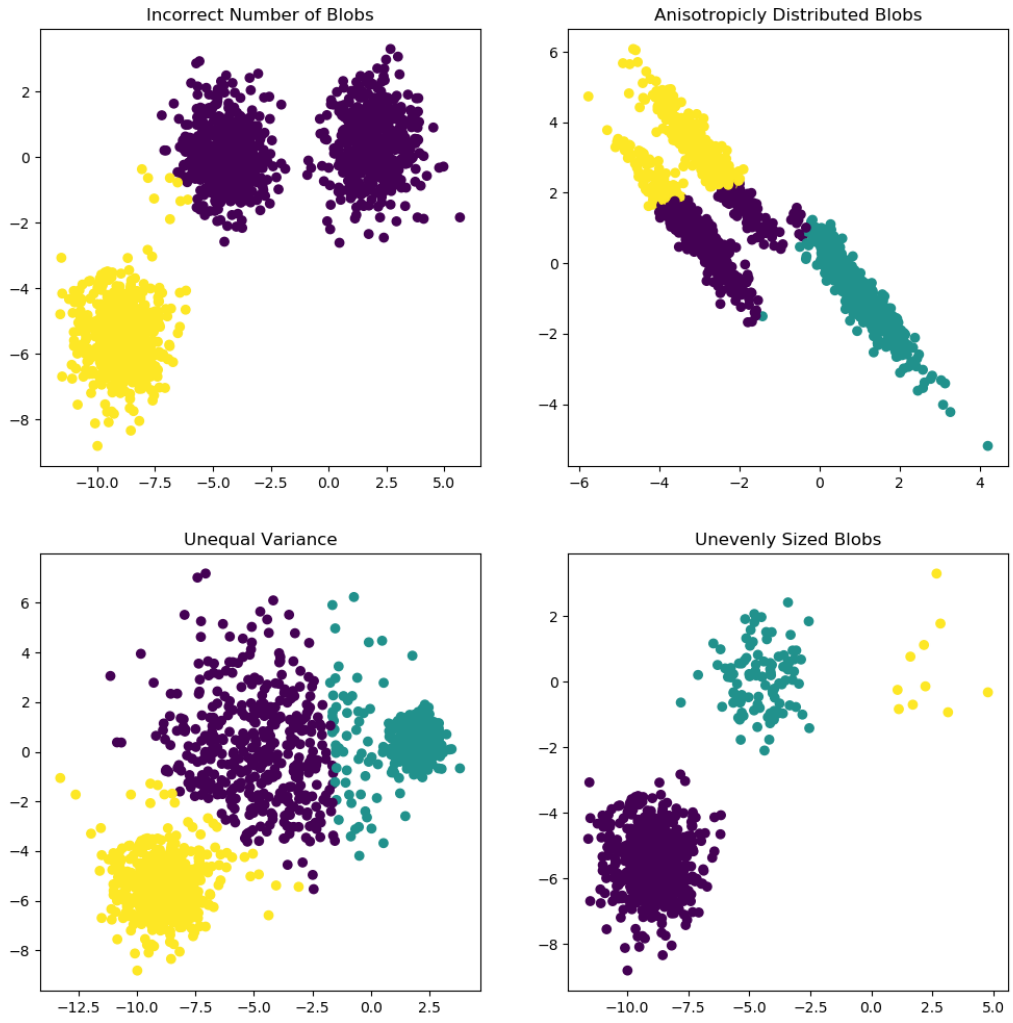
Total running time of the script: (0 minutes 0.570 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.4 Demonstration of k-means assumptions

This example is meant to illustrate situations where k-means will produce unintuitive and possibly unexpected clusters. In the first three plots, the input data does not conform to some implicit assumption that k-means makes and undesirable clusters are produced as a result. In the last plot, k-means returns intuitive clusters despite unevenly sized blobs.




```
print(__doc__)

# Author: Phil Roth <mr.phil.roth@gmail.com>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

plt.figure(figsize=(12, 12))

n_samples = 1500
```

(continues on next page)

(continued from previous page)

```
random_state = 170
X, y = make_blobs(n_samples=n_samples, random_state=random_state)

# Incorrect number of clusters
y_pred = KMeans(n_clusters=2, random_state=random_state).fit_predict(X)

plt.subplot(221)
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
plt.title("Incorrect Number of Blobs")

# Anisotropicly distributed data
transformation = [[0.60834549, -0.63667341], [-0.40887718, 0.85253229]]
X_aniso = np.dot(X, transformation)
y_pred = KMeans(n_clusters=3, random_state=random_state).fit_predict(X_aniso)

plt.subplot(222)
plt.scatter(X_aniso[:, 0], X_aniso[:, 1], c=y_pred)
plt.title("Anisotropicly Distributed Blobs")

# Different variance
X_varied, y_varied = make_blobs(n_samples=n_samples,
                                cluster_std=[1.0, 2.5, 0.5],
                                random_state=random_state)
y_pred = KMeans(n_clusters=3, random_state=random_state).fit_predict(X_varied)

plt.subplot(223)
plt.scatter(X_varied[:, 0], X_varied[:, 1], c=y_pred)
plt.title("Unequal Variance")

# Unevenly sized blobs
X_filtered = np.vstack((X[y == 0][:500], X[y == 1][:100], X[y == 2][:10]))
y_pred = KMeans(n_clusters=3,
                random_state=random_state).fit_predict(X_filtered)

plt.subplot(224)
plt.scatter(X_filtered[:, 0], X_filtered[:, 1], c=y_pred)
plt.title("Unevenly Sized Blobs")

plt.show()
```

Total running time of the script: (0 minutes 0.685 seconds)

Estimated memory usage: 8 MB

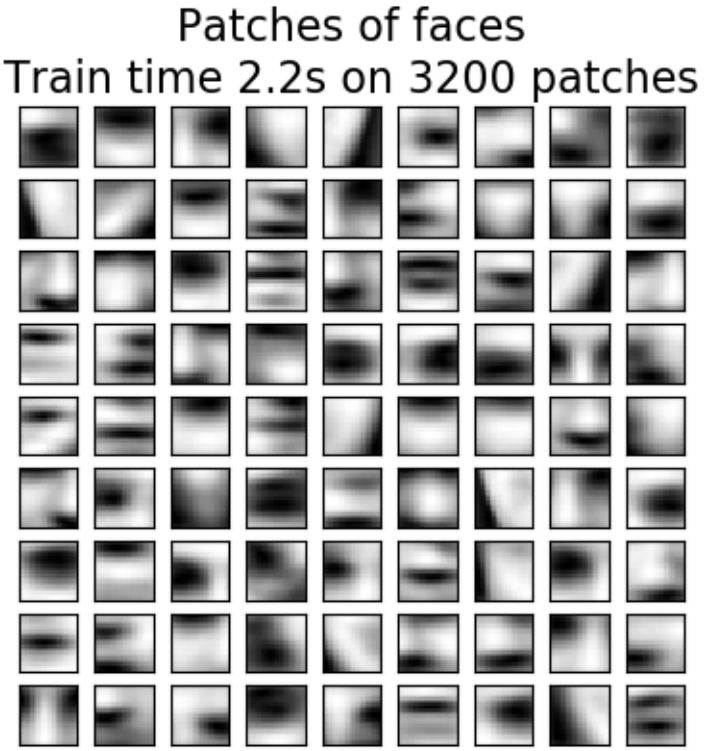
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.5 Online learning of a dictionary of parts of faces

This example uses a large dataset of faces to learn a set of 20 x 20 images patches that constitute faces.

From the programming standpoint, it is interesting because it shows how to use the online API of the scikit-learn to process a very large dataset by chunks. The way we proceed is that we load an image at a time and extract randomly 50 patches from this image. Once we have accumulated 500 of these patches (using 10 images), we run the `partial_fit` method of the online KMeans object, MiniBatchKMeans.

The verbose setting on the MiniBatchKMeans enables us to see that some clusters are reassigned during the successive calls to partial-fit. This is because the number of patches that they represent has become too low, and it is better to choose a random new cluster.



Out:

```
Learning the dictionary...
Partial fit of  100 out of 2400
Partial fit of  200 out of 2400
[MiniBatchKMeans] Reassigning 16 cluster centers.
Partial fit of  300 out of 2400
Partial fit of  400 out of 2400
Partial fit of  500 out of 2400
Partial fit of  600 out of 2400
Partial fit of  700 out of 2400
Partial fit of  800 out of 2400
Partial fit of  900 out of 2400
Partial fit of 1000 out of 2400
Partial fit of 1100 out of 2400
Partial fit of 1200 out of 2400
Partial fit of 1300 out of 2400
Partial fit of 1400 out of 2400
Partial fit of 1500 out of 2400
Partial fit of 1600 out of 2400
Partial fit of 1700 out of 2400
Partial fit of 1800 out of 2400
Partial fit of 1900 out of 2400
Partial fit of 2000 out of 2400
Partial fit of 2100 out of 2400
Partial fit of 2200 out of 2400
Partial fit of 2300 out of 2400
```

(continues on next page)

(continued from previous page)

```
Partial fit of 2400 out of 2400
done in 2.16s.
```

```
print(__doc__)

import time

import matplotlib.pyplot as plt
import numpy as np


from sklearn import datasets
from sklearn.cluster import MiniBatchKMeans
from sklearn.feature_extraction.image import extract_patches_2d

faces = datasets.fetch_olivetti_faces()

# #############################################################################
# Learn the dictionary of images

print('Learning the dictionary... ')
rng = np.random.RandomState(0)
kmeans = MiniBatchKMeans(n_clusters=81, random_state=rng, verbose=True)
patch_size = (20, 20)

buffer = []
t0 = time.time()

# The online learning part: cycle over the whole dataset 6 times
index = 0
for _ in range(6):
    for img in faces.images:
        data = extract_patches_2d(img, patch_size, max_patches=50,
                                  random_state=rng)
        data = np.reshape(data, (len(data), -1))
        buffer.append(data)
        index += 1
        if index % 10 == 0:
            data = np.concatenate(buffer, axis=0)
            data -= np.mean(data, axis=0)
            data /= np.std(data, axis=0)
            kmeans.partial_fit(data)
            buffer = []
        if index % 100 == 0:
            print('Partial fit of %4i out of %i'
                  % (index, 6 * len(faces.images)))

dt = time.time() - t0
print('done in %.2fs.' % dt)

# #############################################################################
```

(continues on next page)

(continued from previous page)

```
# Plot the results
plt.figure(figsize=(4.2, 4))
for i, patch in enumerate(kmeans.cluster_centers_):
    plt.subplot(9, 9, i + 1)
    plt.imshow(patch.reshape(patch_size), cmap=plt.cm.gray,
               interpolation='nearest')
    plt.xticks(())
    plt.yticks(())


plt.suptitle('Patches of faces\nTrain time %.1fs on %d patches' %
             (dt, 8 * len(faces.images)), fontsize=16)
plt.subplots_adjust(0.08, 0.02, 0.92, 0.85, 0.08, 0.23)

plt.show()
```

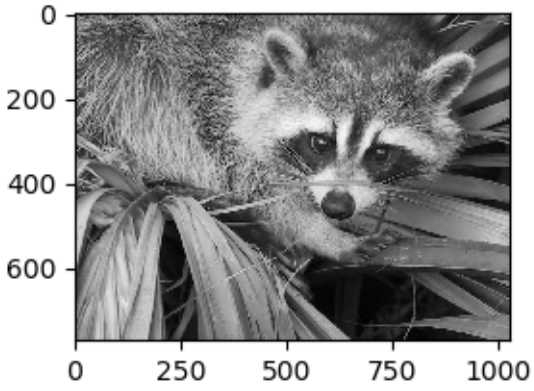
Total running time of the script: (0 minutes 3.372 seconds)

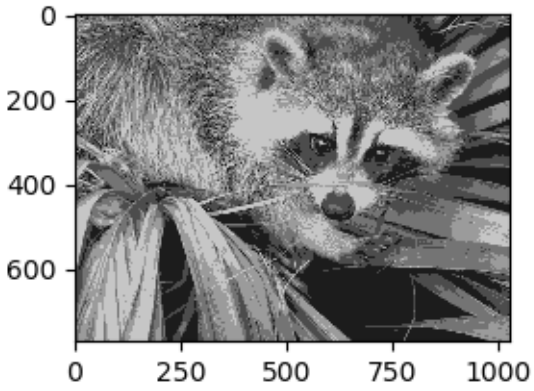
Estimated memory usage: 14 MB

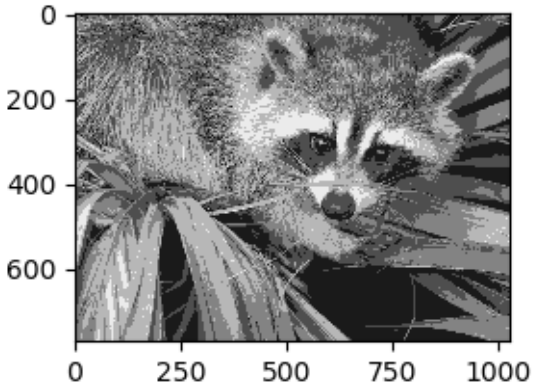
Note: Click *here* to download the full example code or to run this example in your browser via Binder

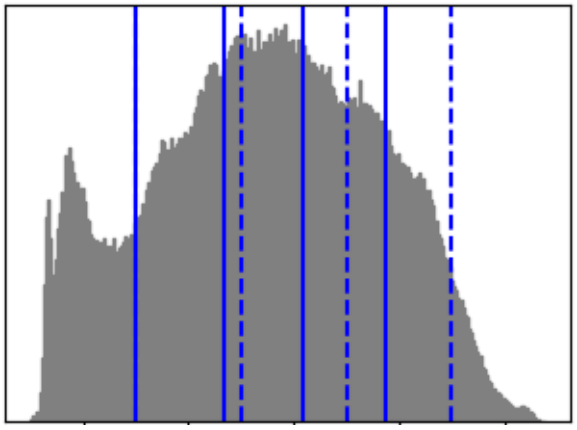
6.5.6 Vector Quantization Example

Face, a 1024 x 768 size image of a raccoon face, is used here to illustrate how `k`-means is used for vector quantization.

- 

- 

- 

- 

```
print(__doc__)


# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

from sklearn import cluster
```

(continues on next page)

(continued from previous page)

```
try:  # SciPy >= 0.16 have face in misc
    from scipy.misc import face
    face = face(gray=True)
except ImportError:
    face = sp.face(gray=True)

n_clusters = 5
np.random.seed(0)

X = face.reshape((-1, 1))  # We need an (n_sample, n_feature) array
k_means = cluster.KMeans(n_clusters=n_clusters, n_init=4)
k_means.fit(X)
values = k_means.cluster_centers_.squeeze()
labels = k_means.labels_

# create an array from labels and values
face_compressed = np.choose(labels, values)
face_compressed.shape = face.shape

vmin = face.min()
vmax = face.max()

# original face
plt.figure(1, figsize=(3, 2.2))
plt.imshow(face, cmap=plt.cm.gray, vmin=vmin, vmax=256)

# compressed face
plt.figure(2, figsize=(3, 2.2))
plt.imshow(face_compressed, cmap=plt.cm.gray, vmin=vmin, vmax=vmax)

# equal bins face
regular_values = np.linspace(0, 256, n_clusters + 1)
regular_labels = np.searchsorted(regular_values, face) - 1
regular_values = .5 * (regular_values[1:] + regular_values[:-1])  # mean
regular_face = np.choose(regular_labels.ravel(), regular_values, mode="clip")
regular_face.shape = face.shape
plt.figure(3, figsize=(3, 2.2))
plt.imshow(regular_face, cmap=plt.cm.gray, vmin=vmin, vmax=vmax)

# histogram
plt.figure(4, figsize=(3, 2.2))
plt.clf()
plt.axes([.01, .01, .98, .98])
plt.hist(X, bins=256, color='.5', edgecolor='.5')
plt.yticks(())
plt.xticks(regular_values)
values = np.sort(values)
for center_1, center_2 in zip(values[:-1], values[1:]):
    plt.axvline(.5 * (center_1 + center_2), color='b')

for center_1, center_2 in zip(regular_values[:-1], regular_values[1:]):
    plt.axvline(.5 * (center_1 + center_2), color='b', linestyle='--')

plt.show()
```

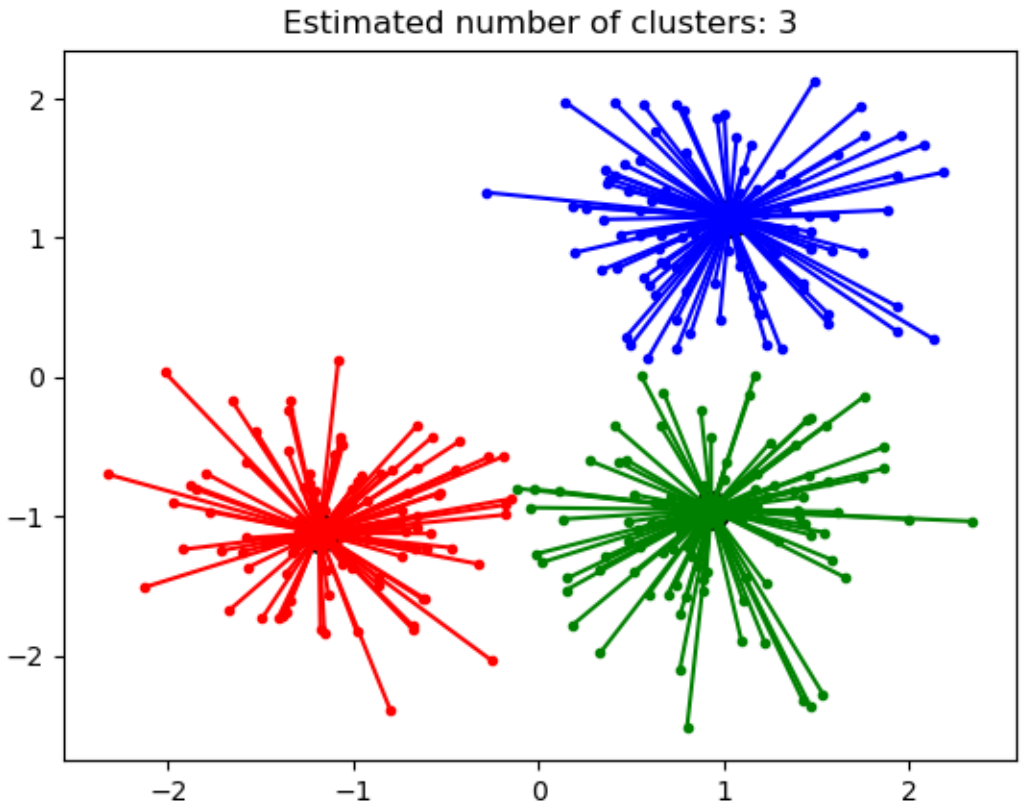
Total running time of the script: (0 minutes 2.997 seconds)

Estimated memory usage: 130 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.7 Demo of affinity propagation clustering algorithm

Reference: Brendan J. Frey and Delbert Dueck, "Clustering by Passing Messages Between Data Points", Science Feb. 2007



Out:

```
Estimated number of clusters: 3
Homogeneity: 0.872
Completeness: 0.872
V-measure: 0.872
Adjusted Rand Index: 0.912
Adjusted Mutual Information: 0.871
Silhouette Coefficient: 0.753
```

```
print(__doc__)

from sklearn.cluster import AffinityPropagation
from sklearn import metrics
from sklearn.datasets import make_blobs

# #############################################################################
# Generate sample data
centers = [[1, 1], [-1, -1], [1, -1]]
X, labels_true = make_blobs(n_samples=300, centers=centers, cluster_std=0.5,
                            random_state=0)

# #############################################################################
# Compute Affinity Propagation
af = AffinityPropagation(preference=-50).fit(X)
cluster_centers_indices = af.cluster_centers_indices_
labels = af.labels_

n_clusters_ = len(cluster_centers_indices)

print('Estimated number of clusters: %d' % n_clusters_)
print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels_true, labels))
print("Completeness: %0.3f" % metrics.completeness_score(labels_true, labels))
print("V-measure: %0.3f" % metrics.v_measure_score(labels_true, labels))
print("Adjusted Rand Index: %0.3f"
      % metrics.adjusted_rand_score(labels_true, labels))
print("Adjusted Mutual Information: %0.3f"
      % metrics.adjusted_mutual_info_score(labels_true, labels))
print("Silhouette Coefficient: %0.3f"
      % metrics.silhouette_score(X, labels, metric='sqeuclidean'))

# #############################################################################
# Plot result
import matplotlib.pyplot as plt
from itertools import cycle

plt.close('all')
plt.figure(1)
plt.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    class_members = labels == k
    cluster_center = X[cluster_centers_indices[k]]
    plt.plot(X[class_members, 0], X[class_members, 1], col + '.')
    plt.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
             markeredgecolor='k', markersize=14)
    for x in X[class_members]:
        plt.plot([cluster_center[0], x[0]], [cluster_center[1], x[1]], col)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

Total running time of the script: (0 minutes 0.665 seconds)

Estimated memory usage: 8 MB

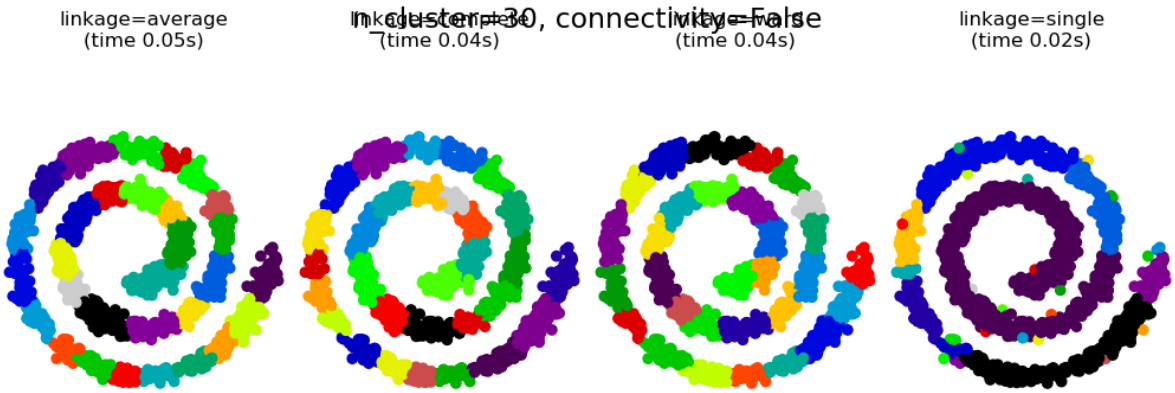
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.8 Agglomerative clustering with and without structure

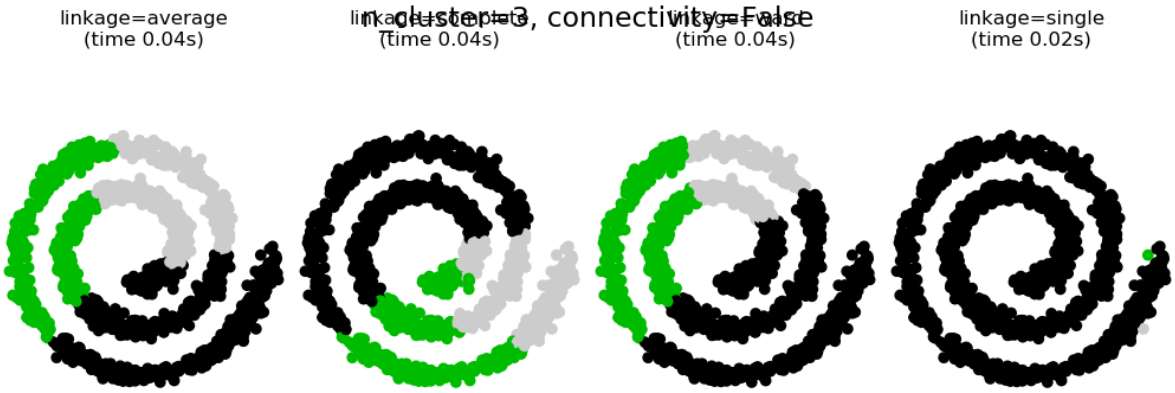
This example shows the effect of imposing a connectivity graph to capture local structure in the data. The graph is simply the graph of 20 nearest neighbors.

Two consequences of imposing a connectivity can be seen. First clustering with a connectivity matrix is much faster.

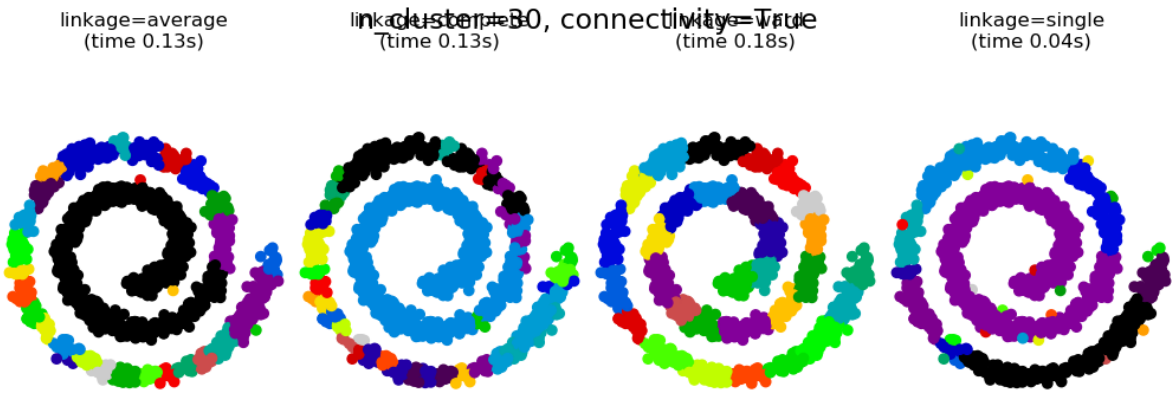
Second, when using a connectivity matrix, single, average and complete linkage are unstable and tend to create a few clusters that grow very quickly. Indeed, average and complete linkage fight this percolation behavior by considering all the distances between two clusters when merging them (while single linkage exaggerates the behaviour by considering only the shortest distance between clusters). The connectivity graph breaks this mechanism for average and complete linkage, making them resemble the more brittle single linkage. This effect is more pronounced for very sparse graphs (try decreasing the number of neighbors in kneighbors_graph) and with complete linkage. In particular, having a very small number of neighbors in the graph, imposes a geometry that is close to that of single linkage, which is well known to have this percolation instability.

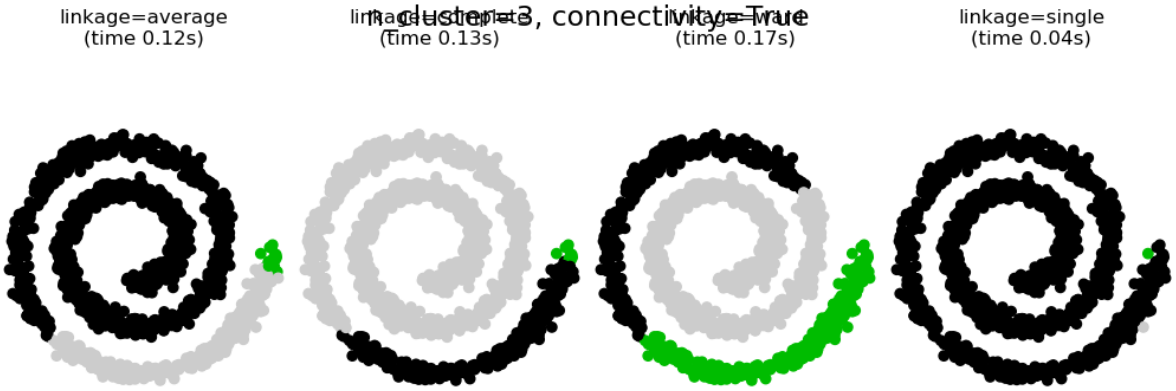


-



-






```
# Authors: Gael Varoquaux, Nelle Varoquaux
# License: BSD 3 clause

import time
import matplotlib.pyplot as plt
import numpy as np

from sklearn.cluster import AgglomerativeClustering
from sklearn.neighbors import kneighbors_graph

# Generate sample data
n_samples = 1500
np.random.seed(0)
t = 1.5 * np.pi * (1 + 3 * np.random.rand(1, n_samples))
x = t * np.cos(t)
y = t * np.sin(t)


X = np.concatenate((x, y))
X += .7 * np.random.randn(2, n_samples)
X = X.T

# Create a graph capturing local connectivity. Larger number of neighbors
# will give more homogeneous clusters to the cost of computation
```

(continues on next page)

(continued from previous page)

```
# time. A very large number of neighbors gives more evenly distributed
# cluster sizes, but may not impose the local manifold structure of
# the data
knn_graph = kneighbors_graph(X, 30, include_self=False)

for connectivity in (None, knn_graph):
    for n_clusters in (30, 3):
        plt.figure(figsize=(10, 4))
        for index, linkage in enumerate(('average',
                                         'complete',
                                         'ward',
                                         'single')):
            plt.subplot(1, 4, index + 1)
            model = AgglomerativeClustering(linkage=linkage,
                                            connectivity=connectivity,
                                            n_clusters=n_clusters)
            t0 = time.time()
            model.fit(X)
            elapsed_time = time.time() - t0
            plt.scatter(X[:, 0], X[:, 1], c=model.labels_,
                        cmap=plt.cm.nipy_spectral)
            plt.title('linkage=%s\n(time %.2fs)' % (linkage, elapsed_time),
                      fontdict=dict(verticalalignment='top'))
            plt.axis('equal')
            plt.axis('off')

            plt.subplots_adjust(bottom=0, top=.89, wspace=0,
                                left=0, right=1)
            plt.suptitle('n_cluster=%i, connectivity=%r' %
                         (n_clusters, connectivity is not None), size=17)


plt.show()
```

Total running time of the script: (0 minutes 2.195 seconds)

Estimated memory usage: 11 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.9 Segmenting the picture of greek coins in regions

This example uses *Spectral clustering* on a graph created from voxel-to-voxel difference on an image to break this image into multiple partly-homogeneous regions.

This procedure (spectral clustering on an image) is an efficient approximate solution for finding normalized graph cuts.

There are two options to assign labels:

- with 'kmeans' spectral clustering will cluster samples in the embedding space using a kmeans algorithm
- whereas 'discrete' will iteratively search for the closest partition space to the embedding space.

```
print(__doc__)
```

(continues on next page)

(continued from previous page)

```
# Author: Gael Varoquaux <gael.varoquaux@normalesup.org>, Brian Cheung
# License: BSD 3 clause

import time

import numpy as np
from distutils.version import LooseVersion
from scipy.ndimage.filters import gaussian_filter
import matplotlib.pyplot as plt
import skimage
from skimage.data import coins
from skimage.transform import rescale

from sklearn.feature_extraction import image
from sklearn.cluster import spectral_clustering

# these were introduced in skimage-0.14
if LooseVersion(skimage.__version__) >= '0.14':
    rescale_params = {'anti_aliasing': False, 'multichannel': False}
else:
    rescale_params = {}

# load the coins as a numpy array
orig_coins = coins()

# Resize it to 20% of the original size to speed up the processing
# Applying a Gaussian filter for smoothing prior to down-scaling
# reduces aliasing artifacts.
smoothened_coins = gaussian_filter(orig_coins, sigma=2)
rescaled_coins = rescale(smoothened_coins, 0.2, mode="reflect",
                         **rescale_params)

# Convert the image into a graph with the value of the gradient on the
# edges.
graph = image.img_to_graph(rescaled_coins)

# Take a decreasing function of the gradient: an exponential
# The smaller beta is, the more independent the segmentation is of the
# actual image. For beta=1, the segmentation is close to a voronoi
beta = 10
eps = 1e-6
graph.data = np.exp(-beta * graph.data / graph.data.std()) + eps

# Apply spectral clustering (this step goes much faster if you have pyamg
# installed)
N_REGIONS = 25
```

Visualize the resulting regions

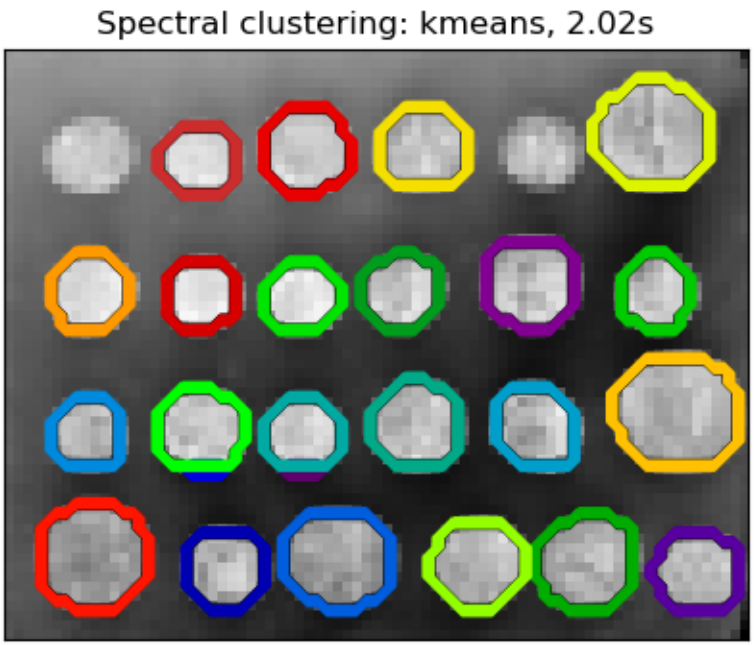
```
for assign_labels in ('kmeans', 'discretize'):
    t0 = time.time()
    labels = spectral_clustering(graph, n_clusters=N_REGIONS,
                                 assign_labels=assign_labels, random_state=42)
    t1 = time.time()
    labels = labels.reshape(rescaled_coins.shape)

    plt.figure(figsize=(5, 5))
```

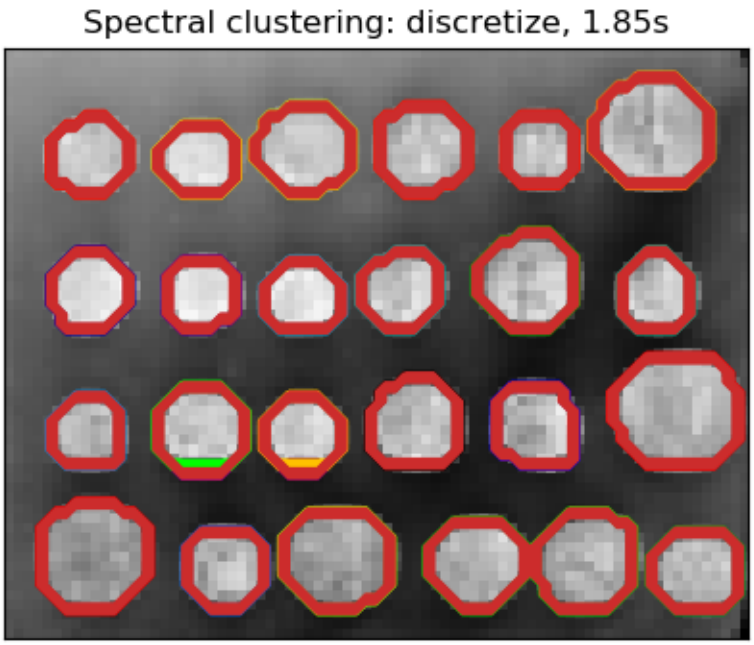
(continues on next page)

(continued from previous page)

```
    plt.imshow(rescaled_coins, cmap=plt.cm.gray)
    for l in range(N_REGIONS):
        plt.contour(labels == l,
                    colors=[plt.cm.nipy_spectral(l / float(N_REGIONS))])
    plt.xticks(())
    plt.yticks(())
    title = 'Spectral clustering: %s, %.2fs' % (assign_labels, (t1 - t0))
    print(title)
    plt.title(title)
plt.show()
```



-



-

Out:

```
Spectral clustering: kmeans, 2.02s
Spectral clustering: discretize, 1.85s
```

Total running time of the script: (0 minutes 4.687 seconds)

Estimated memory usage: 9 MB

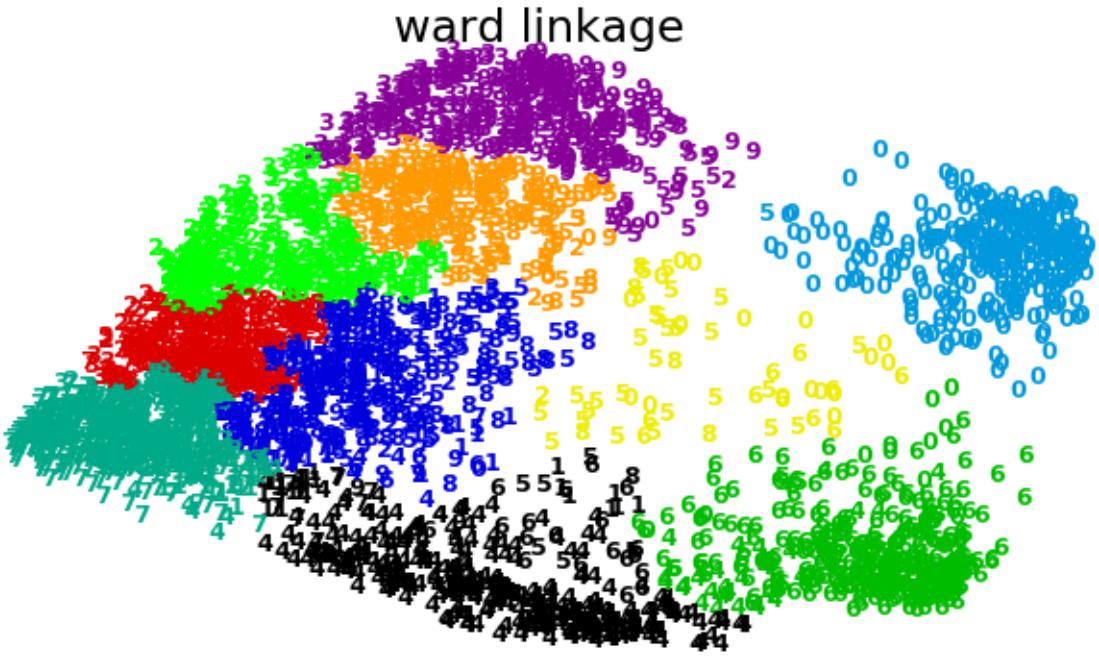
Note: Click *here* to download the full example code or to run this example in your browser via Binder

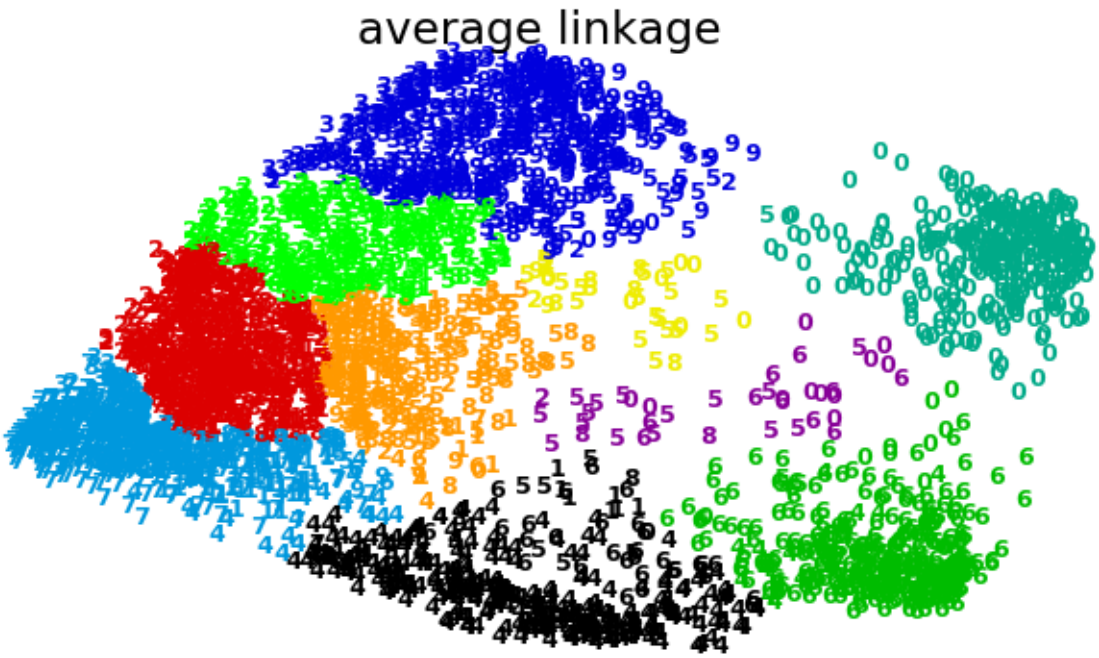
6.5.10 Various Agglomerative Clustering on a 2D embedding of digits

An illustration of various linkage option for agglomerative clustering on a 2D embedding of the digits dataset.

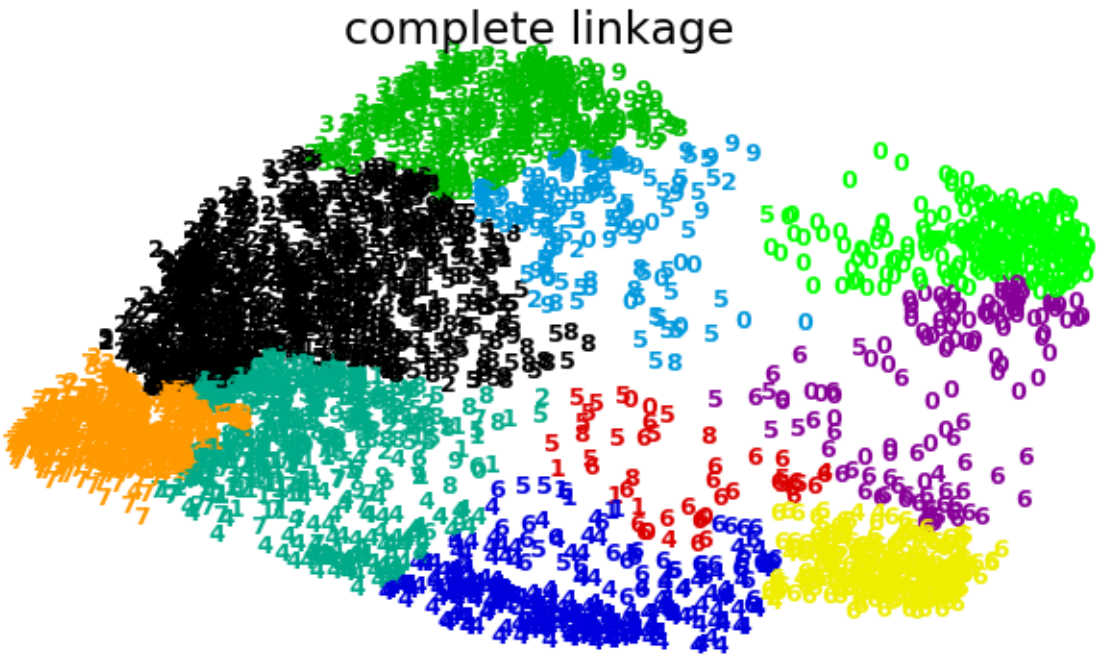
The goal of this example is to show intuitively how the metrics behave, and not to find good clusters for the digits. This is why the example works on a 2D embedding.

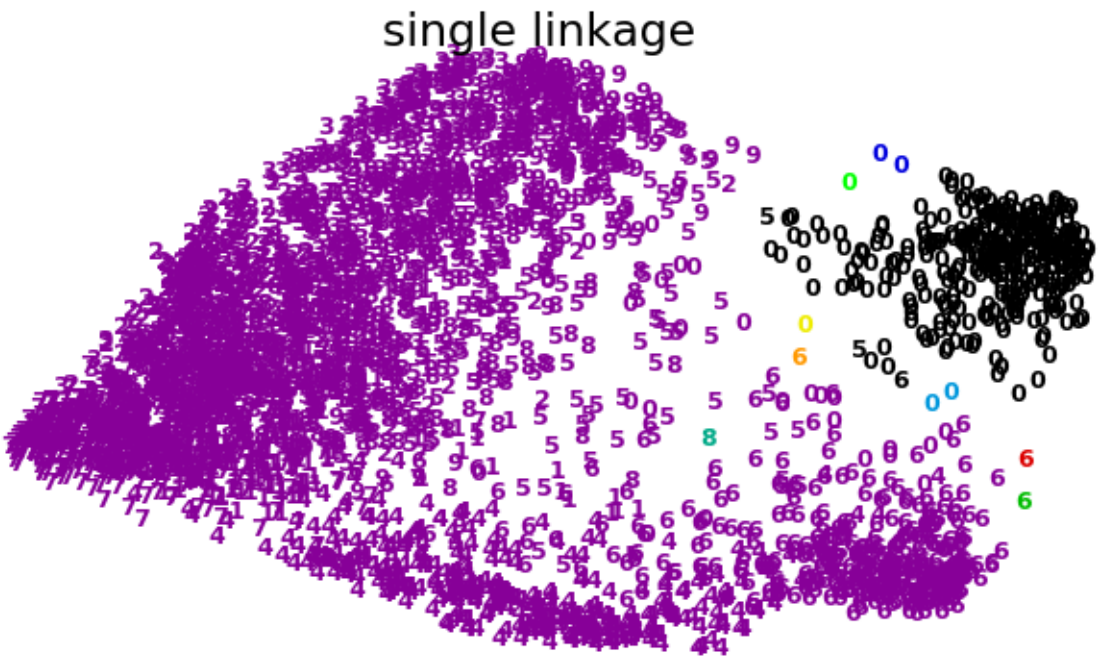
What this example shows us is the behavior "rich getting richer" of agglomerative clustering that tends to create uneven cluster sizes. This behavior is pronounced for the average linkage strategy, that ends up with a couple of singleton clusters, while in the case of single linkage we get a single central cluster with all other clusters being drawn from noise points around the fringes.


ward linkage

- 
average linkage

-



- 

Out:

```
Computing embedding
Done.
```

(continues on next page)

(continued from previous page)

```
ward :  0.31s
average :       0.30s
complete :      0.29s
single :        0.21s
```

```
# Authors: Gael Varoquaux
# License: BSD 3 clause (C) INRIA 2014

print(__doc__)
from time import time

import numpy as np
from scipy import ndimage
from matplotlib import pyplot as plt

from sklearn import manifold, datasets

X, y = datasets.load_digits(return_X_y=True)
n_samples, n_features = X.shape

np.random.seed(0)

def nudge_images(X, y):
    # Having a larger dataset shows more clearly the behavior of the
    # methods, but we multiply the size of the dataset only by 2, as the
    # cost of the hierarchical clustering methods are strongly
    # super-linear in n_samples
    shift = lambda x: ndimage.shift(x.reshape((8, 8)),
                                    .3 * np.random.normal(size=2),
                                    mode='constant',
                                    ).ravel()
    X = np.concatenate([X, np.apply_along_axis(shift, 1, X)])
    Y = np.concatenate([y, y], axis=0)
    return X, Y


X, y = nudge_images(X, y)


#----------------------------------------------------------------------
# Visualize the clustering
def plot_clustering(X_red, labels, title=None):
    x_min, x_max = np.min(X_red, axis=0), np.max(X_red, axis=0)
    X_red = (X_red - x_min) / (x_max - x_min)

    plt.figure(figsize=(6, 4))
    for i in range(X_red.shape[0]):
        plt.text(X_red[i, 0], X_red[i, 1], str(y[i]),
                 color=plt.cm.nipy_spectral(labels[i] / 10.),
                 fontdict={'weight': 'bold', 'size': 9})
```

(continues on next page)

(continued from previous page)

```
    plt.xticks([])
    plt.yticks([])
    if title is not None:
        plt.title(title, size=17)
    plt.axis('off')
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])

#----------------------------------------------------------------------
# 2D embedding of the digits dataset
print("Computing embedding")
X_red = manifold.SpectralEmbedding(n_components=2).fit_transform(X)
print("Done.")

from sklearn.cluster import AgglomerativeClustering

for linkage in ('ward', 'average', 'complete', 'single'):
    clustering = AgglomerativeClustering(linkage=linkage, n_clusters=10)
    t0 = time()
    clustering.fit(X_red)
    print("%s :\t%.2fs" % (linkage, time() - t0))

    plot_clustering(X_red, clustering.labels_, "%s linkage" % linkage)


plt.show()
```

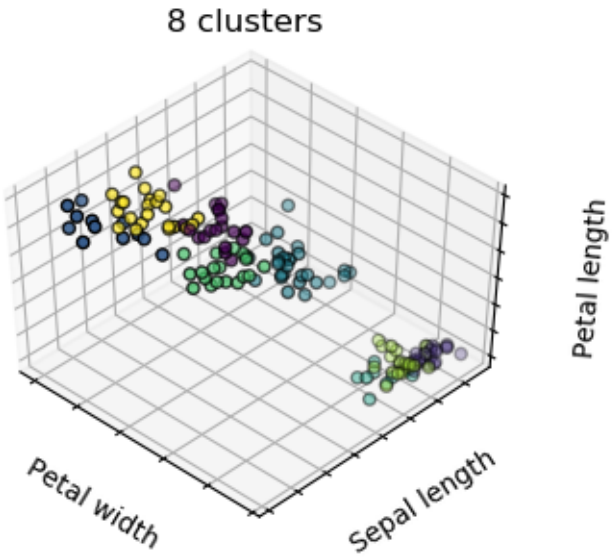
Total running time of the script: (0 minutes 25.325 seconds)

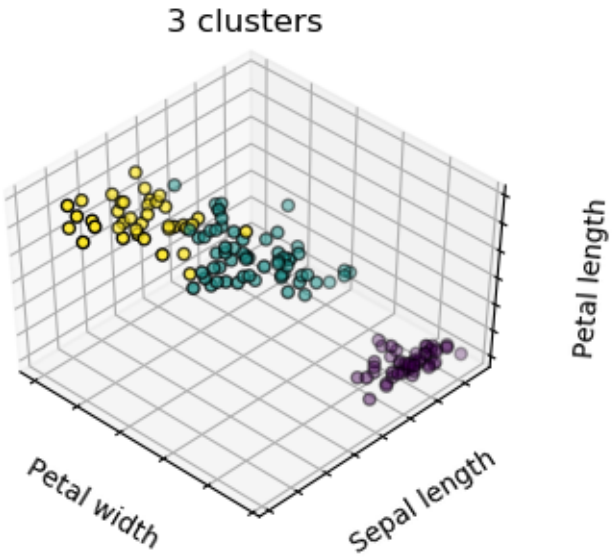
Estimated memory usage: 152 MB

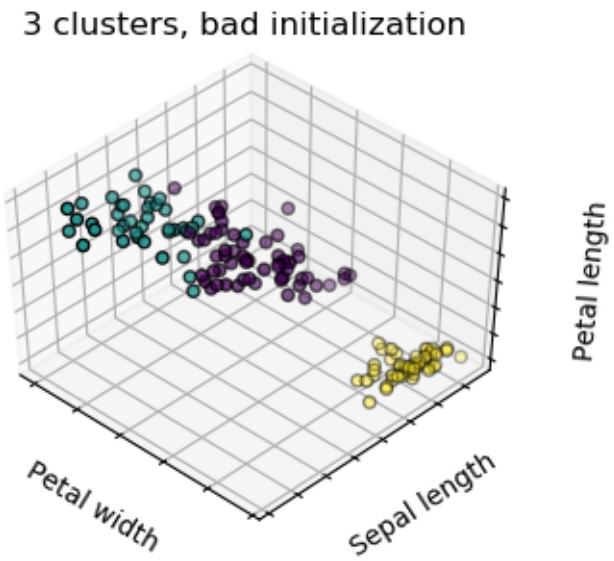
Note: Click *here* to download the full example code or to run this example in your browser via Binder

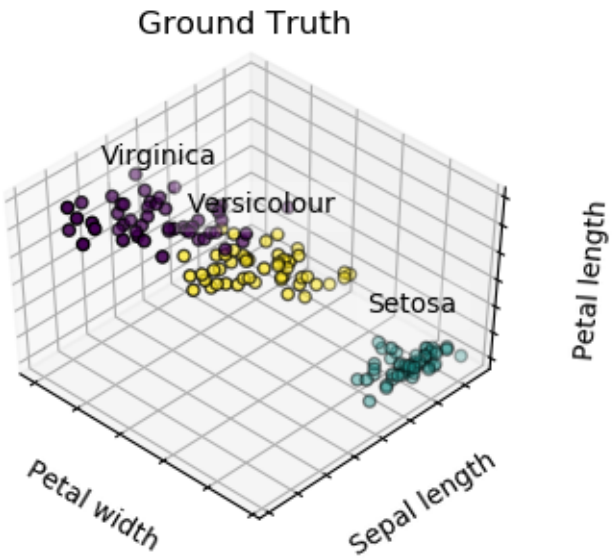
6.5.11 K-means Clustering

The plots display firstly what a K-means algorithm would yield using three clusters. It is then shown what the effect of a bad initialization is on the classification process: By setting n_init to only 1 (default is 10), the amount of times that the algorithm will be run with different centroid seeds is reduced. The next plot displays what using eight clusters would deliver and finally the ground truth.

- 
8 clusters
Petal width
Sepal length
Petal length

- 
3 clusters
Petal width
Sepal length
Petal length



- 

-

```
print(__doc__)


# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
# Though the following import is not directly being used, it is required
# for 3D projection to work
from mpl_toolkits.mplot3d import Axes3D

from sklearn.cluster import KMeans
from sklearn import datasets

np.random.seed(5)
```

(continues on next page)

(continued from previous page)

```
iris = datasets.load_iris()
X = iris.data
y = iris.target

estimators = [('k_means_iris_8', KMeans(n_clusters=8)),
              ('k_means_iris_3', KMeans(n_clusters=3)),
              ('k_means_iris_bad_init', KMeans(n_clusters=3, n_init=1,
                                               init='random'))]

fignum = 1
titles = ['8 clusters', '3 clusters', '3 clusters, bad initialization']
for name, est in estimators:
    fig = plt.figure(fignum, figsize=(4, 3))
    ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)
    est.fit(X)
    labels = est.labels_

    ax.scatter(X[:, 3], X[:, 0], X[:, 2],
               c=labels.astype(np.float), edgecolor='k')

    ax.w_xaxis.set_ticklabels([])
    ax.w_yaxis.set_ticklabels([])
    ax.w_zaxis.set_ticklabels([])
    ax.set_xlabel('Petal width')
    ax.set_ylabel('Sepal length')
    ax.set_zlabel('Petal length')
    ax.set_title(titles[fignum - 1])
    ax.dist = 12
    fignum = fignum + 1

# Plot the ground truth
fig = plt.figure(fignum, figsize=(4, 3))
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

for name, label in [('Setosa', 0),
                    ('Versicolour', 1),
                    ('Virginica', 2)]:
    ax.text3D(X[y == label, 3].mean(),
              X[y == label, 0].mean(),
              X[y == label, 2].mean() + 2, name,
              horizontalalignment='center',
              bbox=dict(alpha=.2, edgecolor='w', facecolor='w'))
# Reorder the labels to have colors matching the cluster results
y = np.choose(y, [1, 2, 0]).astype(np.float)
ax.scatter(X[:, 3], X[:, 0], X[:, 2], c=y, edgecolor='k')

ax.w_xaxis.set_ticklabels([])
ax.w_yaxis.set_ticklabels([])
ax.w_zaxis.set_ticklabels([])
ax.set_xlabel('Petal width')
ax.set_ylabel('Sepal length')
ax.set_zlabel('Petal length')
ax.set_title('Ground Truth')
ax.dist = 12

fig.show()
```

Total running time of the script: (0 minutes 0.629 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.12 Spectral clustering for image segmentation

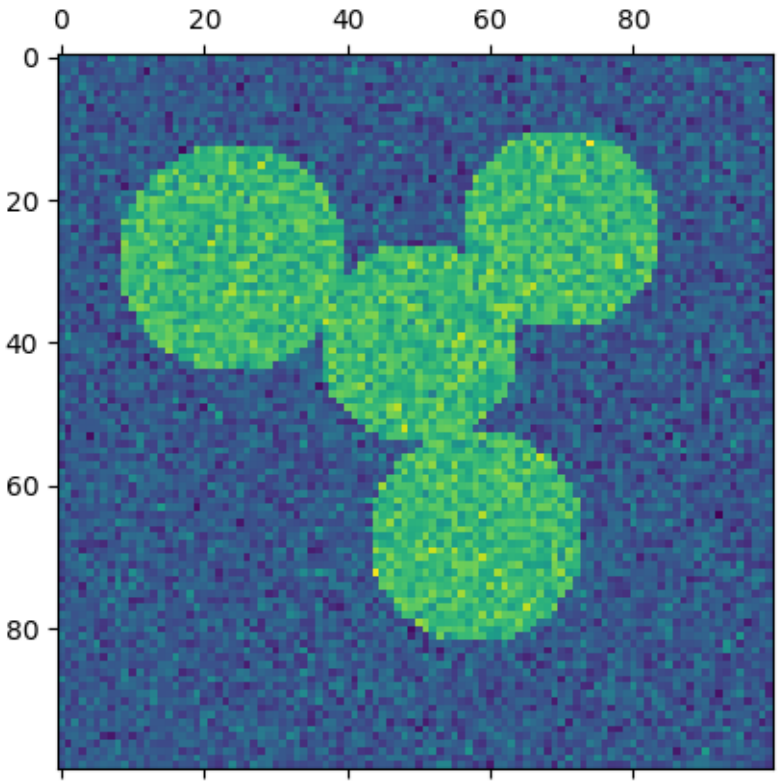
In this example, an image with connected circles is generated and spectral clustering is used to separate the circles.

In these settings, the *Spectral clustering* approach solves the problem know as 'normalized graph cuts': the image is seen as a graph of connected voxels, and the spectral clustering algorithm amounts to choosing graph cuts defining regions while minimizing the ratio of the gradient along the cut, and the volume of the region.

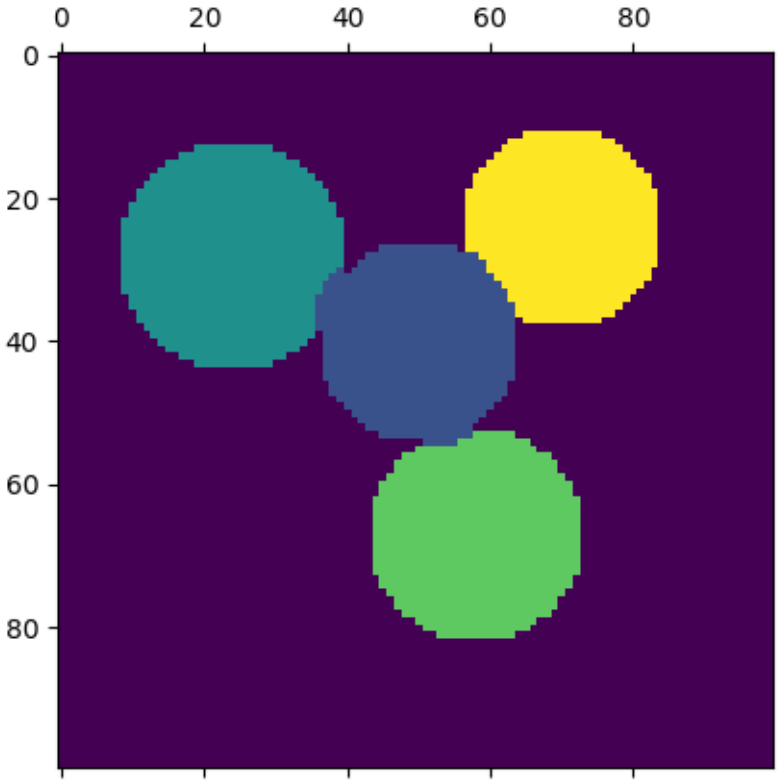
As the algorithm tries to balance the volume (ie balance the region sizes), if we take circles with different sizes, the segmentation fails.

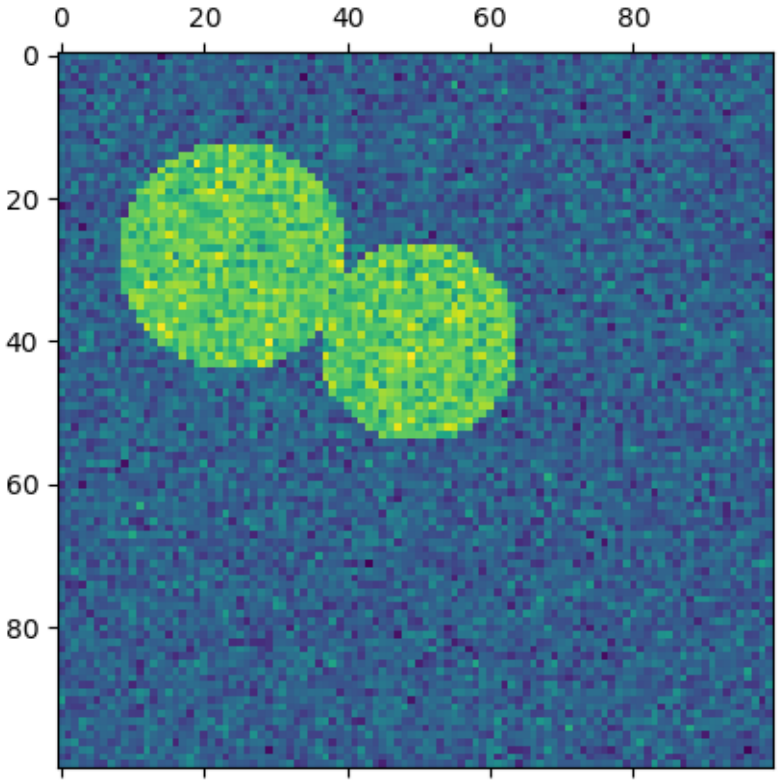
In addition, as there is no useful information in the intensity of the image, or its gradient, we choose to perform the spectral clustering on a graph that is only weakly informed by the gradient. This is close to performing a Voronoi partition of the graph.

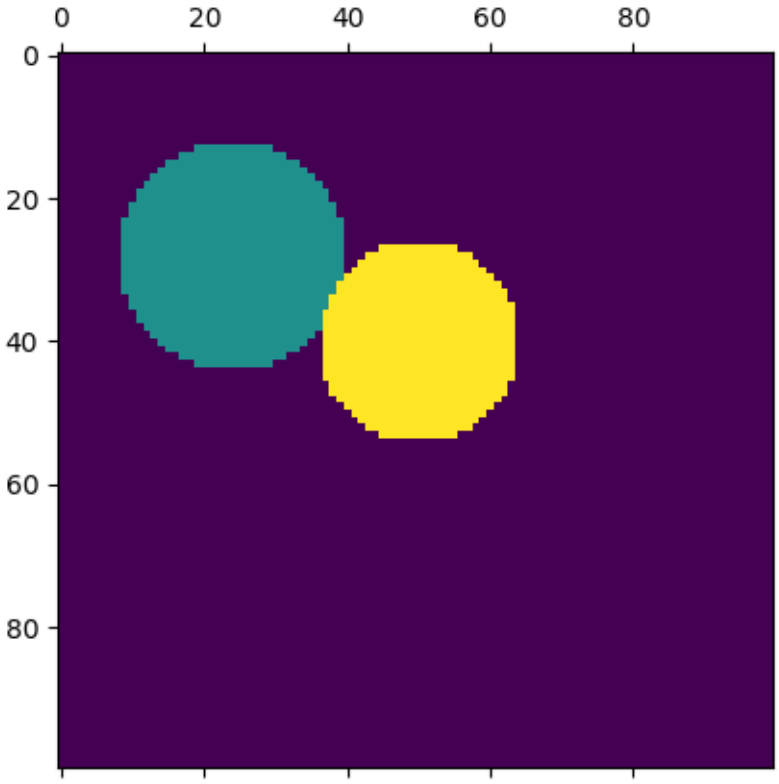
In addition, we use the mask of the objects to restrict the graph to the outline of the objects. In this example, we are interested in separating the objects one from the other, and not from the background.



•


0
20
40
60
80
0
20
40
60
80


0
20
40
60
80
0
20
40
60
80



-

```
print(__doc__)

# Authors:  Emmanuelle Gouillart <emmanuelle.gouillart@normalesup.org>
#           Gael Varoquaux <gael.varoquaux@normalesup.org>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.feature_extraction import image
from sklearn.cluster import spectral_clustering

l = 100
x, y = np.indices((l, l))

center1 = (28, 24)
center2 = (40, 50)
center3 = (67, 58)
center4 = (24, 70)

radius1, radius2, radius3, radius4 = 16, 14, 15, 14

circle1 = (x - center1[0]) ** 2 + (y - center1[1]) ** 2 < radius1 ** 2
circle2 = (x - center2[0]) ** 2 + (y - center2[1]) ** 2 < radius2 ** 2
circle3 = (x - center3[0]) ** 2 + (y - center3[1]) ** 2 < radius3 ** 2
```

(continues on next page)

(continued from previous page)

```
circle4 = (x - center4[0]) ** 2 + (y - center4[1]) ** 2 < radius4 ** 2

# ###############################################################################
# 4 circles
img = circle1 + circle2 + circle3 + circle4

# We use a mask that limits to the foreground: the problem that we are
# interested in here is not separating the objects from the background,
# but separating them one from the other.
mask = img.astype(bool)

img = img.astype(float)
img += 1 + 0.2 * np.random.randn(*img.shape)

# Convert the image into a graph with the value of the gradient on the
# edges.
graph = image.img_to_graph(img, mask=mask)

# Take a decreasing function of the gradient: we take it weakly
# dependent from the gradient the segmentation is close to a voronoi
graph.data = np.exp(-graph.data / graph.data.std())

# Force the solver to be arpack, since amg is numerically
# unstable on this example
labels = spectral_clustering(graph, n_clusters=4, eigen_solver='arpack')
label_im = np.full(mask.shape, -1.)
label_im[mask] = labels

plt.matshow(img)
plt.matshow(label_im)

# ###############################################################################
# 2 circles
img = circle1 + circle2
mask = img.astype(bool)
img = img.astype(float)

img += 1 + 0.2 * np.random.randn(*img.shape)

graph = image.img_to_graph(img, mask=mask)
graph.data = np.exp(-graph.data / graph.data.std())

labels = spectral_clustering(graph, n_clusters=2, eigen_solver='arpack')
label_im = np.full(mask.shape, -1.)
label_im[mask] = labels

plt.matshow(img)
plt.matshow(label_im)

plt.show()
```

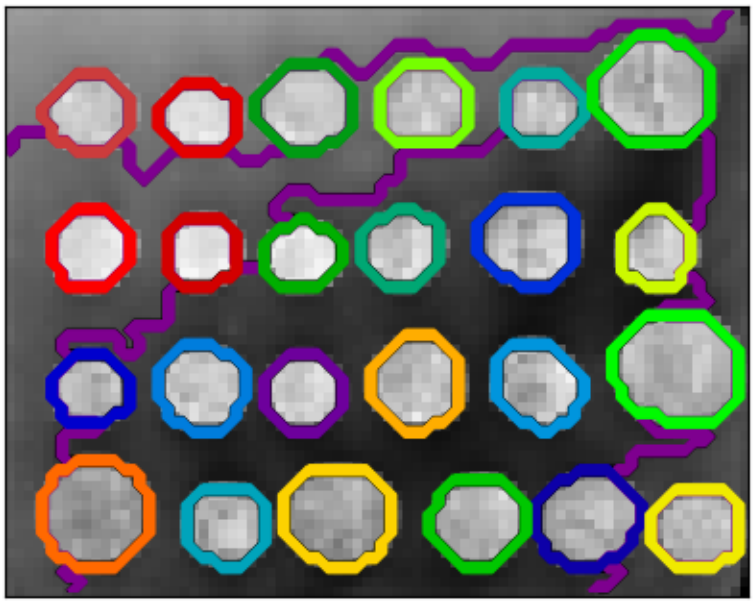
Total running time of the script: (0 minutes 0.875 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.13 A demo of structured Ward hierarchical clustering on an image of coins

Compute the segmentation of a 2D image with Ward hierarchical clustering. The clustering is spatially constrained in order for each segmented region to be in one piece.



Out:

```
Compute structured hierarchical clustering...
Elapsed time:  0.2965068817138672
Number of pixels:  4697
Number of clusters:  27
```

```
# Author : Vincent Michel, 2010
#          Alexandre Gramfort, 2011
# License: BSD 3 clause

print(__doc__)

import time as time
```

(continues on next page)

(continued from previous page)

```
import numpy as np
from distutils.version import LooseVersion
from scipy.ndimage.filters import gaussian_filter

import matplotlib.pyplot as plt

import skimage
from skimage.data import coins
from skimage.transform import rescale

from sklearn.feature_extraction.image import grid_to_graph
from sklearn.cluster import AgglomerativeClustering

# these were introduced in skimage-0.14
if LooseVersion(skimage.__version__) >= '0.14':
    rescale_params = {'anti_aliasing': False, 'multichannel': False}
else:
    rescale_params = {}

# #############################################################################
# Generate data
orig_coins = coins()

# Resize it to 20% of the original size to speed up the processing
# Applying a Gaussian filter for smoothing prior to down-scaling
# reduces aliasing artifacts.
smoothened_coins = gaussian_filter(orig_coins, sigma=2)
rescaled_coins = rescale(smoothened_coins, 0.2, mode="reflect",
                         **rescale_params)

X = np.reshape(rescaled_coins, (-1, 1))

# #############################################################################
# Define the structure A of the data. Pixels connected to their neighbors.
connectivity = grid_to_graph(*rescaled_coins.shape)

# #############################################################################
# Compute clustering
print("Compute structured hierarchical clustering...")
st = time.time()
n_clusters = 27  # number of regions
ward = AgglomerativeClustering(n_clusters=n_clusters, linkage='ward',
                               connectivity=connectivity)
ward.fit(X)
label = np.reshape(ward.labels_, rescaled_coins.shape)
print("Elapsed time: ", time.time() - st)
print("Number of pixels: ", label.size)
print("Number of clusters: ", np.unique(label).size)

# #############################################################################
# Plot the results on an image
plt.figure(figsize=(5, 5))
plt.imshow(rescaled_coins, cmap=plt.cm.gray)
for l in range(n_clusters):
    plt.contour(label == l,
                colors=[plt.cm.nipy_spectral(l / float(n_clusters)), ])
```

(continues on next page)

(continued from previous page)

```
plt.xticks(())
plt.yticks(())
plt.show()
```

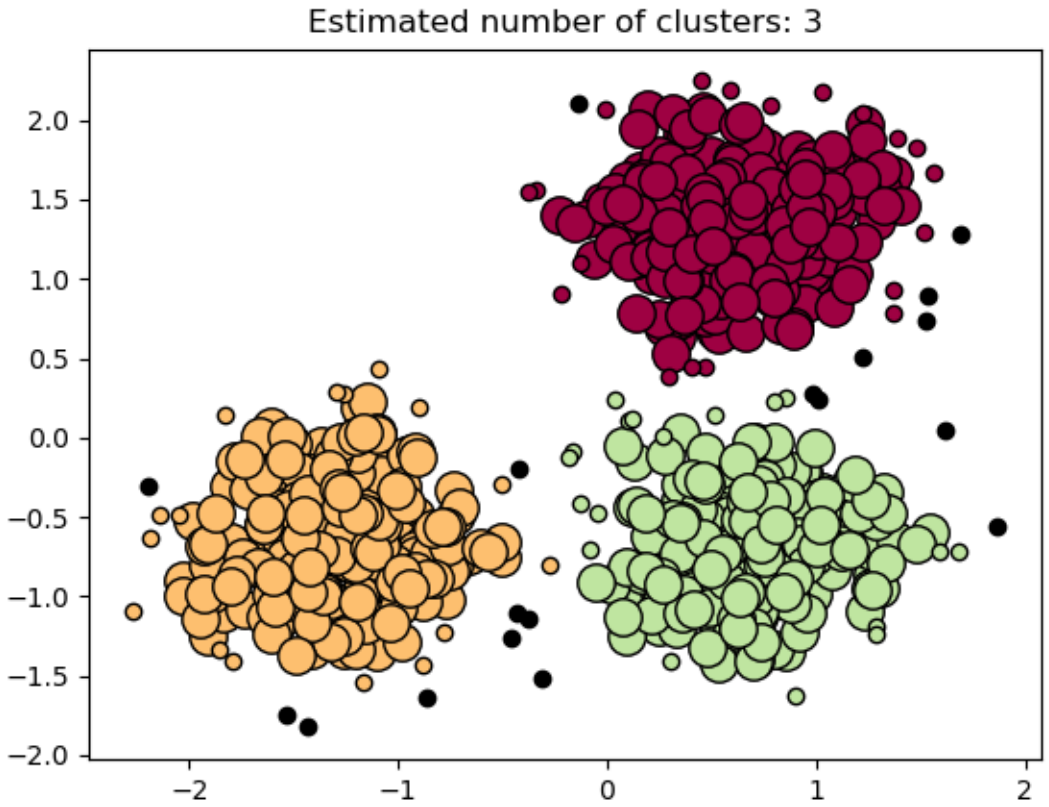
Total running time of the script: (0 minutes 0.744 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.14 Demo of DBSCAN clustering algorithm

Finds core samples of high density and expands clusters from them.



Out:

```
Estimated number of clusters: 3
Estimated number of noise points: 18
Homogeneity: 0.953
Completeness: 0.883
V-measure: 0.917
Adjusted Rand Index: 0.952
```

(continues on next page)

(continued from previous page)

```
Adjusted Mutual Information: 0.916
Silhouette Coefficient: 0.626
```

```
print(__doc__)

import numpy as np

from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler


# #############################################################################
# Generate sample data
centers = [[1, 1], [-1, -1], [1, -1]]
X, labels_true = make_blobs(n_samples=750, centers=centers, cluster_std=0.4,
                            random_state=0)

X = StandardScaler().fit_transform(X)

# #############################################################################
# Compute DBSCAN
db = DBSCAN(eps=0.3, min_samples=10).fit(X)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)

print('Estimated number of clusters: %d' % n_clusters_)
print('Estimated number of noise points: %d' % n_noise_)
print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels_true, labels))
print("Completeness: %0.3f" % metrics.completeness_score(labels_true, labels))
print("V-measure: %0.3f" % metrics.v_measure_score(labels_true, labels))
print("Adjusted Rand Index: %0.3f"
      % metrics.adjusted_rand_score(labels_true, labels))
print("Adjusted Mutual Information: %0.3f"
      % metrics.adjusted_mutual_info_score(labels_true, labels))
print("Silhouette Coefficient: %0.3f"
      % metrics.silhouette_score(X, labels))

# #############################################################################
# Plot result
import matplotlib.pyplot as plt

# Black removed and is used for noise instead.
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
```

(continues on next page)

(continued from previous page)

```
          for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    xy = X[class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=14)

    xy = X[class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=6)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

Total running time of the script: (0 minutes 0.431 seconds)

Estimated memory usage: 9 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.15 Color Quantization using K-Means

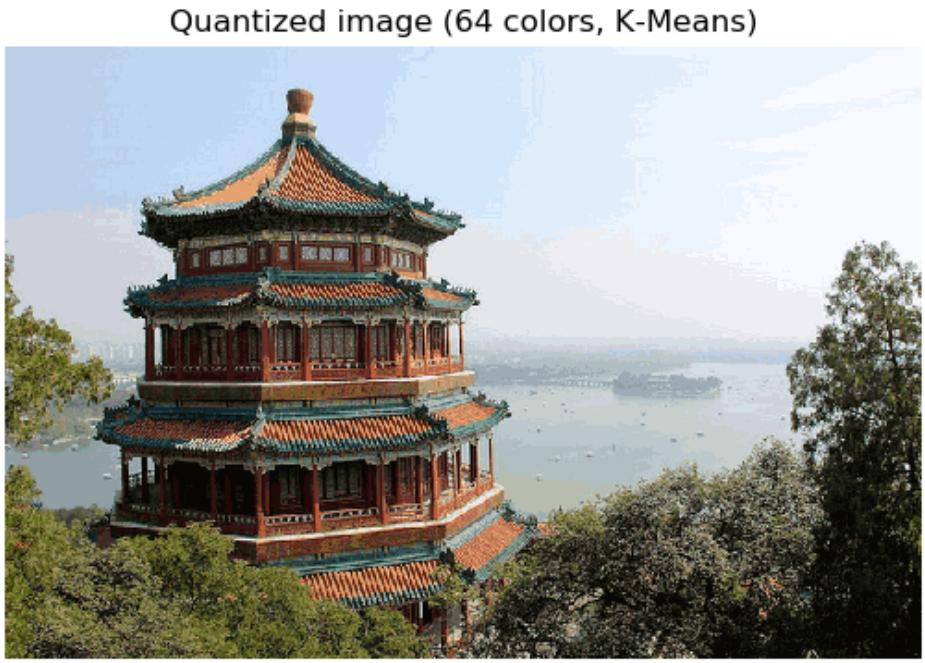
Performs a pixel-wise Vector Quantization (VQ) of an image of the summer palace (China), reducing the number of colors required to show the image from 96,615 unique colors to 64, while preserving the overall appearance quality.

In this example, pixels are represented in a 3D-space and K-means is used to find 64 color clusters. In the image processing literature, the codebook obtained from K-means (the cluster centers) is called the color palette. Using a single byte, up to 256 colors can be addressed, whereas an RGB encoding requires 3 bytes per pixel. The GIF file format, for example, uses such a palette.

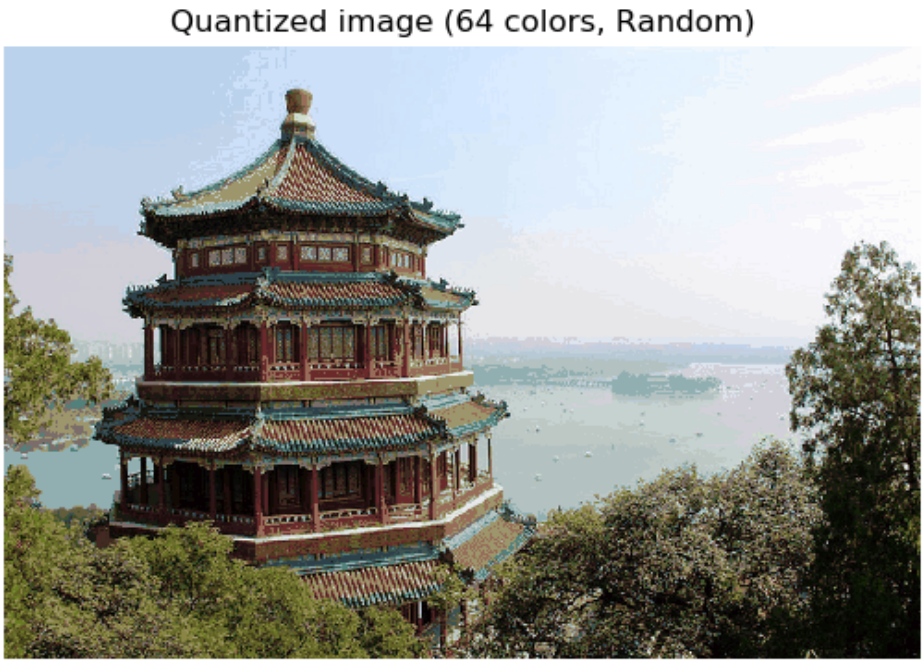
For comparison, a quantized image using a random codebook (colors picked up randomly) is also shown.


Original image (96,615 colors)

-


Quantized image (64 colors, K-Means)

-



-

Out:

```
Fitting model on a small sub-sample of the data
done in 0.356s.
Predicting color indices on the full image (k-means)
done in 0.149s.
Predicting color indices on the full image (random)
done in 0.152s.
```

```
# Authors: Robert Layton <robertlayton@gmail.com>
#          Olivier Grisel <olivier.grisel@ensta.org>
#          Mathieu Blondel <mathieu@mblondel.org>
#
# License: BSD 3 clause

print(__doc__)
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import pairwise_distances_argmin
from sklearn.datasets import load_sample_image
from sklearn.utils import shuffle
```

(continues on next page)

(continued from previous page)

```
from time import time

n_colors = 64

# Load the Summer Palace photo
china = load_sample_image("china.jpg")

# Convert to floats instead of the default 8 bits integer coding. Dividing by
# 255 is important so that plt.imshow behaves works well on float data (need to
# be in the range [0-1])
china = np.array(china, dtype=np.float64) / 255

# Load Image and transform to a 2D numpy array.
w, h, d = original_shape = tuple(china.shape)
assert d == 3
image_array = np.reshape(china, (w * h, d))

print("Fitting model on a small sub-sample of the data")
t0 = time()
image_array_sample = shuffle(image_array, random_state=0)[:1000]
kmeans = KMeans(n_clusters=n_colors, random_state=0).fit(image_array_sample)
print("done in %0.3fs." % (time() - t0))

# Get labels for all points
print("Predicting color indices on the full image (k-means)")
t0 = time()
labels = kmeans.predict(image_array)
print("done in %0.3fs." % (time() - t0))


codebook_random = shuffle(image_array, random_state=0)[:n_colors]
print("Predicting color indices on the full image (random)")
t0 = time()
labels_random = pairwise_distances_argmin(codebook_random,
                                          image_array,
                                          axis=0)
print("done in %0.3fs." % (time() - t0))


def recreate_image(codebook, labels, w, h):
    """Recreate the (compressed) image from the code book & labels"""
    d = codebook.shape[1]
    image = np.zeros((w, h, d))
    label_idx = 0
    for i in range(w):
        for j in range(h):
            image[i][j] = codebook[labels[label_idx]]
            label_idx += 1
    return image

# Display all results, alongside original image
plt.figure(1)
plt.clf()
plt.axis('off')
plt.title('Original image (96,615 colors)')
plt.imshow(china)
```

(continues on next page)

(continued from previous page)

```
plt.figure(2)
plt.clf()
plt.axis('off')
plt.title('Quantized image (64 colors, K-Means)')
plt.imshow(recreate_image(kmeans.cluster_centers_, labels, w, h))

plt.figure(3)
plt.clf()
plt.axis('off')
plt.title('Quantized image (64 colors, Random)')
plt.imshow(recreate_image(codebook_random, labels_random, w, h))
plt.show()
```

Total running time of the script: (0 minutes 1.769 seconds)

Estimated memory usage: 156 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

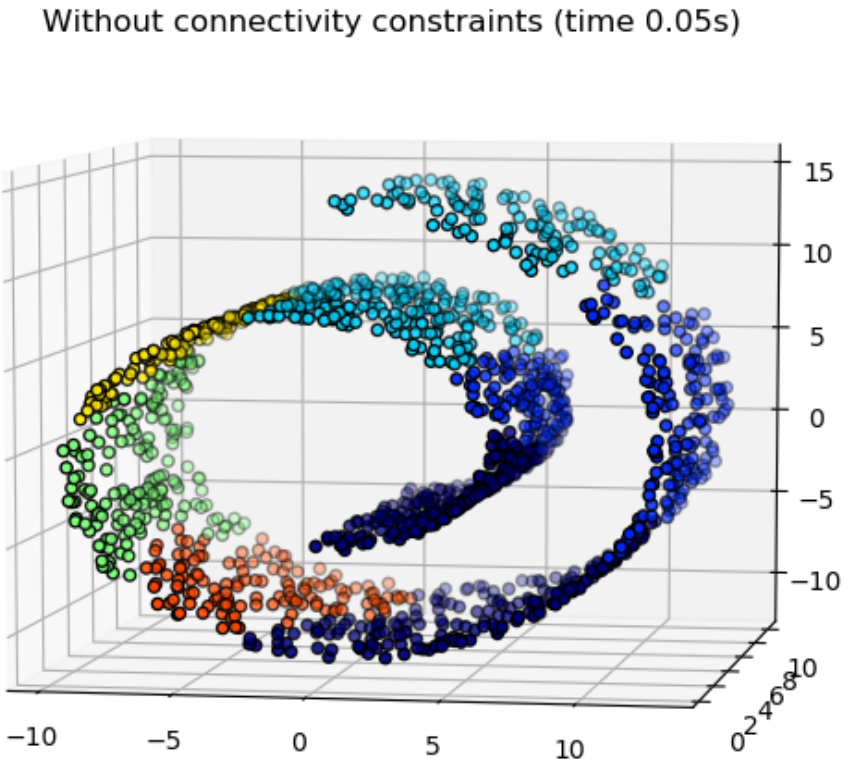
6.5.16 Hierarchical clustering: structured vs unstructured ward

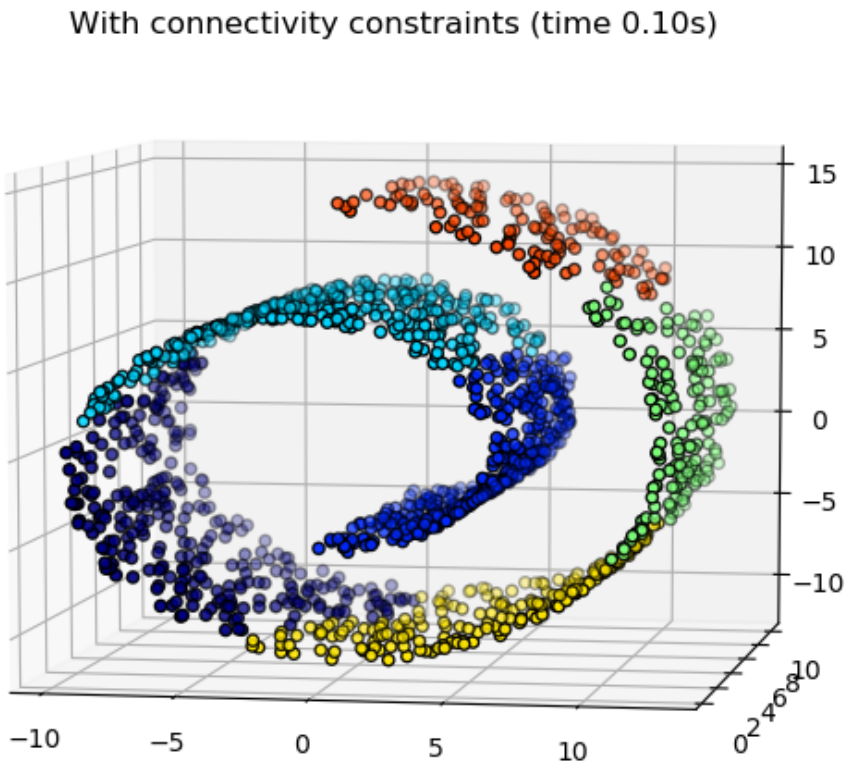
Example builds a swiss roll dataset and runs hierarchical clustering on their position.

For more information, see *Hierarchical clustering*.

In a first step, the hierarchical clustering is performed without connectivity constraints on the structure and is solely based on distance, whereas in a second step the clustering is restricted to the k-Nearest Neighbors graph: it's a hierarchical clustering with structure prior.

Some of the clusters learned without connectivity constraints do not respect the structure of the swiss roll and extend across different folds of the manifolds. On the opposite, when opposing connectivity constraints, the clusters form a nice parcellation of the swiss roll.


Without connectivity constraints (time 0.05s)
15
10
5
0
−5
−10
−10
−5
0
5
10
0
2
4
6
8
10



-

Out:

```
Compute unstructured hierarchical clustering...
Elapsed time: 0.05s
Number of points: 1500
Compute structured hierarchical clustering...
Elapsed time: 0.10s
Number of points: 1500
```

```
# Authors : Vincent Michel, 2010
#           Alexandre Gramfort, 2010
#           Gael Varoquaux, 2010
# License: BSD 3 clause

print(__doc__)

import time as time
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as p3
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets import make_swiss_roll
```

(continues on next page)

(continued from previous page)

```
# ############################################################################
# Generate data (swiss roll dataset)
n_samples = 1500
noise = 0.05
X, _ = make_swiss_roll(n_samples, noise)
# Make it thinner
X[:, 1] *= .5

# ############################################################################
# Compute clustering
print("Compute unstructured hierarchical clustering...")
st = time.time()
ward = AgglomerativeClustering(n_clusters=6, linkage='ward').fit(X)
elapsed_time = time.time() - st
label = ward.labels_
print("Elapsed time: %.2fs" % elapsed_time)
print("Number of points: %i" % label.size)

# ############################################################################
# Plot result
fig = plt.figure()
ax = p3.Axes3D(fig)
ax.view_init(7, -80)
for l in np.unique(label):
    ax.scatter(X[label == l, 0], X[label == l, 1], X[label == l, 2],
               color=plt.cm.jet(np.float(l) / np.max(label + 1)),
               s=20, edgecolor='k')
plt.title('Without connectivity constraints (time %.2fs)' % elapsed_time)


# ############################################################################
# Define the structure A of the data. Here a 10 nearest neighbors
from sklearn.neighbors import kneighbors_graph
connectivity = kneighbors_graph(X, n_neighbors=10, include_self=False)

# ############################################################################
# Compute clustering
print("Compute structured hierarchical clustering...")
st = time.time()
ward = AgglomerativeClustering(n_clusters=6, connectivity=connectivity,
                               linkage='ward').fit(X)
elapsed_time = time.time() - st
label = ward.labels_
print("Elapsed time: %.2fs" % elapsed_time)
print("Number of points: %i" % label.size)

# ############################################################################
# Plot result
fig = plt.figure()
ax = p3.Axes3D(fig)
ax.view_init(7, -80)
for l in np.unique(label):
    ax.scatter(X[label == l, 0], X[label == l, 1], X[label == l, 2],
               color=plt.cm.jet(float(l) / np.max(label + 1)),
               s=20, edgecolor='k')
plt.title('With connectivity constraints (time %.2fs)' % elapsed_time)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

Total running time of the script: (0 minutes 0.646 seconds)

Estimated memory usage: 25 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.17 Agglomerative clustering with different metrics

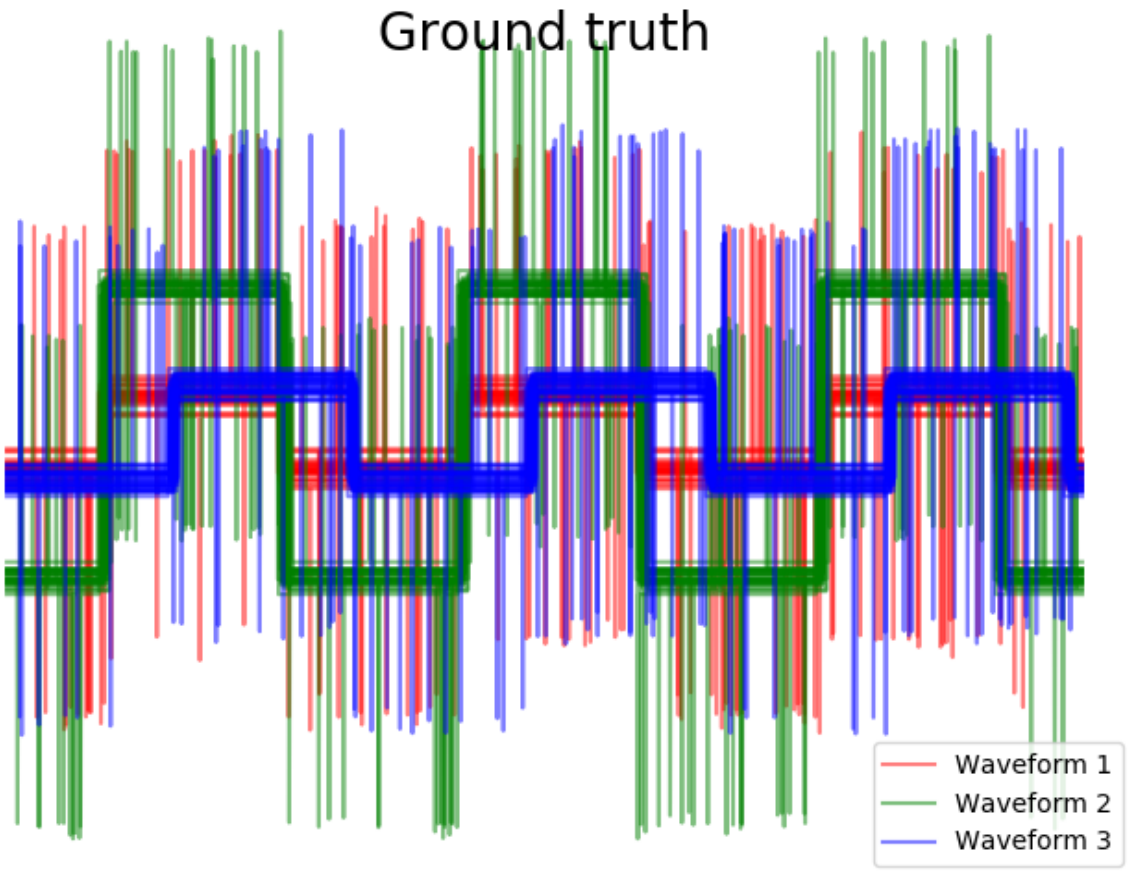
Demonstrates the effect of different metrics on the hierarchical clustering.

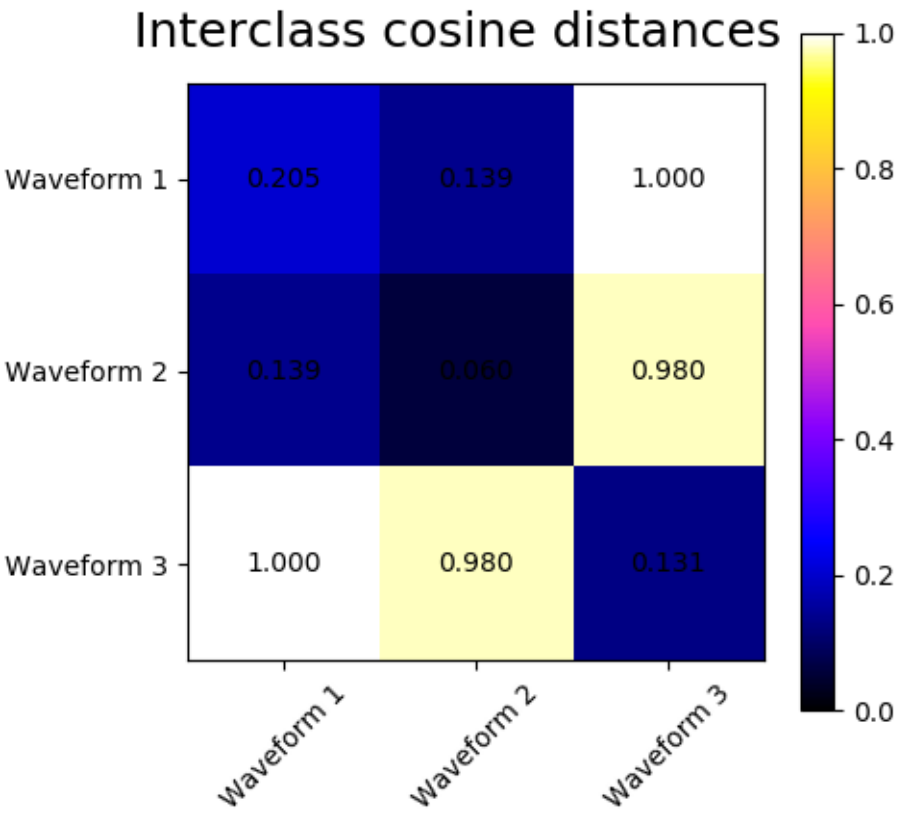
The example is engineered to show the effect of the choice of different metrics. It is applied to waveforms, which can be seen as high-dimensional vector. Indeed, the difference between metrics is usually more pronounced in high dimension (in particular for euclidean and cityblock).

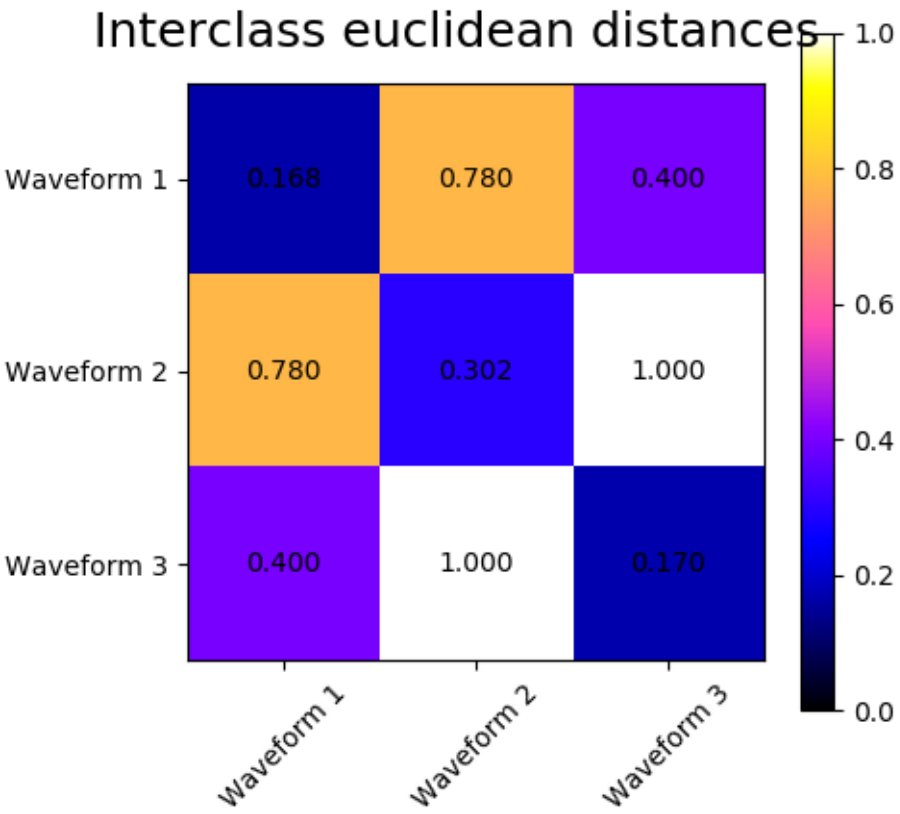
We generate data from three groups of waveforms. Two of the waveforms (waveform 1 and waveform 2) are proportional one to the other. The cosine distance is invariant to a scaling of the data, as a result, it cannot distinguish these two waveforms. Thus even with no noise, clustering using this distance will not separate out waveform 1 and 2.

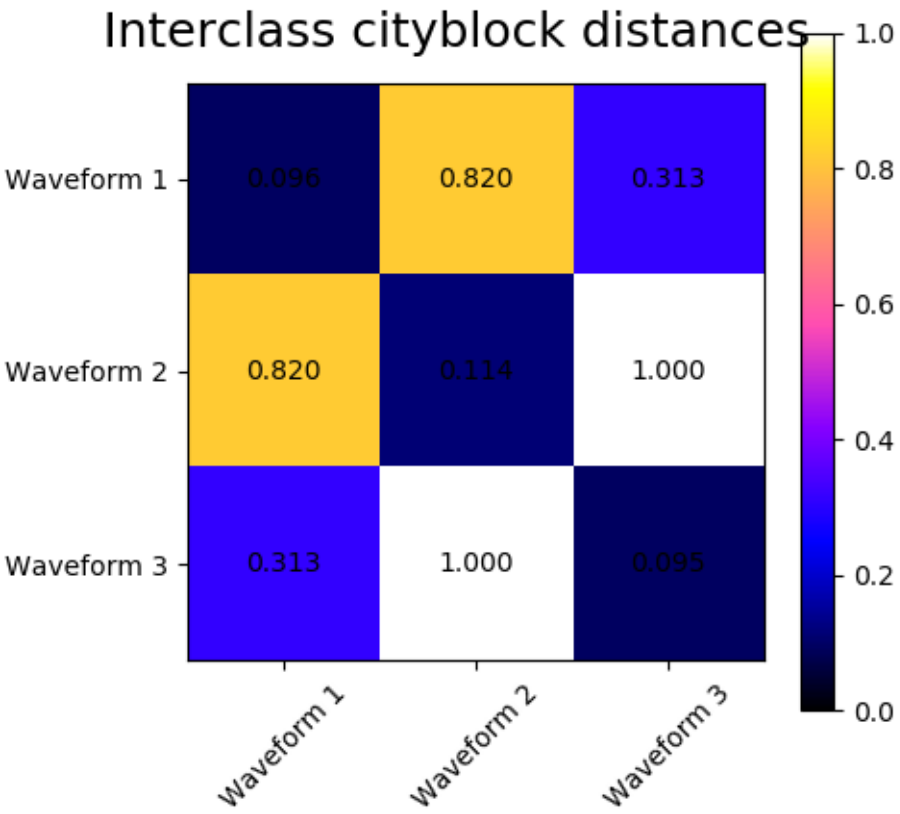
We add observation noise to these waveforms. We generate very sparse noise: only 6% of the time points contain noise. As a result, the l1 norm of this noise (ie "cityblock" distance) is much smaller than it's l2 norm ("euclidean" distance). This can be seen on the inter-class distance matrices: the values on the diagonal, that characterize the spread of the class, are much bigger for the Euclidean distance than for the cityblock distance.

When we apply clustering to the data, we find that the clustering reflects what was in the distance matrices. Indeed, for the Euclidean distance, the classes are ill-separated because of the noise, and thus the clustering does not separate the waveforms. For the cityblock distance, the separation is good and the waveform classes are recovered. Finally, the cosine distance does not separate at all waveform 1 and 2, thus the clustering puts them in the same cluster.

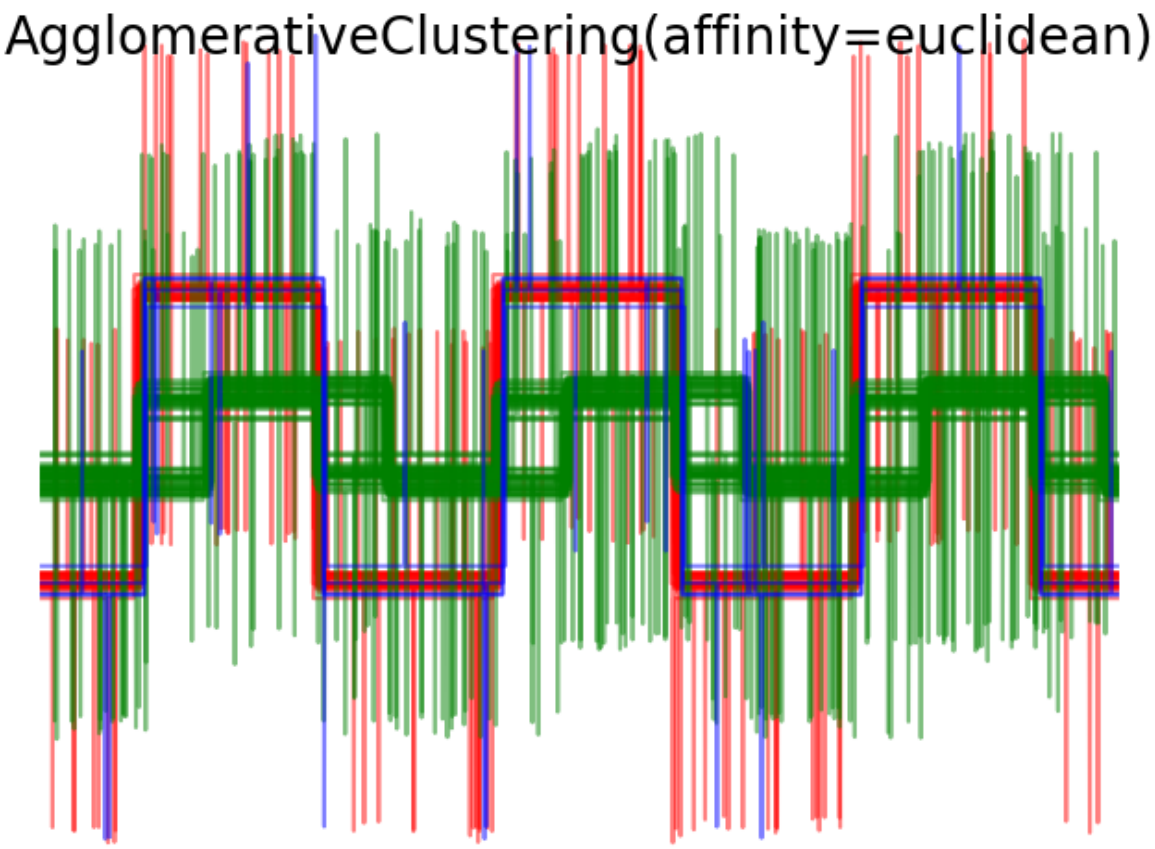

Ground truth
Waveform 1
Waveform 2
Waveform 3

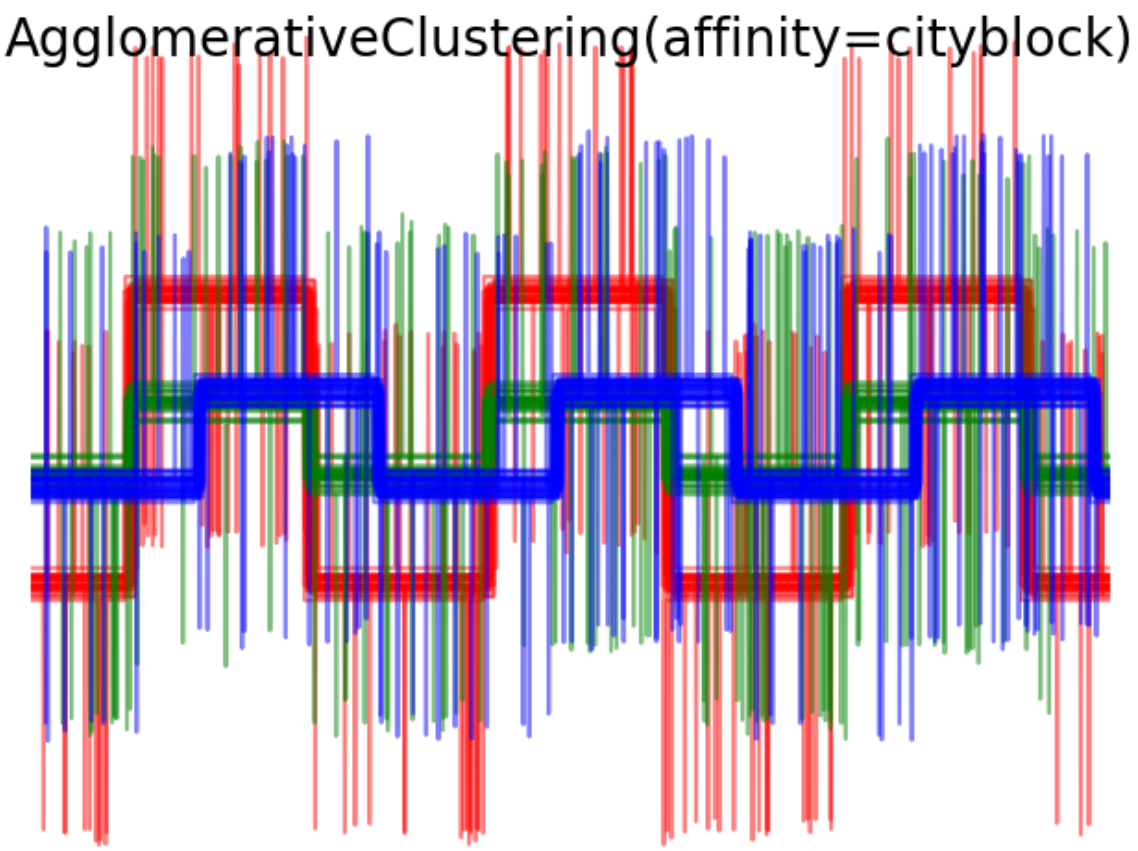

Interclass cosine distances
Waveform 1
Waveform 2
Waveform 3
0.205
0.139
1.000
0.139
0.060
0.980
1.000
0.980
0.131
Waveform 1
Waveform 2
Waveform 3
1.0
0.8
0.6
0.4
0.2
0.0


Interclass euclidean distances
Waveform 1
Waveform 2
Waveform 3
0.168
0.780
0.400
0.780
0.302
1.000
0.400
1.000
0.170
Waveform 1
Waveform 2
Waveform 3
1.0
0.8
0.6
0.4
0.2
0.0


Interclass cityblock distances
Waveform 1
Waveform 2
Waveform 3
0.096
0.820
0.313
0.820
0.114
1.000
0.313
1.000
0.095
Waveform 1
Waveform 2
Waveform 3
1.0
0.8
0.6
0.4
0.2
0.0


AgglomerativeClustering(affinity=cosine)


AgglomerativeClustering(affinity=euclidean)




```
# Author: Gael Varoquaux
# License: BSD 3-Clause or CC-0

import matplotlib.pyplot as plt
import numpy as np

from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import pairwise_distances

np.random.seed(0)

# Generate waveform data
n_features = 2000
t = np.pi * np.linspace(0, 1, n_features)


def sqr(x):
    return np.sign(np.cos(x))

X = list()
y = list()
for i, (phi, a) in enumerate([(.5, .15), (.5, .6), (.3, .2)]):
    for _ in range(30):
        phase_noise = .01 * np.random.normal()
        amplitude_noise = .04 * np.random.normal()
        additional_noise = 1 - 2 * np.random.rand(n_features)
        # Make the noise sparse
```

(continues on next page)

(continued from previous page)

```
        additional_noise[np.abs(additional_noise) < .997] = 0

        X.append(12 * ((a + amplitude_noise)
                 * (sqr(6 * (t + phi + phase_noise)))
                 + additional_noise))
        y.append(i)

X = np.array(X)
y = np.array(y)

n_clusters = 3

labels = ('Waveform 1', 'Waveform 2', 'Waveform 3')

# Plot the ground-truth labelling
plt.figure()
plt.axes([0, 0, 1, 1])
for l, c, n in zip(range(n_clusters), 'rgb',
                   labels):
    lines = plt.plot(X[y == l].T, c=c, alpha=.5)
    lines[0].set_label(n)

plt.legend(loc='best')

plt.axis('tight')
plt.axis('off')
plt.suptitle("Ground truth", size=20)


# Plot the distances
for index, metric in enumerate(["cosine", "euclidean", "cityblock"]):
    avg_dist = np.zeros((n_clusters, n_clusters))
    plt.figure(figsize=(5, 4.5))
    for i in range(n_clusters):
        for j in range(n_clusters):
            avg_dist[i, j] = pairwise_distances(X[y == i], X[y == j],
                                                metric=metric).mean()
    avg_dist /= avg_dist.max()
    for i in range(n_clusters):
        for j in range(n_clusters):
            plt.text(i, j, '%5.3f' % avg_dist[i, j],
                     verticalalignment='center',
                     horizontalalignment='center')

    plt.imshow(avg_dist, interpolation='nearest', cmap=plt.cm.gnuplot2,
               vmin=0)
    plt.xticks(range(n_clusters), labels, rotation=45)
    plt.yticks(range(n_clusters), labels)
    plt.colorbar()
    plt.suptitle("Interclass %s distances" % metric, size=18)
    plt.tight_layout()


# Plot clustering results
for index, metric in enumerate(["cosine", "euclidean", "cityblock"]):
    model = AgglomerativeClustering(n_clusters=n_clusters,
                                    linkage="average", affinity=metric)
```

(continues on next page)

(continued from previous page)

```
    model.fit(X)
    plt.figure()
    plt.axes([0, 0, 1, 1])
    for l, c in zip(np.arange(model.n_clusters), 'rgbk'):
        plt.plot(X[model.labels_ == l].T, c=c, alpha=.5)
    plt.axis('tight')
    plt.axis('off')
    plt.suptitle("AgglomerativeClustering(affinity=%s)" % metric, size=20)


plt.show()
```

Total running time of the script: (0 minutes 1.512 seconds)

Estimated memory usage: 9 MB

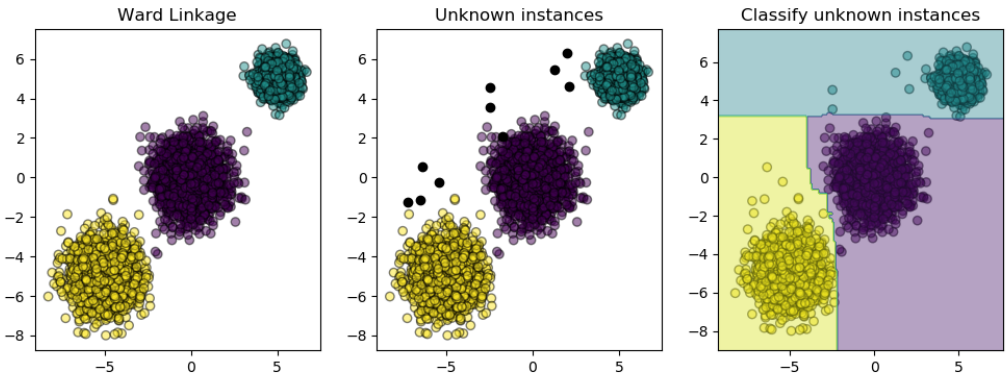
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.18 Inductive Clustering

Clustering can be expensive, especially when our dataset contains millions of datapoints. Many clustering algorithms are not *inductive* and so cannot be directly applied to new data samples without recomputing the clustering, which may be intractable. Instead, we can use clustering to then learn an inductive model with a classifier, which has several benefits:

- it allows the clusters to scale and apply to new data
- unlike re-fitting the clusters to new samples, it makes sure the labelling procedure is consistent over time
- it allows us to use the inferential capabilities of the classifier to describe or explain the clusters

This example illustrates a generic implementation of a meta-estimator which extends clustering by inducing a classifier from the cluster labels.




```
# Authors: Chirag Nagpal
#          Christos Aridas
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
from sklearn.base import BaseEstimator, clone
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets import make_blobs
from sklearn.ensemble import RandomForestClassifier
from sklearn.utils.metaestimators import if_delegate_has_method


N_SAMPLES = 5000
RANDOM_STATE = 42


class InductiveClusterer(BaseEstimator):
    def __init__(self, clusterer, classifier):
        self.clusterer = clusterer
        self.classifier = classifier

    def fit(self, X, y=None):
        self.clusterer_ = clone(self.clusterer)
        self.classifier_ = clone(self.classifier)
        y = self.clusterer_.fit_predict(X)
        self.classifier_.fit(X, y)
        return self

    @if_delegate_has_method(delegate='classifier_')
    def predict(self, X):
        return self.classifier_.predict(X)

    @if_delegate_has_method(delegate='classifier_')
    def decision_function(self, X):
        return self.classifier_.decision_function(X)


def plot_scatter(X,  color, alpha=0.5):
    return plt.scatter(X[:, 0],
                       X[:, 1],
                       c=color,
                       alpha=alpha,
                       edgecolor='k')


# Generate some training data from clustering
X, y = make_blobs(n_samples=N_SAMPLES,
                  cluster_std=[1.0, 1.0, 0.5],
                  centers=[(-5, -5), (0, 0), (5, 5)],
                  random_state=RANDOM_STATE)


# Train a clustering algorithm on the training data and get the cluster labels
clusterer = AgglomerativeClustering(n_clusters=3)
cluster_labels = clusterer.fit_predict(X)

plt.figure(figsize=(12, 4))

plt.subplot(131)
plot_scatter(X, cluster_labels)
plt.title("Ward Linkage")
```

(continues on next page)

(continued from previous page)

```
# Generate new samples and plot them along with the original dataset
X_new, y_new = make_blobs(n_samples=10,
                          centers=[(-7, -1), (-2, 4), (3, 6)],
                          random_state=RANDOM_STATE)

plt.subplot(132)
plot_scatter(X, cluster_labels)
plot_scatter(X_new, 'black', 1)
plt.title("Unknown instances")


# Declare the inductive learning model that it will be used to
# predict cluster membership for unknown instances
classifier = RandomForestClassifier(random_state=RANDOM_STATE)
inductive_learner = InductiveClusterer(clusterer, classifier).fit(X)

probable_clusters = inductive_learner.predict(X_new)


plt.subplot(133)
plot_scatter(X, cluster_labels)
plot_scatter(X_new, probable_clusters)

# Plotting decision regions
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

Z = inductive_learner.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.4)
plt.title("Classify unknown instances")

plt.show()
```

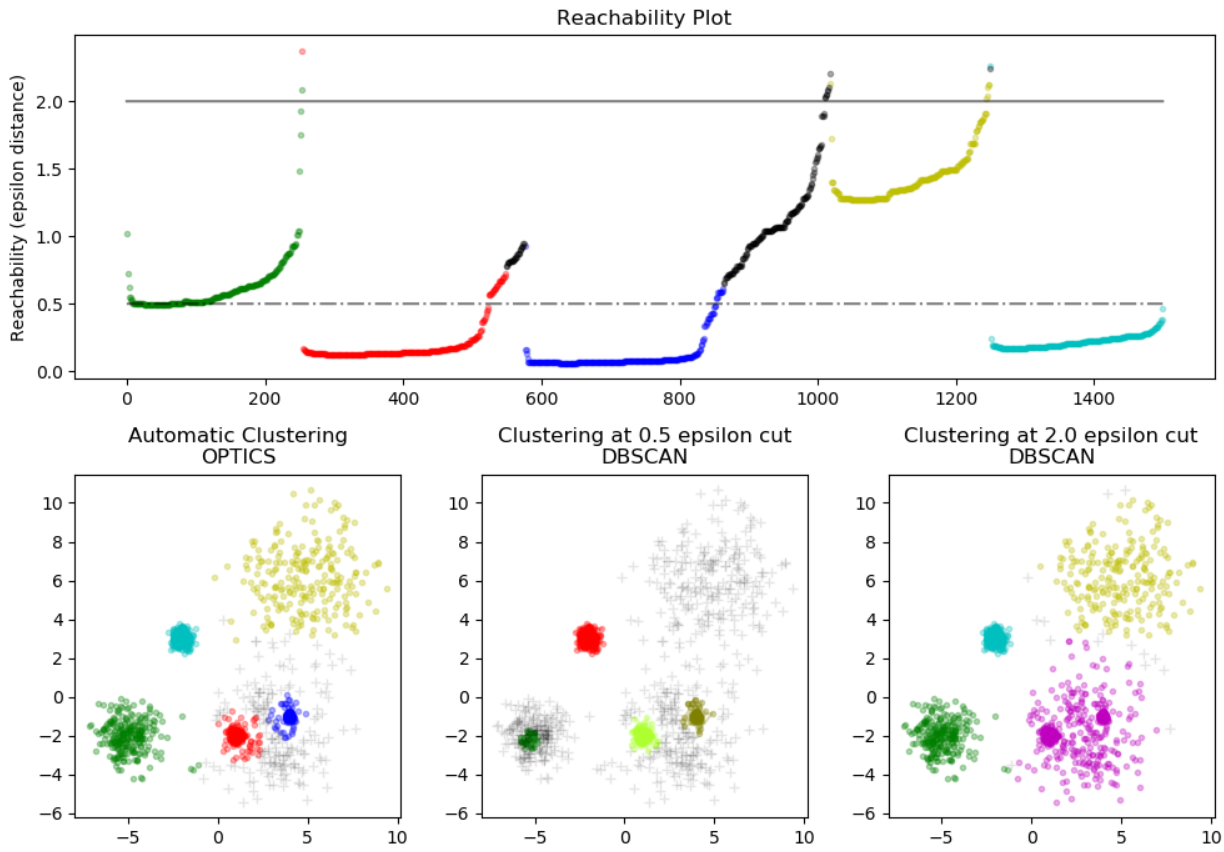
Total running time of the script: (0 minutes 2.723 seconds)

Estimated memory usage: 205 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.19 Demo of OPTICS clustering algorithm

Finds core samples of high density and expands clusters from them. This example uses data that is generated so that the clusters have different densities. The *sklearn.cluster.OPTICS* is first used with its Xi cluster detection method, and then setting specific thresholds on the reachability, which corresponds to *sklearn.cluster.DBSCAN*. We can see that the different clusters of OPTICS's Xi method can be recovered with different choices of thresholds in DBSCAN.




```
# Authors: Shane Grigsby <refuge@rocktalus.com>
#          Adrin Jalali <adrin.jalali@gmail.com>
# License: BSD 3 clause


from sklearn.cluster import OPTICS, cluster_optics_dbscan
import matplotlib.gridspec as gridspec
import matplotlib.pyplot as plt
import numpy as np

# Generate sample data

np.random.seed(0)
n_points_per_cluster = 250

C1 = [-5, -2] + .8 * np.random.randn(n_points_per_cluster, 2)
C2 = [4, -1] + .1 * np.random.randn(n_points_per_cluster, 2)
C3 = [1, -2] + .2 * np.random.randn(n_points_per_cluster, 2)
C4 = [-2, 3] + .3 * np.random.randn(n_points_per_cluster, 2)
C5 = [3, -2] + 1.6 * np.random.randn(n_points_per_cluster, 2)
C6 = [5, 6] + 2 * np.random.randn(n_points_per_cluster, 2)
X = np.vstack((C1, C2, C3, C4, C5, C6))

clust = OPTICS(min_samples=50, xi=.05, min_cluster_size=.05)

# Run the fit
clust.fit(X)
```

(continues on next page)

(continued from previous page)

```
labels_050 = cluster_optics_dbscan(reachability=clust.reachability_,
                                   core_distances=clust.core_distances_,
                                   ordering=clust.ordering_, eps=0.5)
labels_200 = cluster_optics_dbscan(reachability=clust.reachability_,
                                   core_distances=clust.core_distances_,
                                   ordering=clust.ordering_, eps=2)

space = np.arange(len(X))
reachability = clust.reachability_[clust.ordering_]
labels = clust.labels_[clust.ordering_]

plt.figure(figsize=(10, 7))
G = gridspec.GridSpec(2, 3)
ax1 = plt.subplot(G[0, :])
ax2 = plt.subplot(G[1, 0])
ax3 = plt.subplot(G[1, 1])
ax4 = plt.subplot(G[1, 2])

# Reachability plot
colors = ['g.', 'r.', 'b.', 'y.', 'c.']
for klass, color in zip(range(0, 5), colors):
    Xk = space[labels == klass]
    Rk = reachability[labels == klass]
    ax1.plot(Xk, Rk, color, alpha=0.3)
ax1.plot(space[labels == -1], reachability[labels == -1], 'k.', alpha=0.3)
ax1.plot(space, np.full_like(space, 2., dtype=float), 'k-', alpha=0.5)
ax1.plot(space, np.full_like(space, 0.5, dtype=float), 'k-.', alpha=0.5)
ax1.set_ylabel('Reachability (epsilon distance)')
ax1.set_title('Reachability Plot')

# OPTICS
colors = ['g.', 'r.', 'b.', 'y.', 'c.']
for klass, color in zip(range(0, 5), colors):
    Xk = X[clust.labels_ == klass]
    ax2.plot(Xk[:, 0], Xk[:, 1], color, alpha=0.3)
ax2.plot(X[clust.labels_ == -1, 0], X[clust.labels_ == -1, 1], 'k+', alpha=0.1)
ax2.set_title('Automatic Clustering\nOPTICS')

# DBSCAN at 0.5
colors = ['g', 'greenyellow', 'olive', 'r', 'b', 'c']
for klass, color in zip(range(0, 6), colors):
    Xk = X[labels_050 == klass]
    ax3.plot(Xk[:, 0], Xk[:, 1], color, alpha=0.3, marker='.')
ax3.plot(X[labels_050 == -1, 0], X[labels_050 == -1, 1], 'k+', alpha=0.1)
ax3.set_title('Clustering at 0.5 epsilon cut\nDBSCAN')

# DBSCAN at 2.
colors = ['g.', 'm.', 'y.', 'c.']
for klass, color in zip(range(0, 4), colors):
    Xk = X[labels_200 == klass]
    ax4.plot(Xk[:, 0], Xk[:, 1], color, alpha=0.3)
ax4.plot(X[labels_200 == -1, 0], X[labels_200 == -1, 1], 'k+', alpha=0.1)
ax4.set_title('Clustering at 2.0 epsilon cut\nDBSCAN')

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 1.282 seconds)

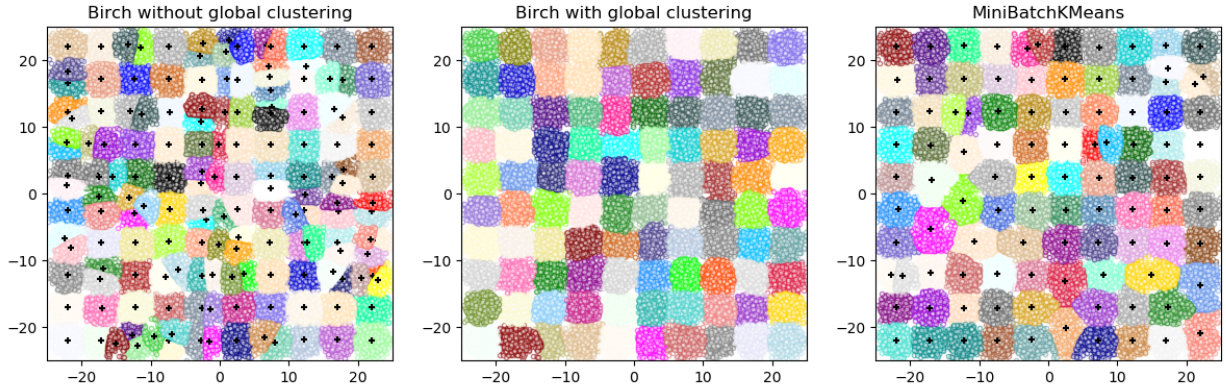
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.20 Compare BIRCH and MiniBatchKMeans

This example compares the timing of Birch (with and without the global clustering step) and MiniBatchKMeans on a synthetic dataset having 100,000 samples and 2 features generated using make_blobs.

If `n_clusters` is set to None, the data is reduced from 100,000 samples to a set of 158 clusters. This can be viewed as a preprocessing step before the final (global) clustering step that further reduces these 158 clusters to 100 clusters.



Out:

```
Birch without global clustering as the final step took 2.70 seconds
n_clusters : 158
Birch with global clustering as the final step took 2.85 seconds
n_clusters : 100
Time taken to run MiniBatchKMeans 3.43 seconds
```

```
# Authors: Manoj Kumar <manojkumarsivaraj334@gmail.com
#          Alexandre Gramfort <alexandre.gramfort@telecom-paristech.fr>
# License: BSD 3 clause

print(__doc__)

from itertools import cycle
from time import time
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors

from sklearn.cluster import Birch, MiniBatchKMeans
```

(continues on next page)

(continued from previous page)

```
from sklearn.datasets import make_blobs


# Generate centers for the blobs so that it forms a 10 X 10 grid.
xx = np.linspace(-22, 22, 10)
yy = np.linspace(-22, 22, 10)
xx, yy = np.meshgrid(xx, yy)
n_centres = np.hstack((np.ravel(xx)[:, np.newaxis],
                       np.ravel(yy)[:, np.newaxis]))

# Generate blobs to do a comparison between MiniBatchKMeans and Birch.
X, y = make_blobs(n_samples=100000, centers=n_centres, random_state=0)

# Use all colors that matplotlib provides by default.
colors_ = cycle(colors.cnames.keys())

fig = plt.figure(figsize=(12, 4))
fig.subplots_adjust(left=0.04, right=0.98, bottom=0.1, top=0.9)

# Compute clustering with Birch with and without the final clustering step
# and plot.
birch_models = [Birch(threshold=1.7, n_clusters=None),
                Birch(threshold=1.7, n_clusters=100)]
final_step = ['without global clustering', 'with global clustering']

for ind, (birch_model, info) in enumerate(zip(birch_models, final_step)):
    t = time()
    birch_model.fit(X)
    time_ = time() - t
    print("Birch %s as the final step took %0.2f seconds" % (
          info, (time() - t)))

    # Plot result
    labels = birch_model.labels_
    centroids = birch_model.subcluster_centers_
    n_clusters = np.unique(labels).size
    print("n_clusters : %d" % n_clusters)

    ax = fig.add_subplot(1, 3, ind + 1)
    for this_centroid, k, col in zip(centroids, range(n_clusters), colors_):
        mask = labels == k
        ax.scatter(X[mask, 0], X[mask, 1],
                   c='w', edgecolor=col, marker='.', alpha=0.5)
        if birch_model.n_clusters is None:
            ax.scatter(this_centroid[0], this_centroid[1], marker='+',
                       c='k', s=25)
    ax.set_ylim([-25, 25])
    ax.set_xlim([-25, 25])
    ax.set_autoscaley_on(False)
    ax.set_title('Birch %s' % info)

# Compute clustering with MiniBatchKMeans.
mbk = MiniBatchKMeans(init='k-means++', n_clusters=100, batch_size=100,
                      n_init=10, max_no_improvement=10, verbose=0,
                      random_state=0)
t0 = time()
mbk.fit(X)
```

(continues on next page)

(continued from previous page)

```
t_mini_batch = time() - t0
print("Time taken to run MiniBatchKMeans %0.2f seconds" % t_mini_batch)
mbk_means_labels_unique = np.unique(mbk.labels_)

ax = fig.add_subplot(1, 3, 3)
for this_centroid, k, col in zip(mbk.cluster_centers_,
                                 range(n_clusters), colors_):
    mask = mbk.labels_ == k
    ax.scatter(X[mask, 0], X[mask, 1], marker='.',
               c='w', edgecolor=col, alpha=0.5)
    ax.scatter(this_centroid[0], this_centroid[1], marker='+',
               c='k', s=25)
ax.set_xlim([-25, 25])
ax.set_ylim([-25, 25])
ax.set_title("MiniBatchKMeans")
ax.set_autoscaley_on(False)
plt.show()
```

Total running time of the script: (0 minutes 11.460 seconds)

Estimated memory usage: 129 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

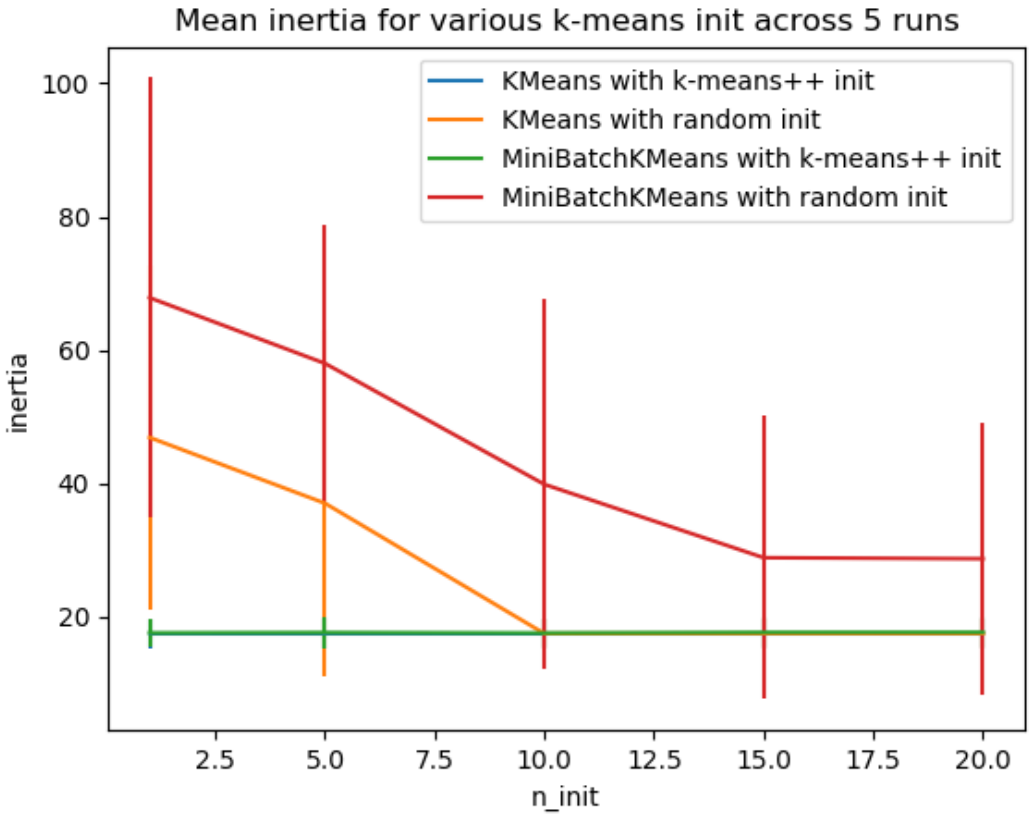
6.5.21 Empirical evaluation of the impact of k-means initialization

Evaluate the ability of k-means initializations strategies to make the algorithm convergence robust as measured by the relative standard deviation of the inertia of the clustering (i.e. the sum of squared distances to the nearest cluster center).

The first plot shows the best inertia reached for each combination of the model (`KMeans` or `MiniBatchKMeans`) and the init method (`init="random"` or `init="kmeans++"`) for increasing values of the `n_init` parameter that controls the number of initializations.

The second plot demonstrate one single run of the `MiniBatchKMeans` estimator using a `init="random"` and `n_init=1`. This run leads to a bad convergence (local optimum) with estimated centers stuck between ground truth clusters.

The dataset used for evaluation is a 2D grid of isotropic Gaussian clusters widely spaced.


Mean inertia for various k-means init across 5 runs
KMeans with k-means++ init
KMeans with random init
MiniBatchKMeans with k-means++ init
MiniBatchKMeans with random init
inertia
n_init
100
80
60
40
20
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Out:

```
Evaluation of KMeans with k-means++ init
Evaluation of KMeans with random init
Evaluation of MiniBatchKMeans with k-means++ init
Evaluation of MiniBatchKMeans with random init
```

```
print(__doc__)

# Author: Olivier Grisel <olivier.grisel@ensta.org>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

from sklearn.utils import shuffle
from sklearn.utils import check_random_state
from sklearn.cluster import MiniBatchKMeans
from sklearn.cluster import KMeans

random_state = np.random.RandomState(0)
```

(continues on next page)

(continued from previous page)

```
# Number of run (with randomly generated dataset) for each strategy so as
# to be able to compute an estimate of the standard deviation
n_runs = 5

# k-means models can do several random inits so as to be able to trade
# CPU time for convergence robustness
n_init_range = np.array([1, 5, 10, 15, 20])

# Datasets generation parameters
n_samples_per_center = 100
grid_size = 3
scale = 0.1
n_clusters = grid_size ** 2


def make_data(random_state, n_samples_per_center, grid_size, scale):
    random_state = check_random_state(random_state)
    centers = np.array([[i, j]
                        for i in range(grid_size)
                        for j in range(grid_size)])
    n_clusters_true, n_features = centers.shape

    noise = random_state.normal(
        scale=scale, size=(n_samples_per_center, centers.shape[1]))

    X = np.concatenate([c + noise for c in centers])
    y = np.concatenate([[i] * n_samples_per_center
                        for i in range(n_clusters_true)])
    return shuffle(X, y, random_state=random_state)

# Part 1: Quantitative evaluation of various init methods

plt.figure()
plots = []
legends = []

cases = [
    (KMeans, 'k-means++', {}),
    (KMeans, 'random', {}),
    (MiniBatchKMeans, 'k-means++', {'max_no_improvement': 3}),
    (MiniBatchKMeans, 'random', {'max_no_improvement': 3, 'init_size': 500}),
]

for factory, init, params in cases:
    print("Evaluation of %s with %s init" % (factory.__name__, init))
    inertia = np.empty((len(n_init_range), n_runs))

    for run_id in range(n_runs):
        X, y = make_data(run_id, n_samples_per_center, grid_size, scale)
        for i, n_init in enumerate(n_init_range):
            km = factory(n_clusters=n_clusters, init=init, random_state=run_id,
                         n_init=n_init, **params).fit(X)
            inertia[i, run_id] = km.inertia_
    p = plt.errorbar(n_init_range, inertia.mean(axis=1), inertia.std(axis=1))
    plots.append(p[0])
    legends.append("%s with %s init" % (factory.__name__, init))
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('n_init')
plt.ylabel('inertia')
plt.legend(plots, legends)
plt.title("Mean inertia for various k-means init across %d runs" % n_runs)

# Part 2: Qualitative visual inspection of the convergence

X, y = make_data(random_state, n_samples_per_center, grid_size, scale)
km = MiniBatchKMeans(n_clusters=n_clusters, init='random', n_init=1,
                     random_state=random_state).fit(X)

plt.figure()
for k in range(n_clusters):
    my_members = km.labels_ == k
    color = cm.nipy_spectral(float(k) / n_clusters, 1)
    plt.plot(X[my_members, 0], X[my_members, 1], 'o', marker='.', c=color)
    cluster_center = km.cluster_centers_[k]
    plt.plot(cluster_center[0], cluster_center[1], 'o',
             markerfacecolor=color, markeredgecolor='k', markersize=6)
    plt.title("Example cluster allocation with a single random init\n"
              "with MiniBatchKMeans")

plt.show()
```

Total running time of the script: (0 minutes 3.557 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

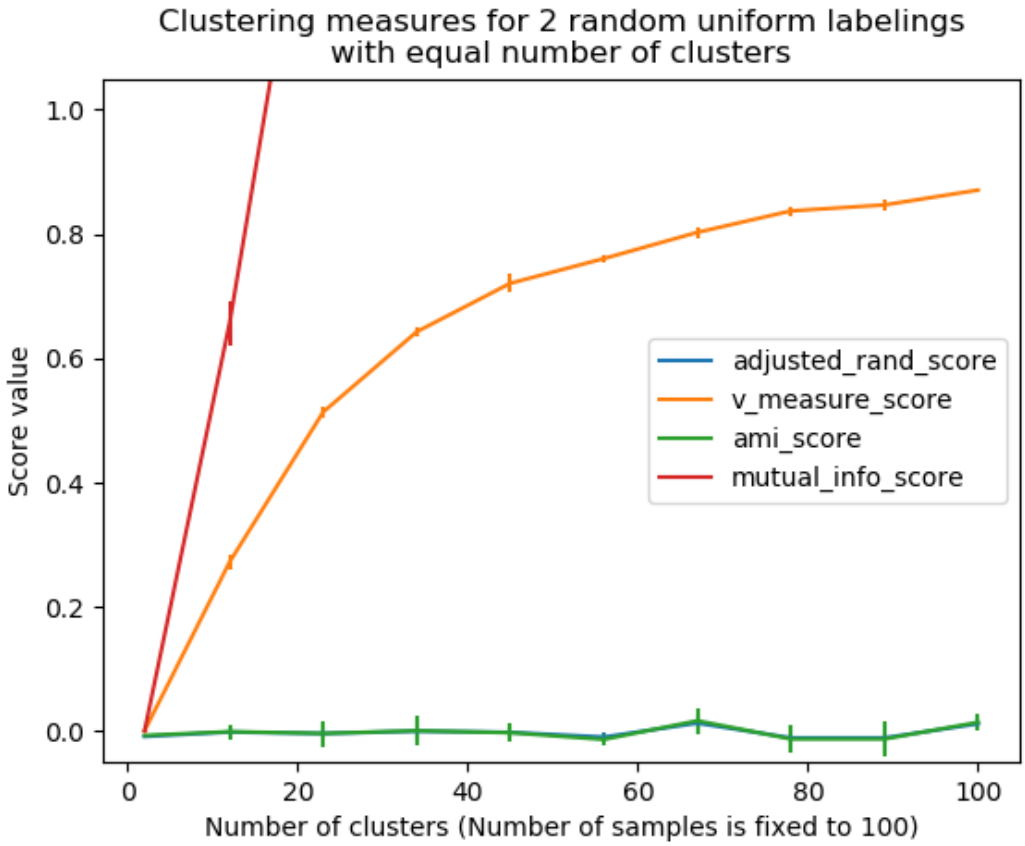
6.5.22 Adjustment for chance in clustering performance evaluation

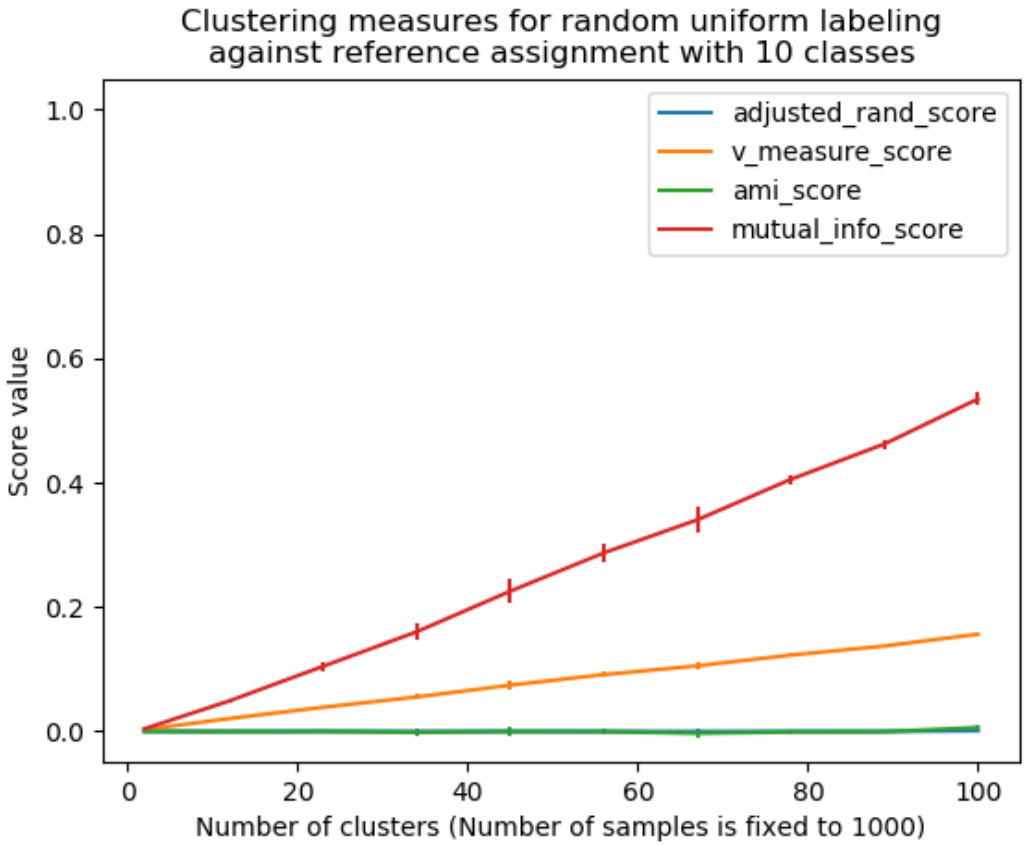
The following plots demonstrate the impact of the number of clusters and number of samples on various clustering performance evaluation metrics.

Non-adjusted measures such as the V-Measure show a dependency between the number of clusters and the number of samples: the mean V-Measure of random labeling increases significantly as the number of clusters is closer to the total number of samples used to compute the measure.

Adjusted for chance measure such as ARI display some random variations centered around a mean score of 0.0 for any number of samples and clusters.

Only adjusted measures can hence safely be used as a consensus index to evaluate the average stability of clustering algorithms for a given value of k on various overlapping sub-samples of the dataset.


Clustering measures for 2 random uniform labelings
with equal number of clusters
Score value
Number of clusters (Number of samples is fixed to 100)
adjusted_rand_score
v_measure_score
ami_score
mutual_info_score
0.0
0.2
0.4
0.6
0.8
1.0
0
20
40
60
80
100



Out:

```
Computing adjusted_rand_score for 10 values of n_clusters and n_samples=100
done in 0.050s
Computing v_measure_score for 10 values of n_clusters and n_samples=100
done in 0.068s
Computing ami_score for 10 values of n_clusters and n_samples=100
done in 0.356s
Computing mutual_info_score for 10 values of n_clusters and n_samples=100
done in 0.044s
Computing adjusted_rand_score for 10 values of n_clusters and n_samples=1000
done in 0.051s
Computing v_measure_score for 10 values of n_clusters and n_samples=1000
done in 0.064s
Computing ami_score for 10 values of n_clusters and n_samples=1000
done in 0.208s
Computing mutual_info_score for 10 values of n_clusters and n_samples=1000
done in 0.048s
```

```
print(__doc__)

# Author: Olivier Grisel <olivier.grisel@ensta.org>
```

(continues on next page)

(continued from previous page)

```
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from time import time
from sklearn import metrics

def uniform_labelings_scores(score_func, n_samples, n_clusters_range,
                             fixed_n_classes=None, n_runs=5, seed=42):
    """Compute score for 2 random uniform cluster labelings.

    Both random labelings have the same number of clusters for each value
    possible value in ``n_clusters_range``.

    When fixed_n_classes is not None the first labeling is considered a ground
    truth class assignment with fixed number of classes.
    """
    random_labels = np.random.RandomState(seed).randint
    scores = np.zeros((len(n_clusters_range), n_runs))

    if fixed_n_classes is not None:
        labels_a = random_labels(low=0, high=fixed_n_classes, size=n_samples)

    for i, k in enumerate(n_clusters_range):
        for j in range(n_runs):
            if fixed_n_classes is None:
                labels_a = random_labels(low=0, high=k, size=n_samples)
            labels_b = random_labels(low=0, high=k, size=n_samples)
            scores[i, j] = score_func(labels_a, labels_b)
    return scores


def ami_score(U, V):
    return metrics.adjusted_mutual_info_score(U, V)

score_funcs = [
    metrics.adjusted_rand_score,
    metrics.v_measure_score,
    ami_score,
    metrics.mutual_info_score,
]

# 2 independent random clusterings with equal cluster number

n_samples = 100
n_clusters_range = np.linspace(2, n_samples, 10).astype(np.int)

plt.figure(1)

plots = []
names = []
for score_func in score_funcs:
    print("Computing %s for %d values of n_clusters and n_samples=%d"
          % (score_func.__name__, len(n_clusters_range), n_samples))

    t0 = time()
    scores = uniform_labelings_scores(score_func, n_samples, n_clusters_range)
```

(continues on next page)

(continued from previous page)

```
    print("done in %0.3fs" % (time() - t0))
    plots.append(plt.errorbar(
        n_clusters_range, np.median(scores, axis=1), scores.std(axis=1))[0])
    names.append(score_func.__name__)

plt.title("Clustering measures for 2 random uniform labelings\n"
          "with equal number of clusters")
plt.xlabel('Number of clusters (Number of samples is fixed to %d)' % n_samples)
plt.ylabel('Score value')
plt.legend(plots, names)
plt.ylim(bottom=-0.05, top=1.05)


# Random labeling with varying n_clusters against ground class labels
# with fixed number of clusters

n_samples = 1000
n_clusters_range = np.linspace(2, 100, 10).astype(np.int)
n_classes = 10

plt.figure(2)

plots = []
names = []
for score_func in score_funcs:
    print("Computing %s for %d values of n_clusters and n_samples=%d"
          % (score_func.__name__, len(n_clusters_range), n_samples))

    t0 = time()
    scores = uniform_labelings_scores(score_func, n_samples, n_clusters_range,
                                      fixed_n_classes=n_classes)
    print("done in %0.3fs" % (time() - t0))
    plots.append(plt.errorbar(
        n_clusters_range, scores.mean(axis=1), scores.std(axis=1))[0])
    names.append(score_func.__name__)

plt.title("Clustering measures for random uniform labeling\n"
          "against reference assignment with %d classes" % n_classes)
plt.xlabel('Number of clusters (Number of samples is fixed to %d)' % n_samples)
plt.ylabel('Score value')
plt.ylim(bottom=-0.05, top=1.05)
plt.legend(plots, names)
plt.show()
```

Total running time of the script: (0 minutes 1.225 seconds)

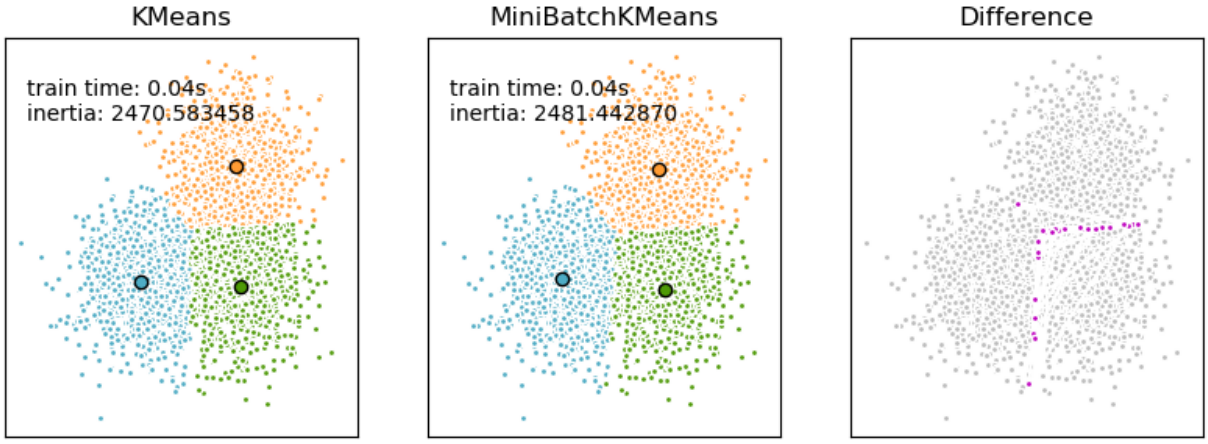
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.23 Comparison of the K-Means and MiniBatchKMeans clustering algorithms

We want to compare the performance of the MiniBatchKMeans and KMeans: the MiniBatchKMeans is faster, but gives slightly different results (see *Mini Batch K-Means*).

We will cluster a set of data, first with KMeans and then with MiniBatchKMeans, and plot the results. We will also plot the points that are labelled differently between the two algorithms.




```
print(__doc__)

import time

import numpy as np
import matplotlib.pyplot as plt

from sklearn.cluster import MiniBatchKMeans, KMeans
from sklearn.metrics.pairwise import pairwise_distances_argmin
from sklearn.datasets import make_blobs

# #############################################################################
# Generate sample data
np.random.seed(0)

batch_size = 45
centers = [[1, 1], [-1, -1], [1, -1]]
n_clusters = len(centers)
X, labels_true = make_blobs(n_samples=3000, centers=centers, cluster_std=0.7)

# #############################################################################
# Compute clustering with Means

k_means = KMeans(init='k-means++', n_clusters=3, n_init=10)
t0 = time.time()
k_means.fit(X)
t_batch = time.time() - t0

# #############################################################################
# Compute clustering with MiniBatchKMeans

mbk = MiniBatchKMeans(init='k-means++', n_clusters=3, batch_size=batch_size,
                      n_init=10, max_no_improvement=10, verbose=0)
t0 = time.time()
mbk.fit(X)
t_mini_batch = time.time() - t0

# #############################################################################
```

(continues on next page)

(continued from previous page)

```
# Plot result

fig = plt.figure(figsize=(8, 3))
fig.subplots_adjust(left=0.02, right=0.98, bottom=0.05, top=0.9)
colors = ['#4EACC5', '#FF9C34', '#4E9A06']

# We want to have the same colors for the same cluster from the
# MiniBatchKMeans and the KMeans algorithm. Let's pair the cluster centers per
# closest one.
k_means_cluster_centers = k_means.cluster_centers_
order = pairwise_distances_argmin(k_means.cluster_centers_,
                                  mbk.cluster_centers_)
mbk_means_cluster_centers = mbk.cluster_centers_[order]

k_means_labels = pairwise_distances_argmin(X, k_means_cluster_centers)
mbk_means_labels = pairwise_distances_argmin(X, mbk_means_cluster_centers)

# KMeans
ax = fig.add_subplot(1, 3, 1)
for k, col in zip(range(n_clusters), colors):
    my_members = k_means_labels == k
    cluster_center = k_means_cluster_centers[k]
    ax.plot(X[my_members, 0], X[my_members, 1], 'w',
            markerfacecolor=col, marker='.')
    ax.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
            markeredgecolor='k', markersize=6)
ax.set_title('KMeans')
ax.set_xticks(())
ax.set_yticks(())
plt.text(-3.5, 1.8,  'train time: %.2fs\ninertia: %f' % (
    t_batch, k_means.inertia_))

# MiniBatchKMeans
ax = fig.add_subplot(1, 3, 2)
for k, col in zip(range(n_clusters), colors):
    my_members = mbk_means_labels == k
    cluster_center = mbk_means_cluster_centers[k]
    ax.plot(X[my_members, 0], X[my_members, 1], 'w',
            markerfacecolor=col, marker='.')
    ax.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
            markeredgecolor='k', markersize=6)
ax.set_title('MiniBatchKMeans')
ax.set_xticks(())
ax.set_yticks(())
plt.text(-3.5, 1.8, 'train time: %.2fs\ninertia: %f' %
         (t_mini_batch, mbk.inertia_))

# Initialise the different array to all False
different = (mbk_means_labels == 4)
ax = fig.add_subplot(1, 3, 3)

for k in range(n_clusters):
    different += ((k_means_labels == k) != (mbk_means_labels == k))

identic = np.logical_not(different)
ax.plot(X[identic, 0], X[identic, 1], 'w',
        markerfacecolor='#bbbbbb', marker='.')
```

(continues on next page)

(continued from previous page)

```
ax.plot(X[different, 0], X[different, 1], 'w',
        markerfacecolor='m', marker='.')
ax.set_title('Difference')
ax.set_xticks(())
ax.set_yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.502 seconds)

Estimated memory usage: 8 MB

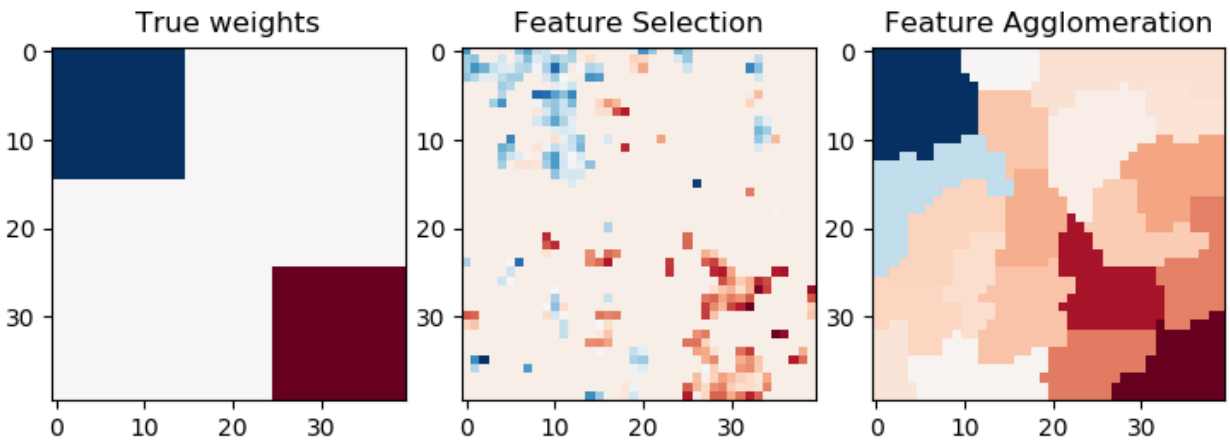
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.24 Feature agglomeration vs. univariate selection

This example compares 2 dimensionality reduction strategies:

- univariate feature selection with Anova
- feature agglomeration with Ward hierarchical clustering

Both methods are compared in a regression problem using a BayesianRidge as supervised estimator.



Out:

```
________________________________________________________________________________
[Memory] Calling sklearn.cluster._agglomerative.ward_tree...
ward_tree(array([[-0.451933, ..., -0.675318],
       ...,
       [ 0.275706, ..., -1.085711]]),
<1600x1600 sparse matrix of type '<class 'numpy.int64'>'
        with 7840 stored elements in COOrdinate format>, n_clusters=None, return_
↪distance=False)
_____________________________________________________________ward_tree - 0.1s, 0.0min
________________________________________________________________________________
[Memory] Calling sklearn.cluster._agglomerative.ward_tree...
ward_tree(array([[ 0.905206, ...,  0.161245],
       ...,
```

(continues on next page)

(continued from previous page)

```
       [-0.849835, ..., -1.091621]]),
<1600x1600 sparse matrix of type '<class 'numpy.int64'>'
        with 7840 stored elements in COOrdinate format>, n_clusters=None, return_
↪distance=False)
_____________________________________________________________ward_tree - 0.1s, 0.0min
_____________________________________________________________________________________
[Memory] Calling sklearn.cluster._agglomerative.ward_tree...
ward_tree(array([[ 0.905206, ..., -0.675318],
       ...,
       [-0.849835, ..., -1.085711]]),
<1600x1600 sparse matrix of type '<class 'numpy.int64'>'
        with 7840 stored elements in COOrdinate format>, n_clusters=None, return_
↪distance=False)
_____________________________________________________________ward_tree - 0.1s, 0.0min
_____________________________________________________________________________________
[Memory] Calling sklearn.feature_selection._univariate_selection.f_regression...
f_regression(array([[-0.451933, ...,  0.275706],
       ...,
       [-0.675318, ..., -1.085711]]),
array([ 25.267703, ..., -25.026711]))
_____________________________________________________f_regression - 0.0s, 0.0min
_____________________________________________________________________________________
[Memory] Calling sklearn.feature_selection._univariate_selection.f_regression...
f_regression(array([[ 0.905206, ..., -0.849835],
       ...,
       [ 0.161245, ..., -1.091621]]),
array([ -27.447268, ..., -112.638768]))
_____________________________________________________f_regression - 0.0s, 0.0min
_____________________________________________________________________________________
[Memory] Calling sklearn.feature_selection._univariate_selection.f_regression...
f_regression(array([[ 0.905206, ..., -0.849835],
       ...,
       [-0.675318, ..., -1.085711]]),
array([-27.447268, ..., -25.026711]))
_____________________________________________________f_regression - 0.0s, 0.0min
```

```
# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

print(__doc__)

import shutil
import tempfile

import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg, ndimage
from joblib import Memory

from sklearn.feature_extraction.image import grid_to_graph
from sklearn import feature_selection
```

(continues on next page)

(continued from previous page)

```
from sklearn.cluster import FeatureAgglomeration
from sklearn.linear_model import BayesianRidge
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import KFold

# #############################################################################
# Generate data
n_samples = 200
size = 40  # image size
roi_size = 15
snr = 5.
np.random.seed(0)
mask = np.ones([size, size], dtype=np.bool)

coef = np.zeros((size, size))
coef[0:roi_size, 0:roi_size] = -1.
coef[-roi_size:, -roi_size:] = 1.

X = np.random.randn(n_samples, size ** 2)
for x in X:  # smooth data
    x[:] = ndimage.gaussian_filter(x.reshape(size, size), sigma=1.0).ravel()
X -= X.mean(axis=0)
X /= X.std(axis=0)

y = np.dot(X, coef.ravel())
noise = np.random.randn(y.shape[0])
noise_coef = (linalg.norm(y, 2) / np.exp(snr / 20.)) / linalg.norm(noise, 2)
y += noise_coef * noise  # add noise

# #############################################################################
# Compute the coefs of a Bayesian Ridge with GridSearch
cv = KFold(2)  # cross-validation generator for model selection
ridge = BayesianRidge()
cachedir = tempfile.mkdtemp()
mem = Memory(location=cachedir, verbose=1)

# Ward agglomeration followed by BayesianRidge
connectivity = grid_to_graph(n_x=size, n_y=size)
ward = FeatureAgglomeration(n_clusters=10, connectivity=connectivity,
                            memory=mem)
clf = Pipeline([('ward', ward), ('ridge', ridge)])
# Select the optimal number of parcels with grid search
clf = GridSearchCV(clf, {'ward__n_clusters': [10, 20, 30]}, n_jobs=1, cv=cv)
clf.fit(X, y)  # set the best parameters
coef_ = clf.best_estimator_.steps[-1][1].coef_
coef_ = clf.best_estimator_.steps[0][1].inverse_transform(coef_)
coef_agglomeration_ = coef_.reshape(size, size)

# Anova univariate feature selection followed by BayesianRidge
f_regression = mem.cache(feature_selection.f_regression)  # caching function
anova = feature_selection.SelectPercentile(f_regression)
clf = Pipeline([('anova', anova), ('ridge', ridge)])
# Select the optimal percentage of features with grid search
clf = GridSearchCV(clf, {'anova__percentile': [5, 10, 20]}, cv=cv)
clf.fit(X, y)  # set the best parameters
coef_ = clf.best_estimator_.steps[-1][1].coef_
```

(continues on next page)

(continued from previous page)

```
coef_ = clf.best_estimator_.steps[0][1].inverse_transform(coef_.reshape(1, -1))
coef_selection_ = coef_.reshape(size, size)

# ############################################################################
# Inverse the transformation to plot the results on an image
plt.close('all')
plt.figure(figsize=(7.3, 2.7))
plt.subplot(1, 3, 1)
plt.imshow(coef, interpolation="nearest", cmap=plt.cm.RdBu_r)
plt.title("True weights")
plt.subplot(1, 3, 2)
plt.imshow(coef_selection_, interpolation="nearest", cmap=plt.cm.RdBu_r)
plt.title("Feature Selection")
plt.subplot(1, 3, 3)
plt.imshow(coef_agglomeration_, interpolation="nearest", cmap=plt.cm.RdBu_r)
plt.title("Feature Agglomeration")
plt.subplots_adjust(0.04, 0.0, 0.98, 0.94, 0.16, 0.26)
plt.show()

# Attempt to remove the temporary cachedir, but don't worry if it fails
shutil.rmtree(cachedir, ignore_errors=True)
```

Total running time of the script: (0 minutes 0.946 seconds)

Estimated memory usage: 9 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

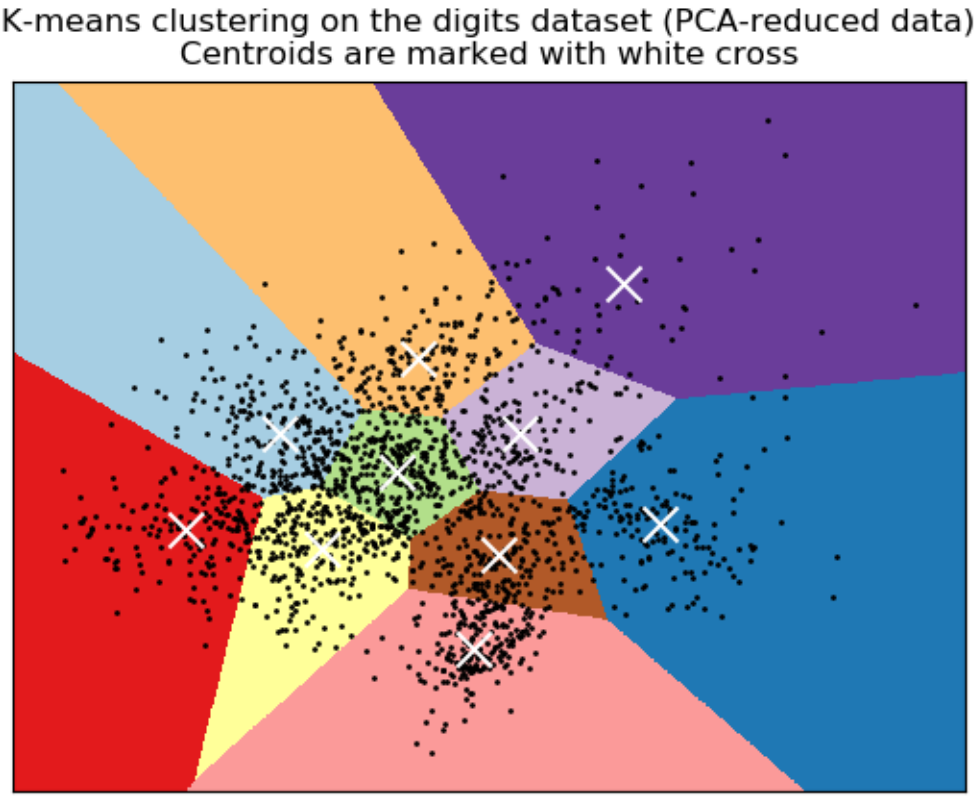
6.5.25 A demo of K-Means clustering on the handwritten digits data

In this example we compare the various initialization strategies for K-means in terms of runtime and quality of the results.

As the ground truth is known here, we also apply different cluster quality metrics to judge the goodness of fit of the cluster labels to the ground truth.

Cluster quality metrics evaluated (see *Clustering performance evaluation* for definitions and discussions of the metrics):

Shorthand	full name
homo	homogeneity score
compl	completeness score
v-meas	V measure
ARI	adjusted Rand index
AMI	adjusted mutual information
silhouette	silhouette coefficient



Out:

```
n_digits: 10,    n_samples 1797,         n_features 64
_____________________________________________________________________________________
init              time    inertia homo    compl   v-meas  ARI     AMI     silhouette
k-means++         0.24s   69510   0.610   0.657   0.633   0.481   0.629   0.129
random            0.24s   69907   0.633   0.674   0.653   0.518   0.649   0.131
PCA-based         0.03s   70768   0.668   0.695   0.681   0.558   0.678   0.142
_____________________________________________________________________________________
```

```
print(__doc__)

from time import time
import numpy as np
import matplotlib.pyplot as plt

from sklearn import metrics
from sklearn.cluster import KMeans
from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
```

(continues on next page)

(continued from previous page)

```
from sklearn.preprocessing import scale

np.random.seed(42)

X_digits, y_digits = load_digits(return_X_y=True)
data = scale(X_digits)

n_samples, n_features = data.shape
n_digits = len(np.unique(y_digits))
labels = y_digits

sample_size = 300

print("n_digits: %d, \t n_samples %d, \t n_features %d"
      % (n_digits, n_samples, n_features))


print(82 * '_')
print('init\t\ttime\tinertia\thomo\tcompl\tv-meas\tARI\tAMI\tsilhouette')


def bench_k_means(estimator, name, data):
    t0 = time()
    estimator.fit(data)
    print('%-9s\t%.2fs\t%i\t%.3f\t%.3f\t%.3f\t%.3f\t%.3f\t%.3f'
          % (name, (time() - t0), estimator.inertia_,
             metrics.homogeneity_score(labels, estimator.labels_),
             metrics.completeness_score(labels, estimator.labels_),
             metrics.v_measure_score(labels, estimator.labels_),
             metrics.adjusted_rand_score(labels, estimator.labels_),
             metrics.adjusted_mutual_info_score(labels,  estimator.labels_),
             metrics.silhouette_score(data, estimator.labels_,
                                      metric='euclidean',
                                      sample_size=sample_size)))

bench_k_means(KMeans(init='k-means++', n_clusters=n_digits, n_init=10),
              name="k-means++", data=data)

bench_k_means(KMeans(init='random', n_clusters=n_digits, n_init=10),
              name="random", data=data)

# in this case the seeding of the centers is deterministic, hence we run the
# kmeans algorithm only once with n_init=1
pca = PCA(n_components=n_digits).fit(data)
bench_k_means(KMeans(init=pca.components_, n_clusters=n_digits, n_init=1),
              name="PCA-based",
              data=data)
print(82 * '_')

# #############################################################################
# Visualize the results on PCA-reduced data

reduced_data = PCA(n_components=2).fit_transform(data)
kmeans = KMeans(init='k-means++', n_clusters=n_digits, n_init=10)
kmeans.fit(reduced_data)

# Step size of the mesh. Decrease to increase the quality of the VQ.
```

(continues on next page)

(continued from previous page)

```
h = .02     # point in the mesh [x_min, x_max]x[y_min, y_max].

# Plot the decision boundary. For that, we will assign a color to each
x_min, x_max = reduced_data[:, 0].min() - 1, reduced_data[:, 0].max() + 1
y_min, y_max = reduced_data[:, 1].min() - 1, reduced_data[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

# Obtain labels for each point in mesh. Use last trained model.
Z = kmeans.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure(1)
plt.clf()
plt.imshow(Z, interpolation='nearest',
           extent=(xx.min(), xx.max(), yy.min(), yy.max()),
           cmap=plt.cm.Paired,
           aspect='auto', origin='lower')

plt.plot(reduced_data[:, 0], reduced_data[:, 1], 'k.', markersize=2)
# Plot the centroids as a white X
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:, 0], centroids[:, 1],
            marker='x', s=169, linewidths=3,
            color='w', zorder=10)
plt.title('K-means clustering on the digits dataset (PCA-reduced data)\n'
          'Centroids are marked with white cross')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
plt.show()
```

Total running time of the script: (0 minutes 1.122 seconds)

Estimated memory usage: 46 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.26 Comparing different hierarchical linkage methods on toy datasets

This example shows characteristics of different linkage methods for hierarchical clustering on datasets that are "interesting" but still in 2D.

The main observations to make are:

- single linkage is fast, and can perform well on non-globular data, but it performs poorly in the presence of noise.
- average and complete linkage perform well on cleanly separated globular clusters, but have mixed results otherwise.
- Ward is the most effective method for noisy data.

While these examples give some intuition about the algorithms, this intuition might not apply to very high dimensional data.

```
print(__doc__)

import time
import warnings

import numpy as np
import matplotlib.pyplot as plt

from sklearn import cluster, datasets
from sklearn.preprocessing import StandardScaler
from itertools import cycle, islice

np.random.seed(0)
```

Generate datasets. We choose the size big enough to see the scalability of the algorithms, but not too big to avoid too long running times

```
n_samples = 1500
noisy_circles = datasets.make_circles(n_samples=n_samples, factor=.5,
                                      noise=.05)
noisy_moons = datasets.make_moons(n_samples=n_samples, noise=.05)
blobs = datasets.make_blobs(n_samples=n_samples, random_state=8)
no_structure = np.random.rand(n_samples, 2), None

# Anisotropicly distributed data
random_state = 170
X, y = datasets.make_blobs(n_samples=n_samples, random_state=random_state)
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X_aniso = np.dot(X, transformation)
aniso = (X_aniso, y)

# blobs with varied variances
varied = datasets.make_blobs(n_samples=n_samples,
                             cluster_std=[1.0, 2.5, 0.5],
                             random_state=random_state)
```

Run the clustering and plot

```
# Set up cluster parameters
plt.figure(figsize=(9 * 1.3 + 2, 14.5))
plt.subplots_adjust(left=.02, right=.98, bottom=.001, top=.96, wspace=.05,
                    hspace=.01)

plot_num = 1

default_base = {'n_neighbors': 10,
                'n_clusters': 3}

datasets = [
    (noisy_circles, {'n_clusters': 2}),
    (noisy_moons, {'n_clusters': 2}),
    (varied, {'n_neighbors': 2}),
    (aniso, {'n_neighbors': 2}),
    (blobs, {}),
    (no_structure, {})]

for i_dataset, (dataset, algo_params) in enumerate(datasets):
```

(continues on next page)

(continued from previous page)

```
    # update parameters with dataset-specific values
    params = default_base.copy()
    params.update(algo_params)

    X, y = dataset

    # normalize dataset for easier parameter selection
    X = StandardScaler().fit_transform(X)

    # ============
    # Create cluster objects
    # ============
    ward = cluster.AgglomerativeClustering(
        n_clusters=params['n_clusters'], linkage='ward')
    complete = cluster.AgglomerativeClustering(
        n_clusters=params['n_clusters'], linkage='complete')
    average = cluster.AgglomerativeClustering(
        n_clusters=params['n_clusters'], linkage='average')
    single = cluster.AgglomerativeClustering(
        n_clusters=params['n_clusters'], linkage='single')

    clustering_algorithms = (
        ('Single Linkage', single),
        ('Average Linkage', average),
        ('Complete Linkage', complete),
        ('Ward Linkage', ward),
    )

    for name, algorithm in clustering_algorithms:
        t0 = time.time()

        # catch warnings related to kneighbors_graph
        with warnings.catch_warnings():
            warnings.filterwarnings(
                "ignore",
                message="the number of connected components of the " +
                "connectivity matrix is [0-9]{1,2}" +
                " > 1. Completing it to avoid stopping the tree early.",
                category=UserWarning)
            algorithm.fit(X)

        t1 = time.time()
        if hasattr(algorithm, 'labels_'):
            y_pred = algorithm.labels_.astype(np.int)
        else:
            y_pred = algorithm.predict(X)

        plt.subplot(len(datasets), len(clustering_algorithms), plot_num)
        if i_dataset == 0:
            plt.title(name, size=18)

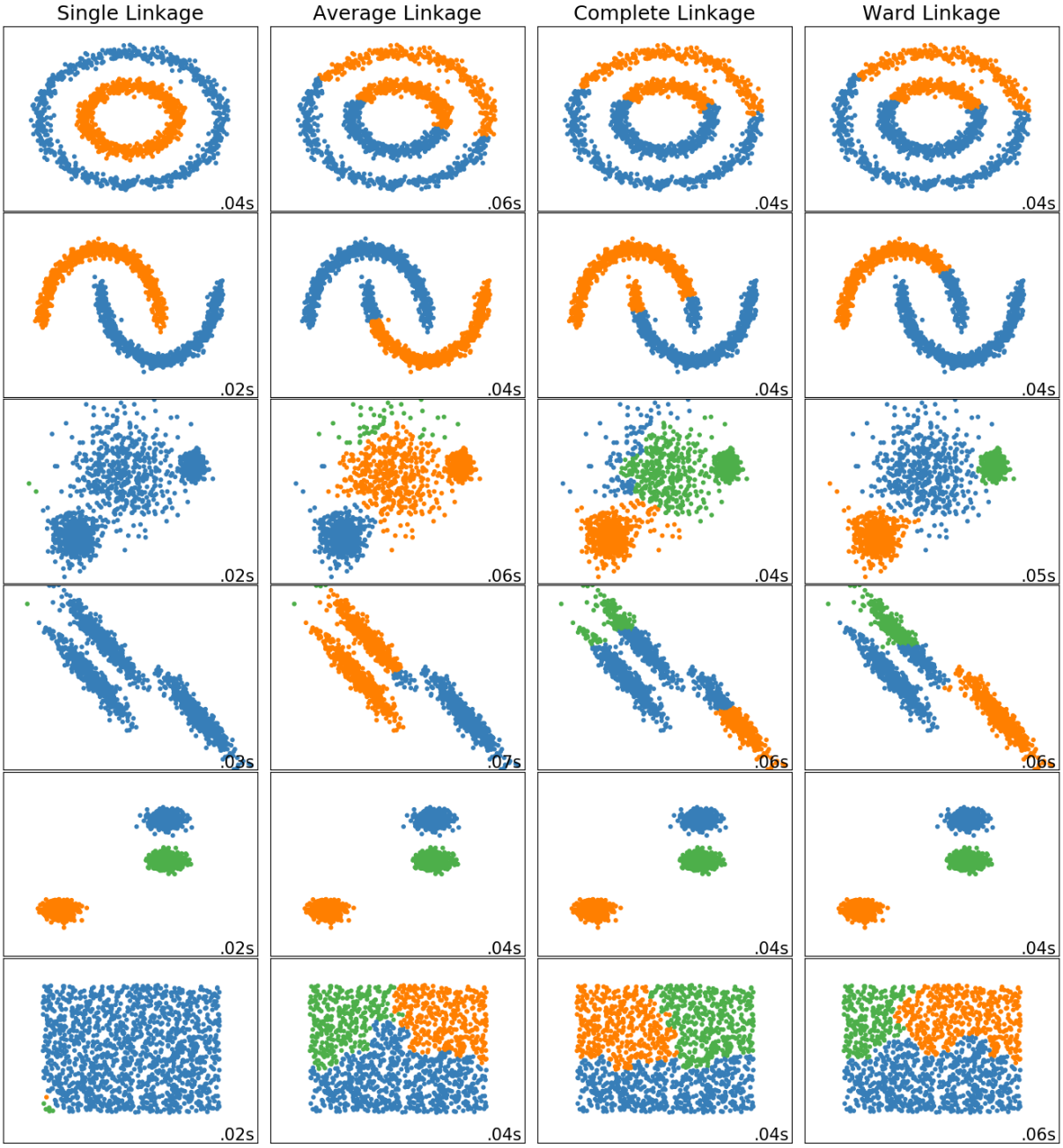
        colors = np.array(list(islice(cycle(['#377eb8', '#ff7f00', '#4daf4a',
                                             '#f781bf', '#a65628', '#984ea3',
                                             '#999999', '#e41a1c', '#dede00']),
                                      int(max(y_pred) + 1))))
        plt.scatter(X[:, 0], X[:, 1], s=10, color=colors[y_pred])
```

(continues on next page)

(continued from previous page)

```
        plt.xlim(-2.5, 2.5)
        plt.ylim(-2.5, 2.5)
        plt.xticks(())
        plt.yticks(())
        plt.text(.99, .01, ('%.2fs' % (t1 - t0)).lstrip('0'),
                 transform=plt.gca().transAxes, size=15,
                 horizontalalignment='right')
        plot_num += 1

plt.show()
```



Total running time of the script: (0 minutes 2.983 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

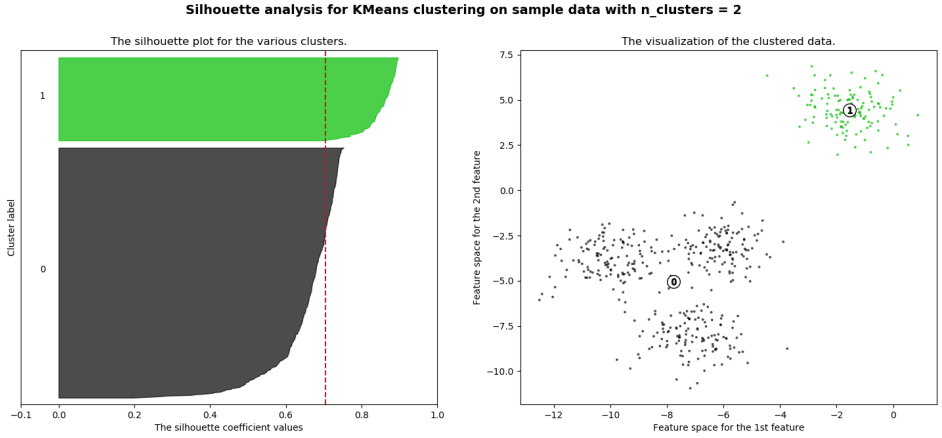
6.5.27 Selecting the number of clusters with silhouette analysis on KMeans clustering

Silhouette analysis can be used to study the separation distance between the resulting clusters. The silhouette plot displays a measure of how close each point in one cluster is to points in the neighboring clusters and thus provides a way to assess parameters like number of clusters visually. This measure has a range of [-1, 1].

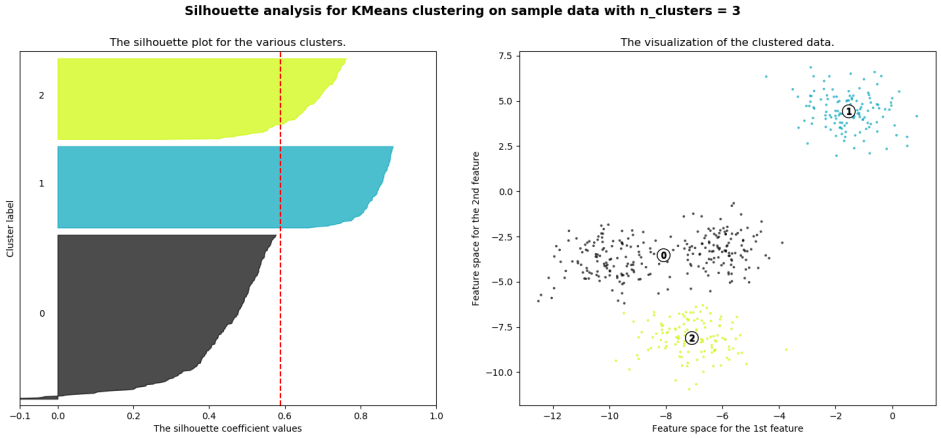
Silhouette coefficients (as these values are referred to as) near +1 indicate that the sample is far away from the neighboring clusters. A value of 0 indicates that the sample is on or very close to the decision boundary between two neighboring clusters and negative values indicate that those samples might have been assigned to the wrong cluster.

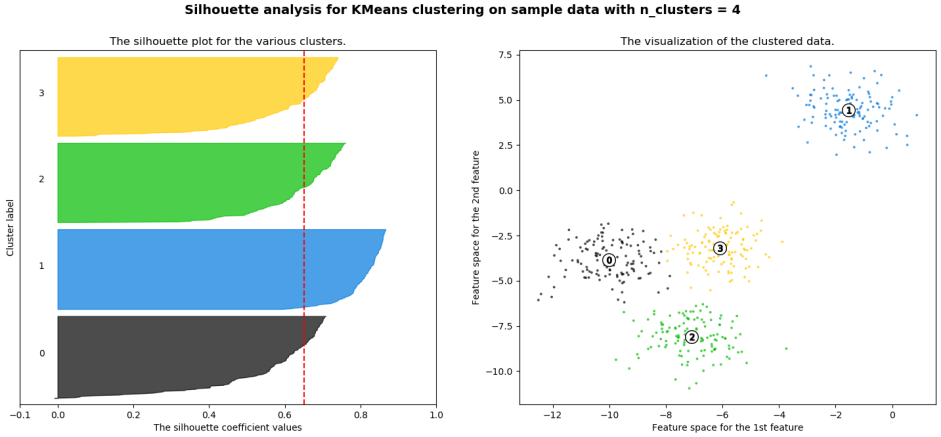
In this example the silhouette analysis is used to choose an optimal value for `n_clusters`. The silhouette plot shows that the `n_clusters` value of 3, 5 and 6 are a bad pick for the given data due to the presence of clusters with below average silhouette scores and also due to wide fluctuations in the size of the silhouette plots. Silhouette analysis is more ambivalent in deciding between 2 and 4.

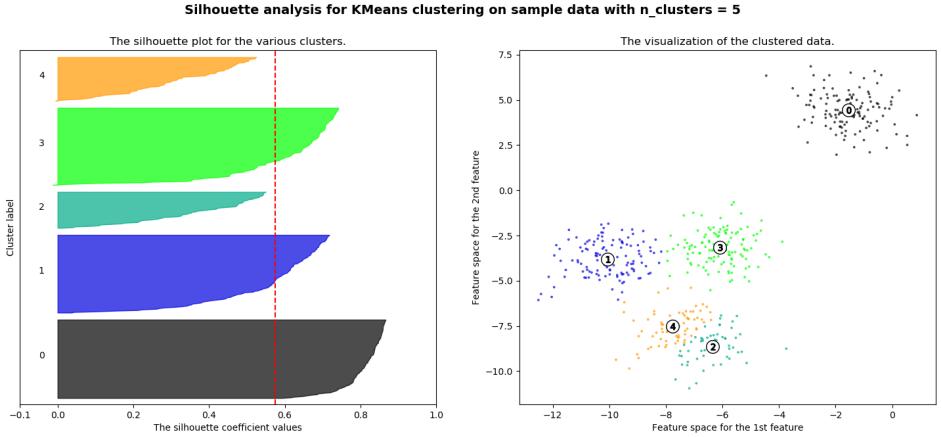
Also from the thickness of the silhouette plot the cluster size can be visualized. The silhouette plot for cluster 0 when `n_clusters` is equal to 2, is bigger in size owing to the grouping of the 3 sub clusters into one big cluster. However when the `n_clusters` is equal to 4, all the plots are more or less of similar thickness and hence are of similar sizes as can be also verified from the labelled scatter plot on the right.

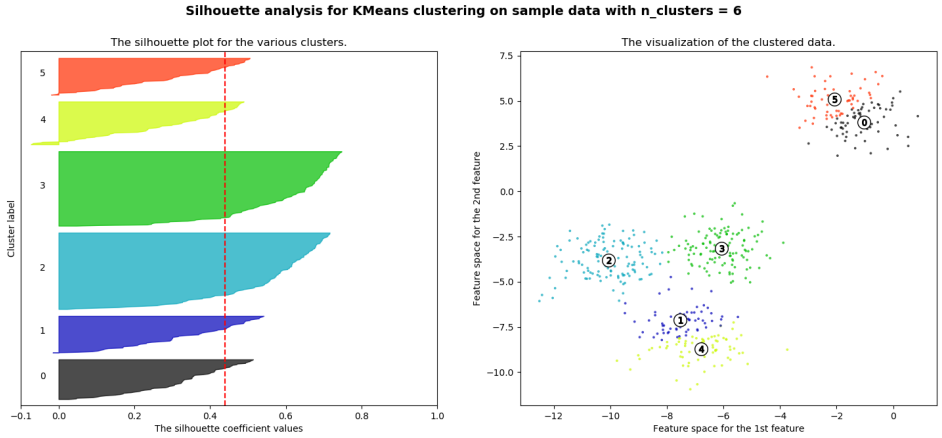


-


Silhouette analysis for KMeans clustering on sample data with n_clusters = 3
The silhouette plot for the various clusters.
Cluster label
The silhouette coefficient values
The visualization of the clustered data.
Feature space for the 2nd feature
Feature space for the 1st feature


Silhouette analysis for KMeans clustering on sample data with n_clusters = 4
The silhouette plot for the various clusters.
Cluster label
The silhouette coefficient values
The visualization of the clustered data.
Feature space for the 2nd feature
Feature space for the 1st feature


Silhouette analysis for KMeans clustering on sample data with n_clusters = 5
The silhouette plot for the various clusters.
Cluster label
The silhouette coefficient values
The visualization of the clustered data.
Feature space for the 2nd feature
Feature space for the 1st feature



-

Out:

```
For n_clusters = 2 The average silhouette_score is : 0.7049787496083262
For n_clusters = 3 The average silhouette_score is : 0.5882004012129721
For n_clusters = 4 The average silhouette_score is : 0.6505186632729437
For n_clusters = 5 The average silhouette_score is : 0.5745566973301872
For n_clusters = 6 The average silhouette_score is : 0.43902711183132426
```

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score

import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np

print(__doc__)

# Generating the sample data from make_blobs
# This particular setting has one distinct cluster and 3 clusters placed close
# together.
X, y = make_blobs(n_samples=500,
                  n_features=2,
                  centers=4,
                  cluster_std=1,
                  center_box=(-10.0, 10.0),
                  shuffle=True,
                  random_state=1)  # For reproducibility

range_n_clusters = [2, 3, 4, 5, 6]

for n_clusters in range_n_clusters:
    # Create a subplot with 1 row and 2 columns
    fig, (ax1, ax2) = plt.subplots(1, 2)
    fig.set_size_inches(18, 7)
```

(continues on next page)

(continued from previous page)

```
    # The 1st subplot is the silhouette plot
    # The silhouette coefficient can range from -1, 1 but in this example all
    # lie within [-0.1, 1]
    ax1.set_xlim([-0.1, 1])
    # The (n_clusters+1)*10 is for inserting blank space between silhouette
    # plots of individual clusters, to demarcate them clearly.
    ax1.set_ylim([0, len(X) + (n_clusters + 1) * 10])

    # Initialize the clusterer with n_clusters value and a random generator
    # seed of 10 for reproducibility.
    clusterer = KMeans(n_clusters=n_clusters, random_state=10)
    cluster_labels = clusterer.fit_predict(X)

    # The silhouette_score gives the average value for all the samples.
    # This gives a perspective into the density and separation of the formed
    # clusters
    silhouette_avg = silhouette_score(X, cluster_labels)
    print("For n_clusters =", n_clusters,
          "The average silhouette_score is :", silhouette_avg)

    # Compute the silhouette scores for each sample
    sample_silhouette_values = silhouette_samples(X, cluster_labels)

    y_lower = 10
    for i in range(n_clusters):
        # Aggregate the silhouette scores for samples belonging to
        # cluster i, and sort them
        ith_cluster_silhouette_values = \
            sample_silhouette_values[cluster_labels == i]

        ith_cluster_silhouette_values.sort()

        size_cluster_i = ith_cluster_silhouette_values.shape[0]
        y_upper = y_lower + size_cluster_i

        color = cm.nipy_spectral(float(i) / n_clusters)
        ax1.fill_betweenx(np.arange(y_lower, y_upper),
                          0, ith_cluster_silhouette_values,
                          facecolor=color, edgecolor=color, alpha=0.7)

        # Label the silhouette plots with their cluster numbers at the middle
        ax1.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))

        # Compute the new y_lower for next plot
        y_lower = y_upper + 10  # 10 for the 0 samples

    ax1.set_title("The silhouette plot for the various clusters.")
    ax1.set_xlabel("The silhouette coefficient values")
    ax1.set_ylabel("Cluster label")

    # The vertical line for average silhouette score of all the values
    ax1.axvline(x=silhouette_avg, color="red", linestyle="--")

    ax1.set_yticks([])  # Clear the yaxis labels / ticks
    ax1.set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])

    # 2nd Plot showing the actual clusters formed
```

(continues on next page)

(continued from previous page)

```
    colors = cm.nipy_spectral(cluster_labels.astype(float) / n_clusters)
    ax2.scatter(X[:, 0], X[:, 1], marker='.', s=30, lw=0, alpha=0.7,
                c=colors, edgecolor='k')

    # Labeling the clusters
    centers = clusterer.cluster_centers_
    # Draw white circles at cluster centers
    ax2.scatter(centers[:, 0], centers[:, 1], marker='o',
                c="white", alpha=1, s=200, edgecolor='k')

    for i, c in enumerate(centers):
        ax2.scatter(c[0], c[1], marker='$%d$' % i, alpha=1,
                    s=50, edgecolor='k')

    ax2.set_title("The visualization of the clustered data.")
    ax2.set_xlabel("Feature space for the 1st feature")
    ax2.set_ylabel("Feature space for the 2nd feature")

    plt.suptitle(("Silhouette analysis for KMeans clustering on sample data "
                  "with n_clusters = %d" % n_clusters),
                 fontsize=14, fontweight='bold')

plt.show()
```

Total running time of the script: (0 minutes 1.106 seconds)

Estimated memory usage: 8 MB

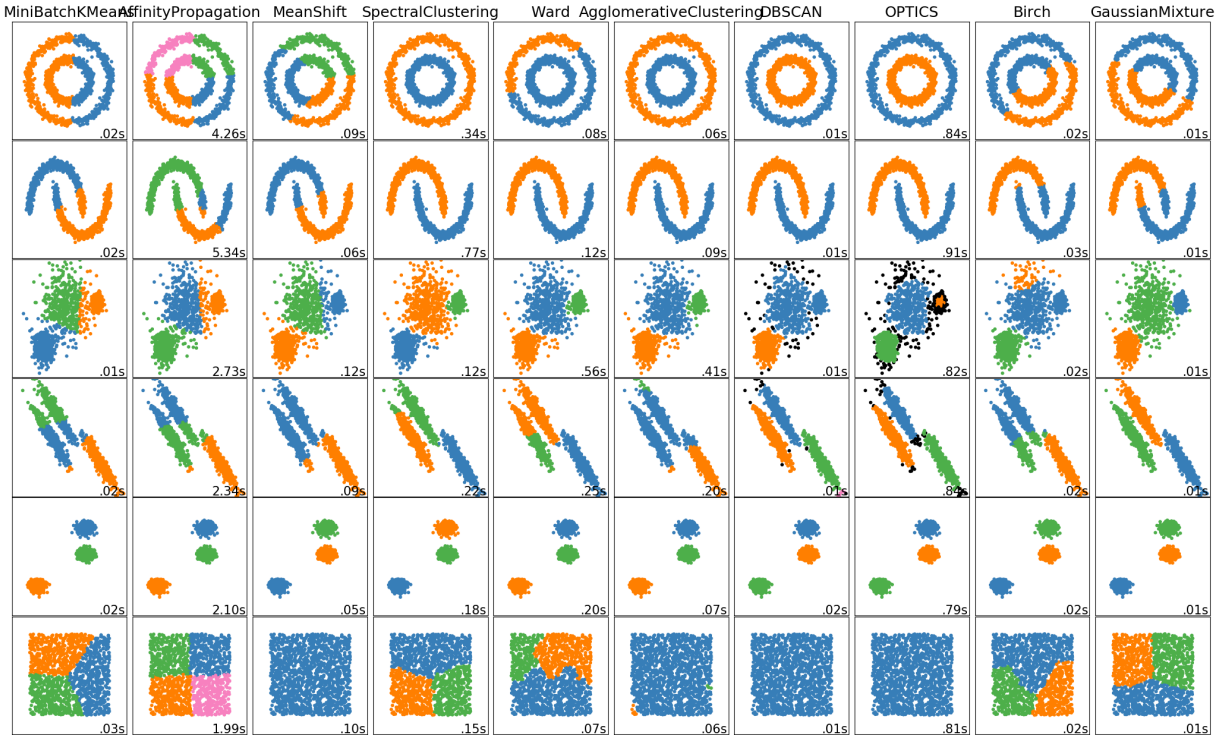
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.5.28 Comparing different clustering algorithms on toy datasets

This example shows characteristics of different clustering algorithms on datasets that are "interesting" but still in 2D. With the exception of the last dataset, the parameters of each of these dataset-algorithm pairs has been tuned to produce good clustering results. Some algorithms are more sensitive to parameter values than others.

The last dataset is an example of a 'null' situation for clustering: the data is homogeneous, and there is no good clustering. For this example, the null dataset uses the same parameters as the dataset in the row above it, which represents a mismatch in the parameter values and the data structure.

While these examples give some intuition about the algorithms, this intuition might not apply to very high dimensional data.




```
print(__doc__)

import time
import warnings

import numpy as np
import matplotlib.pyplot as plt

from sklearn import cluster, datasets, mixture
from sklearn.neighbors import kneighbors_graph
from sklearn.preprocessing import StandardScaler
from itertools import cycle, islice

np.random.seed(0)

# ============
# Generate datasets. We choose the size big enough to see the scalability
# of the algorithms, but not too big to avoid too long running times
# ============
n_samples = 1500
noisy_circles = datasets.make_circles(n_samples=n_samples, factor=.5,
                                      noise=.05)
noisy_moons = datasets.make_moons(n_samples=n_samples, noise=.05)
blobs = datasets.make_blobs(n_samples=n_samples, random_state=8)
no_structure = np.random.rand(n_samples, 2), None

# Anisotropicly distributed data
random_state = 170
X, y = datasets.make_blobs(n_samples=n_samples, random_state=random_state)
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X_aniso = np.dot(X, transformation)
```

(continues on next page)

(continued from previous page)

```
aniso = (X_aniso, y)

# blobs with varied variances
varied = datasets.make_blobs(n_samples=n_samples,
                             cluster_std=[1.0, 2.5, 0.5],
                             random_state=random_state)

# ============
# Set up cluster parameters
# ============
plt.figure(figsize=(9 * 2 + 3, 12.5))
plt.subplots_adjust(left=.02, right=.98, bottom=.001, top=.96, wspace=.05,
                    hspace=.01)

plot_num = 1

default_base = {'quantile': .3,
                'eps': .3,
                'damping': .9,
                'preference': -200,
                'n_neighbors': 10,
                'n_clusters': 3,
                'min_samples': 20,
                'xi': 0.05,
                'min_cluster_size': 0.1}

datasets = [
    (noisy_circles, {'damping': .77, 'preference': -240,
                     'quantile': .2, 'n_clusters': 2,
                     'min_samples': 20, 'xi': 0.25}),
    (noisy_moons, {'damping': .75, 'preference': -220, 'n_clusters': 2}),
    (varied, {'eps': .18, 'n_neighbors': 2,
              'min_samples': 5, 'xi': 0.035, 'min_cluster_size': .2}),
    (aniso, {'eps': .15, 'n_neighbors': 2,
             'min_samples': 20, 'xi': 0.1, 'min_cluster_size': .2}),
    (blobs, {}),
    (no_structure, {})]

for i_dataset, (dataset, algo_params) in enumerate(datasets):
    # update parameters with dataset-specific values
    params = default_base.copy()
    params.update(algo_params)

    X, y = dataset

    # normalize dataset for easier parameter selection
    X = StandardScaler().fit_transform(X)

    # estimate bandwidth for mean shift
    bandwidth = cluster.estimate_bandwidth(X, quantile=params['quantile'])

    # connectivity matrix for structured Ward
    connectivity = kneighbors_graph(
        X, n_neighbors=params['n_neighbors'], include_self=False)
    # make connectivity symmetric
    connectivity = 0.5 * (connectivity + connectivity.T)
```

(continues on next page)

(continued from previous page)

```
    # ============
    # Create cluster objects
    # ============
    ms = cluster.MeanShift(bandwidth=bandwidth, bin_seeding=True)
    two_means = cluster.MiniBatchKMeans(n_clusters=params['n_clusters'])
    ward = cluster.AgglomerativeClustering(
        n_clusters=params['n_clusters'], linkage='ward',
        connectivity=connectivity)
    spectral = cluster.SpectralClustering(
        n_clusters=params['n_clusters'], eigen_solver='arpack',
        affinity="nearest_neighbors")
    dbscan = cluster.DBSCAN(eps=params['eps'])
    optics = cluster.OPTICS(min_samples=params['min_samples'],
                            xi=params['xi'],
                            min_cluster_size=params['min_cluster_size'])
    affinity_propagation = cluster.AffinityPropagation(
        damping=params['damping'], preference=params['preference'])
    average_linkage = cluster.AgglomerativeClustering(
        linkage="average", affinity="cityblock",
        n_clusters=params['n_clusters'], connectivity=connectivity)
    birch = cluster.Birch(n_clusters=params['n_clusters'])
    gmm = mixture.GaussianMixture(
        n_components=params['n_clusters'], covariance_type='full')

    clustering_algorithms = (
        ('MiniBatchKMeans', two_means),
        ('AffinityPropagation', affinity_propagation),
        ('MeanShift', ms),
        ('SpectralClustering', spectral),
        ('Ward', ward),
        ('AgglomerativeClustering', average_linkage),
        ('DBSCAN', dbscan),
        ('OPTICS', optics),
        ('Birch', birch),
        ('GaussianMixture', gmm)
    )

    for name, algorithm in clustering_algorithms:
        t0 = time.time()

        # catch warnings related to kneighbors_graph
        with warnings.catch_warnings():
            warnings.filterwarnings(
                "ignore",
                message="the number of connected components of the " +
                "connectivity matrix is [0-9]{1,2}" +
                " > 1. Completing it to avoid stopping the tree early.",
                category=UserWarning)
            warnings.filterwarnings(
                "ignore",
                message="Graph is not fully connected, spectral embedding" +
                " may not work as expected.",
                category=UserWarning)
            algorithm.fit(X)

        t1 = time.time()
        if hasattr(algorithm, 'labels_'):
```

(continues on next page)

(continued from previous page)

```
            y_pred = algorithm.labels_.astype(np.int)
        else:
            y_pred = algorithm.predict(X)

        plt.subplot(len(datasets), len(clustering_algorithms), plot_num)
        if i_dataset == 0:
            plt.title(name, size=18)

        colors = np.array(list(islice(cycle(['#377eb8', '#ff7f00', '#4daf4a',
                                             '#f781bf', '#a65628', '#984ea3',
                                             '#999999', '#e41a1c', '#dede00']),
                                      int(max(y_pred) + 1))))
        # add black color for outliers (if any)
        colors = np.append(colors, ["#000000"])
        plt.scatter(X[:, 0], X[:, 1], s=10, color=colors[y_pred])

        plt.xlim(-2.5, 2.5)
        plt.ylim(-2.5, 2.5)
        plt.xticks(())
        plt.yticks(())
        plt.text(.99, .01, ('%.2fs' % (t1 - t0)).lstrip('0'),
                 transform=plt.gca().transAxes, size=15,
                 horizontalalignment='right')
        plot_num += 1

plt.show()
```

Total running time of the script: (0 minutes 32.328 seconds)

Estimated memory usage: 75 MB

6.6 Covariance estimation

Examples concerning the *sklearn.covariance* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.6.1 Ledoit-Wolf vs OAS estimation

The usual covariance maximum likelihood estimate can be regularized using shrinkage. Ledoit and Wolf proposed a close formula to compute the asymptotically optimal shrinkage parameter (minimizing a MSE criterion), yielding the Ledoit-Wolf covariance estimate.

Chen et al. proposed an improvement of the Ledoit-Wolf shrinkage parameter, the OAS coefficient, whose convergence is significantly better under the assumption that the data are Gaussian.

This example, inspired from Chen's publication [1], shows a comparison of the estimated MSE of the LW and OAS methods, using Gaussian distributed data.

[1] "Shrinkage Algorithms for MMSE Covariance Estimation" Chen et al., IEEE Trans. on Sign. Proc., Volume 58, Issue 10, October 2010.

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import toeplitz, cholesky

from sklearn.covariance import LedoitWolf, OAS

np.random.seed(0)
```

```
n_features = 100
# simulation covariance matrix (AR(1) process)
r = 0.1
real_cov = toeplitz(r ** np.arange(n_features))
coloring_matrix = cholesky(real_cov)

n_samples_range = np.arange(6, 31, 1)
repeat = 100
lw_mse = np.zeros((n_samples_range.size, repeat))
oa_mse = np.zeros((n_samples_range.size, repeat))
lw_shrinkage = np.zeros((n_samples_range.size, repeat))
oa_shrinkage = np.zeros((n_samples_range.size, repeat))
for i, n_samples in enumerate(n_samples_range):
    for j in range(repeat):
        X = np.dot(
            np.random.normal(size=(n_samples, n_features)), coloring_matrix.T)

        lw = LedoitWolf(store_precision=False, assume_centered=True)
        lw.fit(X)
        lw_mse[i, j] = lw.error_norm(real_cov, scaling=False)
        lw_shrinkage[i, j] = lw.shrinkage_

        oa = OAS(store_precision=False, assume_centered=True)
        oa.fit(X)
        oa_mse[i, j] = oa.error_norm(real_cov, scaling=False)
        oa_shrinkage[i, j] = oa.shrinkage_

# plot MSE
plt.subplot(2, 1, 1)
plt.errorbar(n_samples_range, lw_mse.mean(1), yerr=lw_mse.std(1),
             label='Ledoit-Wolf', color='navy', lw=2)
plt.errorbar(n_samples_range, oa_mse.mean(1), yerr=oa_mse.std(1),
             label='OAS', color='darkorange', lw=2)
plt.ylabel("Squared error")
plt.legend(loc="upper right")
plt.title("Comparison of covariance estimators")
plt.xlim(5, 31)

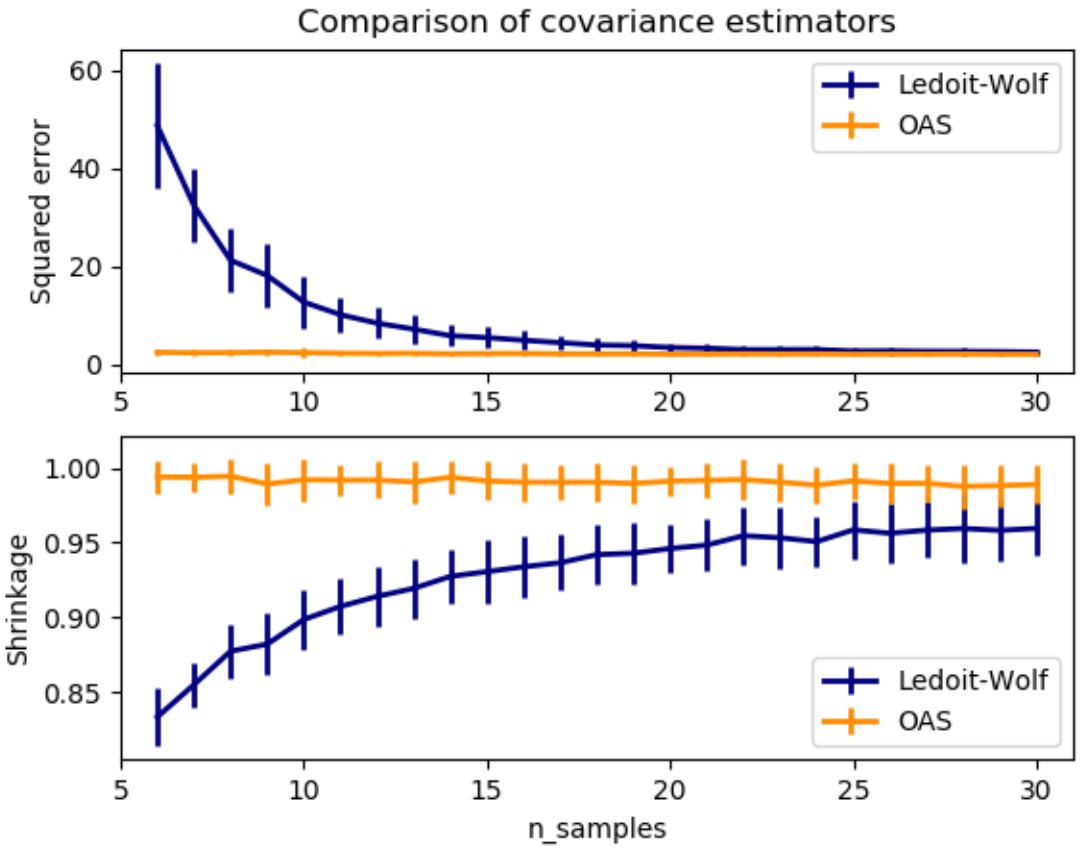
# plot shrinkage coefficient
plt.subplot(2, 1, 2)
plt.errorbar(n_samples_range, lw_shrinkage.mean(1), yerr=lw_shrinkage.std(1),
             label='Ledoit-Wolf', color='navy', lw=2)
plt.errorbar(n_samples_range, oa_shrinkage.mean(1), yerr=oa_shrinkage.std(1),
             label='OAS', color='darkorange', lw=2)
plt.xlabel("n_samples")
plt.ylabel("Shrinkage")
plt.legend(loc="lower right")
```

(continues on next page)

(continued from previous page)

```
plt.ylim(plt.ylim()[0], 1. + (plt.ylim()[1] - plt.ylim()[0]) / 10.)
plt.xlim(5, 31)

plt.show()
```



Total running time of the script: (0 minutes 2.037 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.6.2 Sparse inverse covariance estimation

Using the GraphicalLasso estimator to learn a covariance and sparse precision from a small number of samples.

To estimate a probabilistic model (e.g. a Gaussian model), estimating the precision matrix, that is the inverse covariance matrix, is as important as estimating the covariance matrix. Indeed a Gaussian model is parametrized by the precision matrix.

To be in favorable recovery conditions, we sample the data from a model with a sparse inverse covariance matrix. In addition, we ensure that the data is not too much correlated (limiting the largest coefficient of the precision matrix) and that there a no small coefficients in the precision matrix that cannot be recovered. In addition, with a small number of observations, it is easier to recover a correlation matrix rather than a covariance, thus we scale the time series.

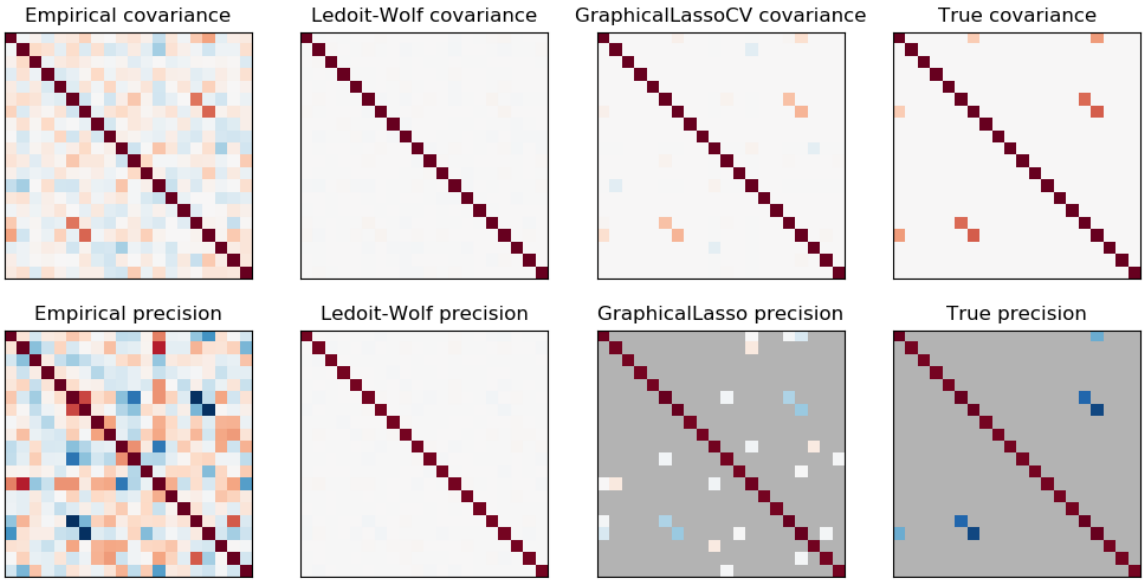
Here, the number of samples is slightly larger than the number of dimensions, thus the empirical covariance is still invertible. However, as the observations are strongly correlated, the empirical covariance matrix is ill-conditioned and as a result its inverse –the empirical precision matrix– is very far from the ground truth.

If we use l2 shrinkage, as with the Ledoit-Wolf estimator, as the number of samples is small, we need to shrink a lot. As a result, the Ledoit-Wolf precision is fairly close to the ground truth precision, that is not far from being diagonal, but the off-diagonal structure is lost.

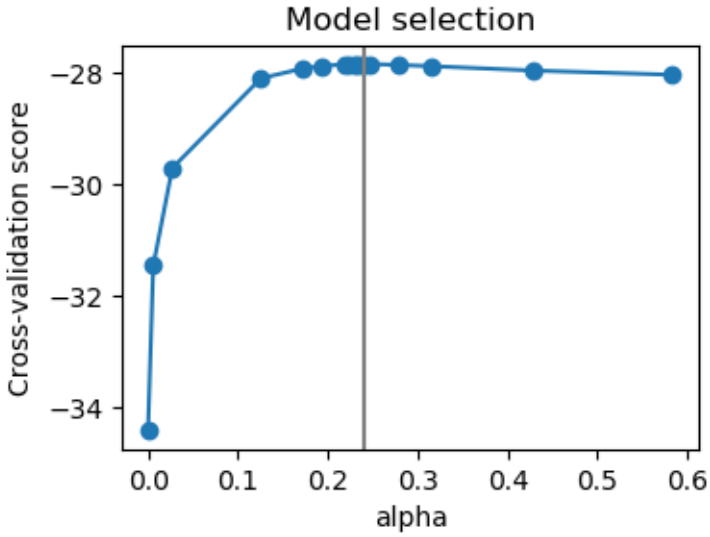
The l1-penalized estimator can recover part of this off-diagonal structure. It learns a sparse precision. It is not able to recover the exact sparsity pattern: it detects too many non-zero coefficients. However, the highest non-zero coefficients of the l1 estimated correspond to the non-zero coefficients in the ground truth. Finally, the coefficients of the l1 precision estimate are biased toward zero: because of the penalty, they are all smaller than the corresponding ground truth value, as can be seen on the figure.

Note that, the color range of the precision matrices is tweaked to improve readability of the figure. The full range of values of the empirical precision is not displayed.

The alpha parameter of the GraphicalLasso setting the sparsity of the model is set by internal cross-validation in the GraphicalLassoCV. As can be seen on figure 2, the grid to compute the cross-validation score is iteratively refined in the neighborhood of the maximum.



-




```
print(__doc__)
# author: Gael Varoquaux <gael.varoquaux@inria.fr>
# License: BSD 3 clause
# Copyright: INRIA

import numpy as np
from scipy import linalg
from sklearn.datasets import make_sparse_spd_matrix
from sklearn.covariance import GraphicalLassoCV, ledoit_wolf
import matplotlib.pyplot as plt

# #############################################################################
# Generate the data
n_samples = 60
n_features = 20

prng = np.random.RandomState(1)
prec = make_sparse_spd_matrix(n_features, alpha=.98,
                              smallest_coef=.4,
                              largest_coef=.7,
                              random_state=prng)
cov = linalg.inv(prec)
d = np.sqrt(np.diag(cov))
cov /= d
cov /= d[:, np.newaxis]
prec *= d
prec *= d[:, np.newaxis]
X = prng.multivariate_normal(np.zeros(n_features), cov, size=n_samples)
X -= X.mean(axis=0)
X /= X.std(axis=0)

# #############################################################################
# Estimate the covariance
emp_cov = np.dot(X.T, X) / n_samples

model = GraphicalLassoCV()
model.fit(X)
```

(continues on next page)

(continued from previous page)

```
cov_ = model.covariance_
prec_ = model.precision_

lw_cov_, _ = ledoit_wolf(X)
lw_prec_ = linalg.inv(lw_cov_)

# ############################################################################
# Plot the results
plt.figure(figsize=(10, 6))
plt.subplots_adjust(left=0.02, right=0.98)

# plot the covariances
covs = [('Empirical', emp_cov), ('Ledoit-Wolf', lw_cov_),
        ('GraphicalLassoCV', cov_), ('True', cov)]
vmax = cov_.max()
for i, (name, this_cov) in enumerate(covs):
    plt.subplot(2, 4, i + 1)
    plt.imshow(this_cov, interpolation='nearest', vmin=-vmax, vmax=vmax,
               cmap=plt.cm.RdBu_r)
    plt.xticks(())
    plt.yticks(())
    plt.title('%s covariance' % name)


# plot the precisions
precs = [('Empirical', linalg.inv(emp_cov)), ('Ledoit-Wolf', lw_prec_),
         ('GraphicalLasso', prec_), ('True', prec)]
vmax = .9 * prec_.max()
for i, (name, this_prec) in enumerate(precs):
    ax = plt.subplot(2, 4, i + 5)
    plt.imshow(np.ma.masked_equal(this_prec, 0),
               interpolation='nearest', vmin=-vmax, vmax=vmax,
               cmap=plt.cm.RdBu_r)
    plt.xticks(())
    plt.yticks(())
    plt.title('%s precision' % name)
    if hasattr(ax, 'set_facecolor'):
        ax.set_facecolor('.7')
    else:
        ax.set_axis_bgcolor('.7')

# plot the model selection metric
plt.figure(figsize=(4, 3))
plt.axes([.2, .15, .75, .7])
plt.plot(model.cv_alphas_, np.mean(model.grid_scores_, axis=1), 'o-')
plt.axvline(model.alpha_, color='.5')
plt.title('Model selection')
plt.ylabel('Cross-validation score')
plt.xlabel('alpha')

plt.show()
```

Total running time of the script: (0 minutes 0.810 seconds)

Estimated memory usage: 9 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.6.3 Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood

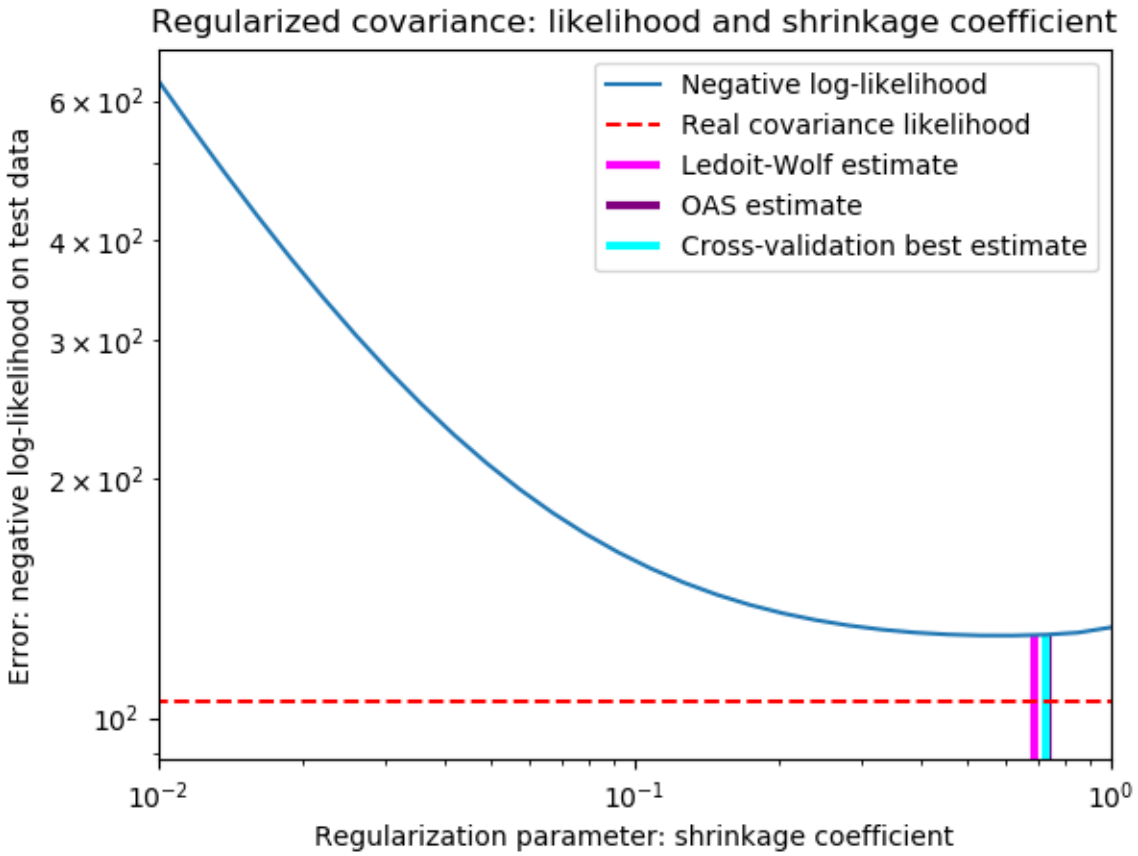
When working with covariance estimation, the usual approach is to use a maximum likelihood estimator, such as the `sklearn.covariance.EmpiricalCovariance`. It is unbiased, i.e. it converges to the true (population) covariance when given many observations. However, it can also be beneficial to regularize it, in order to reduce its variance; this, in turn, introduces some bias. This example illustrates the simple regularization used in *Shrunk Covariance* estimators. In particular, it focuses on how to set the amount of regularization, i.e. how to choose the bias-variance trade-off.

Here we compare 3 approaches:

- Setting the parameter by cross-validating the likelihood on three folds according to a grid of potential shrinkage parameters.
- A close formula proposed by Ledoit and Wolf to compute the asymptotically optimal regularization parameter (minimizing a MSE criterion), yielding the `sklearn.covariance.LedoitWolf` covariance estimate.
- An improvement of the Ledoit-Wolf shrinkage, the `sklearn.covariance.OAS`, proposed by Chen et al. Its convergence is significantly better under the assumption that the data are Gaussian, in particular for small samples.

To quantify estimation error, we plot the likelihood of unseen data for different values of the shrinkage parameter. We also show the choices by cross-validation, or with the LedoitWolf and OAS estimates.

Note that the maximum likelihood estimate corresponds to no shrinkage, and thus performs poorly. The Ledoit-Wolf estimate performs really well, as it is close to the optimal and is computational not costly. In this example, the OAS estimate is a bit further away. Interestingly, both approaches outperform cross-validation, which is significantly most computationally costly.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg

from sklearn.covariance import LedoitWolf, OAS, ShrunkCovariance, \
    log_likelihood, empirical_covariance
from sklearn.model_selection import GridSearchCV


# #############################################################################
# Generate sample data
n_features, n_samples = 40, 20
np.random.seed(42)
base_X_train = np.random.normal(size=(n_samples, n_features))
base_X_test = np.random.normal(size=(n_samples, n_features))

# Color samples
coloring_matrix = np.random.normal(size=(n_features, n_features))
X_train = np.dot(base_X_train, coloring_matrix)
X_test = np.dot(base_X_test, coloring_matrix)

# #############################################################################
# Compute the likelihood on test data
```

(continues on next page)

(continued from previous page)

```
# spanning a range of possible shrinkage coefficient values
shrinkages = np.logspace(-2, 0, 30)
negative_logliks = [-ShrunkCovariance(shrinkage=s).fit(X_train).score(X_test)
                    for s in shrinkages]

# under the ground-truth model, which we would not have access to in real
# settings
real_cov = np.dot(coloring_matrix.T, coloring_matrix)
emp_cov = empirical_covariance(X_train)
loglik_real = -log_likelihood(emp_cov, linalg.inv(real_cov))

# #############################################################################
# Compare different approaches to setting the parameter

# GridSearch for an optimal shrinkage coefficient
tuned_parameters = [{'shrinkage': shrinkages}]
cv = GridSearchCV(ShrunkCovariance(), tuned_parameters)
cv.fit(X_train)

# Ledoit-Wolf optimal shrinkage coefficient estimate
lw = LedoitWolf()
loglik_lw = lw.fit(X_train).score(X_test)

# OAS coefficient estimate
oa = OAS()
loglik_oa = oa.fit(X_train).score(X_test)

# #############################################################################
# Plot results
fig = plt.figure()
plt.title("Regularized covariance: likelihood and shrinkage coefficient")
plt.xlabel('Regularization parameter: shrinkage coefficient')
plt.ylabel('Error: negative log-likelihood on test data')
# range shrinkage curve
plt.loglog(shrinkages, negative_logliks, label="Negative log-likelihood")

plt.plot(plt.xlim(), 2 * [loglik_real], '--r',
         label="Real covariance likelihood")

# adjust view
lik_max = np.amax(negative_logliks)
lik_min = np.amin(negative_logliks)
ymin = lik_min - 6. * np.log((plt.ylim()[1] - plt.ylim()[0]))
ymax = lik_max + 10. * np.log(lik_max - lik_min)
xmin = shrinkages[0]
xmax = shrinkages[-1]
# LW likelihood
plt.vlines(lw.shrinkage_, ymin, -loglik_lw, color='magenta',
           linewidth=3, label='Ledoit-Wolf estimate')
# OAS likelihood
plt.vlines(oa.shrinkage_, ymin, -loglik_oa, color='purple',
           linewidth=3, label='OAS estimate')
# best CV estimator likelihood
plt.vlines(cv.best_estimator_.shrinkage, ymin,
           -cv.best_estimator_.score(X_test), color='cyan',
           linewidth=3, label='Cross-validation best estimate')
```

(continues on next page)

(continued from previous page)

```
plt.ylim(ymin, ymax)
plt.xlim(xmin, xmax)
plt.legend()

plt.show()
```

Total running time of the script: (0 minutes 0.546 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.6.4 Robust covariance estimation and Mahalanobis distances relevance

An example to show covariance estimation with the Mahalanobis distances on Gaussian distributed data.

For Gaussian distributed data, the distance of an observation x_i to the mode of the distribution can be computed using its Mahalanobis distance: $d_{(\mu,\Sigma)}(x_i)^2 = (x_i - \mu)'\Sigma^{-1}(x_i - \mu)$ where μ and Σ are the location and the covariance of the underlying Gaussian distribution.

In practice, μ and Σ are replaced by some estimates. The usual covariance maximum likelihood estimate is very sensitive to the presence of outliers in the data set and therefor, the corresponding Mahalanobis distances are. One would better have to use a robust estimator of covariance to guarantee that the estimation is resistant to "erroneous" observations in the data set and that the associated Mahalanobis distances accurately reflect the true organisation of the observations.

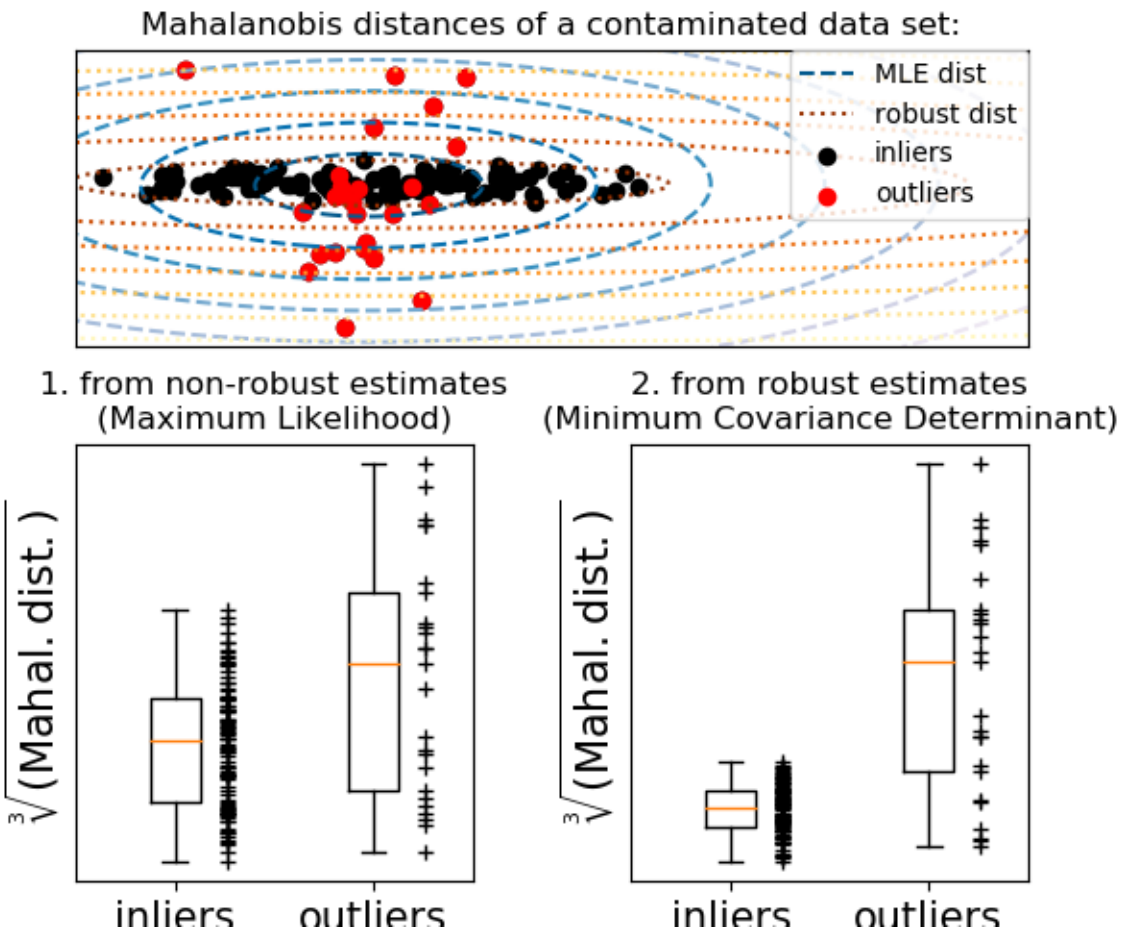
The Minimum Covariance Determinant estimator is a robust, high-breakdown point (i.e. it can be used to estimate the covariance matrix of highly contaminated datasets, up to $\frac{n_{\text{samples}} - n_{\text{features}} - 1}{2}$ outliers) estimator of covariance. The idea is to find $\frac{n_{\text{samples}} + n_{\text{features}} + 1}{2}$ observations whose empirical covariance has the smallest determinant, yielding a "pure" subset of observations from which to compute standards estimates of location and covariance.

The Minimum Covariance Determinant estimator (MCD) has been introduced by P.J.Rousseuw in [1].

This example illustrates how the Mahalanobis distances are affected by outlying data: observations drawn from a contaminating distribution are not distinguishable from the observations coming from the real, Gaussian distribution that one may want to work with. Using MCD-based Mahalanobis distances, the two populations become distinguishable. Associated applications are outliers detection, observations ranking, clustering, ... For visualization purpose, the cubic root of the Mahalanobis distances are represented in the boxplot, as Wilson and Hilferty suggest [2]

[1] P. J. Rousseeuw. Least median of squares regression. J. Am Stat Ass, 79:871, 1984.

[2] Wilson, E. B., & Hilferty, M. M. (1931). The distribution of chi-square. Proceedings of the National Academy of Sciences of the United States of America, 17, 684-688.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.covariance import EmpiricalCovariance, MinCovDet

n_samples = 125
n_outliers = 25
n_features = 2

# generate data
gen_cov = np.eye(n_features)
gen_cov[0, 0] = 2.
X = np.dot(np.random.randn(n_samples, n_features), gen_cov)
# add some outliers
outliers_cov = np.eye(n_features)
outliers_cov[np.arange(1, n_features), np.arange(1, n_features)] = 7.
X[-n_outliers:] = np.dot(np.random.randn(n_outliers, n_features), outliers_cov)

# fit a Minimum Covariance Determinant (MCD) robust estimator to data
robust_cov = MinCovDet().fit(X)

# compare estimators learnt from the full data set with true parameters
emp_cov = EmpiricalCovariance().fit(X)
```

(continues on next page)

(continued from previous page)

```
# ##############################################################################
# Display results
fig = plt.figure()
plt.subplots_adjust(hspace=-.1, wspace=.4, top=.95, bottom=.05)

# Show data set
subfig1 = plt.subplot(3, 1, 1)
inlier_plot = subfig1.scatter(X[:, 0], X[:, 1],
                              color='black', label='inliers')
outlier_plot = subfig1.scatter(X[:, 0][-n_outliers:], X[:, 1][-n_outliers:],
                               color='red', label='outliers')
subfig1.set_xlim(subfig1.get_xlim()[0], 11.)
subfig1.set_title("Mahalanobis distances of a contaminated data set:")

# Show contours of the distance functions
xx, yy = np.meshgrid(np.linspace(plt.xlim()[0], plt.xlim()[1], 100),
                     np.linspace(plt.ylim()[0], plt.ylim()[1], 100))
zz = np.c_[xx.ravel(), yy.ravel()]

mahal_emp_cov = emp_cov.mahalanobis(zz)
mahal_emp_cov = mahal_emp_cov.reshape(xx.shape)
emp_cov_contour = subfig1.contour(xx, yy, np.sqrt(mahal_emp_cov),
                                  cmap=plt.cm.PuBu_r,
                                  linestyles='dashed')

mahal_robust_cov = robust_cov.mahalanobis(zz)
mahal_robust_cov = mahal_robust_cov.reshape(xx.shape)
robust_contour = subfig1.contour(xx, yy, np.sqrt(mahal_robust_cov),
                                 cmap=plt.cm.YlOrBr_r, linestyles='dotted')

subfig1.legend([emp_cov_contour.collections[1], robust_contour.collections[1],
                inlier_plot, outlier_plot],
               ['MLE dist', 'robust dist', 'inliers', 'outliers'],
               loc="upper right", borderaxespad=0)
plt.xticks(())
plt.yticks(())

# Plot the scores for each point
emp_mahal = emp_cov.mahalanobis(X - np.mean(X, 0)) ** (0.33)
subfig2 = plt.subplot(2, 2, 3)
subfig2.boxplot([emp_mahal[:-n_outliers], emp_mahal[-n_outliers:]], widths=.25)
subfig2.plot(np.full(n_samples - n_outliers, 1.26),
             emp_mahal[:-n_outliers], '+k', markeredgewidth=1)
subfig2.plot(np.full(n_outliers, 2.26),
             emp_mahal[-n_outliers:], '+k', markeredgewidth=1)
subfig2.axes.set_xticklabels(('inliers', 'outliers'), size=15)
subfig2.set_ylabel(r"$\sqrt[3]{\rm{(Mahal. dist.)}}$", size=16)
subfig2.set_title("1. from non-robust estimates\n(Maximum Likelihood)")
plt.yticks(())

robust_mahal = robust_cov.mahalanobis(X - robust_cov.location_) ** (0.33)
subfig3 = plt.subplot(2, 2, 4)
subfig3.boxplot([robust_mahal[:-n_outliers], robust_mahal[-n_outliers:]],
                widths=.25)
subfig3.plot(np.full(n_samples - n_outliers, 1.26),
             robust_mahal[:-n_outliers], '+k', markeredgewidth=1)
```

(continues on next page)

(continued from previous page)

```
subfig3.plot(np.full(n_outliers, 2.26),
             robust_mahal[-n_outliers:], '+k', markeredgewidth=1)
subfig3.axes.set_xticklabels(('inliers', 'outliers'), size=15)
subfig3.set_ylabel(r"$\sqrt[3]{\rm{(Mahal. dist.)}}$", size=16)
subfig3.set_title("2. from robust estimates\n(Minimum Covariance Determinant)")
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.404 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.6.5 Robust vs Empirical covariance estimate

The usual covariance maximum likelihood estimate is very sensitive to the presence of outliers in the data set. In such a case, it would be better to use a robust estimator of covariance to guarantee that the estimation is resistant to "erroneous" observations in the data set.[1],[2]

Minimum Covariance Determinant Estimator

The Minimum Covariance Determinant estimator is a robust, high-breakdown point (i.e. it can be used to estimate the covariance matrix of highly contaminated datasets, up to $\frac{n_{\text{samples}}-n_{\text{features}}-1}{2}$ outliers) estimator of covariance. The idea is to find $\frac{n_{\text{samples}}+n_{\text{features}}+1}{2}$ observations whose empirical covariance has the smallest determinant, yielding a "pure" subset of observations from which to compute standards estimates of location and covariance. After a correction step aiming at compensating the fact that the estimates were learned from only a portion of the initial data, we end up with robust estimates of the data set location and covariance.

The Minimum Covariance Determinant estimator (MCD) has been introduced by P.J.Rousseuw in[3].

Evaluation

In this example, we compare the estimation errors that are made when using various types of location and covariance estimates on contaminated Gaussian distributed data sets:

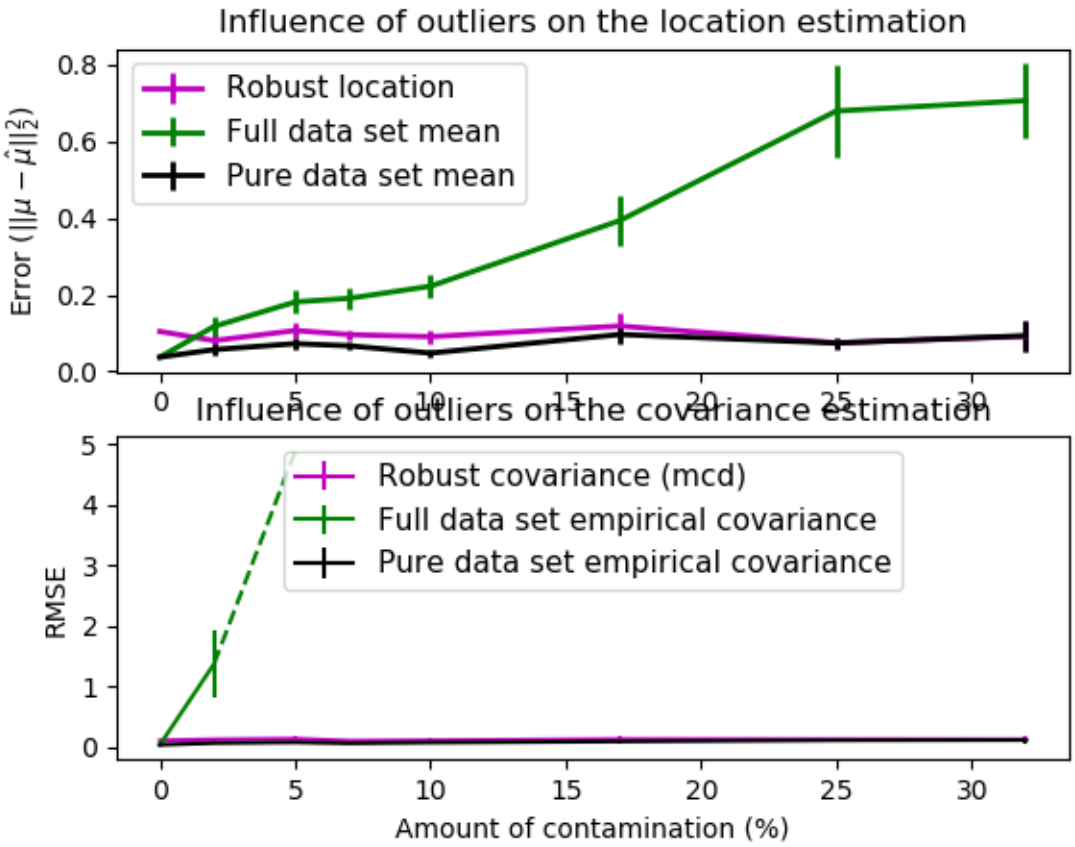
- The mean and the empirical covariance of the full dataset, which break down as soon as there are outliers in the data set
- The robust MCD, that has a low error provided $n_{\text{samples}} > 5n_{\text{features}}$
- The mean and the empirical covariance of the observations that are known to be good ones. This can be considered as a "perfect" MCD estimation, so one can trust our implementation by comparing to this case.

[1] Johanna Hardin, David M Rocke. The distribution of robust distances. Journal of Computational and Graphical Statistics. December 1, 2005, 14(4): 928-946.

[2] Zoubir A., Koivunen V., Chakhchoukh Y. and Muma M. (2012). Robust estimation in signal processing: A tutorial-style treatment of fundamental concepts. IEEE Signal Processing Magazine 29(4), 61-80.

[3] P. J. Rousseeuw. Least median of squares regression. Journal of American Statistical Ass., 79:871, 1984.

References




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.font_manager

from sklearn.covariance import EmpiricalCovariance, MinCovDet

# example settings
n_samples = 80
n_features = 5
repeat = 10

range_n_outliers = np.concatenate(
    (np.linspace(0, n_samples / 8, 5),
     np.linspace(n_samples / 8, n_samples / 2, 5)[1:-1])).astype(np.int)

# definition of arrays to store results
err_loc_mcd = np.zeros((range_n_outliers.size, repeat))
err_cov_mcd = np.zeros((range_n_outliers.size, repeat))
err_loc_emp_full = np.zeros((range_n_outliers.size, repeat))
err_cov_emp_full = np.zeros((range_n_outliers.size, repeat))
err_loc_emp_pure = np.zeros((range_n_outliers.size, repeat))
```

(continues on next page)

(continued from previous page)

```
err_cov_emp_pure = np.zeros((range_n_outliers.size, repeat))

# computation
for i, n_outliers in enumerate(range_n_outliers):
    for j in range(repeat):

        rng = np.random.RandomState(i * j)

        # generate data
        X = rng.randn(n_samples, n_features)
        # add some outliers
        outliers_index = rng.permutation(n_samples)[:n_outliers]
        outliers_offset = 10. * \
            (np.random.randint(2, size=(n_outliers, n_features)) - 0.5)
        X[outliers_index] += outliers_offset
        inliers_mask = np.ones(n_samples).astype(bool)
        inliers_mask[outliers_index] = False

        # fit a Minimum Covariance Determinant (MCD) robust estimator to data
        mcd = MinCovDet().fit(X)
        # compare raw robust estimates with the true location and covariance
        err_loc_mcd[i, j] = np.sum(mcd.location_ ** 2)
        err_cov_mcd[i, j] = mcd.error_norm(np.eye(n_features))

        # compare estimators learned from the full data set with true
        # parameters
        err_loc_emp_full[i, j] = np.sum(X.mean(0) ** 2)
        err_cov_emp_full[i, j] = EmpiricalCovariance().fit(X).error_norm(
            np.eye(n_features))

        # compare with an empirical covariance learned from a pure data set
        # (i.e. "perfect" mcd)
        pure_X = X[inliers_mask]
        pure_location = pure_X.mean(0)
        pure_emp_cov = EmpiricalCovariance().fit(pure_X)
        err_loc_emp_pure[i, j] = np.sum(pure_location ** 2)
        err_cov_emp_pure[i, j] = pure_emp_cov.error_norm(np.eye(n_features))

# Display results
font_prop = matplotlib.font_manager.FontProperties(size=11)
plt.subplot(2, 1, 1)
lw = 2
plt.errorbar(range_n_outliers, err_loc_mcd.mean(1),
             yerr=err_loc_mcd.std(1) / np.sqrt(repeat),
             label="Robust location", lw=lw, color='m')
plt.errorbar(range_n_outliers, err_loc_emp_full.mean(1),
             yerr=err_loc_emp_full.std(1) / np.sqrt(repeat),
             label="Full data set mean", lw=lw, color='green')
plt.errorbar(range_n_outliers, err_loc_emp_pure.mean(1),
             yerr=err_loc_emp_pure.std(1) / np.sqrt(repeat),
             label="Pure data set mean", lw=lw, color='black')
plt.title("Influence of outliers on the location estimation")
plt.ylabel(r"Error ($||\mu - \hat{\mu}||_2^2$)")
plt.legend(loc="upper left", prop=font_prop)

plt.subplot(2, 1, 2)
x_size = range_n_outliers.size
```

(continues on next page)

(continued from previous page)

```
plt.errorbar(range_n_outliers, err_cov_mcd.mean(1),
             yerr=err_cov_mcd.std(1),
             label="Robust covariance (mcd)", color='m')
plt.errorbar(range_n_outliers[:(x_size // 5 + 1)],
             err_cov_emp_full.mean(1)[:(x_size // 5 + 1)],
             yerr=err_cov_emp_full.std(1)[:(x_size // 5 + 1)],
             label="Full data set empirical covariance", color='green')
plt.plot(range_n_outliers[(x_size // 5):(x_size // 2 - 1)],
         err_cov_emp_full.mean(1)[(x_size // 5):(x_size // 2 - 1)],
         color='green', ls='--')
plt.errorbar(range_n_outliers, err_cov_emp_pure.mean(1),
             yerr=err_cov_emp_pure.std(1),
             label="Pure data set empirical covariance", color='black')
plt.title("Influence of outliers on the covariance estimation")
plt.xlabel("Amount of contamination (%)")
plt.ylabel("RMSE")
plt.legend(loc="upper center", prop=font_prop)

plt.show()
```

Total running time of the script: (0 minutes 2.544 seconds)

Estimated memory usage: 8 MB

6.7 Cross decomposition

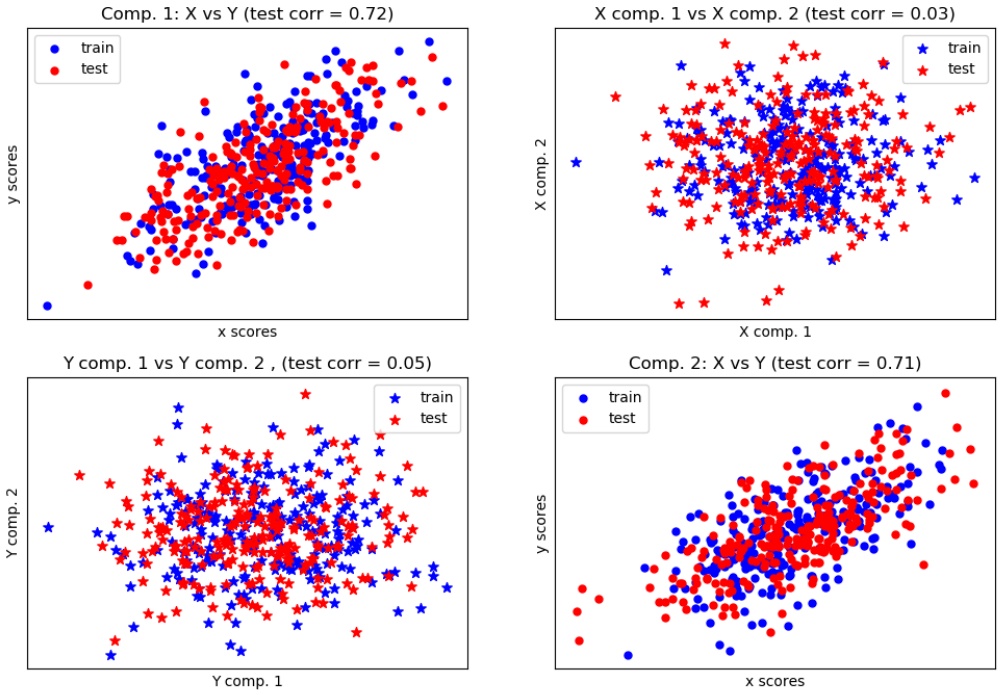
Examples concerning the *sklearn.cross_decomposition* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.7.1 Compare cross decomposition methods

Simple usage of various cross decomposition algorithms: - PLSCanonical - PLSRegression, with multivariate response, a.k.a. PLS2 - PLSRegression, with univariate response, a.k.a. PLS1 - CCA

Given 2 multivariate covarying two-dimensional datasets, X, and Y, PLS extracts the 'directions of covariance', i.e. the components of each datasets that explain the most shared variance between both datasets. This is apparent on the **scatterplot matrix** display: components 1 in dataset X and dataset Y are maximally correlated (points lie around the first diagonal). This is also true for components 2 in both dataset, however, the correlation across datasets for different components is weak: the point cloud is very spherical.



Out:

```
Corr(X)
[[ 1.    0.51  0.07 -0.05]
 [ 0.51  1.    0.11 -0.01]
 [ 0.07  0.11  1.    0.49]
 [-0.05 -0.01  0.49  1.  ]]
Corr(Y)
[[1.   0.48 0.05 0.03]
 [0.48 1.   0.04 0.12]
 [0.05 0.04 1.   0.51]
 [0.03 0.12 0.51 1.  ]]
True B (such that: Y = XB + Err)
[[1 1 1]
 [2 2 2]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]]
Estimated B
[[ 1.   1.   1. ]
 [ 2.   2.   2. ]
 [-0.  -0.   0. ]
 [ 0.   0.   0. ]
 [ 0.   0.   0. ]
```

(continues on next page)

(continued from previous page)

```
 [ 0.   0.  -0. ]
 [-0.  -0.  -0.1]
 [-0.  -0.   0. ]
 [ 0.   0.   0.1]
 [ 0.   0.  -0. ]]
Estimated betas
[[ 1. ]
 [ 2.1]
 [ 0. ]
 [ 0. ]
 [ 0. ]
 [-0. ]
 [-0. ]
 [ 0. ]
 [-0. ]
 [-0. ]]
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.cross_decomposition import PLSCanonical, PLSRegression, CCA

# ############################################################################
# Dataset based latent variables model

n = 500
# 2 latents vars:
l1 = np.random.normal(size=n)
l2 = np.random.normal(size=n)

latents = np.array([l1, l1, l2, l2]).T
X = latents + np.random.normal(size=4 * n).reshape((n, 4))
Y = latents + np.random.normal(size=4 * n).reshape((n, 4))

X_train = X[:n // 2]
Y_train = Y[:n // 2]
X_test = X[n // 2:]
Y_test = Y[n // 2:]

print("Corr(X)")
print(np.round(np.corrcoef(X.T), 2))
print("Corr(Y)")
print(np.round(np.corrcoef(Y.T), 2))

# ############################################################################
# Canonical (symmetric) PLS

# Transform data
# ~~~~~~~~~~~~~~
plsca = PLSCanonical(n_components=2)
```

(continues on next page)

(continued from previous page)

```
plsca.fit(X_train, Y_train)
X_train_r, Y_train_r = plsca.transform(X_train, Y_train)
X_test_r, Y_test_r = plsca.transform(X_test, Y_test)

# Scatter plot of scores
# ~~~~~~~~~~~~~~~~~~~~~~
# 1) On diagonal plot X vs Y scores on each components
plt.figure(figsize=(12, 8))
plt.subplot(221)
plt.scatter(X_train_r[:, 0], Y_train_r[:, 0], label="train",
            marker="o", c="b", s=25)
plt.scatter(X_test_r[:, 0], Y_test_r[:, 0], label="test",
            marker="o", c="r", s=25)
plt.xlabel("x scores")
plt.ylabel("y scores")
plt.title('Comp. 1: X vs Y (test corr = %.2f)' %
          np.corrcoef(X_test_r[:, 0], Y_test_r[:, 0])[0, 1])
plt.xticks(())
plt.yticks(())
plt.legend(loc="best")

plt.subplot(224)
plt.scatter(X_train_r[:, 1], Y_train_r[:, 1], label="train",
            marker="o", c="b", s=25)
plt.scatter(X_test_r[:, 1], Y_test_r[:, 1], label="test",
            marker="o", c="r", s=25)
plt.xlabel("x scores")
plt.ylabel("y scores")
plt.title('Comp. 2: X vs Y (test corr = %.2f)' %
          np.corrcoef(X_test_r[:, 1], Y_test_r[:, 1])[0, 1])
plt.xticks(())
plt.yticks(())
plt.legend(loc="best")

# 2) Off diagonal plot components 1 vs 2 for X and Y
plt.subplot(222)
plt.scatter(X_train_r[:, 0], X_train_r[:, 1], label="train",
            marker="*", c="b", s=50)
plt.scatter(X_test_r[:, 0], X_test_r[:, 1], label="test",
            marker="*", c="r", s=50)
plt.xlabel("X comp. 1")
plt.ylabel("X comp. 2")
plt.title('X comp. 1 vs X comp. 2 (test corr = %.2f)'
          % np.corrcoef(X_test_r[:, 0], X_test_r[:, 1])[0, 1])
plt.legend(loc="best")
plt.xticks(())
plt.yticks(())

plt.subplot(223)
plt.scatter(Y_train_r[:, 0], Y_train_r[:, 1], label="train",
            marker="*", c="b", s=50)
plt.scatter(Y_test_r[:, 0], Y_test_r[:, 1], label="test",
            marker="*", c="r", s=50)
plt.xlabel("Y comp. 1")
plt.ylabel("Y comp. 2")
plt.title('Y comp. 1 vs Y comp. 2 , (test corr = %.2f)'
          % np.corrcoef(Y_test_r[:, 0], Y_test_r[:, 1])[0, 1])
```

(continues on next page)

(continued from previous page)

```
plt.legend(loc="best")
plt.xticks(())
plt.yticks(())
plt.show()

# ############################################################################
# PLS regression, with multivariate response, a.k.a. PLS2

n = 1000
q = 3
p = 10
X = np.random.normal(size=n * p).reshape((n, p))
B = np.array([[1, 2] + [0] * (p - 2)] * q).T
# each Yj = 1*X1 + 2*X2 + noize
Y = np.dot(X, B) + np.random.normal(size=n * q).reshape((n, q)) + 5

pls2 = PLSRegression(n_components=3)
pls2.fit(X, Y)
print("True B (such that: Y = XB + Err)")
print(B)
# compare pls2.coef_ with B
print("Estimated B")
print(np.round(pls2.coef_, 1))
pls2.predict(X)

# PLS regression, with univariate response, a.k.a. PLS1

n = 1000
p = 10
X = np.random.normal(size=n * p).reshape((n, p))
y = X[:, 0] + 2 * X[:, 1] + np.random.normal(size=n * 1) + 5
pls1 = PLSRegression(n_components=3)
pls1.fit(X, y)
# note that the number of components exceeds 1 (the dimension of y)
print("Estimated betas")
print(np.round(pls1.coef_, 1))

# ############################################################################
# CCA (PLS mode B with symmetric deflation)

cca = CCA(n_components=2)
cca.fit(X_train, Y_train)
X_train_r, Y_train_r = cca.transform(X_train, Y_train)
X_test_r, Y_test_r = cca.transform(X_test, Y_test)
```

Total running time of the script: (0 minutes 0.329 seconds)

Estimated memory usage: 8 MB

6.8 Dataset examples

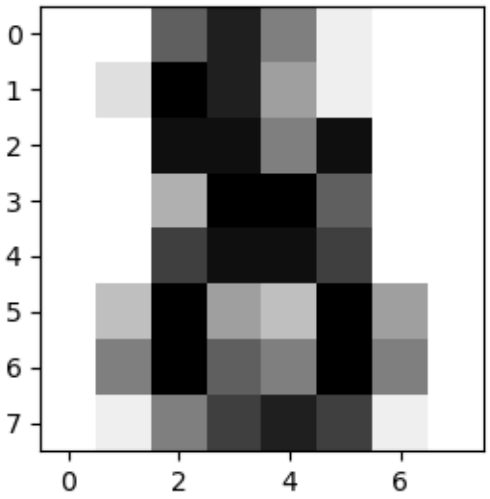
Examples concerning the *sklearn.datasets* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.8.1 The Digit Dataset

This dataset is made up of 1797 8x8 images. Each image, like the one shown below, is of a hand-written digit. In order to utilize an 8x8 figure like this, we'd have to first transform it into a feature vector with length 64.

See here for more information about this dataset.




```
print(__doc__)


# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

from sklearn import datasets

import matplotlib.pyplot as plt

#Load the digits dataset
digits = datasets.load_digits()

#Display the first digit
plt.figure(1, figsize=(3, 3))
plt.imshow(digits.images[-1], cmap=plt.cm.gray_r, interpolation='nearest')
plt.show()
```

Total running time of the script: (0 minutes 0.339 seconds)

Estimated memory usage: 8 MB

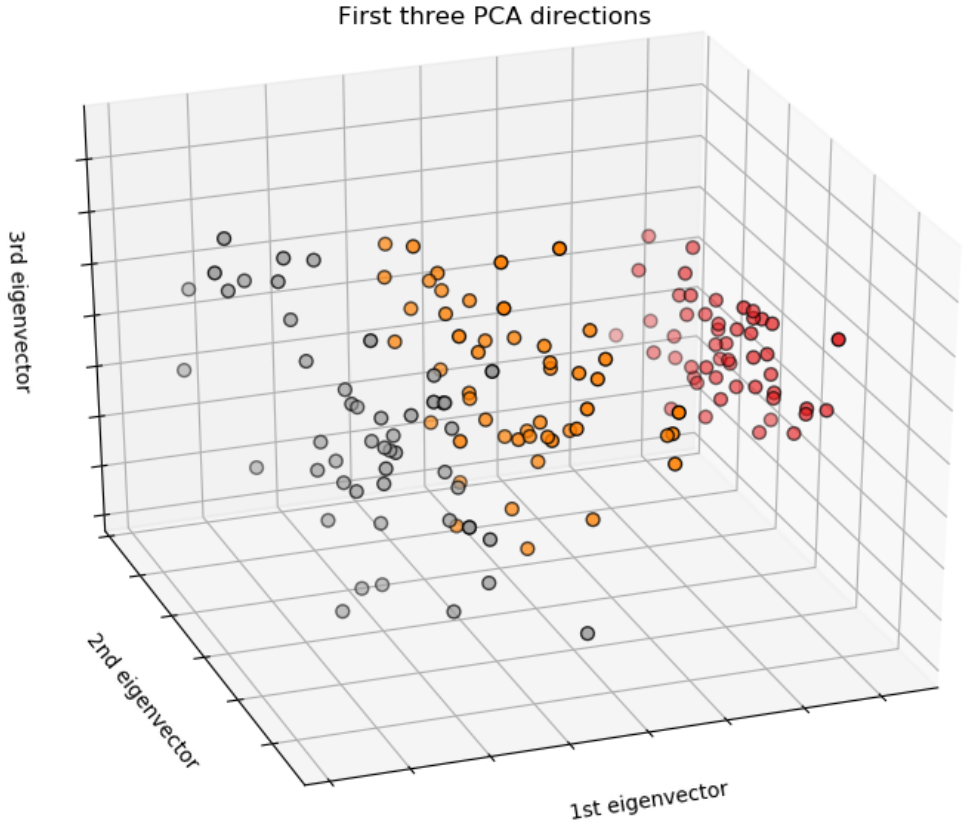
Note: Click *here* to download the full example code or to run this example in your browser via Binder

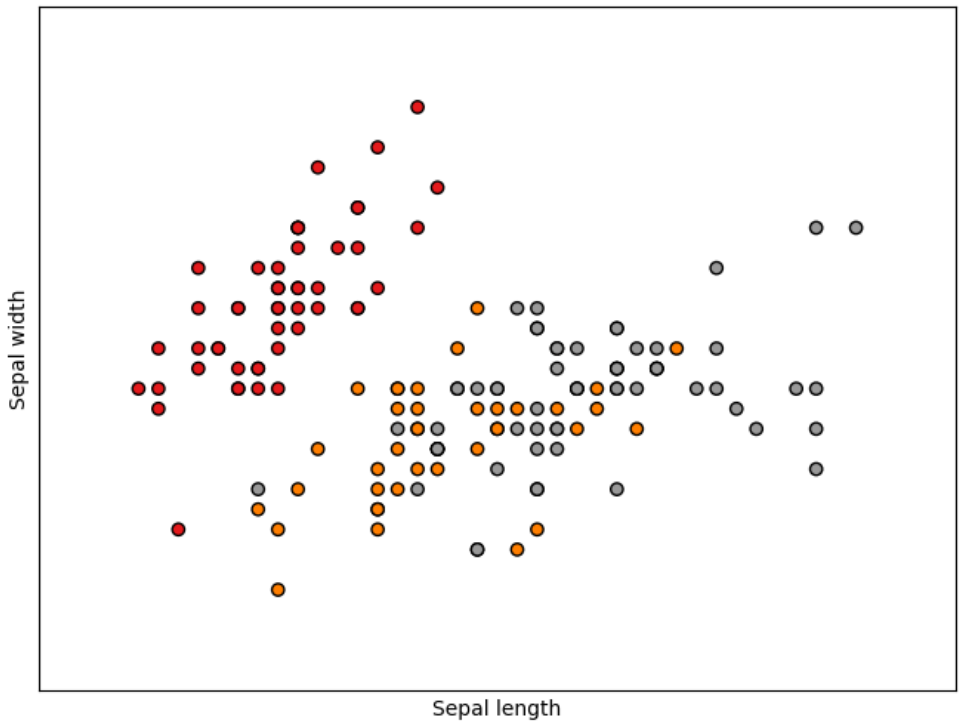
6.8.2 The Iris Dataset

This data sets consists of 3 different types of irises' (Setosa, Versicolour, and Virginica) petal and sepal length, stored in a 150x4 numpy.ndarray

The rows being the samples and the columns being: Sepal Length, Sepal Width, Petal Length and Petal Width.

The below plot uses the first two features. See here for more information on this dataset.





-

```
print(__doc__)


# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets
from sklearn.decomposition import PCA

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2]  # we only take the first two features.
y = iris.target

x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

plt.figure(2, figsize=(8, 6))
plt.clf()

# Plot the training points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1,
            edgecolor='k')
plt.xlabel('Sepal length')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Sepal width')

plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())

# To getter a better understanding of interaction of the dimensions
# plot the first three PCA dimensions
fig = plt.figure(1, figsize=(8, 6))
ax = Axes3D(fig, elev=-150, azim=110)
X_reduced = PCA(n_components=3).fit_transform(iris.data)
ax.scatter(X_reduced[:, 0], X_reduced[:, 1], X_reduced[:, 2], c=y,
           cmap=plt.cm.Set1, edgecolor='k', s=40)
ax.set_title("First three PCA directions")
ax.set_xlabel("1st eigenvector")
ax.w_xaxis.set_ticklabels([])
ax.set_ylabel("2nd eigenvector")
ax.w_yaxis.set_ticklabels([])
ax.set_zlabel("3rd eigenvector")
ax.w_zaxis.set_ticklabels([])

plt.show()
```

Total running time of the script: (0 minutes 0.431 seconds)

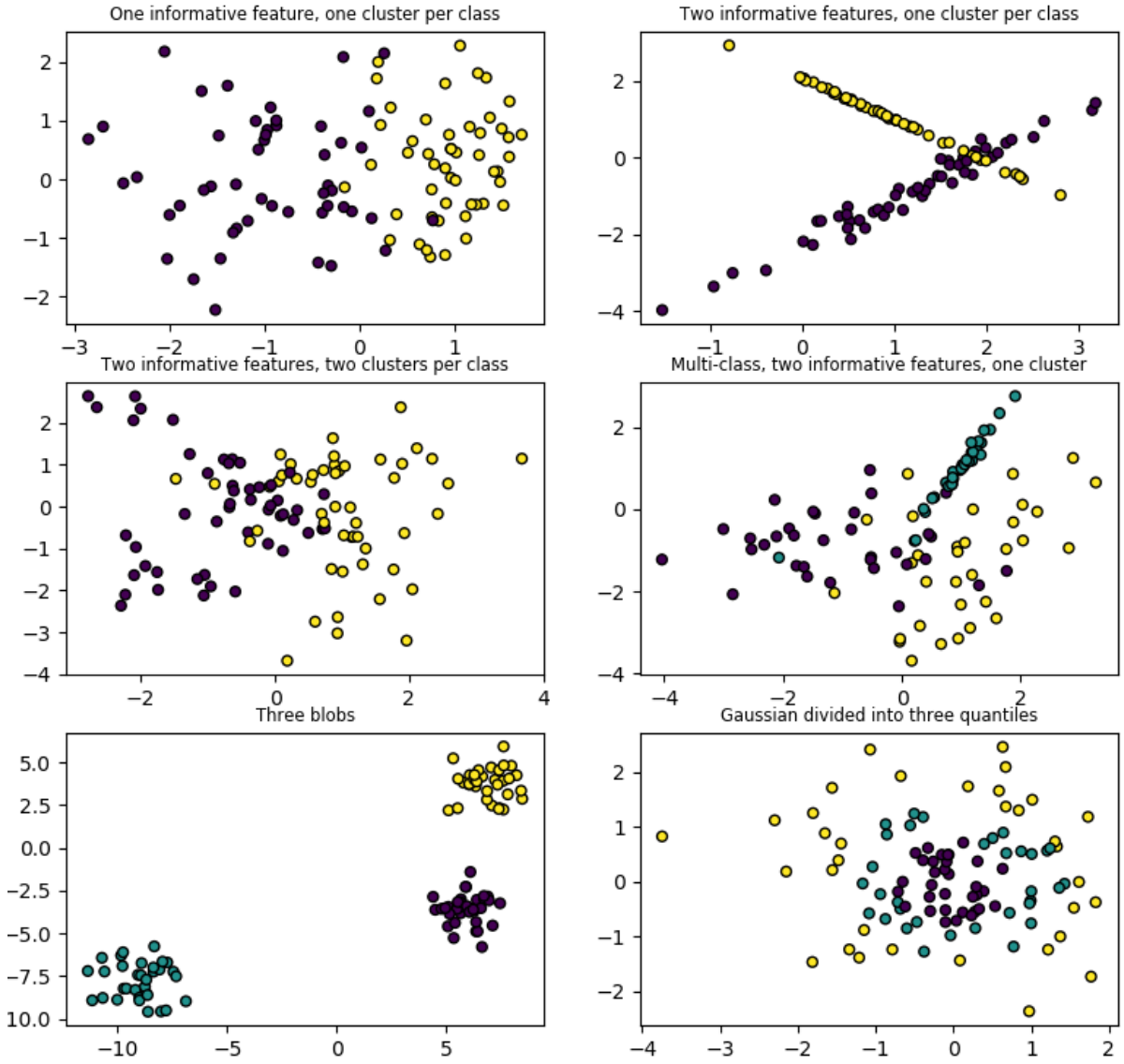
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.8.3 Plot randomly generated classification dataset

Plot several randomly generated 2D classification datasets. This example illustrates the `datasets.make_classification` `datasets.make_blobs` and `datasets.make_gaussian_quantiles` functions.

For `make_classification`, three binary and two multi-class classification datasets are generated, with different numbers of informative features and clusters per class.




```
print(__doc__)

import matplotlib.pyplot as plt

from sklearn.datasets import make_classification
from sklearn.datasets import make_blobs
from sklearn.datasets import make_gaussian_quantiles

plt.figure(figsize=(8, 8))
plt.subplots_adjust(bottom=.05, top=.9, left=.05, right=.95)

plt.subplot(321)
plt.title("One informative feature, one cluster per class", fontsize='small')
X1, Y1 = make_classification(n_features=2, n_redundant=0, n_informative=1,
```

(continues on next page)

(continued from previous page)

```
                               n_clusters_per_class=1)
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=25, edgecolor='k')

plt.subplot(322)
plt.title("Two informative features, one cluster per class", fontsize='small')
X1, Y1 = make_classification(n_features=2, n_redundant=0, n_informative=2,
                               n_clusters_per_class=1)
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=25, edgecolor='k')

plt.subplot(323)
plt.title("Two informative features, two clusters per class",
          fontsize='small')
X2, Y2 = make_classification(n_features=2, n_redundant=0, n_informative=2)
plt.scatter(X2[:, 0], X2[:, 1], marker='o', c=Y2,
            s=25, edgecolor='k')

plt.subplot(324)
plt.title("Multi-class, two informative features, one cluster",
          fontsize='small')
X1, Y1 = make_classification(n_features=2, n_redundant=0, n_informative=2,
                               n_clusters_per_class=1, n_classes=3)
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=25, edgecolor='k')

plt.subplot(325)
plt.title("Three blobs", fontsize='small')
X1, Y1 = make_blobs(n_features=2, centers=3)
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=25, edgecolor='k')

plt.subplot(326)
plt.title("Gaussian divided into three quantiles", fontsize='small')
X1, Y1 = make_gaussian_quantiles(n_features=2, n_classes=3)
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=25, edgecolor='k')

plt.show()
```

Total running time of the script: (0 minutes 0.540 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.8.4 Plot randomly generated multilabel dataset

This illustrates the *make_multilabel_classification* dataset generator. Each sample consists of counts of two features (up to 50 in total), which are differently distributed in each of two classes.

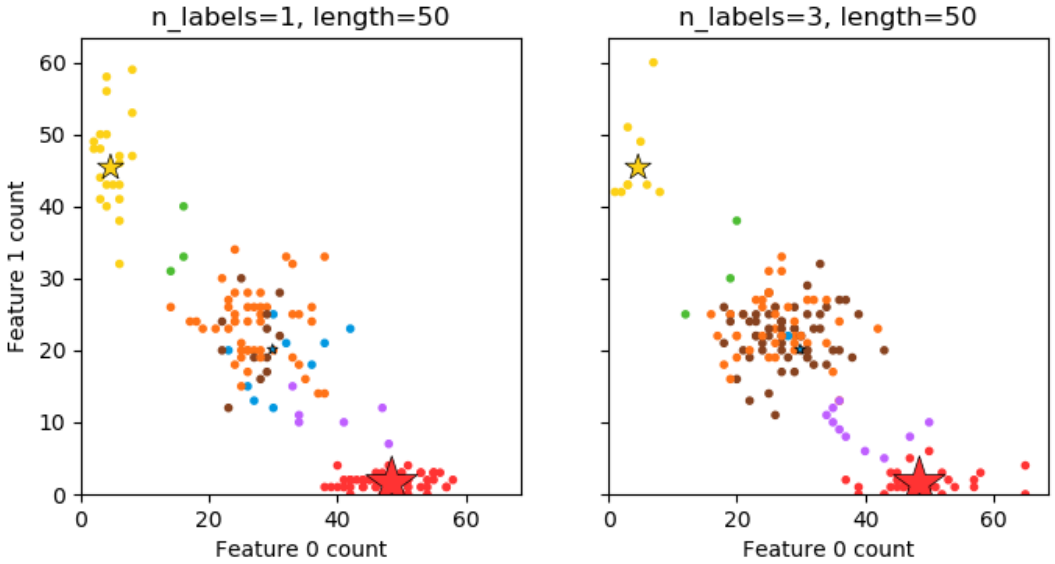
Points are labeled as follows, where Y means the class is present:

1	2	3	Color
Y	N	N	Red
N	Y	N	Blue
N	N	Y	Yellow
Y	Y	N	Purple
Y	N	Y	Orange
Y	Y	N	Green
Y	Y	Y	Brown

A star marks the expected sample for each class; its size reflects the probability of selecting that class label.

The left and right examples highlight the `n_labels` parameter: more of the samples in the right plot have 2 or 3 labels.

Note that this two-dimensional example is very degenerate: generally the number of features would be much greater than the "document length", while here we have much larger documents than vocabulary. Similarly, with `n_classes > n_features`, it is much less likely that a feature distinguishes a particular class.



Out:

```
The data was generated from (random_state=1013):
Class   P(C)    P(w0|C) P(w1|C)
red     0.64    0.97    0.03
blue    0.06    0.60    0.40
yellow  0.30    0.09    0.91
```

```
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
from sklearn.datasets import make_multilabel_classification as make_ml_clf

print(__doc__)

COLORS = np.array(['!',
                   '#FF3333',  # red
                   '#0198E1',  # blue
                   '#BF5FFF',  # purple
                   '#FCD116',  # yellow
                   '#FF7216',  # orange
                   '#4DBD33',  # green
                   '#87421F'   # brown
                   ])

# Use same random seed for multiple calls to make_multilabel_classification to
# ensure same distributions
RANDOM_SEED = np.random.randint(2 ** 10)


def plot_2d(ax, n_labels=1, n_classes=3, length=50):
    X, Y, p_c, p_w_c = make_ml_clf(n_samples=150, n_features=2,
                                   n_classes=n_classes, n_labels=n_labels,
                                   length=length, allow_unlabeled=False,
                                   return_distributions=True,
                                   random_state=RANDOM_SEED)

    ax.scatter(X[:, 0], X[:, 1], color=COLORS.take((Y * [1, 2, 4]
                                                    ).sum(axis=1)),
               marker='.')
    ax.scatter(p_w_c[0] * length, p_w_c[1] * length,
               marker='*', linewidth=.5, edgecolor='black',
               s=20 + 1500 * p_c ** 2,
               color=COLORS.take([1, 2, 4]))
    ax.set_xlabel('Feature 0 count')
    return p_c, p_w_c


_, (ax1, ax2) = plt.subplots(1, 2, sharex='row', sharey='row', figsize=(8, 4))
plt.subplots_adjust(bottom=.15)

p_c, p_w_c = plot_2d(ax1, n_labels=1)
ax1.set_title('n_labels=1, length=50')
ax1.set_ylabel('Feature 1 count')

plot_2d(ax2, n_labels=3)
ax2.set_title('n_labels=3, length=50')
ax2.set_xlim(left=0, auto=True)
ax2.set_ylim(bottom=0, auto=True)

plt.show()

print('The data was generated from (random_state=%d):' % RANDOM_SEED)
print('Class', 'P(C)', 'P(w0|C)', 'P(w1|C)', sep='\t')
for k, p, p_w in zip(['red', 'blue', 'yellow'], p_c, p_w_c.T):
    print('%s\t%0.2f\t%0.2f\t%0.2f' % (k, p, p_w[0], p_w[1]))
```

Total running time of the script: (0 minutes 0.257 seconds)

Estimated memory usage: 8 MB

6.9 Decision Trees

Examples concerning the `sklearn.tree` module.

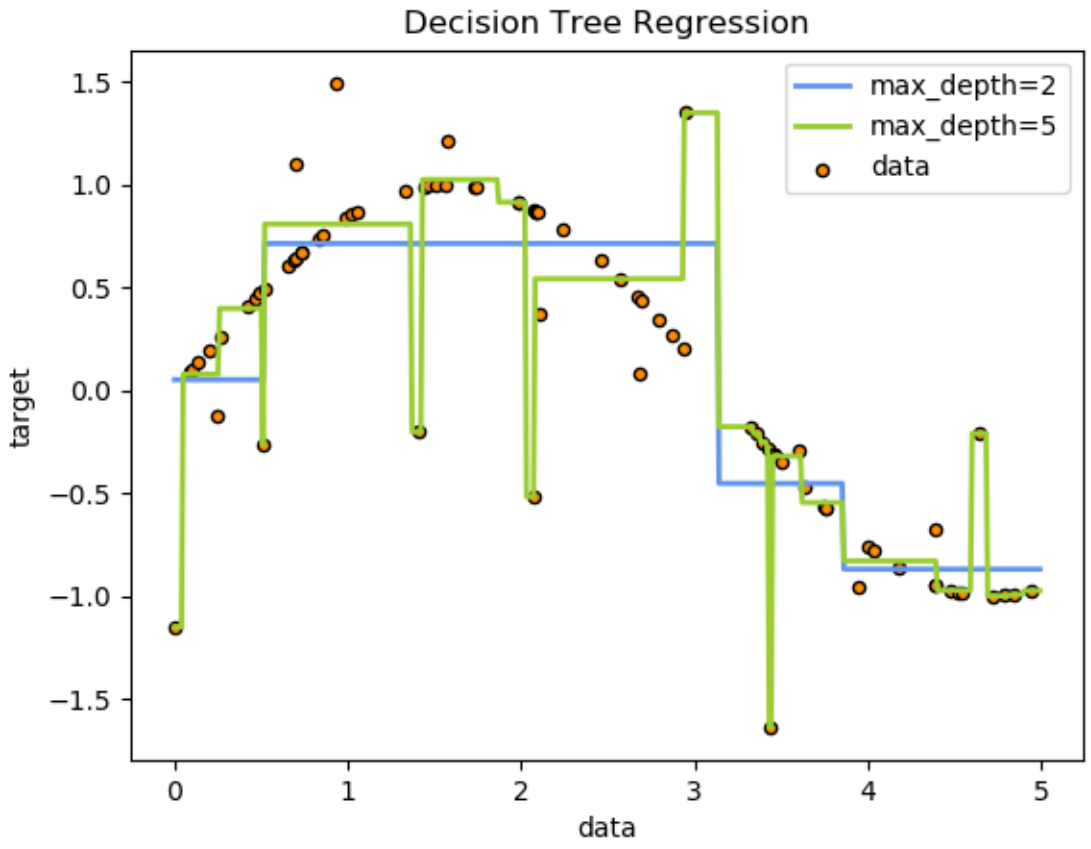
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.9.1 Decision Tree Regression

A 1D regression with decision tree.

The *decision trees* is used to fit a sine curve with addition noisy observation. As a result, it learns local linear regressions approximating the sine curve.

We can see that if the maximum depth of the tree (controlled by the `max_depth` parameter) is set too high, the decision trees learn too fine details of the training data and learn from the noise, i.e. they overfit.




```
print(__doc__)

# Import the necessary modules and libraries
```

(continues on next page)

(continued from previous page)

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor
import matplotlib.pyplot as plt

# Create a random dataset
rng = np.random.RandomState(1)
X = np.sort(5 * rng.rand(80, 1), axis=0)
y = np.sin(X).ravel()
y[::5] += 3 * (0.5 - rng.rand(16))

# Fit regression model
regr_1 = DecisionTreeRegressor(max_depth=2)
regr_2 = DecisionTreeRegressor(max_depth=5)
regr_1.fit(X, y)
regr_2.fit(X, y)

# Predict
X_test = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]
y_1 = regr_1.predict(X_test)
y_2 = regr_2.predict(X_test)

# Plot the results
plt.figure()
plt.scatter(X, y, s=20, edgecolor="black",
            c="darkorange", label="data")
plt.plot(X_test, y_1, color="cornflowerblue",
         label="max_depth=2", linewidth=2)
plt.plot(X_test, y_2, color="yellowgreen", label="max_depth=5", linewidth=2)
plt.xlabel("data")
plt.ylabel("target")
plt.title("Decision Tree Regression")
plt.legend()
plt.show()
```

Total running time of the script: (0 minutes 0.354 seconds)

Estimated memory usage: 8 MB

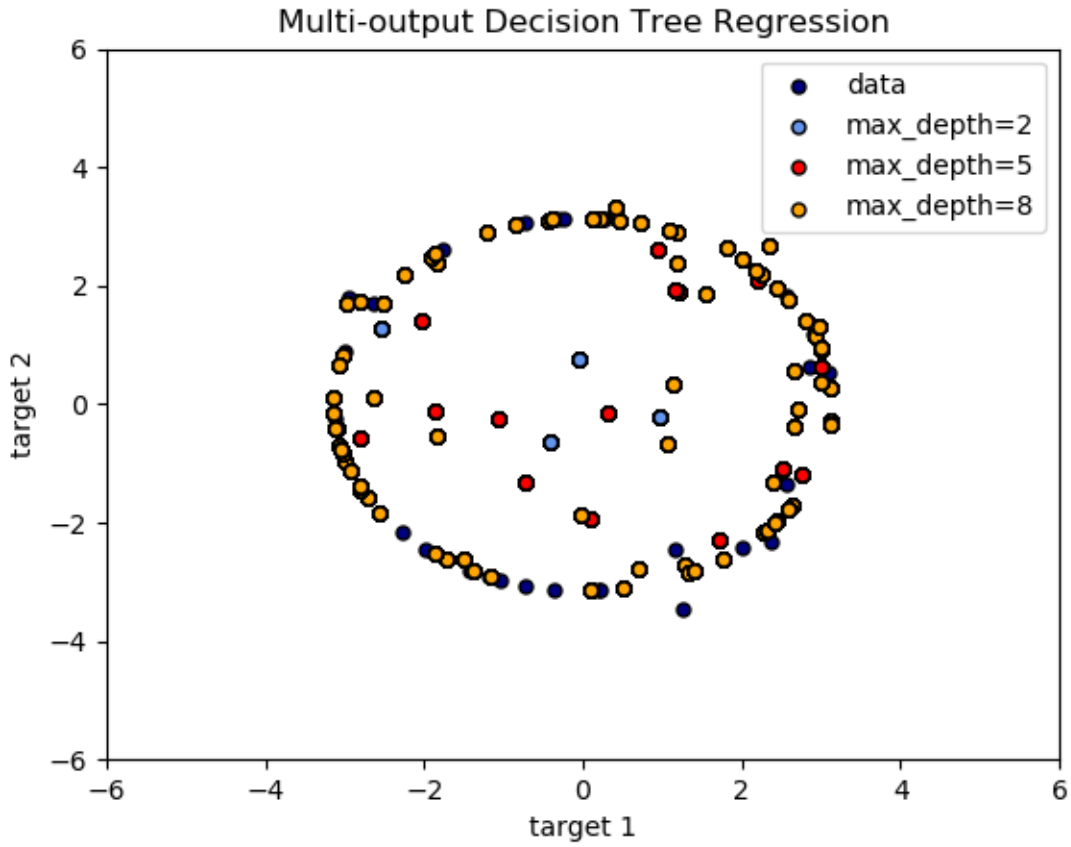
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.9.2 Multi-output Decision Tree Regression

An example to illustrate multi-output regression with decision tree.

The *decision trees* is used to predict simultaneously the noisy x and y observations of a circle given a single underlying feature. As a result, it learns local linear regressions approximating the circle.

We can see that if the maximum depth of the tree (controlled by the `max_depth` parameter) is set too high, the decision trees learn too fine details of the training data and learn from the noise, i.e. they overfit.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor

# Create a random dataset
rng = np.random.RandomState(1)
X = np.sort(200 * rng.rand(100, 1) - 100, axis=0)
y = np.array([np.pi * np.sin(X).ravel(), np.pi * np.cos(X).ravel()]).T
y[::5, :] += (0.5 - rng.rand(20, 2))

# Fit regression model
regr_1 = DecisionTreeRegressor(max_depth=2)
regr_2 = DecisionTreeRegressor(max_depth=5)
regr_3 = DecisionTreeRegressor(max_depth=8)
regr_1.fit(X, y)
regr_2.fit(X, y)
regr_3.fit(X, y)

# Predict
X_test = np.arange(-100.0, 100.0, 0.01)[:, np.newaxis]
y_1 = regr_1.predict(X_test)
y_2 = regr_2.predict(X_test)
y_3 = regr_3.predict(X_test)
```

(continues on next page)

(continued from previous page)

```
# Plot the results
plt.figure()
s = 25
plt.scatter(y[:, 0], y[:, 1], c="navy", s=s,
            edgecolor="black", label="data")
plt.scatter(y_1[:, 0], y_1[:, 1], c="cornflowerblue", s=s,
            edgecolor="black", label="max_depth=2")
plt.scatter(y_2[:, 0], y_2[:, 1], c="red", s=s,
            edgecolor="black", label="max_depth=5")
plt.scatter(y_3[:, 0], y_3[:, 1], c="orange", s=s,
            edgecolor="black", label="max_depth=8")
plt.xlim([-6, 6])
plt.ylim([-6, 6])
plt.xlabel("target 1")
plt.ylabel("target 2")
plt.title("Multi-output Decision Tree Regression")
plt.legend(loc="best")
plt.show()
```

Total running time of the script: (0 minutes 0.583 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

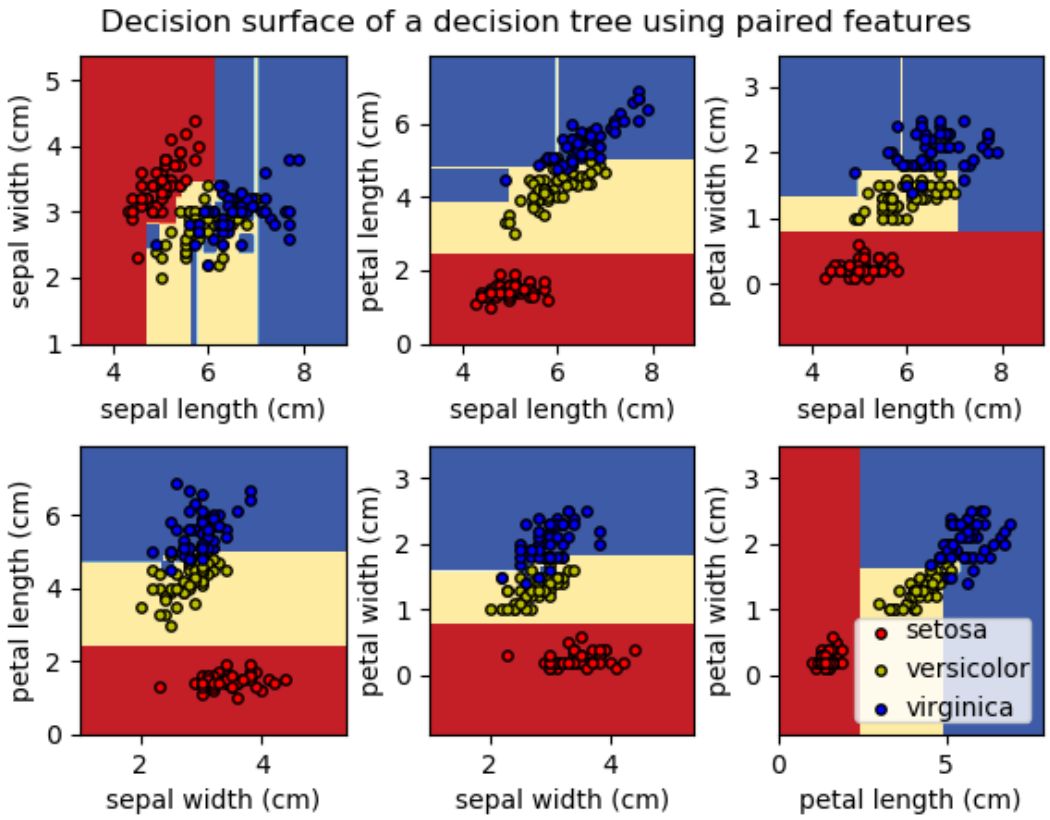
6.9.3 Plot the decision surface of a decision tree on the iris dataset

Plot the decision surface of a decision tree trained on pairs of features of the iris dataset.

See *decision tree* for more information on the estimator.

For each pair of iris features, the decision tree learns decision boundaries made of combinations of simple thresholding rules inferred from the training samples.

We also show the tree structure of a model built on all of the features.


Decision surface of a decision tree using paired features
sepal width (cm)
sepal length (cm)
petal length (cm)
sepal length (cm)
petal width (cm)
sepal length (cm)
petal length (cm)
sepal width (cm)
petal width (cm)
sepal width (cm)
petal width (cm)
petal length (cm)
setosa
versicolor
virginica

-

-

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree

# Parameters
n_classes = 3
plot_colors = "ryb"
plot_step = 0.02

# Load data
iris = load_iris()

for pairidx, pair in enumerate([[0, 1], [0, 2], [0, 3],
                                [1, 2], [1, 3], [2, 3]]):
    # We only take the two corresponding features
    X = iris.data[:, pair]
    y = iris.target

    # Train
    clf = DecisionTreeClassifier().fit(X, y)

    # Plot the decision boundary
    plt.subplot(2, 3, pairidx + 1)
```

(continues on next page)

(continued from previous page)

```
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                         np.arange(y_min, y_max, plot_step))
    plt.tight_layout(h_pad=0.5, w_pad=0.5, pad=2.5)

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    cs = plt.contourf(xx, yy, Z, cmap=plt.cm.RdYlBu)

    plt.xlabel(iris.feature_names[pair[0]])
    plt.ylabel(iris.feature_names[pair[1]])

    # Plot the training points
    for i, color in zip(range(n_classes), plot_colors):
        idx = np.where(y == i)
        plt.scatter(X[idx, 0], X[idx, 1], c=color, label=iris.target_names[i],
                    cmap=plt.cm.RdYlBu, edgecolor='black', s=15)

plt.suptitle("Decision surface of a decision tree using paired features")
plt.legend(loc='lower right', borderpad=0, handletextpad=0)
plt.axis("tight")

plt.figure()
clf = DecisionTreeClassifier().fit(iris.data, iris.target)
plot_tree(clf, filled=True)
plt.show()
```

Total running time of the script: (0 minutes 0.909 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.9.4 Post pruning decision trees with cost complexity pruning

The *DecisionTreeClassifier* provides parameters such as `min_samples_leaf` and `max_depth` to prevent a tree from overfiting. Cost complexity pruning provides another option to control the size of a tree. In *DecisionTreeClassifier*, this pruning technique is parameterized by the cost complexity parameter, `ccp_alpha`. Greater values of `ccp_alpha` increase the number of nodes pruned. Here we only show the effect of `ccp_alpha` on regularizing the trees and how to choose a `ccp_alpha` based on validation scores.

See also *Minimal Cost-Complexity Pruning* for details on pruning.

```
print(__doc__)
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.tree import DecisionTreeClassifier
```

Total impurity of leaves vs effective alphas of pruned tree

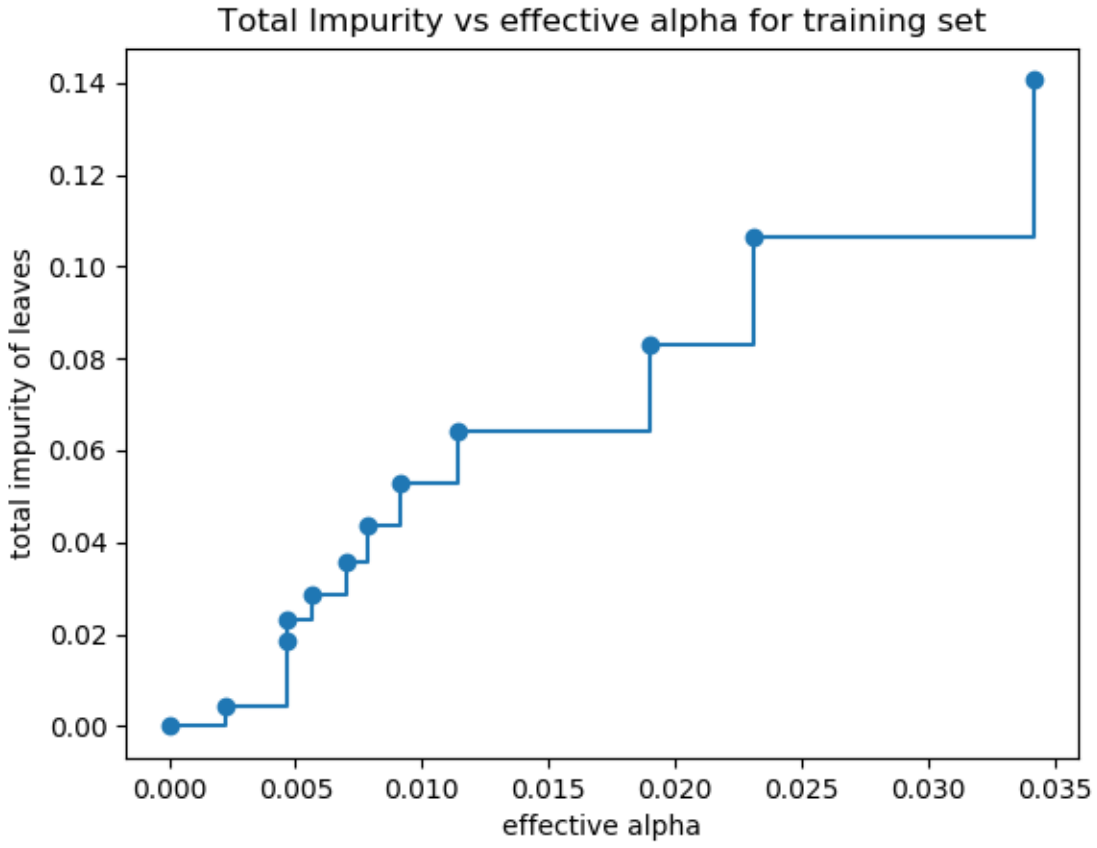
Minimal cost complexity pruning recursively finds the node with the "weakest link". The weakest link is characterized by an effective alpha, where the nodes with the smallest effective alpha are pruned first. To get an idea of what values of `ccp_alpha` could be appropriate, scikit-learn provides *DecisionTreeClassifier.cost_complexity_pruning_path* that returns the effective alphas and the corresponding total leaf impurities at each step of the pruning process. As alpha increases, more of the tree is pruned, which increases the total impurity of its leaves.

```
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

clf = DecisionTreeClassifier(random_state=0)
path = clf.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities
```

In the following plot, the maximum effective alpha value is removed, because it is the trivial tree with only one node.

```
fig, ax = plt.subplots()
ax.plot(ccp_alphas[:-1], impurities[:-1], marker='o', drawstyle="steps-post")
ax.set_xlabel("effective alpha")
ax.set_ylabel("total impurity of leaves")
ax.set_title("Total Impurity vs effective alpha for training set")
```



Out:

```
Text(0.5, 1.0, 'Total Impurity vs effective alpha for training set')
```

Next, we train a decision tree using the effective alphas. The last value in `ccp_alphas` is the alpha value that prunes the whole tree, leaving the tree, `clfs[-1]`, with one node.

```
clfs = []
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    clf.fit(X_train, y_train)
    clfs.append(clf)
print("Number of nodes in the last tree is: {} with ccp_alpha: {}".format(
      clfs[-1].tree_.node_count, ccp_alphas[-1]))
```

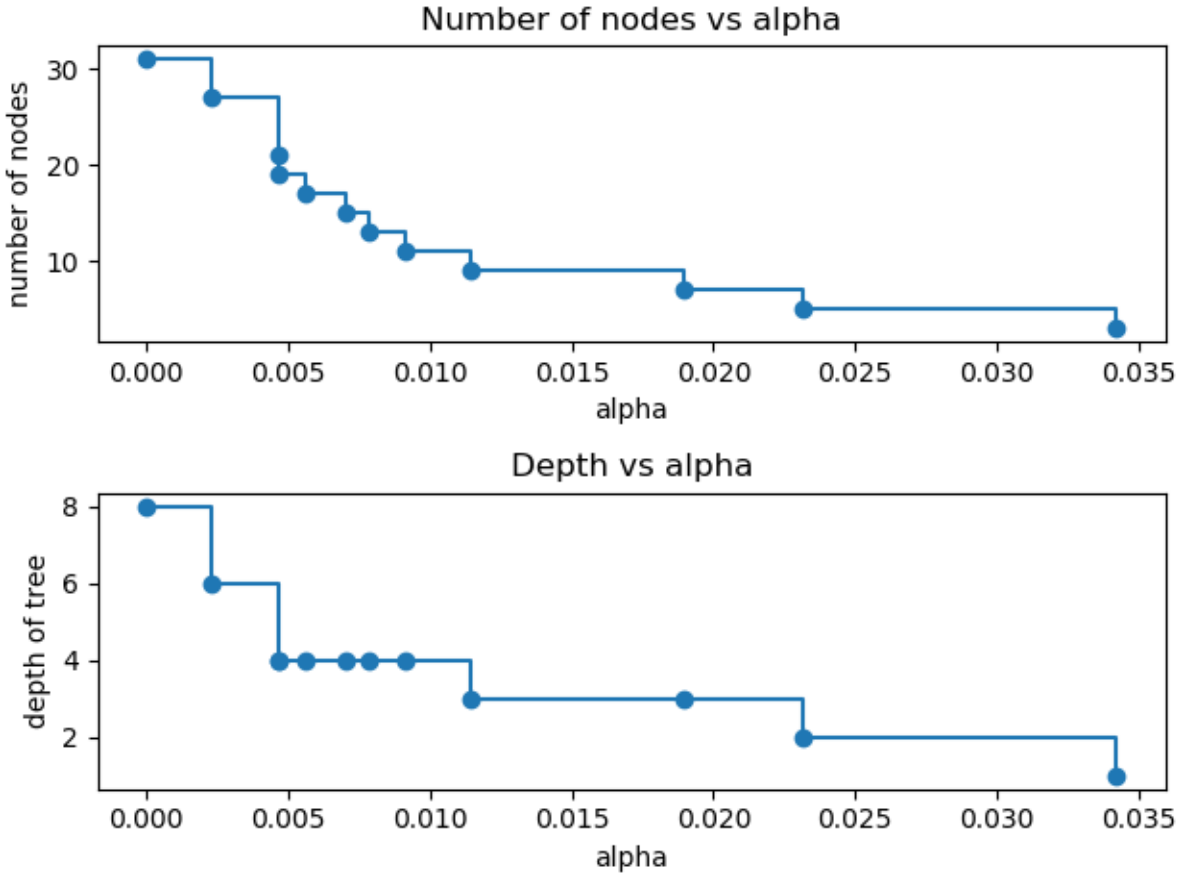
Out:

```
Number of nodes in the last tree is: 1 with ccp_alpha: 0.3272984419327777
```

For the remainder of this example, we remove the last element in `clfs` and `ccp_alphas`, because it is the trivial tree with only one node. Here we show that the number of nodes and tree depth decreases as alpha increases.

```
clfs = clfs[:-1]
ccp_alphas = ccp_alphas[:-1]

node_counts = [clf.tree_.node_count for clf in clfs]
depth = [clf.tree_.max_depth for clf in clfs]
fig, ax = plt.subplots(2, 1)
ax[0].plot(ccp_alphas, node_counts, marker='o', drawstyle="steps-post")
ax[0].set_xlabel("alpha")
ax[0].set_ylabel("number of nodes")
ax[0].set_title("Number of nodes vs alpha")
ax[1].plot(ccp_alphas, depth, marker='o', drawstyle="steps-post")
ax[1].set_xlabel("alpha")
ax[1].set_ylabel("depth of tree")
ax[1].set_title("Depth vs alpha")
fig.tight_layout()
```

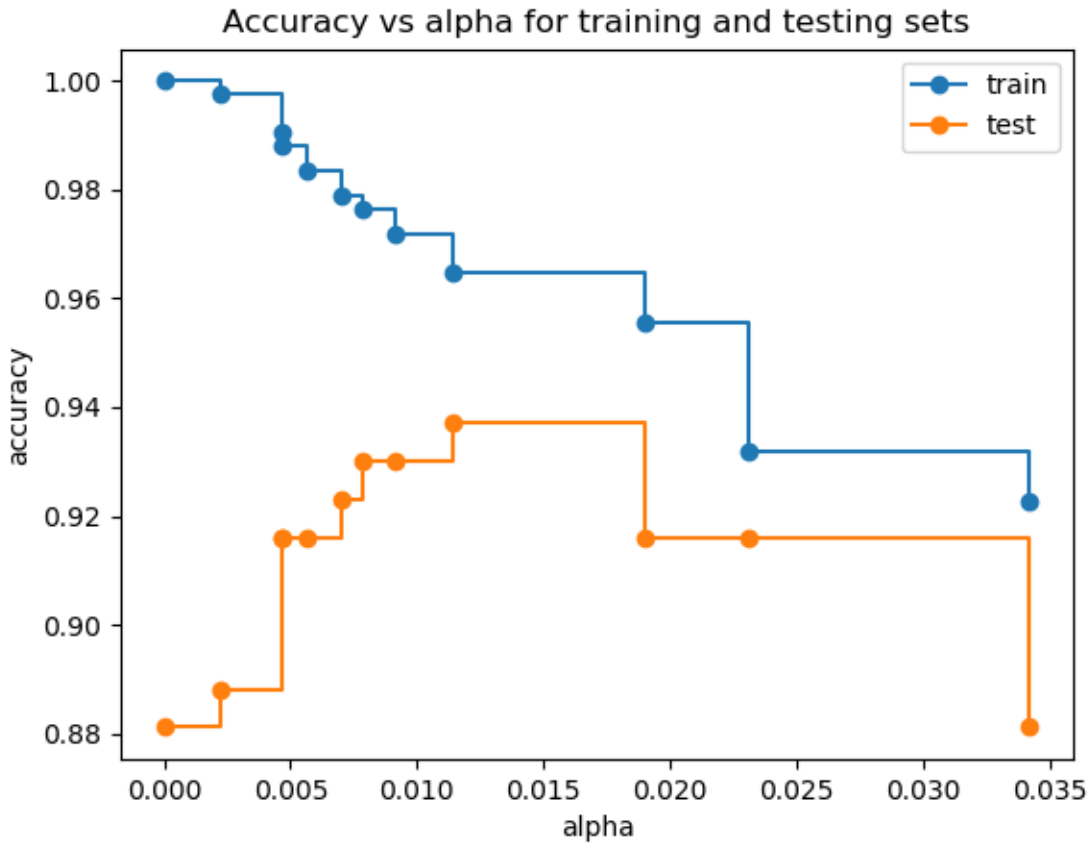


Accuracy vs alpha for training and testing sets

When `ccp_alpha` is set to zero and keeping the other default parameters of *DecisionTreeClassifier*, the tree overfits, leading to a 100% training accuracy and 88% testing accuracy. As alpha increases, more of the tree is pruned, thus creating a decision tree that generalizes better. In this example, setting `ccp_alpha=0.015` maximizes the testing accuracy.

```
train_scores = [clf.score(X_train, y_train) for clf in clfs]
test_scores = [clf.score(X_test, y_test) for clf in clfs]

fig, ax = plt.subplots()
ax.set_xlabel("alpha")
ax.set_ylabel("accuracy")
ax.set_title("Accuracy vs alpha for training and testing sets")
ax.plot(ccp_alphas, train_scores, marker='o', label="train",
        drawstyle="steps-post")
ax.plot(ccp_alphas, test_scores, marker='o', label="test",
        drawstyle="steps-post")
ax.legend()
plt.show()
```



Total running time of the script: (0 minutes 1.627 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.9.5 Understanding the decision tree structure

The decision tree structure can be analysed to gain further insight on the relation between the features and the target to predict. In this example, we show how to retrieve:

- the binary tree structure;
- the depth of each node and whether or not it's a leaf;
- the nodes that were reached by a sample using the `decision_path` method;
- the leaf that was reached by a sample using the apply method;
- the rules that were used to predict a sample;
- the decision path shared by a group of samples.

Out:

```
The binary tree structure has 5 nodes and has the following tree structure:
node=0 test node: go to node 1 if X[:, 3] <= 0.800000011920929 else to node 2.
        node=1 leaf node.
        node=2 test node: go to node 3 if X[:, 2] <= 4.950000047683716 else to node 4.
                node=3 leaf node.
                node=4 leaf node.

Rules used to predict sample 0:
decision id node 0 : (X_test[0, 3] (= 2.4) > 0.800000011920929)
decision id node 2 : (X_test[0, 2] (= 5.1) > 4.950000047683716)

The following samples [0, 1] share the node [0 2] in the tree
It is 40.0 % of all nodes.
```

```
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

estimator = DecisionTreeClassifier(max_leaf_nodes=3, random_state=0)
estimator.fit(X_train, y_train)

# The decision estimator has an attribute called tree_  which stores the entire
# tree structure and allows access to low level attributes. The binary tree
# tree_ is represented as a number of parallel arrays. The i-th element of each
# array holds information about the node `i`. Node 0 is the tree's root. NOTE:
# Some of the arrays only apply to either leaves or split nodes, resp. In this
# case the values of nodes of the other type are arbitrary!
#
# Among those arrays, we have:
#   - left_child, id of the left child of the node
#   - right_child, id of the right child of the node
#   - feature, feature used for splitting the node
#   - threshold, threshold value at the node
#

# Using those arrays, we can parse the tree structure:

n_nodes = estimator.tree_.node_count
children_left = estimator.tree_.children_left
children_right = estimator.tree_.children_right
feature = estimator.tree_.feature
threshold = estimator.tree_.threshold


# The tree structure can be traversed to compute various properties such
```

(continues on next page)

(continued from previous page)

```
# as the depth of each node and whether or not it is a leaf.
node_depth = np.zeros(shape=n_nodes, dtype=np.int64)
is_leaves = np.zeros(shape=n_nodes, dtype=bool)
stack = [(0, -1)]  # seed is the root node id and its parent depth
while len(stack) > 0:
    node_id, parent_depth = stack.pop()
    node_depth[node_id] = parent_depth + 1

    # If we have a test node
    if (children_left[node_id] != children_right[node_id]):
        stack.append((children_left[node_id], parent_depth + 1))
        stack.append((children_right[node_id], parent_depth + 1))
    else:
        is_leaves[node_id] = True

print("The binary tree structure has %s nodes and has "
      "the following tree structure:"
      % n_nodes)
for i in range(n_nodes):
    if is_leaves[i]:
        print("%snode=%s leaf node." % (node_depth[i] * "\t", i))
    else:
        print("%snode=%s test node: go to node %s if X[:, %s] <= %s else to "
              "node %s."
              % (node_depth[i] * "\t",
                 i,
                 children_left[i],
                 feature[i],
                 threshold[i],
                 children_right[i],
                 ))
print()

# First let's retrieve the decision path of each sample. The decision_path
# method allows to retrieve the node indicator functions. A non zero element of
# indicator matrix at the position (i, j) indicates that the sample i goes
# through the node j.

node_indicator = estimator.decision_path(X_test)

# Similarly, we can also have the leaves ids reached by each sample.

leave_id = estimator.apply(X_test)

# Now, it's possible to get the tests that were used to predict a sample or
# a group of samples. First, let's make it for the sample.

sample_id = 0
node_index = node_indicator.indices[node_indicator.indptr[sample_id]:
                                    node_indicator.indptr[sample_id + 1]]

print('Rules used to predict sample %s: ' % sample_id)
for node_id in node_index:
    if leave_id[sample_id] == node_id:
        continue

    if (X_test[sample_id, feature[node_id]] <= threshold[node_id]):
```

(continues on next page)

(continued from previous page)

```
        threshold_sign = "<="
    else:
        threshold_sign = ">"

    print("decision id node %s : (X_test[%s, %s] (= %s) %s %s)"
          % (node_id,
             sample_id,
             feature[node_id],
             X_test[sample_id, feature[node_id]],
             threshold_sign,
             threshold[node_id]))

# For a group of samples, we have the following common node.
sample_ids = [0, 1]
common_nodes = (node_indicator.toarray()[sample_ids].sum(axis=0) ==
                len(sample_ids))

common_node_id = np.arange(n_nodes)[common_nodes]

print("\nThe following samples %s share the node %s in the tree"
      % (sample_ids, common_node_id))
print("It is %s %% of all nodes." % (100 * len(common_node_id) / n_nodes,))
```

Total running time of the script: (0 minutes 0.259 seconds)

Estimated memory usage: 8 MB

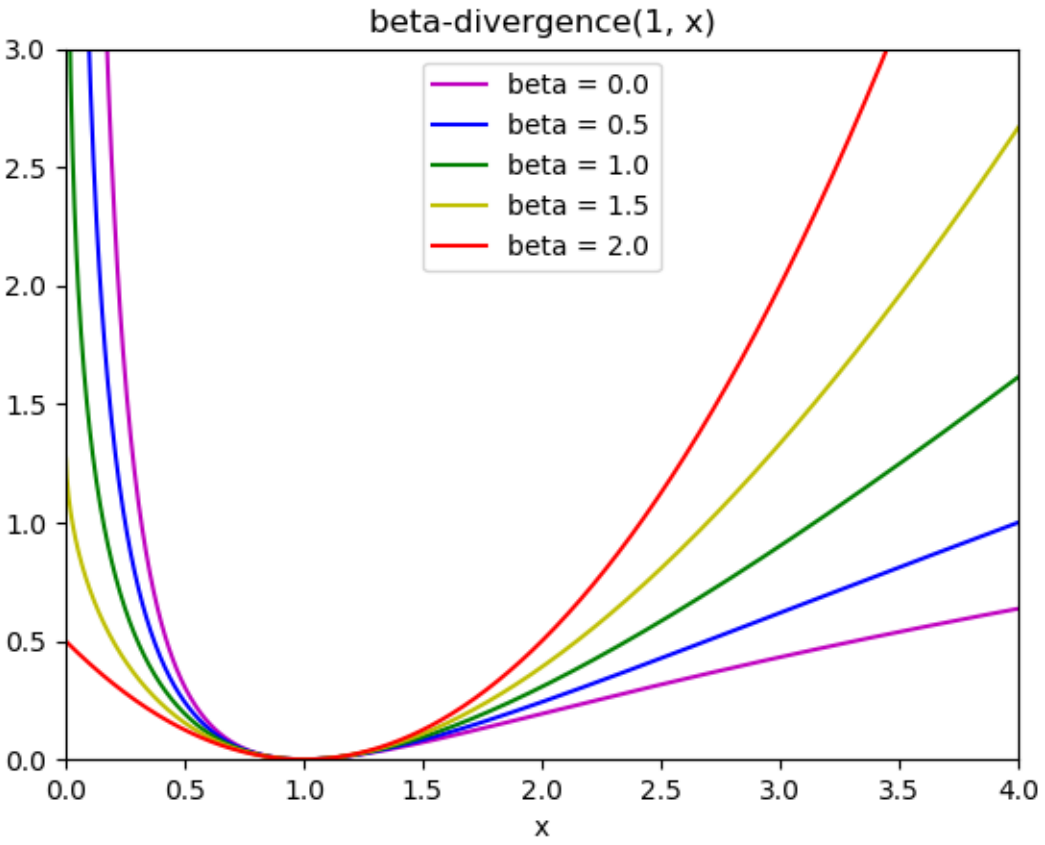
6.10 Decomposition

Examples concerning the *sklearn.decomposition* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.10.1 Beta-divergence loss functions

A plot that compares the various Beta-divergence loss functions supported by the Multiplicative-Update ('mu') solver in *sklearn.decomposition.NMF*.




```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition._nmf import _beta_divergence

print(__doc__)

x = np.linspace(0.001, 4, 1000)
y = np.zeros(x.shape)

colors = 'mbgyr'
for j, beta in enumerate((0., 0.5, 1., 1.5, 2.)):
    for i, xi in enumerate(x):
        y[i] = _beta_divergence(1, xi, 1, beta)
    name = "beta = %1.1f" % beta
    plt.plot(x, y, label=name, color=colors[j])

plt.xlabel("x")
plt.title("beta-divergence(1, x)")
plt.legend(loc=0)
plt.axis([0, 4, 0, 3])
plt.show()
```

Total running time of the script: (0 minutes 0.422 seconds)

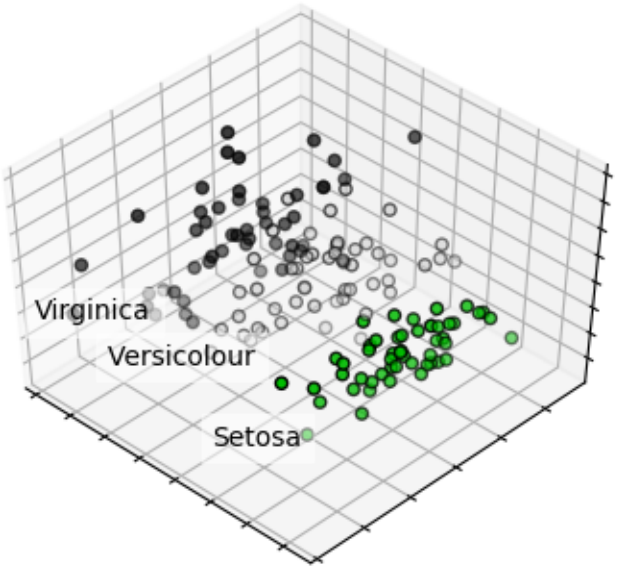
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.10.2 PCA example with Iris Data-set

Principal Component Analysis applied to the Iris dataset.

See here for more information on this dataset.




```
print(__doc__)


# Code source: Gaël Varoquaux
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


from sklearn import decomposition
from sklearn import datasets

np.random.seed(5)

centers = [[1, 1], [-1, -1], [1, -1]]
iris = datasets.load_iris()
X = iris.data
y = iris.target

fig = plt.figure(1, figsize=(4, 3))
plt.clf()
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

plt.cla()
pca = decomposition.PCA(n_components=3)
```

(continues on next page)

(continued from previous page)

```
pca.fit(X)
X = pca.transform(X)

for name, label in [('Setosa', 0), ('Versicolour', 1), ('Virginica', 2)]:
    ax.text3D(X[y == label, 0].mean(),
              X[y == label, 1].mean() + 1.5,
              X[y == label, 2].mean(), name,
              horizontalalignment='center',
              bbox=dict(alpha=.5, edgecolor='w', facecolor='w'))
# Reorder the labels to have colors matching the cluster results
y = np.choose(y, [1, 2, 0]).astype(np.float)
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=y, cmap=plt.cm.nipy_spectral,
           edgecolor='k')

ax.w_xaxis.set_ticklabels([])
ax.w_yaxis.set_ticklabels([])
ax.w_zaxis.set_ticklabels([])

plt.show()
```

Total running time of the script: (0 minutes 0.338 seconds)

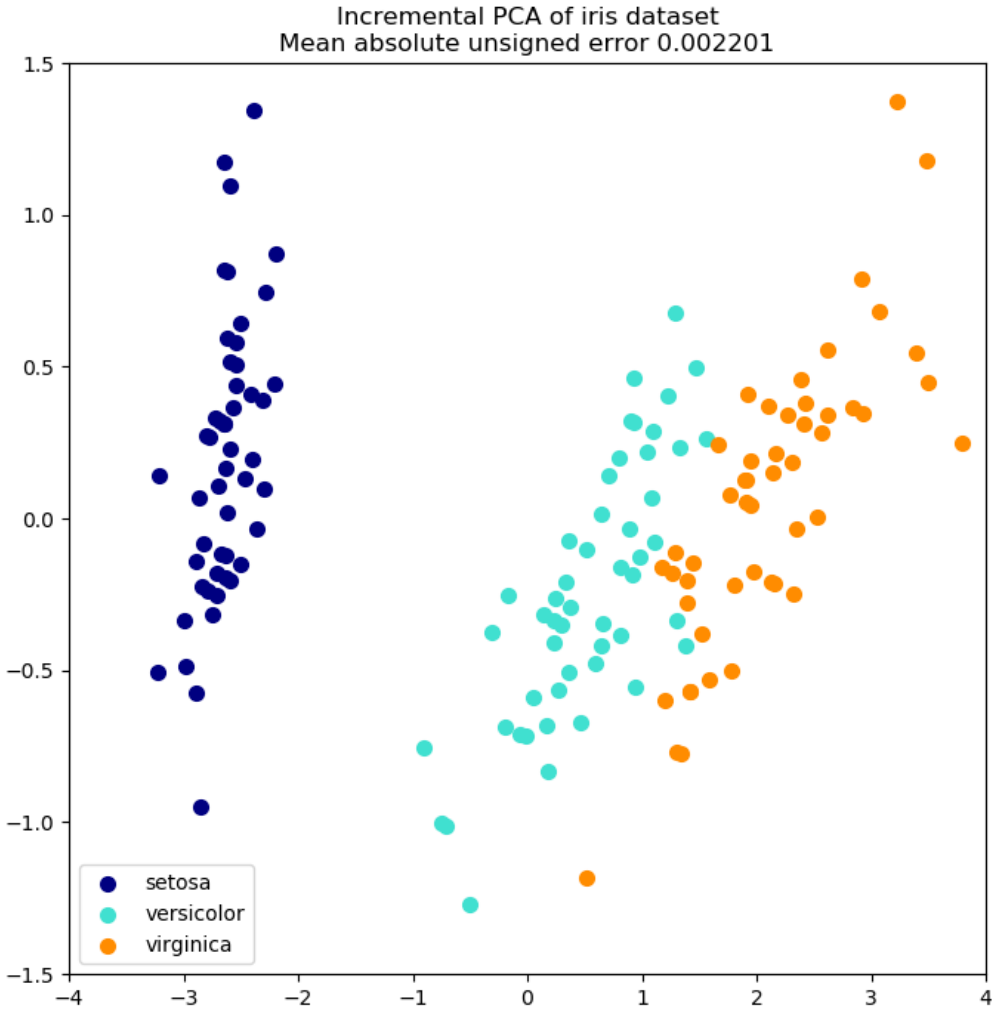
Estimated memory usage: 8 MB

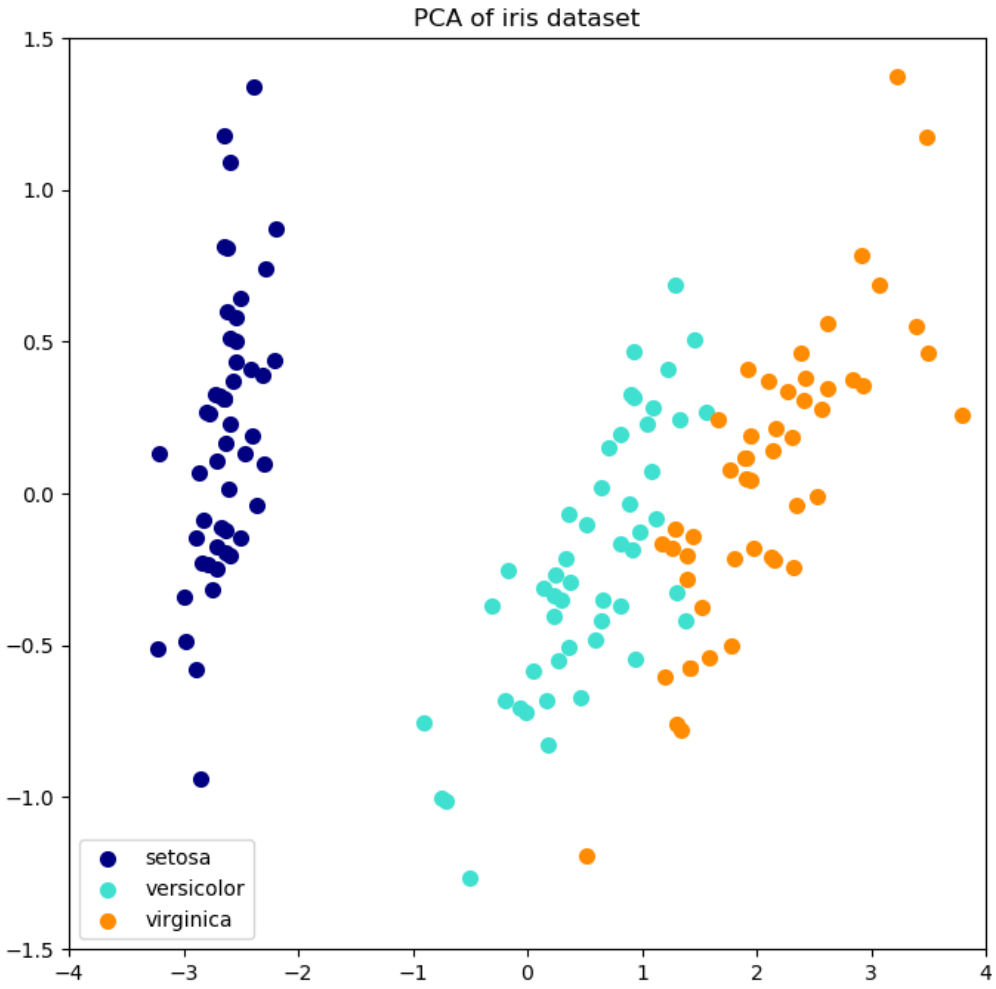
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.10.3 Incremental PCA

Incremental principal component analysis (IPCA) is typically used as a replacement for principal component analysis (PCA) when the dataset to be decomposed is too large to fit in memory. IPCA builds a low-rank approximation for the input data using an amount of memory which is independent of the number of input data samples. It is still dependent on the input data features, but changing the batch size allows for control of memory usage.

This example serves as a visual check that IPCA is able to find a similar projection of the data to PCA (to a sign flip), while only processing a few samples at a time. This can be considered a "toy example", as IPCA is intended for large datasets which do not fit in main memory, requiring incremental approaches.


Incremental PCA of iris dataset
Mean absolute unsigned error 0.002201
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−4
−3
−2
−1
0
1
2
3
4
setosa
versicolor
virginica



-

```
print(__doc__)

# Authors: Kyle Kastner
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_iris
from sklearn.decomposition import PCA, IncrementalPCA

iris = load_iris()
X = iris.data
y = iris.target

n_components = 2
```

(continues on next page)

(continued from previous page)

```
ipca = IncrementalPCA(n_components=n_components, batch_size=10)
X_ipca = ipca.fit_transform(X)

pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(X)

colors = ['navy', 'turquoise', 'darkorange']

for X_transformed, title in [(X_ipca, "Incremental PCA"), (X_pca, "PCA")]:
    plt.figure(figsize=(8, 8))
    for color, i, target_name in zip(colors, [0, 1, 2], iris.target_names):
        plt.scatter(X_transformed[y == i, 0], X_transformed[y == i, 1],
                    color=color, lw=2, label=target_name)

    if "Incremental" in title:
        err = np.abs(np.abs(X_pca) - np.abs(X_ipca)).mean()
        plt.title(title + " of iris dataset\nMean absolute unsigned error "
                  "%.6f" % err)
    else:
        plt.title(title + " of iris dataset")
    plt.legend(loc="best", shadow=False, scatterpoints=1)
    plt.axis([-4, 4, -1.5, 1.5])

plt.show()
```

Total running time of the script: (0 minutes 0.469 seconds)

Estimated memory usage: 8 MB

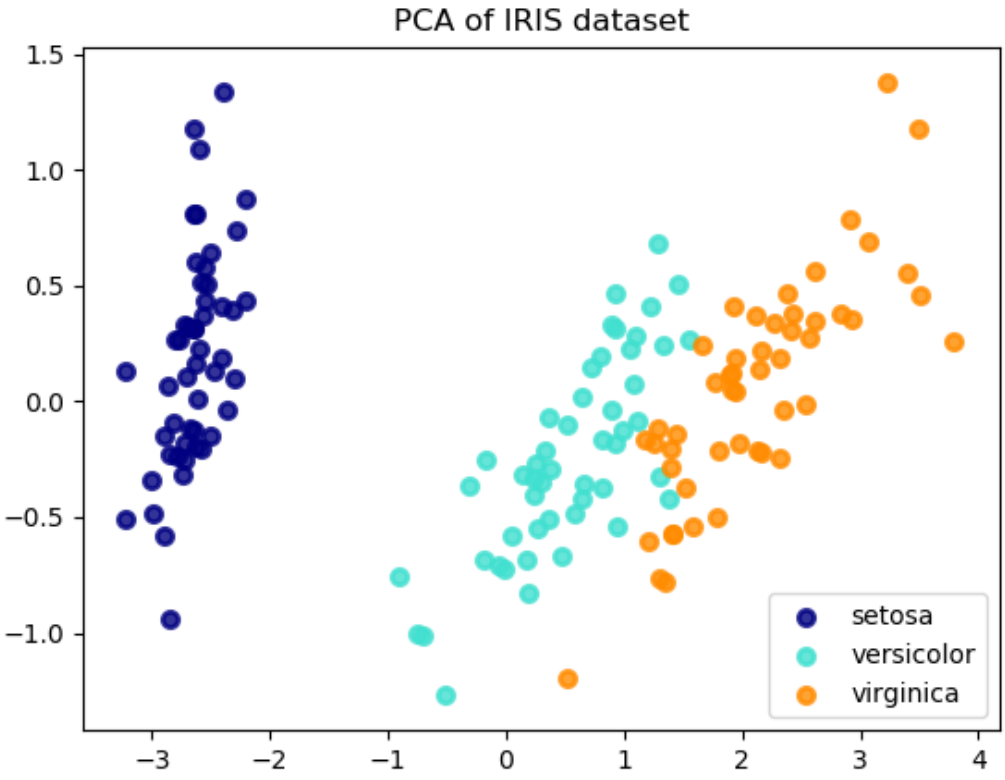
Note: Click *here* to download the full example code or to run this example in your browser via Binder

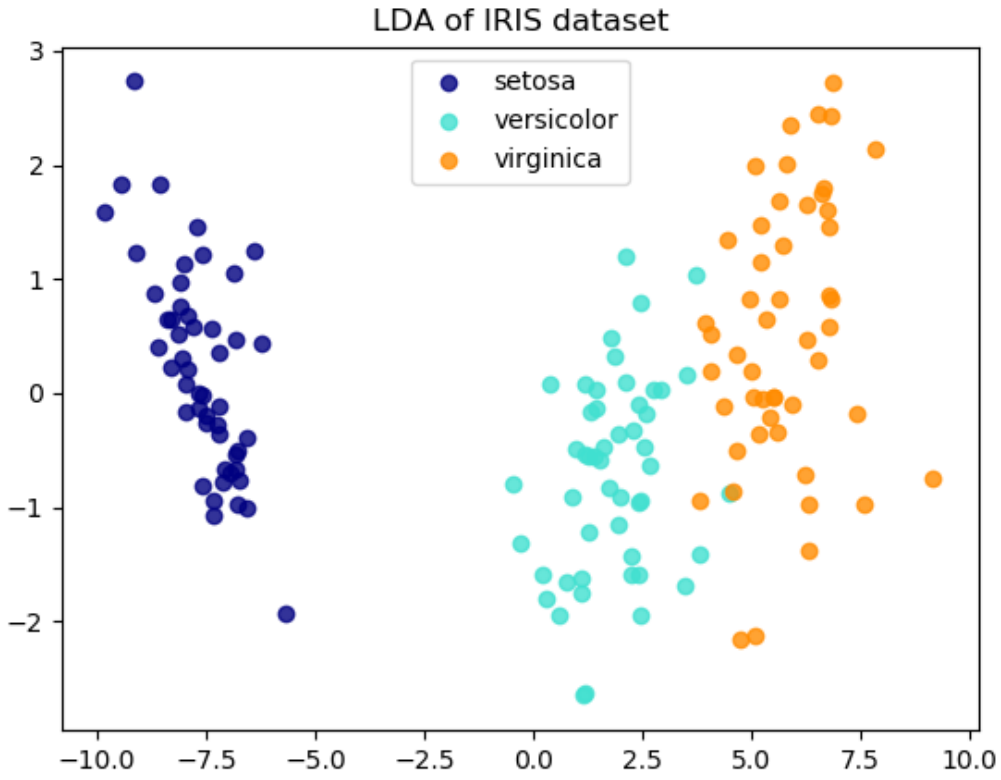
6.10.4 Comparison of LDA and PCA 2D projection of Iris dataset

The Iris dataset represents 3 kind of Iris flowers (Setosa, Versicolour and Virginica) with 4 attributes: sepal length, sepal width, petal length and petal width.

Principal Component Analysis (PCA) applied to this data identifies the combination of attributes (principal components, or directions in the feature space) that account for the most variance in the data. Here we plot the different samples on the 2 first principal components.

Linear Discriminant Analysis (LDA) tries to identify attributes that account for the most variance *between classes*. In particular, LDA, in contrast to PCA, is a supervised method, using known class labels.


PCA of IRIS dataset
1.5
1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3
4
setosa
versicolor
virginica



-

Out:

```
explained variance ratio (first two components): [0.92461872 0.05306648]
```

```
print(__doc__)

import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

iris = datasets.load_iris()

X = iris.data
y = iris.target
target_names = iris.target_names

pca = PCA(n_components=2)
X_r = pca.fit(X).transform(X)

lda = LinearDiscriminantAnalysis(n_components=2)
```

(continues on next page)

(continued from previous page)

```
X_r2 = lda.fit(X, y).transform(X)

# Percentage of variance explained for each components
print('explained variance ratio (first two components): %s'
      % str(pca.explained_variance_ratio_))

plt.figure()
colors = ['navy', 'turquoise', 'darkorange']
lw = 2

for color, i, target_name in zip(colors, [0, 1, 2], target_names):
    plt.scatter(X_r[y == i, 0], X_r[y == i, 1], color=color, alpha=.8, lw=lw,
                label=target_name)
plt.legend(loc='best', shadow=False, scatterpoints=1)
plt.title('PCA of IRIS dataset')

plt.figure()
for color, i, target_name in zip(colors, [0, 1, 2], target_names):
    plt.scatter(X_r2[y == i, 0], X_r2[y == i, 1], alpha=.8, color=color,
                label=target_name)
plt.legend(loc='best', shadow=False, scatterpoints=1)
plt.title('LDA of IRIS dataset')

plt.show()
```

Total running time of the script: (0 minutes 0.418 seconds)

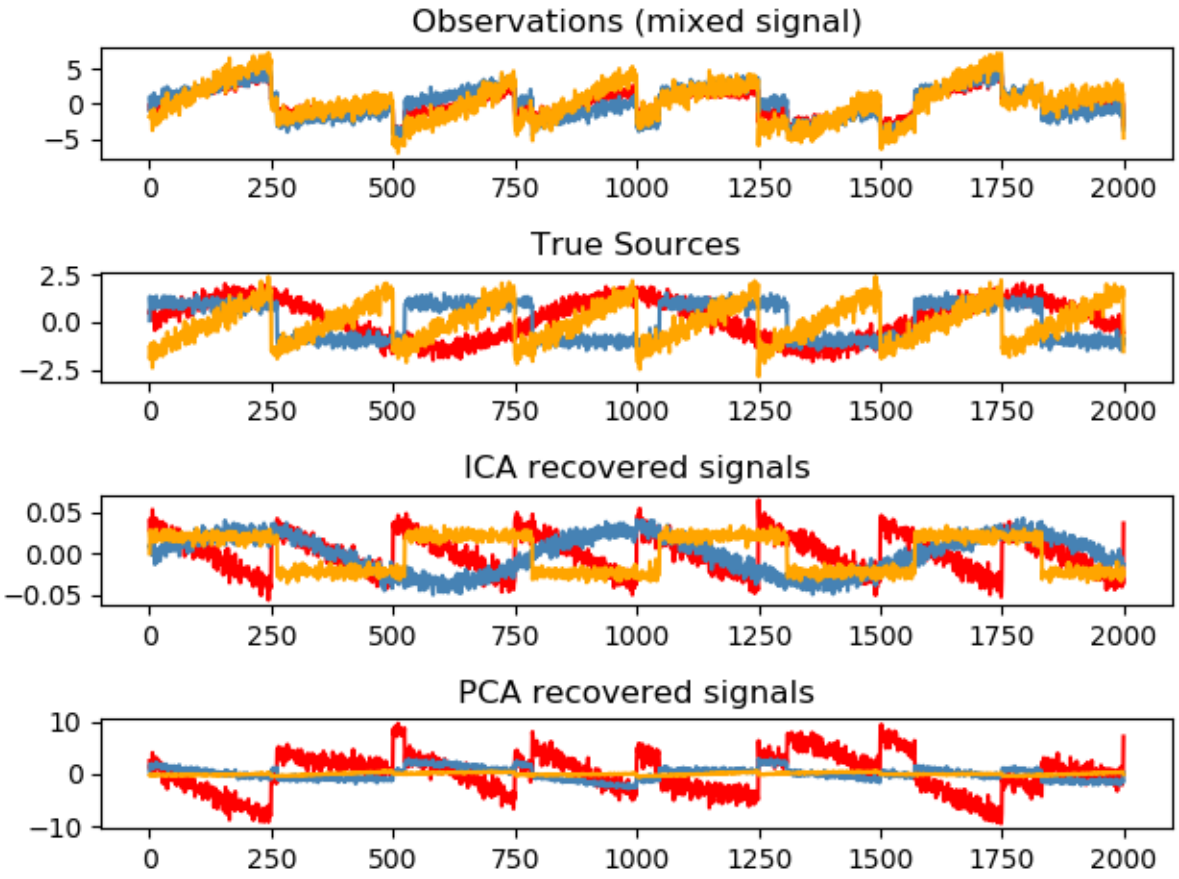
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.10.5 Blind source separation using FastICA

An example of estimating sources from noisy data.

Independent component analysis (ICA) is used to estimate sources given noisy measurements. Imagine 3 instruments playing simultaneously and 3 microphones recording the mixed signals. ICA is used to recover the sources ie. what is played by each instrument. Importantly, PCA fails at recovering our `instruments` since the related signals reflect non-Gaussian processes.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

from sklearn.decomposition import FastICA, PCA

# #############################################################################
# Generate sample data
np.random.seed(0)
n_samples = 2000
time = np.linspace(0, 8, n_samples)

s1 = np.sin(2 * time)  # Signal 1 : sinusoidal signal
s2 = np.sign(np.sin(3 * time))  # Signal 2 : square signal
s3 = signal.sawtooth(2 * np.pi * time)  # Signal 3: saw tooth signal

S = np.c_[s1, s2, s3]
S += 0.2 * np.random.normal(size=S.shape)  # Add noise

S /= S.std(axis=0)  # Standardize data
# Mix data
A = np.array([[1, 1, 1], [0.5, 2, 1.0], [1.5, 1.0, 2.0]])  # Mixing matrix
X = np.dot(S, A.T)  # Generate observations
```

(continues on next page)

(continued from previous page)

```
# Compute ICA
ica = FastICA(n_components=3)
S_ = ica.fit_transform(X)  # Reconstruct signals
A_ = ica.mixing_  # Get estimated mixing matrix

# We can `prove` that the ICA model applies by reverting the unmixing.
assert np.allclose(X, np.dot(S_, A_.T) + ica.mean_)

# For comparison, compute PCA
pca = PCA(n_components=3)
H = pca.fit_transform(X)  # Reconstruct signals based on orthogonal components

# #############################################################################
# Plot results

plt.figure()

models = [X, S, S_, H]
names = ['Observations (mixed signal)',
         'True Sources',
         'ICA recovered signals',
         'PCA recovered signals']
colors = ['red', 'steelblue', 'orange']

for ii, (model, name) in enumerate(zip(models, names), 1):
    plt.subplot(4, 1, ii)
    plt.title(name)
    for sig, color in zip(model.T, colors):
        plt.plot(sig, color=color)

plt.tight_layout()
plt.show()
```

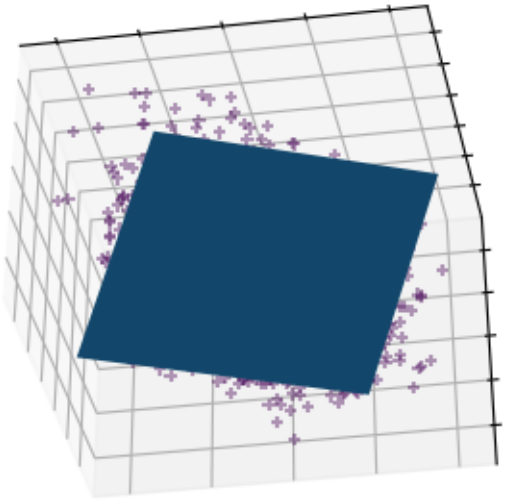
Total running time of the script: (0 minutes 0.559 seconds)

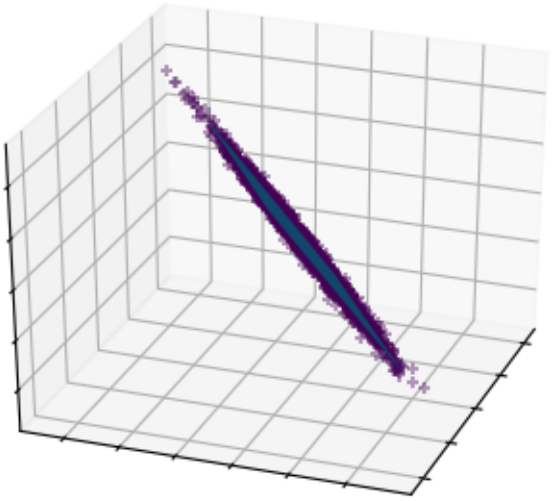
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.10.6 Principal components analysis (PCA)

These figures aid in illustrating how a point cloud can be very flat in one direction–which is where PCA comes in to choose a direction that is not flat.



- 

-

```
print(__doc__)

# Authors: Gael Varoquaux
#          Jaques Grobler
#          Kevin Hughes
# License: BSD 3 clause

from sklearn.decomposition import PCA

from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats


# ############################################################################
# Create the data
```

(continues on next page)

(continued from previous page)

```
e = np.exp(1)
np.random.seed(4)


def pdf(x):
    return 0.5 * (stats.norm(scale=0.25 / e).pdf(x)
                  + stats.norm(scale=4 / e).pdf(x))

y = np.random.normal(scale=0.5, size=(30000))
x = np.random.normal(scale=0.5, size=(30000))
z = np.random.normal(scale=0.1, size=len(x))

density = pdf(x) * pdf(y)
pdf_z = pdf(5 * z)

density *= pdf_z

a = x + y
b = 2 * y
c = a - b + z

norm = np.sqrt(a.var() + b.var())
a /= norm
b /= norm


# ############################################################################
# Plot the figures
def plot_figs(fig_num, elev, azim):
    fig = plt.figure(fig_num, figsize=(4, 3))
    plt.clf()
    ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=elev, azim=azim)

    ax.scatter(a[::10], b[::10], c[::10], c=density[::10], marker='+', alpha=.4)
    Y = np.c_[a, b, c]

    # Using SciPy's SVD, this would be:
    # _, pca_score, V = scipy.linalg.svd(Y, full_matrices=False)

    pca = PCA(n_components=3)
    pca.fit(Y)
    pca_score = pca.explained_variance_ratio_
    V = pca.components_

    x_pca_axis, y_pca_axis, z_pca_axis = 3 * V.T
    x_pca_plane = np.r_[x_pca_axis[:2], - x_pca_axis[1::-1]]
    y_pca_plane = np.r_[y_pca_axis[:2], - y_pca_axis[1::-1]]
    z_pca_plane = np.r_[z_pca_axis[:2], - z_pca_axis[1::-1]]
    x_pca_plane.shape = (2, 2)
    y_pca_plane.shape = (2, 2)
    z_pca_plane.shape = (2, 2)
    ax.plot_surface(x_pca_plane, y_pca_plane, z_pca_plane)
    ax.w_xaxis.set_ticklabels([])
    ax.w_yaxis.set_ticklabels([])
    ax.w_zaxis.set_ticklabels([])
```

(continues on next page)

(continued from previous page)

```
elev = -40
azim = -80
plot_figs(1, elev, azim)

elev = 30
azim = 20
plot_figs(2, elev, azim)

plt.show()
```

Total running time of the script: (0 minutes 0.421 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.10.7 FastICA on 2D point clouds

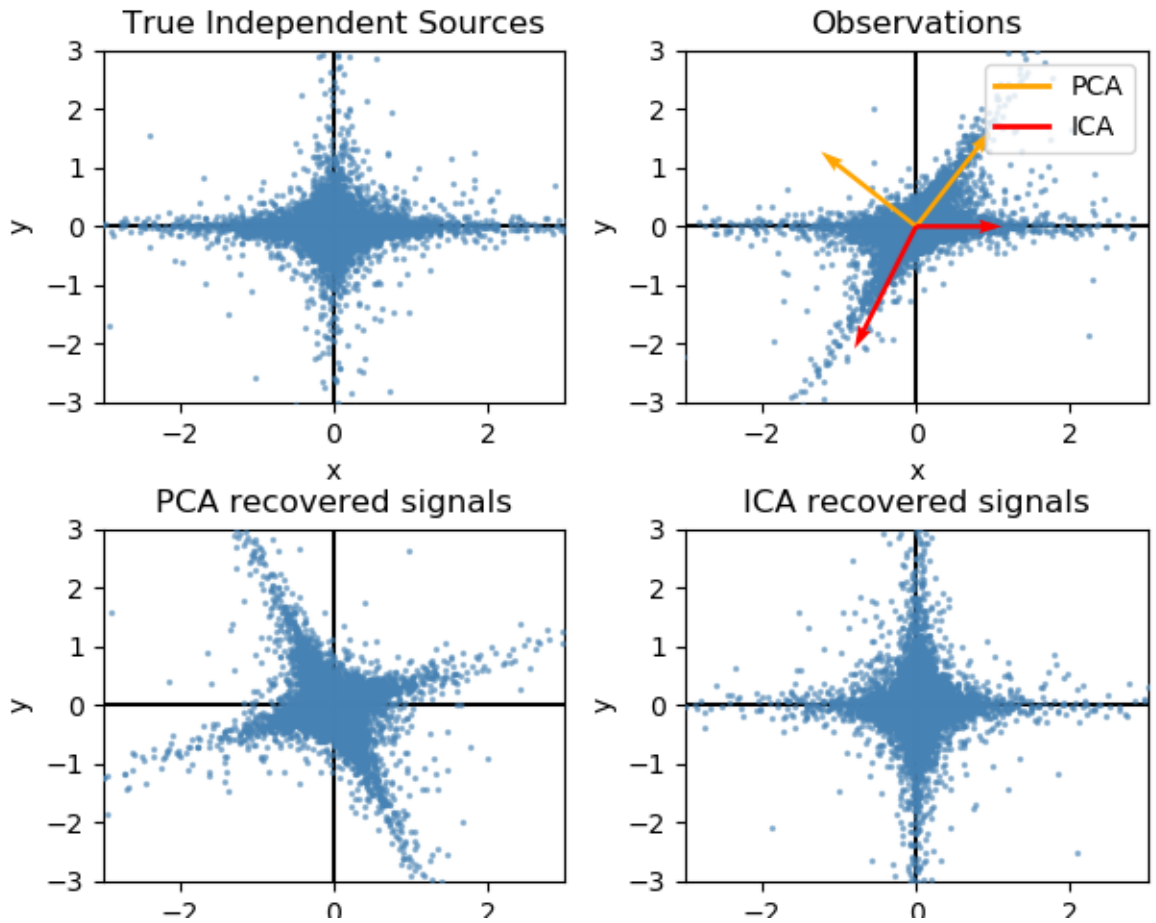
This example illustrates visually in the feature space a comparison by results using two different component analysis techniques.

Independent component analysis (ICA) vs *Principal component analysis (PCA)*.

Representing ICA in the feature space gives the view of 'geometric ICA': ICA is an algorithm that finds directions in the feature space corresponding to projections with high non-Gaussianity. These directions need not be orthogonal in the original feature space, but they are orthogonal in the whitened feature space, in which all directions correspond to the same variance.

PCA, on the other hand, finds orthogonal directions in the raw feature space that correspond to directions accounting for maximum variance.

Here we simulate independent sources using a highly non-Gaussian process, 2 student T with a low number of degrees of freedom (top left figure). We mix them to create observations (top right figure). In this raw observation space, directions identified by PCA are represented by orange vectors. We represent the signal in the PCA space, after whitening by the variance corresponding to the PCA vectors (lower left). Running ICA corresponds to finding a rotation in this space to identify the directions of largest non-Gaussianity (lower right).




```
print(__doc__)

# Authors: Alexandre Gramfort, Gael Varoquaux
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.decomposition import PCA, FastICA

# #############################################################################
# Generate sample data
rng = np.random.RandomState(42)
S = rng.standard_t(1.5, size=(20000, 2))
S[:, 0] *= 2.

# Mix data
A = np.array([[1, 1], [0, 2]])  # Mixing matrix

X = np.dot(S, A.T)  # Generate observations

pca = PCA()
S_pca_ = pca.fit(X).transform(X)

ica = FastICA(random_state=rng)
```

(continues on next page)

(continued from previous page)

```
S_ica_ = ica.fit(X).transform(X)  # Estimate the sources

S_ica_ /= S_ica_.std(axis=0)


# #############################################################################
# Plot results

def plot_samples(S, axis_list=None):
    plt.scatter(S[:, 0], S[:, 1], s=2, marker='o', zorder=10,
                color='steelblue', alpha=0.5)
    if axis_list is not None:
        colors = ['orange', 'red']
        for color, axis in zip(colors, axis_list):
            axis /= axis.std()
            x_axis, y_axis = axis
            # Trick to get legend to work
            plt.plot(0.1 * x_axis, 0.1 * y_axis, linewidth=2, color=color)
            plt.quiver(0, 0, x_axis, y_axis, zorder=11, width=0.01, scale=6,
                       color=color)

    plt.hlines(0, -3, 3)
    plt.vlines(0, -3, 3)
    plt.xlim(-3, 3)
    plt.ylim(-3, 3)
    plt.xlabel('x')
    plt.ylabel('y')

plt.figure()
plt.subplot(2, 2, 1)
plot_samples(S / S.std())
plt.title('True Independent Sources')

axis_list = [pca.components_.T, ica.mixing_]
plt.subplot(2, 2, 2)
plot_samples(X / np.std(X), axis_list=axis_list)
legend = plt.legend(['PCA', 'ICA'], loc='upper right')
legend.set_zorder(100)

plt.title('Observations')

plt.subplot(2, 2, 3)
plot_samples(S_pca_ / np.std(S_pca_, axis=0))
plt.title('PCA recovered signals')

plt.subplot(2, 2, 4)
plot_samples(S_ica_ / np.std(S_ica_))
plt.title('ICA recovered signals')

plt.subplots_adjust(0.09, 0.04, 0.94, 0.94, 0.26, 0.36)
plt.show()
```

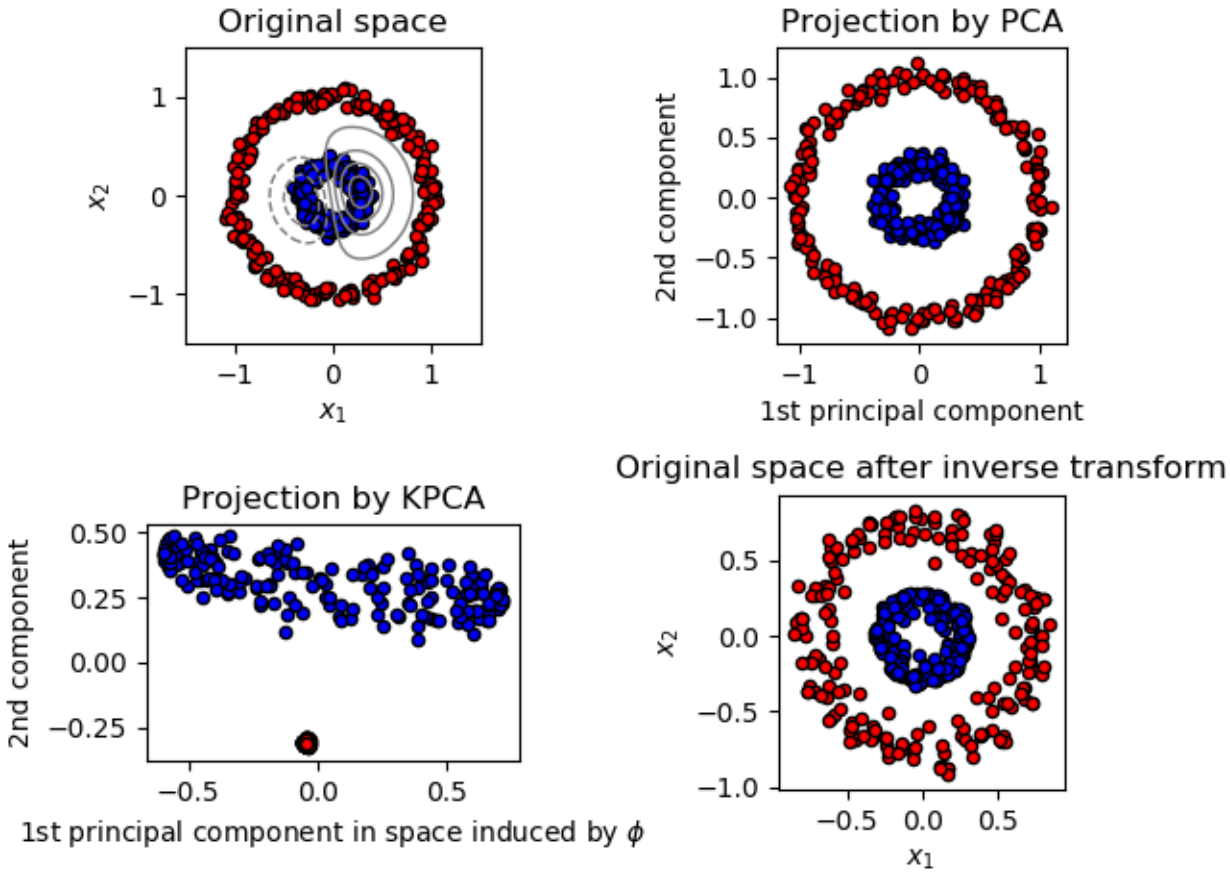
Total running time of the script: (0 minutes 0.599 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.10.8 Kernel PCA

This example shows that Kernel PCA is able to find a projection of the data that makes data linearly separable.




```
print(__doc__)

# Authors: Mathieu Blondel
#          Andreas Mueller
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.decomposition import PCA, KernelPCA
from sklearn.datasets import make_circles

np.random.seed(0)

X, y = make_circles(n_samples=400, factor=.3, noise=.05)
```

(continues on next page)

(continued from previous page)

```
kpca = KernelPCA(kernel="rbf", fit_inverse_transform=True, gamma=10)
X_kpca = kpca.fit_transform(X)
X_back = kpca.inverse_transform(X_kpca)
pca = PCA()
X_pca = pca.fit_transform(X)

# Plot results

plt.figure()
plt.subplot(2, 2, 1, aspect='equal')
plt.title("Original space")
reds = y == 0
blues = y == 1

plt.scatter(X[reds, 0], X[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X[blues, 0], X[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")

X1, X2 = np.meshgrid(np.linspace(-1.5, 1.5, 50), np.linspace(-1.5, 1.5, 50))
X_grid = np.array([np.ravel(X1), np.ravel(X2)]).T
# projection on the first principal component (in the phi space)
Z_grid = kpca.transform(X_grid)[:, 0].reshape(X1.shape)
plt.contour(X1, X2, Z_grid, colors='grey', linewidths=1, origin='lower')

plt.subplot(2, 2, 2, aspect='equal')
plt.scatter(X_pca[reds, 0], X_pca[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_pca[blues, 0], X_pca[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title("Projection by PCA")
plt.xlabel("1st principal component")
plt.ylabel("2nd component")

plt.subplot(2, 2, 3, aspect='equal')
plt.scatter(X_kpca[reds, 0], X_kpca[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_kpca[blues, 0], X_kpca[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title("Projection by KPCA")
plt.xlabel(r"1st principal component in space induced by $\phi$")
plt.ylabel("2nd component")

plt.subplot(2, 2, 4, aspect='equal')
plt.scatter(X_back[reds, 0], X_back[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_back[blues, 0], X_back[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title("Original space after inverse transform")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.474 seconds)

Estimated memory usage: 8 MB

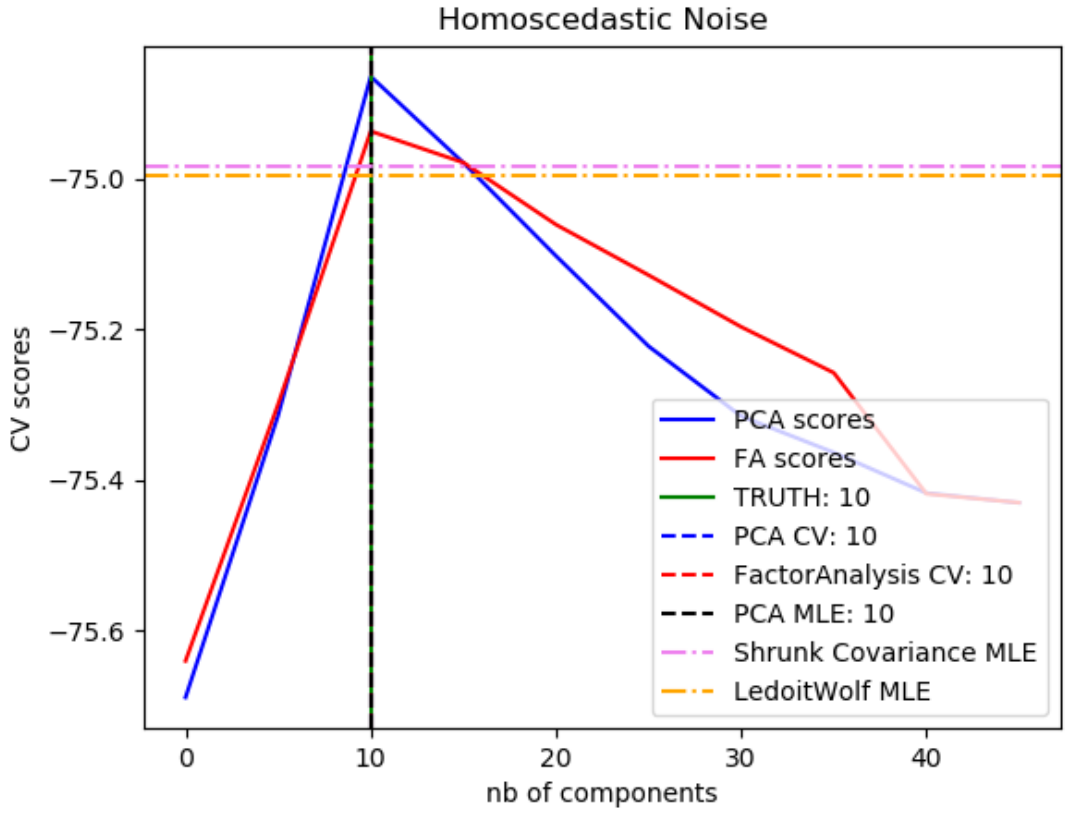
Note: Click *here* to download the full example code or to run this example in your browser via Binder

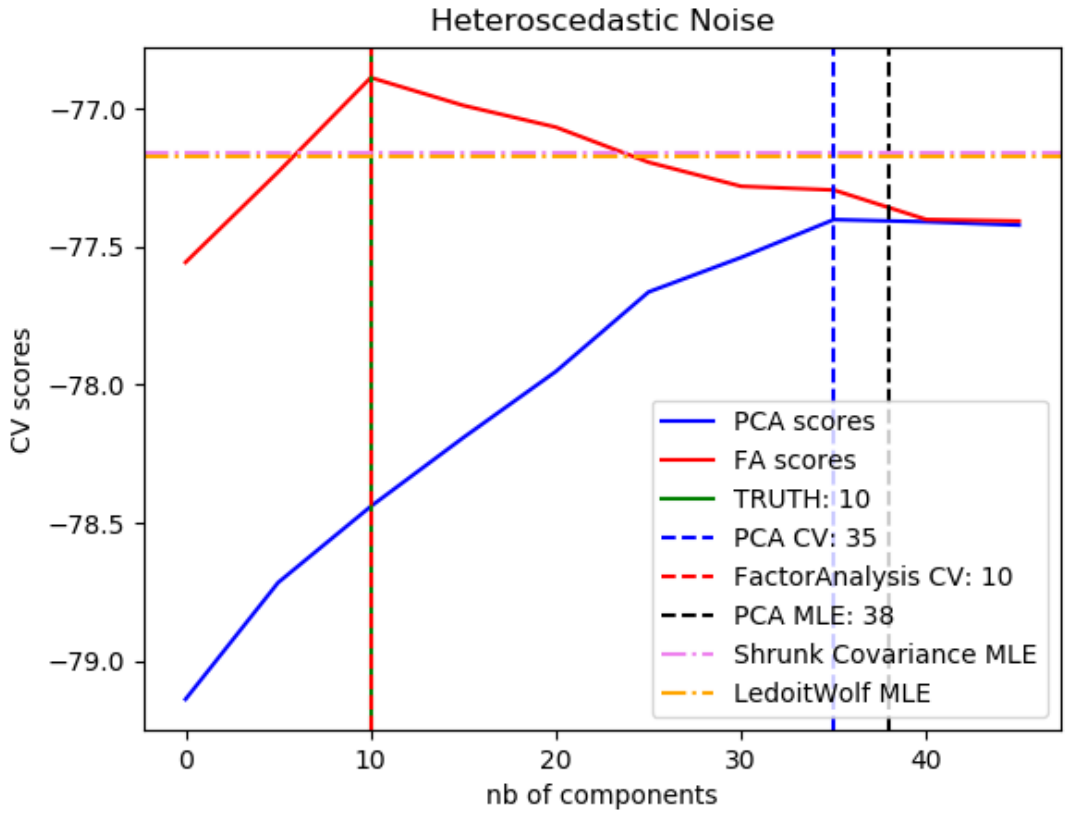
6.10.9 Model selection with Probabilistic PCA and Factor Analysis (FA)

Probabilistic PCA and Factor Analysis are probabilistic models. The consequence is that the likelihood of new data can be used for model selection and covariance estimation. Here we compare PCA and FA with cross-validation on low rank data corrupted with homoscedastic noise (noise variance is the same for each feature) or heteroscedastic noise (noise variance is the different for each feature). In a second step we compare the model likelihood to the likelihoods obtained from shrinkage covariance estimators.

One can observe that with homoscedastic noise both FA and PCA succeed in recovering the size of the low rank subspace. The likelihood with PCA is higher than FA in this case. However PCA fails and overestimates the rank when heteroscedastic noise is present. Under appropriate circumstances the low rank models are more likely than shrinkage models.

The automatic estimation from Automatic Choice of Dimensionality for PCA. NIPS 2000: 598-604 by Thomas P. Minka is also compared.





Out:

```
best n_components by PCA CV = 10
best n_components by FactorAnalysis CV = 10
best n_components by PCA MLE = 10
best n_components by PCA CV = 35
best n_components by FactorAnalysis CV = 10
best n_components by PCA MLE = 38
```

```
# Authors: Alexandre Gramfort
#          Denis A. Engemann
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg

from sklearn.decomposition import PCA, FactorAnalysis
from sklearn.covariance import ShrunkCovariance, LedoitWolf
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
```

(continues on next page)

(continued from previous page)

```
print(__doc__)

# #############################################################################
# Create the data

n_samples, n_features, rank = 1000, 50, 10
sigma = 1.
rng = np.random.RandomState(42)
U, _, _ = linalg.svd(rng.randn(n_features, n_features))
X = np.dot(rng.randn(n_samples, rank), U[:, :rank].T)

# Adding homoscedastic noise
X_homo = X + sigma * rng.randn(n_samples, n_features)

# Adding heteroscedastic noise
sigmas = sigma * rng.rand(n_features) + sigma / 2.
X_hetero = X + rng.randn(n_samples, n_features) * sigmas

# #############################################################################
# Fit the models

n_components = np.arange(0, n_features, 5)  # options for n_components


def compute_scores(X):
    pca = PCA(svd_solver='full')
    fa = FactorAnalysis()

    pca_scores, fa_scores = [], []
    for n in n_components:
        pca.n_components = n
        fa.n_components = n
        pca_scores.append(np.mean(cross_val_score(pca, X)))
        fa_scores.append(np.mean(cross_val_score(fa, X)))

    return pca_scores, fa_scores


def shrunk_cov_score(X):
    shrinkages = np.logspace(-2, 0, 30)
    cv = GridSearchCV(ShrunkCovariance(), {'shrinkage': shrinkages})
    return np.mean(cross_val_score(cv.fit(X).best_estimator_, X))


def lw_score(X):
    return np.mean(cross_val_score(LedoitWolf(), X))


for X, title in [(X_homo, 'Homoscedastic Noise'),
                 (X_hetero, 'Heteroscedastic Noise')]:
    pca_scores, fa_scores = compute_scores(X)
    n_components_pca = n_components[np.argmax(pca_scores)]
    n_components_fa = n_components[np.argmax(fa_scores)]

    pca = PCA(svd_solver='full', n_components='mle')
    pca.fit(X)
    n_components_pca_mle = pca.n_components_
```

(continues on next page)

(continued from previous page)

```
    print("best n_components by PCA CV = %d" % n_components_pca)
    print("best n_components by FactorAnalysis CV = %d" % n_components_fa)
    print("best n_components by PCA MLE = %d" % n_components_pca_mle)

    plt.figure()
    plt.plot(n_components, pca_scores, 'b', label='PCA scores')
    plt.plot(n_components, fa_scores, 'r', label='FA scores')
    plt.axvline(rank, color='g', label='TRUTH: %d' % rank, linestyle='-')
    plt.axvline(n_components_pca, color='b',
                label='PCA CV: %d' % n_components_pca, linestyle='--')
    plt.axvline(n_components_fa, color='r',
                label='FactorAnalysis CV: %d' % n_components_fa,
                linestyle='--')
    plt.axvline(n_components_pca_mle, color='k',
                label='PCA MLE: %d' % n_components_pca_mle, linestyle='--')

    # compare with other covariance estimators
    plt.axhline(shrunk_cov_score(X), color='violet',
                label='Shrunk Covariance MLE', linestyle='-.')
    plt.axhline(lw_score(X), color='orange',
                label='LedoitWolf MLE' % n_components_pca_mle, linestyle='-.')

    plt.xlabel('nb of components')
    plt.ylabel('CV scores')
    plt.legend(loc='lower right')
    plt.title(title)

plt.show()
```

Total running time of the script: (0 minutes 8.397 seconds)

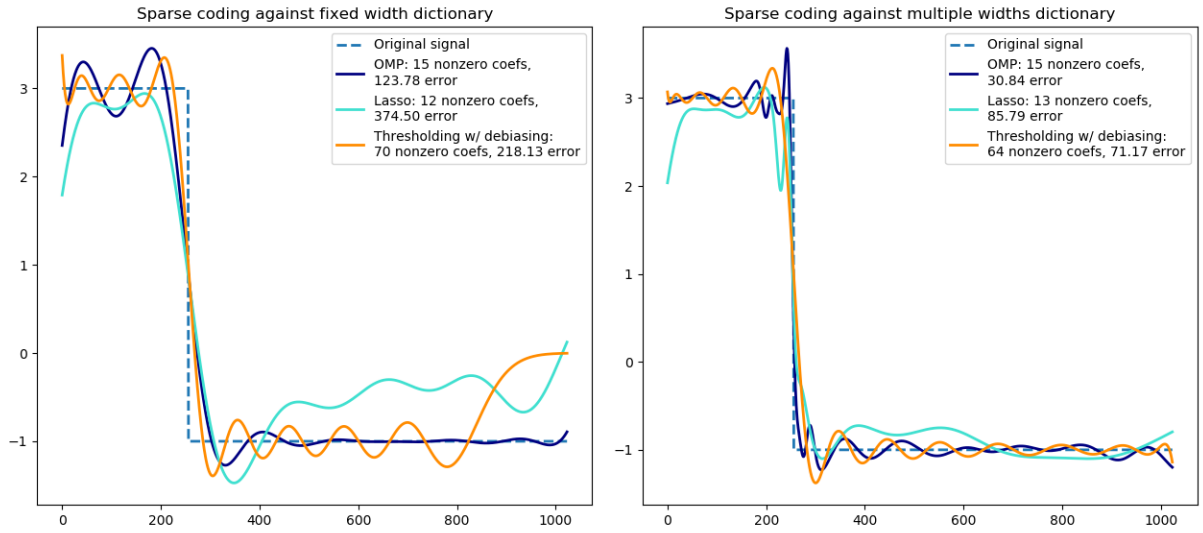
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.10.10 Sparse coding with a precomputed dictionary

Transform a signal as a sparse combination of Ricker wavelets. This example visually compares different sparse coding methods using the *sklearn.decomposition.SparseCoder* estimator. The Ricker (also known as Mexican hat or the second derivative of a Gaussian) is not a particularly good kernel to represent piecewise constant signals like this one. It can therefore be seen how much adding different widths of atoms matters and it therefore motivates learning the dictionary to best fit your type of signals.

The richer dictionary on the right is not larger in size, heavier subsampling is performed in order to stay on the same order of magnitude.




```
print(__doc__)

from distutils.version import LooseVersion

import numpy as np
import matplotlib.pyplot as plt

from sklearn.decomposition import SparseCoder


def ricker_function(resolution, center, width):
    """Discrete sub-sampled Ricker (Mexican hat) wavelet"""
    x = np.linspace(0, resolution - 1, resolution)
    x = ((2 / (np.sqrt(3 * width) * np.pi ** .25))
         * (1 - (x - center) ** 2 / width ** 2)
         * np.exp(-(x - center) ** 2 / (2 * width ** 2)))
    return x


def ricker_matrix(width, resolution, n_components):
    """Dictionary of Ricker (Mexican hat) wavelets"""
    centers = np.linspace(0, resolution - 1, n_components)
    D = np.empty((n_components, resolution))
    for i, center in enumerate(centers):
        D[i] = ricker_function(resolution, center, width)
    D /= np.sqrt(np.sum(D ** 2, axis=1))[:, np.newaxis]
    return D


resolution = 1024
subsampling = 3  # subsampling factor
width = 100
n_components = resolution // subsampling

# Compute a wavelet dictionary
D_fixed = ricker_matrix(width=width, resolution=resolution,
                        n_components=n_components)
```

(continues on next page)

(continued from previous page)

```
D_multi = np.r_[tuple(ricker_matrix(width=w, resolution=resolution,
                      n_components=n_components // 5)
                for w in (10, 50, 100, 500, 1000))]

# Generate a signal
y = np.linspace(0, resolution - 1, resolution)
first_quarter = y < resolution / 4
y[first_quarter] = 3.
y[np.logical_not(first_quarter)] = -1.

# List the different sparse coding methods in the following format:
# (title, transform_algorithm, transform_alpha,
#  transform_n_nozero_coefs, color)
estimators = [('OMP', 'omp', None, 15, 'navy'),
              ('Lasso', 'lasso_lars', 2, None, 'turquoise'), ]
lw = 2
# Avoid FutureWarning about default value change when numpy >= 1.14
lstsq_rcond = None if LooseVersion(np.__version__) >= '1.14' else -1

plt.figure(figsize=(13, 6))
for subplot, (D, title) in enumerate(zip((D_fixed, D_multi),
                                         ('fixed width', 'multiple widths'))):
    plt.subplot(1, 2, subplot + 1)
    plt.title('Sparse coding against %s dictionary' % title)
    plt.plot(y, lw=lw, linestyle='--', label='Original signal')
    # Do a wavelet approximation
    for title, algo, alpha, n_nonzero, color in estimators:
        coder = SparseCoder(dictionary=D, transform_n_nonzero_coefs=n_nonzero,
                            transform_alpha=alpha, transform_algorithm=algo)
        x = coder.transform(y.reshape(1, -1))
        density = len(np.flatnonzero(x))
        x = np.ravel(np.dot(x, D))
        squared_error = np.sum((y - x) ** 2)
        plt.plot(x, color=color, lw=lw,
                 label='%s: %s nonzero coefs,\n%.2f error'
                 % (title, density, squared_error))

    # Soft thresholding debiasing
    coder = SparseCoder(dictionary=D, transform_algorithm='threshold',
                        transform_alpha=20)
    x = coder.transform(y.reshape(1, -1))
    _, idx = np.where(x != 0)
    x[0, idx], _, _, _ = np.linalg.lstsq(D[idx, :].T, y, rcond=lstsq_rcond)
    x = np.ravel(np.dot(x, D))
    squared_error = np.sum((y - x) ** 2)
    plt.plot(x, color='darkorange', lw=lw,
             label='Thresholding w/ debiasing:\n%d nonzero coefs, %.2f error'
             % (len(idx), squared_error))
    plt.axis('tight')
    plt.legend(shadow=False, loc='best')
plt.subplots_adjust(.04, .07, .97, .90, .09, .2)
plt.show()
```

Total running time of the script: (0 minutes 0.474 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.10.11 Image denoising using dictionary learning

An example comparing the effect of reconstructing noisy fragments of a raccoon face image using firstly online *Dictionary Learning* and various transform methods.

The dictionary is fitted on the distorted left half of the image, and subsequently used to reconstruct the right half. Note that even better performance could be achieved by fitting to an undistorted (i.e. noiseless) image, but here we start from the assumption that it is not available.

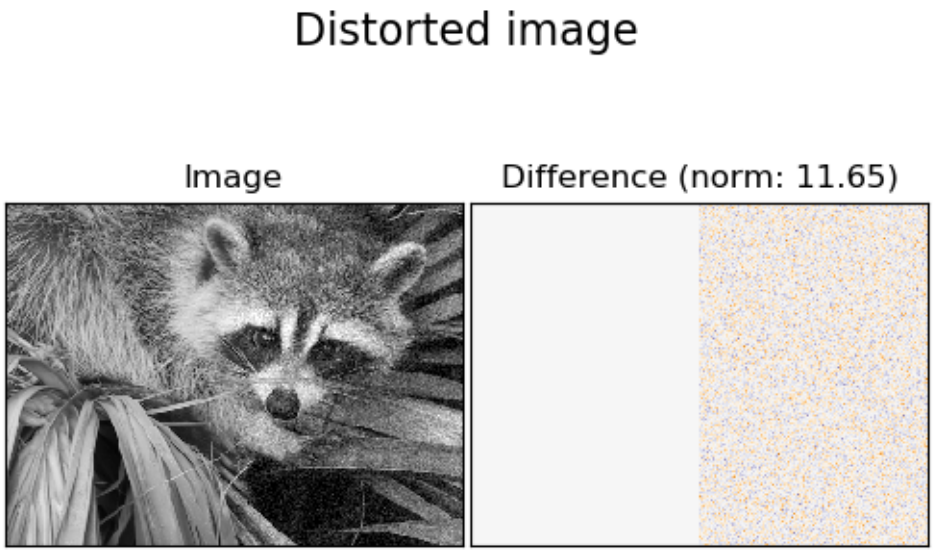
A common practice for evaluating the results of image denoising is by looking at the difference between the reconstruction and the original image. If the reconstruction is perfect this will look like Gaussian noise.

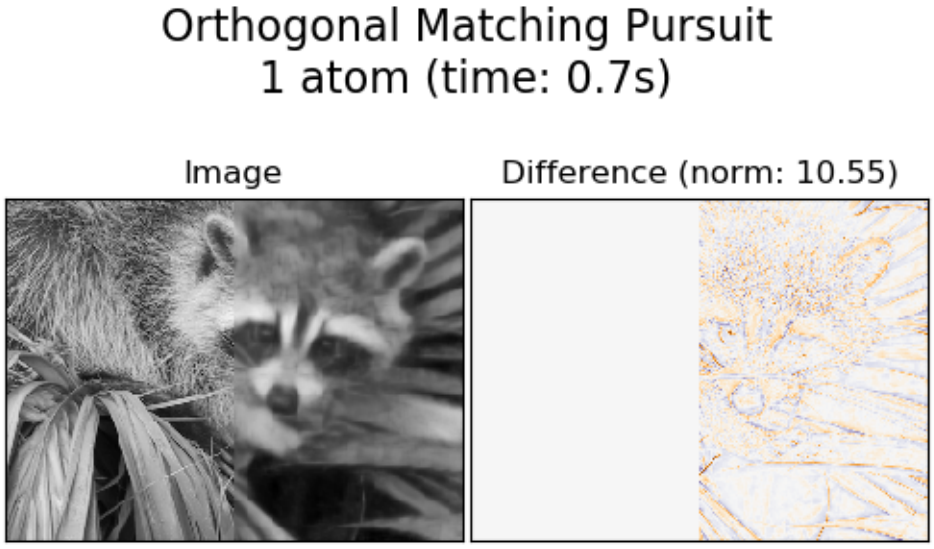
It can be seen from the plots that the results of *Orthogonal Matching Pursuit (OMP)* with two non-zero coefficients is a bit less biased than when keeping only one (the edges look less prominent). It is in addition closer from the ground truth in Frobenius norm.

The result of *Least Angle Regression* is much more strongly biased: the difference is reminiscent of the local intensity value of the original image.

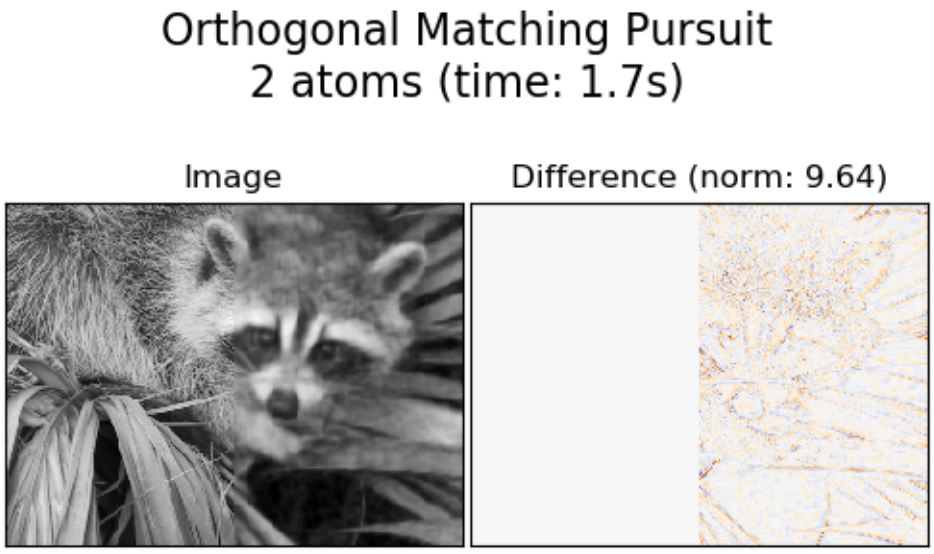
Thresholding is clearly not useful for denoising, but it is here to show that it can produce a suggestive output with very high speed, and thus be useful for other tasks such as object classification, where performance is not necessarily related to visualisation.

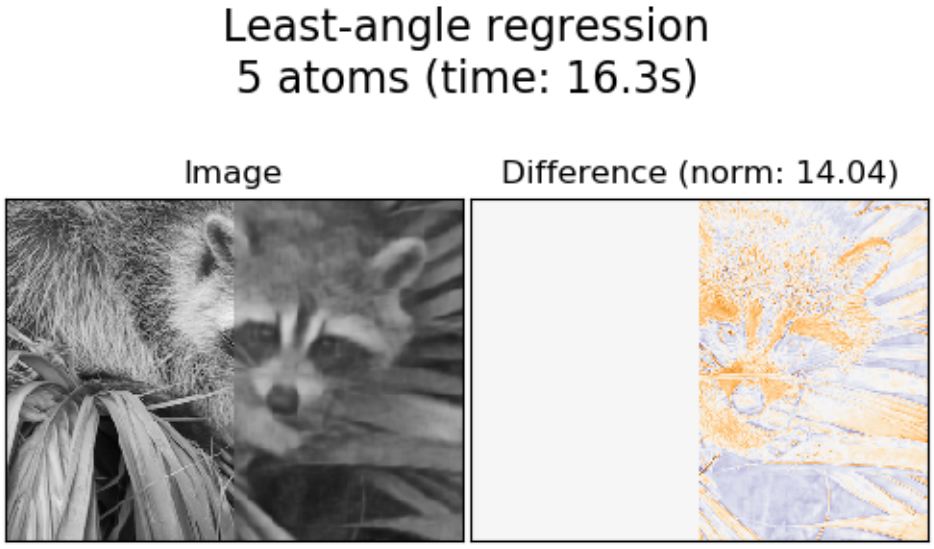
- 


Distorted image
Image
Difference (norm: 11.65)

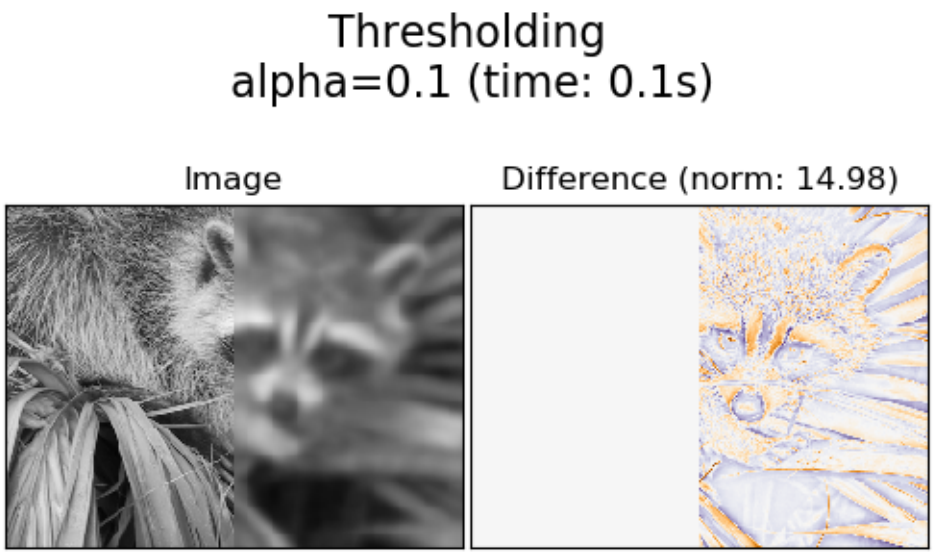
- 
Orthogonal Matching Pursuit
1 atom (time: 0.7s)
Image
Difference (norm: 10.55)

-


Orthogonal Matching Pursuit
2 atoms (time: 1.7s)
Image
Difference (norm: 9.64)

- 
Least-angle regression
5 atoms (time: 16.3s)
Image
Difference (norm: 14.04)

-



-

Out:

```
Distorting image...
Extracting reference patches...
done in 0.01s.
Learning the dictionary...
done in 2.81s.
Extracting noisy patches...
done in 0.00s.
Orthogonal Matching Pursuit
1 atom...
done in 0.73s.
Orthogonal Matching Pursuit
2 atoms...
done in 1.66s.
Least-angle regression
5 atoms...
done in 16.25s.
Thresholding
 alpha=0.1...
done in 0.13s.
```

```
print(__doc__)

from time import time

import matplotlib.pyplot as plt
import numpy as np
import scipy as sp
```

(continues on next page)

(continued from previous page)

```
from sklearn.decomposition import MiniBatchDictionaryLearning
from sklearn.feature_extraction.image import extract_patches_2d
from sklearn.feature_extraction.image import reconstruct_from_patches_2d


try:  # SciPy >= 0.16 have face in misc
    from scipy.misc import face
    face = face(gray=True)
except ImportError:
    face = sp.face(gray=True)

# Convert from uint8 representation with values between 0 and 255 to
# a floating point representation with values between 0 and 1.
face = face / 255.

# downsample for higher speed
face = face[::4, ::4] + face[1::4, ::4] + face[::4, 1::4] + face[1::4, 1::4]
face /= 4.0
height, width = face.shape

# Distort the right half of the image
print('Distorting image...')
distorted = face.copy()
distorted[:, width // 2:] += 0.075 * np.random.randn(height, width // 2)

# Extract all reference patches from the left half of the image
print('Extracting reference patches...')
t0 = time()
patch_size = (7, 7)
data = extract_patches_2d(distorted[:, :width // 2], patch_size)
data = data.reshape(data.shape[0], -1)
data -= np.mean(data, axis=0)
data /= np.std(data, axis=0)
print('done in %.2fs.' % (time() - t0))

# ##############################################################################
# Learn the dictionary from reference patches

print('Learning the dictionary...')
t0 = time()
dico = MiniBatchDictionaryLearning(n_components=100, alpha=1, n_iter=500)
V = dico.fit(data).components_
dt = time() - t0
print('done in %.2fs.' % dt)

plt.figure(figsize=(4.2, 4))
for i, comp in enumerate(V[:100]):
    plt.subplot(10, 10, i + 1)
    plt.imshow(comp.reshape(patch_size), cmap=plt.cm.gray_r,
               interpolation='nearest')
    plt.xticks(())
    plt.yticks(())
plt.suptitle('Dictionary learned from face patches\n' +
             'Train time %.1fs on %d patches' % (dt, len(data)),
             fontsize=16)
plt.subplots_adjust(0.08, 0.02, 0.92, 0.85, 0.08, 0.23)
```

(continues on next page)

(continued from previous page)

```
# ##############################################################################
# Display the distorted image

def show_with_diff(image, reference, title):
    """Helper function to display denoising"""
    plt.figure(figsize=(5, 3.3))
    plt.subplot(1, 2, 1)
    plt.title('Image')
    plt.imshow(image, vmin=0, vmax=1, cmap=plt.cm.gray,
               interpolation='nearest')
    plt.xticks(())
    plt.yticks(())
    plt.subplot(1, 2, 2)
    difference = image - reference

    plt.title('Difference (norm: %.2f)' % np.sqrt(np.sum(difference ** 2)))
    plt.imshow(difference, vmin=-0.5, vmax=0.5, cmap=plt.cm.PuOr,
               interpolation='nearest')
    plt.xticks(())
    plt.yticks(())
    plt.suptitle(title, size=16)
    plt.subplots_adjust(0.02, 0.02, 0.98, 0.79, 0.02, 0.2)

show_with_diff(distorted, face, 'Distorted image')

# ##############################################################################
# Extract noisy patches and reconstruct them using the dictionary

print('Extracting noisy patches... ')
t0 = time()
data = extract_patches_2d(distorted[:, width // 2:], patch_size)
data = data.reshape(data.shape[0], -1)
intercept = np.mean(data, axis=0)
data -= intercept
print('done in %.2fs.' % (time() - t0))

transform_algorithms = [
    ('Orthogonal Matching Pursuit\n1 atom', 'omp',
     {'transform_n_nonzero_coefs': 1}),
    ('Orthogonal Matching Pursuit\n2 atoms', 'omp',
     {'transform_n_nonzero_coefs': 2}),
    ('Least-angle regression\n5 atoms', 'lars',
     {'transform_n_nonzero_coefs': 5}),
    ('Thresholding\n alpha=0.1', 'threshold', {'transform_alpha': .1})]

reconstructions = {}
for title, transform_algorithm, kwargs in transform_algorithms:
    print(title + '...')
    reconstructions[title] = face.copy()
    t0 = time()
    dico.set_params(transform_algorithm=transform_algorithm, **kwargs)
    code = dico.transform(data)
    patches = np.dot(code, V)

    patches += intercept
    patches = patches.reshape(len(data), *patch_size)
```

(continues on next page)

(continued from previous page)

```
    if transform_algorithm == 'threshold':
        patches -= patches.min()
        patches /= patches.max()
    reconstructions[title][:, width // 2:] = reconstruct_from_patches_2d(
        patches, (height, width // 2))
    dt = time() - t0
    print('done in %.2fs.' % dt)
    show_with_diff(reconstructions[title], face,
                   title + ' (time: %.1fs)' % dt)

plt.show()
```

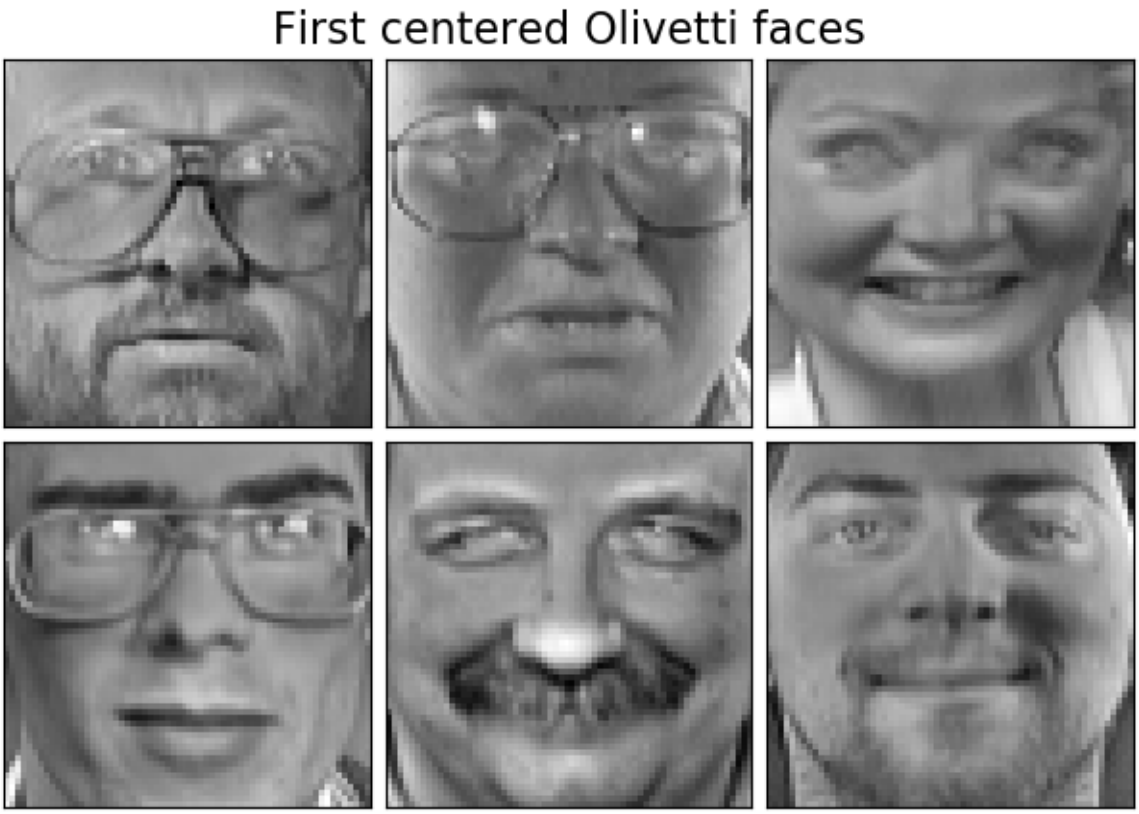
Total running time of the script: (0 minutes 23.829 seconds)

Estimated memory usage: 73 MB

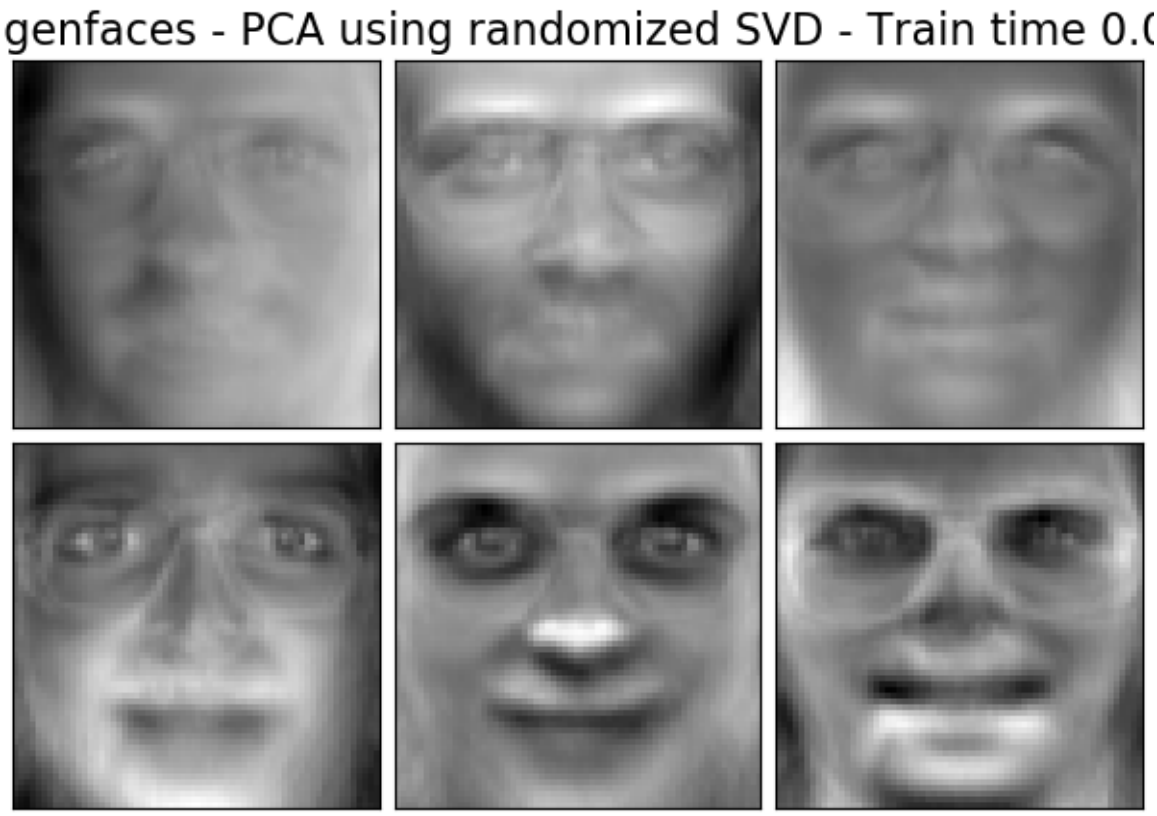
Note: Click *here* to download the full example code or to run this example in your browser via Binder

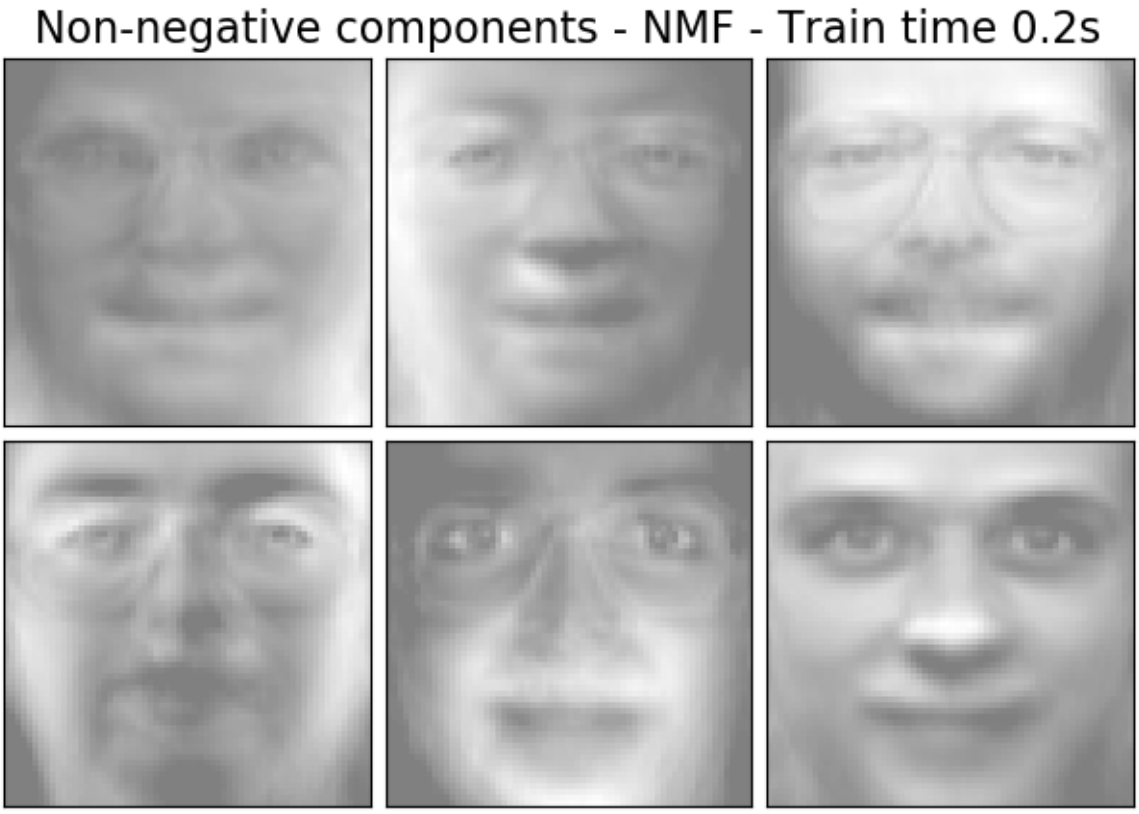
6.10.12 Faces dataset decompositions

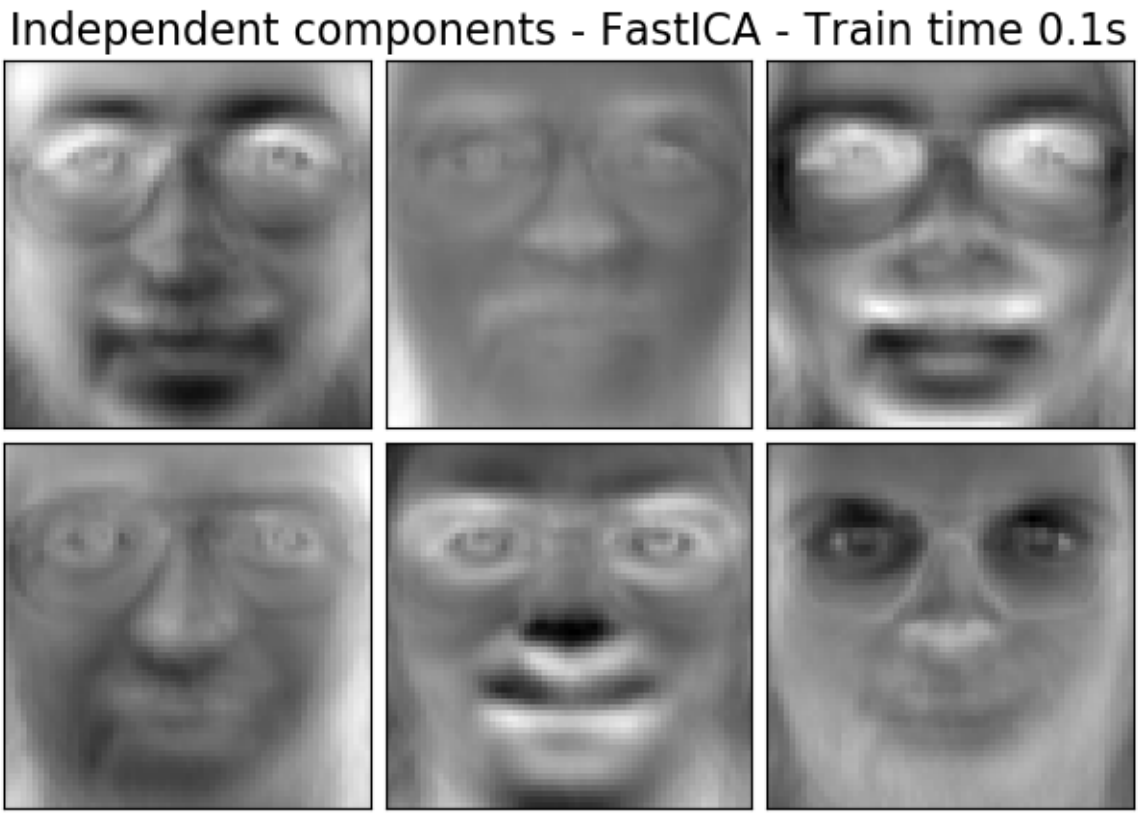
This example applies to *The Olivetti faces dataset* different unsupervised matrix decomposition (dimension reduction) methods from the module `sklearn.decomposition` (see the documentation chapter *Decomposing signals in components (matrix factorization problems)*) .

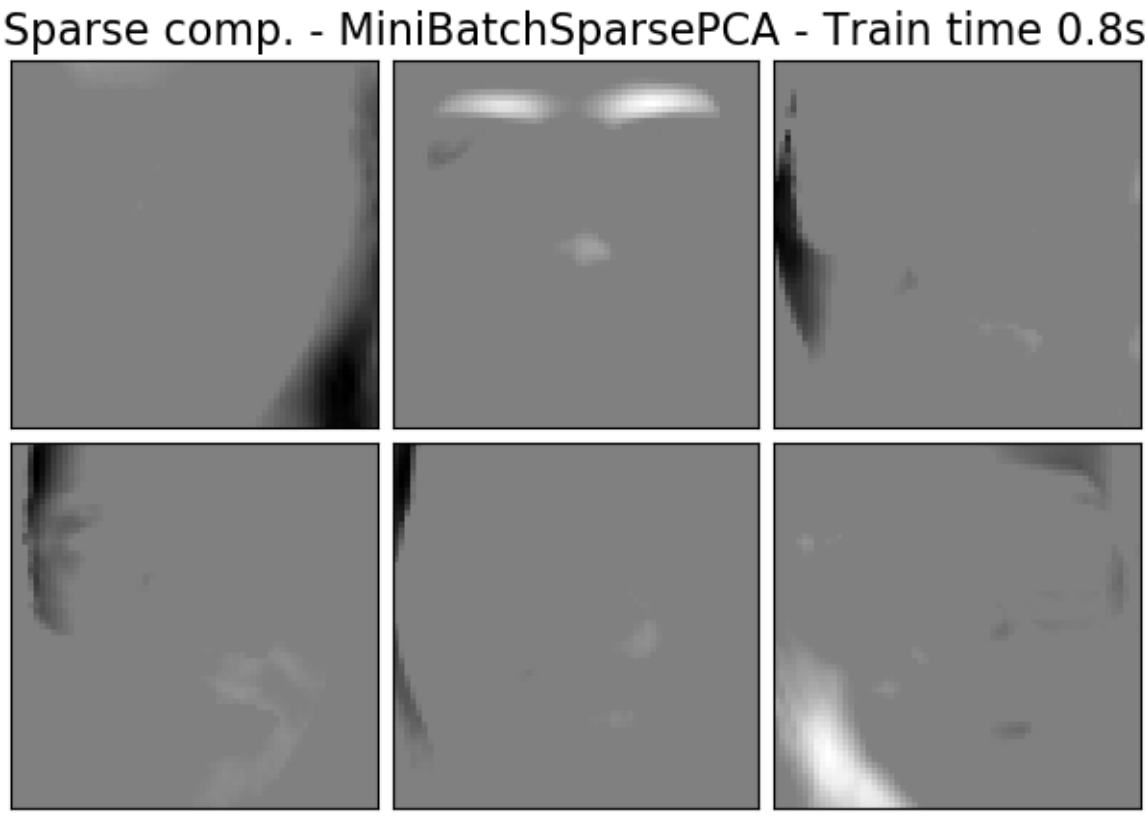


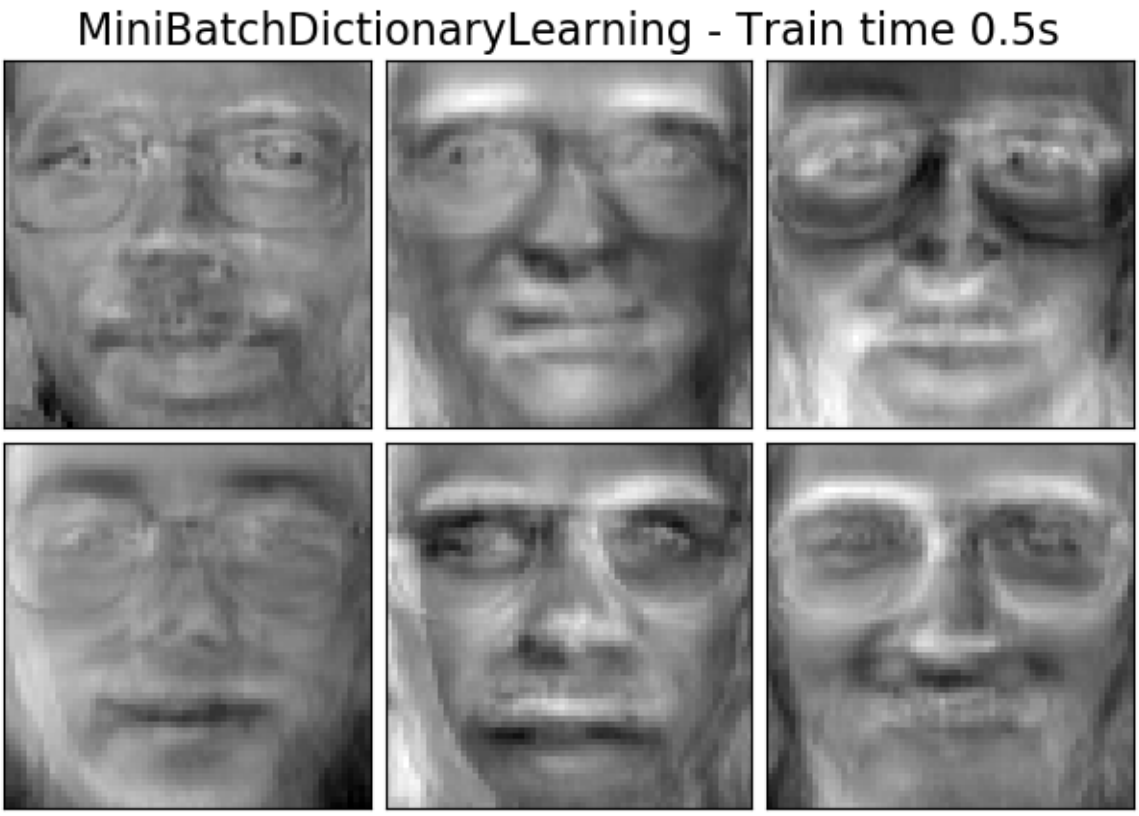
-

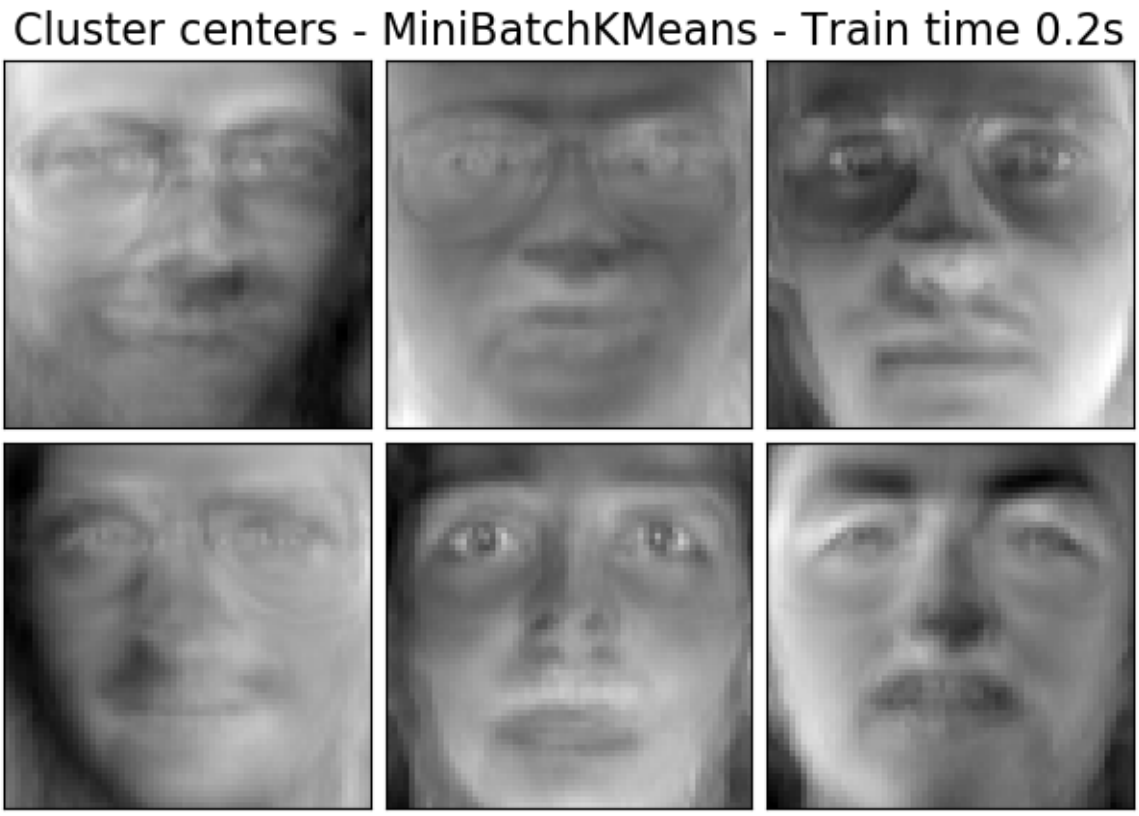

genfaces - PCA using randomized SVD - Train time 0.(


Non-negative components - NMF - Train time 0.2s

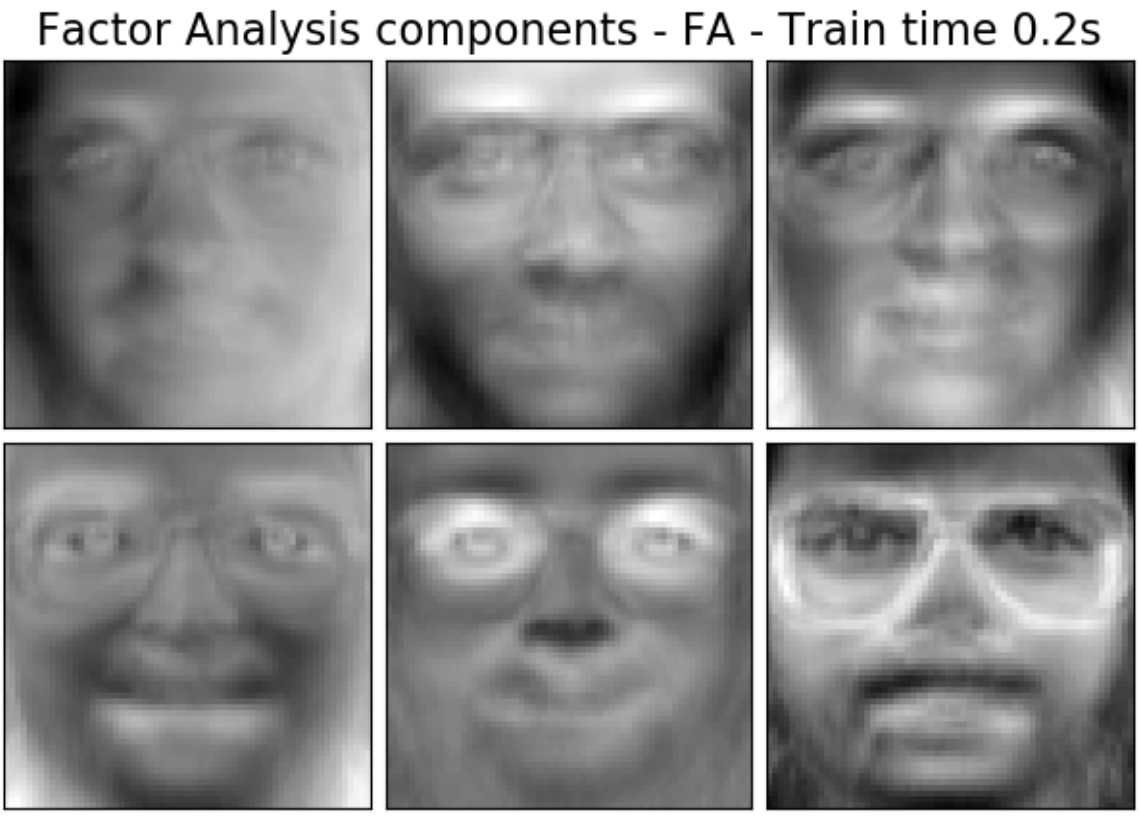

Independent components - FastICA - Train time 0.1s

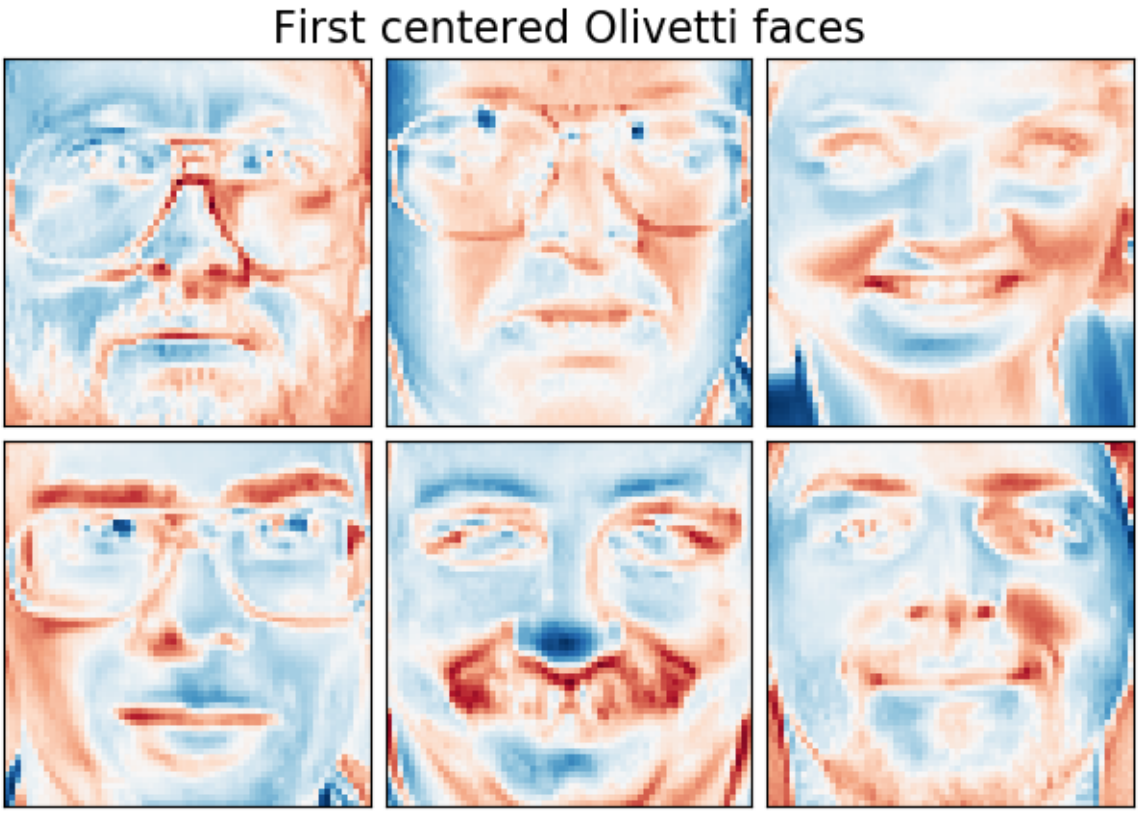

Sparse comp. - MiniBatchSparsePCA - Train time 0.8s

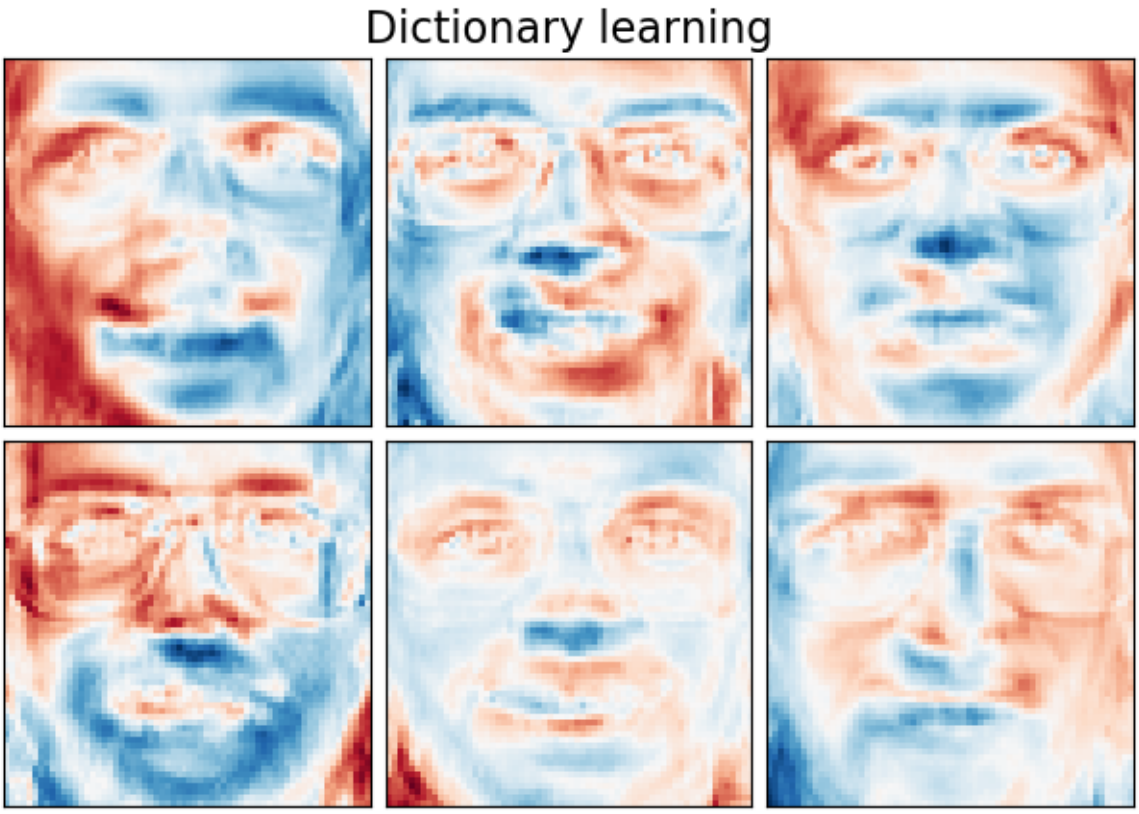

MiniBatchDictionaryLearning - Train time 0.5s

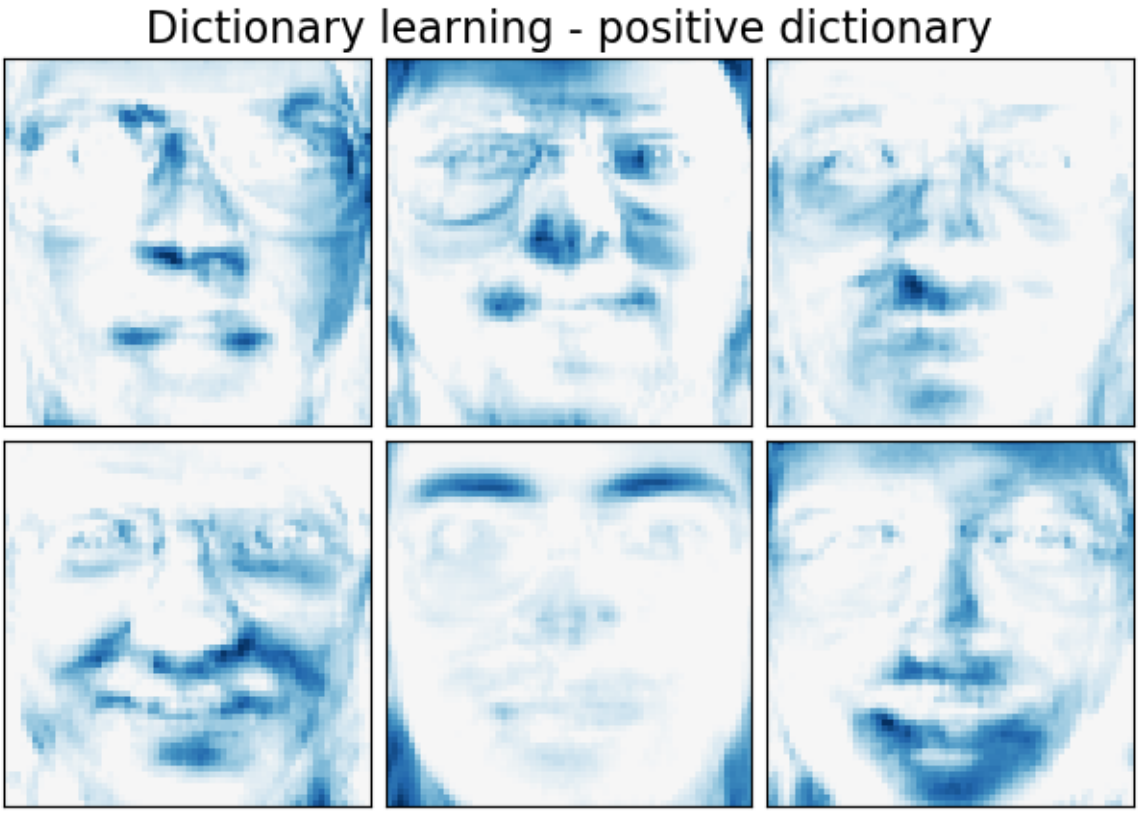

Cluster centers - MiniBatchKMeans - Train time 0.2s

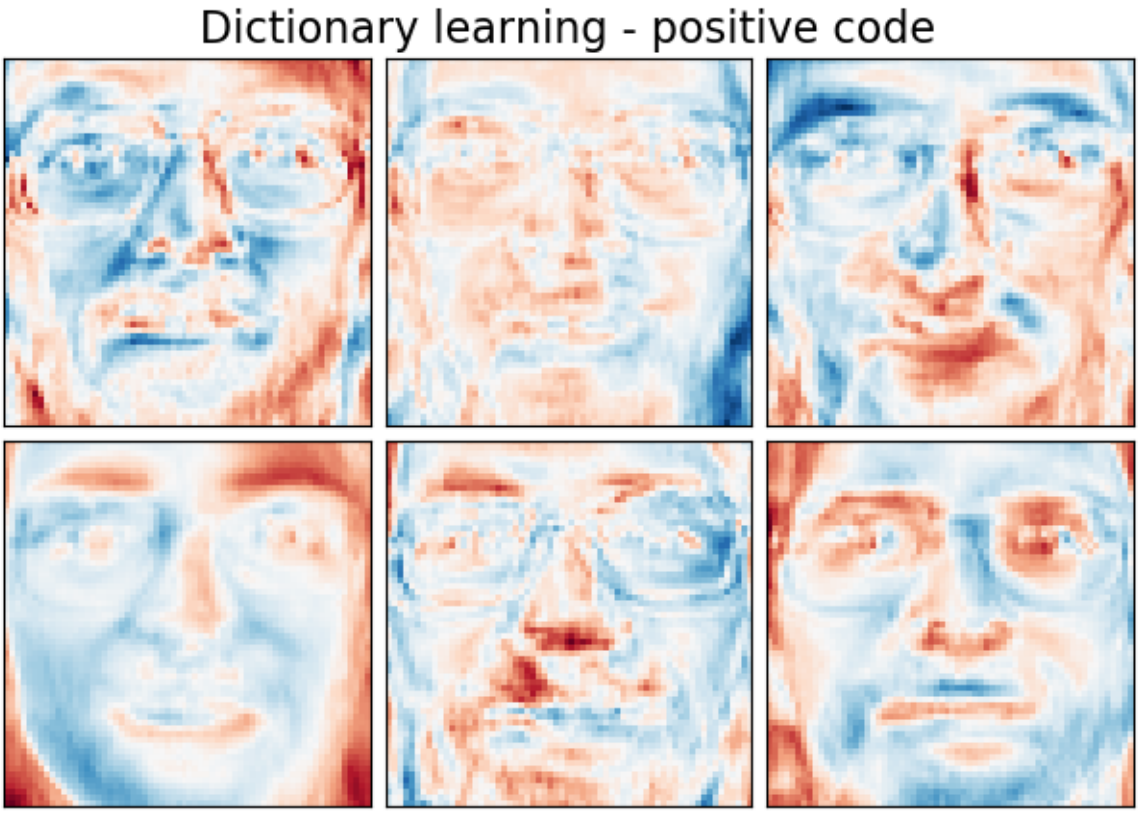
- 
Pixelwise variance

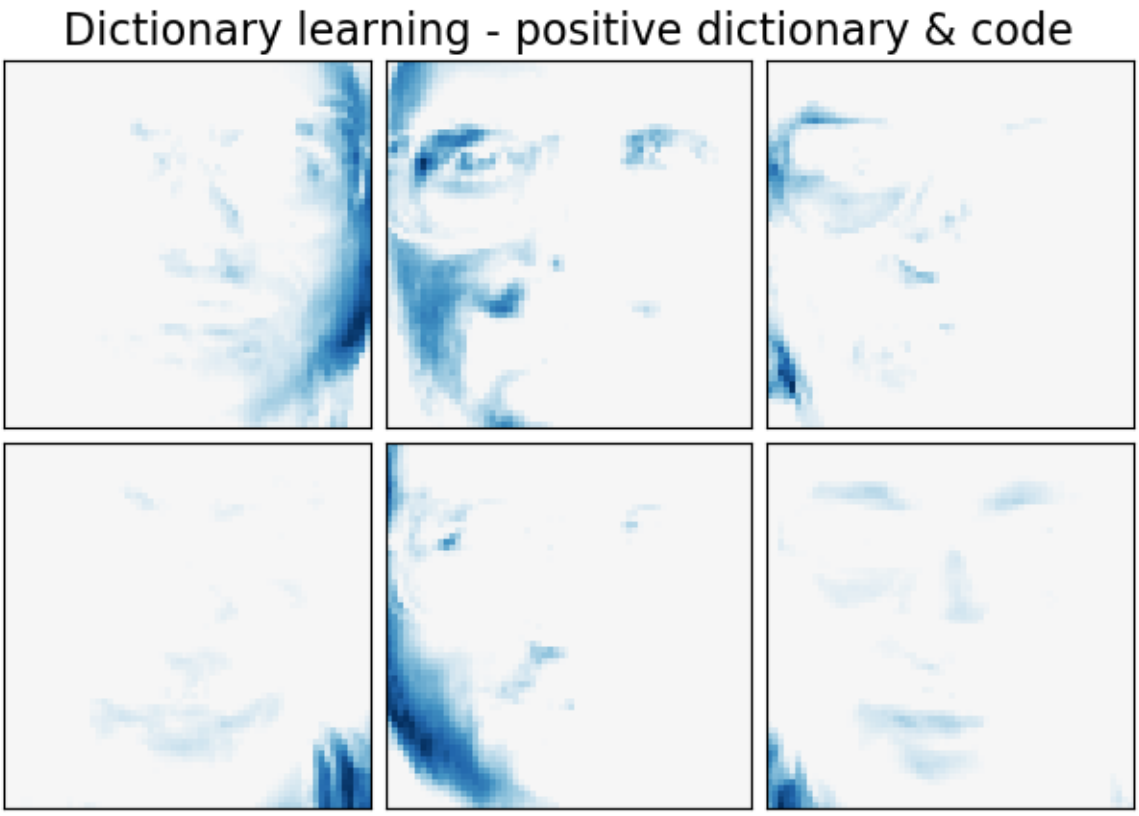

Factor Analysis components - FA - Train time 0.2s


First centered Olivetti faces


Dictionary learning


Dictionary learning - positive dictionary


Dictionary learning - positive code



-

Out:

```
Dataset consists of 400 faces
Extracting the top 6 Eigenfaces - PCA using randomized SVD...
done in 0.035s
Extracting the top 6 Non-negative components - NMF...
done in 0.166s
Extracting the top 6 Independent components - FastICA...
done in 0.143s
Extracting the top 6 Sparse comp. - MiniBatchSparsePCA...
done in 0.788s
Extracting the top 6 MiniBatchDictionaryLearning...
done in 0.466s
Extracting the top 6 Cluster centers - MiniBatchKMeans...
done in 0.158s
Extracting the top 6 Factor Analysis components - FA...
done in 0.177s
Extracting the top 6 Dictionary learning...
done in 0.588s
Extracting the top 6 Dictionary learning - positive dictionary...
done in 0.502s
Extracting the top 6 Dictionary learning - positive code...
done in 0.152s
Extracting the top 6 Dictionary learning - positive dictionary & code...
done in 0.230s
```

```
print(__doc__)

# Authors: Vlad Niculae, Alexandre Gramfort
# License: BSD 3 clause

import logging
from time import time

from numpy.random import RandomState
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_olivetti_faces
from sklearn.cluster import MiniBatchKMeans
from sklearn import decomposition

# Display progress logs on stdout
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s %(message)s')
n_row, n_col = 2, 3
n_components = n_row * n_col
image_shape = (64, 64)
rng = RandomState(0)

# ############################################################################
# Load faces data
faces, _ = fetch_olivetti_faces(return_X_y=True, shuffle=True,
                                random_state=rng)
n_samples, n_features = faces.shape

# global centering
faces_centered = faces - faces.mean(axis=0)

# local centering
faces_centered -= faces_centered.mean(axis=1).reshape(n_samples, -1)

print("Dataset consists of %d faces" % n_samples)


def plot_gallery(title, images, n_col=n_col, n_row=n_row, cmap=plt.cm.gray):
    plt.figure(figsize=(2. * n_col, 2.26 * n_row))
    plt.suptitle(title, size=16)
    for i, comp in enumerate(images):
        plt.subplot(n_row, n_col, i + 1)
        vmax = max(comp.max(), -comp.min())
        plt.imshow(comp.reshape(image_shape), cmap=cmap,
                   interpolation='nearest',
                   vmin=-vmax, vmax=vmax)
        plt.xticks(())
        plt.yticks(())
    plt.subplots_adjust(0.01, 0.05, 0.99, 0.93, 0.04, 0.)

# ############################################################################
# List of the different estimators, whether to center and transpose the
# problem, and whether the transformer uses the clustering API.
estimators = [
    ('Eigenfaces - PCA using randomized SVD',
     decomposition.PCA(n_components=n_components, svd_solver='randomized',
```

(continues on next page)

(continued from previous page)

```
                        whiten=True),
     True),

    ('Non-negative components - NMF',
     decomposition.NMF(n_components=n_components, init='nndsvda', tol=5e-3),
     False),

    ('Independent components - FastICA',
     decomposition.FastICA(n_components=n_components, whiten=True),
     True),

    ('Sparse comp. - MiniBatchSparsePCA',
     decomposition.MiniBatchSparsePCA(n_components=n_components, alpha=0.8,
                                      n_iter=100, batch_size=3,
                                      random_state=rng),
     True),

    ('MiniBatchDictionaryLearning',
         decomposition.MiniBatchDictionaryLearning(n_components=15, alpha=0.1,
                                                   n_iter=50, batch_size=3,
                                                   random_state=rng),
     True),

    ('Cluster centers - MiniBatchKMeans',
         MiniBatchKMeans(n_clusters=n_components, tol=1e-3, batch_size=20,
                         max_iter=50, random_state=rng),
     True),

    ('Factor Analysis components - FA',
     decomposition.FactorAnalysis(n_components=n_components, max_iter=20),
     True),
]


# ##############################################################################
# Plot a sample of the input data

plot_gallery("First centered Olivetti faces", faces_centered[:n_components])

# ##############################################################################
# Do the estimation and plot it

for name, estimator, center in estimators:
    print("Extracting the top %d %s..." % (n_components, name))
    t0 = time()
    data = faces
    if center:
        data = faces_centered
    estimator.fit(data)
    train_time = (time() - t0)
    print("done in %0.3fs" % train_time)
    if hasattr(estimator, 'cluster_centers_'):
        components_ = estimator.cluster_centers_
    else:
        components_ = estimator.components_

    # Plot an image representing the pixelwise variance provided by the
```

(continues on next page)

(continued from previous page)

```
    # estimator e.g its noise_variance_ attribute. The Eigenfaces estimator,
    # via the PCA decomposition, also provides a scalar noise_variance_
    # (the mean of pixelwise variance) that cannot be displayed as an image
    # so we skip it.
    if (hasattr(estimator, 'noise_variance_') and
            estimator.noise_variance_.ndim > 0):  # Skip the Eigenfaces case
        plot_gallery("Pixelwise variance",
                     estimator.noise_variance_.reshape(1, -1), n_col=1,
                     n_row=1)
    plot_gallery('%s - Train time %.1fs' % (name, train_time),
                 components_[:n_components])

plt.show()

# ############################################################################
# Various positivity constraints applied to dictionary learning.
estimators = [
    ('Dictionary learning',
        decomposition.MiniBatchDictionaryLearning(n_components=15, alpha=0.1,
                                                  n_iter=50, batch_size=3,
                                                  random_state=rng),
     True),
    ('Dictionary learning - positive dictionary',
        decomposition.MiniBatchDictionaryLearning(n_components=15, alpha=0.1,
                                                  n_iter=50, batch_size=3,
                                                  random_state=rng,
                                                  positive_dict=True),
     True),
    ('Dictionary learning - positive code',
        decomposition.MiniBatchDictionaryLearning(n_components=15, alpha=0.1,
                                                  n_iter=50, batch_size=3,
                                                  fit_algorithm='cd',
                                                  random_state=rng,
                                                  positive_code=True),
     True),
    ('Dictionary learning - positive dictionary & code',
        decomposition.MiniBatchDictionaryLearning(n_components=15, alpha=0.1,
                                                  n_iter=50, batch_size=3,
                                                  fit_algorithm='cd',
                                                  random_state=rng,
                                                  positive_dict=True,
                                                  positive_code=True),
     True),
]


# ############################################################################
# Plot a sample of the input data

plot_gallery("First centered Olivetti faces", faces_centered[:n_components],
             cmap=plt.cm.RdBu)

# ############################################################################
# Do the estimation and plot it

for name, estimator, center in estimators:
    print("Extracting the top %d %s..." % (n_components, name))
```

(continues on next page)

(continued from previous page)

```
    t0 = time()
    data = faces
    if center:
        data = faces_centered
    estimator.fit(data)
    train_time = (time() - t0)
    print("done in %0.3fs" % train_time)
    components_ = estimator.components_
    plot_gallery(name, components_[:n_components], cmap=plt.cm.RdBu)

plt.show()
```

Total running time of the script: (0 minutes 5.499 seconds)

Estimated memory usage: 43 MB

6.11 Ensemble methods

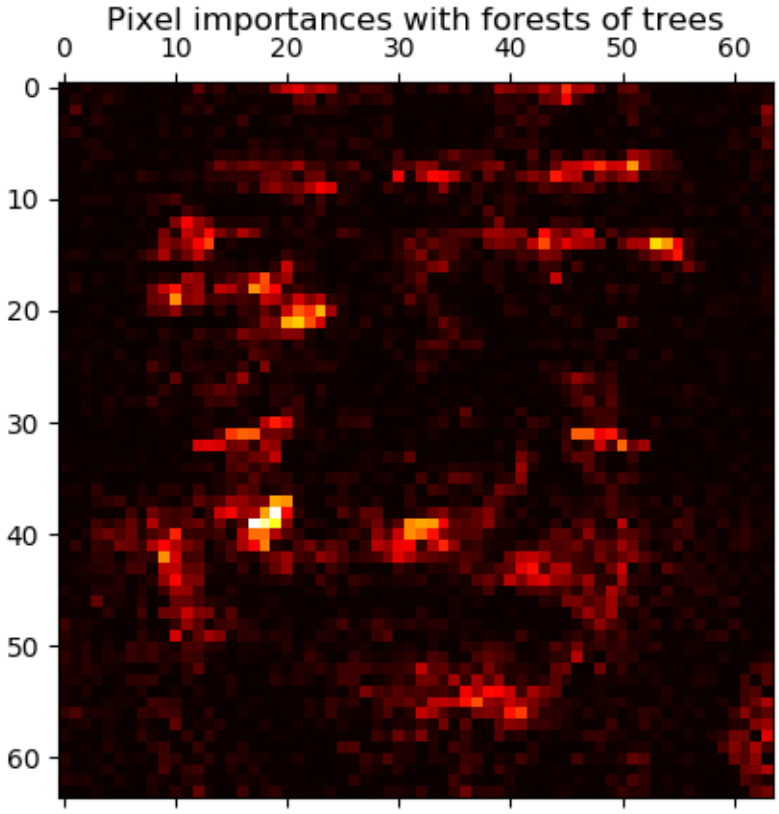
Examples concerning the *sklearn.ensemble* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.1 Pixel importances with a parallel forest of trees

This example shows the use of forests of trees to evaluate the importance of the pixels in an image classification task (faces). The hotter the pixel, the more important.

The code below also illustrates how the construction and the computation of the predictions can be parallelized within multiple jobs.



Out:

```
Fitting ExtraTreesClassifier on faces data with 1 cores...
done in 1.143s
```

```
print(__doc__)

from time import time
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_olivetti_faces
from sklearn.ensemble import ExtraTreesClassifier

# Number of cores to use to perform parallel fitting of the forest model
n_jobs = 1

# Load the faces dataset
data = fetch_olivetti_faces()
X, y = data.data, data.target
```

(continues on next page)

(continued from previous page)

```
mask = y < 5  # Limit to 5 classes
X = X[mask]
y = y[mask]

# Build a forest and compute the pixel importances
print("Fitting ExtraTreesClassifier on faces data with %d cores..." % n_jobs)
t0 = time()
forest = ExtraTreesClassifier(n_estimators=1000,
                              max_features=128,
                              n_jobs=n_jobs,
                              random_state=0)

forest.fit(X, y)
print("done in %0.3fs" % (time() - t0))
importances = forest.feature_importances_
importances = importances.reshape(data.images[0].shape)

# Plot pixel importances
plt.matshow(importances, cmap=plt.cm.hot)
plt.title("Pixel importances with forests of trees")
plt.show()
```

Total running time of the script: (0 minutes 1.517 seconds)

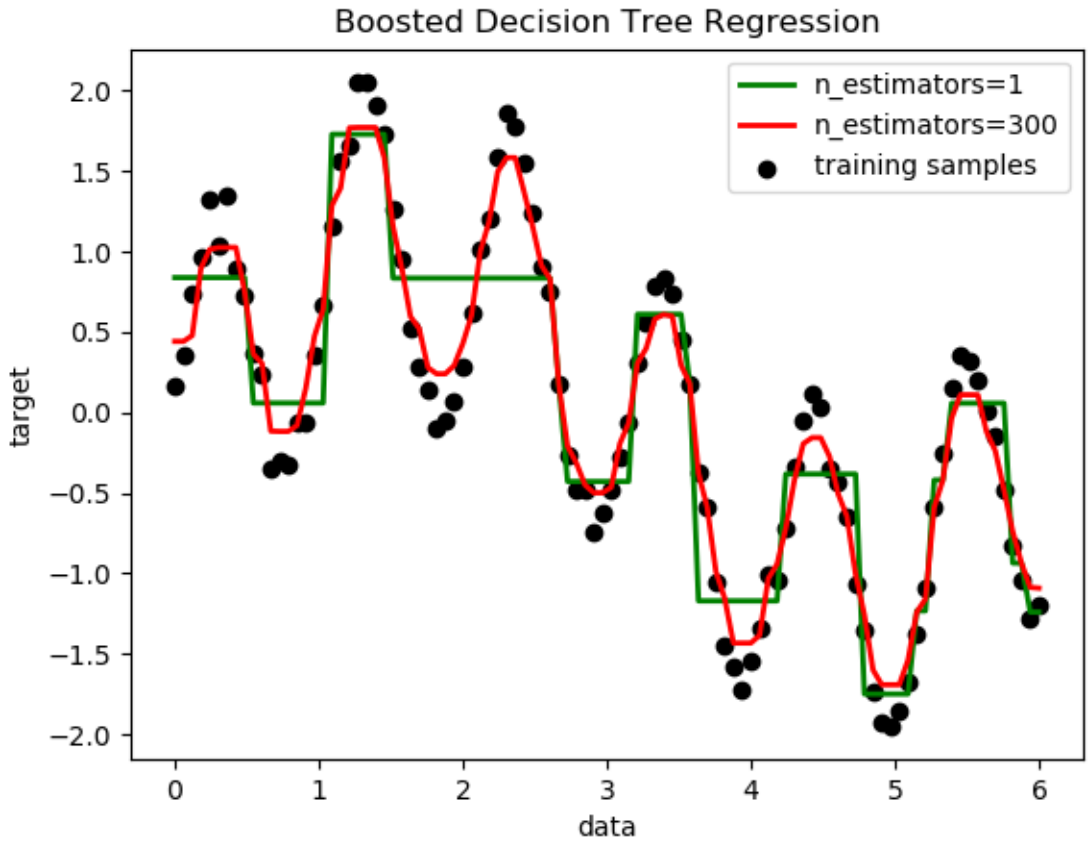
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.2 Decision Tree Regression with AdaBoost

A decision tree is boosted using the AdaBoost.R2[1] algorithm on a 1D sinusoidal dataset with a small amount of Gaussian noise. 299 boosts (300 decision trees) is compared with a single decision tree regressor. As the number of boosts is increased the regressor can fit more detail.

[1] H. Drucker, "Improving Regressors using Boosting Techniques", 1997.




```
print(__doc__)

# Author: Noel Dawe <noel.dawe@gmail.com>
#
# License: BSD 3 clause

# importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor

# Create the dataset
rng = np.random.RandomState(1)
X = np.linspace(0, 6, 100)[:, np.newaxis]
y = np.sin(X).ravel() + np.sin(6 * X).ravel() + rng.normal(0, 0.1, X.shape[0])

# Fit regression model
regr_1 = DecisionTreeRegressor(max_depth=4)

regr_2 = AdaBoostRegressor(DecisionTreeRegressor(max_depth=4),
                           n_estimators=300, random_state=rng)

regr_1.fit(X, y)
regr_2.fit(X, y)
```

(continues on next page)

(continued from previous page)

```
# Predict
y_1 = regr_1.predict(X)
y_2 = regr_2.predict(X)

# Plot the results
plt.figure()
plt.scatter(X, y, c="k", label="training samples")
plt.plot(X, y_1, c="g", label="n_estimators=1", linewidth=2)
plt.plot(X, y_2, c="r", label="n_estimators=300", linewidth=2)
plt.xlabel("data")
plt.ylabel("target")
plt.title("Boosted Decision Tree Regression")
plt.legend()
plt.show()
```

Total running time of the script: (0 minutes 0.517 seconds)

Estimated memory usage: 8 MB

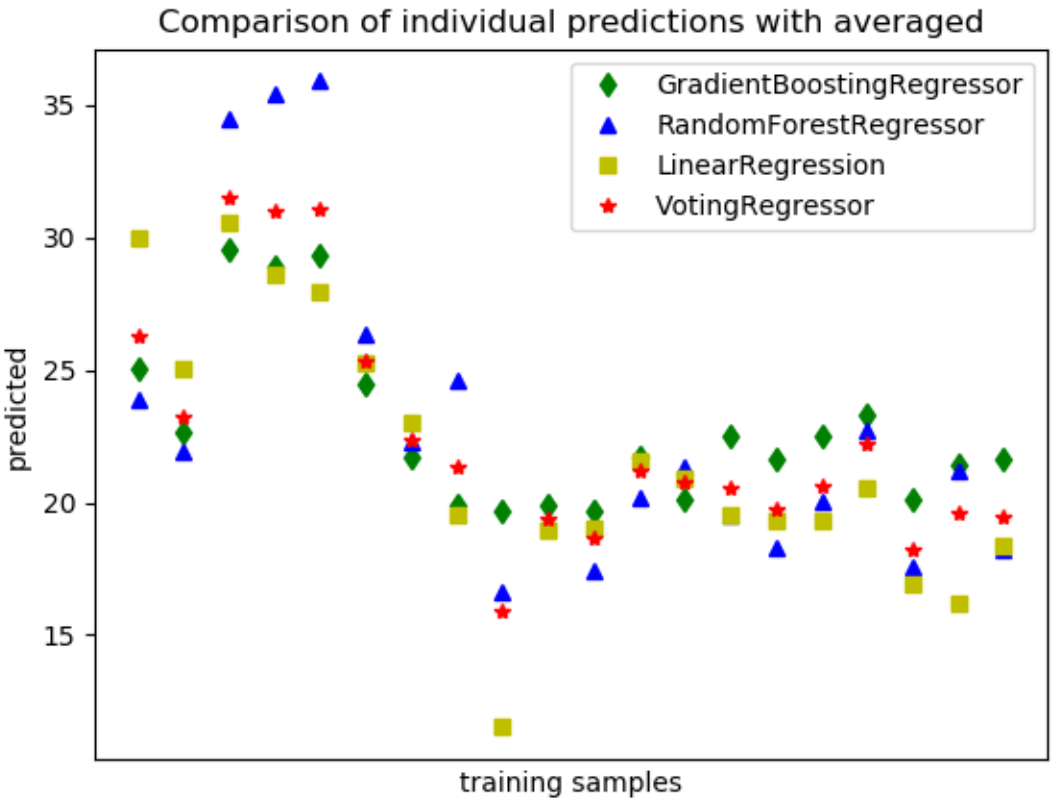
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.3 Plot individual and voting regression predictions

Plot individual and averaged regression predictions for Boston dataset.

First, three exemplary regressors are initialized (*GradientBoostingRegressor*, *RandomForestRegressor*, and *LinearRegression*) and used to initialize a *VotingRegressor*.

The red starred dots are the averaged predictions.




```
print(__doc__)

import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import VotingRegressor

# Loading some example data
X, y = datasets.load_boston(return_X_y=True)

# Training classifiers
reg1 = GradientBoostingRegressor(random_state=1, n_estimators=10)
reg2 = RandomForestRegressor(random_state=1, n_estimators=10)
reg3 = LinearRegression()
ereg = VotingRegressor([('gb', reg1), ('rf', reg2), ('lr', reg3)])
reg1.fit(X, y)
reg2.fit(X, y)
reg3.fit(X, y)
ereg.fit(X, y)

xt = X[:20]
```

(continues on next page)

(continued from previous page)

```
plt.figure()
plt.plot(reg1.predict(xt), 'gd', label='GradientBoostingRegressor')
plt.plot(reg2.predict(xt), 'b^', label='RandomForestRegressor')
plt.plot(reg3.predict(xt), 'ys', label='LinearRegression')
plt.plot(ereg.predict(xt), 'r*', label='VotingRegressor')
plt.tick_params(axis='x', which='both', bottom=False, top=False,
                labelbottom=False)
plt.ylabel('predicted')
plt.xlabel('training samples')
plt.legend(loc="best")
plt.title('Comparison of individual predictions with averaged')
plt.show()
```

Total running time of the script: (0 minutes 0.461 seconds)

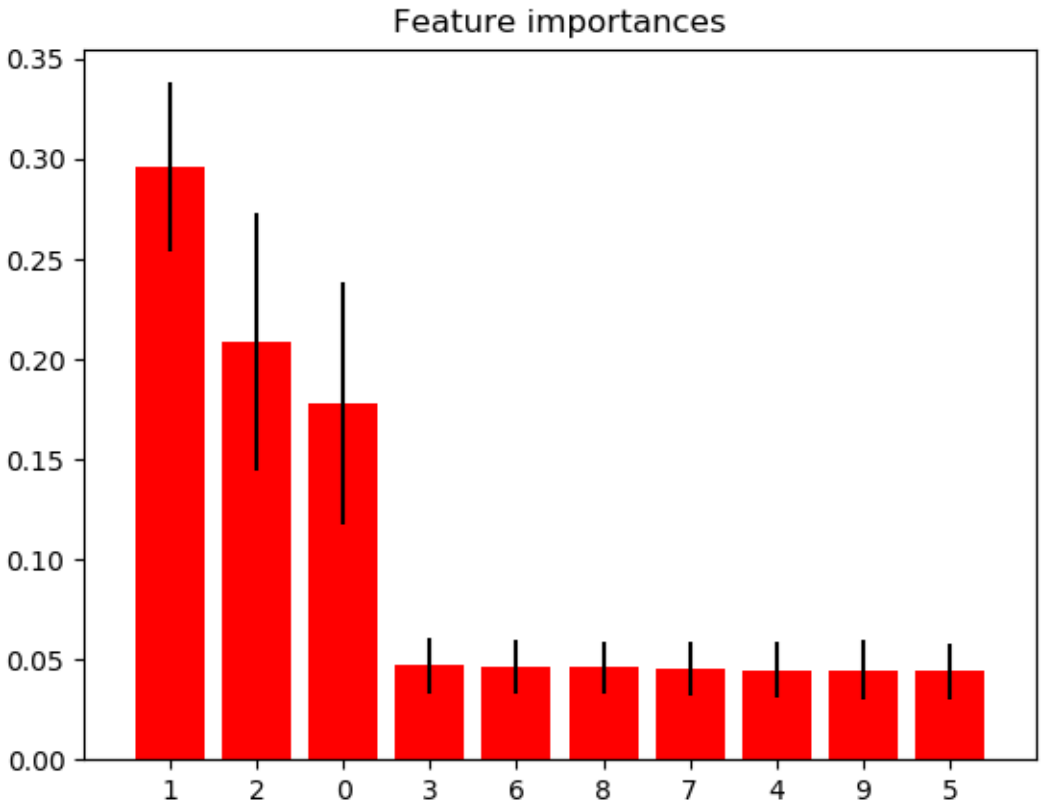
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.4 Feature importances with forests of trees

This examples shows the use of forests of trees to evaluate the importance of features on an artificial classification task. The red bars are the feature importances of the forest, along with their inter-trees variability.

As expected, the plot suggests that 3 features are informative, while the remaining are not.



Out:

```
Feature ranking:
1. feature 1 (0.295902)
2. feature 2 (0.208351)
3. feature 0 (0.177632)
4. feature 3 (0.047121)
5. feature 6 (0.046303)
6. feature 8 (0.046013)
7. feature 7 (0.045575)
8. feature 4 (0.044614)
9. feature 9 (0.044577)
10. feature 5 (0.043912)
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_classification
```

(continues on next page)

(continued from previous page)

```
from sklearn.ensemble import ExtraTreesClassifier

# Build a classification task using 3 informative features
X, y = make_classification(n_samples=1000,
                           n_features=10,
                           n_informative=3,
                           n_redundant=0,
                           n_repeated=0,
                           n_classes=2,
                           random_state=0,
                           shuffle=False)

# Build a forest and compute the feature importances
forest = ExtraTreesClassifier(n_estimators=250,
                              random_state=0)

forest.fit(X, y)
importances = forest.feature_importances_
std = np.std([tree.feature_importances_ for tree in forest.estimators_],
             axis=0)
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")

for f in range(X.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))

# Plot the feature importances of the forest
plt.figure()
plt.title("Feature importances")
plt.bar(range(X.shape[1]), importances[indices],
       color="r", yerr=std[indices], align="center")
plt.xticks(range(X.shape[1]), indices)
plt.xlim([-1, X.shape[1]])
plt.show()
```

Total running time of the script: (0 minutes 0.552 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.5 IsolationForest example

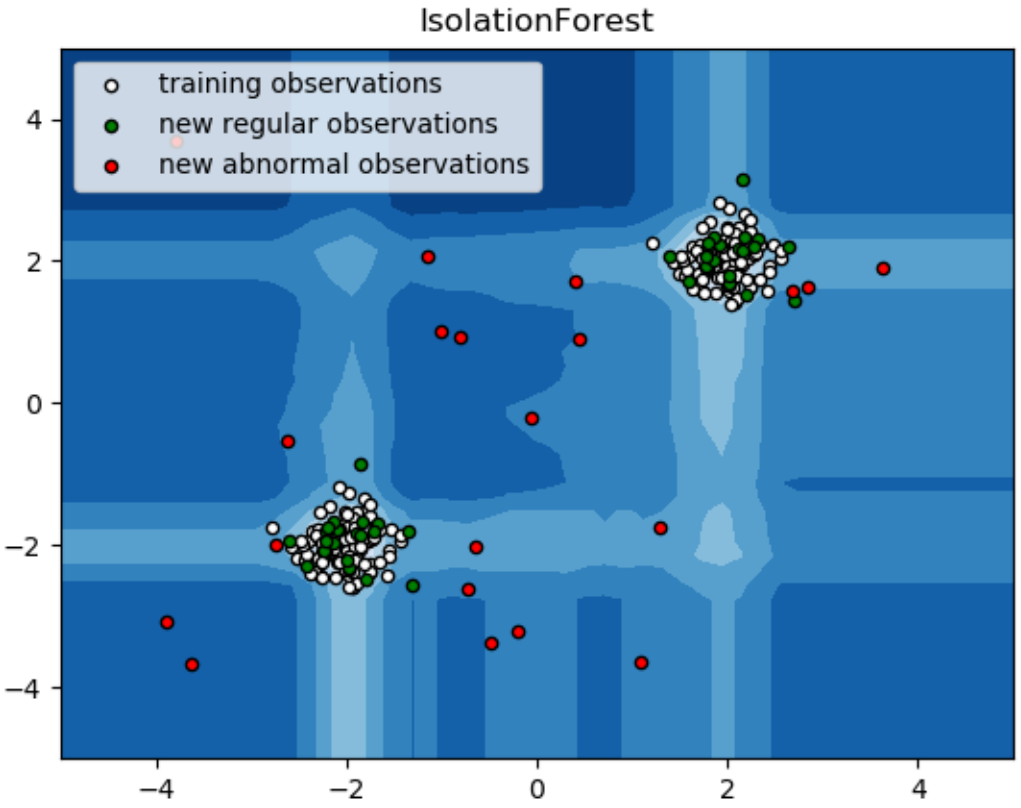
An example using *sklearn.ensemble.IsolationForest* for anomaly detection.

The IsolationForest 'isolates' observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature.

Since recursive partitioning can be represented by a tree structure, the number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node.

This path length, averaged over a forest of such random trees, is a measure of normality and our decision function.

Random partitioning produces noticeable shorter paths for anomalies. Hence, when a forest of random trees collectively produce shorter path lengths for particular samples, they are highly likely to be anomalies.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import IsolationForest

rng = np.random.RandomState(42)

# Generate train data
X = 0.3 * rng.randn(100, 2)
X_train = np.r_[X + 2, X - 2]
# Generate some regular novel observations
X = 0.3 * rng.randn(20, 2)
X_test = np.r_[X + 2, X - 2]
# Generate some abnormal novel observations
X_outliers = rng.uniform(low=-4, high=4, size=(20, 2))

# fit the model
clf = IsolationForest(max_samples=100, random_state=rng)
clf.fit(X_train)
y_pred_train = clf.predict(X_train)
y_pred_test = clf.predict(X_test)
y_pred_outliers = clf.predict(X_outliers)
```

(continues on next page)

(continued from previous page)

```
# plot the line, the samples, and the nearest vectors to the plane
xx, yy = np.meshgrid(np.linspace(-5, 5, 50), np.linspace(-5, 5, 50))
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.title("IsolationForest")
plt.contourf(xx, yy, Z, cmap=plt.cm.Blues_r)

b1 = plt.scatter(X_train[:, 0], X_train[:, 1], c='white',
                 s=20, edgecolor='k')
b2 = plt.scatter(X_test[:, 0], X_test[:, 1], c='green',
                 s=20, edgecolor='k')
c = plt.scatter(X_outliers[:, 0], X_outliers[:, 1], c='red',
                s=20, edgecolor='k')
plt.axis('tight')
plt.xlim((-5, 5))
plt.ylim((-5, 5))
plt.legend([b1, b2, c],
           ["training observations",
            "new regular observations", "new abnormal observations"],
           loc="upper left")
plt.show()
```

Total running time of the script: (0 minutes 0.534 seconds)

Estimated memory usage: 8 MB

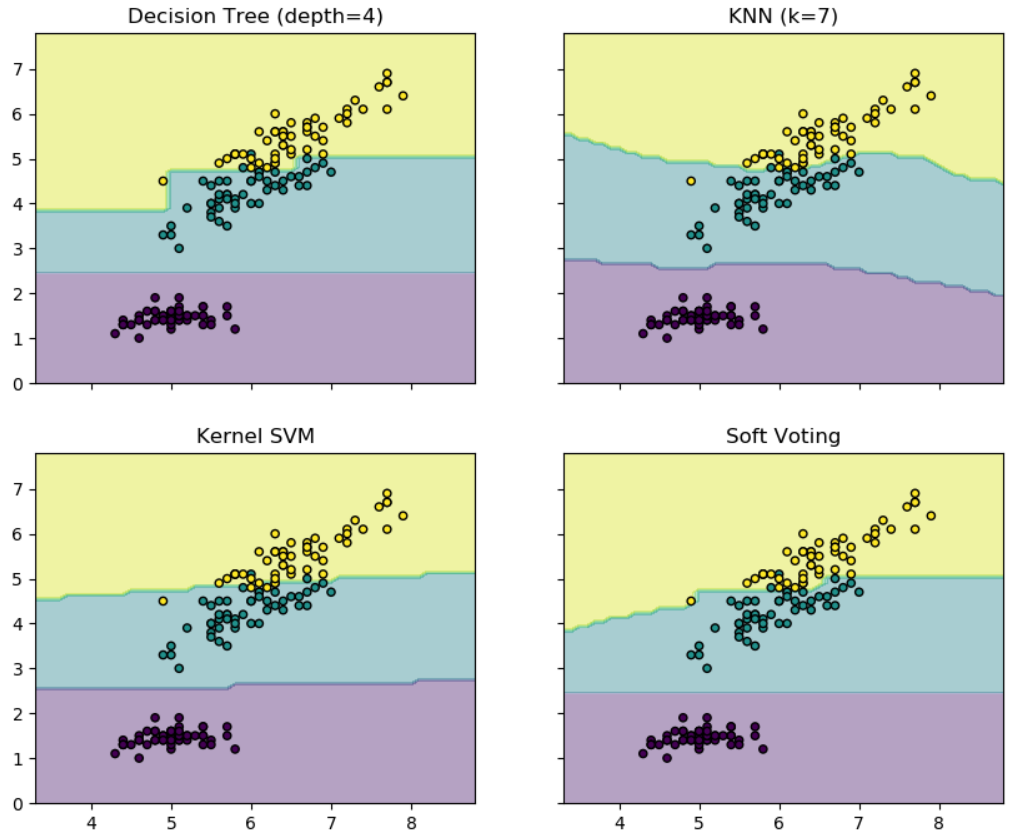
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.6 Plot the decision boundaries of a VotingClassifier

Plot the decision boundaries of a *VotingClassifier* for two features of the Iris dataset.

Plot the class probabilities of the first sample in a toy dataset predicted by three different classifiers and averaged by the *VotingClassifier*.

First, three exemplary classifiers are initialized (*DecisionTreeClassifier*, *KNeighborsClassifier*, and *SVC*) and used to initialize a soft-voting *VotingClassifier* with weights `[2, 1, 2]`, which means that the predicted probabilities of the *DecisionTreeClassifier* and *SVC* each count 2 times as much as the weights of the *KNeighborsClassifier* classifier when the averaged probability is calculated.




```
print(__doc__)

from itertools import product

import numpy as np
import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import VotingClassifier

# Loading some example data
iris = datasets.load_iris()
X = iris.data[:, [0, 2]]
y = iris.target

# Training classifiers
clf1 = DecisionTreeClassifier(max_depth=4)
clf2 = KNeighborsClassifier(n_neighbors=7)
clf3 = SVC(gamma=.1, kernel='rbf', probability=True)
eclf = VotingClassifier(estimators=[('dt', clf1), ('knn', clf2),
```

(continues on next page)

(continued from previous page)

```
                                    ('svc', clf3)],
                        voting='soft', weights=[2, 1, 2])

clf1.fit(X, y)
clf2.fit(X, y)
clf3.fit(X, y)
eclf.fit(X, y)

# Plotting decision regions
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

f, axarr = plt.subplots(2, 2, sharex='col', sharey='row', figsize=(10, 8))

for idx, clf, tt in zip(product([0, 1], [0, 1]),
                        [clf1, clf2, clf3, eclf],
                        ['Decision Tree (depth=4)', 'KNN (k=7)',
                         'Kernel SVM', 'Soft Voting']):

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.4)
    axarr[idx[0], idx[1]].scatter(X[:, 0], X[:, 1], c=y,
                                  s=20, edgecolor='k')
    axarr[idx[0], idx[1]].set_title(tt)

plt.show()
```

Total running time of the script: (0 minutes 0.611 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

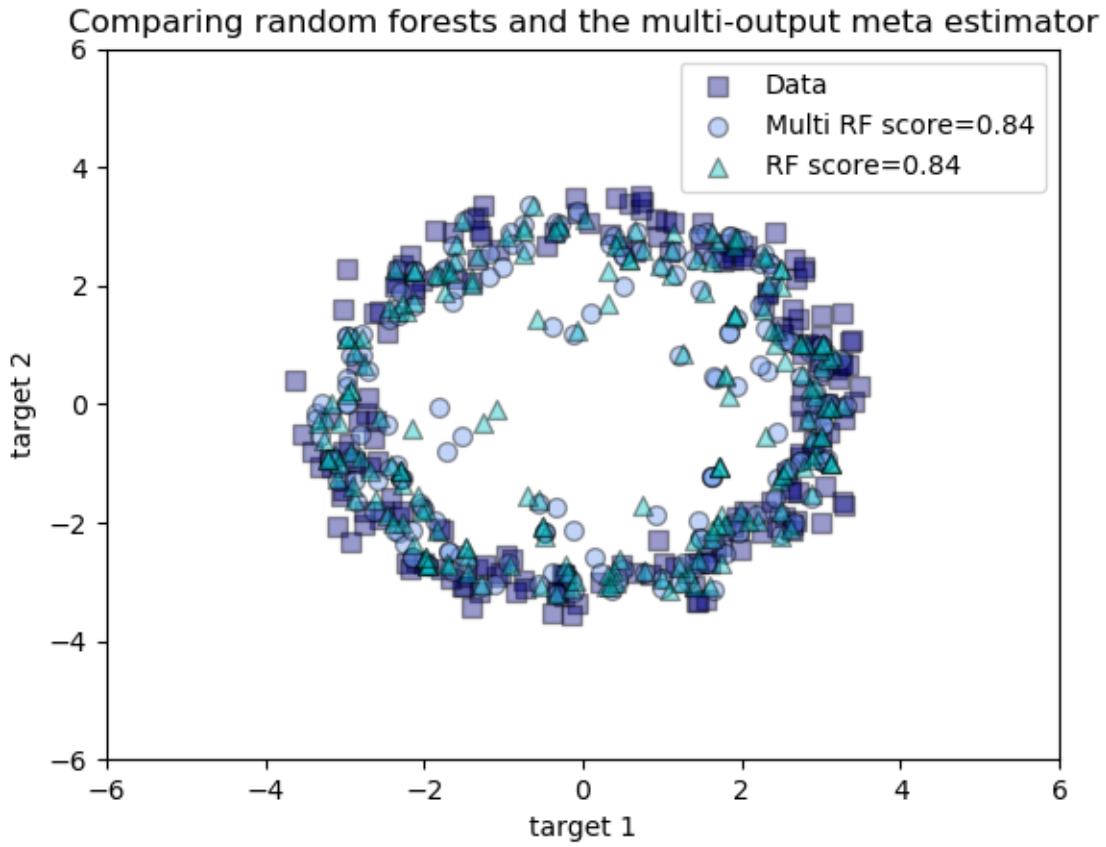
6.11.7 Comparing random forests and the multi-output meta estimator

An example to compare multi-output regression with random forest and the *multioutput.MultiOutputRegressor* meta-estimator.

This example illustrates the use of the *multioutput.MultiOutputRegressor* meta-estimator to perform multi-output regression. A random forest regressor is used, which supports multi-output regression natively, so the results can be compared.

The random forest regressor will only ever predict values within the range of observations or closer to zero for each of the targets. As a result the predictions are biased towards the centre of the circle.

Using a single underlying feature the model learns both the x and y coordinate as output.



Out:

```
/home/circleci/project/sklearn/base.py:426: FutureWarning: The default value of␣
↪multioutput (not exposed in score method) will change from 'variance_weighted' to
↪'uniform_average' in 0.23 to keep consistent with 'metrics.r2_score'. To specify␣
↪the default value manually and avoid the warning, please either call 'metrics.r2_
↪score' directly or make a custom scorer with 'metrics.make_scorer' (the built-in␣
↪scorer 'r2' uses multioutput='uniform_average').
  warnings.warn("The default value of multioutput (not exposed in "
```

```
print(__doc__)

# Author: Tim Head <betatim@gmail.com>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
```

(continues on next page)

(continued from previous page)

```
from sklearn.multioutput import MultiOutputRegressor


# Create a random dataset
rng = np.random.RandomState(1)
X = np.sort(200 * rng.rand(600, 1) - 100, axis=0)
y = np.array([np.pi * np.sin(X).ravel(), np.pi * np.cos(X).ravel()]).T
y += (0.5 - rng.rand(*y.shape))

X_train, X_test, y_train, y_test = train_test_split(
    X, y, train_size=400, test_size=200, random_state=4)

max_depth = 30
regr_multirf = MultiOutputRegressor(RandomForestRegressor(n_estimators=100,
                                                          max_depth=max_depth,
                                                          random_state=0))
regr_multirf.fit(X_train, y_train)

regr_rf = RandomForestRegressor(n_estimators=100, max_depth=max_depth,
                                random_state=2)
regr_rf.fit(X_train, y_train)

# Predict on new data
y_multirf = regr_multirf.predict(X_test)
y_rf = regr_rf.predict(X_test)

# Plot the results
plt.figure()
s = 50
a = 0.4
plt.scatter(y_test[:, 0], y_test[:, 1], edgecolor='k',
            c="navy", s=s, marker="s", alpha=a, label="Data")
plt.scatter(y_multirf[:, 0], y_multirf[:, 1], edgecolor='k',
            c="cornflowerblue", s=s, alpha=a,
            label="Multi RF score=%.2f" % regr_multirf.score(X_test, y_test))
plt.scatter(y_rf[:, 0], y_rf[:, 1], edgecolor='k',
            c="c", s=s, marker="^", alpha=a,
            label="RF score=%.2f" % regr_rf.score(X_test, y_test))
plt.xlim([-6, 6])
plt.ylim([-6, 6])
plt.xlabel("target 1")
plt.ylabel("target 2")
plt.title("Comparing random forests and the multi-output meta estimator")
plt.legend()
plt.show()
```

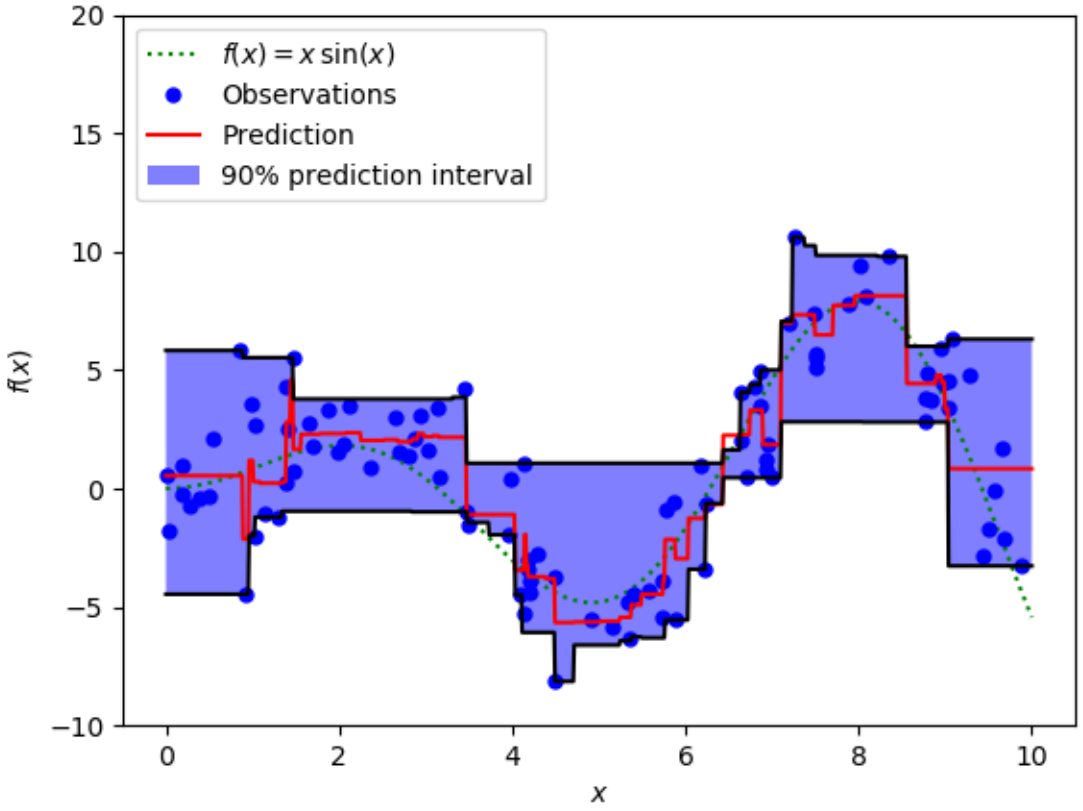
Total running time of the script: (0 minutes 0.608 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.8 Prediction Intervals for Gradient Boosting Regression

This example shows how quantile regression can be used to create prediction intervals.




```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.ensemble import GradientBoostingRegressor

np.random.seed(1)


def f(x):
    """The function to predict."""
    return x * np.sin(x)

#----------------------------------------------------------------------
#  First the noiseless case
X = np.atleast_2d(np.random.uniform(0, 10.0, size=100)).T
X = X.astype(np.float32)

# Observations
y = f(X).ravel()

dy = 1.5 + 1.0 * np.random.random(y.shape)
noise = np.random.normal(0, dy)
y += noise
y = y.astype(np.float32)
```

(continues on next page)

(continued from previous page)

```
# Mesh the input space for evaluations of the real function, the prediction and
# its MSE
xx = np.atleast_2d(np.linspace(0, 10, 1000)).T
xx = xx.astype(np.float32)

alpha = 0.95

clf = GradientBoostingRegressor(loss='quantile', alpha=alpha,
                                n_estimators=250, max_depth=3,
                                learning_rate=.1, min_samples_leaf=9,
                                min_samples_split=9)

clf.fit(X, y)

# Make the prediction on the meshed x-axis
y_upper = clf.predict(xx)

clf.set_params(alpha=1.0 - alpha)
clf.fit(X, y)

# Make the prediction on the meshed x-axis
y_lower = clf.predict(xx)

clf.set_params(loss='ls')
clf.fit(X, y)

# Make the prediction on the meshed x-axis
y_pred = clf.predict(xx)

# Plot the function, the prediction and the 90% confidence interval based on
# the MSE
fig = plt.figure()
plt.plot(xx, f(xx), 'g:', label=r'$f(x) = x\,\sin(x)$')
plt.plot(X, y, 'b.', markersize=10, label=u'Observations')
plt.plot(xx, y_pred, 'r-', label=u'Prediction')
plt.plot(xx, y_upper, 'k-')
plt.plot(xx, y_lower, 'k-')
plt.fill(np.concatenate([xx, xx[::-1]]),
         np.concatenate([y_upper, y_lower[::-1]]),
         alpha=.5, fc='b', ec='None', label='90% prediction interval')
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
plt.ylim(-10, 20)
plt.legend(loc='upper left')
plt.show()
```

Total running time of the script: (0 minutes 0.666 seconds)

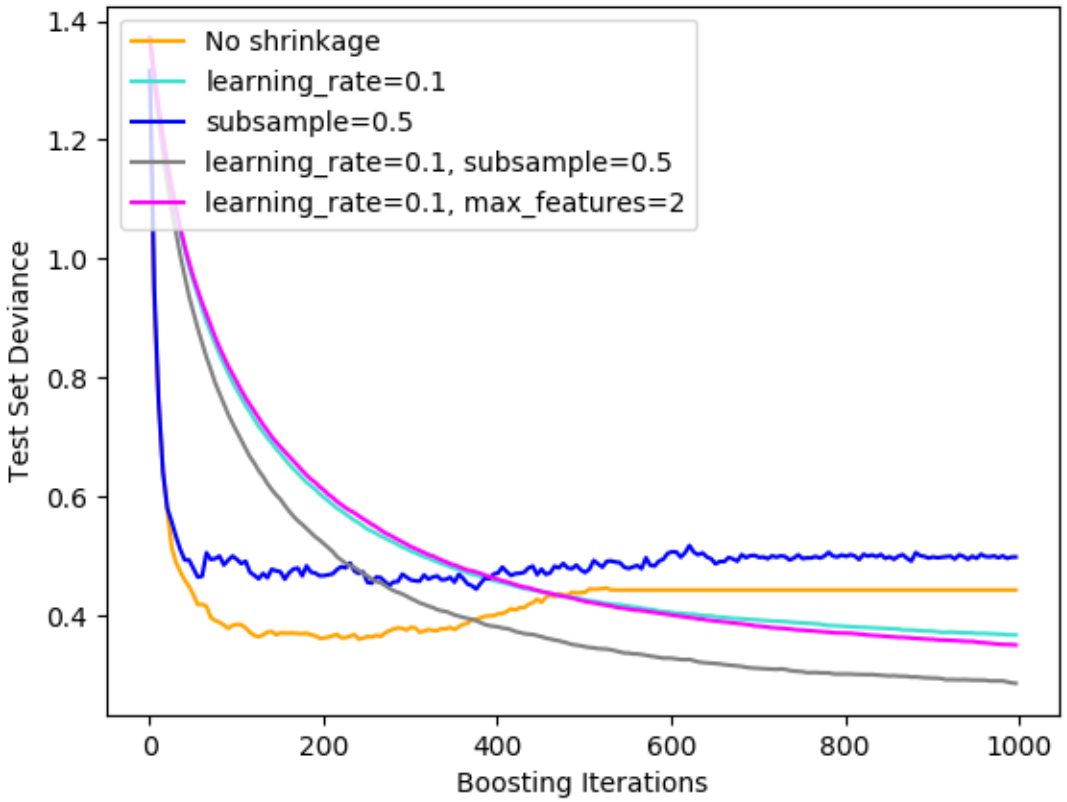
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.9 Gradient Boosting regularization

Illustration of the effect of different regularization strategies for Gradient Boosting. The example is taken from Hastie et al 2009[1].

The loss function used is binomial deviance. Regularization via shrinkage (`learning_rate < 1.0`) improves performance considerably. In combination with shrinkage, stochastic gradient boosting (`subsample < 1.0`) can produce more accurate models by reducing the variance via bagging. Subsampling without shrinkage usually does poorly. Another strategy to reduce the variance is by subsampling the features analogous to the random splits in Random Forests (via the `max_features` parameter).




```
print(__doc__)

# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn import ensemble
from sklearn import datasets
```

(continues on next page)

[1] T. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning Ed. 2", Springer, 2009.

(continued from previous page)

```
X, y = datasets.make_hastie_10_2(n_samples=12000, random_state=1)
X = X.astype(np.float32)

# map labels from {-1, 1} to {0, 1}
labels, y = np.unique(y, return_inverse=True)

X_train, X_test = X[:2000], X[2000:]
y_train, y_test = y[:2000], y[2000:]

original_params = {'n_estimators': 1000, 'max_leaf_nodes': 4, 'max_depth': None,
↪'random_state': 2,
                   'min_samples_split': 5}

plt.figure()

for label, color, setting in [('No shrinkage', 'orange',
                               {'learning_rate': 1.0, 'subsample': 1.0}),
                              ('learning_rate=0.1', 'turquoise',
                               {'learning_rate': 0.1, 'subsample': 1.0}),
                              ('subsample=0.5', 'blue',
                               {'learning_rate': 1.0, 'subsample': 0.5}),
                              ('learning_rate=0.1, subsample=0.5', 'gray',
                               {'learning_rate': 0.1, 'subsample': 0.5}),
                              ('learning_rate=0.1, max_features=2', 'magenta',
                               {'learning_rate': 0.1, 'max_features': 2})]:
    params = dict(original_params)
    params.update(setting)

    clf = ensemble.GradientBoostingClassifier(**params)
    clf.fit(X_train, y_train)

    # compute test set deviance
    test_deviance = np.zeros((params['n_estimators'],), dtype=np.float64)

    for i, y_pred in enumerate(clf.staged_decision_function(X_test)):
        # clf.loss_ assumes that y_test[i] in {0, 1}
        test_deviance[i] = clf.loss_(y_test, y_pred)

    plt.plot((np.arange(test_deviance.shape[0]) + 1)[::5], test_deviance[::5],
             '-', color=color, label=label)

plt.legend(loc='upper left')
plt.xlabel('Boosting Iterations')
plt.ylabel('Test Set Deviance')

plt.show()
```

Total running time of the script: (0 minutes 19.492 seconds)

Estimated memory usage: 8 MB

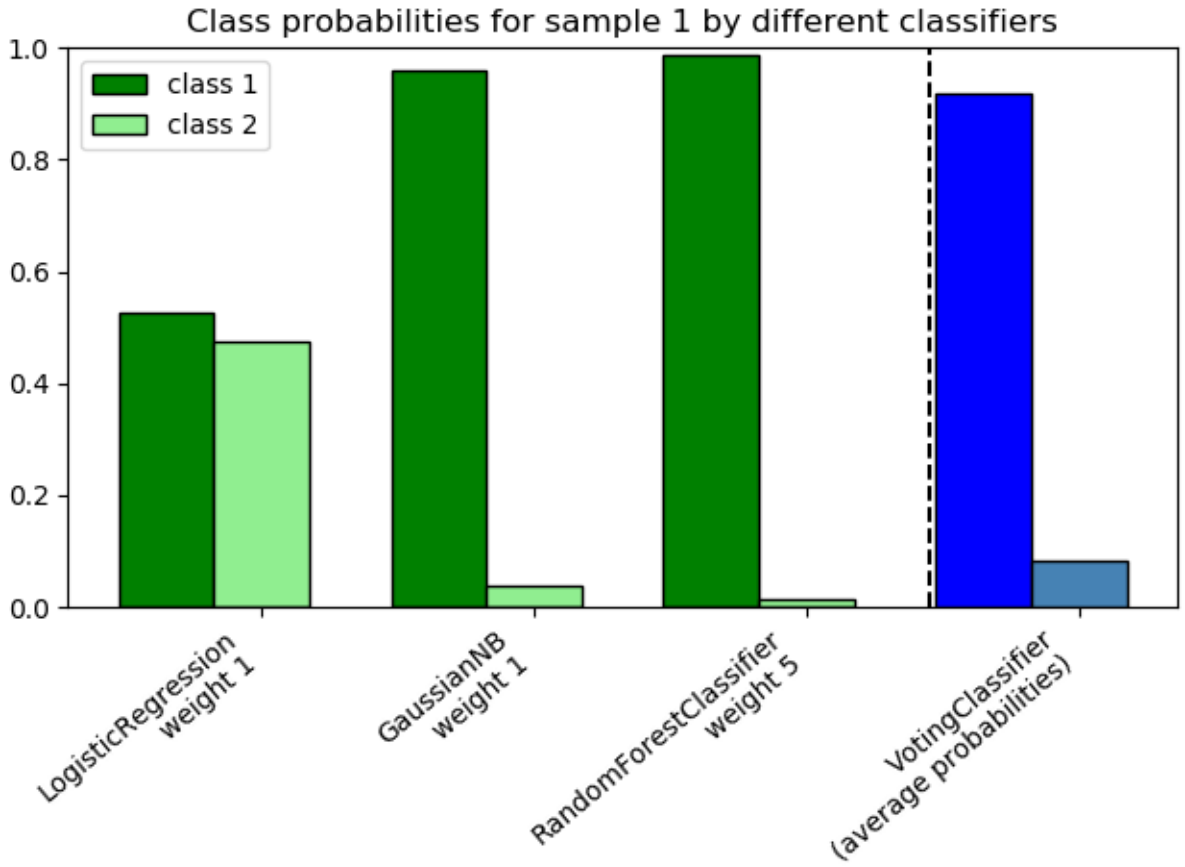
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.10 Plot class probabilities calculated by the VotingClassifier

Plot the class probabilities of the first sample in a toy dataset predicted by three different classifiers and averaged by the *VotingClassifier*.

First, three examplary classifiers are initialized (*LogisticRegression*, *GaussianNB*, and *RandomForestClassifier*) and used to initialize a soft-voting *VotingClassifier* with weights `[1, 1, 5]`, which means that the predicted probabilities of the *RandomForestClassifier* count 5 times as much as the weights of the other classifiers when the averaged probability is calculated.

To visualize the probability weighting, we fit each classifier on the training set and plot the predicted class probabilities for the first sample in this example dataset.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier

clf1 = LogisticRegression(max_iter=1000, random_state=123)
clf2 = RandomForestClassifier(n_estimators=100, random_state=123)
clf3 = GaussianNB()
```

(continues on next page)

(continued from previous page)

```
X = np.array([[-1.0, -1.0], [-1.2, -1.4], [-3.4, -2.2], [1.1, 1.2]])
y = np.array([1, 1, 2, 2])

eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],
                        voting='soft',
                        weights=[1, 1, 5])

# predict class probabilities for all classifiers
probas = [c.fit(X, y).predict_proba(X) for c in (clf1, clf2, clf3, eclf)]

# get class probabilities for the first sample in the dataset
class1_1 = [pr[0, 0] for pr in probas]
class2_1 = [pr[0, 1] for pr in probas]


# plotting

N = 4  # number of groups
ind = np.arange(N)  # group positions
width = 0.35  # bar width

fig, ax = plt.subplots()

# bars for classifier 1-3
p1 = ax.bar(ind, np.hstack(([class1_1[:-1], [0]])), width,
            color='green', edgecolor='k')
p2 = ax.bar(ind + width, np.hstack(([class2_1[:-1], [0]])), width,
            color='lightgreen', edgecolor='k')

# bars for VotingClassifier
p3 = ax.bar(ind, [0, 0, 0, class1_1[-1]], width,
            color='blue', edgecolor='k')
p4 = ax.bar(ind + width, [0, 0, 0, class2_1[-1]], width,
            color='steelblue', edgecolor='k')

# plot annotations
plt.axvline(2.8, color='k', linestyle='dashed')
ax.set_xticks(ind + width)
ax.set_xticklabels(['LogisticRegression\nweight 1',
                    'GaussianNB\nweight 1',
                    'RandomForestClassifier\nweight 5',
                    'VotingClassifier\n(average probabilities)'],
                   rotation=40,
                   ha='right')
plt.ylim([0, 1])
plt.title('Class probabilities for sample 1 by different classifiers')
plt.legend([p1[0], p2[0]], ['class 1', 'class 2'], loc='upper left')
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.456 seconds)

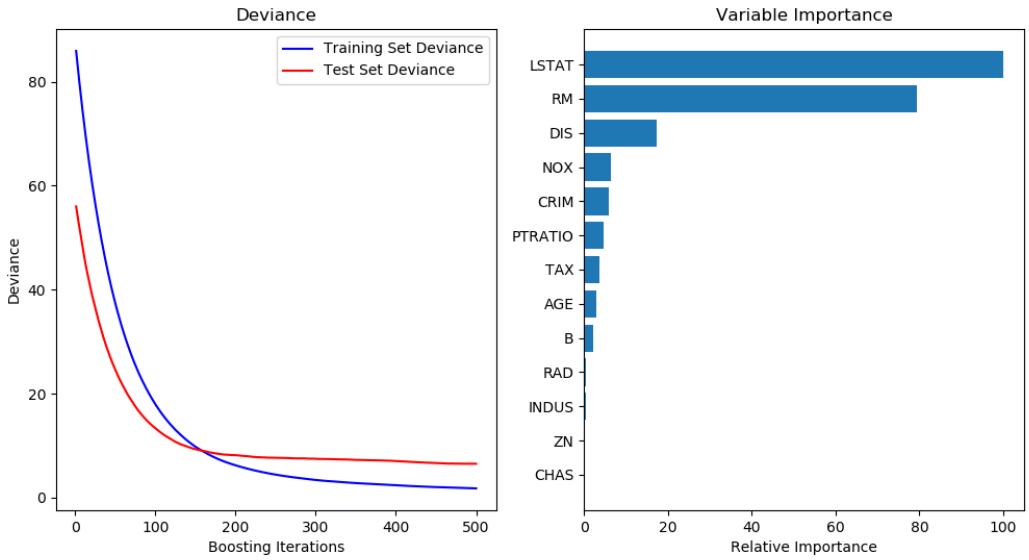
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.11 Gradient Boosting regression

Demonstrate Gradient Boosting on the Boston housing dataset.

This example fits a Gradient Boosting model with least squares loss and 500 regression trees of depth 4.



Out:

```
MSE: 6.4961
```

```
print(__doc__)

# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn import ensemble
from sklearn import datasets
from sklearn.utils import shuffle
from sklearn.metrics import mean_squared_error

# #############################################################################
# Load data
boston = datasets.load_boston()
X, y = shuffle(boston.data, boston.target, random_state=13)
X = X.astype(np.float32)
offset = int(X.shape[0] * 0.9)
X_train, y_train = X[:offset], y[:offset]
```

(continues on next page)

(continued from previous page)

```
X_test, y_test = X[offset:], y[offset:]

# #############################################################################
# Fit regression model
params = {'n_estimators': 500, 'max_depth': 4, 'min_samples_split': 2,
          'learning_rate': 0.01, 'loss': 'ls'}
clf = ensemble.GradientBoostingRegressor(**params)

clf.fit(X_train, y_train)
mse = mean_squared_error(y_test, clf.predict(X_test))
print("MSE: %.4f" % mse)

# #############################################################################
# Plot training deviance

# compute test set deviance
test_score = np.zeros((params['n_estimators'],), dtype=np.float64)

for i, y_pred in enumerate(clf.staged_predict(X_test)):
    test_score[i] = clf.loss_(y_test, y_pred)

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title('Deviance')
plt.plot(np.arange(params['n_estimators']) + 1, clf.train_score_, 'b-',
         label='Training Set Deviance')
plt.plot(np.arange(params['n_estimators']) + 1, test_score, 'r-',
         label='Test Set Deviance')
plt.legend(loc='upper right')
plt.xlabel('Boosting Iterations')
plt.ylabel('Deviance')

# #############################################################################
# Plot feature importance
feature_importance = clf.feature_importances_
# make importances relative to max importance
feature_importance = 100.0 * (feature_importance / feature_importance.max())
sorted_idx = np.argsort(feature_importance)
pos = np.arange(sorted_idx.shape[0]) + .5
plt.subplot(1, 2, 2)
plt.barh(pos, feature_importance[sorted_idx], align='center')
plt.yticks(pos, boston.feature_names[sorted_idx])
plt.xlabel('Relative Importance')
plt.title('Variable Importance')
plt.show()
```

Total running time of the script: (0 minutes 0.998 seconds)

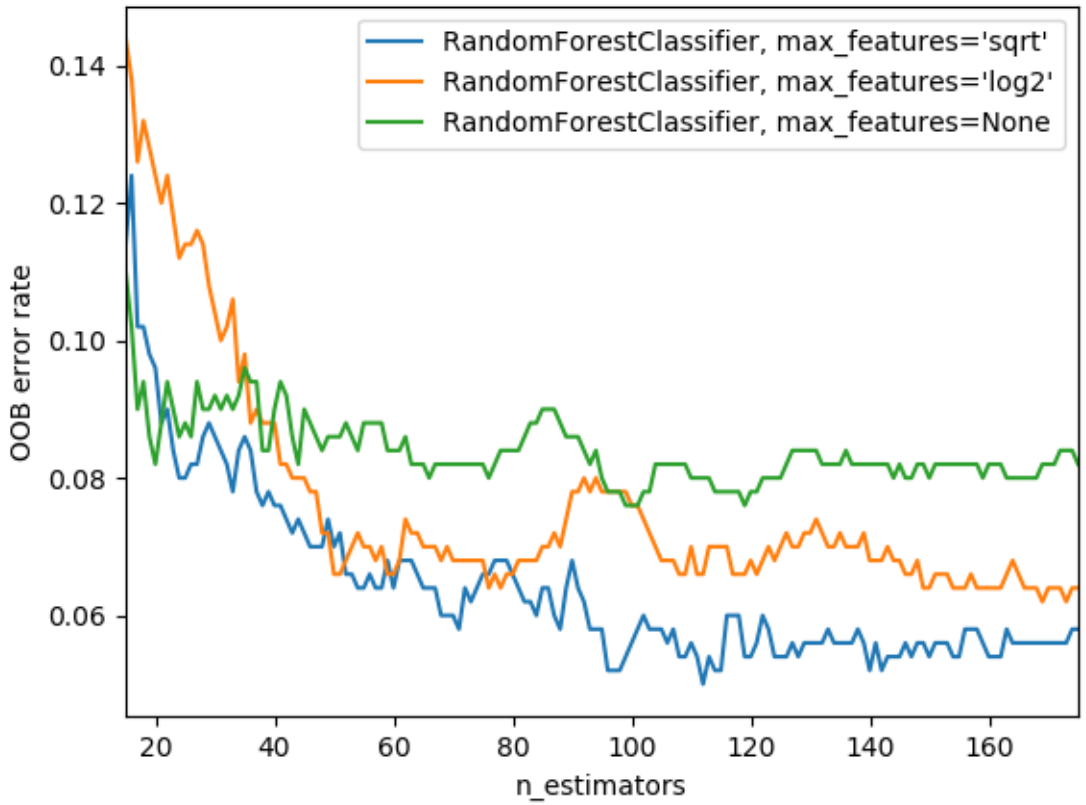
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.12 OOB Errors for Random Forests

The `RandomForestClassifier` is trained using *bootstrap aggregation*, where each new tree is fit from a bootstrap sample of the training observations $z_i = (x_i, y_i)$. The *out-of-bag* (OOB) error is the average error for each z_i calculated using predictions from the trees that do not contain z_i in their respective bootstrap sample. This allows the `RandomForestClassifier` to be fit and validated whilst being trained[1].

The example below demonstrates how the OOB error can be measured at the addition of each new tree during training. The resulting plot allows a practitioner to approximate a suitable value of `n_estimators` at which the error stabilizes.




```
import matplotlib.pyplot as plt

from collections import OrderedDict
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier

# Author: Kian Ho <hui.kian.ho@gmail.com>
#         Gilles Louppe <g.louppe@gmail.com>
#         Andreas Mueller <amueller@ais.uni-bonn.de>
#
# License: BSD 3 Clause

print(__doc__)
```

(continues on next page)

[1] T. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning Ed. 2", p592-593, Springer, 2009.

(continued from previous page)

```
RANDOM_STATE = 123

# Generate a binary classification dataset.
X, y = make_classification(n_samples=500, n_features=25,
                           n_clusters_per_class=1, n_informative=15,
                           random_state=RANDOM_STATE)

# NOTE: Setting the `warm_start` construction parameter to `True` disables
# support for parallelized ensembles but is necessary for tracking the OOB
# error trajectory during training.
ensemble_clfs = [
    ("RandomForestClassifier, max_features='sqrt'",
        RandomForestClassifier(warm_start=True, oob_score=True,
                               max_features="sqrt",
                               random_state=RANDOM_STATE)),
    ("RandomForestClassifier, max_features='log2'",
        RandomForestClassifier(warm_start=True, max_features='log2',
                               oob_score=True,
                               random_state=RANDOM_STATE)),
    ("RandomForestClassifier, max_features=None",
        RandomForestClassifier(warm_start=True, max_features=None,
                               oob_score=True,
                               random_state=RANDOM_STATE))
]

# Map a classifier name to a list of (<n_estimators>, <error rate>) pairs.
error_rate = OrderedDict((label, []) for label, _ in ensemble_clfs)

# Range of `n_estimators` values to explore.
min_estimators = 15
max_estimators = 175

for label, clf in ensemble_clfs:
    for i in range(min_estimators, max_estimators + 1):
        clf.set_params(n_estimators=i)
        clf.fit(X, y)

        # Record the OOB error for each `n_estimators=i` setting.
        oob_error = 1 - clf.oob_score_
        error_rate[label].append((i, oob_error))

# Generate the "OOB error rate" vs. "n_estimators" plot.
for label, clf_err in error_rate.items():
    xs, ys = zip(*clf_err)
    plt.plot(xs, ys, label=label)

plt.xlim(min_estimators, max_estimators)
plt.xlabel("n_estimators")
plt.ylabel("OOB error rate")
plt.legend(loc="upper right")
plt.show()
```

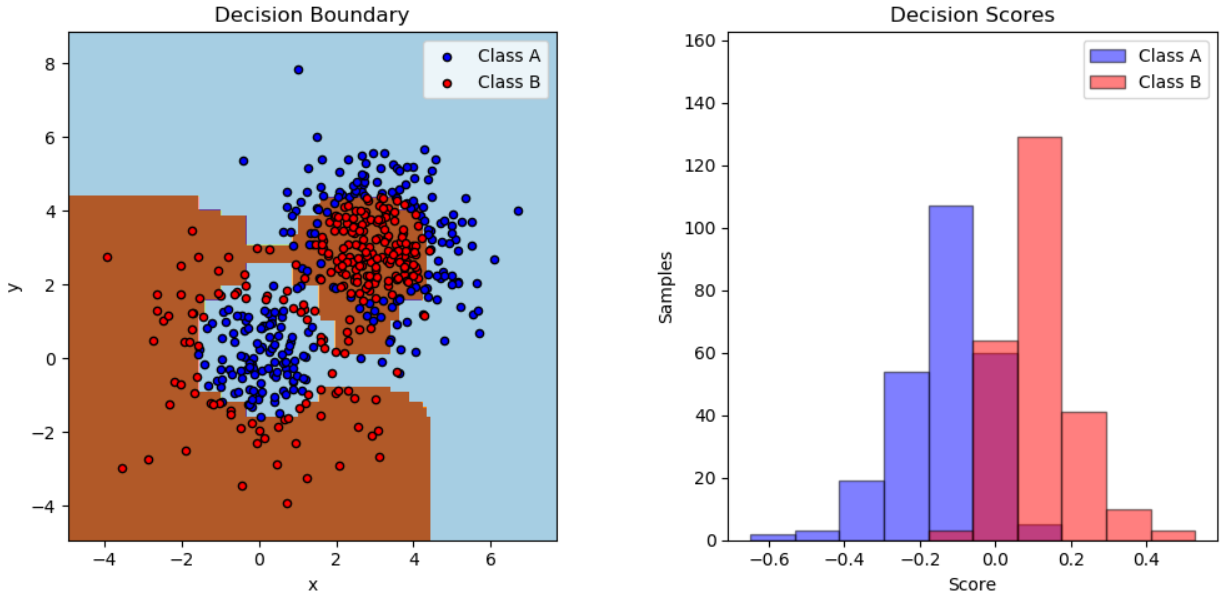
Total running time of the script: (0 minutes 15.248 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.13 Two-class AdaBoost

This example fits an AdaBoosted decision stump on a non-linearly separable classification dataset composed of two "Gaussian quantiles" clusters (see *sklearn.datasets.make_gaussian_quantiles*) and plots the decision boundary and decision scores. The distributions of decision scores are shown separately for samples of class A and B. The predicted class label for each sample is determined by the sign of the decision score. Samples with decision scores greater than zero are classified as B, and are otherwise classified as A. The magnitude of a decision score determines the degree of likeness with the predicted class label. Additionally, a new dataset could be constructed containing a desired purity of class B, for example, by only selecting samples with a decision score above some value.




```
print(__doc__)

# Author: Noel Dawe <noel.dawe@gmail.com>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_gaussian_quantiles


# Construct dataset
X1, y1 = make_gaussian_quantiles(cov=2.,
                                 n_samples=200, n_features=2,
                                 n_classes=2, random_state=1)
X2, y2 = make_gaussian_quantiles(mean=(3, 3), cov=1.5,
                                 n_samples=300, n_features=2,
```

(continues on next page)

(continued from previous page)

```
                                    n_classes=2, random_state=1)
X = np.concatenate((X1, X2))
y = np.concatenate((y1, - y2 + 1))

# Create and fit an AdaBoosted decision tree
bdt = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1),
                         algorithm="SAMME",
                         n_estimators=200)

bdt.fit(X, y)

plot_colors = "br"
plot_step = 0.02
class_names = "AB"

plt.figure(figsize=(10, 5))

# Plot the decision boundaries
plt.subplot(121)
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                     np.arange(y_min, y_max, plot_step))

Z = bdt.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
cs = plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
plt.axis("tight")

# Plot the training points
for i, n, c in zip(range(2), class_names, plot_colors):
    idx = np.where(y == i)
    plt.scatter(X[idx, 0], X[idx, 1],
                c=c, cmap=plt.cm.Paired,
                s=20, edgecolor='k',
                label="Class %s" % n)
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.legend(loc='upper right')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Decision Boundary')

# Plot the two-class decision scores
twoclass_output = bdt.decision_function(X)
plot_range = (twoclass_output.min(), twoclass_output.max())
plt.subplot(122)
for i, n, c in zip(range(2), class_names, plot_colors):
    plt.hist(twoclass_output[y == i],
             bins=10,
             range=plot_range,
             facecolor=c,
             label='Class %s' % n,
             alpha=.5,
             edgecolor='k')
x1, x2, y1, y2 = plt.axis()
plt.axis((x1, x2, y1, y2 * 1.2))
```

(continues on next page)

(continued from previous page)

```
plt.legend(loc='upper right')
plt.ylabel('Samples')
plt.xlabel('Score')
plt.title('Decision Scores')

plt.tight_layout()
plt.subplots_adjust(wspace=0.35)
plt.show()
```

Total running time of the script: (0 minutes 2.616 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

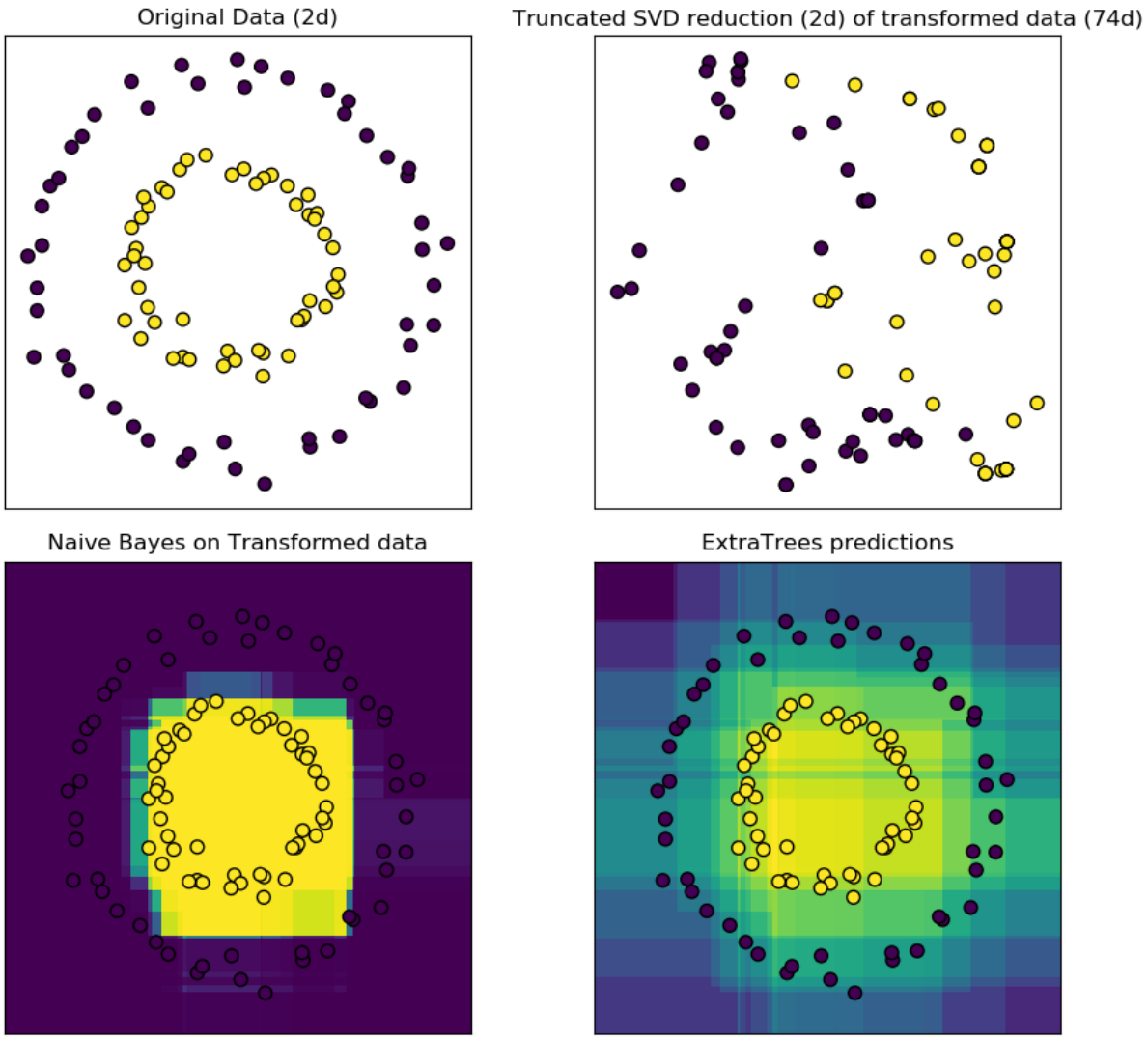
6.11.14 Hashing feature transformation using Totally Random Trees

RandomTreesEmbedding provides a way to map data to a very high-dimensional, sparse representation, which might be beneficial for classification. The mapping is completely unsupervised and very efficient.

This example visualizes the partitions given by several trees and shows how the transformation can also be used for non-linear dimensionality reduction or non-linear classification.

Points that are neighboring often share the same leaf of a tree and therefore share large parts of their hashed representation. This allows to separate two concentric circles simply based on the principal components of the transformed data with truncated SVD.

In high-dimensional spaces, linear classifiers often achieve excellent accuracy. For sparse binary data, BernoulliNB is particularly well-suited. The bottom row compares the decision boundary obtained by BernoulliNB in the transformed space with an ExtraTreesClassifier forests learned on the original data.




```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_circles
from sklearn.ensemble import RandomTreesEmbedding, ExtraTreesClassifier
from sklearn.decomposition import TruncatedSVD
from sklearn.naive_bayes import BernoulliNB

# make a synthetic dataset
X, y = make_circles(factor=0.5, random_state=0, noise=0.05)

# use RandomTreesEmbedding to transform data
hasher = RandomTreesEmbedding(n_estimators=10, random_state=0, max_depth=3)
X_transformed = hasher.fit_transform(X)

# Visualize result after dimensionality reduction using truncated SVD
svd = TruncatedSVD(n_components=2)
X_reduced = svd.fit_transform(X_transformed)
```

(continues on next page)

(continued from previous page)

```
# Learn a Naive Bayes classifier on the transformed data
nb = BernoulliNB()
nb.fit(X_transformed, y)


# Learn an ExtraTreesClassifier for comparison
trees = ExtraTreesClassifier(max_depth=3, n_estimators=10, random_state=0)
trees.fit(X, y)


# scatter plot of original and reduced data
fig = plt.figure(figsize=(9, 8))

ax = plt.subplot(221)
ax.scatter(X[:, 0], X[:, 1], c=y, s=50, edgecolor='k')
ax.set_title("Original Data (2d)")
ax.set_xticks(())
ax.set_yticks(())

ax = plt.subplot(222)
ax.scatter(X_reduced[:, 0], X_reduced[:, 1], c=y, s=50, edgecolor='k')
ax.set_title("Truncated SVD reduction (2d) of transformed data (%dd)" %
             X_transformed.shape[1])
ax.set_xticks(())
ax.set_yticks(())

# Plot the decision in original space. For that, we will assign a color
# to each point in the mesh [x_min, x_max]x[y_min, y_max].
h = .01
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

# transform grid using RandomTreesEmbedding
transformed_grid = hasher.transform(np.c_[xx.ravel(), yy.ravel()])
y_grid_pred = nb.predict_proba(transformed_grid)[:, 1]

ax = plt.subplot(223)
ax.set_title("Naive Bayes on Transformed data")
ax.pcolormesh(xx, yy, y_grid_pred.reshape(xx.shape))
ax.scatter(X[:, 0], X[:, 1], c=y, s=50, edgecolor='k')
ax.set_ylim(-1.4, 1.4)
ax.set_xlim(-1.4, 1.4)
ax.set_xticks(())
ax.set_yticks(())

# transform grid using ExtraTreesClassifier
y_grid_pred = trees.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

ax = plt.subplot(224)
ax.set_title("ExtraTrees predictions")
ax.pcolormesh(xx, yy, y_grid_pred.reshape(xx.shape))
ax.scatter(X[:, 0], X[:, 1], c=y, s=50, edgecolor='k')
ax.set_ylim(-1.4, 1.4)
ax.set_xlim(-1.4, 1.4)
ax.set_xticks(())
ax.set_yticks(())
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.570 seconds)

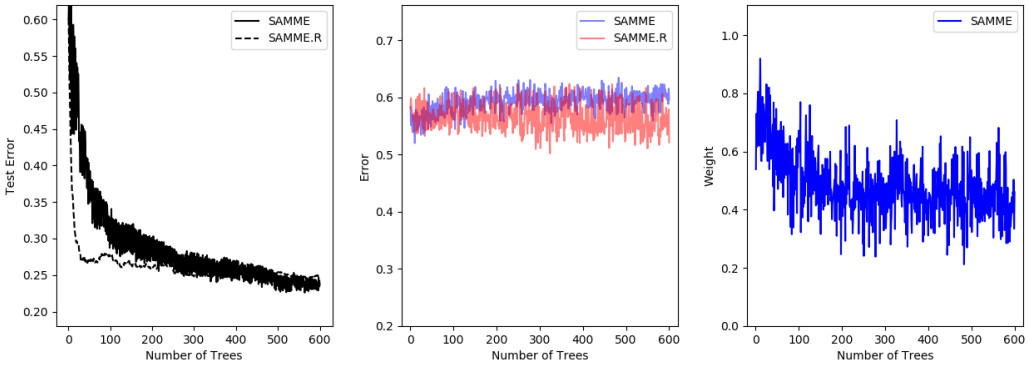
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.15 Multi-class AdaBoosted Decision Trees

This example reproduces Figure 1 of Zhu et al[1] and shows how boosting can improve prediction accuracy on a multi-class problem. The classification dataset is constructed by taking a ten-dimensional standard normal distribution and defining three classes separated by nested concentric ten-dimensional spheres such that roughly equal numbers of samples are in each class (quantiles of the χ^2 distribution).

The performance of the SAMME and SAMME.R[1] algorithms are compared. SAMME.R uses the probability estimates to update the additive model, while SAMME uses the classifications only. As the example illustrates, the SAMME.R algorithm typically converges faster than SAMME, achieving a lower test error with fewer boosting iterations. The error of each algorithm on the test set after each boosting iteration is shown on the left, the classification error on the test set of each tree is shown in the middle, and the boost weight of each tree is shown on the right. All trees have a weight of one in the SAMME.R algorithm and therefore are not shown.




```
print(__doc__)

# Author: Noel Dawe <noel.dawe@gmail.com>
#
# License: BSD 3 clause

import matplotlib.pyplot as plt

from sklearn.datasets import make_gaussian_quantiles
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score
```

(continues on next page)

[1] J. Zhu, H. Zou, S. Rosset, T. Hastie, "Multi-class AdaBoost", 2009.

(continued from previous page)

```
from sklearn.tree import DecisionTreeClassifier


X, y = make_gaussian_quantiles(n_samples=13000, n_features=10,
                               n_classes=3, random_state=1)

n_split = 3000

X_train, X_test = X[:n_split], X[n_split:]
y_train, y_test = y[:n_split], y[n_split:]

bdt_real = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=2),
    n_estimators=600,
    learning_rate=1)

bdt_discrete = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=2),
    n_estimators=600,
    learning_rate=1.5,
    algorithm="SAMME")

bdt_real.fit(X_train, y_train)
bdt_discrete.fit(X_train, y_train)

real_test_errors = []
discrete_test_errors = []

for real_test_predict, discrete_train_predict in zip(
        bdt_real.staged_predict(X_test), bdt_discrete.staged_predict(X_test)):
    real_test_errors.append(
        1. - accuracy_score(real_test_predict, y_test))
    discrete_test_errors.append(
        1. - accuracy_score(discrete_train_predict, y_test))

n_trees_discrete = len(bdt_discrete)
n_trees_real = len(bdt_real)

# Boosting might terminate early, but the following arrays are always
# n_estimators long. We crop them to the actual number of trees here:
discrete_estimator_errors = bdt_discrete.estimator_errors_[:n_trees_discrete]
real_estimator_errors = bdt_real.estimator_errors_[:n_trees_real]
discrete_estimator_weights = bdt_discrete.estimator_weights_[:n_trees_discrete]

plt.figure(figsize=(15, 5))

plt.subplot(131)
plt.plot(range(1, n_trees_discrete + 1),
         discrete_test_errors, c='black', label='SAMME')
plt.plot(range(1, n_trees_real + 1),
         real_test_errors, c='black',
         linestyle='dashed', label='SAMME.R')
plt.legend()
plt.ylim(0.18, 0.62)
plt.ylabel('Test Error')
plt.xlabel('Number of Trees')
```

(continues on next page)

(continued from previous page)

```
plt.subplot(132)
plt.plot(range(1, n_trees_discrete + 1), discrete_estimator_errors,
         "b", label='SAMME', alpha=.5)
plt.plot(range(1, n_trees_real + 1), real_estimator_errors,
         "r", label='SAMME.R', alpha=.5)
plt.legend()
plt.ylabel('Error')
plt.xlabel('Number of Trees')
plt.ylim((.2,
         max(real_estimator_errors.max(),
             discrete_estimator_errors.max()) * 1.2))
plt.xlim((-20, len(bdt_discrete) + 20))

plt.subplot(133)
plt.plot(range(1, n_trees_discrete + 1), discrete_estimator_weights,
         "b", label='SAMME')
plt.legend()
plt.ylabel('Weight')
plt.xlabel('Number of Trees')
plt.ylim((0, discrete_estimator_weights.max() * 1.2))
plt.xlim((-20, n_trees_discrete + 20))

# prevent overlapping y-axis labels
plt.subplots_adjust(wspace=0.25)
plt.show()
```

Total running time of the script: (0 minutes 12.285 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

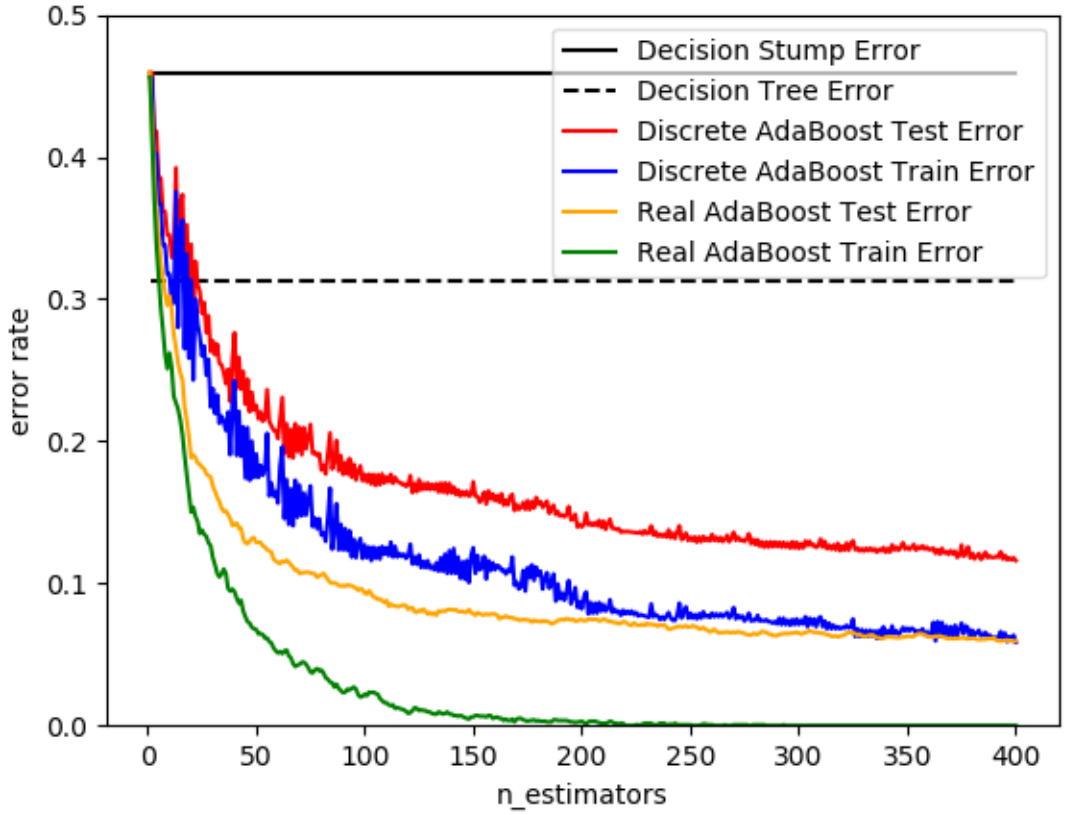
6.11.16 Discrete versus Real AdaBoost

This example is based on Figure 10.2 from Hastie et al 2009[1] and illustrates the difference in performance between the discrete SAMME[2] boosting algorithm and real SAMME.R boosting algorithm. Both algorithms are evaluated on a binary classification task where the target Y is a non-linear function of 10 input features.

Discrete SAMME AdaBoost adapts based on errors in predicted class labels whereas real SAMME.R uses the predicted class probabilities.

[1] T. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning Ed. 2", Springer, 2009.

[2] J. Zhu, H. Zou, S. Rosset, T. Hastie, "Multi-class AdaBoost", 2009.




```
print(__doc__)

# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>,
#         Noel Dawe <noel.dawe@gmail.com>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import zero_one_loss
from sklearn.ensemble import AdaBoostClassifier


n_estimators = 400
# A learning rate of 1. may not be optimal for both SAMME and SAMME.R
learning_rate = 1.

X, y = datasets.make_hastie_10_2(n_samples=12000, random_state=1)

X_test, y_test = X[2000:], y[2000:]
X_train, y_train = X[:2000], y[:2000]
```

(continues on next page)

(continued from previous page)

```
dt_stump = DecisionTreeClassifier(max_depth=1, min_samples_leaf=1)
dt_stump.fit(X_train, y_train)
dt_stump_err = 1.0 - dt_stump.score(X_test, y_test)

dt = DecisionTreeClassifier(max_depth=9, min_samples_leaf=1)
dt.fit(X_train, y_train)
dt_err = 1.0 - dt.score(X_test, y_test)

ada_discrete = AdaBoostClassifier(
    base_estimator=dt_stump,
    learning_rate=learning_rate,
    n_estimators=n_estimators,
    algorithm="SAMME")
ada_discrete.fit(X_train, y_train)

ada_real = AdaBoostClassifier(
    base_estimator=dt_stump,
    learning_rate=learning_rate,
    n_estimators=n_estimators,
    algorithm="SAMME.R")
ada_real.fit(X_train, y_train)

fig = plt.figure()
ax = fig.add_subplot(111)

ax.plot([1, n_estimators], [dt_stump_err] * 2, 'k-',
        label='Decision Stump Error')
ax.plot([1, n_estimators], [dt_err] * 2, 'k--',
        label='Decision Tree Error')

ada_discrete_err = np.zeros((n_estimators,))
for i, y_pred in enumerate(ada_discrete.staged_predict(X_test)):
    ada_discrete_err[i] = zero_one_loss(y_pred, y_test)

ada_discrete_err_train = np.zeros((n_estimators,))
for i, y_pred in enumerate(ada_discrete.staged_predict(X_train)):
    ada_discrete_err_train[i] = zero_one_loss(y_pred, y_train)

ada_real_err = np.zeros((n_estimators,))
for i, y_pred in enumerate(ada_real.staged_predict(X_test)):
    ada_real_err[i] = zero_one_loss(y_pred, y_test)

ada_real_err_train = np.zeros((n_estimators,))
for i, y_pred in enumerate(ada_real.staged_predict(X_train)):
    ada_real_err_train[i] = zero_one_loss(y_pred, y_train)

ax.plot(np.arange(n_estimators) + 1, ada_discrete_err,
        label='Discrete AdaBoost Test Error',
        color='red')
ax.plot(np.arange(n_estimators) + 1, ada_discrete_err_train,
        label='Discrete AdaBoost Train Error',
        color='blue')
ax.plot(np.arange(n_estimators) + 1, ada_real_err,
        label='Real AdaBoost Test Error',
        color='orange')
ax.plot(np.arange(n_estimators) + 1, ada_real_err_train,
        label='Real AdaBoost Train Error',
```

(continues on next page)

(continued from previous page)

```
        color='green')

ax.set_ylim((0.0, 0.5))
ax.set_xlabel('n_estimators')
ax.set_ylabel('error rate')

leg = ax.legend(loc='upper right', fancybox=True)
leg.get_frame().set_alpha(0.7)

plt.show()
```

Total running time of the script: (0 minutes 5.030 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.17 Combine predictors using stacking

Stacking refers to a method to blend estimators. In this strategy, some estimators are individually fitted on some training data while a final estimator is trained using the stacked predictions of these base estimators.

In this example, we illustrate the use case in which different regressors are stacked together and a final linear penalized regressor is used to output the prediction. We compare the performance of each individual regressor with the stacking strategy. Stacking slightly improves the overall performance.

```
print(__doc__)

# Authors: Guillaume Lemaitre <g.lemaitre58@gmail.com>
# License: BSD 3 clause
```

The function `plot_regression_results` is used to plot the predicted and true targets.

```
import matplotlib.pyplot as plt


def plot_regression_results(ax, y_true, y_pred, title, scores, elapsed_time):
    """Scatter plot of the predicted vs true targets."""
    ax.plot([y_true.min(), y_true.max()],
            [y_true.min(), y_true.max()],
            '--r', linewidth=2)
    ax.scatter(y_true, y_pred, alpha=0.2)

    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()
    ax.spines['left'].set_position(('outward', 10))
    ax.spines['bottom'].set_position(('outward', 10))
    ax.set_xlim([y_true.min(), y_true.max()])
    ax.set_ylim([y_true.min(), y_true.max()])
    ax.set_xlabel('Measured')
    ax.set_ylabel('Predicted')
    extra = plt.Rectangle((0, 0), 0, 0, fc="w", fill=False,
```

(continues on next page)

(continued from previous page)

```
                           edgecolor='none', linewidth=0)
    ax.legend([extra], [scores], loc='upper left')
    title = title + '\n Evaluation in {:.2f} seconds'.format(elapsed_time)
    ax.set_title(title)
```

Stack of predictors on a single data set

It is sometimes tedious to find the model which will best perform on a given dataset. Stacking provide an alternative by combining the outputs of several learners, without the need to choose a model specifically. The performance of stacking is usually close to the best model and sometimes it can outperform the prediction performance of each individual model.

Here, we combine 3 learners (linear and non-linear) and use a ridge regressor to combine their outputs together.

```
from sklearn.ensemble import StackingRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.experimental import enable_hist_gradient_boosting  # noqa
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.linear_model import LassoCV
from sklearn.linear_model import RidgeCV

estimators = [
    ('Random Forest', RandomForestRegressor(random_state=42)),
    ('Lasso', LassoCV()),
    ('Gradient Boosting', HistGradientBoostingRegressor(random_state=0))
]
stacking_regressor = StackingRegressor(
    estimators=estimators, final_estimator=RidgeCV()
)
```

We used the Boston data set (prediction of house prices). We check the performance of each individual predictor as well as the stack of the regressors.

```
import time
import numpy as np
from sklearn.datasets import load_boston
from sklearn.model_selection import cross_validate, cross_val_predict

X, y = load_boston(return_X_y=True)

fig, axs = plt.subplots(2, 2, figsize=(9, 7))
axs = np.ravel(axs)

for ax, (name, est) in zip(axs, estimators + [('Stacking Regressor',
                                               stacking_regressor)]):
    start_time = time.time()
    score = cross_validate(est, X, y,
                           scoring=['r2', 'neg_mean_absolute_error'],
                           n_jobs=-1, verbose=0)
    elapsed_time = time.time() - start_time

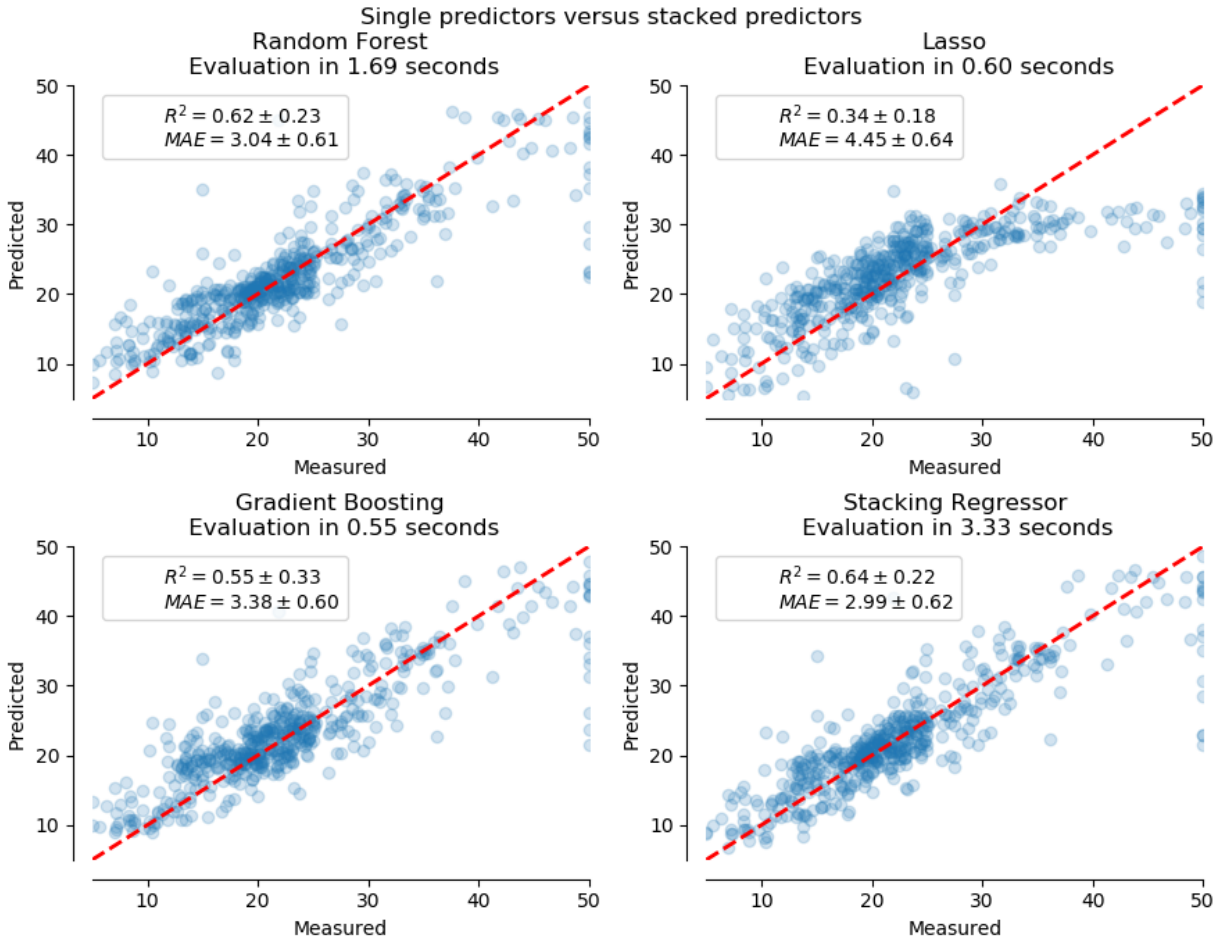
    y_pred = cross_val_predict(est, X, y, n_jobs=-1, verbose=0)
    plot_regression_results(
        ax, y, y_pred,
```

(continues on next page)

(continued from previous page)

```
        name,
        (r'$R^2={:.2f} \pm {:.2f}$' + '\n' + r'$MAE={:.2f} \pm {:.2f}$')
        .format(np.mean(score['test_r2']),
                np.std(score['test_r2']),
                -np.mean(score['test_neg_mean_absolute_error']),
                np.std(score['test_neg_mean_absolute_error'])),
        elapsed_time)

plt.suptitle('Single predictors versus stacked predictors')
plt.tight_layout()
plt.subplots_adjust(top=0.9)
plt.show()
```



The stacked regressor will combine the strengths of the different regressors. However, we also see that training the stacked regressor is much more computationally expensive.

Total running time of the script: (0 minutes 11.613 seconds)

Estimated memory usage: 10 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.18 Early stopping of Gradient Boosting

Gradient boosting is an ensembling technique where several weak learners (regression trees) are combined to yield a powerful single model, in an iterative fashion.

Early stopping support in Gradient Boosting enables us to find the least number of iterations which is sufficient to build a model that generalizes well to unseen data.

The concept of early stopping is simple. We specify a `validation_fraction` which denotes the fraction of the whole dataset that will be kept aside from training to assess the validation loss of the model. The gradient boosting model is trained using the training set and evaluated using the validation set. When each additional stage of regression tree is added, the validation set is used to score the model. This is continued until the scores of the model in the last `n_iter_no_change` stages do not improve by atleast `tol`. After that the model is considered to have converged and further addition of stages is "stopped early".

The number of stages of the final model is available at the attribute `n_estimators_`.

This example illustrates how the early stopping can used in the *sklearn.ensemble.GradientBoostingClassifier* model to achieve almost the same accuracy as compared to a model built without early stopping using many fewer estimators. This can significantly reduce training time, memory usage and prediction latency.

```
# Authors: Vighnesh Birodkar <vighneshbirodkar@nyu.edu>
#          Raghav RV <rvraghav93@gmail.com>
# License: BSD 3 clause

import time

import numpy as np
import matplotlib.pyplot as plt

from sklearn import ensemble
from sklearn import datasets
from sklearn.model_selection import train_test_split

print(__doc__)

data_list = [datasets.load_iris(), datasets.load_digits()]
data_list = [(d.data, d.target) for d in data_list]
data_list += [datasets.make_hastie_10_2()]
names = ['Iris Data', 'Digits Data', 'Hastie Data']

n_gb = []
score_gb = []
time_gb = []
n_gbes = []
score_gbes = []
time_gbes = []

n_estimators = 500

for X, y in data_list:
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                        random_state=0)

    # We specify that if the scores don't improve by atleast 0.01 for the last
    # 10 stages, stop fitting additional stages
    gbes = ensemble.GradientBoostingClassifier(n_estimators=n_estimators,
```

(continues on next page)

(continued from previous page)

```
                                                validation_fraction=0.2,
                                                n_iter_no_change=5, tol=0.01,
                                                random_state=0)
    gb = ensemble.GradientBoostingClassifier(n_estimators=n_estimators,
                                             random_state=0)
    start = time.time()
    gb.fit(X_train, y_train)
    time_gb.append(time.time() - start)

    start = time.time()
    gbes.fit(X_train, y_train)
    time_gbes.append(time.time() - start)

    score_gb.append(gb.score(X_test, y_test))
    score_gbes.append(gbes.score(X_test, y_test))

    n_gb.append(gb.n_estimators_)
    n_gbes.append(gbes.n_estimators_)

bar_width = 0.2
n = len(data_list)
index = np.arange(0, n * bar_width, bar_width) * 2.5
index = index[0:n]
```

Compare scores with and without early stopping

```
plt.figure(figsize=(9, 5))

bar1 = plt.bar(index, score_gb, bar_width, label='Without early stopping',
               color='crimson')
bar2 = plt.bar(index + bar_width, score_gbes, bar_width,
               label='With early stopping', color='coral')

plt.xticks(index + bar_width, names)
plt.yticks(np.arange(0, 1.3, 0.1))


def autolabel(rects, n_estimators):
    """
    Attach a text label above each bar displaying n_estimators of each model
    """
    for i, rect in enumerate(rects):
        plt.text(rect.get_x() + rect.get_width() / 2.,
                 1.05 * rect.get_height(), 'n_est=%d' % n_estimators[i],
                 ha='center', va='bottom')


autolabel(bar1, n_gb)
autolabel(bar2, n_gbes)

plt.ylim([0, 1.3])
plt.legend(loc='best')
plt.grid(True)

plt.xlabel('Datasets')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Test score')

plt.show()
```

Compare fit times with and without early stopping

```
plt.figure(figsize=(9, 5))

bar1 = plt.bar(index, time_gb, bar_width, label='Without early stopping',
               color='crimson')
bar2 = plt.bar(index + bar_width, time_gbes, bar_width,
               label='With early stopping', color='coral')

max_y = np.amax(np.maximum(time_gb, time_gbes))

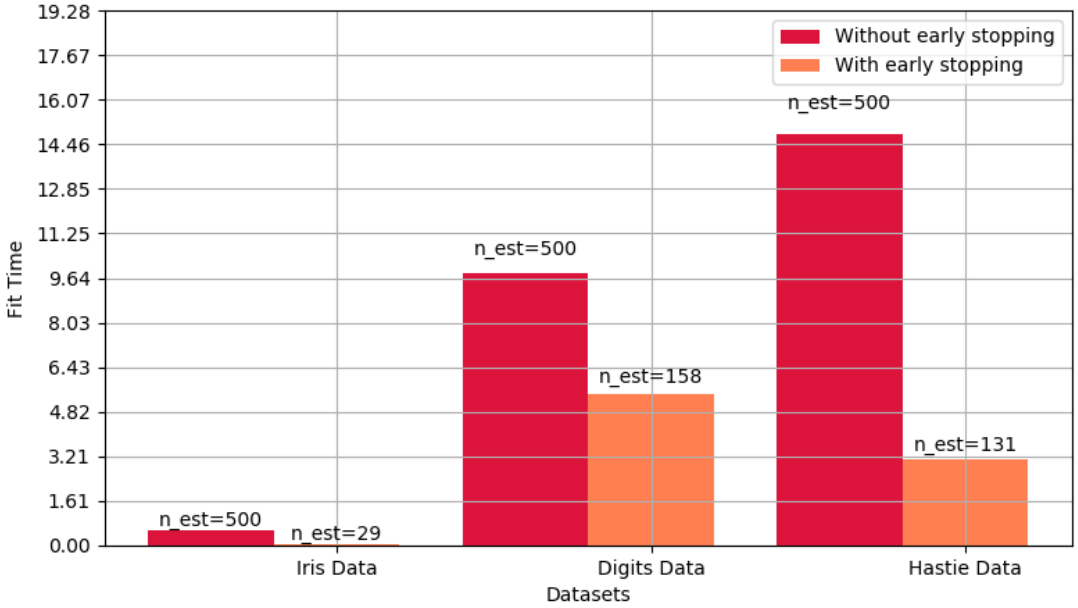
plt.xticks(index + bar_width, names)
plt.yticks(np.linspace(0, 1.3 * max_y, 13))

autolabel(bar1, n_gb)
autolabel(bar2, n_gbes)

plt.ylim([0, 1.3 * max_y])
plt.legend(loc='best')
plt.grid(True)

plt.xlabel('Datasets')
plt.ylabel('Fit Time')

plt.show()
```



Total running time of the script: (0 minutes 34.557 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

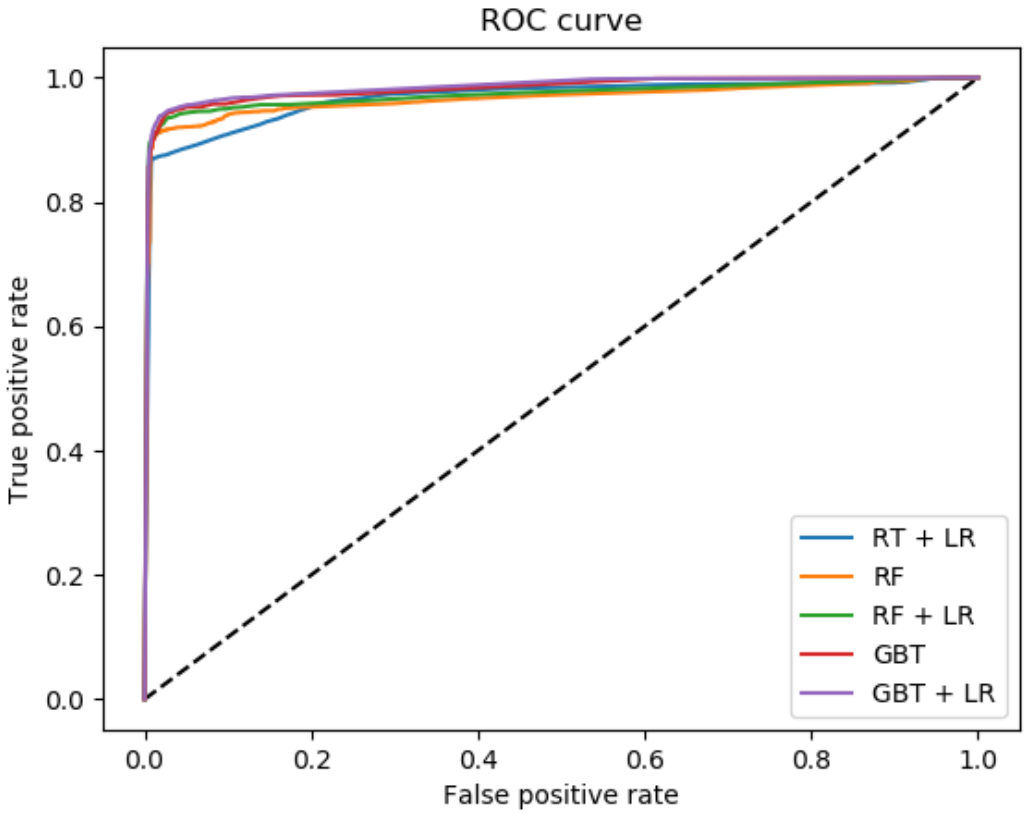
6.11.19 Feature transformations with ensembles of trees

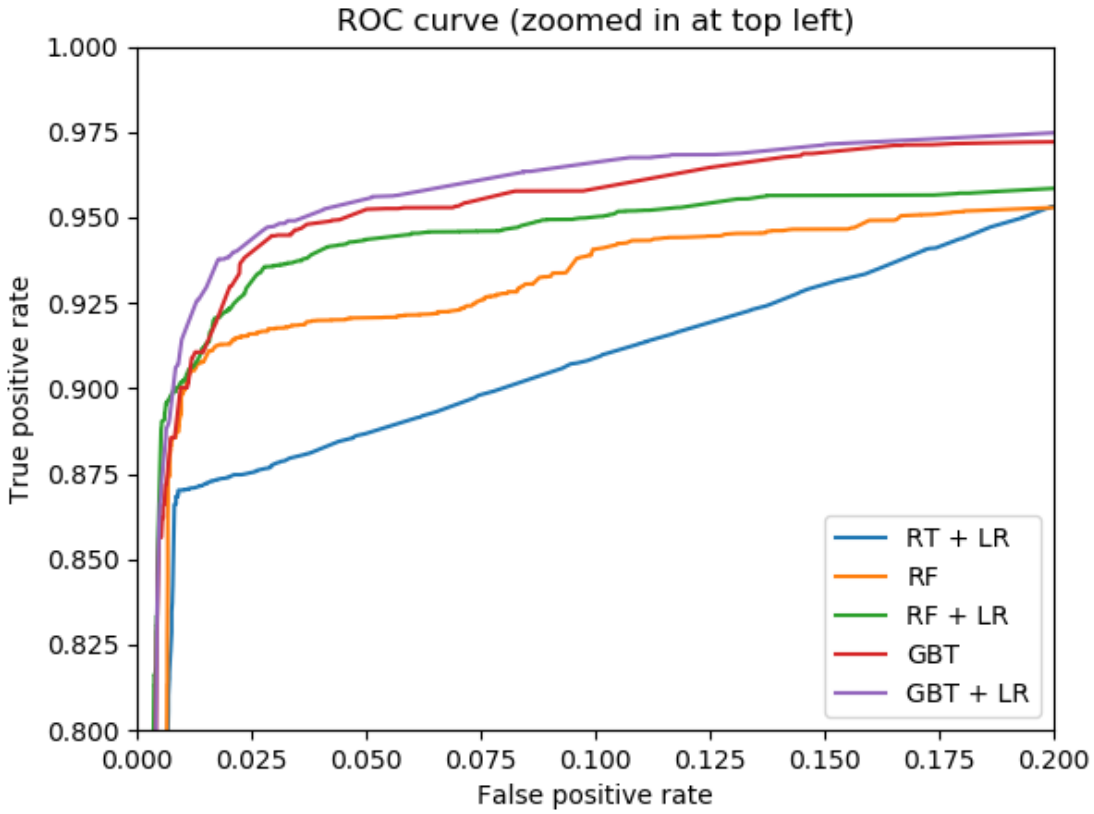
Transform your features into a higher dimensional, sparse space. Then train a linear model on these features.

First fit an ensemble of trees (totally random trees, a random forest, or gradient boosted trees) on the training set. Then each leaf of each tree in the ensemble is assigned a fixed arbitrary feature index in a new feature space. These leaf indices are then encoded in a one-hot fashion.

Each sample goes through the decisions of each tree of the ensemble and ends up in one leaf per tree. The sample is encoded by setting feature values for these leaves to 1 and the other feature values to 0.

The resulting transformer has then learned a supervised, sparse, high-dimensional categorical embedding of the data.


ROC curve
True positive rate
False positive rate
0.0
0.2
0.4
0.6
0.8
1.0
RT + LR
RF
RF + LR
GBT
GBT + LR




```
# Author: Tim Head <betatim@gmail.com>
#
# License: BSD 3 clause

import numpy as np
np.random.seed(10)

import matplotlib.pyplot as plt

from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import (RandomTreesEmbedding, RandomForestClassifier,
                              GradientBoostingClassifier)
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve
from sklearn.pipeline import make_pipeline

n_estimator = 10
X, y = make_classification(n_samples=80000)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)

# It is important to train the ensemble of trees on a different subset
# of the training data than the linear regression model to avoid
# overfitting, in particular if the total number of leaves is
# similar to the number of training samples
X_train, X_train_lr, y_train, y_train_lr = train_test_split(
```

(continues on next page)

(continued from previous page)

```
    X_train, y_train, test_size=0.5)

# Unsupervised transformation based on totally random trees
rt = RandomTreesEmbedding(max_depth=3, n_estimators=n_estimator,
                          random_state=0)

rt_lm = LogisticRegression(max_iter=1000)
pipeline = make_pipeline(rt, rt_lm)
pipeline.fit(X_train, y_train)
y_pred_rt = pipeline.predict_proba(X_test)[:, 1]
fpr_rt_lm, tpr_rt_lm, _ = roc_curve(y_test, y_pred_rt)

# Supervised transformation based on random forests
rf = RandomForestClassifier(max_depth=3, n_estimators=n_estimator)
rf_enc = OneHotEncoder()
rf_lm = LogisticRegression(max_iter=1000)
rf.fit(X_train, y_train)
rf_enc.fit(rf.apply(X_train))
rf_lm.fit(rf_enc.transform(rf.apply(X_train_lr)), y_train_lr)

y_pred_rf_lm = rf_lm.predict_proba(rf_enc.transform(rf.apply(X_test)))[:, 1]
fpr_rf_lm, tpr_rf_lm, _ = roc_curve(y_test, y_pred_rf_lm)

# Supervised transformation based on gradient boosted trees
grd = GradientBoostingClassifier(n_estimators=n_estimator)
grd_enc = OneHotEncoder()
grd_lm = LogisticRegression(max_iter=1000)
grd.fit(X_train, y_train)
grd_enc.fit(grd.apply(X_train)[:, :, 0])
grd_lm.fit(grd_enc.transform(grd.apply(X_train_lr)[:, :, 0]), y_train_lr)

y_pred_grd_lm = grd_lm.predict_proba(
    grd_enc.transform(grd.apply(X_test)[:, :, 0]))[:, 1]
fpr_grd_lm, tpr_grd_lm, _ = roc_curve(y_test, y_pred_grd_lm)

# The gradient boosted model by itself
y_pred_grd = grd.predict_proba(X_test)[:, 1]
fpr_grd, tpr_grd, _ = roc_curve(y_test, y_pred_grd)

# The random forest model by itself
y_pred_rf = rf.predict_proba(X_test)[:, 1]
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_pred_rf)

plt.figure(1)
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr_rt_lm, tpr_rt_lm, label='RT + LR')
plt.plot(fpr_rf, tpr_rf, label='RF')
plt.plot(fpr_rf_lm, tpr_rf_lm, label='RF + LR')
plt.plot(fpr_grd, tpr_grd, label='GBT')
plt.plot(fpr_grd_lm, tpr_grd_lm, label='GBT + LR')
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve')
plt.legend(loc='best')
plt.show()

plt.figure(2)
```

(continues on next page)

(continued from previous page)

```
plt.xlim(0, 0.2)
plt.ylim(0.8, 1)
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr_rt_lm, tpr_rt_lm, label='RT + LR')
plt.plot(fpr_rf, tpr_rf, label='RF')
plt.plot(fpr_rf_lm, tpr_rf_lm, label='RF + LR')
plt.plot(fpr_grd, tpr_grd, label='GBT')
plt.plot(fpr_grd_lm, tpr_grd_lm, label='GBT + LR')
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve (zoomed in at top left)')
plt.legend(loc='best')
plt.show()
```

Total running time of the script: (0 minutes 3.257 seconds)

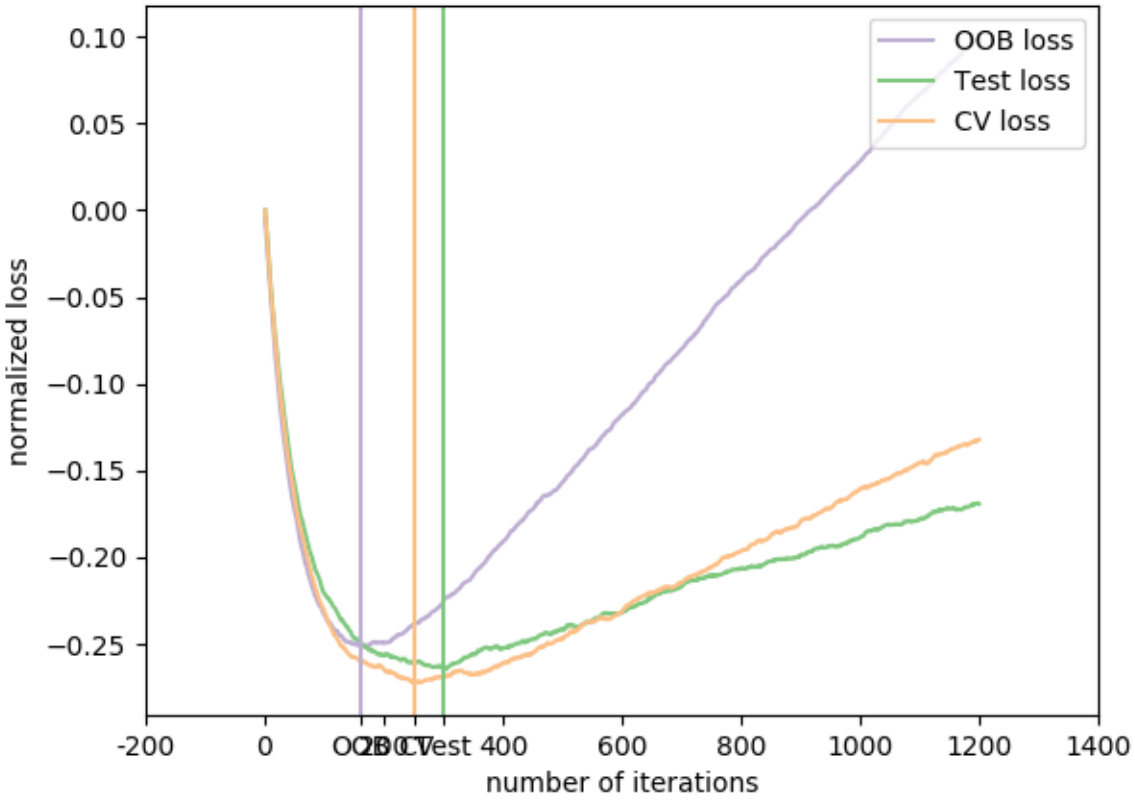
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.20 Gradient Boosting Out-of-Bag estimates

Out-of-bag (OOB) estimates can be a useful heuristic to estimate the "optimal" number of boosting iterations. OOB estimates are almost identical to cross-validation estimates but they can be computed on-the-fly without the need for repeated model fitting. OOB estimates are only available for Stochastic Gradient Boosting (i.e. `subsample < 1.0`), the estimates are derived from the improvement in loss based on the examples not included in the bootstrap sample (the so-called out-of-bag examples). The OOB estimator is a pessimistic estimator of the true test loss, but remains a fairly good approximation for a small number of trees.

The figure shows the cumulative sum of the negative OOB improvements as a function of the boosting iteration. As you can see, it tracks the test loss for the first hundred iterations but then diverges in a pessimistic way. The figure also shows the performance of 3-fold cross validation which usually gives a better estimate of the test loss but is computationally more demanding.



Out:

```
Accuracy: 0.6820
```

```
print(__doc__)

# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn import ensemble
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split

from scipy.special import expit

# Generate data (adapted from G. Ridgeway's gbm example)
```

(continues on next page)

(continued from previous page)

```
n_samples = 1000
random_state = np.random.RandomState(13)
x1 = random_state.uniform(size=n_samples)
x2 = random_state.uniform(size=n_samples)
x3 = random_state.randint(0, 4, size=n_samples)

p = expit(np.sin(3 * x1) - 4 * x2 + x3)
y = random_state.binomial(1, p, size=n_samples)

X = np.c_[x1, x2, x3]

X = X.astype(np.float32)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,
                                                    random_state=9)

# Fit classifier with out-of-bag estimates
params = {'n_estimators': 1200, 'max_depth': 3, 'subsample': 0.5,
          'learning_rate': 0.01, 'min_samples_leaf': 1, 'random_state': 3}
clf = ensemble.GradientBoostingClassifier(**params)

clf.fit(X_train, y_train)
acc = clf.score(X_test, y_test)
print("Accuracy: {:.4f}".format(acc))

n_estimators = params['n_estimators']
x = np.arange(n_estimators) + 1


def heldout_score(clf, X_test, y_test):
    """compute deviance scores on ``X_test`` and ``y_test``. """
    score = np.zeros((n_estimators,), dtype=np.float64)
    for i, y_pred in enumerate(clf.staged_decision_function(X_test)):
        score[i] = clf.loss_(y_test, y_pred)
    return score


def cv_estimate(n_splits=None):
    cv = KFold(n_splits=n_splits)
    cv_clf = ensemble.GradientBoostingClassifier(**params)
    val_scores = np.zeros((n_estimators,), dtype=np.float64)
    for train, test in cv.split(X_train, y_train):
        cv_clf.fit(X_train[train], y_train[train])
        val_scores += heldout_score(cv_clf, X_train[test], y_train[test])
    val_scores /= n_splits
    return val_scores


# Estimate best n_estimator using cross-validation
cv_score = cv_estimate(3)

# Compute best n_estimator for test data
test_score = heldout_score(clf, X_test, y_test)

# negative cumulative sum of oob improvements
cumsum = -np.cumsum(clf.oob_improvement_)

# min loss according to OOB
```

(continues on next page)

(continued from previous page)

```
oob_best_iter = x[np.argmin(cumsum)]

# min loss according to test (normalize such that first loss is 0)
test_score -= test_score[0]
test_best_iter = x[np.argmin(test_score)]

# min loss according to cv (normalize such that first loss is 0)
cv_score -= cv_score[0]
cv_best_iter = x[np.argmin(cv_score)]

# color brew for the three curves
oob_color = list(map(lambda x: x / 256.0, (190, 174, 212)))
test_color = list(map(lambda x: x / 256.0, (127, 201, 127)))
cv_color = list(map(lambda x: x / 256.0, (253, 192, 134)))

# plot curves and vertical lines for best iterations
plt.plot(x, cumsum, label='OOB loss', color=oob_color)
plt.plot(x, test_score, label='Test loss', color=test_color)
plt.plot(x, cv_score, label='CV loss', color=cv_color)
plt.axvline(x=oob_best_iter, color=oob_color)
plt.axvline(x=test_best_iter, color=test_color)
plt.axvline(x=cv_best_iter, color=cv_color)

# add three vertical lines to xticks
xticks = plt.xticks()
xticks_pos = np.array(xticks[0].tolist() +
                      [oob_best_iter, cv_best_iter, test_best_iter])
xticks_label = np.array(list(map(lambda t: int(t), xticks[0])) +
                        ['OOB', 'CV', 'Test'])
ind = np.argsort(xticks_pos)
xticks_pos = xticks_pos[ind]
xticks_label = xticks_label[ind]
plt.xticks(xticks_pos, xticks_label)

plt.legend(loc='upper right')
plt.ylabel('normalized loss')
plt.xlabel('number of iterations')

plt.show()
```

Total running time of the script: (0 minutes 3.271 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.21 Single estimator versus bagging: bias-variance decomposition

This example illustrates and compares the bias-variance decomposition of the expected mean squared error of a single estimator against a bagging ensemble.

In regression, the expected mean squared error of an estimator can be decomposed in terms of bias, variance and noise. On average over datasets of the regression problem, the bias term measures the average amount by which the predictions of the estimator differ from the predictions of the best possible estimator for the problem (i.e., the Bayes

model). The variance term measures the variability of the predictions of the estimator when fit over different instances LS of the problem. Finally, the noise measures the irreducible part of the error which is due the variability in the data.

The upper left figure illustrates the predictions (in dark red) of a single decision tree trained over a random dataset LS (the blue dots) of a toy 1d regression problem. It also illustrates the predictions (in light red) of other single decision trees trained over other (and different) randomly drawn instances LS of the problem. Intuitively, the variance term here corresponds to the width of the beam of predictions (in light red) of the individual estimators. The larger the variance, the more sensitive are the predictions for `x` to small changes in the training set. The bias term corresponds to the difference between the average prediction of the estimator (in cyan) and the best possible model (in dark blue). On this problem, we can thus observe that the bias is quite low (both the cyan and the blue curves are close to each other) while the variance is large (the red beam is rather wide).

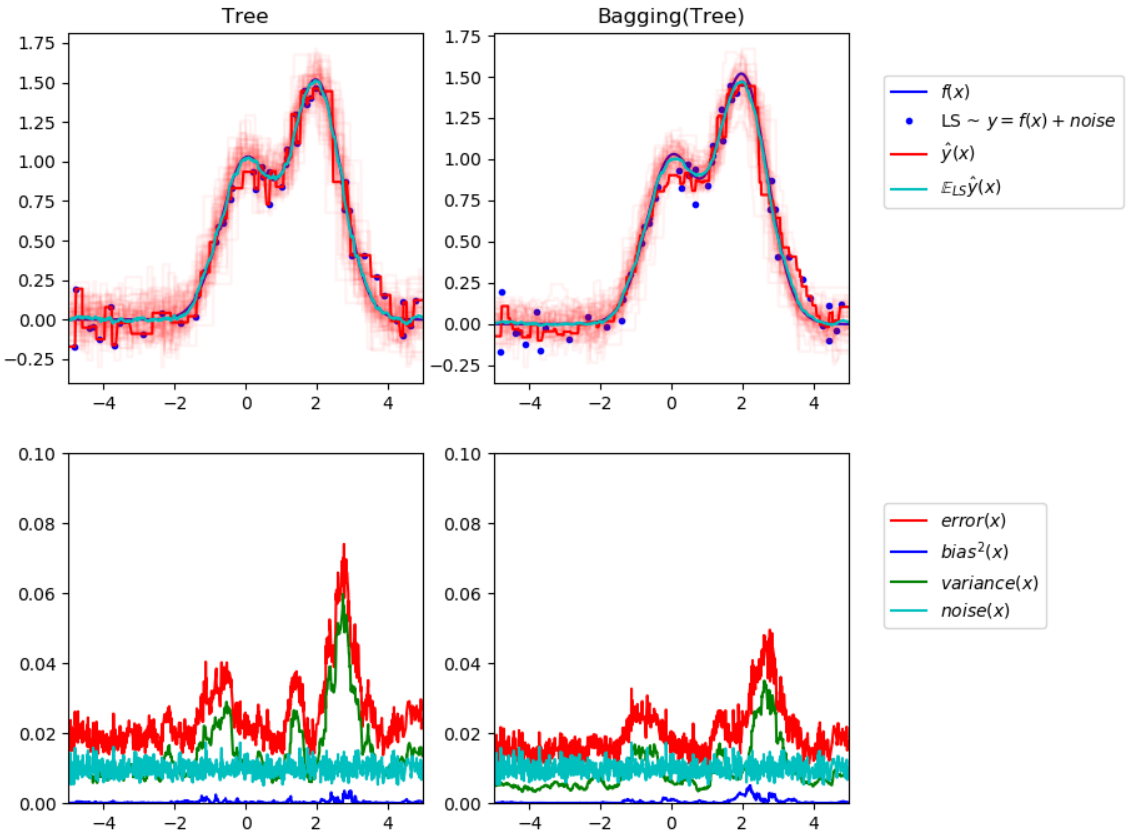
The lower left figure plots the pointwise decomposition of the expected mean squared error of a single decision tree. It confirms that the bias term (in blue) is low while the variance is large (in green). It also illustrates the noise part of the error which, as expected, appears to be constant and around `0.01`.

The right figures correspond to the same plots but using instead a bagging ensemble of decision trees. In both figures, we can observe that the bias term is larger than in the previous case. In the upper right figure, the difference between the average prediction (in cyan) and the best possible model is larger (e.g., notice the offset around `x=2`). In the lower right figure, the bias curve is also slightly higher than in the lower left figure. In terms of variance however, the beam of predictions is narrower, which suggests that the variance is lower. Indeed, as the lower right figure confirms, the variance term (in green) is lower than for single decision trees. Overall, the bias- variance decomposition is therefore no longer the same. The tradeoff is better for bagging: averaging several decision trees fit on bootstrap copies of the dataset slightly increases the bias term but allows for a larger reduction of the variance, which results in a lower overall mean squared error (compare the red curves int the lower figures). The script output also confirms this intuition. The total error of the bagging ensemble is lower than the total error of a single decision tree, and this difference indeed mainly stems from a reduced variance.

For further details on bias-variance decomposition, see section 7.3 of[1].

[1] T. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning", Springer, 2009.

References



Out:

```
Tree: 0.0255 (error) = 0.0003 (bias^2)  + 0.0152 (var) + 0.0098 (noise)
Bagging(Tree): 0.0196 (error) = 0.0004 (bias^2)  + 0.0092 (var) + 0.0098 (noise)
```

```
print(__doc__)

# Author: Gilles Louppe <g.louppe@gmail.com>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.ensemble import BaggingRegressor
from sklearn.tree import DecisionTreeRegressor
```

(continues on next page)

(continued from previous page)

```
# Settings
n_repeat = 50       # Number of iterations for computing expectations
n_train = 50        # Size of the training set
n_test = 1000       # Size of the test set
noise = 0.1         # Standard deviation of the noise
np.random.seed(0)

# Change this for exploring the bias-variance decomposition of other
# estimators. This should work well for estimators with high variance (e.g.,
# decision trees or KNN), but poorly for estimators with low variance (e.g.,
# linear models).
estimators = [("Tree", DecisionTreeRegressor()),
              ("Bagging(Tree)", BaggingRegressor(DecisionTreeRegressor()))]

n_estimators = len(estimators)


# Generate data
def f(x):
    x = x.ravel()

    return np.exp(-x ** 2) + 1.5 * np.exp(-(x - 2) ** 2)


def generate(n_samples, noise, n_repeat=1):
    X = np.random.rand(n_samples) * 10 - 5
    X = np.sort(X)

    if n_repeat == 1:
        y = f(X) + np.random.normal(0.0, noise, n_samples)
    else:
        y = np.zeros((n_samples, n_repeat))

        for i in range(n_repeat):
            y[:, i] = f(X) + np.random.normal(0.0, noise, n_samples)

    X = X.reshape((n_samples, 1))

    return X, y


X_train = []
y_train = []

for i in range(n_repeat):
    X, y = generate(n_samples=n_train, noise=noise)
    X_train.append(X)
    y_train.append(y)

X_test, y_test = generate(n_samples=n_test, noise=noise, n_repeat=n_repeat)

plt.figure(figsize=(10, 8))

# Loop over estimators to compare
for n, (name, estimator) in enumerate(estimators):
    # Compute predictions
```

(continues on next page)

(continued from previous page)

```
    y_predict = np.zeros((n_test, n_repeat))

    for i in range(n_repeat):
        estimator.fit(X_train[i], y_train[i])
        y_predict[:, i] = estimator.predict(X_test)

    # Bias^2 + Variance + Noise decomposition of the mean squared error
    y_error = np.zeros(n_test)

    for i in range(n_repeat):
        for j in range(n_repeat):
            y_error += (y_test[:, j] - y_predict[:, i]) ** 2

    y_error /= (n_repeat * n_repeat)

    y_noise = np.var(y_test, axis=1)
    y_bias = (f(X_test) - np.mean(y_predict, axis=1)) ** 2
    y_var = np.var(y_predict, axis=1)

    print("{0}: {1:.4f} (error) = {2:.4f} (bias^2) "
          " + {3:.4f} (var) + {4:.4f} (noise)".format(name,
                                                      np.mean(y_error),
                                                      np.mean(y_bias),
                                                      np.mean(y_var),
                                                      np.mean(y_noise)))

    # Plot figures
    plt.subplot(2, n_estimators, n + 1)
    plt.plot(X_test, f(X_test), "b", label="$f(x)$")
    plt.plot(X_train[0], y_train[0], ".b", label="LS ~ $y = f(x)+noise$")

    for i in range(n_repeat):
        if i == 0:
            plt.plot(X_test, y_predict[:, i], "r", label=r"$\^y(x)$")
        else:
            plt.plot(X_test, y_predict[:, i], "r", alpha=0.05)

    plt.plot(X_test, np.mean(y_predict, axis=1), "c",
             label=r"$\mathbb{E}_{LS} \^y(x)$")

    plt.xlim([-5, 5])
    plt.title(name)

    if n == n_estimators - 1:
        plt.legend(loc=(1.1, .5))

    plt.subplot(2, n_estimators, n_estimators + n + 1)
    plt.plot(X_test, y_error, "r", label="$error(x)$")
    plt.plot(X_test, y_bias, "b", label="$bias^2(x)$"),
    plt.plot(X_test, y_var, "g", label="$variance(x)$"),
    plt.plot(X_test, y_noise, "c", label="$noise(x)$")

    plt.xlim([-5, 5])
    plt.ylim([0, 0.1])

    if n == n_estimators - 1:
```

(continues on next page)

(continued from previous page)

```
        plt.legend(loc=(1.1, .5))

plt.subplots_adjust(right=.75)
plt.show()
```

Total running time of the script: (0 minutes 1.151 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.11.22 Plot the decision surfaces of ensembles of trees on the iris dataset

Plot the decision surfaces of forests of randomized trees trained on pairs of features of the iris dataset.

This plot compares the decision surfaces learned by a decision tree classifier (first column), by a random forest classifier (second column), by an extra- trees classifier (third column) and by an AdaBoost classifier (fourth column).

In the first row, the classifiers are built using the sepal width and the sepal length features only, on the second row using the petal length and sepal length only, and on the third row using the petal width and the petal length only.

In descending order of quality, when trained (outside of this example) on all 4 features using 30 estimators and scored using 10 fold cross validation, we see:

```
ExtraTreesClassifier()  # 0.95 score
RandomForestClassifier()  # 0.94 score
AdaBoost(DecisionTree(max_depth=3))  # 0.94 score
DecisionTree(max_depth=None)  # 0.94 score
```

Increasing `max_depth` for AdaBoost lowers the standard deviation of the scores (but the average score does not improve).

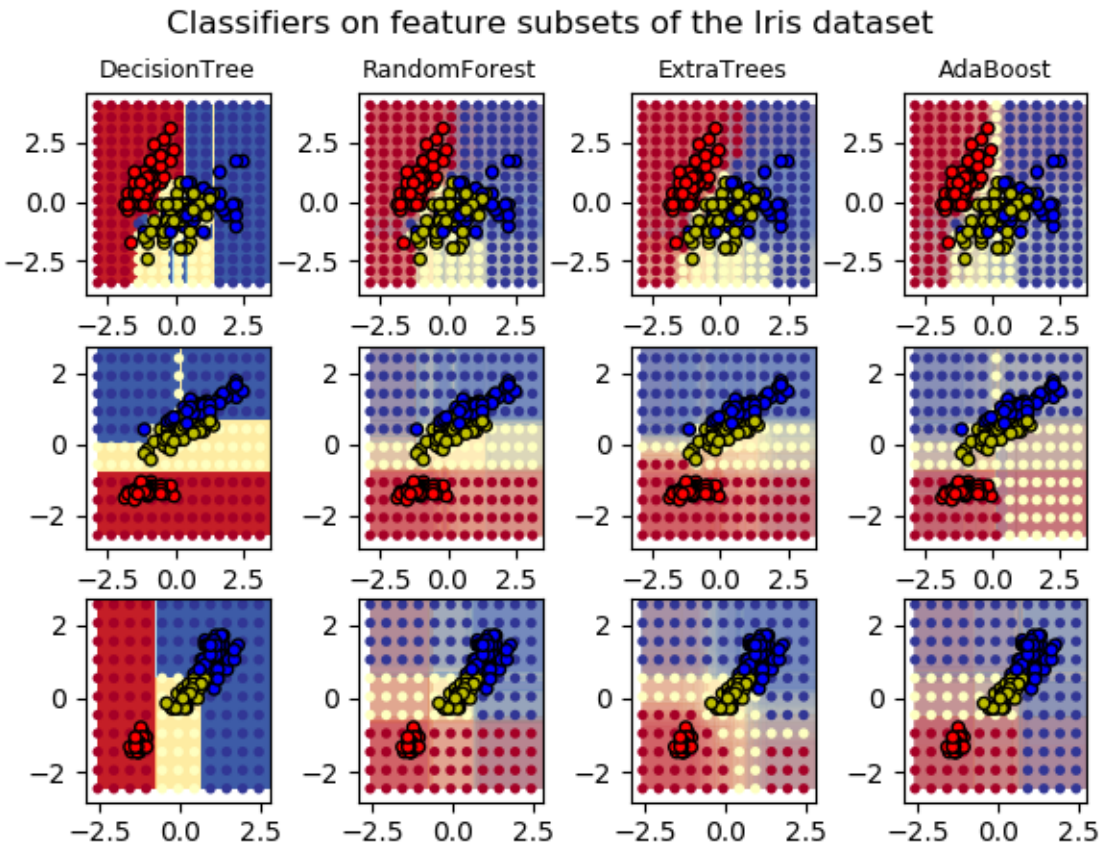
See the console's output for further details about each model.

In this example you might try to:

1) vary the `max_depth` for the `DecisionTreeClassifier` and `AdaBoostClassifier`, perhaps try `max_depth=3` for the `DecisionTreeClassifier` or `max_depth=None` for `AdaBoostClassifier`

2) vary `n_estimators`

It is worth noting that RandomForests and ExtraTrees can be fitted in parallel on many cores as each tree is built independently of the others. AdaBoost's samples are built sequentially and so do not use multiple cores.



Out:

```
DecisionTree with features [0, 1] has a score of 0.9266666666666666
RandomForest with 30 estimators with features [0, 1] has a score of 0.9266666666666666
ExtraTrees with 30 estimators with features [0, 1] has a score of 0.9266666666666666
AdaBoost with 30 estimators with features [0, 1] has a score of 0.84
DecisionTree with features [0, 2] has a score of 0.9933333333333333
RandomForest with 30 estimators with features [0, 2] has a score of 0.9933333333333333
ExtraTrees with 30 estimators with features [0, 2] has a score of 0.9933333333333333
AdaBoost with 30 estimators with features [0, 2] has a score of 0.9933333333333333
DecisionTree with features [2, 3] has a score of 0.9933333333333333
RandomForest with 30 estimators with features [2, 3] has a score of 0.9933333333333333
ExtraTrees with 30 estimators with features [2, 3] has a score of 0.9933333333333333
AdaBoost with 30 estimators with features [2, 3] has a score of 0.9933333333333333
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
```

(continues on next page)

(continued from previous page)

```
from sklearn.datasets import load_iris
from sklearn.ensemble import (RandomForestClassifier, ExtraTreesClassifier,
                              AdaBoostClassifier)
from sklearn.tree import DecisionTreeClassifier

# Parameters
n_classes = 3
n_estimators = 30
cmap = plt.cm.RdYlBu
plot_step = 0.02  # fine step width for decision surface contours
plot_step_coarser = 0.5  # step widths for coarse classifier guesses
RANDOM_SEED = 13  # fix the seed on each iteration

# Load data
iris = load_iris()

plot_idx = 1

models = [DecisionTreeClassifier(max_depth=None),
          RandomForestClassifier(n_estimators=n_estimators),
          ExtraTreesClassifier(n_estimators=n_estimators),
          AdaBoostClassifier(DecisionTreeClassifier(max_depth=3),
                             n_estimators=n_estimators)]

for pair in ([0, 1], [0, 2], [2, 3]):
    for model in models:
        # We only take the two corresponding features
        X = iris.data[:, pair]
        y = iris.target

        # Shuffle
        idx = np.arange(X.shape[0])
        np.random.seed(RANDOM_SEED)
        np.random.shuffle(idx)
        X = X[idx]
        y = y[idx]

        # Standardize
        mean = X.mean(axis=0)
        std = X.std(axis=0)
        X = (X - mean) / std

        # Train
        model.fit(X, y)

        scores = model.score(X, y)
        # Create a title for each column and the console by using str() and
        # slicing away useless parts of the string
        model_title = str(type(model)).split(
            ".")[-1][:-2][:-len("Classifier")]

        model_details = model_title
        if hasattr(model, "estimators_"):
            model_details += " with {} estimators".format(
                len(model.estimators_))
        print(model_details + " with features", pair,
```

(continues on next page)

(continued from previous page)

```
              "has a score of", scores)

        plt.subplot(3, 4, plot_idx)
        if plot_idx <= len(models):
            # Add a title at the top of each column
            plt.title(model_title, fontsize=9)

        # Now plot the decision boundary using a fine mesh as input to a
        # filled contour plot
        x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
        y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
        xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                             np.arange(y_min, y_max, plot_step))

        # Plot either a single DecisionTreeClassifier or alpha blend the
        # decision surfaces of the ensemble of classifiers
        if isinstance(model, DecisionTreeClassifier):
            Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
            Z = Z.reshape(xx.shape)
            cs = plt.contourf(xx, yy, Z, cmap=cmap)
        else:
            # Choose alpha blend level with respect to the number
            # of estimators
            # that are in use (noting that AdaBoost can use fewer estimators
            # than its maximum if it achieves a good enough fit early on)
            estimator_alpha = 1.0 / len(model.estimators_)
            for tree in model.estimators_:
                Z = tree.predict(np.c_[xx.ravel(), yy.ravel()])
                Z = Z.reshape(xx.shape)
                cs = plt.contourf(xx, yy, Z, alpha=estimator_alpha, cmap=cmap)

        # Build a coarser grid to plot a set of ensemble classifications
        # to show how these are different to what we see in the decision
        # surfaces. These points are regularly space and do not have a
        # black outline
        xx_coarser, yy_coarser = np.meshgrid(
            np.arange(x_min, x_max, plot_step_coarser),
            np.arange(y_min, y_max, plot_step_coarser))
        Z_points_coarser = model.predict(np.c_[xx_coarser.ravel(),
                                               yy_coarser.ravel()]
                                         ).reshape(xx_coarser.shape)
        cs_points = plt.scatter(xx_coarser, yy_coarser, s=15,
                                c=Z_points_coarser, cmap=cmap,
                                edgecolors="none")

        # Plot the training points, these are clustered together and have a
        # black outline
        plt.scatter(X[:, 0], X[:, 1], c=y,
                    cmap=ListedColormap(['r', 'y', 'b']),
                    edgecolor='k', s=20)
        plot_idx += 1  # move on to the next plot in sequence

plt.suptitle("Classifiers on feature subsets of the Iris dataset", fontsize=12)
plt.axis("tight")
plt.tight_layout(h_pad=0.2, w_pad=0.2, pad=2.5)
plt.show()
```

Total running time of the script: (0 minutes 6.692 seconds)

Estimated memory usage: 14 MB

6.12 Examples based on real world datasets

Applications to real world problems with some medium sized datasets or interactive user interface.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.12.1 Outlier detection on a real data set

This example illustrates the need for robust covariance estimation on a real data set. It is useful both for outlier detection and for a better understanding of the data structure.

We selected two sets of two variables from the Boston housing data set as an illustration of what kind of analysis can be done with several outlier detection tools. For the purpose of visualization, we are working with two-dimensional examples, but one should be aware that things are not so trivial in high-dimension, as it will be pointed out.

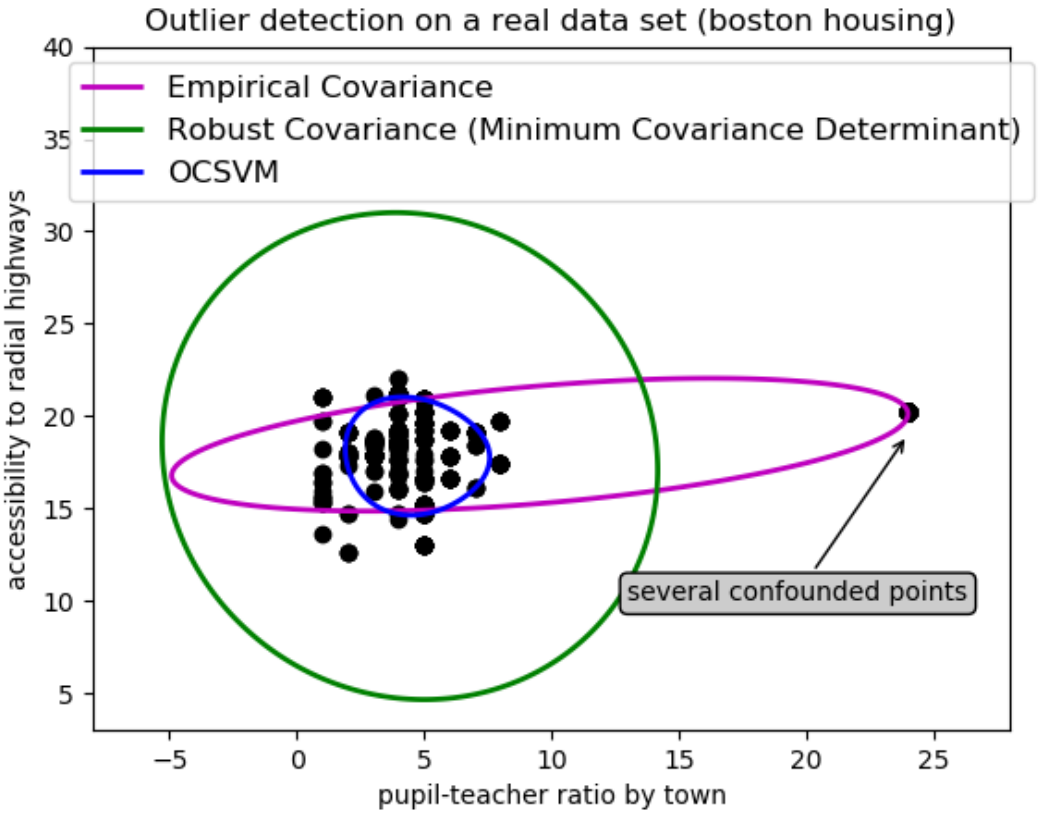
In both examples below, the main result is that the empirical covariance estimate, as a non-robust one, is highly influenced by the heterogeneous structure of the observations. Although the robust covariance estimate is able to focus on the main mode of the data distribution, it sticks to the assumption that the data should be Gaussian distributed, yielding some biased estimation of the data structure, but yet accurate to some extent. The One-Class SVM does not assume any parametric form of the data distribution and can therefore model the complex shape of the data much better.

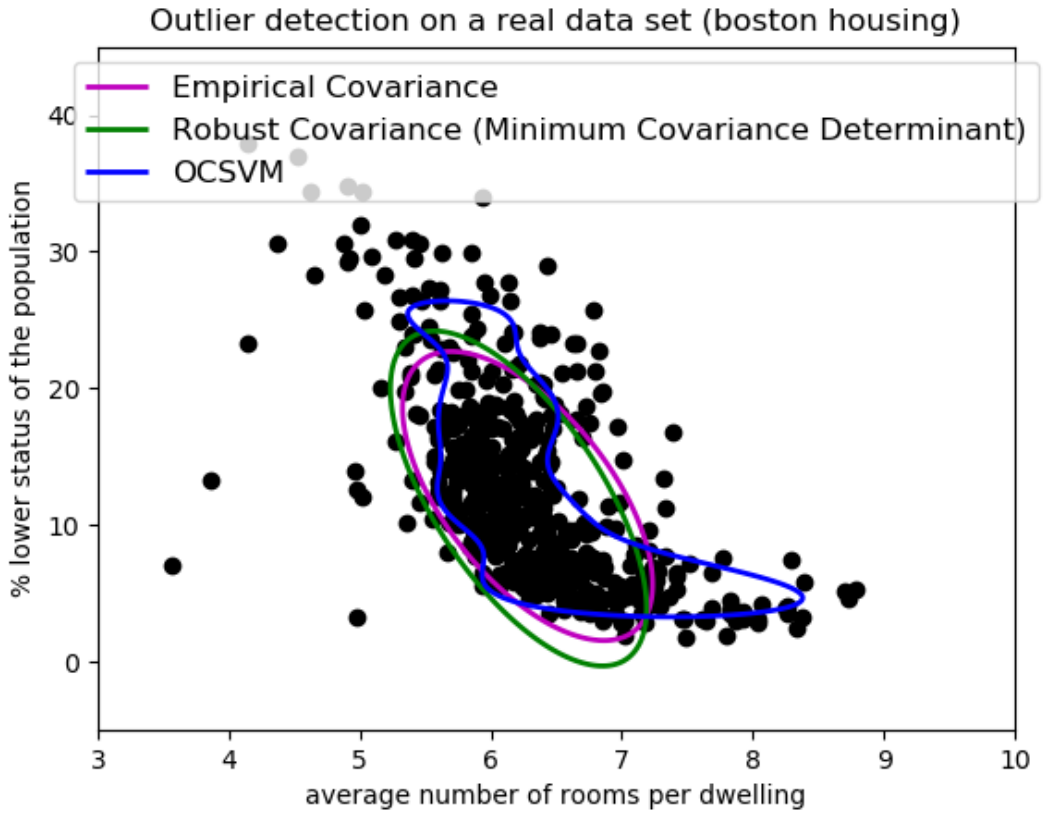
First example

The first example illustrates how robust covariance estimation can help concentrating on a relevant cluster when another one exists. Here, many observations are confounded into one and break down the empirical covariance estimation. Of course, some screening tools would have pointed out the presence of two clusters (Support Vector Machines, Gaussian Mixture Models, univariate outlier detection, ...). But had it been a high-dimensional example, none of these could be applied that easily.

Second example

The second example shows the ability of the Minimum Covariance Determinant robust estimator of covariance to concentrate on the main mode of the data distribution: the location seems to be well estimated, although the covariance is hard to estimate due to the banana-shaped distribution. Anyway, we can get rid of some outlying observations. The One-Class SVM is able to capture the real data structure, but the difficulty is to adjust its kernel bandwidth parameter so as to obtain a good compromise between the shape of the data scatter matrix and the risk of over-fitting the data.


Outlier detection on a real data set (boston housing)
Empirical Covariance
Robust Covariance (Minimum Covariance Determinant)
OCSVM
accessibility to radial highways
pupil-teacher ratio by town
several confounded points
40
35
30
25
20
15
10
5
−5
0
5
10
15
20
25




```
print(__doc__)

# Author: Virgile Fritsch <virgile.fritsch@inria.fr>
# License: BSD 3 clause

import numpy as np
from sklearn.covariance import EllipticEnvelope
from sklearn.svm import OneClassSVM
import matplotlib.pyplot as plt
import matplotlib.font_manager
from sklearn.datasets import load_boston

# Get data
X1 = load_boston()['data'][:, [8, 10]]  # two clusters
X2 = load_boston()['data'][:, [5, 12]]  # "banana"-shaped

# Define "classifiers" to be used
classifiers = {
    "Empirical Covariance": EllipticEnvelope(support_fraction=1.,
                                             contamination=0.261),
    "Robust Covariance (Minimum Covariance Determinant)":
    EllipticEnvelope(contamination=0.261),
    "OCSVM": OneClassSVM(nu=0.261, gamma=0.05)}
colors = ['m', 'g', 'b']
legend1 = {}
legend2 = {}
```

(continues on next page)

(continued from previous page)

```
# Learn a frontier for outlier detection with several classifiers
xx1, yy1 = np.meshgrid(np.linspace(-8, 28, 500), np.linspace(3, 40, 500))
xx2, yy2 = np.meshgrid(np.linspace(3, 10, 500), np.linspace(-5, 45, 500))
for i, (clf_name, clf) in enumerate(classifiers.items()):
    plt.figure(1)
    clf.fit(X1)
    Z1 = clf.decision_function(np.c_[xx1.ravel(), yy1.ravel()])
    Z1 = Z1.reshape(xx1.shape)
    legend1[clf_name] = plt.contour(
        xx1, yy1, Z1, levels=[0], linewidths=2, colors=colors[i])
    plt.figure(2)
    clf.fit(X2)
    Z2 = clf.decision_function(np.c_[xx2.ravel(), yy2.ravel()])
    Z2 = Z2.reshape(xx2.shape)
    legend2[clf_name] = plt.contour(
        xx2, yy2, Z2, levels=[0], linewidths=2, colors=colors[i])

legend1_values_list = list(legend1.values())
legend1_keys_list = list(legend1.keys())

# Plot the results (= shape of the data points cloud)
plt.figure(1)  # two clusters
plt.title("Outlier detection on a real data set (boston housing)")
plt.scatter(X1[:, 0], X1[:, 1], color='black')
bbox_args = dict(boxstyle="round", fc="0.8")
arrow_args = dict(arrowstyle="->")
plt.annotate("several confounded points", xy=(24, 19),
             xycoords="data", textcoords="data",
             xytext=(13, 10), bbox=bbox_args, arrowprops=arrow_args)
plt.xlim((xx1.min(), xx1.max()))
plt.ylim((yy1.min(), yy1.max()))
plt.legend((legend1_values_list[0].collections[0],
            legend1_values_list[1].collections[0],
            legend1_values_list[2].collections[0]),
           (legend1_keys_list[0], legend1_keys_list[1], legend1_keys_list[2]),
           loc="upper center",
           prop=matplotlib.font_manager.FontProperties(size=12))
plt.ylabel("accessibility to radial highways")
plt.xlabel("pupil-teacher ratio by town")

legend2_values_list = list(legend2.values())
legend2_keys_list = list(legend2.keys())

plt.figure(2)  # "banana" shape
plt.title("Outlier detection on a real data set (boston housing)")
plt.scatter(X2[:, 0], X2[:, 1], color='black')
plt.xlim((xx2.min(), xx2.max()))
plt.ylim((yy2.min(), yy2.max()))
plt.legend((legend2_values_list[0].collections[0],
            legend2_values_list[1].collections[0],
            legend2_values_list[2].collections[0]),
           (legend2_keys_list[0], legend2_keys_list[1], legend2_keys_list[2]),
           loc="upper center",
           prop=matplotlib.font_manager.FontProperties(size=12))
plt.ylabel("% lower status of the population")
plt.xlabel("average number of rooms per dwelling")
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

Total running time of the script: (0 minutes 3.275 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

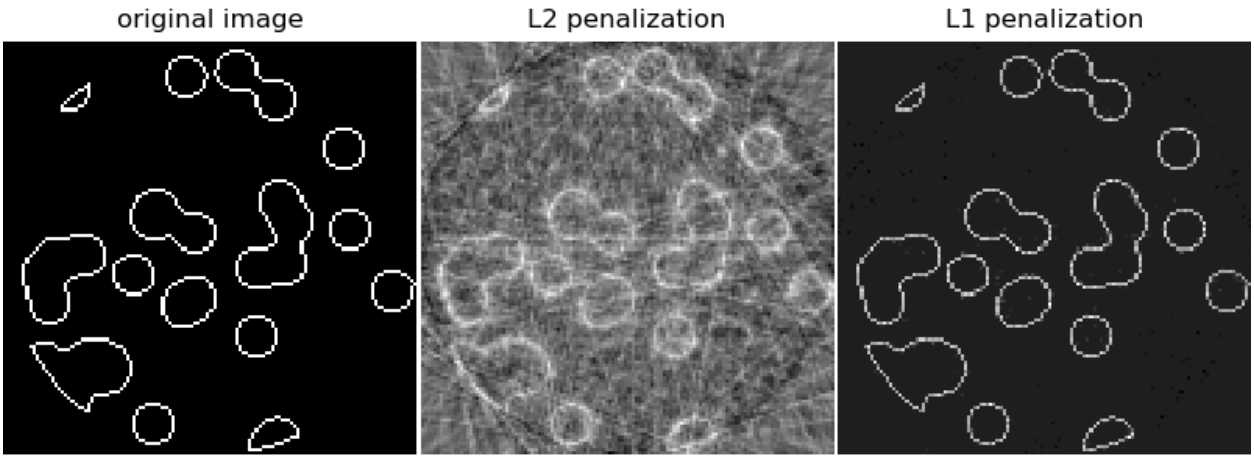
6.12.2 Compressive sensing: tomography reconstruction with L1 prior (Lasso)

This example shows the reconstruction of an image from a set of parallel projections, acquired along different angles. Such a dataset is acquired in **computed tomography** (CT).

Without any prior information on the sample, the number of projections required to reconstruct the image is of the order of the linear size `l` of the image (in pixels). For simplicity we consider here a sparse image, where only pixels on the boundary of objects have a non-zero value. Such data could correspond for example to a cellular material. Note however that most images are sparse in a different basis, such as the Haar wavelets. Only `l/7` projections are acquired, therefore it is necessary to use prior information available on the sample (its sparsity): this is an example of **compressive sensing**.

The tomography projection operation is a linear transformation. In addition to the data-fidelity term corresponding to a linear regression, we penalize the L1 norm of the image to account for its sparsity. The resulting optimization problem is called the *Lasso*. We use the class `sklearn.linear_model.Lasso`, that uses the coordinate descent algorithm. Importantly, this implementation is more computationally efficient on a sparse matrix, than the projection operator used here.

The reconstruction with L1 penalization gives a result with zero error (all pixels are successfully labeled with 0 or 1), even if noise was added to the projections. In comparison, an L2 penalization (`sklearn.linear_model.Ridge`) produces a large number of labeling errors for the pixels. Important artifacts are observed on the reconstructed image, contrary to the L1 penalization. Note in particular the circular artifact separating the pixels in the corners, that have contributed to fewer projections than the central disk.




```
print(__doc__)

# Author: Emmanuelle Gouillart <emmanuelle.gouillart@nsup.org>
```

(continues on next page)

(continued from previous page)

```
# License: BSD 3 clause

import numpy as np
from scipy import sparse
from scipy import ndimage
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
import matplotlib.pyplot as plt


def _weights(x, dx=1, orig=0):
    x = np.ravel(x)
    floor_x = np.floor((x - orig) / dx).astype(np.int64)
    alpha = (x - orig - floor_x * dx) / dx
    return np.hstack((floor_x, floor_x + 1)), np.hstack((1 - alpha, alpha))


def _generate_center_coordinates(l_x):
    X, Y = np.mgrid[:l_x, :l_x].astype(np.float64)
    center = l_x / 2.
    X += 0.5 - center
    Y += 0.5 - center
    return X, Y


def build_projection_operator(l_x, n_dir):
    """ Compute the tomography design matrix.

    Parameters
    ----------

    l_x : int
        linear size of image array

    n_dir : int
        number of angles at which projections are acquired.

    Returns
    -------
    p : sparse matrix of shape (n_dir l_x, l_x**2)
    """
    X, Y = _generate_center_coordinates(l_x)
    angles = np.linspace(0, np.pi, n_dir, endpoint=False)
    data_inds, weights, camera_inds = [], [], []
    data_unravel_indices = np.arange(l_x ** 2)
    data_unravel_indices = np.hstack((data_unravel_indices,
                                      data_unravel_indices))
    for i, angle in enumerate(angles):
        Xrot = np.cos(angle) * X - np.sin(angle) * Y
        inds, w = _weights(Xrot, dx=1, orig=X.min())
        mask = np.logical_and(inds >= 0, inds < l_x)
        weights += list(w[mask])
        camera_inds += list(inds[mask] + i * l_x)
        data_inds += list(data_unravel_indices[mask])
    proj_operator = sparse.coo_matrix((weights, (camera_inds, data_inds)))
    return proj_operator
```

(continues on next page)

(continued from previous page)

```
def generate_synthetic_data():
    """ Synthetic binary data """
    rs = np.random.RandomState(0)
    n_pts = 36
    x, y = np.ogrid[0:l, 0:l]
    mask_outer = (x - l / 2.) ** 2 + (y - l / 2.) ** 2 < (l / 2.) ** 2
    mask = np.zeros((l, l))
    points = l * rs.rand(2, n_pts)
    mask[(points[0]).astype(np.int), (points[1]).astype(np.int)] = 1
    mask = ndimage.gaussian_filter(mask, sigma=l / n_pts)
    res = np.logical_and(mask > mask.mean(), mask_outer)
    return np.logical_xor(res, ndimage.binary_erosion(res))


# Generate synthetic images, and projections
l = 128
proj_operator = build_projection_operator(l, l // 7)
data = generate_synthetic_data()
proj = proj_operator * data.ravel()[:, np.newaxis]
proj += 0.15 * np.random.randn(*proj.shape)

# Reconstruction with L2 (Ridge) penalization
rgr_ridge = Ridge(alpha=0.2)
rgr_ridge.fit(proj_operator, proj.ravel())
rec_l2 = rgr_ridge.coef_.reshape(l, l)

# Reconstruction with L1 (Lasso) penalization
# the best value of alpha was determined using cross validation
# with LassoCV
rgr_lasso = Lasso(alpha=0.001)
rgr_lasso.fit(proj_operator, proj.ravel())
rec_l1 = rgr_lasso.coef_.reshape(l, l)

plt.figure(figsize=(8, 3.3))
plt.subplot(131)
plt.imshow(data, cmap=plt.cm.gray, interpolation='nearest')
plt.axis('off')
plt.title('original image')
plt.subplot(132)
plt.imshow(rec_l2, cmap=plt.cm.gray, interpolation='nearest')
plt.title('L2 penalization')
plt.axis('off')
plt.subplot(133)
plt.imshow(rec_l1, cmap=plt.cm.gray, interpolation='nearest')
plt.title('L1 penalization')
plt.axis('off')

plt.subplots_adjust(hspace=0.01, wspace=0.01, top=1, bottom=0, left=0,
                    right=1)

plt.show()
```

Total running time of the script: (0 minutes 8.929 seconds)

Estimated memory usage: 40 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.12.3 Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation

This is an example of applying *sklearn.decomposition.NMF* and *sklearn.decomposition.LatentDirichletAllocation* on a corpus of documents and extract additive models of the topic structure of the corpus. The output is a list of topics, each represented as a list of terms (weights are not shown).

Non-negative Matrix Factorization is applied with two different objective functions: the Frobenius norm, and the generalized Kullback-Leibler divergence. The latter is equivalent to Probabilistic Latent Semantic Indexing.

The default parameters (n_samples / n_features / n_components) should make the example runnable in a couple of tens of seconds. You can try to increase the dimensions of the problem, but be aware that the time complexity is polynomial in NMF. In LDA, the time complexity is proportional to (n_samples * iterations).

Out:

```
Loading dataset...
done in 1.111s.
Extracting tf-idf features for NMF...
done in 0.307s.
Extracting tf features for LDA...
done in 0.280s.

Fitting the NMF model (Frobenius norm) with tf-idf features, n_samples=2000 and n_
↪features=1000...
done in 0.233s.

Topics in NMF model (Frobenius norm):
Topic #0: just people don think like know time good make way really say right ve want␣
↪did ll new use years
Topic #1: windows use dos using window program os drivers application help software␣
↪pc running ms screen files version card code work
Topic #2: god jesus bible faith christian christ christians does heaven sin believe␣
↪lord life church mary atheism belief human love religion
Topic #3: thanks know does mail advance hi info interested email anybody looking card␣
↪help like appreciated information send list video need
Topic #4: car cars tires miles 00 new engine insurance price condition oil power␣
↪speed good 000 brake year models used bought
Topic #5: edu soon com send university internet mit ftp mail cc pub article␣
↪information hope program mac email home contact blood
Topic #6: file problem files format win sound ftp pub read save site help image␣
↪available create copy running memory self version
Topic #7: game team games year win play season players nhl runs goal hockey toronto␣
↪division flyers player defense leafs bad teams
Topic #8: drive drives hard disk floppy software card mac computer power scsi␣
↪controller apple mb 00 pc rom sale problem internal
Topic #9: key chip clipper keys encryption government public use secure enforcement␣
↪phone nsa communications law encrypted security clinton used legal standard

Fitting the NMF model (generalized Kullback-Leibler divergence) with tf-idf features,␣
↪n_samples=2000 and n_features=1000...
done in 0.814s.
```

(continues on next page)

(continued from previous page)

```
Topics in NMF model (generalized Kullback-Leibler divergence):
Topic #0: people don just like think did say time make know really right said things␣
↪way ve course didn question probably
Topic #1: windows help thanks using hi looking info video dos pc does anybody ftp␣
↪appreciated mail know advance available use card
Topic #2: god does jesus true book christian bible christians religion faith believe␣
↪life church christ says know read exist lord people
Topic #3: thanks know bike interested mail like new car edu heard just price list␣
↪email hear want cars thing sounds reply
Topic #4: 10 00 sale time power 12 new 15 year 30 offer condition 14 16 model 11␣
↪monitor 100 old 25
Topic #5: space government number public data states earth security water research␣
↪nasa general 1993 phone information science technology provide blood internet
Topic #6: edu file com program soon try window problem remember files sun send␣
↪library article mike wrong think code win manager
Topic #7: game team year games play win season points world division won players nhl␣
↪flyers toronto case cubs teams ll record
Topic #8: drive think hard software disk drives apple computer mac need scsi card don␣
↪problem read floppy post cable going ii
Topic #9: use good just key chip got like ll way clipper doesn keys don better speed␣
↪stuff want sure going need

Fitting LDA models with tf features, n_samples=2000 and n_features=1000...
done in 3.504s.

Topics in LDA model:
Topic #0: edu com mail send graphics ftp pub available contact university list faq ca␣
↪information cs 1993 program sun uk mit
Topic #1: don like just know think ve way use right good going make sure ll point got␣
↪need really time doesn
Topic #2: christian think atheism faith pittsburgh new bible radio games alt lot just␣
↪religion like book read play time subject believe
Topic #3: drive disk windows thanks use card drives hard version pc software file␣
↪using scsi help does new dos controller 16
Topic #4: hiv health aids disease april medical care research 1993 light information␣
↪study national service test led 10 page new drug
Topic #5: god people does just good don jesus say israel way life know true fact time␣
↪law want believe make think
Topic #6: 55 10 11 18 15 team game 19 period play 23 12 13 flyers 20 25 22 17 24 16
Topic #7: car year just cars new engine like bike good oil insurance better tires 000␣
↪thing speed model brake driving performance
Topic #8: people said did just didn know time like went think children came come don␣
↪took years say dead told started
Topic #9: key space law government public use encryption earth section security moon␣
↪probe enforcement keys states lunar military crime surface technology
```

```
# Author: Olivier Grisel <olivier.grisel@ensta.org>
#         Lars Buitinck
#         Chyi-Kwei Yau <chyikwei.yau@gmail.com>
# License: BSD 3 clause
```

(continues on next page)

(continued from previous page)

```
from time import time

from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.decomposition import NMF, LatentDirichletAllocation
from sklearn.datasets import fetch_20newsgroups

n_samples = 2000
n_features = 1000
n_components = 10
n_top_words = 20


def print_top_words(model, feature_names, n_top_words):
    for topic_idx, topic in enumerate(model.components_):
        message = "Topic #%d: " % topic_idx
        message += " ".join([feature_names[i]
                             for i in topic.argsort()[:-n_top_words - 1:-1]])
        print(message)
    print()


# Load the 20 newsgroups dataset and vectorize it. We use a few heuristics
# to filter out useless terms early on: the posts are stripped of headers,
# footers and quoted replies, and common English words, words occurring in
# only one document or in at least 95% of the documents are removed.

print("Loading dataset...")
t0 = time()
data, _ = fetch_20newsgroups(shuffle=True, random_state=1,
                             remove=('headers', 'footers', 'quotes'),
                             return_X_y=True)
data_samples = data[:n_samples]
print("done in %0.3fs." % (time() - t0))

# Use tf-idf features for NMF.
print("Extracting tf-idf features for NMF...")
tfidf_vectorizer = TfidfVectorizer(max_df=0.95, min_df=2,
                                   max_features=n_features,
                                   stop_words='english')
t0 = time()
tfidf = tfidf_vectorizer.fit_transform(data_samples)
print("done in %0.3fs." % (time() - t0))

# Use tf (raw term count) features for LDA.
print("Extracting tf features for LDA...")
tf_vectorizer = CountVectorizer(max_df=0.95, min_df=2,
                                max_features=n_features,
                                stop_words='english')
t0 = time()
tf = tf_vectorizer.fit_transform(data_samples)
print("done in %0.3fs." % (time() - t0))
print()

# Fit the NMF model
print("Fitting the NMF model (Frobenius norm) with tf-idf features, "
      "n_samples=%d and n_features=%d..."
      % (n_samples, n_features))
```

(continues on next page)

(continued from previous page)

```
t0 = time()
nmf = NMF(n_components=n_components, random_state=1,
          alpha=.1, l1_ratio=.5).fit(tfidf)
print("done in %0.3fs." % (time() - t0))

print("\nTopics in NMF model (Frobenius norm):")
tfidf_feature_names = tfidf_vectorizer.get_feature_names()
print_top_words(nmf, tfidf_feature_names, n_top_words)

# Fit the NMF model
print("Fitting the NMF model (generalized Kullback-Leibler divergence) with "
      "tf-idf features, n_samples=%d and n_features=%d..."
      % (n_samples, n_features))
t0 = time()
nmf = NMF(n_components=n_components, random_state=1,
          beta_loss='kullback-leibler', solver='mu', max_iter=1000, alpha=.1,
          l1_ratio=.5).fit(tfidf)
print("done in %0.3fs." % (time() - t0))

print("\nTopics in NMF model (generalized Kullback-Leibler divergence):")
tfidf_feature_names = tfidf_vectorizer.get_feature_names()
print_top_words(nmf, tfidf_feature_names, n_top_words)

print("Fitting LDA models with tf features, "
      "n_samples=%d and n_features=%d..."
      % (n_samples, n_features))
lda = LatentDirichletAllocation(n_components=n_components, max_iter=5,
                                learning_method='online',
                                learning_offset=50.,
                                random_state=0)
t0 = time()
lda.fit(tf)
print("done in %0.3fs." % (time() - t0))

print("\nTopics in LDA model:")
tf_feature_names = tf_vectorizer.get_feature_names()
print_top_words(lda, tf_feature_names, n_top_words)
```

Total running time of the script: (0 minutes 6.446 seconds)

Estimated memory usage: 50 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

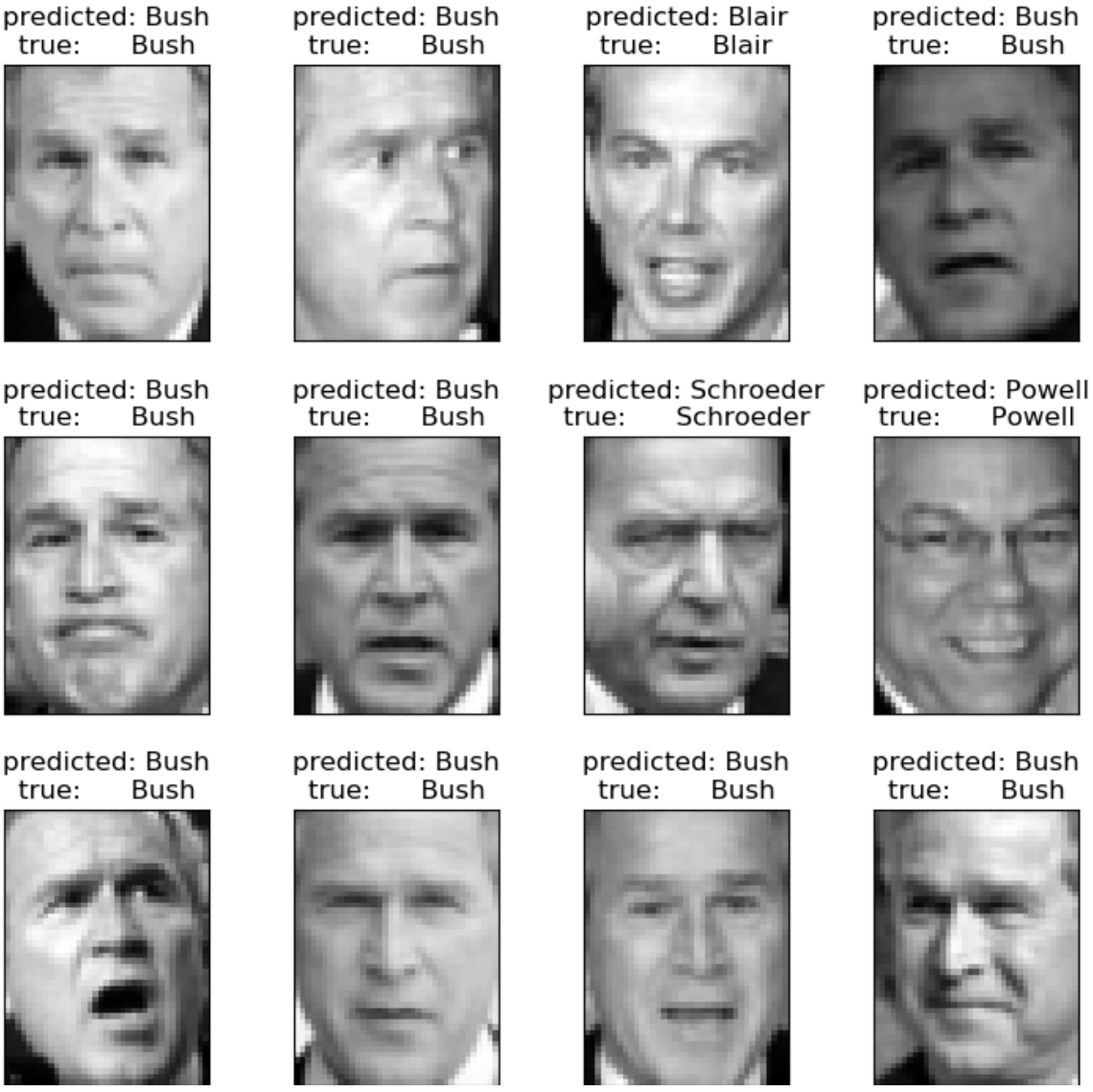
6.12.4 Faces recognition example using eigenfaces and SVMs

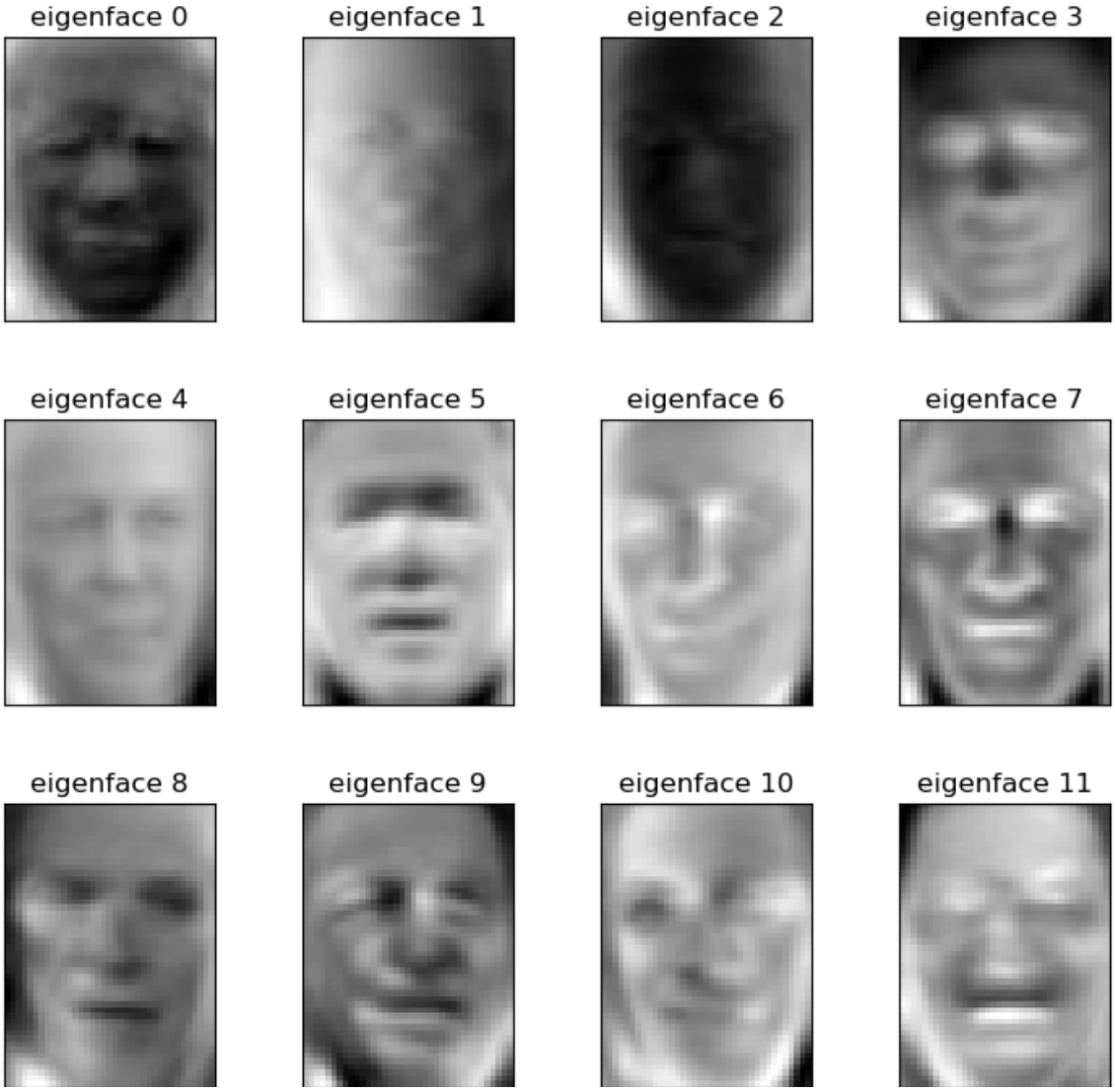
The dataset used in this example is a preprocessed excerpt of the "Labeled Faces in the Wild", aka LFW:

http://vis-www.cs.umass.edu/lfw/lfw-funneled.tgz (233MB)

Expected results for the top 5 most represented people in the dataset:

Ariel Sharon	0.67	0.92	0.77	13
Colin Powell	0.75	0.78	0.76	60
Donald Rumsfeld	0.78	0.67	0.72	27
George W Bush	0.86	0.86	0.86	146
Gerhard Schroeder	0.76	0.76	0.76	25
Hugo Chavez	0.67	0.67	0.67	15
Tony Blair	0.81	0.69	0.75	36
avg / total	0.80	0.80	0.80	322



- 

Out:

```
Total dataset size:
n_samples: 1288
n_features: 1850
n_classes: 7
Extracting the top 150 eigenfaces from 966 faces
done in 0.090s
Projecting the input data on the eigenfaces orthonormal basis
done in 0.008s
Fitting the classifier to the training set
done in 37.394s
Best estimator found by grid search:
SVC(C=1000.0, class_weight='balanced', gamma=0.005)
Predicting people's names on the test set
done in 0.068s
                   precision    recall  f1-score   support
```

(continues on next page)

(continued from previous page)

```
     Ariel Sharon       0.75      0.46      0.57        13
     Colin Powell       0.79      0.87      0.83        60
  Donald Rumsfeld       0.89      0.63      0.74        27
    George W Bush       0.83      0.98      0.90       146
Gerhard Schroeder       0.95      0.80      0.87        25
      Hugo Chavez       1.00      0.47      0.64        15
       Tony Blair       0.97      0.78      0.86        36

         accuracy                           0.85       322
        macro avg       0.88      0.71      0.77       322
     weighted avg       0.86      0.85      0.84       322

[[  6   3   0   4   0   0   0]
 [  1  52   0   7   0   0   0]
 [  1   2  17   7   0   0   0]
 [  0   3   0 143   0   0   0]
 [  0   1   0   3  20   0   1]
 [  0   4   0   3   1   7   0]
 [  0   1   2   5   0   0  28]]
```

```
from time import time
import logging
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.datasets import fetch_lfw_people
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.decomposition import PCA
from sklearn.svm import SVC


print(__doc__)

# Display progress logs on stdout
logging.basicConfig(level=logging.INFO, format='%(asctime)s %(message)s')


# #############################################################################
# Download the data, if not already on disk and load it as numpy arrays

lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)

# introspect the images arrays to find the shapes (for plotting)
n_samples, h, w = lfw_people.images.shape

# for machine learning we use the 2 data directly (as relative pixel
# positions info is ignored by this model)
X = lfw_people.data
```

(continues on next page)

(continued from previous page)

```
n_features = X.shape[1]

# the label to predict is the id of the person
y = lfw_people.target
target_names = lfw_people.target_names
n_classes = target_names.shape[0]

print("Total dataset size:")
print("n_samples: %d" % n_samples)
print("n_features: %d" % n_features)
print("n_classes: %d" % n_classes)


# #############################################################################
# Split into a training set and a test set using a stratified k fold

# split into a training and testing set
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42)


# #############################################################################
# Compute a PCA (eigenfaces) on the face dataset (treated as unlabeled
# dataset): unsupervised feature extraction / dimensionality reduction
n_components = 150

print("Extracting the top %d eigenfaces from %d faces"
      % (n_components, X_train.shape[0]))
t0 = time()
pca = PCA(n_components=n_components, svd_solver='randomized',
          whiten=True).fit(X_train)
print("done in %0.3fs" % (time() - t0))

eigenfaces = pca.components_.reshape((n_components, h, w))

print("Projecting the input data on the eigenfaces orthonormal basis")
t0 = time()
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
print("done in %0.3fs" % (time() - t0))


# #############################################################################
# Train a SVM classification model

print("Fitting the classifier to the training set")
t0 = time()
param_grid = {'C': [1e3, 5e3, 1e4, 5e4, 1e5],
              'gamma': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1], }
clf = GridSearchCV(
    SVC(kernel='rbf', class_weight='balanced'), param_grid
)
clf = clf.fit(X_train_pca, y_train)
print("done in %0.3fs" % (time() - t0))
print("Best estimator found by grid search:")
print(clf.best_estimator_)
```

(continues on next page)

(continued from previous page)

```
# ##############################################################################
# Quantitative evaluation of the model quality on the test set

print("Predicting people's names on the test set")
t0 = time()
y_pred = clf.predict(X_test_pca)
print("done in %0.3fs" % (time() - t0))

print(classification_report(y_test, y_pred, target_names=target_names))
print(confusion_matrix(y_test, y_pred, labels=range(n_classes)))


# ##############################################################################
# Qualitative evaluation of the predictions using matplotlib

def plot_gallery(images, titles, h, w, n_row=3, n_col=4):
    """Helper function to plot a gallery of portraits"""
    plt.figure(figsize=(1.8 * n_col, 2.4 * n_row))
    plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray)
        plt.title(titles[i], size=12)
        plt.xticks(())
        plt.yticks(())


# plot the result of the prediction on a portion of the test set

def title(y_pred, y_test, target_names, i):
    pred_name = target_names[y_pred[i]].rsplit(' ', 1)[-1]
    true_name = target_names[y_test[i]].rsplit(' ', 1)[-1]
    return 'predicted: %s\ntrue:      %s' % (pred_name, true_name)

prediction_titles = [title(y_pred, y_test, target_names, i)
                     for i in range(y_pred.shape[0])]

plot_gallery(X_test, prediction_titles, h, w)

# plot the gallery of the most significative eigenfaces

eigenface_titles = ["eigenface %d" % i for i in range(eigenfaces.shape[0])]
plot_gallery(eigenfaces, eigenface_titles, h, w)

plt.show()
```

Total running time of the script: (0 minutes 38.265 seconds)

Estimated memory usage: 9 MB

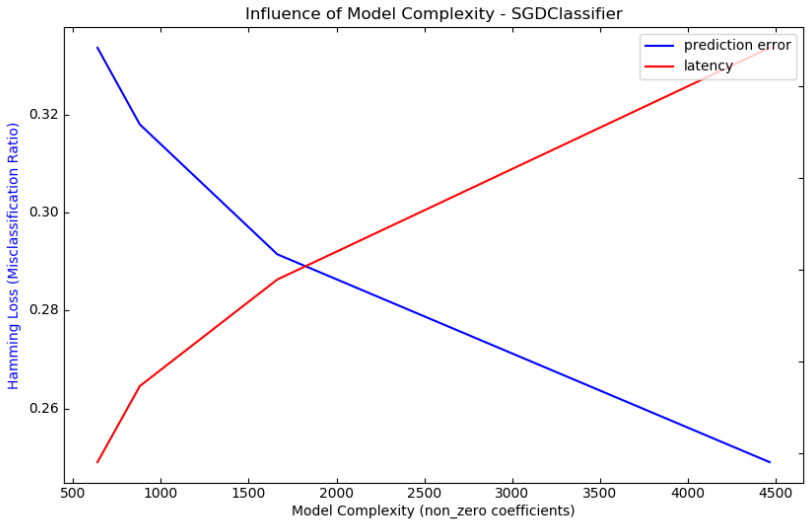
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.12.5 Model Complexity Influence

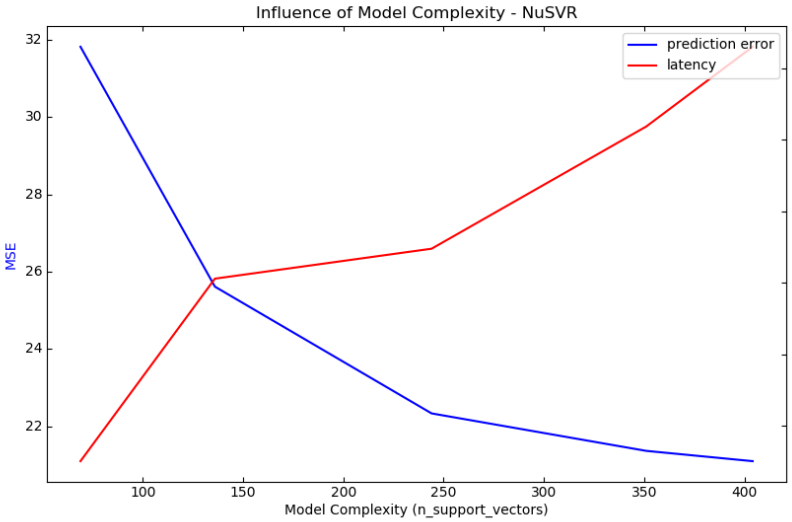
Demonstrate how model complexity influences both prediction accuracy and computational performance.

The dataset is the Boston Housing dataset (resp. 20 Newsgroups) for regression (resp. classification).

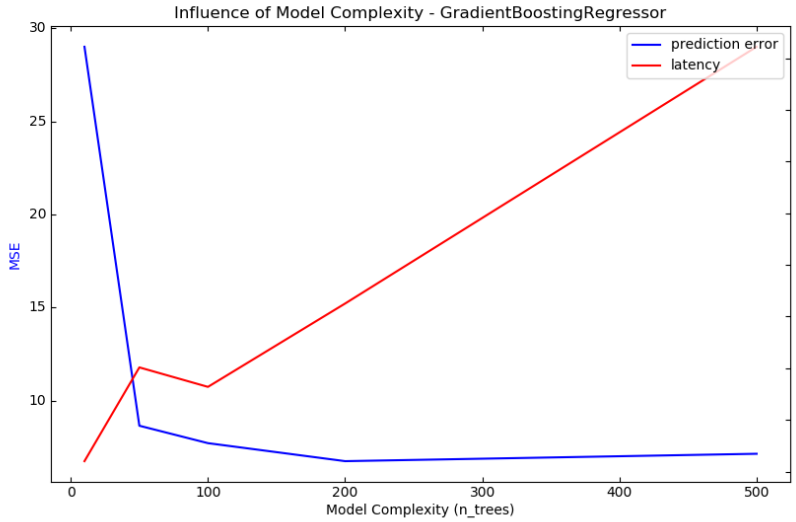
For each class of models we make the model complexity vary through the choice of relevant model parameters and measure the influence on both computational performance (latency) and predictive power (MSE or Hamming Loss).



-



-



-

Out:

```
Benchmarking SGDClassifier(alpha=0.001, l1_ratio=0.25, loss='modified_huber',
              penalty='elasticnet')
Complexity: 4466 | Hamming Loss (Misclassification Ratio): 0.2491 | Pred. Time: 0.
↪020835s

Benchmarking SGDClassifier(alpha=0.001, l1_ratio=0.5, loss='modified_huber',
              penalty='elasticnet')
Complexity: 1663 | Hamming Loss (Misclassification Ratio): 0.2915 | Pred. Time: 0.
↪015789s

Benchmarking SGDClassifier(alpha=0.001, l1_ratio=0.75, loss='modified_huber',
              penalty='elasticnet')
Complexity: 880 | Hamming Loss (Misclassification Ratio): 0.3180 | Pred. Time: 0.
↪013469s

Benchmarking SGDClassifier(alpha=0.001, l1_ratio=0.9, loss='modified_huber',
              penalty='elasticnet')
Complexity: 639 | Hamming Loss (Misclassification Ratio): 0.3337 | Pred. Time: 0.
↪011812s

Benchmarking NuSVR(C=1000.0, gamma=3.0517578125e-05, nu=0.1)
Complexity: 69 | MSE: 31.8139 | Pred. Time: 0.000301s

Benchmarking NuSVR(C=1000.0, gamma=3.0517578125e-05, nu=0.25)
Complexity: 136 | MSE: 25.6140 | Pred. Time: 0.000811s

Benchmarking NuSVR(C=1000.0, gamma=3.0517578125e-05)
Complexity: 244 | MSE: 22.3375 | Pred. Time: 0.000895s

Benchmarking NuSVR(C=1000.0, gamma=3.0517578125e-05, nu=0.75)
Complexity: 351 | MSE: 21.3688 | Pred. Time: 0.001237s

Benchmarking NuSVR(C=1000.0, gamma=3.0517578125e-05, nu=0.9)
Complexity: 404 | MSE: 21.1033 | Pred. Time: 0.001460s
```

(continues on next page)

(continued from previous page)

```
Benchmarking GradientBoostingRegressor(n_estimators=10)
Complexity: 10 | MSE: 29.0148 | Pred. Time: 0.000120s

Benchmarking GradientBoostingRegressor(n_estimators=50)
Complexity: 50 | MSE: 8.6545 | Pred. Time: 0.000302s

Benchmarking GradientBoostingRegressor()
Complexity: 100 | MSE: 7.7179 | Pred. Time: 0.000264s

Benchmarking GradientBoostingRegressor(n_estimators=200)
Complexity: 200 | MSE: 6.7507 | Pred. Time: 0.000425s

Benchmarking GradientBoostingRegressor(n_estimators=500)
Complexity: 500 | MSE: 7.1471 | Pred. Time: 0.000922s
```

```
print(__doc__)

# Author: Eustache Diemert <eustache@diemert.fr>
# License: BSD 3 clause

import time
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.parasite_axes import host_subplot
from mpl_toolkits.axisartist.axislines import Axes
from scipy.sparse.csr import csr_matrix

from sklearn import datasets
from sklearn.utils import shuffle
from sklearn.metrics import mean_squared_error
from sklearn.svm import NuSVR
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import hamming_loss

# ##############################################################################
# Routines


# Initialize random generator
np.random.seed(0)


def generate_data(case, sparse=False):
    """Generate regression/classification data."""
    if case == 'regression':
        X, y = datasets.load_boston(return_X_y=True)
    elif case == 'classification':
        X, y = datasets.fetch_20newsgroups_vectorized(subset='all',
                                                      return_X_y=True)
    X, y = shuffle(X, y)
```

(continues on next page)

(continued from previous page)

```
    offset = int(X.shape[0] * 0.8)
    X_train, y_train = X[:offset], y[:offset]
    X_test, y_test = X[offset:], y[offset:]
    if sparse:
        X_train = csr_matrix(X_train)
        X_test = csr_matrix(X_test)
    else:
        X_train = np.array(X_train)
        X_test = np.array(X_test)
    y_test = np.array(y_test)
    y_train = np.array(y_train)
    data = {'X_train': X_train, 'X_test': X_test, 'y_train': y_train,
            'y_test': y_test}
    return data


def benchmark_influence(conf):
    """
    Benchmark influence of :changing_param: on both MSE and latency.
    """
    prediction_times = []
    prediction_powers = []
    complexities = []
    for param_value in conf['changing_param_values']:
        conf['tuned_params'][conf['changing_param']] = param_value
        estimator = conf['estimator'](**conf['tuned_params'])
        print("Benchmarking %s" % estimator)
        estimator.fit(conf['data']['X_train'], conf['data']['y_train'])
        conf['postfit_hook'](estimator)
        complexity = conf['complexity_computer'](estimator)
        complexities.append(complexity)
        start_time = time.time()
        for _ in range(conf['n_samples']):
            y_pred = estimator.predict(conf['data']['X_test'])
        elapsed_time = (time.time() - start_time) / float(conf['n_samples'])
        prediction_times.append(elapsed_time)
        pred_score = conf['prediction_performance_computer'](
            conf['data']['y_test'], y_pred)
        prediction_powers.append(pred_score)
        print("Complexity: %d | %s: %.4f | Pred. Time: %fs\n" % (
            complexity, conf['prediction_performance_label'], pred_score,
            elapsed_time))
    return prediction_powers, prediction_times, complexities


def plot_influence(conf, mse_values, prediction_times, complexities):
    """
    Plot influence of model complexity on both accuracy and latency.
    """
    plt.figure(figsize=(12, 6))
    host = host_subplot(111, axes_class=Axes)
    plt.subplots_adjust(right=0.75)
    par1 = host.twinx()
    host.set_xlabel('Model Complexity (%s)' % conf['complexity_label'])
    y1_label = conf['prediction_performance_label']
    y2_label = "Time (s)"
    host.set_ylabel(y1_label)
```

(continues on next page)

(continued from previous page)

```
    par1.set_ylabel(y2_label)
    p1, = host.plot(complexities, mse_values, 'b-', label="prediction error")
    p2, = par1.plot(complexities, prediction_times, 'r-',
                    label="latency")
    host.legend(loc='upper right')
    host.axis["left"].label.set_color(p1.get_color())
    par1.axis["right"].label.set_color(p2.get_color())
    plt.title('Influence of Model Complexity - %s' % conf['estimator'].__name__)
    plt.show()


def _count_nonzero_coefficients(estimator):
    a = estimator.coef_.toarray()
    return np.count_nonzero(a)

# ##############################################################################
# Main code
regression_data = generate_data('regression')
classification_data = generate_data('classification', sparse=True)
configurations = [
    {'estimator': SGDClassifier,
     'tuned_params': {'penalty': 'elasticnet', 'alpha': 0.001, 'loss':
                      'modified_huber', 'fit_intercept': True, 'tol': 1e-3},
     'changing_param': 'l1_ratio',
     'changing_param_values': [0.25, 0.5, 0.75, 0.9],
     'complexity_label': 'non_zero coefficients',
     'complexity_computer': _count_nonzero_coefficients,
     'prediction_performance_computer': hamming_loss,
     'prediction_performance_label': 'Hamming Loss (Misclassification Ratio)',
     'postfit_hook': lambda x: x.sparsify(),
     'data': classification_data,
     'n_samples': 30},
    {'estimator': NuSVR,
     'tuned_params': {'C': 1e3, 'gamma': 2 ** -15},
     'changing_param': 'nu',
     'changing_param_values': [0.1, 0.25, 0.5, 0.75, 0.9],
     'complexity_label': 'n_support_vectors',
     'complexity_computer': lambda x: len(x.support_vectors_),
     'data': regression_data,
     'postfit_hook': lambda x: x,
     'prediction_performance_computer': mean_squared_error,
     'prediction_performance_label': 'MSE',
     'n_samples': 30},
    {'estimator': GradientBoostingRegressor,
     'tuned_params': {'loss': 'ls'},
     'changing_param': 'n_estimators',
     'changing_param_values': [10, 50, 100, 200, 500],
     'complexity_label': 'n_trees',
     'complexity_computer': lambda x: x.n_estimators,
     'data': regression_data,
     'postfit_hook': lambda x: x,
     'prediction_performance_computer': mean_squared_error,
     'prediction_performance_label': 'MSE',
     'n_samples': 30},
]
for conf in configurations:
    prediction_performances, prediction_times, complexities = \
```

(continues on next page)

(continued from previous page)

```
        benchmark_influence(conf)
    plot_influence(conf, prediction_performances, prediction_times,
                   complexities)
```

Total running time of the script: (0 minutes 20.261 seconds)

Estimated memory usage: 60 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.12.6 Visualizing the stock market structure

This example employs several unsupervised learning techniques to extract the stock market structure from variations in historical quotes.

The quantity that we use is the daily variation in quote price: quotes that are linked tend to cofluctuate during a day.

Learning a graph structure

We use sparse inverse covariance estimation to find which quotes are correlated conditionally on the others. Specifically, sparse inverse covariance gives us a graph, that is a list of connection. For each symbol, the symbols that it is connected too are those useful to explain its fluctuations.

Clustering

We use clustering to group together quotes that behave similarly. Here, amongst the *various clustering techniques* available in the scikit-learn, we use *Affinity Propagation* as it does not enforce equal-size clusters, and it can choose automatically the number of clusters from the data.

Note that this gives us a different indication than the graph, as the graph reflects conditional relations between variables, while the clustering reflects marginal properties: variables clustered together can be considered as having a similar impact at the level of the full stock market.

Embedding in 2D space

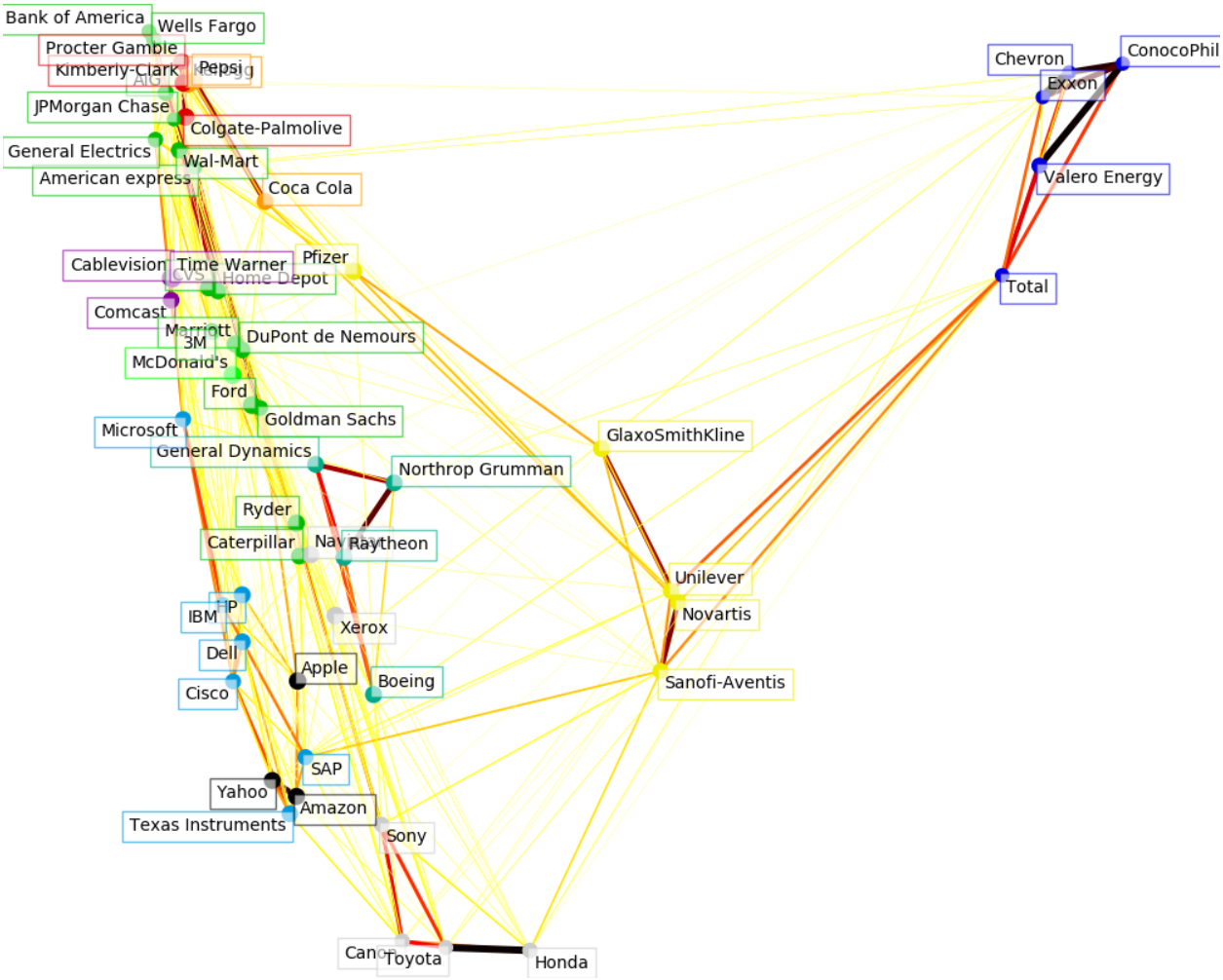
For visualization purposes, we need to lay out the different symbols on a 2D canvas. For this we use *Manifold learning* techniques to retrieve 2D embedding.

Visualization

The output of the 3 models are combined in a 2D graph where nodes represents the stocks and edges the:

- cluster labels are used to define the color of the nodes
- the sparse covariance model is used to display the strength of the edges
- the 2D embedding is used to position the nodes in the plan

This example has a fair amount of visualization-related code, as visualization is crucial here to display the graph. One of the challenge is to position the labels minimizing overlap. For this we use an heuristic based on the direction of the nearest neighbor along each axis.



Out:

```
Fetching quote history for 'AAPL'
Fetching quote history for 'AIG'
Fetching quote history for 'AMZN'
Fetching quote history for 'AXP'
Fetching quote history for 'BA'
Fetching quote history for 'BAC'
Fetching quote history for 'CAJ'
Fetching quote history for 'CAT'
Fetching quote history for 'CL'
Fetching quote history for 'CMCSA'
Fetching quote history for 'COP'
Fetching quote history for 'CSCO'
Fetching quote history for 'CVC'
Fetching quote history for 'CVS'
Fetching quote history for 'CVX'
Fetching quote history for 'DD'
Fetching quote history for 'DELL'
Fetching quote history for 'F'
Fetching quote history for 'GD'
Fetching quote history for 'GE'
Fetching quote history for 'GS'
```

(continues on next page)

(continued from previous page)

```
Fetching quote history for 'GSK'
Fetching quote history for 'HD'
Fetching quote history for 'HMC'
Fetching quote history for 'HPQ'
Fetching quote history for 'IBM'
Fetching quote history for 'JPM'
Fetching quote history for 'K'
Fetching quote history for 'KMB'
Fetching quote history for 'KO'
Fetching quote history for 'MAR'
Fetching quote history for 'MCD'
Fetching quote history for 'MMM'
Fetching quote history for 'MSFT'
Fetching quote history for 'NAV'
Fetching quote history for 'NOC'
Fetching quote history for 'NVS'
Fetching quote history for 'PEP'
Fetching quote history for 'PFE'
Fetching quote history for 'PG'
Fetching quote history for 'R'
Fetching quote history for 'RTN'
Fetching quote history for 'SAP'
Fetching quote history for 'SNE'
Fetching quote history for 'SNY'
Fetching quote history for 'TM'
Fetching quote history for 'TOT'
Fetching quote history for 'TWX'
Fetching quote history for 'TXN'
Fetching quote history for 'UN'
Fetching quote history for 'VLO'
Fetching quote history for 'WFC'
Fetching quote history for 'WMT'
Fetching quote history for 'XOM'
Fetching quote history for 'XRX'
Fetching quote history for 'YHOO'
Cluster 1: Apple, Amazon, Yahoo
Cluster 2: Comcast, Cablevision, Time Warner
Cluster 3: ConocoPhillips, Chevron, Total, Valero Energy, Exxon
Cluster 4: Cisco, Dell, HP, IBM, Microsoft, SAP, Texas Instruments
Cluster 5: Boeing, General Dynamics, Northrop Grumman, Raytheon
Cluster 6: AIG, American express, Bank of America, Caterpillar, CVS, DuPont de␣
↪Nemours, Ford, General Electrics, Goldman Sachs, Home Depot, JPMorgan Chase,␣
↪Marriott, 3M, Ryder, Wells Fargo, Wal-Mart
Cluster 7: McDonald's
Cluster 8: GlaxoSmithKline, Novartis, Pfizer, Sanofi-Aventis, Unilever
Cluster 9: Kellogg, Coca Cola, Pepsi
Cluster 10: Colgate-Palmolive, Kimberly-Clark, Procter Gamble
Cluster 11: Canon, Honda, Navistar, Sony, Toyota, Xerox
```

```
# Author: Gael Varoquaux gael.varoquaux@normalesup.org
# License: BSD 3 clause
```

(continues on next page)

(continued from previous page)

```
import sys

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection

import pandas as pd

from sklearn import cluster, covariance, manifold

print(__doc__)


# #############################################################################
# Retrieve the data from Internet

# The data is from 2003 - 2008. This is reasonably calm: (not too long ago so
# that we get high-tech firms, and before the 2008 crash). This kind of
# historical data can be obtained for from APIs like the quandl.com and
# alphavantage.co ones.

symbol_dict = {
    'TOT': 'Total',
    'XOM': 'Exxon',
    'CVX': 'Chevron',
    'COP': 'ConocoPhillips',
    'VLO': 'Valero Energy',
    'MSFT': 'Microsoft',
    'IBM': 'IBM',
    'TWX': 'Time Warner',
    'CMCSA': 'Comcast',
    'CVC': 'Cablevision',
    'YHOO': 'Yahoo',
    'DELL': 'Dell',
    'HPQ': 'HP',
    'AMZN': 'Amazon',
    'TM': 'Toyota',
    'CAJ': 'Canon',
    'SNE': 'Sony',
    'F': 'Ford',
    'HMC': 'Honda',
    'NAV': 'Navistar',
    'NOC': 'Northrop Grumman',
    'BA': 'Boeing',
    'KO': 'Coca Cola',
    'MMM': '3M',
    'MCD': 'McDonald\'s',
    'PEP': 'Pepsi',
    'K': 'Kellogg',
    'UN': 'Unilever',
    'MAR': 'Marriott',
    'PG': 'Procter Gamble',
    'CL': 'Colgate-Palmolive',
    'GE': 'General Electrics',
    'WFC': 'Wells Fargo',
    'JPM': 'JPMorgan Chase',
```

(continues on next page)

(continued from previous page)

```
    'AIG': 'AIG',
    'AXP': 'American express',
    'BAC': 'Bank of America',
    'GS': 'Goldman Sachs',
    'AAPL': 'Apple',
    'SAP': 'SAP',
    'CSCO': 'Cisco',
    'TXN': 'Texas Instruments',
    'XRX': 'Xerox',
    'WMT': 'Wal-Mart',
    'HD': 'Home Depot',
    'GSK': 'GlaxoSmithKline',
    'PFE': 'Pfizer',
    'SNY': 'Sanofi-Aventis',
    'NVS': 'Novartis',
    'KMB': 'Kimberly-Clark',
    'R': 'Ryder',
    'GD': 'General Dynamics',
    'RTN': 'Raytheon',
    'CVS': 'CVS',
    'CAT': 'Caterpillar',
    'DD': 'DuPont de Nemours'}


symbols, names = np.array(sorted(symbol_dict.items())).T

quotes = []

for symbol in symbols:
    print('Fetching quote history for %r' % symbol, file=sys.stderr)
    url = ('https://raw.githubusercontent.com/scikit-learn/examples-data/'
           'master/financial-data/{}.csv')
    quotes.append(pd.read_csv(url.format(symbol)))

close_prices = np.vstack([q['close'] for q in quotes])
open_prices = np.vstack([q['open'] for q in quotes])

# The daily variations of the quotes are what carry most information
variation = close_prices - open_prices


# #############################################################################
# Learn a graphical structure from the correlations
edge_model = covariance.GraphicalLassoCV()

# standardize the time series: using correlations rather than covariance
# is more efficient for structure recovery
X = variation.copy().T
X /= X.std(axis=0)
edge_model.fit(X)

# #############################################################################
# Cluster using affinity propagation

_, labels = cluster.affinity_propagation(edge_model.covariance_)
n_labels = labels.max()
```

(continues on next page)

(continued from previous page)

```
for i in range(n_labels + 1):
    print('Cluster %i: %s' % ((i + 1), ', '.join(names[labels == i])))

# #############################################################################
# Find a low-dimension embedding for visualization: find the best position of
# the nodes (the stocks) on a 2D plane

# We use a dense eigen_solver to achieve reproducibility (arpack is
# initiated with random vectors that we don't control). In addition, we
# use a large number of neighbors to capture the large-scale structure.
node_position_model = manifold.LocallyLinearEmbedding(
    n_components=2, eigen_solver='dense', n_neighbors=6)

embedding = node_position_model.fit_transform(X.T).T

# #############################################################################
# Visualization
plt.figure(1, facecolor='w', figsize=(10, 8))
plt.clf()
ax = plt.axes([0., 0., 1., 1.])
plt.axis('off')

# Display a graph of the partial correlations
partial_correlations = edge_model.precision_.copy()
d = 1 / np.sqrt(np.diag(partial_correlations))
partial_correlations *= d
partial_correlations *= d[:, np.newaxis]
non_zero = (np.abs(np.triu(partial_correlations, k=1)) > 0.02)

# Plot the nodes using the coordinates of our embedding
plt.scatter(embedding[0], embedding[1], s=100 * d ** 2, c=labels,
            cmap=plt.cm.nipy_spectral)

# Plot the edges
start_idx, end_idx = np.where(non_zero)
# a sequence of (*line0*, *line1*, *line2*), where::
#            linen = (x0, y0), (x1, y1), ... (xm, ym)
segments = [[embedding[:, start], embedding[:, stop]]
            for start, stop in zip(start_idx, end_idx)]
values = np.abs(partial_correlations[non_zero])
lc = LineCollection(segments,
                    zorder=0, cmap=plt.cm.hot_r,
                    norm=plt.Normalize(0, .7 * values.max()))
lc.set_array(values)
lc.set_linewidths(15 * values)
ax.add_collection(lc)

# Add a label to each node. The challenge here is that we want to
# position the labels to avoid overlap with other labels
for index, (name, label, (x, y)) in enumerate(
        zip(names, labels, embedding.T)):

    dx = x - embedding[0]
    dx[index] = 1
    dy = y - embedding[1]
    dy[index] = 1
    this_dx = dx[np.argmin(np.abs(dy))]
```

(continues on next page)

(continued from previous page)

```
    this_dy = dy[np.argmin(np.abs(dx))]
    if this_dx > 0:
        horizontalalignment = 'left'
        x = x + .002
    else:
        horizontalalignment = 'right'
        x = x - .002
    if this_dy > 0:
        verticalalignment = 'bottom'
        y = y + .002
    else:
        verticalalignment = 'top'
        y = y - .002
    plt.text(x, y, name, size=10,
             horizontalalignment=horizontalalignment,
             verticalalignment=verticalalignment,
             bbox=dict(facecolor='w',
                       edgecolor=plt.cm.nipy_spectral(label / float(n_labels)),
                       alpha=.6))

plt.xlim(embedding[0].min() - .15 * embedding[0].ptp(),
         embedding[0].max() + .10 * embedding[0].ptp(),)
plt.ylim(embedding[1].min() - .03 * embedding[1].ptp(),
         embedding[1].max() + .03 * embedding[1].ptp())

plt.show()
```

Total running time of the script: (0 minutes 16.487 seconds)

Estimated memory usage: 9 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.12.7 Wikipedia principal eigenvector

A classical way to assert the relative importance of vertices in a graph is to compute the principal eigenvector of the adjacency matrix so as to assign to each vertex the values of the components of the first eigenvector as a centrality score:

https://en.wikipedia.org/wiki/Eigenvector_centrality

On the graph of webpages and links those values are called the PageRank scores by Google.

The goal of this example is to analyze the graph of links inside wikipedia articles to rank articles by relative importance according to this eigenvector centrality.

The traditional way to compute the principal eigenvector is to use the power iteration method:

https://en.wikipedia.org/wiki/Power_iteration

Here the computation is achieved thanks to Martinsson's Randomized SVD algorithm implemented in scikit-learn.

The graph data is fetched from the DBpedia dumps. DBpedia is an extraction of the latent structured data of the Wikipedia content.

```
# Author: Olivier Grisel <olivier.grisel@ensta.org>
# License: BSD 3 clause

from bz2 import BZ2File
import os
from datetime import datetime
from pprint import pprint
from time import time

import numpy as np

from scipy import sparse

from joblib import Memory

from sklearn.decomposition import randomized_svd
from urllib.request import urlopen


print(__doc__)

# #############################################################################
# Where to download the data, if not already on disk
redirects_url = "http://downloads.dbpedia.org/3.5.1/en/redirects_en.nt.bz2"
redirects_filename = redirects_url.rsplit("/", 1)[1]

page_links_url = "http://downloads.dbpedia.org/3.5.1/en/page_links_en.nt.bz2"
page_links_filename = page_links_url.rsplit("/", 1)[1]

resources = [
    (redirects_url, redirects_filename),
    (page_links_url, page_links_filename),
]

for url, filename in resources:
    if not os.path.exists(filename):
        print("Downloading data from '%s', please wait..." % url)
        opener = urlopen(url)
        open(filename, 'wb').write(opener.read())
        print()


# #############################################################################
# Loading the redirect files

memory = Memory(cachedir=".")


def index(redirects, index_map, k):
    """Find the index of an article name after redirect resolution"""
    k = redirects.get(k, k)
    return index_map.setdefault(k, len(index_map))


DBPEDIA_RESOURCE_PREFIX_LEN = len("http://dbpedia.org/resource/")
SHORTNAME_SLICE = slice(DBPEDIA_RESOURCE_PREFIX_LEN + 1, -1)
```

(continues on next page)

(continued from previous page)

```
def short_name(nt_uri):
    """Remove the < and > URI markers and the common URI prefix"""
    return nt_uri[SHORTNAME_SLICE]


def get_redirects(redirects_filename):
    """Parse the redirections and build a transitively closed map out of it"""
    redirects = {}
    print("Parsing the NT redirect file")
    for l, line in enumerate(BZ2File(redirects_filename)):
        split = line.split()
        if len(split) != 4:
            print("ignoring malformed line: " + line)
            continue
        redirects[short_name(split[0])] = short_name(split[2])
        if l % 1000000 == 0:
            print("[%s] line: %08d" % (datetime.now().isoformat(), l))

    # compute the transitive closure
    print("Computing the transitive closure of the redirect relation")
    for l, source in enumerate(redirects.keys()):
        transitive_target = None
        target = redirects[source]
        seen = {source}
        while True:
            transitive_target = target
            target = redirects.get(target)
            if target is None or target in seen:
                break
            seen.add(target)
        redirects[source] = transitive_target
        if l % 1000000 == 0:
            print("[%s] line: %08d" % (datetime.now().isoformat(), l))

    return redirects


# disabling joblib as the pickling of large dicts seems much too slow
#@memory.cache
def get_adjacency_matrix(redirects_filename, page_links_filename, limit=None):
    """Extract the adjacency graph as a scipy sparse matrix

    Redirects are resolved first.

    Returns X, the scipy sparse adjacency matrix, redirects as python
    dict from article names to article names and index_map a python dict
    from article names to python int (article indexes).
    """

    print("Computing the redirect map")
    redirects = get_redirects(redirects_filename)

    print("Computing the integer index map")
    index_map = dict()
    links = list()
    for l, line in enumerate(BZ2File(page_links_filename)):
```

(continues on next page)

(continued from previous page)

```
        split = line.split()
        if len(split) != 4:
            print("ignoring malformed line: " + line)
            continue
        i = index(redirects, index_map, short_name(split[0]))
        j = index(redirects, index_map, short_name(split[2]))
        links.append((i, j))
        if l % 1000000 == 0:
            print("[%s] line: %08d" % (datetime.now().isoformat(), l))

        if limit is not None and l >= limit - 1:
            break

    print("Computing the adjacency matrix")
    X = sparse.lil_matrix((len(index_map), len(index_map)), dtype=np.float32)
    for i, j in links:
        X[i, j] = 1.0
    del links
    print("Converting to CSR representation")
    X = X.tocsr()
    print("CSR conversion done")
    return X, redirects, index_map


# stop after 5M links to make it possible to work in RAM
X, redirects, index_map = get_adjacency_matrix(
    redirects_filename, page_links_filename, limit=5000000)
names = {i: name for name, i in index_map.items()}

print("Computing the principal singular vectors using randomized_svd")
t0 = time()
U, s, V = randomized_svd(X, 5, n_iter=3)
print("done in %0.3fs" % (time() - t0))

# print the names of the wikipedia related strongest components of the
# principal singular vector which should be similar to the highest eigenvector
print("Top wikipedia pages according to principal singular vectors")
pprint([names[i] for i in np.abs(U.T[0]).argsort()[-10:]])
pprint([names[i] for i in np.abs(V[0]).argsort()[-10:]])


def centrality_scores(X, alpha=0.85, max_iter=100, tol=1e-10):
    """Power iteration computation of the principal eigenvector

    This method is also known as Google PageRank and the implementation
    is based on the one from the NetworkX project (BSD licensed too)
    with copyrights by:

      Aric Hagberg <hagberg@lanl.gov>
      Dan Schult <dschult@colgate.edu>
      Pieter Swart <swart@lanl.gov>
    """
    n = X.shape[0]
    X = X.copy()
    incoming_counts = np.asarray(X.sum(axis=1)).ravel()

    print("Normalizing the graph")
```

(continues on next page)

(continued from previous page)

```
    for i in incoming_counts.nonzero()[0]:
        X.data[X.indptr[i]:X.indptr[i + 1]] *= 1.0 / incoming_counts[i]
    dangle = np.asarray(np.where(np.isclose(X.sum(axis=1), 0),
                                 1.0 / n, 0)).ravel()

    scores = np.full(n, 1. / n, dtype=np.float32)  # initial guess
    for i in range(max_iter):
        print("power iteration #%d" % i)
        prev_scores = scores
        scores = (alpha * (scores * X + np.dot(dangle, prev_scores))
                  + (1 - alpha) * prev_scores.sum() / n)
        # check convergence: normalized l_inf norm
        scores_max = np.abs(scores).max()
        if scores_max == 0.0:
            scores_max = 1.0
        err = np.abs(scores - prev_scores).max() / scores_max
        print("error: %0.6f" % err)
        if err < n * tol:
            return scores

    return scores

print("Computing principal eigenvector score using a power iteration method")
t0 = time()
scores = centrality_scores(X, max_iter=100)
print("done in %0.3fs" % (time() - t0))
pprint([names[i] for i in np.abs(scores).argsort()[-10:]])
```

Total running time of the script: (0 minutes 0.000 seconds)

Estimated memory usage: 0 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.12.8 Species distribution modeling

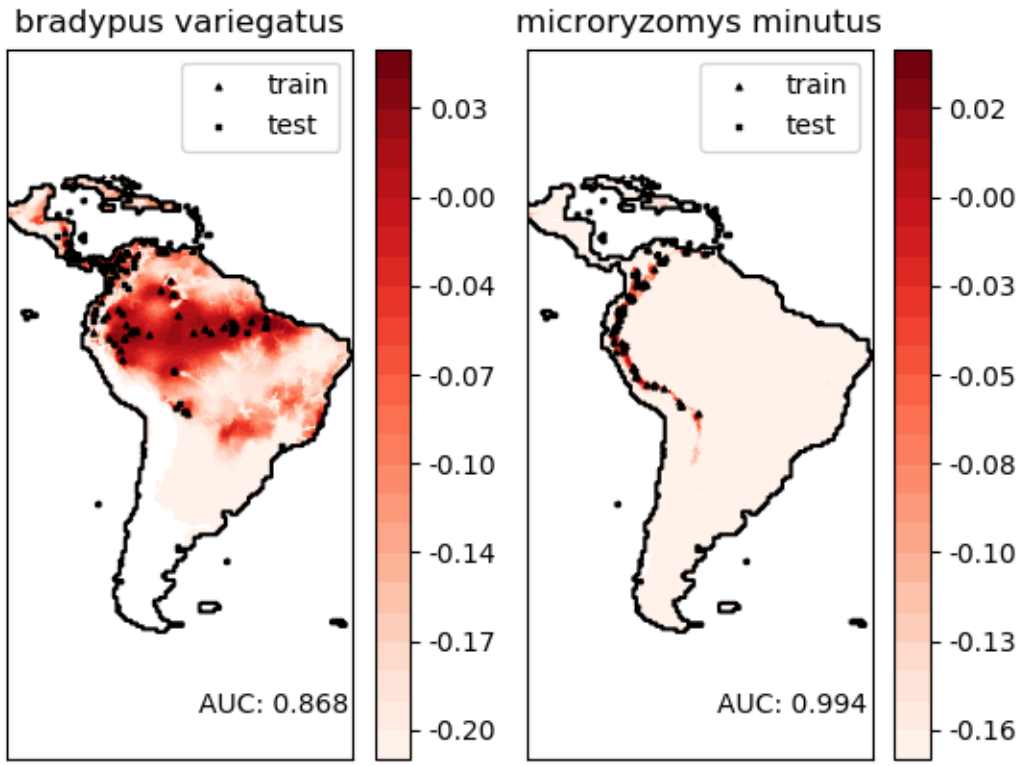
Modeling species' geographic distributions is an important problem in conservation biology. In this example we model the geographic distribution of two south american mammals given past observations and 14 environmental variables. Since we have only positive examples (there are no unsuccessful observations), we cast this problem as a density estimation problem and use the *sklearn.svm.OneClassSVM* as our modeling tool. The dataset is provided by Phillips et. al. (2006). If available, the example uses basemap to plot the coast lines and national boundaries of South America.

The two species are:

- "Bradypus variegatus" , the Brown-throated Sloth.
- "Microryzomys minutus" , also known as the Forest Small Rice Rat, a rodent that lives in Peru, Colombia, Ecuador, Peru, and Venezuela.

References

- "Maximum entropy modeling of species geographic distributions" S. J. Phillips, R. P. Anderson, R. E. Schapire - Ecological Modelling, 190:231-259, 2006.



Out:

```
________________________________________________________________________________
Modeling distribution of species 'bradypus variegatus'
 - fit OneClassSVM ... done.
 - plot coastlines from coverage
 - predict species distribution

 Area under the ROC curve : 0.868443
________________________________________________________________________________
Modeling distribution of species 'microryzomys minutus'
 - fit OneClassSVM ... done.
 - plot coastlines from coverage
 - predict species distribution

 Area under the ROC curve : 0.993919

time elapsed: 5.01s
```

```
# Authors: Peter Prettenhofer <peter.prettenhofer@gmail.com>
```

(continues on next page)

(continued from previous page)

```
#          Jake Vanderplas <vanderplas@astro.washington.edu>
#
# License: BSD 3 clause

from time import time

import numpy as np
import matplotlib.pyplot as plt

from sklearn.utils import Bunch
from sklearn.datasets import fetch_species_distributions
from sklearn import svm, metrics

# if basemap is available, we'll use it.
# otherwise, we'll improvise later...
try:
    from mpl_toolkits.basemap import Basemap
    basemap = True
except ImportError:
    basemap = False

print(__doc__)


def construct_grids(batch):
    """Construct the map grid from the batch object

    Parameters
    ----------
    batch : Batch object
        The object returned by :func:`fetch_species_distributions`

    Returns
    -------
    (xgrid, ygrid) : 1-D arrays
        The grid corresponding to the values in batch.coverages
    """
    # x,y coordinates for corner cells
    xmin = batch.x_left_lower_corner + batch.grid_size
    xmax = xmin + (batch.Nx * batch.grid_size)
    ymin = batch.y_left_lower_corner + batch.grid_size
    ymax = ymin + (batch.Ny * batch.grid_size)

    # x coordinates of the grid cells
    xgrid = np.arange(xmin, xmax, batch.grid_size)
    # y coordinates of the grid cells
    ygrid = np.arange(ymin, ymax, batch.grid_size)

    return (xgrid, ygrid)


def create_species_bunch(species_name, train, test, coverages, xgrid, ygrid):
    """Create a bunch with information about a particular organism

    This will use the test/train record arrays to extract the
    data specific to the given species name.
    """
```

(continues on next page)

(continued from previous page)

```
    bunch = Bunch(name=' '.join(species_name.split("_")[:2]))
    species_name = species_name.encode('ascii')
    points = dict(test=test, train=train)

    for label, pts in points.items():
        # choose points associated with the desired species
        pts = pts[pts['species'] == species_name]
        bunch['pts_%s' % label] = pts

        # determine coverage values for each of the training & testing points
        ix = np.searchsorted(xgrid, pts['dd long'])
        iy = np.searchsorted(ygrid, pts['dd lat'])
        bunch['cov_%s' % label] = coverages[:, -iy, ix].T

    return bunch


def plot_species_distribution(species=("bradypus_variegatus_0",
                                       "microryzomys_minutus_0")):
    """
    Plot the species distribution.
    """
    if len(species) > 2:
        print("Note: when more than two species are provided,"
              " only the first two will be used")

    t0 = time()

    # Load the compressed data
    data = fetch_species_distributions()

    # Set up the data grid
    xgrid, ygrid = construct_grids(data)

    # The grid in x,y coordinates
    X, Y = np.meshgrid(xgrid, ygrid[::-1])

    # create a bunch for each species
    BV_bunch = create_species_bunch(species[0],
                                    data.train, data.test,
                                    data.coverages, xgrid, ygrid)
    MM_bunch = create_species_bunch(species[1],
                                    data.train, data.test,
                                    data.coverages, xgrid, ygrid)

    # background points (grid coordinates) for evaluation
    np.random.seed(13)
    background_points = np.c_[np.random.randint(low=0, high=data.Ny,
                                                size=10000),
                              np.random.randint(low=0, high=data.Nx,
                                                size=10000)].T

    # We'll make use of the fact that coverages[6] has measurements at all
    # land points.  This will help us decide between land and water.
    land_reference = data.coverages[6]

    # Fit, predict, and plot for each species.
```

(continues on next page)

(continued from previous page)

```
    for i, species in enumerate([BV_bunch, MM_bunch]):
        print("_" * 80)
        print("Modeling distribution of species '%s'" % species.name)

        # Standardize features
        mean = species.cov_train.mean(axis=0)
        std = species.cov_train.std(axis=0)
        train_cover_std = (species.cov_train - mean) / std

        # Fit OneClassSVM
        print(" - fit OneClassSVM ... ", end='')
        clf = svm.OneClassSVM(nu=0.1, kernel="rbf", gamma=0.5)
        clf.fit(train_cover_std)
        print("done.")

        # Plot map of South America
        plt.subplot(1, 2, i + 1)
        if basemap:
            print(" - plot coastlines using basemap")
            m = Basemap(projection='cyl', llcrnrlat=Y.min(),
                        urcrnrlat=Y.max(), llcrnrlon=X.min(),
                        urcrnrlon=X.max(), resolution='c')
            m.drawcoastlines()
            m.drawcountries()
        else:
            print(" - plot coastlines from coverage")
            plt.contour(X, Y, land_reference,
                        levels=[-9998], colors="k",
                        linestyles="solid")
            plt.xticks([])
            plt.yticks([])

        print(" - predict species distribution")

        # Predict species distribution using the training data
        Z = np.ones((data.Ny, data.Nx), dtype=np.float64)

        # We'll predict only for the land points.
        idx = np.where(land_reference > -9999)
        coverages_land = data.coverages[:, idx[0], idx[1]].T

        pred = clf.decision_function((coverages_land - mean) / std)
        Z *= pred.min()
        Z[idx[0], idx[1]] = pred

        levels = np.linspace(Z.min(), Z.max(), 25)
        Z[land_reference == -9999] = -9999

        # plot contours of the prediction
        plt.contourf(X, Y, Z, levels=levels, cmap=plt.cm.Reds)
        plt.colorbar(format='%.2f')

        # scatter training/testing points
        plt.scatter(species.pts_train['dd long'], species.pts_train['dd lat'],
                    s=2 ** 2, c='black',
                    marker='^', label='train')
        plt.scatter(species.pts_test['dd long'], species.pts_test['dd lat'],
```

(continues on next page)

(continued from previous page)

```
                    s=2 ** 2, c='black',
                    marker='x', label='test')
        plt.legend()
        plt.title(species.name)
        plt.axis('equal')

        # Compute AUC with regards to background points
        pred_background = Z[background_points[0], background_points[1]]
        pred_test = clf.decision_function((species.cov_test - mean) / std)
        scores = np.r_[pred_test, pred_background]
        y = np.r_[np.ones(pred_test.shape), np.zeros(pred_background.shape)]
        fpr, tpr, thresholds = metrics.roc_curve(y, scores)
        roc_auc = metrics.auc(fpr, tpr)
        plt.text(-35, -70, "AUC: %.3f" % roc_auc, ha="right")
        print("\n Area under the ROC curve : %f" % roc_auc)

    print("\ntime elapsed: %.2fs" % (time() - t0))


plot_species_distribution()
plt.show()
```

Total running time of the script: (0 minutes 5.365 seconds)

Estimated memory usage: 287 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.12.9 Libsvm GUI

A simple graphical frontend for Libsvm mainly intended for didactic purposes. You can create data points by point and click and visualize the decision region induced by different kernels and parameter settings.

To create positive examples click the left mouse button; to create negative examples click the right button.

If all examples are from the same class, it uses a one-class SVM.

```
print(__doc__)

# Author: Peter Prettenhoer <peter.prettenhofer@gmail.com>
#
# License: BSD 3 clause

import matplotlib
matplotlib.use('TkAgg')

from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.backends.backend_tkagg import NavigationToolbar2TkAgg
from matplotlib.figure import Figure
from matplotlib.contour import ContourSet

import sys
import numpy as np
import tkinter as Tk
```

(continues on next page)

(continued from previous page)

```
from sklearn import svm
from sklearn.datasets import dump_svmlight_file

y_min, y_max = -50, 50
x_min, x_max = -50, 50


class Model:
    """The Model which hold the data. It implements the
    observable in the observer pattern and notifies the
    registered observers on change event.
    """

    def __init__(self):
        self.observers = []
        self.surface = None
        self.data = []
        self.cls = None
        self.surface_type = 0

    def changed(self, event):
        """Notify the observers. """
        for observer in self.observers:
            observer.update(event, self)

    def add_observer(self, observer):
        """Register an observer. """
        self.observers.append(observer)

    def set_surface(self, surface):
        self.surface = surface

    def dump_svmlight_file(self, file):
        data = np.array(self.data)
        X = data[:, 0:2]
        y = data[:, 2]
        dump_svmlight_file(X, y, file)


class Controller:
    def __init__(self, model):
        self.model = model
        self.kernel = Tk.IntVar()
        self.surface_type = Tk.IntVar()
        # Whether or not a model has been fitted
        self.fitted = False

    def fit(self):
        print("fit the model")
        train = np.array(self.model.data)
        X = train[:, 0:2]
        y = train[:, 2]

        C = float(self.complexity.get())
        gamma = float(self.gamma.get())
        coef0 = float(self.coef0.get())
```

(continues on next page)

(continued from previous page)

```
        degree = int(self.degree.get())
        kernel_map = {0: "linear", 1: "rbf", 2: "poly"}
        if len(np.unique(y)) == 1:
            clf = svm.OneClassSVM(kernel=kernel_map[self.kernel.get()],
                                  gamma=gamma, coef0=coef0, degree=degree)
            clf.fit(X)
        else:
            clf = svm.SVC(kernel=kernel_map[self.kernel.get()], C=C,
                          gamma=gamma, coef0=coef0, degree=degree)
            clf.fit(X, y)
        if hasattr(clf, 'score'):
            print("Accuracy:", clf.score(X, y) * 100)
        X1, X2, Z = self.decision_surface(clf)
        self.model.clf = clf
        self.model.set_surface((X1, X2, Z))
        self.model.surface_type = self.surface_type.get()
        self.fitted = True
        self.model.changed("surface")

    def decision_surface(self, cls):
        delta = 1
        x = np.arange(x_min, x_max + delta, delta)
        y = np.arange(y_min, y_max + delta, delta)
        X1, X2 = np.meshgrid(x, y)
        Z = cls.decision_function(np.c_[X1.ravel(), X2.ravel()])
        Z = Z.reshape(X1.shape)
        return X1, X2, Z

    def clear_data(self):
        self.model.data = []
        self.fitted = False
        self.model.changed("clear")

    def add_example(self, x, y, label):
        self.model.data.append((x, y, label))
        self.model.changed("example_added")

        # update decision surface if already fitted.
        self.refit()

    def refit(self):
        """Refit the model if already fitted. """
        if self.fitted:
            self.fit()


class View:
    """Test docstring. """
    def __init__(self, root, controller):
        f = Figure()
        ax = f.add_subplot(111)
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_xlim((x_min, x_max))
        ax.set_ylim((y_min, y_max))
        canvas = FigureCanvasTkAgg(f, master=root)
        canvas.show()
```

(continues on next page)

(continued from previous page)

```
        canvas.get_tk_widget().pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)
        canvas._tkcanvas.pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)
        canvas.mpl_connect('button_press_event', self.onclick)
        toolbar = NavigationToolbar2TkAgg(canvas, root)
        toolbar.update()
        self.controllbar = ControllBar(root, controller)
        self.f = f
        self.ax = ax
        self.canvas = canvas
        self.controller = controller
        self.contours = []
        self.c_labels = None
        self.plot_kernels()

    def plot_kernels(self):
        self.ax.text(-50, -60, "Linear: $u^T v$")
        self.ax.text(-20, -60, r"RBF: $\exp (-\gamma \| u-v \|^2)$")
        self.ax.text(10, -60, r"Poly: $(\gamma \, u^T v + r)^d$")

    def onclick(self, event):
        if event.xdata and event.ydata:
            if event.button == 1:
                self.controller.add_example(event.xdata, event.ydata, 1)
            elif event.button == 3:
                self.controller.add_example(event.xdata, event.ydata, -1)

    def update_example(self, model, idx):
        x, y, l = model.data[idx]
        if l == 1:
            color = 'w'
        elif l == -1:
            color = 'k'
        self.ax.plot([x], [y], "%so" % color, scalex=0.0, scaley=0.0)

    def update(self, event, model):
        if event == "examples_loaded":
            for i in range(len(model.data)):
                self.update_example(model, i)

        if event == "example_added":
            self.update_example(model, -1)

        if event == "clear":
            self.ax.clear()
            self.ax.set_xticks([])
            self.ax.set_yticks([])
            self.contours = []
            self.c_labels = None
            self.plot_kernels()

        if event == "surface":
            self.remove_surface()
            self.plot_support_vectors(model.clf.support_vectors_)
            self.plot_decision_surface(model.surface, model.surface_type)

        self.canvas.draw()
```

(continues on next page)

(continued from previous page)

```
    def remove_surface(self):
        """Remove old decision surface."""
        if len(self.contours) > 0:
            for contour in self.contours:
                if isinstance(contour, ContourSet):
                    for lineset in contour.collections:
                        lineset.remove()
                else:
                    contour.remove()
            self.contours = []

    def plot_support_vectors(self, support_vectors):
        """Plot the support vectors by placing circles over the
        corresponding data points and adds the circle collection
        to the contours list."""
        cs = self.ax.scatter(support_vectors[:, 0], support_vectors[:, 1],
                             s=80, edgecolors="k", facecolors="none")
        self.contours.append(cs)

    def plot_decision_surface(self, surface, type):
        X1, X2, Z = surface
        if type == 0:
            levels = [-1.0, 0.0, 1.0]
            linestyles = ['dashed', 'solid', 'dashed']
            colors = 'k'
            self.contours.append(self.ax.contour(X1, X2, Z, levels,
                                                 colors=colors,
                                                 linestyles=linestyles))
        elif type == 1:
            self.contours.append(self.ax.contourf(X1, X2, Z, 10,
                                                  cmap=matplotlib.cm.bone,
                                                  origin='lower', alpha=0.85))
            self.contours.append(self.ax.contour(X1, X2, Z, [0.0], colors='k',
                                                 linestyles=['solid']))
        else:
            raise ValueError("surface type unknown")


class ControllBar:
    def __init__(self, root, controller):
        fm = Tk.Frame(root)
        kernel_group = Tk.Frame(fm)
        Tk.Radiobutton(kernel_group, text="Linear", variable=controller.kernel,
                       value=0, command=controller.refit).pack(anchor=Tk.W)
        Tk.Radiobutton(kernel_group, text="RBF", variable=controller.kernel,
                       value=1, command=controller.refit).pack(anchor=Tk.W)
        Tk.Radiobutton(kernel_group, text="Poly", variable=controller.kernel,
                       value=2, command=controller.refit).pack(anchor=Tk.W)
        kernel_group.pack(side=Tk.LEFT)

        valbox = Tk.Frame(fm)
        controller.complexity = Tk.StringVar()
        controller.complexity.set("1.0")
        c = Tk.Frame(valbox)
        Tk.Label(c, text="C:", anchor="e", width=7).pack(side=Tk.LEFT)
        Tk.Entry(c, width=6, textvariable=controller.complexity).pack(
            side=Tk.LEFT)
```

(continues on next page)

(continued from previous page)

```
        c.pack()

        controller.gamma = Tk.StringVar()
        controller.gamma.set("0.01")
        g = Tk.Frame(valbox)
        Tk.Label(g, text="gamma:", anchor="e", width=7).pack(side=Tk.LEFT)
        Tk.Entry(g, width=6, textvariable=controller.gamma).pack(side=Tk.LEFT)
        g.pack()

        controller.degree = Tk.StringVar()
        controller.degree.set("3")
        d = Tk.Frame(valbox)
        Tk.Label(d, text="degree:", anchor="e", width=7).pack(side=Tk.LEFT)
        Tk.Entry(d, width=6, textvariable=controller.degree).pack(side=Tk.LEFT)
        d.pack()

        controller.coef0 = Tk.StringVar()
        controller.coef0.set("0")
        r = Tk.Frame(valbox)
        Tk.Label(r, text="coef0:", anchor="e", width=7).pack(side=Tk.LEFT)
        Tk.Entry(r, width=6, textvariable=controller.coef0).pack(side=Tk.LEFT)
        r.pack()
        valbox.pack(side=Tk.LEFT)

        cmap_group = Tk.Frame(fm)
        Tk.Radiobutton(cmap_group, text="Hyperplanes",
                       variable=controller.surface_type, value=0,
                       command=controller.refit).pack(anchor=Tk.W)
        Tk.Radiobutton(cmap_group, text="Surface",
                       variable=controller.surface_type, value=1,
                       command=controller.refit).pack(anchor=Tk.W)

        cmap_group.pack(side=Tk.LEFT)

        train_button = Tk.Button(fm, text='Fit', width=5,
                                 command=controller.fit)
        train_button.pack()
        fm.pack(side=Tk.LEFT)
        Tk.Button(fm, text='Clear', width=5,
                  command=controller.clear_data).pack(side=Tk.LEFT)


def get_parser():
    from optparse import OptionParser
    op = OptionParser()
    op.add_option("--output",
                  action="store", type="str", dest="output",
                  help="Path where to dump data.")
    return op


def main(argv):
    op = get_parser()
    opts, args = op.parse_args(argv[1:])
    root = Tk.Tk()
    model = Model()
    controller = Controller(model)
```

(continues on next page)

(continued from previous page)

```
    root.wm_title("Scikit-learn Libsvm GUI")
    view = View(root, controller)
    model.add_observer(view)
    Tk.mainloop()

    if opts.output:
        model.dump_svmlight_file(opts.output)

if __name__ == "__main__":
    main(sys.argv)
```

Total running time of the script: (0 minutes 0.000 seconds)

Estimated memory usage: 0 MB

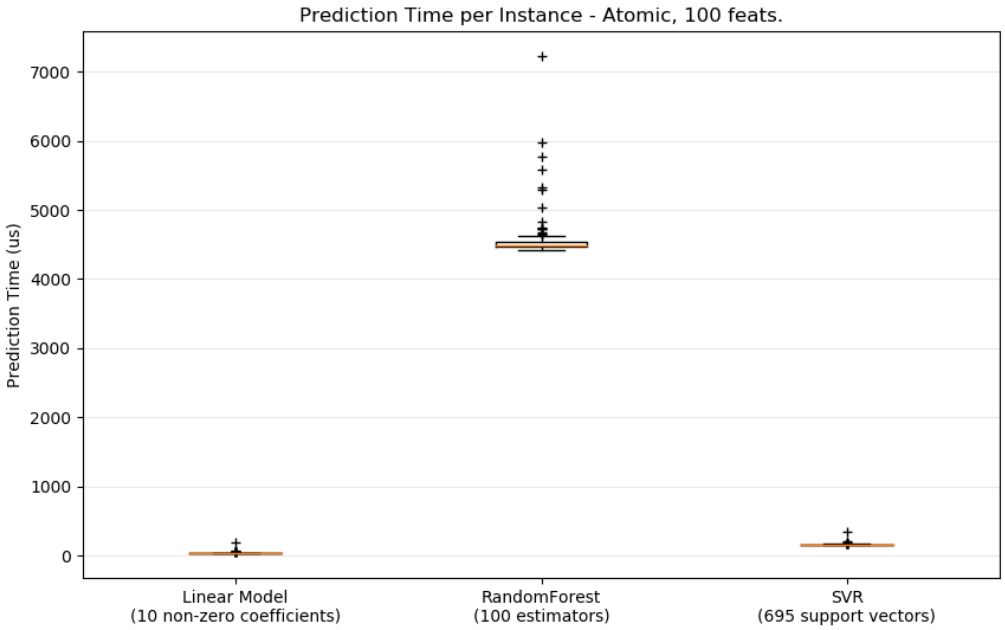
Note: Click *here* to download the full example code or to run this example in your browser via Binder

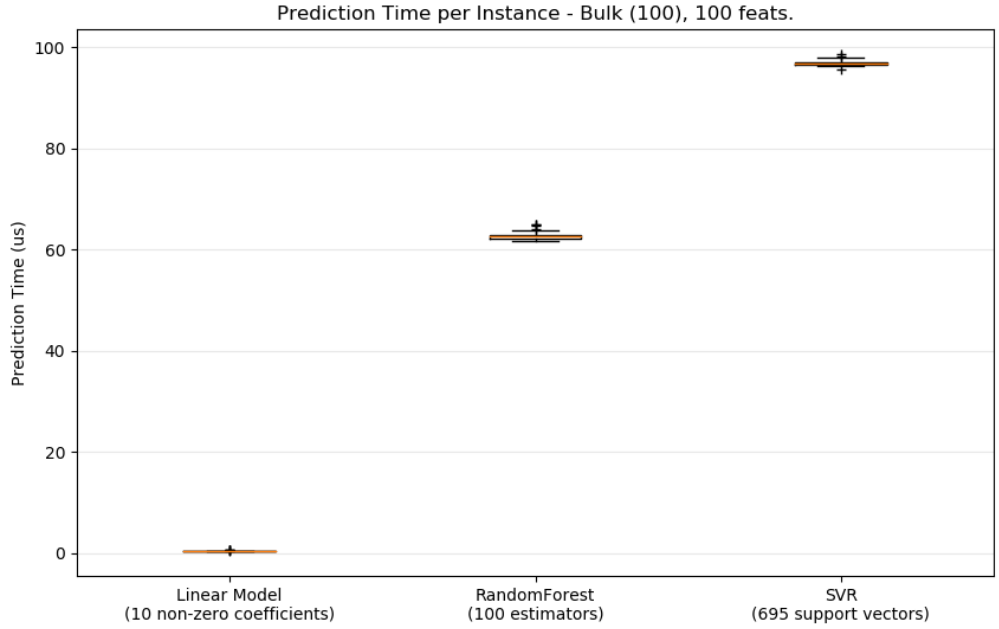
6.12.10 Prediction Latency

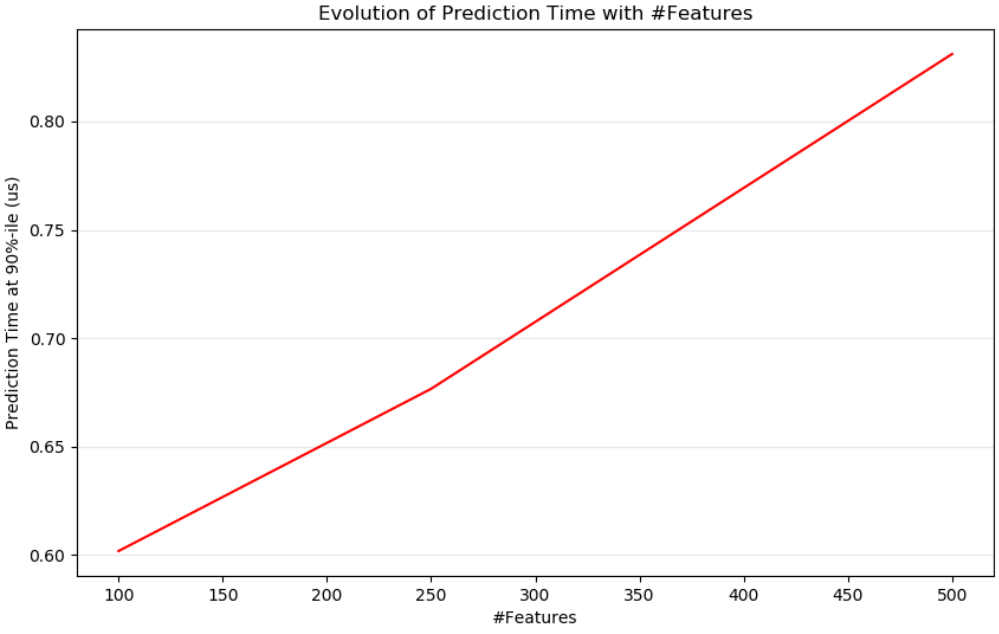
This is an example showing the prediction latency of various scikit-learn estimators.

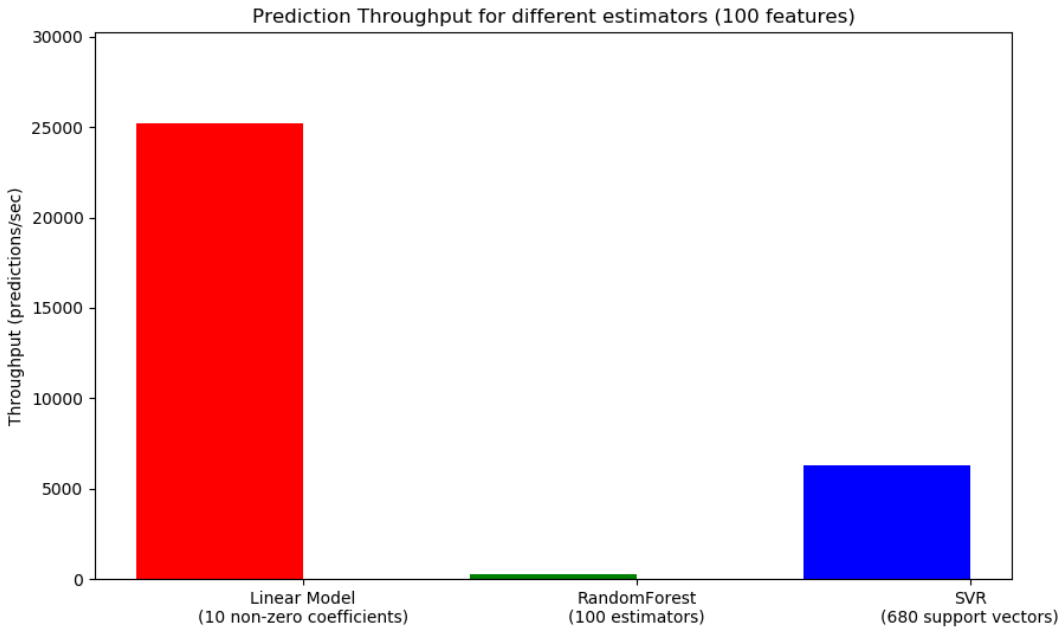
The goal is to measure the latency one can expect when doing predictions either in bulk or atomic (i.e. one by one) mode.

The plots represent the distribution of the prediction latency as a boxplot.



- 
Prediction Time per Instance - Bulk (100), 100 feats.
Prediction Time (us)
100
80
60
40
20
0
Linear Model
(10 non-zero coefficients)
RandomForest
(100 estimators)
SVR
(695 support vectors)

- 
Evolution of Prediction Time with #Features
Prediction Time at 90%-ile (us)
0.80
0.75
0.70
0.65
0.60
100
150
200
250
300
350
400
450
500
#Features



Out:

```
Benchmarking SGDRegressor(alpha=0.01, l1_ratio=0.25, penalty='elasticnet', tol=0.0001)
Benchmarking RandomForestRegressor()
Benchmarking SVR()
benchmarking with 100 features
benchmarking with 250 features
benchmarking with 500 features
example run in 10.47s
```

```
# Authors: Eustache Diemert <eustache@diemert.fr>
# License: BSD 3 clause

from collections import defaultdict

import time
import gc
import numpy as np
import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_regression
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import Ridge
from sklearn.linear_model import SGDRegressor
from sklearn.svm import SVR
from sklearn.utils import shuffle
```

(continues on next page)

(continued from previous page)

```
def _not_in_sphinx():
    # Hack to detect whether we are running by the sphinx builder
    return '__file__' in globals()


def atomic_benchmark_estimator(estimator, X_test, verbose=False):
    """Measure runtime prediction of each instance."""
    n_instances = X_test.shape[0]
    runtimes = np.zeros(n_instances, dtype=np.float)
    for i in range(n_instances):
        instance = X_test[[i], :]
        start = time.time()
        estimator.predict(instance)
        runtimes[i] = time.time() - start
    if verbose:
        print("atomic_benchmark runtimes:", min(runtimes), np.percentile(
            runtimes, 50), max(runtimes))
    return runtimes


def bulk_benchmark_estimator(estimator, X_test, n_bulk_repeats, verbose):
    """Measure runtime prediction of the whole input."""
    n_instances = X_test.shape[0]
    runtimes = np.zeros(n_bulk_repeats, dtype=np.float)
    for i in range(n_bulk_repeats):
        start = time.time()
        estimator.predict(X_test)
        runtimes[i] = time.time() - start
    runtimes = np.array(list(map(lambda x: x / float(n_instances), runtimes)))
    if verbose:
        print("bulk_benchmark runtimes:", min(runtimes), np.percentile(
            runtimes, 50), max(runtimes))
    return runtimes


def benchmark_estimator(estimator, X_test, n_bulk_repeats=30, verbose=False):
    """
    Measure runtimes of prediction in both atomic and bulk mode.

    Parameters
    ----------
    estimator : already trained estimator supporting `predict()`
    X_test : test input
    n_bulk_repeats : how many times to repeat when evaluating bulk mode

    Returns
    -------
    atomic_runtimes, bulk_runtimes : a pair of `np.array` which contain the
    runtimes in seconds.

    """
    atomic_runtimes = atomic_benchmark_estimator(estimator, X_test, verbose)
    bulk_runtimes = bulk_benchmark_estimator(estimator, X_test, n_bulk_repeats,
                                             verbose)
    return atomic_runtimes, bulk_runtimes
```

(continues on next page)

(continued from previous page)

```
def generate_dataset(n_train, n_test, n_features, noise=0.1, verbose=False):
    """Generate a regression dataset with the given parameters."""
    if verbose:
        print("generating dataset...")

    X, y, coef = make_regression(n_samples=n_train + n_test,
                                 n_features=n_features, noise=noise, coef=True)

    random_seed = 13
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, train_size=n_train, test_size=n_test, random_state=random_seed)
    X_train, y_train = shuffle(X_train, y_train, random_state=random_seed)

    X_scaler = StandardScaler()
    X_train = X_scaler.fit_transform(X_train)
    X_test = X_scaler.transform(X_test)

    y_scaler = StandardScaler()
    y_train = y_scaler.fit_transform(y_train[:, None])[:, 0]
    y_test = y_scaler.transform(y_test[:, None])[:, 0]

    gc.collect()
    if verbose:
        print("ok")
    return X_train, y_train, X_test, y_test


def boxplot_runtimes(runtimes, pred_type, configuration):
    """
    Plot a new `Figure` with boxplots of prediction runtimes.

    Parameters
    ----------
    runtimes : list of `np.array` of latencies in micro-seconds
    cls_names : list of estimator class names that generated the runtimes
    pred_type : 'bulk' or 'atomic'

    """

    fig, ax1 = plt.subplots(figsize=(10, 6))
    bp = plt.boxplot(runtimes, )

    cls_infos = ['%s\n(%d %s)' % (estimator_conf['name'],
                                  estimator_conf['complexity_computer'](
                                      estimator_conf['instance']),
                                  estimator_conf['complexity_label']) for
                 estimator_conf in configuration['estimators']]
    plt.setp(ax1, xticklabels=cls_infos)
    plt.setp(bp['boxes'], color='black')
    plt.setp(bp['whiskers'], color='black')
    plt.setp(bp['fliers'], color='red', marker='+')

    ax1.yaxis.grid(True, linestyle='-', which='major', color='lightgrey',
                   alpha=0.5)
```

(continues on next page)

(continued from previous page)

```
    ax1.set_axisbelow(True)
    ax1.set_title('Prediction Time per Instance - %s, %d feats.' % (
        pred_type.capitalize(),
        configuration['n_features']))
    ax1.set_ylabel('Prediction Time (us)')

    plt.show()


def benchmark(configuration):
    """Run the whole benchmark."""
    X_train, y_train, X_test, y_test = generate_dataset(
        configuration['n_train'], configuration['n_test'],
        configuration['n_features'])

    stats = {}
    for estimator_conf in configuration['estimators']:
        print("Benchmarking", estimator_conf['instance'])
        estimator_conf['instance'].fit(X_train, y_train)
        gc.collect()
        a, b = benchmark_estimator(estimator_conf['instance'], X_test)
        stats[estimator_conf['name']] = {'atomic': a, 'bulk': b}

    cls_names = [estimator_conf['name'] for estimator_conf in configuration[
        'estimators']]
    runtimes = [1e6 * stats[clf_name]['atomic'] for clf_name in cls_names]
    boxplot_runtimes(runtimes, 'atomic', configuration)
    runtimes = [1e6 * stats[clf_name]['bulk'] for clf_name in cls_names]
    boxplot_runtimes(runtimes, 'bulk (%d)' % configuration['n_test'],
                     configuration)


def n_feature_influence(estimators, n_train, n_test, n_features, percentile):
    """
    Estimate influence of the number of features on prediction time.

    Parameters
    ----------

    estimators : dict of (name (str), estimator) to benchmark
    n_train : nber of training instances (int)
    n_test : nber of testing instances (int)
    n_features : list of feature-space dimensionality to test (int)
    percentile : percentile at which to measure the speed (int [0-100])

    Returns:
    --------

    percentiles : dict(estimator_name,
                       dict(n_features, percentile_perf_in_us))

    """
    percentiles = defaultdict(defaultdict)
    for n in n_features:
        print("benchmarking with %d features" % n)
        X_train, y_train, X_test, y_test = generate_dataset(n_train, n_test, n)
        for cls_name, estimator in estimators.items():
```

(continues on next page)

(continued from previous page)

```
            estimator.fit(X_train, y_train)
            gc.collect()
            runtimes = bulk_benchmark_estimator(estimator, X_test, 30, False)
            percentiles[cls_name][n] = 1e6 * np.percentile(runtimes,
                                                           percentile)
    return percentiles


def plot_n_features_influence(percentiles, percentile):
    fig, ax1 = plt.subplots(figsize=(10, 6))
    colors = ['r', 'g', 'b']
    for i, cls_name in enumerate(percentiles.keys()):
        x = np.array(sorted([n for n in percentiles[cls_name].keys()]))
        y = np.array([percentiles[cls_name][n] for n in x])
        plt.plot(x, y, color=colors[i], )
    ax1.yaxis.grid(True, linestyle='-', which='major', color='lightgrey',
                   alpha=0.5)
    ax1.set_axisbelow(True)
    ax1.set_title('Evolution of Prediction Time with #Features')
    ax1.set_xlabel('#Features')
    ax1.set_ylabel('Prediction Time at %d%%-ile (us)' % percentile)
    plt.show()


def benchmark_throughputs(configuration, duration_secs=0.1):
    """benchmark throughput for different estimators."""
    X_train, y_train, X_test, y_test = generate_dataset(
        configuration['n_train'], configuration['n_test'],
        configuration['n_features'])
    throughputs = dict()
    for estimator_config in configuration['estimators']:
        estimator_config['instance'].fit(X_train, y_train)
        start_time = time.time()
        n_predictions = 0
        while (time.time() - start_time) < duration_secs:
            estimator_config['instance'].predict(X_test[[0]])
            n_predictions += 1
        throughputs[estimator_config['name']] = n_predictions / duration_secs
    return throughputs


def plot_benchmark_throughput(throughputs, configuration):
    fig, ax = plt.subplots(figsize=(10, 6))
    colors = ['r', 'g', 'b']
    cls_infos = ['%s\n(%d %s)' % (estimator_conf['name'],
                                  estimator_conf['complexity_computer'](
                                      estimator_conf['instance']),
                                  estimator_conf['complexity_label']) for
                 estimator_conf in configuration['estimators']]
    cls_values = [throughputs[estimator_conf['name']] for estimator_conf in
                  configuration['estimators']]
    plt.bar(range(len(throughputs)), cls_values, width=0.5, color=colors)
    ax.set_xticks(np.linspace(0.25, len(throughputs) - 0.75, len(throughputs)))
    ax.set_xticklabels(cls_infos, fontsize=10)
    ymax = max(cls_values) * 1.2
    ax.set_ylim((0, ymax))
    ax.set_ylabel('Throughput (predictions/sec)')
```

(continues on next page)

(continued from previous page)

```
    ax.set_title('Prediction Throughput for different estimators (%d '
                 'features)' % configuration['n_features'])
    plt.show()


# ############################################################################
# Main code

start_time = time.time()

# ############################################################################
# Benchmark bulk/atomic prediction speed for various regressors
configuration = {
    'n_train': int(1e3),
    'n_test': int(1e2),
    'n_features': int(1e2),
    'estimators': [
        {'name': 'Linear Model',
         'instance': SGDRegressor(penalty='elasticnet', alpha=0.01,
                                  l1_ratio=0.25, tol=1e-4),
         'complexity_label': 'non-zero coefficients',
         'complexity_computer': lambda clf: np.count_nonzero(clf.coef_)},
        {'name': 'RandomForest',
         'instance': RandomForestRegressor(),
         'complexity_label': 'estimators',
         'complexity_computer': lambda clf: clf.n_estimators},
        {'name': 'SVR',
         'instance': SVR(kernel='rbf'),
         'complexity_label': 'support vectors',
         'complexity_computer': lambda clf: len(clf.support_vectors_)},
    ]
}
benchmark(configuration)

# benchmark n_features influence on prediction speed
percentile = 90
percentiles = n_feature_influence({'ridge': Ridge()},
                                  configuration['n_train'],
                                  configuration['n_test'],
                                  [100, 250, 500], percentile)
plot_n_features_influence(percentiles, percentile)

# benchmark throughput
throughputs = benchmark_throughputs(configuration)
plot_benchmark_throughput(throughputs, configuration)

stop_time = time.time()
print("example run in %.2fs" % (stop_time - start_time))
```

Total running time of the script: (0 minutes 10.898 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.12.11 Out-of-core classification of text documents

This is an example showing how scikit-learn can be used for classification using an out-of-core approach: learning from data that doesn't fit into main memory. We make use of an online classifier, i.e., one that supports the partial_fit method, that will be fed with batches of examples. To guarantee that the features space remains the same over time we leverage a HashingVectorizer that will project each example into the same feature space. This is especially useful in the case of text classification where new features (words) may appear in each batch.

```
# Authors: Eustache Diemert <eustache@diemert.fr>
#          @FedericoV <https://github.com/FedericoV/>
# License: BSD 3 clause

from glob import glob
import itertools
import os.path
import re
import tarfile
import time
import sys

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams

from html.parser import HTMLParser
from urllib.request import urlretrieve
from sklearn.datasets import get_data_home
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.linear_model import SGDClassifier
from sklearn.linear_model import PassiveAggressiveClassifier
from sklearn.linear_model import Perceptron
from sklearn.naive_bayes import MultinomialNB


def _not_in_sphinx():
    # Hack to detect whether we are running by the sphinx builder
    return '__file__' in globals()
```

Reuters Dataset related routines

The dataset used in this example is Reuters-21578 as provided by the UCI ML repository. It will be automatically downloaded and uncompressed on first run.

```
class ReutersParser(HTMLParser):
    """Utility class to parse a SGML file and yield documents one at a time."""

    def __init__(self, encoding='latin-1'):
        HTMLParser.__init__(self)
        self._reset()
        self.encoding = encoding

    def handle_starttag(self, tag, attrs):
        method = 'start_' + tag
        getattr(self, method, lambda x: None)(attrs)

    def handle_endtag(self, tag):
```

(continues on next page)

(continued from previous page)

```
        method = 'end_' + tag
        getattr(self, method, lambda: None)()

    def _reset(self):
        self.in_title = 0
        self.in_body = 0
        self.in_topics = 0
        self.in_topic_d = 0
        self.title = ""
        self.body = ""
        self.topics = []
        self.topic_d = ""

    def parse(self, fd):
        self.docs = []
        for chunk in fd:
            self.feed(chunk.decode(self.encoding))
            for doc in self.docs:
                yield doc
            self.docs = []
        self.close()

    def handle_data(self, data):
        if self.in_body:
            self.body += data
        elif self.in_title:
            self.title += data
        elif self.in_topic_d:
            self.topic_d += data

    def start_reuters(self, attributes):
        pass

    def end_reuters(self):
        self.body = re.sub(r'\s+', r' ', self.body)
        self.docs.append({'title': self.title,
                          'body': self.body,
                          'topics': self.topics})
        self._reset()

    def start_title(self, attributes):
        self.in_title = 1

    def end_title(self):
        self.in_title = 0

    def start_body(self, attributes):
        self.in_body = 1

    def end_body(self):
        self.in_body = 0

    def start_topics(self, attributes):
        self.in_topics = 1

    def end_topics(self):
        self.in_topics = 0
```

(continues on next page)

(continued from previous page)

```
    def start_d(self, attributes):
        self.in_topic_d = 1

    def end_d(self):
        self.in_topic_d = 0
        self.topics.append(self.topic_d)
        self.topic_d = ""


def stream_reuters_documents(data_path=None):
    """Iterate over documents of the Reuters dataset.

    The Reuters archive will automatically be downloaded and uncompressed if
    the `data_path` directory does not exist.

    Documents are represented as dictionaries with 'body' (str),
    'title' (str), 'topics' (list(str)) keys.

    """

    DOWNLOAD_URL = ('http://archive.ics.uci.edu/ml/machine-learning-databases/'
                    'reuters21578-mld/reuters21578.tar.gz')
    ARCHIVE_FILENAME = 'reuters21578.tar.gz'

    if data_path is None:
        data_path = os.path.join(get_data_home(), "reuters")
    if not os.path.exists(data_path):
        """Download the dataset."""
        print("downloading dataset (once and for all) into %s" %
              data_path)
        os.mkdir(data_path)

        def progress(blocknum, bs, size):
            total_sz_mb = '%.2f MB' % (size / 1e6)
            current_sz_mb = '%.2f MB' % ((blocknum * bs) / 1e6)
            if _not_in_sphinx():
                sys.stdout.write(
                    '\rdownloaded %s / %s' % (current_sz_mb, total_sz_mb))

        archive_path = os.path.join(data_path, ARCHIVE_FILENAME)
        urlretrieve(DOWNLOAD_URL, filename=archive_path,
                    reporthook=progress)
        if _not_in_sphinx():
            sys.stdout.write('\r')
        print("untarring Reuters dataset...")
        tarfile.open(archive_path, 'r:gz').extractall(data_path)
        print("done.")

    parser = ReutersParser()
    for filename in glob(os.path.join(data_path, "*.sgm")):
        for doc in parser.parse(open(filename, 'rb')):
            yield doc
```

Main

Create the vectorizer and limit the number of features to a reasonable maximum

```
vectorizer = HashingVectorizer(decode_error='ignore', n_features=2 ** 18,
                               alternate_sign=False)


# Iterator over parsed Reuters SGML files.
data_stream = stream_reuters_documents()

# We learn a binary classification between the "acq" class and all the others.
# "acq" was chosen as it is more or less evenly distributed in the Reuters
# files. For other datasets, one should take care of creating a test set with
# a realistic portion of positive instances.
all_classes = np.array([0, 1])
positive_class = 'acq'

# Here are some classifiers that support the `partial_fit` method
partial_fit_classifiers = {
    'SGD': SGDClassifier(max_iter=5),
    'Perceptron': Perceptron(),
    'NB Multinomial': MultinomialNB(alpha=0.01),
    'Passive-Aggressive': PassiveAggressiveClassifier(),
}


def get_minibatch(doc_iter, size, pos_class=positive_class):
    """Extract a minibatch of examples, return a tuple X_text, y.

    Note: size is before excluding invalid docs with no topics assigned.

    """
    data = [('{title}\n\n{body}'.format(**doc), pos_class in doc['topics'])
            for doc in itertools.islice(doc_iter, size)
            if doc['topics']]
    if not len(data):
        return np.asarray([], dtype=int), np.asarray([], dtype=int)
    X_text, y = zip(*data)
    return X_text, np.asarray(y, dtype=int)


def iter_minibatches(doc_iter, minibatch_size):
    """Generator of minibatches."""
    X_text, y = get_minibatch(doc_iter, minibatch_size)
    while len(X_text):
        yield X_text, y
        X_text, y = get_minibatch(doc_iter, minibatch_size)


# test data statistics
test_stats = {'n_test': 0, 'n_test_pos': 0}

# First we hold out a number of examples to estimate accuracy
n_test_documents = 1000
tick = time.time()
X_test_text, y_test = get_minibatch(data_stream, 1000)
parsing_time = time.time() - tick
```

(continues on next page)

(continued from previous page)

```
tick = time.time()
X_test = vectorizer.transform(X_test_text)
vectorizing_time = time.time() - tick
test_stats['n_test'] += len(y_test)
test_stats['n_test_pos'] += sum(y_test)
print("Test set is %d documents (%d positive)" % (len(y_test), sum(y_test)))


def progress(cls_name, stats):
    """Report progress information, return a string."""
    duration = time.time() - stats['t0']
    s = "%20s classifier : \t" % cls_name
    s += "%(n_train)6d train docs (%(n_train_pos)6d positive) " % stats
    s += "%(n_test)6d test docs (%(n_test_pos)6d positive) " % test_stats
    s += "accuracy: %(accuracy).3f " % stats
    s += "in %.2fs (%5d docs/s)" % (duration, stats['n_train'] / duration)
    return s


cls_stats = {}

for cls_name in partial_fit_classifiers:
    stats = {'n_train': 0, 'n_train_pos': 0,
             'accuracy': 0.0, 'accuracy_history': [(0, 0)], 't0': time.time(),
             'runtime_history': [(0, 0)], 'total_fit_time': 0.0}
    cls_stats[cls_name] = stats

get_minibatch(data_stream, n_test_documents)
# Discard test set

# We will feed the classifier with mini-batches of 1000 documents; this means
# we have at most 1000 docs in memory at any time.  The smaller the document
# batch, the bigger the relative overhead of the partial fit methods.
minibatch_size = 1000

# Create the data_stream that parses Reuters SGML files and iterates on
# documents as a stream.
minibatch_iterators = iter_minibatches(data_stream, minibatch_size)
total_vect_time = 0.0

# Main loop : iterate on mini-batches of examples
for i, (X_train_text, y_train) in enumerate(minibatch_iterators):

    tick = time.time()
    X_train = vectorizer.transform(X_train_text)
    total_vect_time += time.time() - tick

    for cls_name, cls in partial_fit_classifiers.items():
        tick = time.time()
        # update estimator with examples in the current mini-batch
        cls.partial_fit(X_train, y_train, classes=all_classes)

        # accumulate test accuracy stats
        cls_stats[cls_name]['total_fit_time'] += time.time() - tick
        cls_stats[cls_name]['n_train'] += X_train.shape[0]
        cls_stats[cls_name]['n_train_pos'] += sum(y_train)
        tick = time.time()
```

(continues on next page)

(continued from previous page)

```
        cls_stats[cls_name]['accuracy'] = cls.score(X_test, y_test)
        cls_stats[cls_name]['prediction_time'] = time.time() - tick
        acc_history = (cls_stats[cls_name]['accuracy'],
                       cls_stats[cls_name]['n_train'])
        cls_stats[cls_name]['accuracy_history'].append(acc_history)
        run_history = (cls_stats[cls_name]['accuracy'],
                       total_vect_time + cls_stats[cls_name]['total_fit_time'])
        cls_stats[cls_name]['runtime_history'].append(run_history)

        if i % 3 == 0:
            print(progress(cls_name, cls_stats[cls_name]))
    if i % 3 == 0:
        print('\n')
```

Out:

```
Test set is 973 documents (125 positive)
                SGD classifier :          965 train docs (   134 positive)     973␣
↪test docs (   125 positive) accuracy: 0.921 in 0.68s ( 1419 docs/s)
          Perceptron classifier :          965 train docs (   134 positive)     973␣
↪test docs (   125 positive) accuracy: 0.872 in 0.69s ( 1407 docs/s)
     NB Multinomial classifier :          965 train docs (   134 positive)     973␣
↪test docs (   125 positive) accuracy: 0.874 in 0.71s ( 1355 docs/s)
  Passive-Aggressive classifier :          965 train docs (   134 positive)     973␣
↪test docs (   125 positive) accuracy: 0.920 in 0.71s ( 1350 docs/s)


                SGD classifier :         3790 train docs (   506 positive)     973␣
↪test docs (   125 positive) accuracy: 0.963 in 1.96s ( 1932 docs/s)
          Perceptron classifier :         3790 train docs (   506 positive)     973␣
↪test docs (   125 positive) accuracy: 0.949 in 1.96s ( 1929 docs/s)
     NB Multinomial classifier :         3790 train docs (   506 positive)     973␣
↪test docs (   125 positive) accuracy: 0.884 in 1.98s ( 1912 docs/s)
  Passive-Aggressive classifier :         3790 train docs (   506 positive)     973␣
↪test docs (   125 positive) accuracy: 0.947 in 1.98s ( 1910 docs/s)


                SGD classifier :         6523 train docs (   916 positive)     973␣
↪test docs (   125 positive) accuracy: 0.950 in 3.21s ( 2031 docs/s)
          Perceptron classifier :         6523 train docs (   916 positive)     973␣
↪test docs (   125 positive) accuracy: 0.923 in 3.21s ( 2030 docs/s)
     NB Multinomial classifier :         6523 train docs (   916 positive)     973␣
↪test docs (   125 positive) accuracy: 0.909 in 3.23s ( 2019 docs/s)
  Passive-Aggressive classifier :         6523 train docs (   916 positive)     973␣
↪test docs (   125 positive) accuracy: 0.953 in 3.23s ( 2017 docs/s)


                SGD classifier :         9434 train docs (  1242 positive)     973␣
↪test docs (   125 positive) accuracy: 0.927 in 4.48s ( 2107 docs/s)
          Perceptron classifier :         9434 train docs (  1242 positive)     973␣
↪test docs (   125 positive) accuracy: 0.947 in 4.48s ( 2106 docs/s)
     NB Multinomial classifier :         9434 train docs (  1242 positive)     973␣
↪test docs (   125 positive) accuracy: 0.918 in 4.50s ( 2098 docs/s)
  Passive-Aggressive classifier :         9434 train docs (  1242 positive)     973␣
↪test docs (   125 positive) accuracy: 0.956 in 4.50s ( 2096 docs/s)


```

(continues on next page)

(continued from previous page)

```
                SGD classifier :        11845 train docs (  1468 positive)     973␣
↪test docs (   125 positive) accuracy: 0.949 in 5.75s ( 2061 docs/s)
         Perceptron classifier :        11845 train docs (  1468 positive)     973␣
↪test docs (   125 positive) accuracy: 0.942 in 5.75s ( 2060 docs/s)
      NB Multinomial classifier :        11845 train docs (  1468 positive)     973␣
↪test docs (   125 positive) accuracy: 0.922 in 5.77s ( 2054 docs/s)
  Passive-Aggressive classifier :        11845 train docs (  1468 positive)     973␣
↪test docs (   125 positive) accuracy: 0.942 in 5.77s ( 2053 docs/s)


                SGD classifier :        14770 train docs (  1856 positive)     973␣
↪test docs (   125 positive) accuracy: 0.959 in 7.10s ( 2079 docs/s)
         Perceptron classifier :        14770 train docs (  1856 positive)     973␣
↪test docs (   125 positive) accuracy: 0.956 in 7.11s ( 2078 docs/s)
      NB Multinomial classifier :        14770 train docs (  1856 positive)     973␣
↪test docs (   125 positive) accuracy: 0.925 in 7.12s ( 2073 docs/s)
  Passive-Aggressive classifier :        14770 train docs (  1856 positive)     973␣
↪test docs (   125 positive) accuracy: 0.957 in 7.13s ( 2072 docs/s)


                SGD classifier :        17723 train docs (  2218 positive)     973␣
↪test docs (   125 positive) accuracy: 0.959 in 8.47s ( 2093 docs/s)
         Perceptron classifier :        17723 train docs (  2218 positive)     973␣
↪test docs (   125 positive) accuracy: 0.938 in 8.47s ( 2092 docs/s)
      NB Multinomial classifier :        17723 train docs (  2218 positive)     973␣
↪test docs (   125 positive) accuracy: 0.925 in 8.49s ( 2087 docs/s)
  Passive-Aggressive classifier :        17723 train docs (  2218 positive)     973␣
↪test docs (   125 positive) accuracy: 0.962 in 8.49s ( 2087 docs/s)
```

Plot results

The plot represents the learning curve of the classifier: the evolution of classification accuracy over the course of the mini-batches. Accuracy is measured on the first 1000 samples, held out as a validation set.

To limit the memory consumption, we queue examples up to a fixed amount before feeding them to the learner.

```
def plot_accuracy(x, y, x_legend):
    """Plot accuracy as a function of x."""
    x = np.array(x)
    y = np.array(y)
    plt.title('Classification accuracy as a function of %s' % x_legend)
    plt.xlabel('%s' % x_legend)
    plt.ylabel('Accuracy')
    plt.grid(True)
    plt.plot(x, y)


rcParams['legend.fontsize'] = 10
cls_names = list(sorted(cls_stats.keys()))

# Plot accuracy evolution
plt.figure()
for _, stats in sorted(cls_stats.items()):
    # Plot accuracy evolution with #examples
    accuracy, n_examples = zip(*stats['accuracy_history'])
```

(continues on next page)

(continued from previous page)

```
    plot_accuracy(n_examples, accuracy, "training examples (#)")
    ax = plt.gca()
    ax.set_ylim((0.8, 1))
plt.legend(cls_names, loc='best')

plt.figure()
for _, stats in sorted(cls_stats.items()):
    # Plot accuracy evolution with runtime
    accuracy, runtime = zip(*stats['runtime_history'])
    plot_accuracy(runtime, accuracy, 'runtime (s)')
    ax = plt.gca()
    ax.set_ylim((0.8, 1))
plt.legend(cls_names, loc='best')

# Plot fitting times
plt.figure()
fig = plt.gcf()
cls_runtime = [stats['total_fit_time']
               for cls_name, stats in sorted(cls_stats.items())]

cls_runtime.append(total_vect_time)
cls_names.append('Vectorization')
bar_colors = ['b', 'g', 'r', 'c', 'm', 'y']

ax = plt.subplot(111)
rectangles = plt.bar(range(len(cls_names)), cls_runtime, width=0.5,
                     color=bar_colors)

ax.set_xticks(np.linspace(0, len(cls_names) - 1, len(cls_names)))
ax.set_xticklabels(cls_names, fontsize=10)
ymax = max(cls_runtime) * 1.2
ax.set_ylim((0, ymax))
ax.set_ylabel('runtime (s)')
ax.set_title('Training Times')


def autolabel(rectangles):
    """attach some text vi autolabel on rectangles."""
    for rect in rectangles:
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width() / 2.,
                1.05 * height, '%.4f' % height,
                ha='center', va='bottom')
        plt.setp(plt.xticks()[1], rotation=30)


autolabel(rectangles)
plt.tight_layout()
plt.show()

# Plot prediction times
plt.figure()
cls_runtime = []
cls_names = list(sorted(cls_stats.keys()))
for cls_name, stats in sorted(cls_stats.items()):
    cls_runtime.append(stats['prediction_time'])
cls_runtime.append(parsing_time)
```

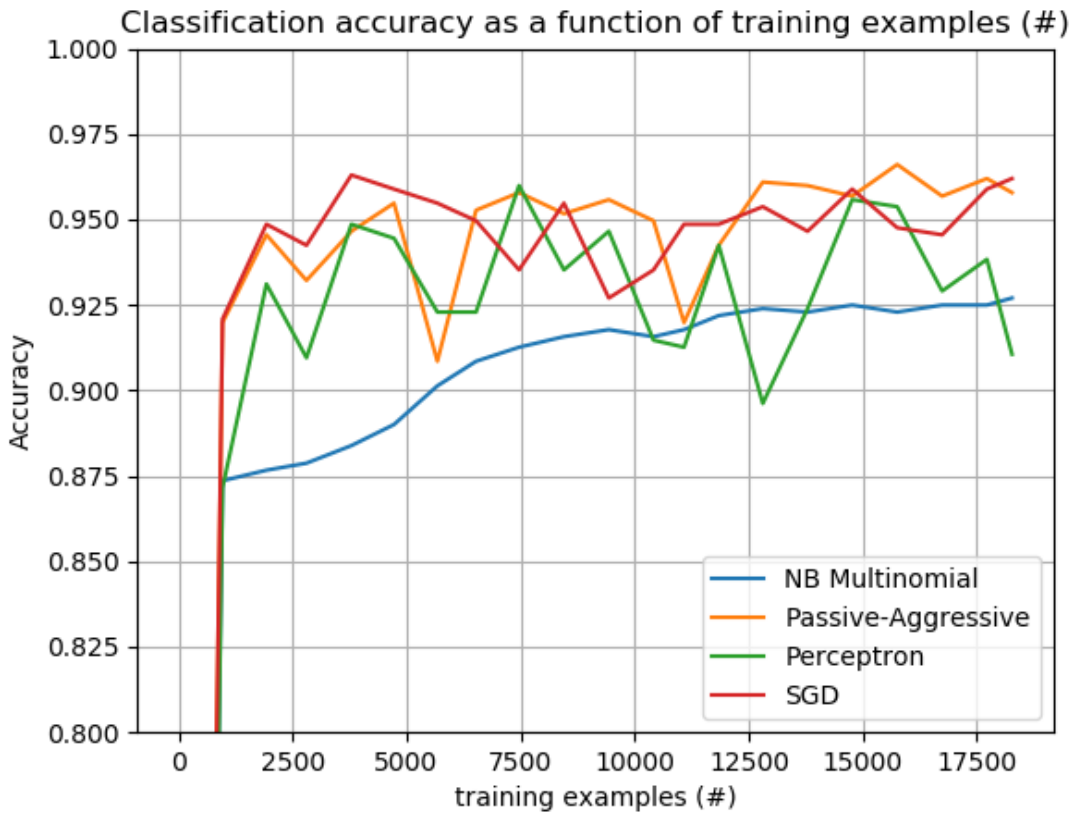
(continues on next page)

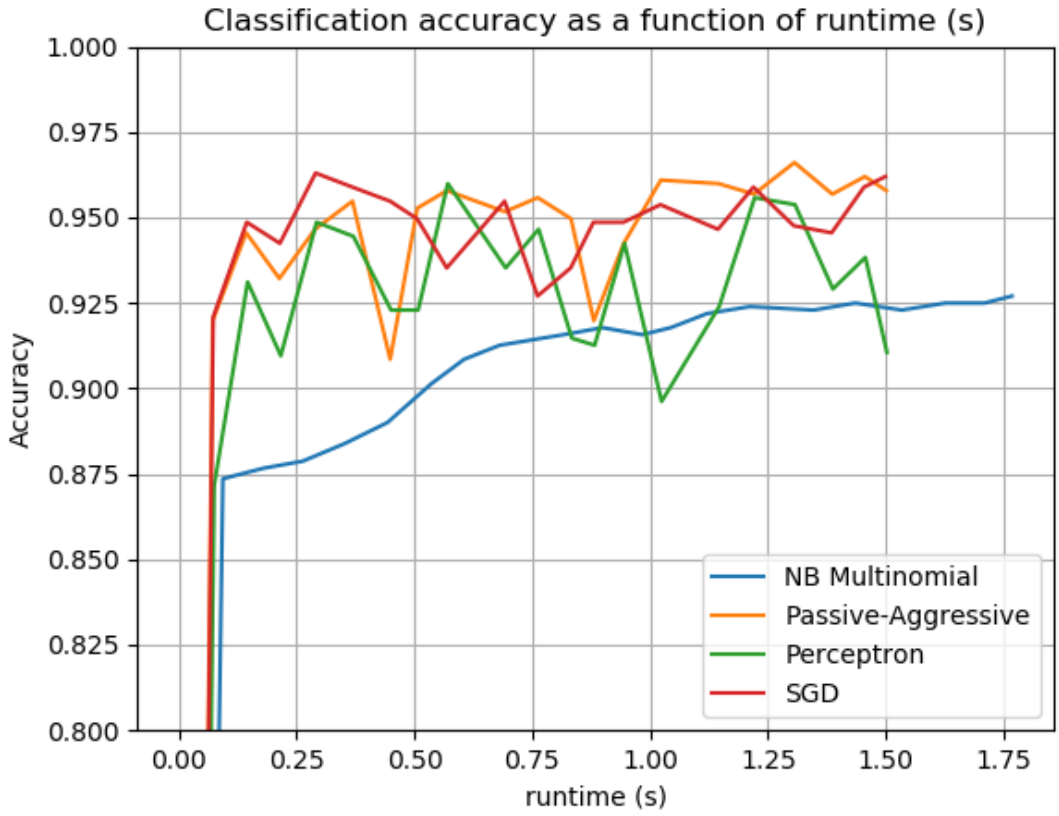
(continued from previous page)

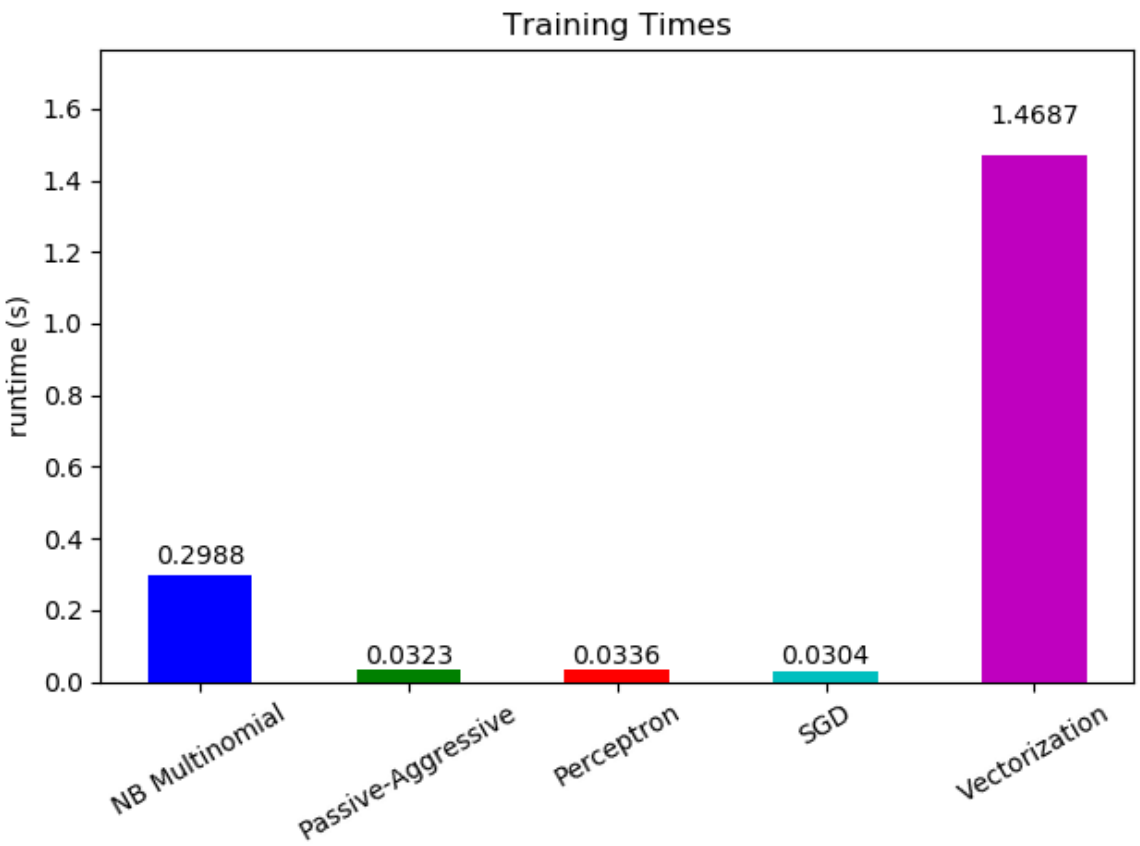
```
cls_names.append('Read/Parse\n+Feat.Extr.')
cls_runtime.append(vectorizing_time)
cls_names.append('Hashing\n+Vect.')

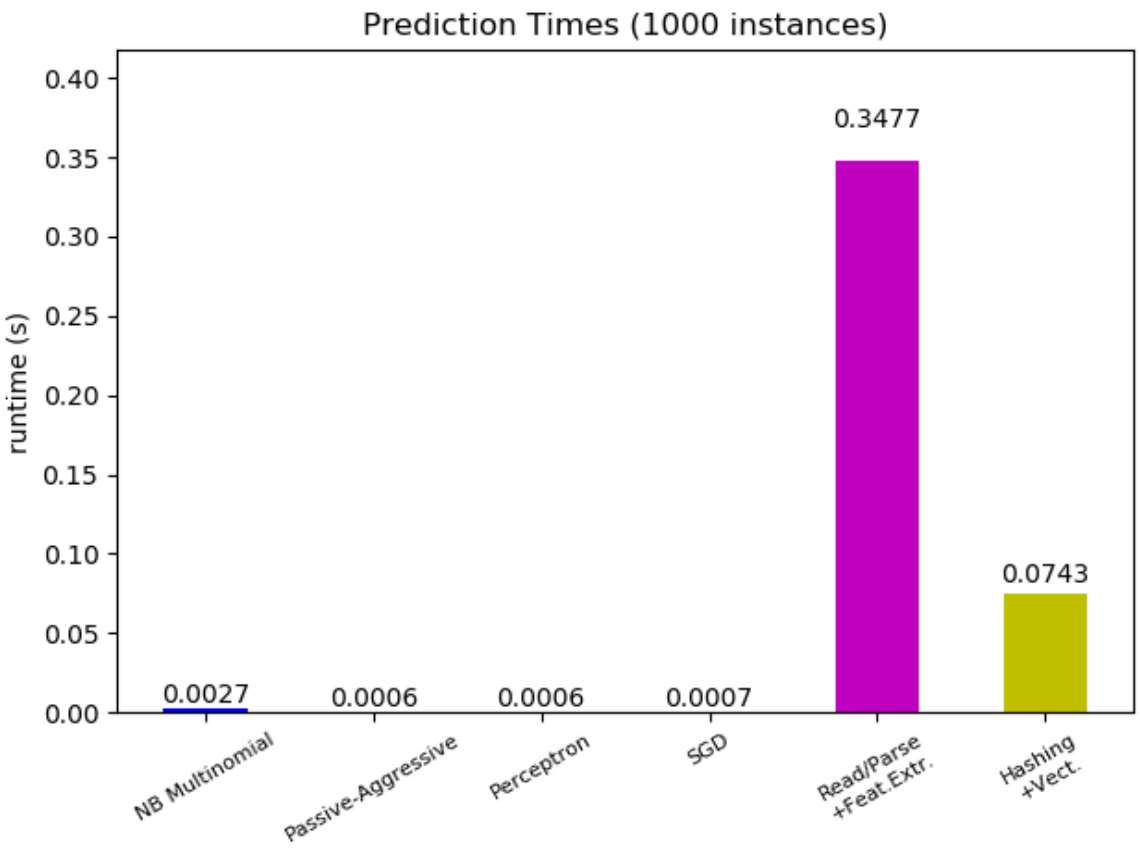
ax = plt.subplot(111)
rectangles = plt.bar(range(len(cls_names)), cls_runtime, width=0.5,
                     color=bar_colors)

ax.set_xticks(np.linspace(0, len(cls_names) - 1, len(cls_names)))
ax.set_xticklabels(cls_names, fontsize=8)
plt.setp(plt.xticks()[1], rotation=30)
ymax = max(cls_runtime) * 1.2
ax.set_ylim((0, ymax))
ax.set_ylabel('runtime (s)')
ax.set_title('Prediction Times (%d instances)' % n_test_documents)
autolabel(rectangles)
plt.tight_layout()
plt.show()
```




Classification accuracy as a function of runtime (s)
Accuracy
runtime (s)
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0.800
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
NB Multinomial
Passive-Aggressive
Perceptron
SGD


Training Times
runtime (s)
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.2988
0.0323
0.0336
0.0304
1.4687
NB Multinomial
Passive-Aggressive
Perceptron
SGD
Vectorization



Total running time of the script: (0 minutes 10.253 seconds)

Estimated memory usage: 8 MB

6.13 Feature Selection

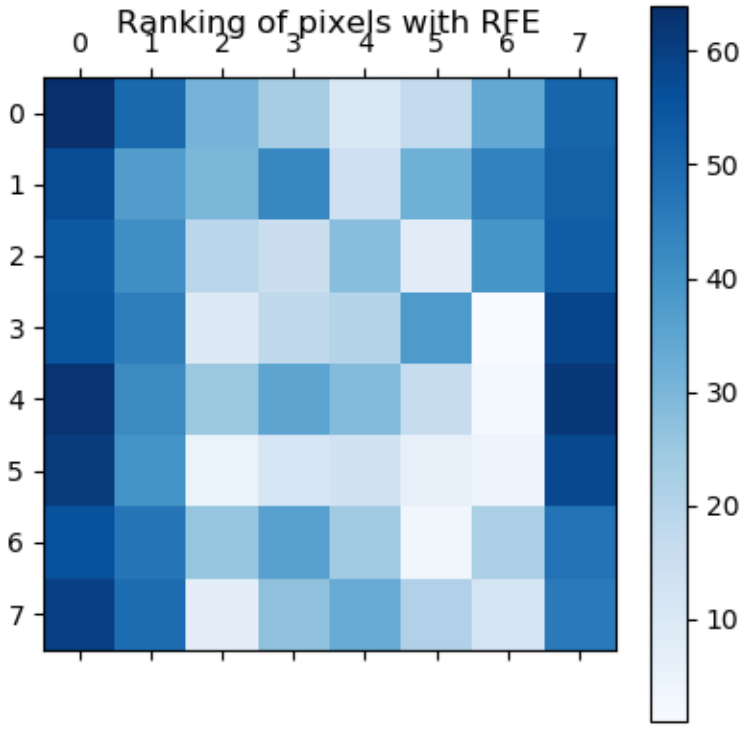
Examples concerning the `sklearn.feature_selection` module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.13.1 Recursive feature elimination

A recursive feature elimination example showing the relevance of pixels in a digit classification task.

Note: See also *Recursive feature elimination with cross-validation*




```
print(__doc__)

from sklearn.svm import SVC
from sklearn.datasets import load_digits
from sklearn.feature_selection import RFE
import matplotlib.pyplot as plt

# Load the digits dataset
digits = load_digits()
X = digits.images.reshape((len(digits.images), -1))
y = digits.target

# Create the RFE object and rank each pixel
svc = SVC(kernel="linear", C=1)
rfe = RFE(estimator=svc, n_features_to_select=1, step=1)
rfe.fit(X, y)
ranking = rfe.ranking_.reshape(digits.images[0].shape)

# Plot pixel ranking
plt.matshow(ranking, cmap=plt.cm.Blues)
plt.colorbar()
plt.title("Ranking of pixels with RFE")
plt.show()
```

Total running time of the script: (0 minutes 3.766 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.13.2 Comparison of F-test and mutual information

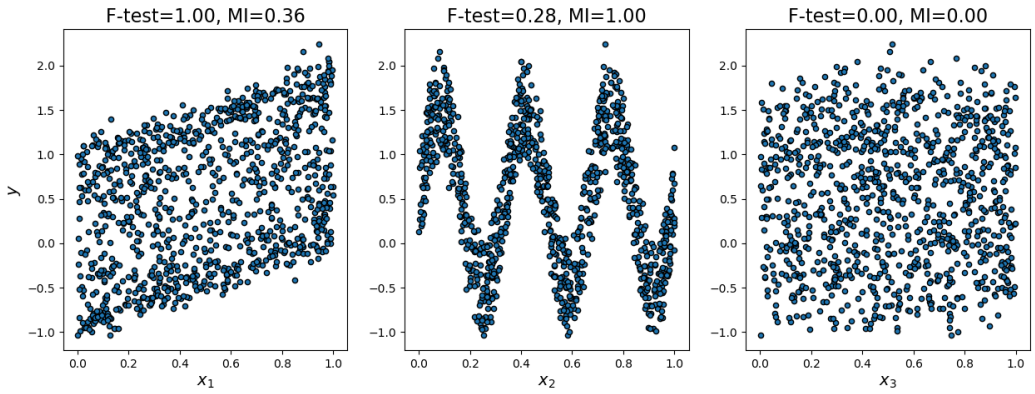
This example illustrates the differences between univariate F-test statistics and mutual information.

We consider 3 features x_1, x_2, x_3 distributed uniformly over [0, 1], the target depends on them as follows:

y = x_1 + sin(6 * pi * x_2) + 0.1 * N(0, 1), that is the third features is completely irrelevant.

The code below plots the dependency of y against individual x_i and normalized values of univariate F-tests statistics and mutual information.

As F-test captures only linear dependency, it rates x_1 as the most discriminative feature. On the other hand, mutual information can capture any kind of dependency between variables and it rates x_2 as the most discriminative feature, which probably agrees better with our intuitive perception for this example. Both methods correctly marks x_3 as irrelevant.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.feature_selection import f_regression, mutual_info_regression

np.random.seed(0)
X = np.random.rand(1000, 3)
y = X[:, 0] + np.sin(6 * np.pi * X[:, 1]) + 0.1 * np.random.randn(1000)

f_test, _ = f_regression(X, y)
f_test /= np.max(f_test)

mi = mutual_info_regression(X, y)
mi /= np.max(mi)

plt.figure(figsize=(15, 5))
for i in range(3):
    plt.subplot(1, 3, i + 1)
    plt.scatter(X[:, i], y, edgecolor='black', s=20)
    plt.xlabel("$x_{}$".format(i + 1), fontsize=14)
```

(continues on next page)

(continued from previous page)

```
    if i == 0:
        plt.ylabel("$y$", fontsize=14)
    plt.title("F-test={:.2f}, MI={:.2f}".format(f_test[i], mi[i]),
              fontsize=16)
plt.show()
```

Total running time of the script: (0 minutes 0.582 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.13.3 Pipeline Anova SVM

Simple usage of Pipeline that runs successively a univariate feature selection with anova and then a SVM of the selected features.

Using a sub-pipeline, the fitted coefficients can be mapped back into the original feature space.

Out:

```
              precision    recall  f1-score   support

           0       0.75      0.50      0.60         6
           1       0.67      1.00      0.80         6
           2       0.67      0.80      0.73         5
           3       1.00      0.75      0.86         8

    accuracy                           0.76        25
   macro avg       0.77      0.76      0.75        25
weighted avg       0.79      0.76      0.76        25

[[-0.23912131  0.          0.          0.         -0.3236911   0.
   0.          0.          0.          0.          0.          0.
   0.10836648  0.          0.          0.          0.          0.
   0.          0.        ]
 [ 0.43878747  0.          0.          0.         -0.51415652  0.
   0.          0.          0.          0.          0.          0.
   0.04845652  0.          0.          0.          0.          0.
   0.          0.        ]
 [-0.65382998  0.          0.          0.          0.57962856  0.
   0.          0.          0.          0.          0.          0.
  -0.04736524  0.          0.          0.          0.          0.
   0.          0.        ]
 [ 0.54403412  0.          0.          0.          0.58478491  0.
   0.          0.          0.          0.          0.          0.
  -0.11344659  0.          0.          0.          0.          0.
   0.          0.        ]]
```

```
from sklearn import svm
from sklearn.datasets import make_classification
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

print(__doc__)

# import some data to play with
X, y = make_classification(
    n_features=20, n_informative=3, n_redundant=0, n_classes=4,
    n_clusters_per_class=2)

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

# ANOVA SVM-C
# 1) anova filter, take 3 best ranked features
anova_filter = SelectKBest(f_regression, k=3)
# 2) svm
clf = svm.LinearSVC()

anova_svm = make_pipeline(anova_filter, clf)
anova_svm.fit(X_train, y_train)
y_pred = anova_svm.predict(X_test)
print(classification_report(y_test, y_pred))

coef = anova_svm[:-1].inverse_transform(anova_svm['linearsvc'].coef_)
print(coef)
```

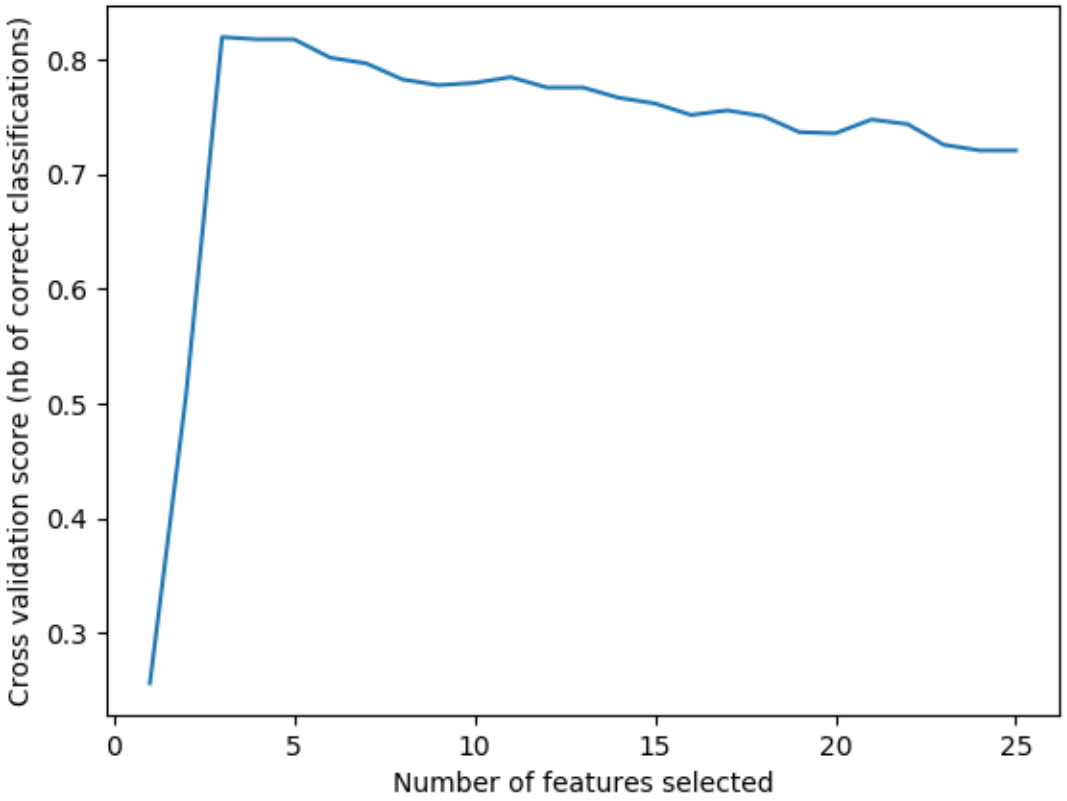
Total running time of the script: (0 minutes 0.341 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.13.4 Recursive feature elimination with cross-validation

A recursive feature elimination example with automatic tuning of the number of features selected with cross-validation.



Out:

```
Optimal number of features : 3
```

```
print(__doc__)

import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.model_selection import StratifiedKFold
from sklearn.feature_selection import RFECV
from sklearn.datasets import make_classification

# Build a classification task using 3 informative features
X, y = make_classification(n_samples=1000, n_features=25, n_informative=3,
                           n_redundant=2, n_repeated=0, n_classes=8,
                           n_clusters_per_class=1, random_state=0)

# Create the RFE object and compute a cross-validated score.
svc = SVC(kernel="linear")
# The "accuracy" scoring is proportional to the number of correct
```

(continues on next page)

(continued from previous page)

```
# classifications
rfecv = RFECV(estimator=svc, step=1, cv=StratifiedKFold(2),
              scoring='accuracy')
rfecv.fit(X, y)

print("Optimal number of features : %d" % rfecv.n_features_)

# Plot number of features VS. cross-validation scores
plt.figure()
plt.xlabel("Number of features selected")
plt.ylabel("Cross validation score (nb of correct classifications)")
plt.plot(range(1, len(rfecv.grid_scores_) + 1), rfecv.grid_scores_)
plt.show()
```

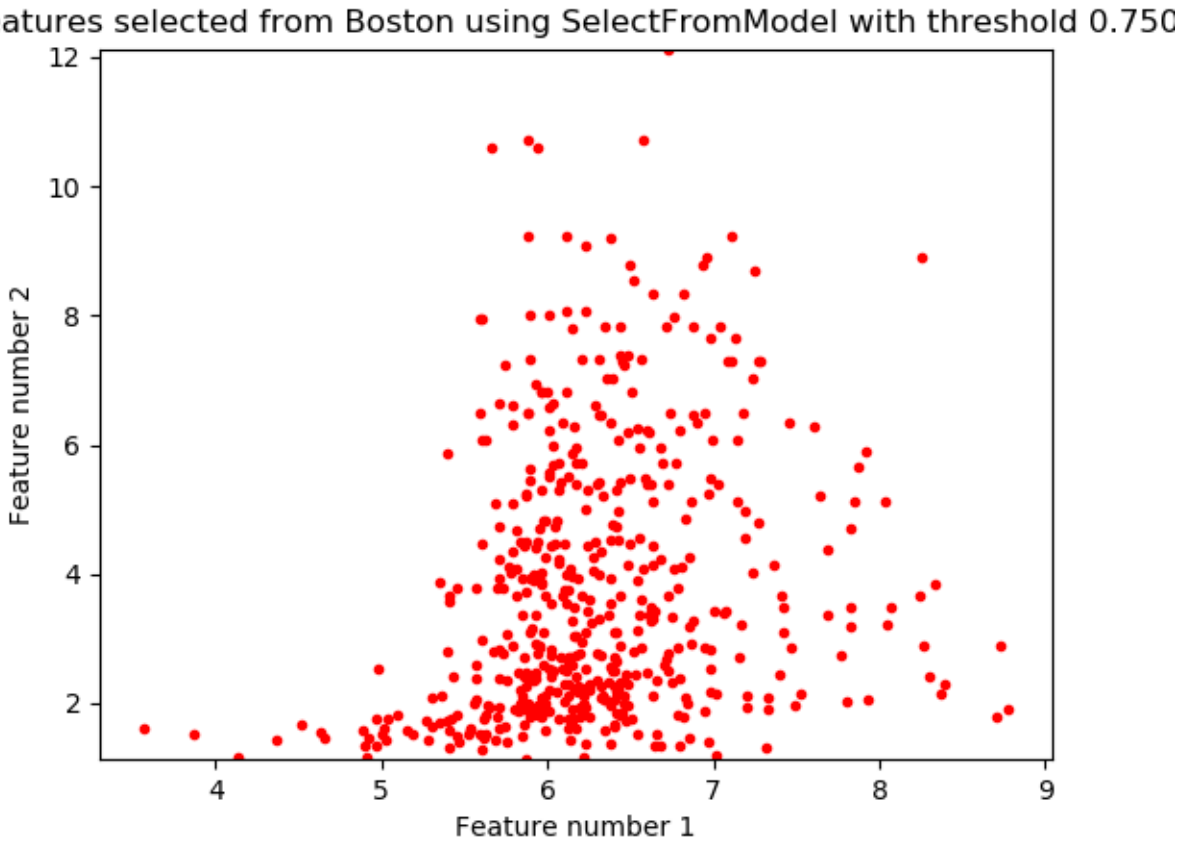
Total running time of the script: (0 minutes 1.984 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.13.5 Feature selection using SelectFromModel and LassoCV

Use SelectFromModel meta-transformer along with Lasso to select the best couple of features from the Boston dataset.




```
# Author: Manoj Kumar <mks542@nyu.edu>
# License: BSD 3 clause

print(__doc__)

import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import load_boston
from sklearn.feature_selection import SelectFromModel
from sklearn.linear_model import LassoCV

# Load the boston dataset.
X, y = load_boston(return_X_y=True)

# We use the base estimator LassoCV since the L1 norm promotes sparsity of features.
clf = LassoCV()

# Set a minimum threshold of 0.25
sfm = SelectFromModel(clf, threshold=0.25)
sfm.fit(X, y)
n_features = sfm.transform(X).shape[1]

# Reset the threshold till the number of features equals two.
# Note that the attribute can be set directly instead of repeatedly
```

(continues on next page)

(continued from previous page)

```
# fitting the metatransformer.
while n_features > 2:
    sfm.threshold += 0.1
    X_transform = sfm.transform(X)
    n_features = X_transform.shape[1]

# Plot the selected two features from X.
plt.title(
    "Features selected from Boston using SelectFromModel with "
    "threshold %0.3f." % sfm.threshold)
feature1 = X_transform[:, 0]
feature2 = X_transform[:, 1]
plt.plot(feature1, feature2, 'r.')
plt.xlabel("Feature number 1")
plt.ylabel("Feature number 2")
plt.ylim([np.min(feature2), np.max(feature2)])
plt.show()
```

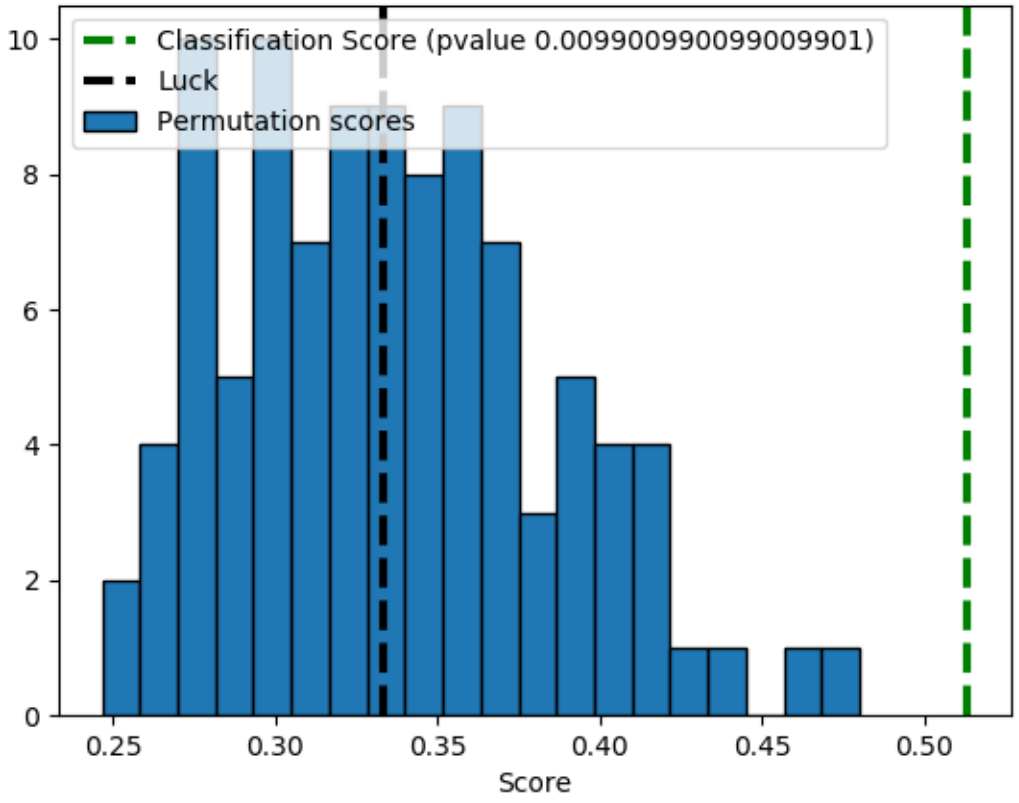
Total running time of the script: (0 minutes 0.455 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.13.6 Test with permutations the significance of a classification score

In order to test if a classification score is significative a technique in repeating the classification procedure after randomizing, permuting, the labels. The p-value is then given by the percentage of runs for which the score obtained is greater than the classification score obtained in the first place.



Out:

```
Classification score 0.5133333333333333 (pvalue : 0.009900990099009901)
```

```
# Author:  Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.svm import SVC
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import permutation_test_score
from sklearn import datasets


# ##############################################################################
# Loading a dataset
```

(continues on next page)

(continued from previous page)

```
iris = datasets.load_iris()
X = iris.data
y = iris.target
n_classes = np.unique(y).size

# Some noisy data not correlated
random = np.random.RandomState(seed=0)
E = random.normal(size=(len(X), 2200))

# Add noisy data to the informative features for make the task harder
X = np.c_[X, E]

svm = SVC(kernel='linear')
cv = StratifiedKFold(2)

score, permutation_scores, pvalue = permutation_test_score(
    svm, X, y, scoring="accuracy", cv=cv, n_permutations=100, n_jobs=1)

print("Classification score %s (pvalue : %s)" % (score, pvalue))

# #############################################################################
# View histogram of permutation scores
plt.hist(permutation_scores, 20, label='Permutation scores',
         edgecolor='black')
ylim = plt.ylim()
# BUG: vlines(..., linestyle='--') fails on older versions of matplotlib
# plt.vlines(score, ylim[0], ylim[1], linestyle='--',
#          color='g', linewidth=3, label='Classification Score'
#          ' (pvalue %s)' % pvalue)
# plt.vlines(1.0 / n_classes, ylim[0], ylim[1], linestyle='--',
#          color='k', linewidth=3, label='Luck')
plt.plot(2 * [score], ylim, '--g', linewidth=3,
         label='Classification Score'
         ' (pvalue %s)' % pvalue)
plt.plot(2 * [1. / n_classes], ylim, '--k', linewidth=3, label='Luck')

plt.ylim(ylim)
plt.legend()
plt.xlabel('Score')
plt.show()
```

Total running time of the script: (0 minutes 7.920 seconds)

Estimated memory usage: 8 MB

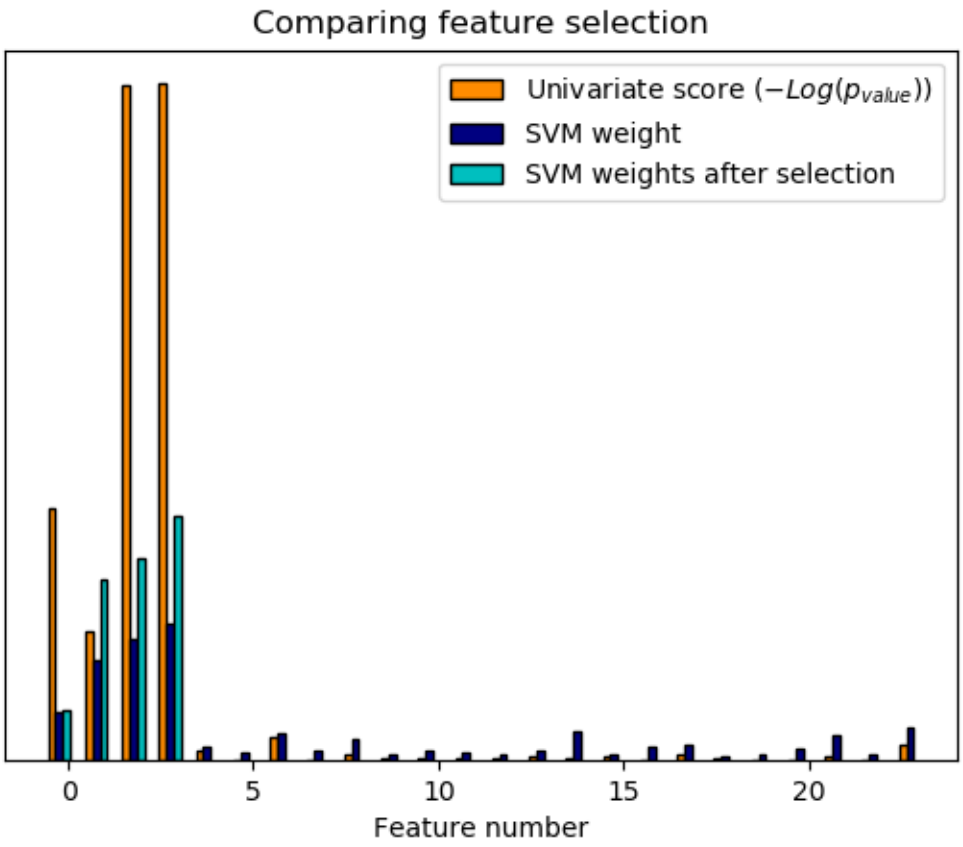
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.13.7 Univariate Feature Selection

An example showing univariate feature selection.

Noisy (non informative) features are added to the iris data and univariate feature selection is applied. For each feature, we plot the p-values for the univariate feature selection and the corresponding weights of an SVM. We can see that univariate feature selection selects the informative features and that these have larger SVM weights.

In the total set of features, only the 4 first ones are significant. We can see that they have the highest score with univariate feature selection. The SVM assigns a large weight to one of these features, but also Selects many of the non-informative features. Applying univariate feature selection before the SVM increases the SVM weight attributed to the significant features, and will thus improve classification.



Out:

```
Classification accuracy without selecting features: 0.789
Classification accuracy after univariate feature selection: 0.868
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import LinearSVC
from sklearn.pipeline import make_pipeline
```

(continues on next page)

(continued from previous page)

```
from sklearn.feature_selection import SelectKBest, f_classif

# #############################################################################
# Import some data to play with

# The iris dataset
X, y = load_iris(return_X_y=True)

# Some noisy data not correlated
E = np.random.RandomState(42).uniform(0, 0.1, size=(X.shape[0], 20))

# Add the noisy data to the informative features
X = np.hstack((X, E))

# Split dataset to select feature and evaluate the classifier
X_train, X_test, y_train, y_test = train_test_split(
        X, y, stratify=y, random_state=0
)

plt.figure(1)
plt.clf()

X_indices = np.arange(X.shape[-1])

# #############################################################################
# Univariate feature selection with F-test for feature scoring
# We use the default selection function to select the four
# most significant features
selector = SelectKBest(f_classif, k=4)
selector.fit(X_train, y_train)
scores = -np.log10(selector.pvalues_)
scores /= scores.max()
plt.bar(X_indices - .45, scores, width=.2,
        label=r'Univariate score ($-Log(p_{value})$)', color='darkorange',
        edgecolor='black')

# #############################################################################
# Compare to the weights of an SVM
clf = make_pipeline(MinMaxScaler(), LinearSVC())
clf.fit(X_train, y_train)
print('Classification accuracy without selecting features: {:.3f}'
      .format(clf.score(X_test, y_test)))

svm_weights = np.abs(clf[-1].coef_).sum(axis=0)
svm_weights /= svm_weights.sum()

plt.bar(X_indices - .25, svm_weights, width=.2, label='SVM weight',
        color='navy', edgecolor='black')

clf_selected = make_pipeline(
        SelectKBest(f_classif, k=4), MinMaxScaler(), LinearSVC()
)
clf_selected.fit(X_train, y_train)
print('Classification accuracy after univariate feature selection: {:.3f}'
      .format(clf_selected.score(X_test, y_test)))

svm_weights_selected = np.abs(clf_selected[-1].coef_).sum(axis=0)
```

(continues on next page)

(continued from previous page)

```
svm_weights_selected /= svm_weights_selected.sum()

plt.bar(X_indices[selector.get_support()] - .05, svm_weights_selected,
        width=.2, label='SVM weights after selection', color='c',
        edgecolor='black')


plt.title("Comparing feature selection")
plt.xlabel('Feature number')
plt.yticks(())
plt.axis('tight')
plt.legend(loc='upper right')
plt.show()
```

Total running time of the script: (0 minutes 0.441 seconds)

Estimated memory usage: 8 MB

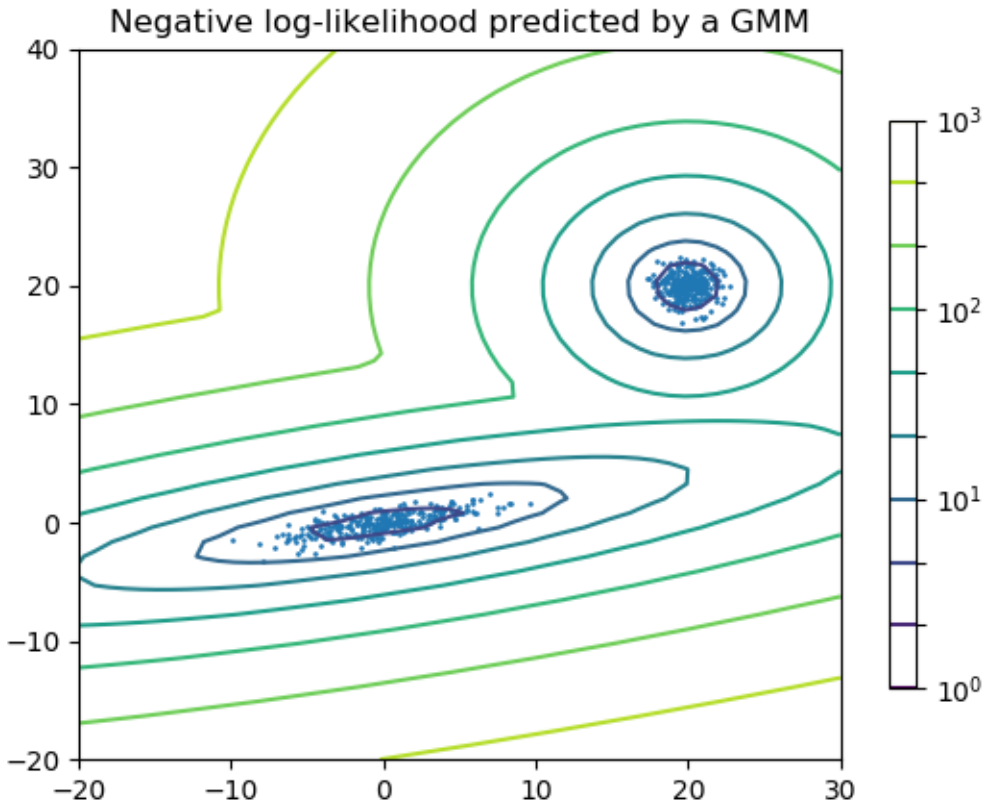
6.14 Gaussian Mixture Models

Examples concerning the *sklearn.mixture* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.14.1 Density Estimation for a Gaussian mixture

Plot the density estimation of a mixture of two Gaussians. Data is generated from two Gaussians with different centers and covariance matrices.




```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
from sklearn import mixture

n_samples = 300

# generate random sample, two components
np.random.seed(0)

# generate spherical data centered on (20, 20)
shifted_gaussian = np.random.randn(n_samples, 2) + np.array([20, 20])

# generate zero centered stretched Gaussian data
C = np.array([[0., -0.7], [3.5, .7]])
stretched_gaussian = np.dot(np.random.randn(n_samples, 2), C)

# concatenate the two datasets into the final training set
X_train = np.vstack([shifted_gaussian, stretched_gaussian])

# fit a Gaussian Mixture Model with two components
clf = mixture.GaussianMixture(n_components=2, covariance_type='full')
clf.fit(X_train)

# display predicted scores by the model as a contour plot
```

(continues on next page)

(continued from previous page)

```
x = np.linspace(-20., 30.)
y = np.linspace(-20., 40.)
X, Y = np.meshgrid(x, y)
XX = np.array([X.ravel(), Y.ravel()]).T
Z = -clf.score_samples(XX)
Z = Z.reshape(X.shape)

CS = plt.contour(X, Y, Z, norm=LogNorm(vmin=1.0, vmax=1000.0),
                 levels=np.logspace(0, 3, 10))
CB = plt.colorbar(CS, shrink=0.8, extend='both')
plt.scatter(X_train[:, 0], X_train[:, 1], .8)

plt.title('Negative log-likelihood predicted by a GMM')
plt.axis('tight')
plt.show()
```

Total running time of the script: (0 minutes 0.472 seconds)

Estimated memory usage: 8 MB

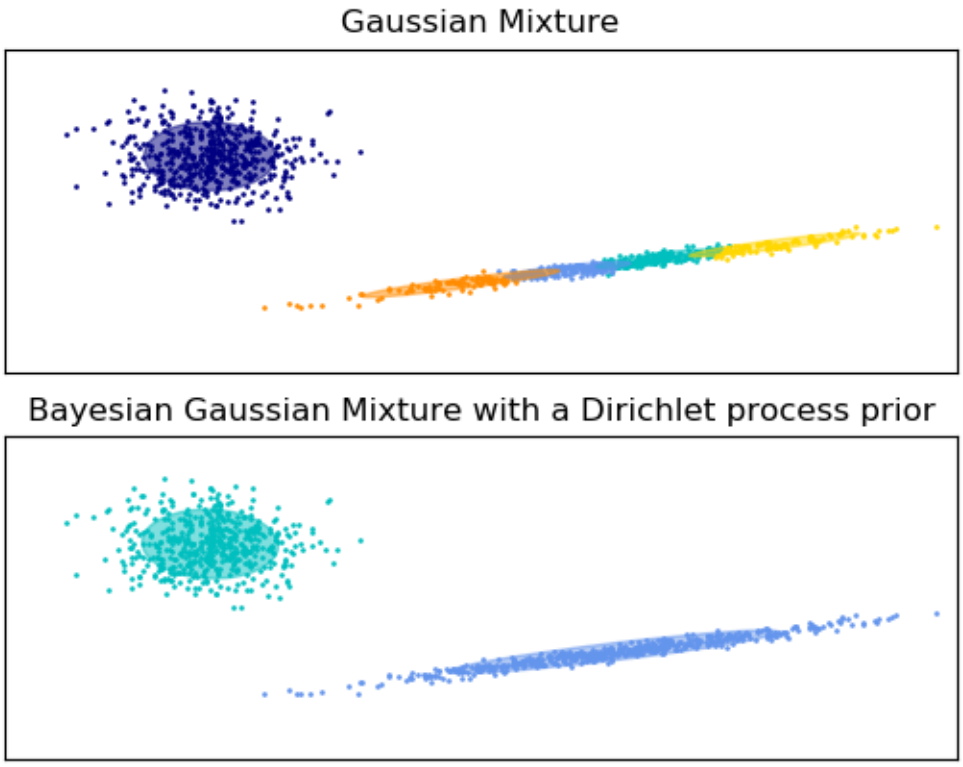
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.14.2 Gaussian Mixture Model Ellipsoids

Plot the confidence ellipsoids of a mixture of two Gaussians obtained with Expectation Maximisation (`GaussianMixture` class) and Variational Inference (`BayesianGaussianMixture` class models with a Dirichlet process prior).

Both models have access to five components with which to fit the data. Note that the Expectation Maximisation model will necessarily use all five components while the Variational Inference model will effectively only use as many as are needed for a good fit. Here we can see that the Expectation Maximisation model splits some components arbitrarily, because it is trying to fit too many components, while the Dirichlet Process model adapts it number of state automatically.

This example doesn't show it, as we're in a low-dimensional space, but another advantage of the Dirichlet process model is that it can fit full covariance matrices effectively even when there are less examples per cluster than there are dimensions in the data, due to regularization properties of the inference algorithm.




```
import itertools

import numpy as np
from scipy import linalg
import matplotlib.pyplot as plt
import matplotlib as mpl

from sklearn import mixture

color_iter = itertools.cycle(['navy', 'c', 'cornflowerblue', 'gold',
                              'darkorange'])


def plot_results(X, Y_, means, covariances, index, title):
    splot = plt.subplot(2, 1, 1 + index)
    for i, (mean, covar, color) in enumerate(zip(
            means, covariances, color_iter)):
        v, w = linalg.eigh(covar)
        v = 2. * np.sqrt(2.) * np.sqrt(v)
        u = w[0] / linalg.norm(w[0])
        # as the DP will not use every component it has access to
        # unless it needs it, we shouldn't plot the redundant
        # components.
        if not np.any(Y_ == i):
            continue
```

(continues on next page)

(continued from previous page)

```
        plt.scatter(X[Y_ == i, 0], X[Y_ == i, 1], .8, color=color)

        # Plot an ellipse to show the Gaussian component
        angle = np.arctan(u[1] / u[0])
        angle = 180. * angle / np.pi  # convert to degrees
        ell = mpl.patches.Ellipse(mean, v[0], v[1], 180. + angle, color=color)
        ell.set_clip_box(splot.bbox)
        ell.set_alpha(0.5)
        splot.add_artist(ell)

    plt.xlim(-9., 5.)
    plt.ylim(-3., 6.)
    plt.xticks(())
    plt.yticks(())
    plt.title(title)


# Number of samples per component
n_samples = 500

# Generate random sample, two components
np.random.seed(0)
C = np.array([[0., -0.1], [1.7, .4]])
X = np.r_[np.dot(np.random.randn(n_samples, 2), C),
          .7 * np.random.randn(n_samples, 2) + np.array([-6, 3])]

# Fit a Gaussian mixture with EM using five components
gmm = mixture.GaussianMixture(n_components=5, covariance_type='full').fit(X)
plot_results(X, gmm.predict(X), gmm.means_, gmm.covariances_, 0,
             'Gaussian Mixture')

# Fit a Dirichlet process Gaussian mixture using five components
dpgmm = mixture.BayesianGaussianMixture(n_components=5,
                                        covariance_type='full').fit(X)
plot_results(X, dpgmm.predict(X), dpgmm.means_, dpgmm.covariances_, 1,
             'Bayesian Gaussian Mixture with a Dirichlet process prior')

plt.show()
```

Total running time of the script: (0 minutes 0.496 seconds)

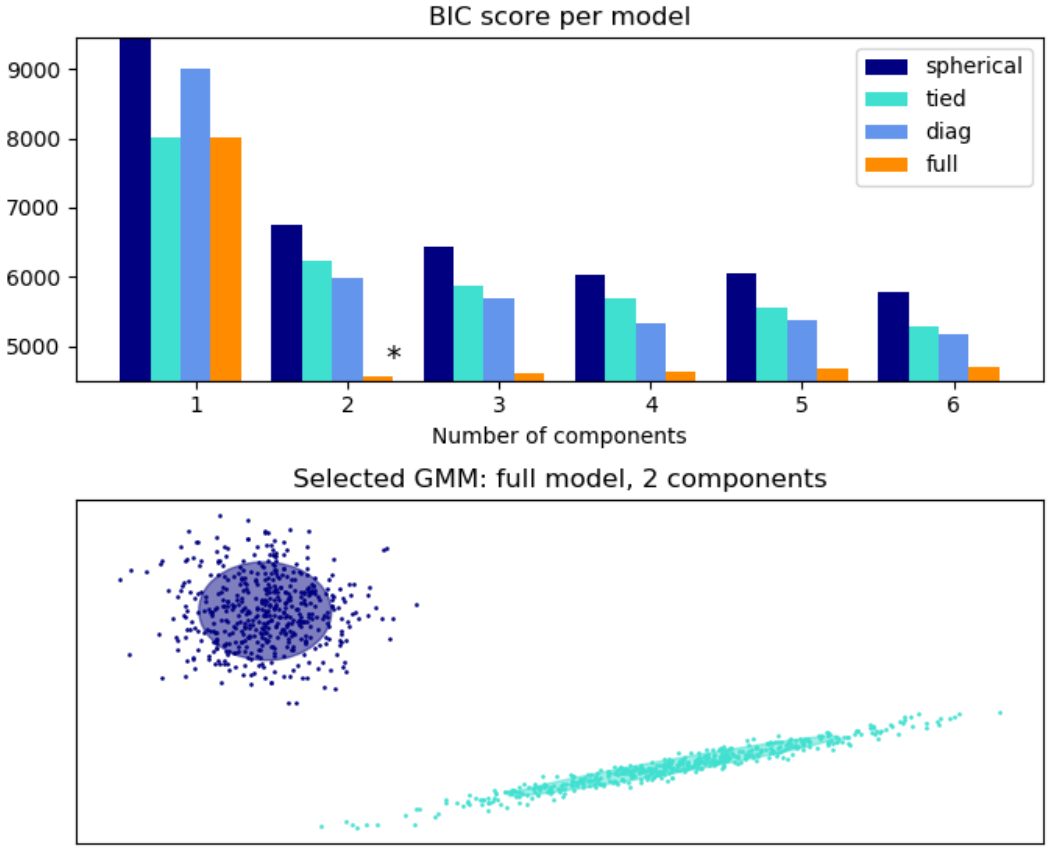
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.14.3 Gaussian Mixture Model Selection

This example shows that model selection can be performed with Gaussian Mixture Models using information-theoretic criteria (BIC). Model selection concerns both the covariance type and the number of components in the model. In that case, AIC also provides the right result (not shown to save time), but BIC is better suited if the problem is to identify the right model. Unlike Bayesian procedures, such inferences are prior-free.

In that case, the model with 2 components and full covariance (which corresponds to the true generative model) is selected.




```
import numpy as np
import itertools

from scipy import linalg
import matplotlib.pyplot as plt
import matplotlib as mpl

from sklearn import mixture

print(__doc__)

# Number of samples per component
n_samples = 500

# Generate random sample, two components
np.random.seed(0)
C = np.array([[0., -0.1], [1.7, .4]])
X = np.r_[np.dot(np.random.randn(n_samples, 2), C),
          .7 * np.random.randn(n_samples, 2) + np.array([-6, 3])]

lowest_bic = np.infty
bic = []
n_components_range = range(1, 7)
cv_types = ['spherical', 'tied', 'diag', 'full']
for cv_type in cv_types:
```

(continues on next page)

(continued from previous page)

```
    for n_components in n_components_range:
        # Fit a Gaussian mixture with EM
        gmm = mixture.GaussianMixture(n_components=n_components,
                                      covariance_type=cv_type)
        gmm.fit(X)
        bic.append(gmm.bic(X))
        if bic[-1] < lowest_bic:
            lowest_bic = bic[-1]
            best_gmm = gmm

bic = np.array(bic)
color_iter = itertools.cycle(['navy', 'turquoise', 'cornflowerblue',
                              'darkorange'])
clf = best_gmm
bars = []

# Plot the BIC scores
plt.figure(figsize=(8, 6))
spl = plt.subplot(2, 1, 1)
for i, (cv_type, color) in enumerate(zip(cv_types, color_iter)):
    xpos = np.array(n_components_range) + .2 * (i - 2)
    bars.append(plt.bar(xpos, bic[i * len(n_components_range):
                                  (i + 1) * len(n_components_range)],
                        width=.2, color=color))
plt.xticks(n_components_range)
plt.ylim([bic.min() * 1.01 - .01 * bic.max(), bic.max()])
plt.title('BIC score per model')
xpos = np.mod(bic.argmin(), len(n_components_range)) + .65 +\
    .2 * np.floor(bic.argmin() / len(n_components_range))
plt.text(xpos, bic.min() * 0.97 + .03 * bic.max(), '*', fontsize=14)
spl.set_xlabel('Number of components')
spl.legend([b[0] for b in bars], cv_types)

# Plot the winner
splot = plt.subplot(2, 1, 2)
Y_ = clf.predict(X)
for i, (mean, cov, color) in enumerate(zip(clf.means_, clf.covariances_,
                                           color_iter)):
    v, w = linalg.eigh(cov)
    if not np.any(Y_ == i):
        continue
    plt.scatter(X[Y_ == i, 0], X[Y_ == i, 1], .8, color=color)

    # Plot an ellipse to show the Gaussian component
    angle = np.arctan2(w[0][1], w[0][0])
    angle = 180. * angle / np.pi  # convert to degrees
    v = 2. * np.sqrt(2.) * np.sqrt(v)
    ell = mpl.patches.Ellipse(mean, v[0], v[1], 180. + angle, color=color)
    ell.set_clip_box(splot.bbox)
    ell.set_alpha(.5)
    splot.add_artist(ell)

plt.xticks(())
plt.yticks(())
plt.title('Selected GMM: full model, 2 components')
plt.subplots_adjust(hspace=.35, bottom=.02)
plt.show()
```

Total running time of the script: (0 minutes 0.511 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.14.4 GMM covariances

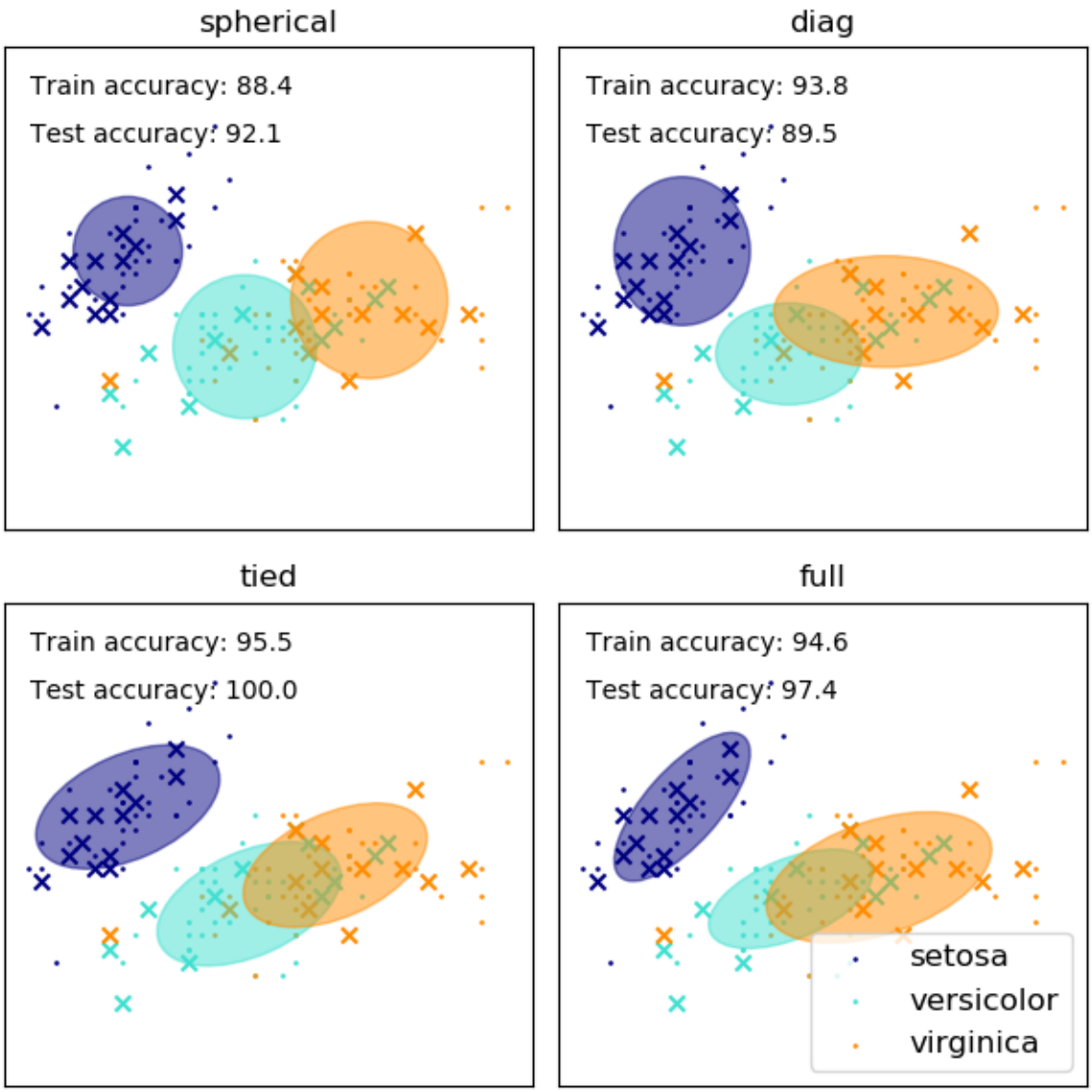
Demonstration of several covariances types for Gaussian mixture models.

See *Gaussian mixture models* for more information on the estimator.

Although GMM are often used for clustering, we can compare the obtained clusters with the actual classes from the dataset. We initialize the means of the Gaussians with the means of the classes from the training set to make this comparison valid.

We plot predicted labels on both training and held out test data using a variety of GMM covariance types on the iris dataset. We compare GMMs with spherical, diagonal, full, and tied covariance matrices in increasing order of performance. Although one would expect full covariance to perform best in general, it is prone to overfitting on small datasets and does not generalize well to held out test data.

On the plots, train data is shown as dots, while test data is shown as crosses. The iris dataset is four-dimensional. Only the first two dimensions are shown here, and thus some points are separated in other dimensions.




```
# Author: Ron Weiss <ronweiss@gmail.com>, Gael Varoquaux
# Modified by Thierry Guillemot <thierry.guillemot.work@gmail.com>
# License: BSD 3 clause

import matplotlib as mpl
import matplotlib.pyplot as plt

import numpy as np

from sklearn import datasets
from sklearn.mixture import GaussianMixture
from sklearn.model_selection import StratifiedKFold

print(__doc__)

colors = ['navy', 'turquoise', 'darkorange']
```

(continues on next page)

(continued from previous page)

```
def make_ellipses(gmm, ax):
    for n, color in enumerate(colors):
        if gmm.covariance_type == 'full':
            covariances = gmm.covariances_[n][:2, :2]
        elif gmm.covariance_type == 'tied':
            covariances = gmm.covariances_[:2, :2]
        elif gmm.covariance_type == 'diag':
            covariances = np.diag(gmm.covariances_[n][:2])
        elif gmm.covariance_type == 'spherical':
            covariances = np.eye(gmm.means_.shape[1]) * gmm.covariances_[n]
        v, w = np.linalg.eigh(covariances)
        u = w[0] / np.linalg.norm(w[0])
        angle = np.arctan2(u[1], u[0])
        angle = 180 * angle / np.pi  # convert to degrees
        v = 2. * np.sqrt(2.) * np.sqrt(v)
        ell = mpl.patches.Ellipse(gmm.means_[n, :2], v[0], v[1],
                                  180 + angle, color=color)
        ell.set_clip_box(ax.bbox)
        ell.set_alpha(0.5)
        ax.add_artist(ell)
        ax.set_aspect('equal', 'datalim')

iris = datasets.load_iris()

# Break up the dataset into non-overlapping training (75%) and testing
# (25%) sets.
skf = StratifiedKFold(n_splits=4)
# Only take the first fold.
train_index, test_index = next(iter(skf.split(iris.data, iris.target)))


X_train = iris.data[train_index]
y_train = iris.target[train_index]
X_test = iris.data[test_index]
y_test = iris.target[test_index]

n_classes = len(np.unique(y_train))

# Try GMMs using different types of covariances.
estimators = {cov_type: GaussianMixture(n_components=n_classes,
              covariance_type=cov_type, max_iter=20, random_state=0)
              for cov_type in ['spherical', 'diag', 'tied', 'full']}

n_estimators = len(estimators)

plt.figure(figsize=(3 * n_estimators // 2, 6))
plt.subplots_adjust(bottom=.01, top=0.95, hspace=.15, wspace=.05,
                    left=.01, right=.99)


for index, (name, estimator) in enumerate(estimators.items()):
    # Since we have class labels for the training data, we can
    # initialize the GMM parameters in a supervised manner.
    estimator.means_init = np.array([X_train[y_train == i].mean(axis=0)
                                     for i in range(n_classes)])
```

(continues on next page)

(continued from previous page)

```
    # Train the other parameters using the EM algorithm.
    estimator.fit(X_train)

    h = plt.subplot(2, n_estimators // 2, index + 1)
    make_ellipses(estimator, h)

    for n, color in enumerate(colors):
        data = iris.data[iris.target == n]
        plt.scatter(data[:, 0], data[:, 1], s=0.8, color=color,
                    label=iris.target_names[n])
    # Plot the test data with crosses
    for n, color in enumerate(colors):
        data = X_test[y_test == n]
        plt.scatter(data[:, 0], data[:, 1], marker='x', color=color)

    y_train_pred = estimator.predict(X_train)
    train_accuracy = np.mean(y_train_pred.ravel() == y_train.ravel()) * 100
    plt.text(0.05, 0.9, 'Train accuracy: %.1f' % train_accuracy,
             transform=h.transAxes)

    y_test_pred = estimator.predict(X_test)
    test_accuracy = np.mean(y_test_pred.ravel() == y_test.ravel()) * 100
    plt.text(0.05, 0.8, 'Test accuracy: %.1f' % test_accuracy,
             transform=h.transAxes)

    plt.xticks(())
    plt.yticks(())
    plt.title(name)

plt.legend(scatterpoints=1, loc='lower right', prop=dict(size=12))


plt.show()
```

Total running time of the script: (0 minutes 0.501 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.14.5 Gaussian Mixture Model Sine Curve

This example demonstrates the behavior of Gaussian mixture models fit on data that was not sampled from a mixture of Gaussian random variables. The dataset is formed by 100 points loosely spaced following a noisy sine curve. There is therefore no ground truth value for the number of Gaussian components.

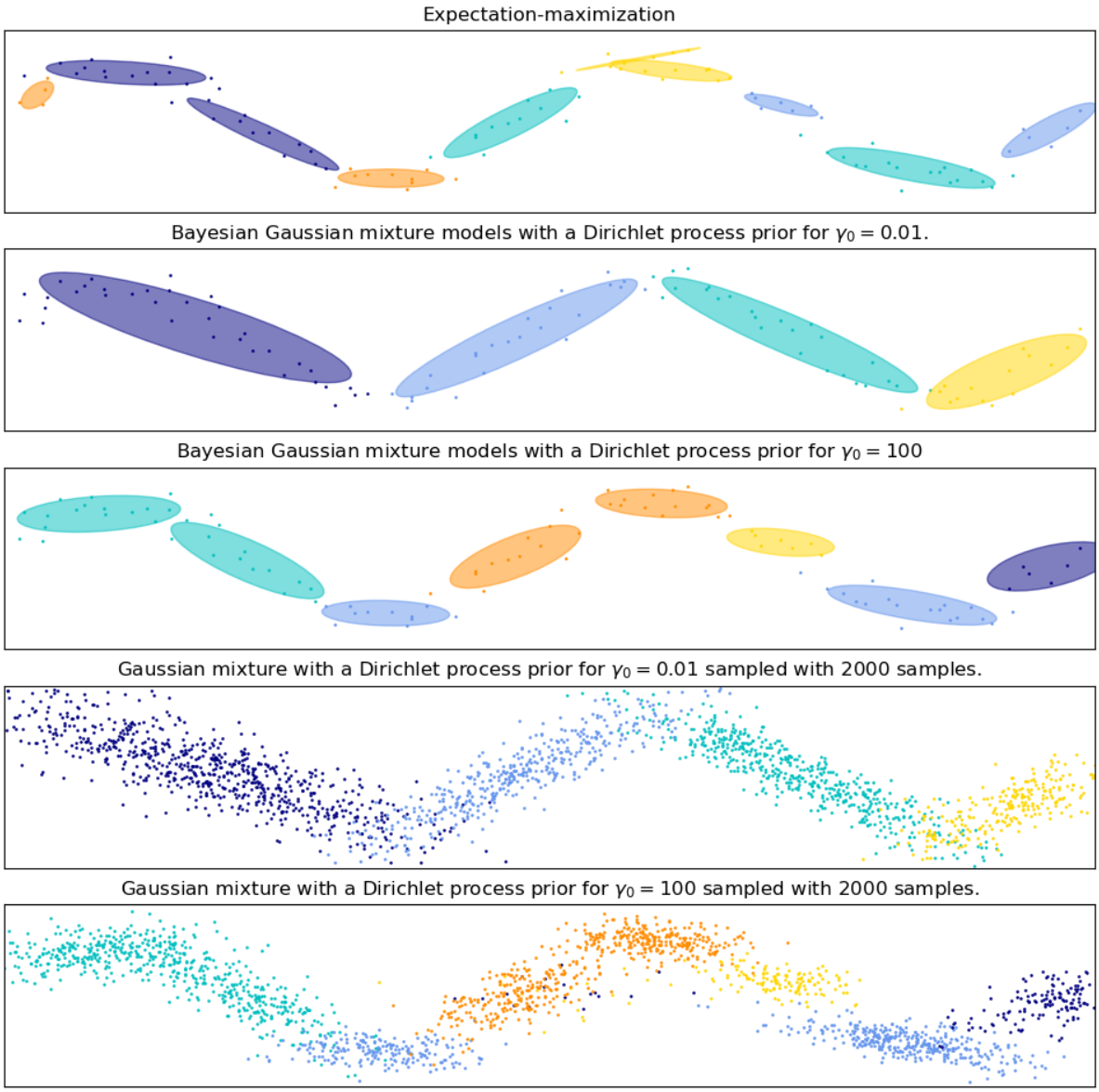
The first model is a classical Gaussian Mixture Model with 10 components fit with the Expectation-Maximization algorithm.

The second model is a Bayesian Gaussian Mixture Model with a Dirichlet process prior fit with variational inference. The low value of the concentration prior makes the model favor a lower number of active components. This models "decides" to focus its modeling power on the big picture of the structure of the dataset: groups of points with alternating directions modeled by non-diagonal covariance matrices. Those alternating directions roughly capture the alternating nature of the original sine signal.

The third model is also a Bayesian Gaussian mixture model with a Dirichlet process prior but this time the value of the concentration prior is higher giving the model more liberty to model the fine-grained structure of the data. The result is a mixture with a larger number of active components that is similar to the first model where we arbitrarily decided to fix the number of components to 10.

Which model is the best is a matter of subjective judgement: do we want to favor models that only capture the big picture to summarize and explain most of the structure of the data while ignoring the details or do we prefer models that closely follow the high density regions of the signal?

The last two panels show how we can sample from the last two models. The resulting samples distributions do not look exactly like the original data distribution. The difference primarily stems from the approximation error we made by using a model that assumes that the data was generated by a finite number of Gaussian components instead of a continuous noisy sine curve.



```
import itertools

import numpy as np
from scipy import linalg
import matplotlib.pyplot as plt
import matplotlib as mpl

from sklearn import mixture

print(__doc__)

color_iter = itertools.cycle(['navy', 'c', 'cornflowerblue', 'gold',
                              'darkorange'])


def plot_results(X, Y, means, covariances, index, title):
    splot = plt.subplot(5, 1, 1 + index)
    for i, (mean, covar, color) in enumerate(zip(
            means, covariances, color_iter)):
        v, w = linalg.eigh(covar)
        v = 2. * np.sqrt(2.) * np.sqrt(v)
        u = w[0] / linalg.norm(w[0])
        # as the DP will not use every component it has access to
        # unless it needs it, we shouldn't plot the redundant
        # components.
        if not np.any(Y == i):
            continue
        plt.scatter(X[Y == i, 0], X[Y == i, 1], .8, color=color)

        # Plot an ellipse to show the Gaussian component
        angle = np.arctan(u[1] / u[0])
        angle = 180. * angle / np.pi  # convert to degrees
        ell = mpl.patches.Ellipse(mean, v[0], v[1], 180. + angle, color=color)
        ell.set_clip_box(splot.bbox)
        ell.set_alpha(0.5)
        splot.add_artist(ell)

    plt.xlim(-6., 4. * np.pi - 6.)
    plt.ylim(-5., 5.)
    plt.title(title)
    plt.xticks(())
    plt.yticks(())


def plot_samples(X, Y, n_components, index, title):
    plt.subplot(5, 1, 4 + index)
    for i, color in zip(range(n_components), color_iter):
        # as the DP will not use every component it has access to
        # unless it needs it, we shouldn't plot the redundant
        # components.
        if not np.any(Y == i):
            continue
        plt.scatter(X[Y == i, 0], X[Y == i, 1], .8, color=color)

    plt.xlim(-6., 4. * np.pi - 6.)
    plt.ylim(-5., 5.)
    plt.title(title)
```

(continues on next page)

(continued from previous page)

```
    plt.xticks(())
    plt.yticks(())


# Parameters
n_samples = 100

# Generate random sample following a sine curve
np.random.seed(0)
X = np.zeros((n_samples, 2))
step = 4. * np.pi / n_samples

for i in range(X.shape[0]):
    x = i * step - 6.
    X[i, 0] = x + np.random.normal(0, 0.1)
    X[i, 1] = 3. * (np.sin(x) + np.random.normal(0, .2))

plt.figure(figsize=(10, 10))
plt.subplots_adjust(bottom=.04, top=0.95, hspace=.2, wspace=.05,
                    left=.03, right=.97)

# Fit a Gaussian mixture with EM using ten components
gmm = mixture.GaussianMixture(n_components=10, covariance_type='full',
                              max_iter=100).fit(X)
plot_results(X, gmm.predict(X), gmm.means_, gmm.covariances_, 0,
             'Expectation-maximization')

dpgmm = mixture.BayesianGaussianMixture(
    n_components=10, covariance_type='full', weight_concentration_prior=1e-2,
    weight_concentration_prior_type='dirichlet_process',
    mean_precision_prior=1e-2, covariance_prior=1e0 * np.eye(2),
    init_params="random", max_iter=100, random_state=2).fit(X)
plot_results(X, dpgmm.predict(X), dpgmm.means_, dpgmm.covariances_, 1,
             "Bayesian Gaussian mixture models with a Dirichlet process prior "
             r"for $\gamma_0=0.01$.")

X_s, y_s = dpgmm.sample(n_samples=2000)
plot_samples(X_s, y_s, dpgmm.n_components, 0,
             "Gaussian mixture with a Dirichlet process prior "
             r"for $\gamma_0=0.01$ sampled with $2000$ samples.")

dpgmm = mixture.BayesianGaussianMixture(
    n_components=10, covariance_type='full', weight_concentration_prior=1e+2,
    weight_concentration_prior_type='dirichlet_process',
    mean_precision_prior=1e-2, covariance_prior=1e0 * np.eye(2),
    init_params="kmeans", max_iter=100, random_state=2).fit(X)
plot_results(X, dpgmm.predict(X), dpgmm.means_, dpgmm.covariances_, 2,
             "Bayesian Gaussian mixture models with a Dirichlet process prior "
             r"for $\gamma_0=100$")

X_s, y_s = dpgmm.sample(n_samples=2000)
plot_samples(X_s, y_s, dpgmm.n_components, 1,
             "Gaussian mixture with a Dirichlet process prior "
             r"for $\gamma_0=100$ sampled with $2000$ samples.")

plt.show()
```

Total running time of the script: (0 minutes 0.614 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

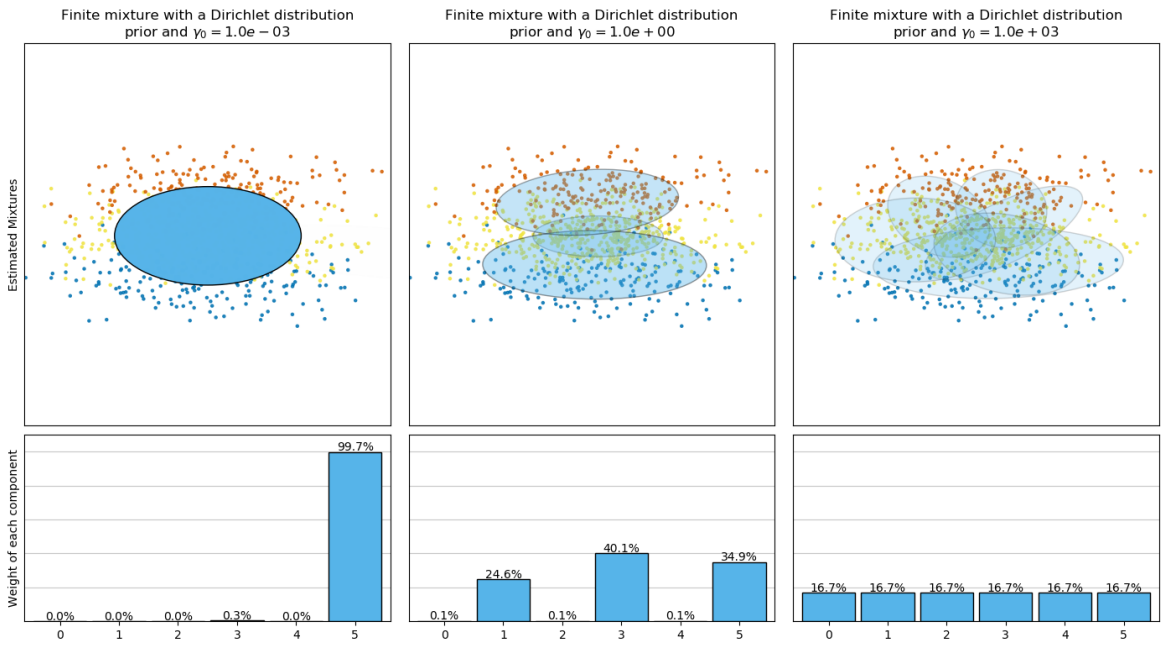
6.14.6 Concentration Prior Type Analysis of Variation Bayesian Gaussian Mixture

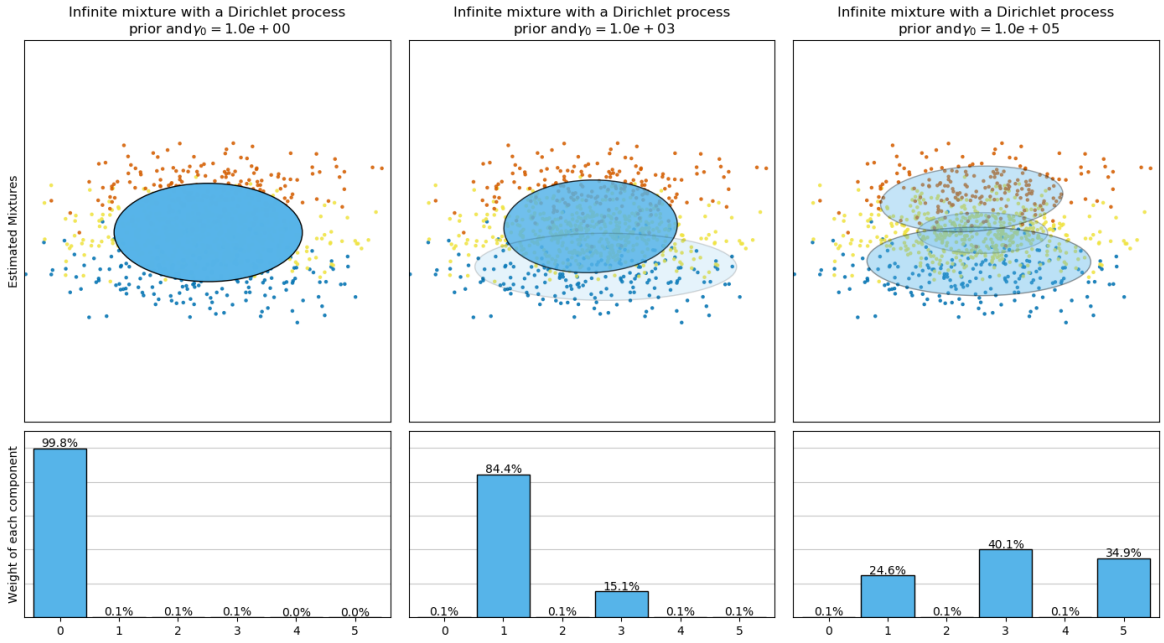
This example plots the ellipsoids obtained from a toy dataset (mixture of three Gaussians) fitted by the `BayesianGaussianMixture` class models with a Dirichlet distribution prior (`weight_concentration_prior_type='dirichlet_distribution'`) and a Dirichlet process prior (`weight_concentration_prior_type='dirichlet_process'`). On each figure, we plot the results for three different values of the weight concentration prior.

The `BayesianGaussianMixture` class can adapt its number of mixture components automatically. The parameter `weight_concentration_prior` has a direct link with the resulting number of components with non-zero weights. Specifying a low value for the concentration prior will make the model put most of the weight on few components set the remaining components weights very close to zero. High values of the concentration prior will allow a larger number of components to be active in the mixture.

The Dirichlet process prior allows to define an infinite number of components and automatically selects the correct number of components: it activates a component only if it is necessary.

On the contrary the classical finite mixture model with a Dirichlet distribution prior will favor more uniformly weighted components and therefore tends to divide natural clusters into unnecessary sub-components.






```
# Author: Thierry Guillemot <thierry.guillemot.work@gmail.com>
# License: BSD 3 clause

import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from sklearn.mixture import BayesianGaussianMixture

print(__doc__)


def plot_ellipses(ax, weights, means, covars):
    for n in range(means.shape[0]):
        eig_vals, eig_vecs = np.linalg.eigh(covars[n])
        unit_eig_vec = eig_vecs[0] / np.linalg.norm(eig_vecs[0])
        angle = np.arctan2(unit_eig_vec[1], unit_eig_vec[0])
        # Ellipse needs degrees
        angle = 180 * angle / np.pi
        # eigenvector normalization
        eig_vals = 2 * np.sqrt(2) * np.sqrt(eig_vals)
        ell = mpl.patches.Ellipse(means[n], eig_vals[0], eig_vals[1],
                                  180 + angle, edgecolor='black')
        ell.set_clip_box(ax.bbox)
        ell.set_alpha(weights[n])
        ell.set_facecolor('#56B4E9')
        ax.add_artist(ell)


def plot_results(ax1, ax2, estimator, X, y, title, plot_title=False):
    ax1.set_title(title)
    ax1.scatter(X[:, 0], X[:, 1], s=5, marker='o', color=colors[y], alpha=0.8)
    ax1.set_xlim(-2., 2.)
```

(continues on next page)

(continued from previous page)

```
    ax1.set_ylim(-3., 3.)
    ax1.set_xticks(())
    ax1.set_yticks(())
    plot_ellipses(ax1, estimator.weights_, estimator.means_,
                  estimator.covariances_)

    ax2.get_xaxis().set_tick_params(direction='out')
    ax2.yaxis.grid(True, alpha=0.7)
    for k, w in enumerate(estimator.weights_):
        ax2.bar(k, w, width=0.9, color='#56B4E9', zorder=3,
                align='center', edgecolor='black')
        ax2.text(k, w + 0.007, "%.1f%%" % (w * 100.),
                 horizontalalignment='center')
    ax2.set_xlim(-.6, 2 * n_components - .4)
    ax2.set_ylim(0., 1.1)
    ax2.tick_params(axis='y', which='both', left=False,
                    right=False, labelleft=False)
    ax2.tick_params(axis='x', which='both', top=False)

    if plot_title:
        ax1.set_ylabel('Estimated Mixtures')
        ax2.set_ylabel('Weight of each component')

# Parameters of the dataset
random_state, n_components, n_features = 2, 3, 2
colors = np.array(['#0072B2', '#F0E442', '#D55E00'])

covars = np.array([[[.7, .0], [.0, .1]],
                   [[.5, .0], [.0, .1]],
                   [[.5, .0], [.0, .1]]])
samples = np.array([200, 500, 200])
means = np.array([[.0, -.70],
                  [.0, .0],
                  [.0, .70]])

# mean_precision_prior= 0.8 to minimize the influence of the prior
estimators = [
    ("Finite mixture with a Dirichlet distribution\nprior and "
     r"$\gamma_0=$", BayesianGaussianMixture(
        weight_concentration_prior_type="dirichlet_distribution",
        n_components=2 * n_components, reg_covar=0, init_params='random',
        max_iter=1500, mean_precision_prior=.8,
        random_state=random_state), [0.001, 1, 1000]),
    ("Infinite mixture with a Dirichlet process\n prior and" r"$\gamma_0=$",
     BayesianGaussianMixture(
        weight_concentration_prior_type="dirichlet_process",
        n_components=2 * n_components, reg_covar=0, init_params='random',
        max_iter=1500, mean_precision_prior=.8,
        random_state=random_state), [1, 1000, 100000])]

# Generate data
rng = np.random.RandomState(random_state)
X = np.vstack([
    rng.multivariate_normal(means[j], covars[j], samples[j])
    for j in range(n_components)])
y = np.concatenate([np.full(samples[j], j, dtype=int)
                    for j in range(n_components)])
```

(continues on next page)

(continued from previous page)

```
# Plot results in two different figures
for (title, estimator, concentrations_prior) in estimators:
    plt.figure(figsize=(4.7 * 3, 8))
    plt.subplots_adjust(bottom=.04, top=0.90, hspace=.05, wspace=.05,
                        left=.03, right=.99)

    gs = gridspec.GridSpec(3, len(concentrations_prior))
    for k, concentration in enumerate(concentrations_prior):
        estimator.weight_concentration_prior = concentration
        estimator.fit(X)
        plot_results(plt.subplot(gs[0:2, k]), plt.subplot(gs[2, k]), estimator,
                     X, y, r"%s$%.1e$" % (title, concentration),
                     plot_title=k == 0)

plt.show()
```

Total running time of the script: (0 minutes 7.900 seconds)

Estimated memory usage: 8 MB

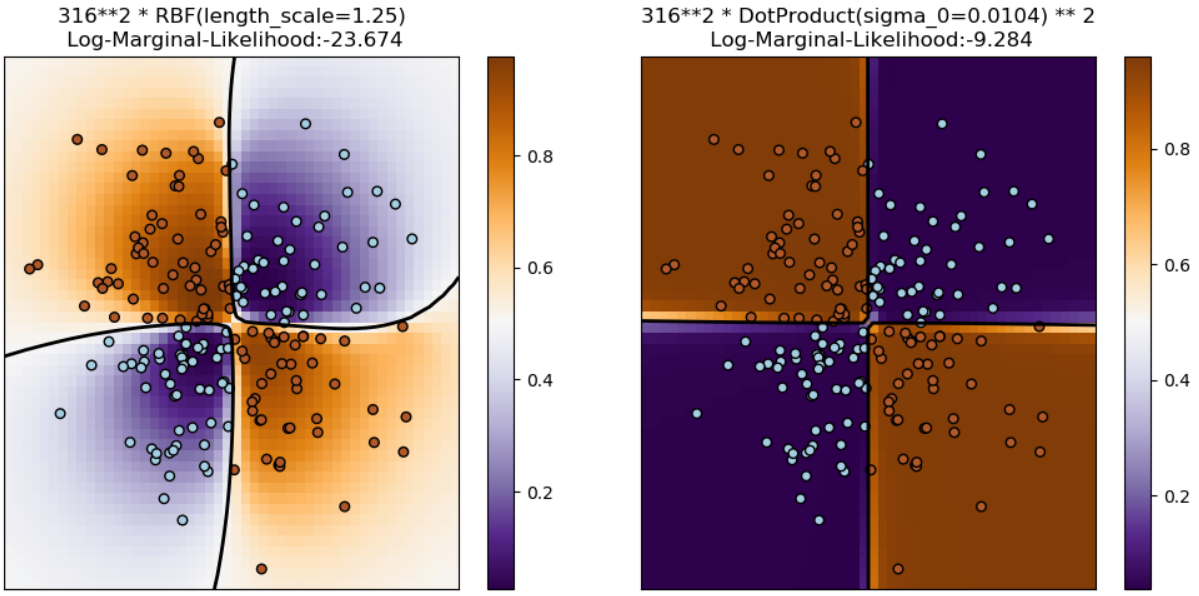
6.15 Gaussian Process for Machine Learning

Examples concerning the *sklearn.gaussian_process* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.15.1 Illustration of Gaussian process classification (GPC) on the XOR dataset

This example illustrates GPC on XOR data. Compared are a stationary, isotropic kernel (RBF) and a non-stationary kernel (DotProduct). On this particular dataset, the DotProduct kernel obtains considerably better results because the class-boundaries are linear and coincide with the coordinate axes. In general, stationary kernels often obtain better results.




```
print(__doc__)

# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF, DotProduct


xx, yy = np.meshgrid(np.linspace(-3, 3, 50),
                     np.linspace(-3, 3, 50))
rng = np.random.RandomState(0)
X = rng.randn(200, 2)
Y = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)

# fit the model
plt.figure(figsize=(10, 5))
kernels = [1.0 * RBF(length_scale=1.0), 1.0 * DotProduct(sigma_0=1.0)**2]
for i, kernel in enumerate(kernels):
    clf = GaussianProcessClassifier(kernel=kernel, warm_start=True).fit(X, Y)

    # plot the decision function for each datapoint on the grid
    Z = clf.predict_proba(np.vstack((xx.ravel(), yy.ravel())).T)[:, 1]
    Z = Z.reshape(xx.shape)

    plt.subplot(1, 2, i + 1)
    image = plt.imshow(Z, interpolation='nearest',
                       extent=(xx.min(), xx.max(), yy.min(), yy.max()),
                       aspect='auto', origin='lower', cmap=plt.cm.PuOr_r)
    contours = plt.contour(xx, yy, Z, levels=[0.5], linewidths=2,
                           colors=['k'])
    plt.scatter(X[:, 0], X[:, 1], s=30, c=Y, cmap=plt.cm.Paired,
```

(continues on next page)

(continued from previous page)

```
                edgecolors=(0, 0, 0))
    plt.xticks(())
    plt.yticks(())
    plt.axis([-3, 3, -3, 3])
    plt.colorbar(image)
    plt.title("%s\n Log-Marginal-Likelihood:%.3f"
              % (clf.kernel_, clf.log_marginal_likelihood(clf.kernel_.theta)),
              fontsize=12)

plt.tight_layout()
plt.show()
```

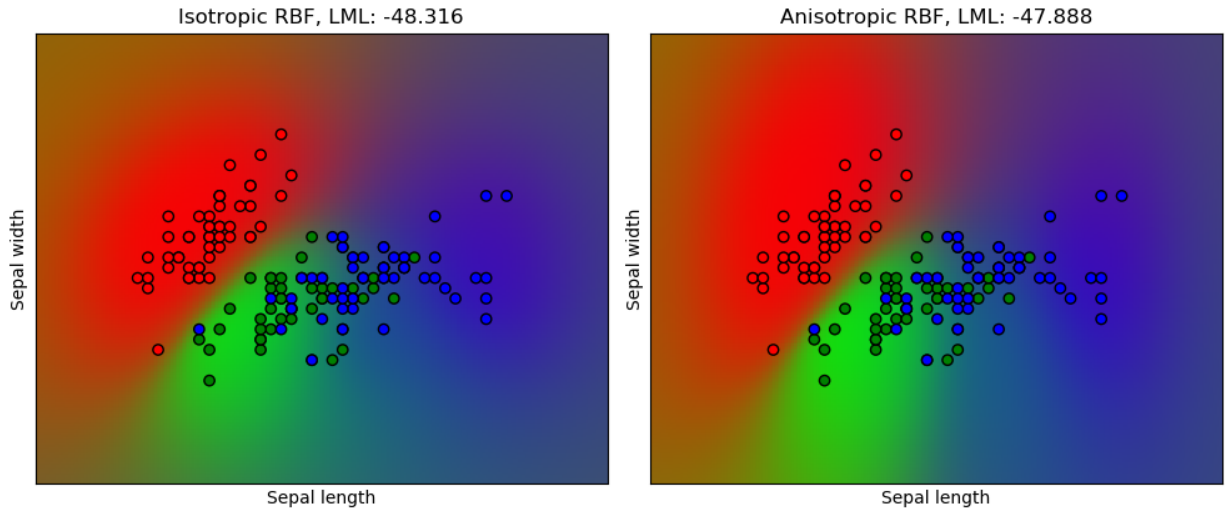
Total running time of the script: (0 minutes 0.649 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.15.2 Gaussian process classification (GPC) on iris dataset

This example illustrates the predicted probability of GPC for an isotropic and anisotropic RBF kernel on a two-dimensional version for the iris-dataset. The anisotropic RBF kernel obtains slightly higher log-marginal-likelihood by assigning different length-scales to the two feature dimensions.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
```

(continues on next page)

(continued from previous page)

```
# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2]  # we only take the first two features.
y = np.array(iris.target, dtype=int)

h = .02  # step size in the mesh

kernel = 1.0 * RBF([1.0])
gpc_rbf_isotropic = GaussianProcessClassifier(kernel=kernel).fit(X, y)
kernel = 1.0 * RBF([1.0, 1.0])
gpc_rbf_anisotropic = GaussianProcessClassifier(kernel=kernel).fit(X, y)

# create a mesh to plot in
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

titles = ["Isotropic RBF", "Anisotropic RBF"]
plt.figure(figsize=(10, 5))
for i, clf in enumerate((gpc_rbf_isotropic, gpc_rbf_anisotropic)):
    # Plot the predicted probabilities. For that, we will assign a color to
    # each point in the mesh [x_min, m_max]x[y_min, y_max].
    plt.subplot(1, 2, i + 1)

    Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape((xx.shape[0], xx.shape[1], 3))
    plt.imshow(Z, extent=(x_min, x_max, y_min, y_max), origin="lower")

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=np.array(["r", "g", "b"])[y],
                edgecolors=(0, 0, 0))
    plt.xlabel('Sepal length')
    plt.ylabel('Sepal width')
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.xticks(())
    plt.yticks(())
    plt.title("%s, LML: %.3f" %
              (titles[i], clf.log_marginal_likelihood(clf.kernel_.theta)))

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 3.233 seconds)

Estimated memory usage: 286 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

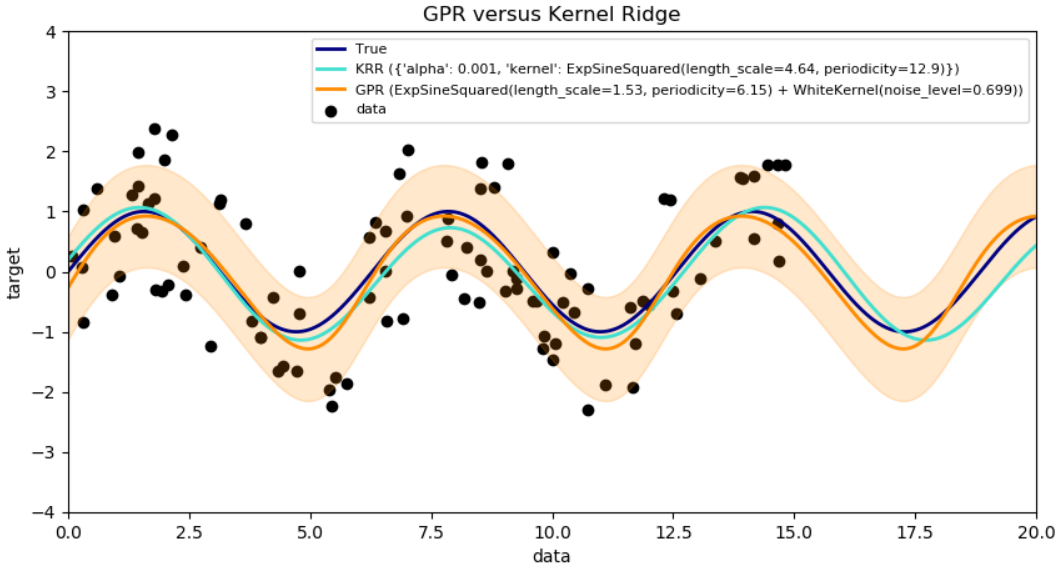
6.15.3 Comparison of kernel ridge and Gaussian process regression

Both kernel ridge regression (KRR) and Gaussian process regression (GPR) learn a target function by employing internally the "kernel trick". KRR learns a linear function in the space induced by the respective kernel which corresponds to a non-linear function in the original space. The linear function in the kernel space is chosen based on the mean-squared error loss with ridge regularization. GPR uses the kernel to define the covariance of a prior distribution over the target functions and uses the observed training data to define a likelihood function. Based on Bayes theorem, a (Gaussian) posterior distribution over target functions is defined, whose mean is used for prediction.

A major difference is that GPR can choose the kernel's hyperparameters based on gradient-ascent on the marginal likelihood function while KRR needs to perform a grid search on a cross-validated loss function (mean-squared error loss). A further difference is that GPR learns a generative, probabilistic model of the target function and can thus provide meaningful confidence intervals and posterior samples along with the predictions while KRR only provides predictions.

This example illustrates both methods on an artificial dataset, which consists of a sinusoidal target function and strong noise. The figure compares the learned model of KRR and GPR based on a ExpSineSquared kernel, which is suited for learning periodic functions. The kernel's hyperparameters control the smoothness (l) and periodicity of the kernel (p). Moreover, the noise level of the data is learned explicitly by GPR by an additional WhiteKernel component in the kernel and by the regularization parameter alpha of KRR.

The figure shows that both methods learn reasonable models of the target function. GPR correctly identifies the periodicity of the function to be roughly 2*pi (6.28), while KRR chooses the doubled periodicity 4*pi. Besides that, GPR provides reasonable confidence bounds on the prediction which are not available for KRR. A major difference between the two methods is the time required for fitting and predicting: while fitting KRR is fast in principle, the grid-search for hyperparameter optimization scales exponentially with the number of hyperparameters ("curse of dimensionality"). The gradient-based optimization of the parameters in GPR does not suffer from this exponential scaling and is thus considerable faster on this example with 3-dimensional hyperparameter space. The time for predicting is similar; however, generating the variance of the predictive distribution of GPR takes considerable longer than just predicting the mean.



Out:

```
Time for KRR fitting: 3.275
Time for GPR fitting: 0.074
```

(continues on next page)

(continued from previous page)

```
Time for KRR prediction: 0.054
Time for GPR prediction: 0.058
Time for GPR prediction with standard-deviation: 0.057
```

```
print(__doc__)

# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD 3 clause


import time

import numpy as np

import matplotlib.pyplot as plt

from sklearn.kernel_ridge import KernelRidge
from sklearn.model_selection import GridSearchCV
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import WhiteKernel, ExpSineSquared

rng = np.random.RandomState(0)

# Generate sample data
X = 15 * rng.rand(100, 1)
y = np.sin(X).ravel()
y += 3 * (0.5 - rng.rand(X.shape[0]))  # add noise

# Fit KernelRidge with parameter selection based on 5-fold cross validation
param_grid = {"alpha": [1e0, 1e-1, 1e-2, 1e-3],
              "kernel": [ExpSineSquared(l, p)
                         for l in np.logspace(-2, 2, 10)
                         for p in np.logspace(0, 2, 10)]}
kr = GridSearchCV(KernelRidge(), param_grid=param_grid)
stime = time.time()
kr.fit(X, y)
print("Time for KRR fitting: %.3f" % (time.time() - stime))

gp_kernel = ExpSineSquared(1.0, 5.0, periodicity_bounds=(1e-2, 1e1)) \
    + WhiteKernel(1e-1)
gpr = GaussianProcessRegressor(kernel=gp_kernel)
stime = time.time()
gpr.fit(X, y)
print("Time for GPR fitting: %.3f" % (time.time() - stime))

# Predict using kernel ridge
X_plot = np.linspace(0, 20, 10000)[:, None]
stime = time.time()
y_kr = kr.predict(X_plot)
print("Time for KRR prediction: %.3f" % (time.time() - stime))
```

(continues on next page)

(continued from previous page)

```
# Predict using gaussian process regressor
stime = time.time()
y_gpr = gpr.predict(X_plot, return_std=False)
print("Time for GPR prediction: %.3f" % (time.time() - stime))

stime = time.time()
y_gpr, y_std = gpr.predict(X_plot, return_std=True)
print("Time for GPR prediction with standard-deviation: %.3f"
      % (time.time() - stime))

# Plot results
plt.figure(figsize=(10, 5))
lw = 2
plt.scatter(X, y, c='k', label='data')
plt.plot(X_plot, np.sin(X_plot), color='navy', lw=lw, label='True')
plt.plot(X_plot, y_kr, color='turquoise', lw=lw,
         label='KRR (%s)' % kr.best_params_)
plt.plot(X_plot, y_gpr, color='darkorange', lw=lw,
         label='GPR (%s)' % gpr.kernel_)
plt.fill_between(X_plot[:, 0], y_gpr - y_std, y_gpr + y_std, color='darkorange',
                 alpha=0.2)
plt.xlabel('data')
plt.ylabel('target')
plt.xlim(0, 20)
plt.ylim(-4, 4)
plt.title('GPR versus Kernel Ridge')
plt.legend(loc="best",  scatterpoints=1, prop={'size': 8})
plt.show()
```

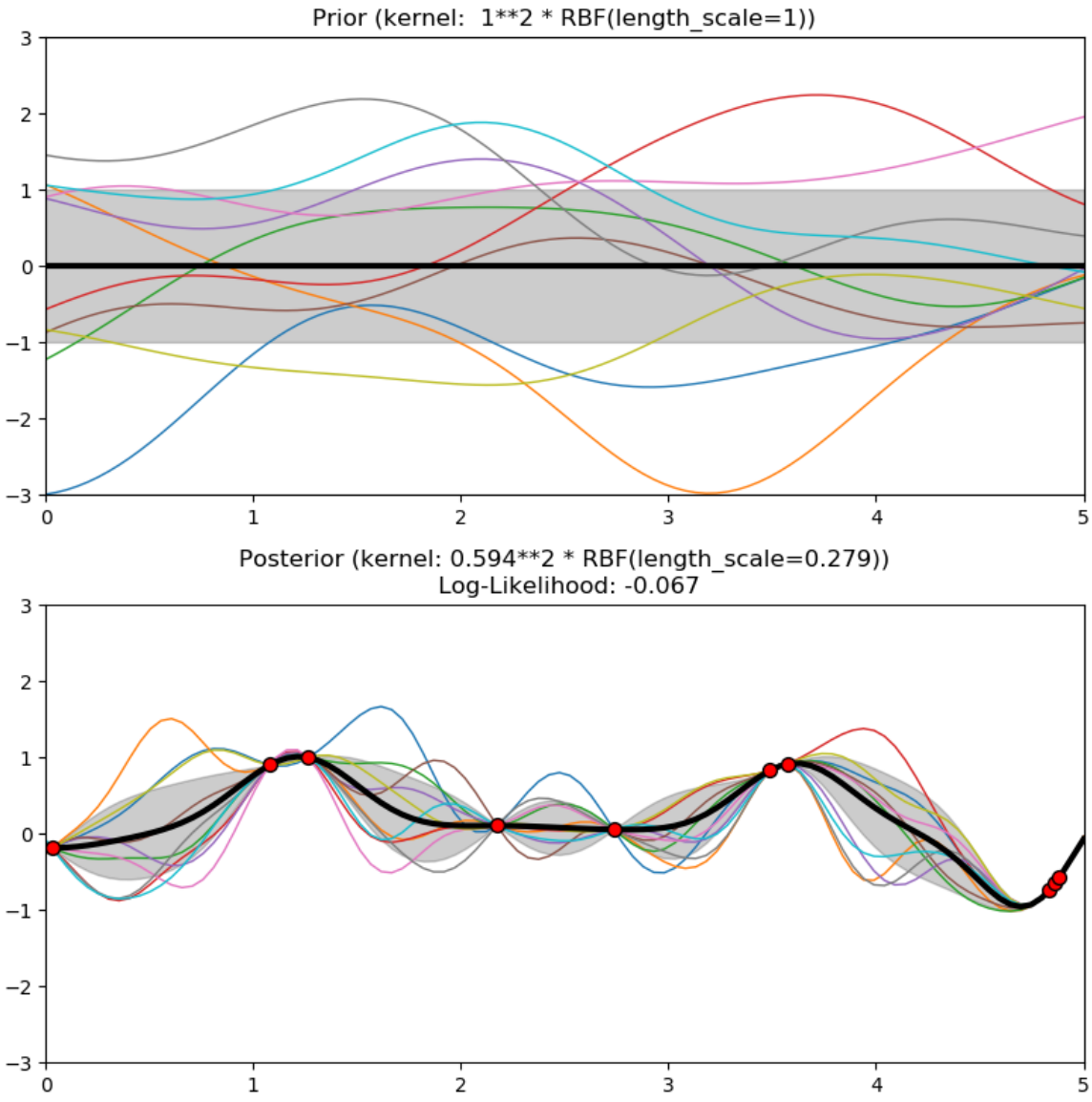
Total running time of the script: (0 minutes 3.858 seconds)

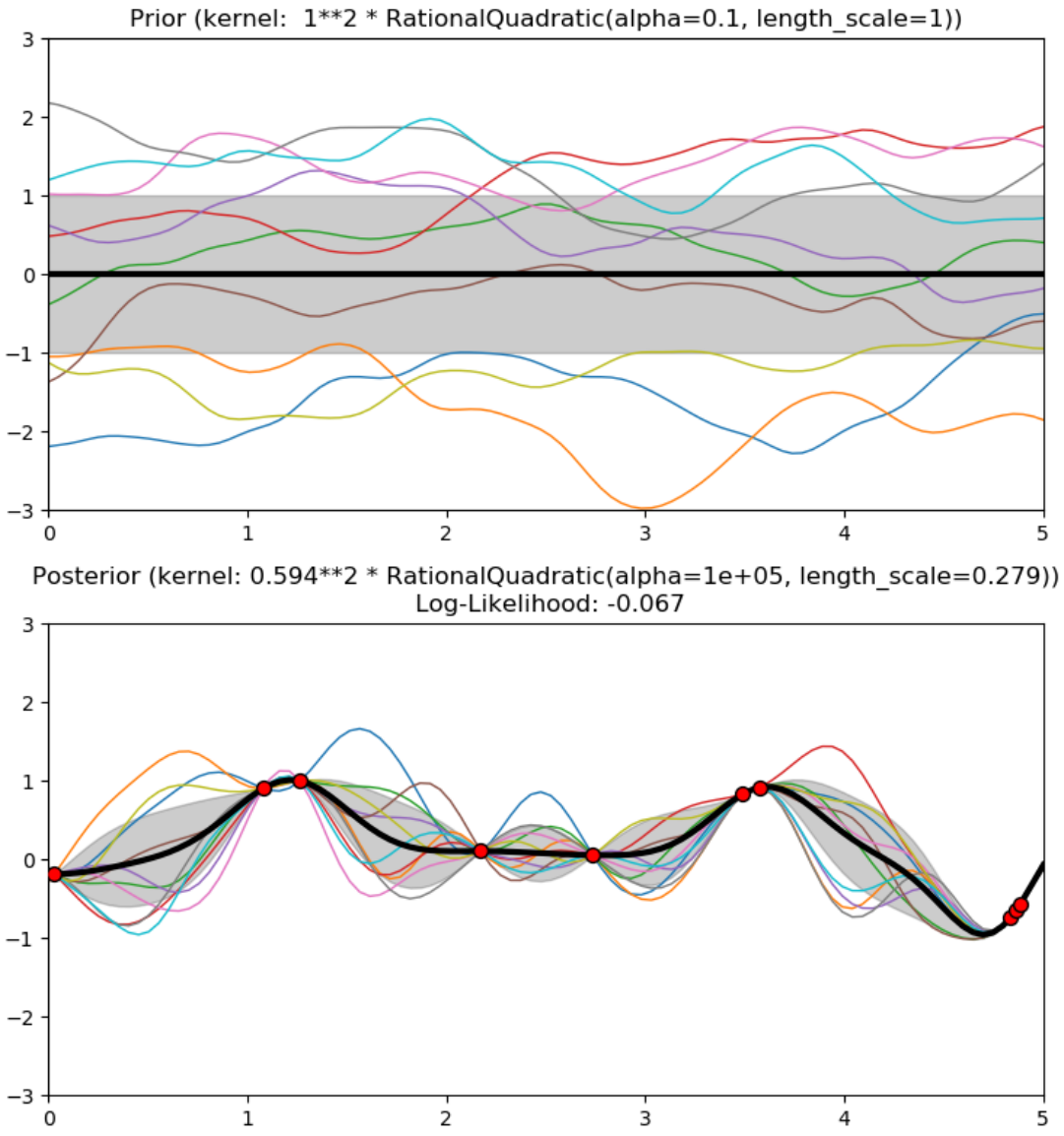
Estimated memory usage: 8 MB

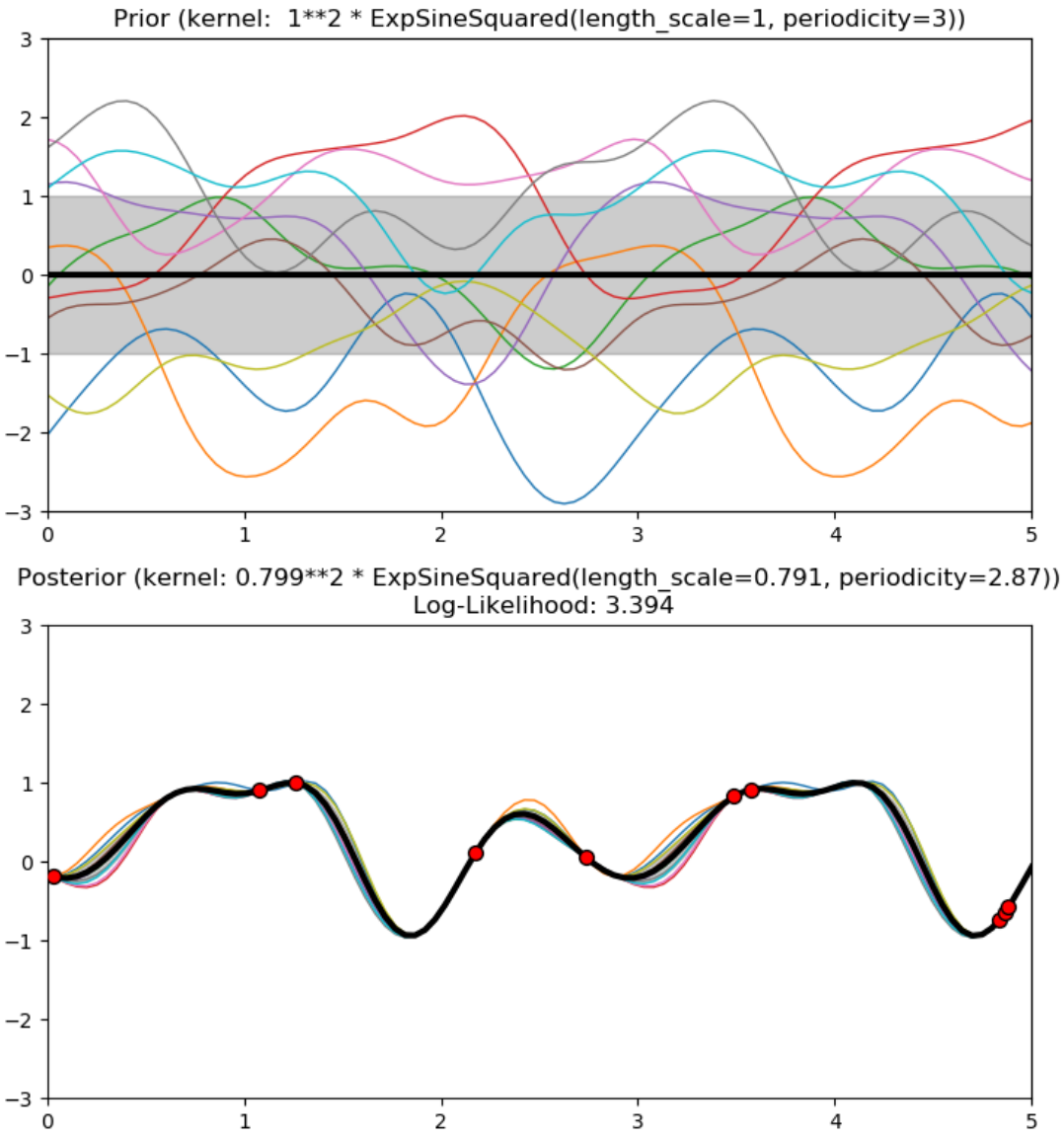
Note: Click *here* to download the full example code or to run this example in your browser via Binder

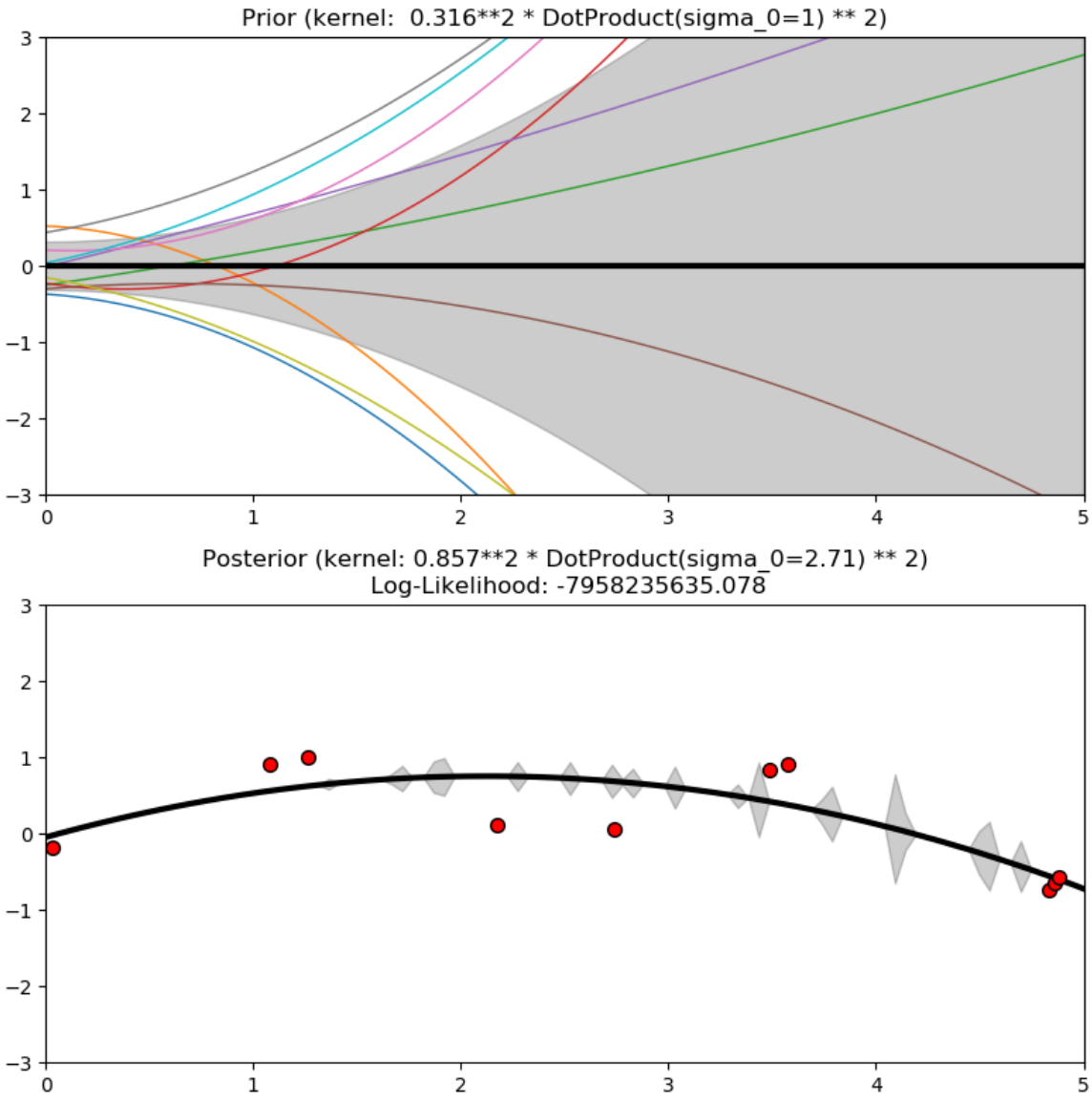
6.15.4 Illustration of prior and posterior Gaussian process for different kernels

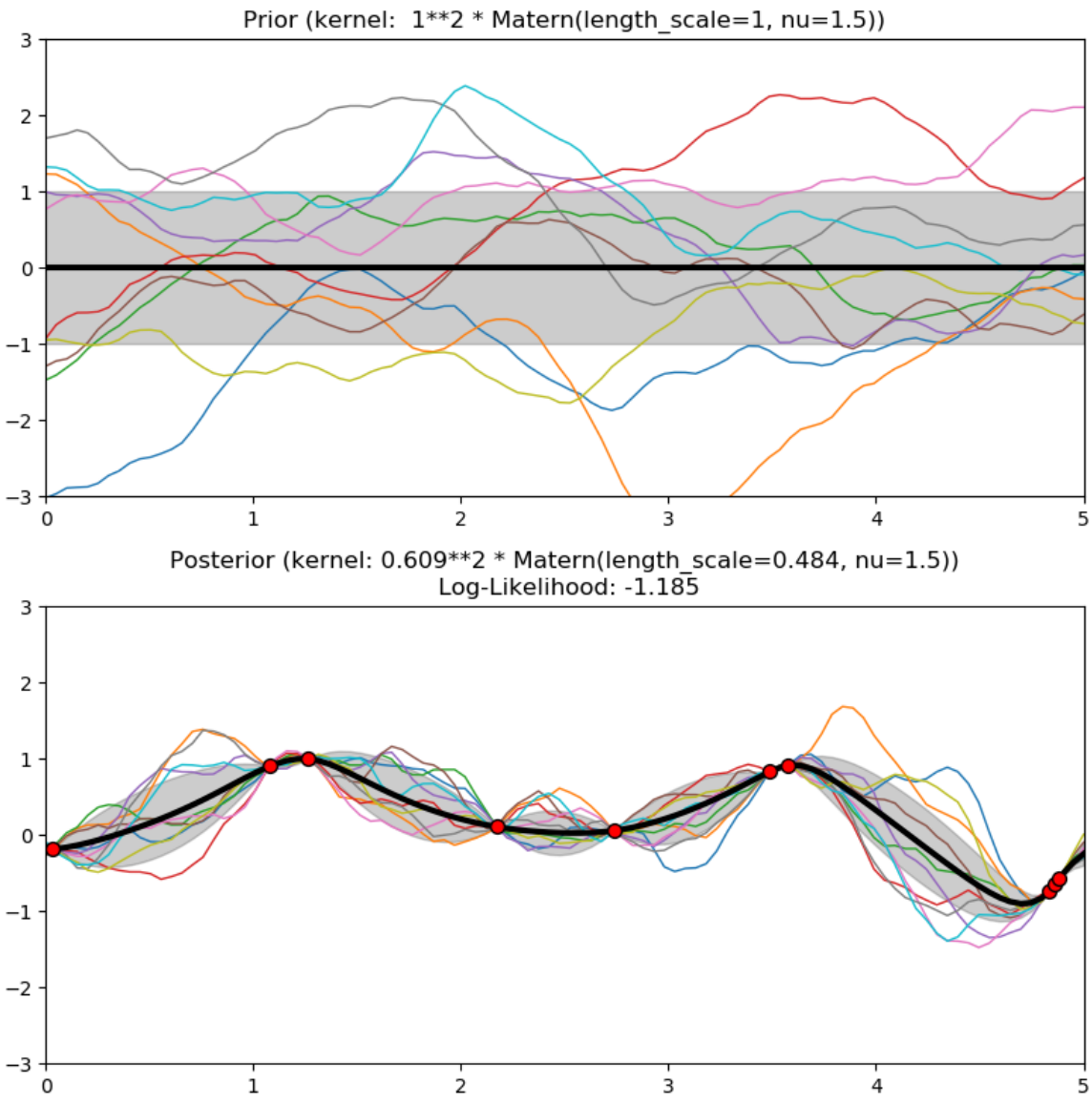
This example illustrates the prior and posterior of a GPR with different kernels. Mean, standard deviation, and 10 samples are shown for both prior and posterior.


Prior (kernel: 1**2 * RBF(length_scale=1))
Posterior (kernel: 0.594**2 * RBF(length_scale=0.279))
Log-Likelihood: -0.067


Prior (kernel: 1**2 * RationalQuadratic(alpha=0.1, length_scale=1))
Posterior (kernel: 0.594**2 * RationalQuadratic(alpha=1e+05, length_scale=0.279))
Log-Likelihood: -0.067


Prior (kernel: 1**2 * ExpSineSquared(length_scale=1, periodicity=3))
Posterior (kernel: 0.799**2 * ExpSineSquared(length_scale=0.791, periodicity=2.87))
Log-Likelihood: 3.394


Prior (kernel: 0.316**2 * DotProduct(sigma_0=1) ** 2)
Posterior (kernel: 0.857**2 * DotProduct(sigma_0=2.71) ** 2)
Log-Likelihood: -7958235635.078



Out:

```
/home/circleci/project/sklearn/gaussian_process/_gpr.py:494: ConvergenceWarning:␣
↪lbfgs failed to converge (status=2):
ABNORMAL_TERMINATION_IN_LNSRCH.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
  _check_optimize_result("lbfgs", opt_res)
/home/circleci/project/sklearn/gaussian_process/_gpr.py:362: UserWarning: Predicted␣
↪variances smaller than 0. Setting those variances to 0.
  warnings.warn("Predicted variances smaller than 0. "
```

```
print(__doc__)

# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
#
# License: BSD 3 clause

import numpy as np

from matplotlib import pyplot as plt

from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import (RBF, Matern, RationalQuadratic,
                                              ExpSineSquared, DotProduct,
                                              ConstantKernel)


kernels = [1.0 * RBF(length_scale=1.0, length_scale_bounds=(1e-1, 10.0)),
           1.0 * RationalQuadratic(length_scale=1.0, alpha=0.1),
           1.0 * ExpSineSquared(length_scale=1.0, periodicity=3.0,
                                length_scale_bounds=(0.1, 10.0),
                                periodicity_bounds=(1.0, 10.0)),
           ConstantKernel(0.1, (0.01, 10.0))
               * (DotProduct(sigma_0=1.0, sigma_0_bounds=(0.1, 10.0)) ** 2),
           1.0 * Matern(length_scale=1.0, length_scale_bounds=(1e-1, 10.0),
                        nu=1.5)]

for kernel in kernels:
    # Specify Gaussian Process
    gp = GaussianProcessRegressor(kernel=kernel)

    # Plot prior
    plt.figure(figsize=(8, 8))
    plt.subplot(2, 1, 1)
    X_ = np.linspace(0, 5, 100)
    y_mean, y_std = gp.predict(X_[:, np.newaxis], return_std=True)
    plt.plot(X_, y_mean, 'k', lw=3, zorder=9)
    plt.fill_between(X_, y_mean - y_std, y_mean + y_std,
                     alpha=0.2, color='k')
    y_samples = gp.sample_y(X_[:, np.newaxis], 10)
    plt.plot(X_, y_samples, lw=1)
    plt.xlim(0, 5)
    plt.ylim(-3, 3)
    plt.title("Prior (kernel:  %s)" % kernel, fontsize=12)

    # Generate data and fit GP
    rng = np.random.RandomState(4)
    X = rng.uniform(0, 5, 10)[:, np.newaxis]
    y = np.sin((X[:, 0] - 2.5) ** 2)
    gp.fit(X, y)

    # Plot posterior
    plt.subplot(2, 1, 2)
    X_ = np.linspace(0, 5, 100)
    y_mean, y_std = gp.predict(X_[:, np.newaxis], return_std=True)
    plt.plot(X_, y_mean, 'k', lw=3, zorder=9)
    plt.fill_between(X_, y_mean - y_std, y_mean + y_std,
                     alpha=0.2, color='k')
```

(continues on next page)

(continued from previous page)

```
    y_samples = gp.sample_y(X_[:, np.newaxis], 10)
    plt.plot(X_, y_samples, lw=1)
    plt.scatter(X[:, 0], y, c='r', s=50, zorder=10, edgecolors=(0, 0, 0))
    plt.xlim(0, 5)
    plt.ylim(-3, 3)
    plt.title("Posterior (kernel: %s)\n Log-Likelihood: %.3f"
              % (gp.kernel_, gp.log_marginal_likelihood(gp.kernel_.theta)),
              fontsize=12)
    plt.tight_layout()

plt.show()
```

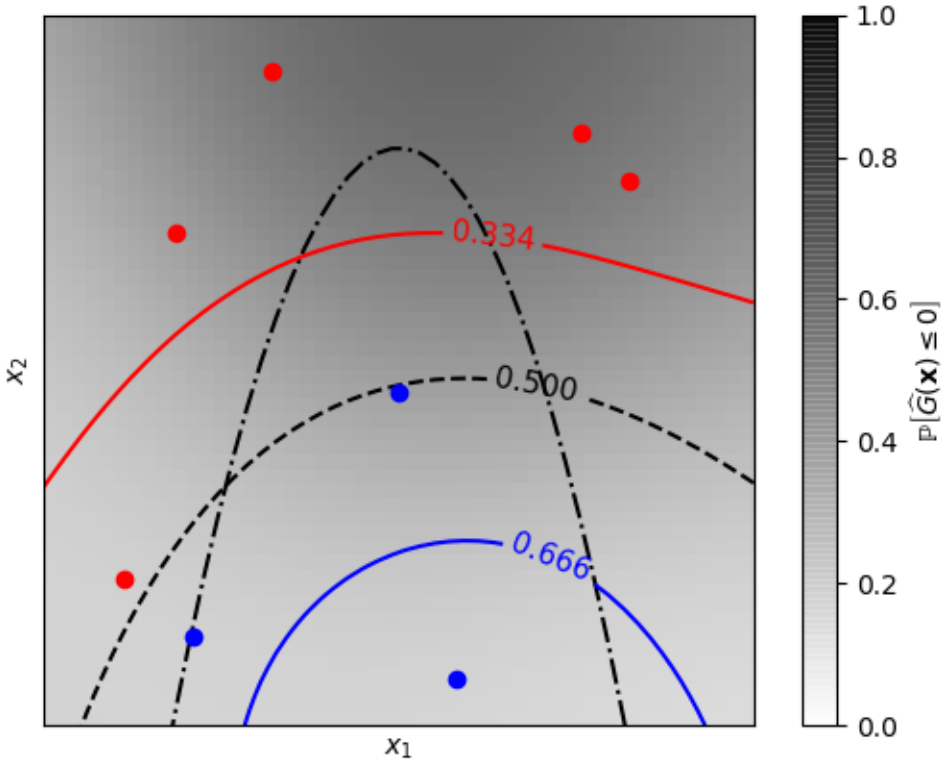
Total running time of the script: (0 minutes 1.239 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.15.5 Iso-probability lines for Gaussian Processes classification (GPC)

A two-dimensional classification example showing iso-probability lines for the predicted probabilities.



Out:

```
Learned kernel: 0.0256**2 * DotProduct(sigma_0=5.72) ** 2
```

```
print(__doc__)

# Author: Vincent Dubourg <vincent.dubourg@gmail.com>
# Adapted to GaussianProcessClassifier:
#         Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD 3 clause

import numpy as np

from matplotlib import pyplot as plt
from matplotlib import cm

from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import DotProduct, ConstantKernel as C

# A few constants
lim = 8


def g(x):
    """The function to predict (classification will then consist in predicting
    whether g(x) <= 0 or not)"""
    return 5. - x[:, 1] - .5 * x[:, 0] ** 2.

# Design of experiments
X = np.array([[-4.61611719, -6.00099547],
              [4.10469096, 5.32782448],
              [0.00000000, -0.50000000],
              [-6.17289014, -4.6984743],
              [1.3109306, -6.93271427],
              [-5.03823144, 3.10584743],
              [-2.87600388, 6.74310541],
              [5.21301203, 4.26386883]])

# Observations
y = np.array(g(X) > 0, dtype=int)

# Instantiate and fit Gaussian Process Model
kernel = C(0.1, (1e-5, np.inf)) * DotProduct(sigma_0=0.1) ** 2
gp = GaussianProcessClassifier(kernel=kernel)
gp.fit(X, y)
print("Learned kernel: %s " % gp.kernel_)

# Evaluate real function and the predicted probability
res = 50
x1, x2 = np.meshgrid(np.linspace(- lim, lim, res),
                     np.linspace(- lim, lim, res))
xx = np.vstack([x1.reshape(x1.size), x2.reshape(x2.size)]).T
```

(continues on next page)

(continued from previous page)

```
y_true = g(xx)
y_prob = gp.predict_proba(xx)[:, 1]
y_true = y_true.reshape((res, res))
y_prob = y_prob.reshape((res, res))

# Plot the probabilistic classification iso-values
fig = plt.figure(1)
ax = fig.gca()
ax.axes.set_aspect('equal')
plt.xticks([])
plt.yticks([])
ax.set_xticklabels([])
ax.set_yticklabels([])
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

cax = plt.imshow(y_prob, cmap=cm.gray_r, alpha=0.8,
                 extent=(-lim, lim, -lim, lim))
norm = plt.matplotlib.colors.Normalize(vmin=0., vmax=0.9)
cb = plt.colorbar(cax, ticks=[0., 0.2, 0.4, 0.6, 0.8, 1.], norm=norm)
cb.set_label(r'${\rm \mathbb{P}}\left[\widehat{G}(\mathbf{x}) \leq 0\right]$')
plt.clim(0, 1)

plt.plot(X[y <= 0, 0], X[y <= 0, 1], 'r.', markersize=12)

plt.plot(X[y > 0, 0], X[y > 0, 1], 'b.', markersize=12)

plt.contour(x1, x2, y_true, [0.], colors='k', linestyles='dashdot')

cs = plt.contour(x1, x2, y_prob, [0.666], colors='b',
                 linestyles='solid')
plt.clabel(cs, fontsize=11)

cs = plt.contour(x1, x2, y_prob, [0.5], colors='k',
                 linestyles='dashed')
plt.clabel(cs, fontsize=11)

cs = plt.contour(x1, x2, y_prob, [0.334], colors='r',
                 linestyles='solid')
plt.clabel(cs, fontsize=11)

plt.show()
```

Total running time of the script: (0 minutes 0.573 seconds)

Estimated memory usage: 8 MB

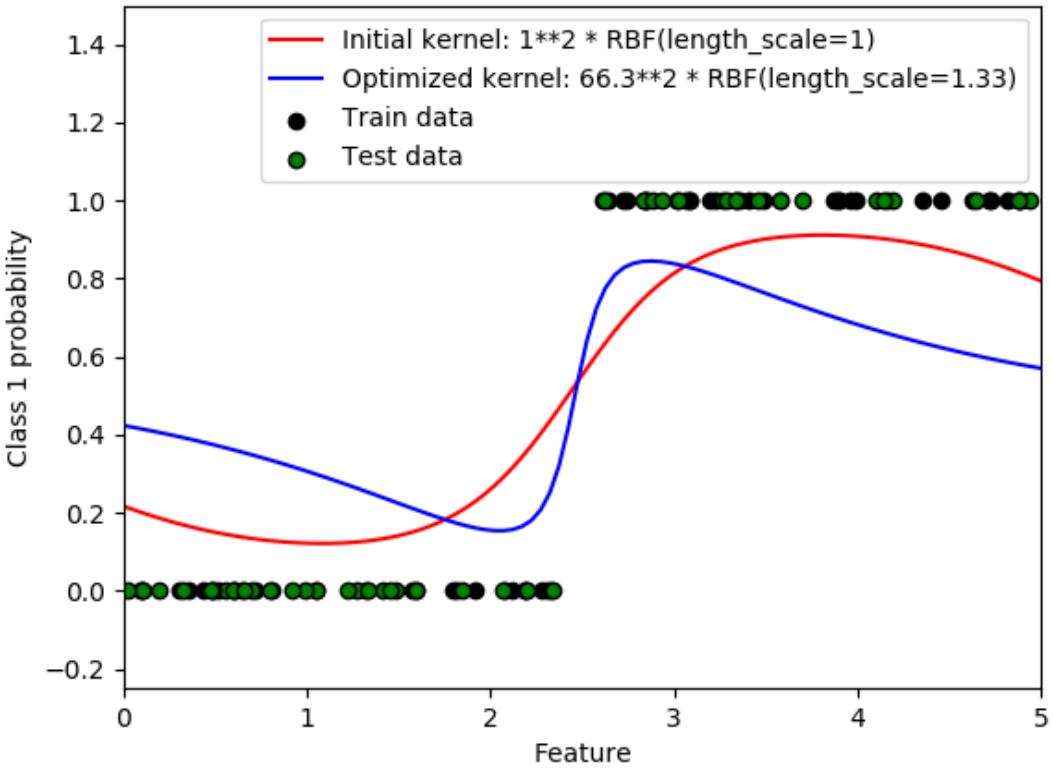
Note: Click *here* to download the full example code or to run this example in your browser via Binder

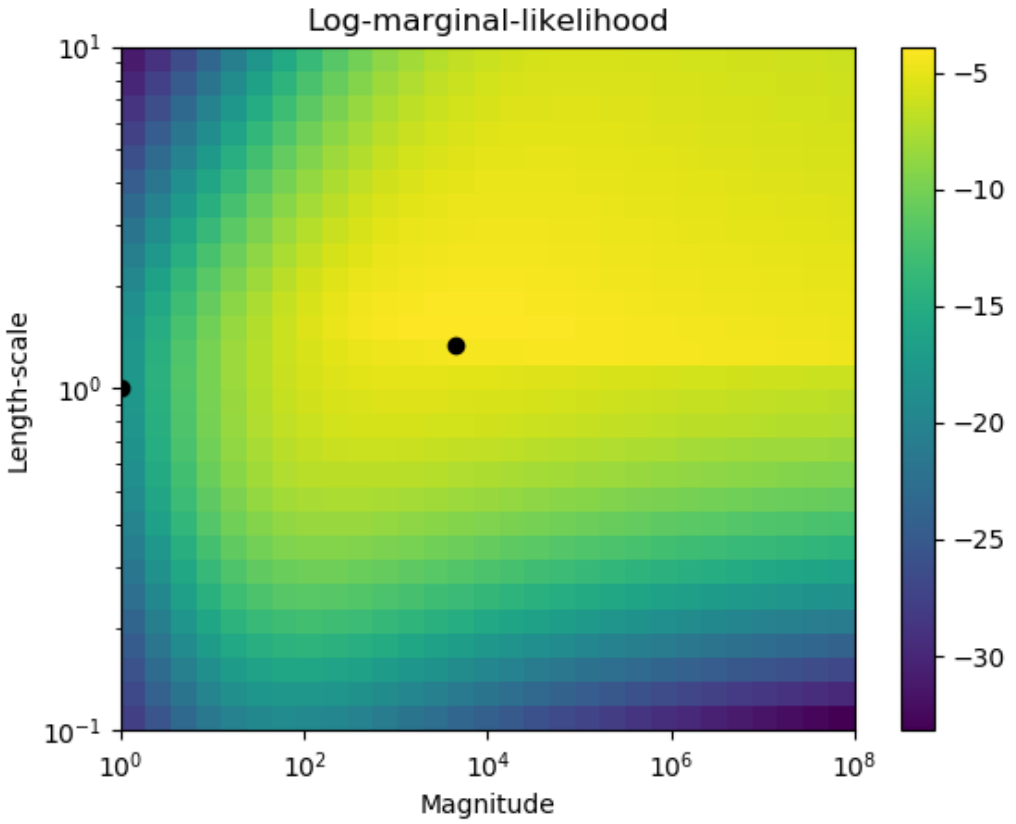
6.15.6 Probabilistic predictions with Gaussian process classification (GPC)

This example illustrates the predicted probability of GPC for an RBF kernel with different choices of the hyperparameters. The first figure shows the predicted probability of GPC with arbitrarily chosen hyperparameters and with the hyperparameters corresponding to the maximum log-marginal-likelihood (LML).

While the hyperparameters chosen by optimizing LML have a considerable larger LML, they perform slightly worse according to the log-loss on test data. The figure shows that this is because they exhibit a steep change of the class probabilities at the class boundaries (which is good) but have predicted probabilities close to 0.5 far away from the class boundaries (which is bad) This undesirable effect is caused by the Laplace approximation used internally by GPC.

The second figure shows the log-marginal-likelihood for different choices of the kernel's hyperparameters, highlighting the two choices of the hyperparameters used in the first figure by black dots.





Out:

```
Log Marginal Likelihood (initial): -17.598
Log Marginal Likelihood (optimized): -3.875
Accuracy: 1.000 (initial) 1.000 (optimized)
Log-loss: 0.214 (initial) 0.319 (optimized)
```

```
print(__doc__)

# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
#
# License: BSD 3 clause

import numpy as np

from matplotlib import pyplot as plt

from sklearn.metrics import accuracy_score, log_loss
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
```

(continues on next page)

(continued from previous page)

```
# Generate data
train_size = 50
rng = np.random.RandomState(0)
X = rng.uniform(0, 5, 100)[:, np.newaxis]
y = np.array(X[:, 0] > 2.5, dtype=int)

# Specify Gaussian Processes with fixed and optimized hyperparameters
gp_fix = GaussianProcessClassifier(kernel=1.0 * RBF(length_scale=1.0),
                                   optimizer=None)
gp_fix.fit(X[:train_size], y[:train_size])

gp_opt = GaussianProcessClassifier(kernel=1.0 * RBF(length_scale=1.0))
gp_opt.fit(X[:train_size], y[:train_size])

print("Log Marginal Likelihood (initial): %.3f"
      % gp_fix.log_marginal_likelihood(gp_fix.kernel_.theta))
print("Log Marginal Likelihood (optimized): %.3f"
      % gp_opt.log_marginal_likelihood(gp_opt.kernel_.theta))

print("Accuracy: %.3f (initial) %.3f (optimized)"
      % (accuracy_score(y[:train_size], gp_fix.predict(X[:train_size])),
         accuracy_score(y[:train_size], gp_opt.predict(X[:train_size]))))
print("Log-loss: %.3f (initial) %.3f (optimized)"
      % (log_loss(y[:train_size], gp_fix.predict_proba(X[:train_size])[:, 1]),
         log_loss(y[:train_size], gp_opt.predict_proba(X[:train_size])[:, 1])))


# Plot posteriors
plt.figure()
plt.scatter(X[:train_size, 0], y[:train_size], c='k', label="Train data",
            edgecolors=(0, 0, 0))
plt.scatter(X[train_size:, 0], y[train_size:], c='g', label="Test data",
            edgecolors=(0, 0, 0))
X_ = np.linspace(0, 5, 100)
plt.plot(X_, gp_fix.predict_proba(X_[:, np.newaxis])[:, 1], 'r',
         label="Initial kernel: %s" % gp_fix.kernel_)
plt.plot(X_, gp_opt.predict_proba(X_[:, np.newaxis])[:, 1], 'b',
         label="Optimized kernel: %s" % gp_opt.kernel_)
plt.xlabel("Feature")
plt.ylabel("Class 1 probability")
plt.xlim(0, 5)
plt.ylim(-0.25, 1.5)
plt.legend(loc="best")

# Plot LML landscape
plt.figure()
theta0 = np.logspace(0, 8, 30)
theta1 = np.logspace(-1, 1, 29)
Theta0, Theta1 = np.meshgrid(theta0, theta1)
LML = [[gp_opt.log_marginal_likelihood(np.log([Theta0[i, j], Theta1[i, j]]))
        for i in range(Theta0.shape[0])] for j in range(Theta0.shape[1])]
LML = np.array(LML).T
plt.plot(np.exp(gp_fix.kernel_.theta)[0], np.exp(gp_fix.kernel_.theta)[1],
         'ko', zorder=10)
plt.plot(np.exp(gp_opt.kernel_.theta)[0], np.exp(gp_opt.kernel_.theta)[1],
         'ko', zorder=10)
plt.pcolor(Theta0, Theta1, LML)
```

(continues on next page)

(continued from previous page)

```
plt.xscale("log")
plt.yscale("log")
plt.colorbar()
plt.xlabel("Magnitude")
plt.ylabel("Length-scale")
plt.title("Log-marginal-likelihood")

plt.show()
```

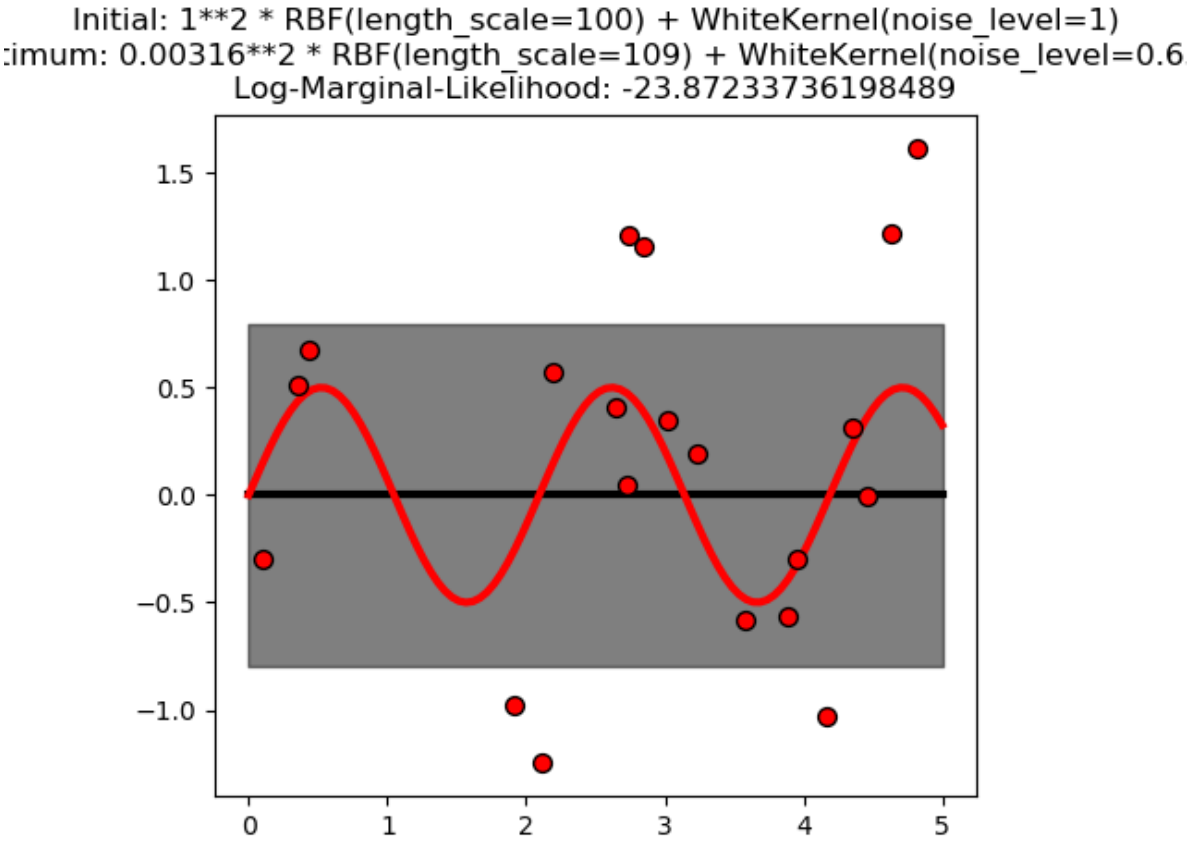
Total running time of the script: (0 minutes 2.692 seconds)

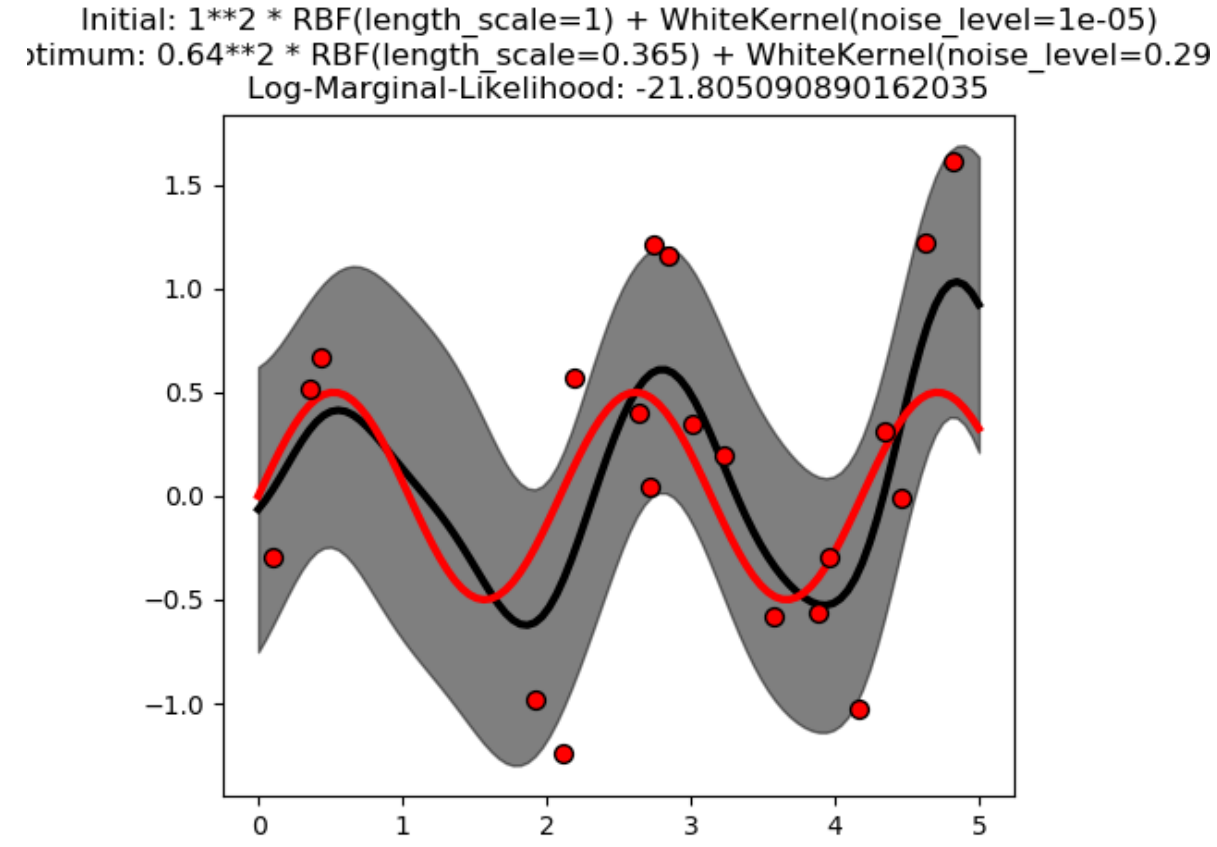
Estimated memory usage: 8 MB

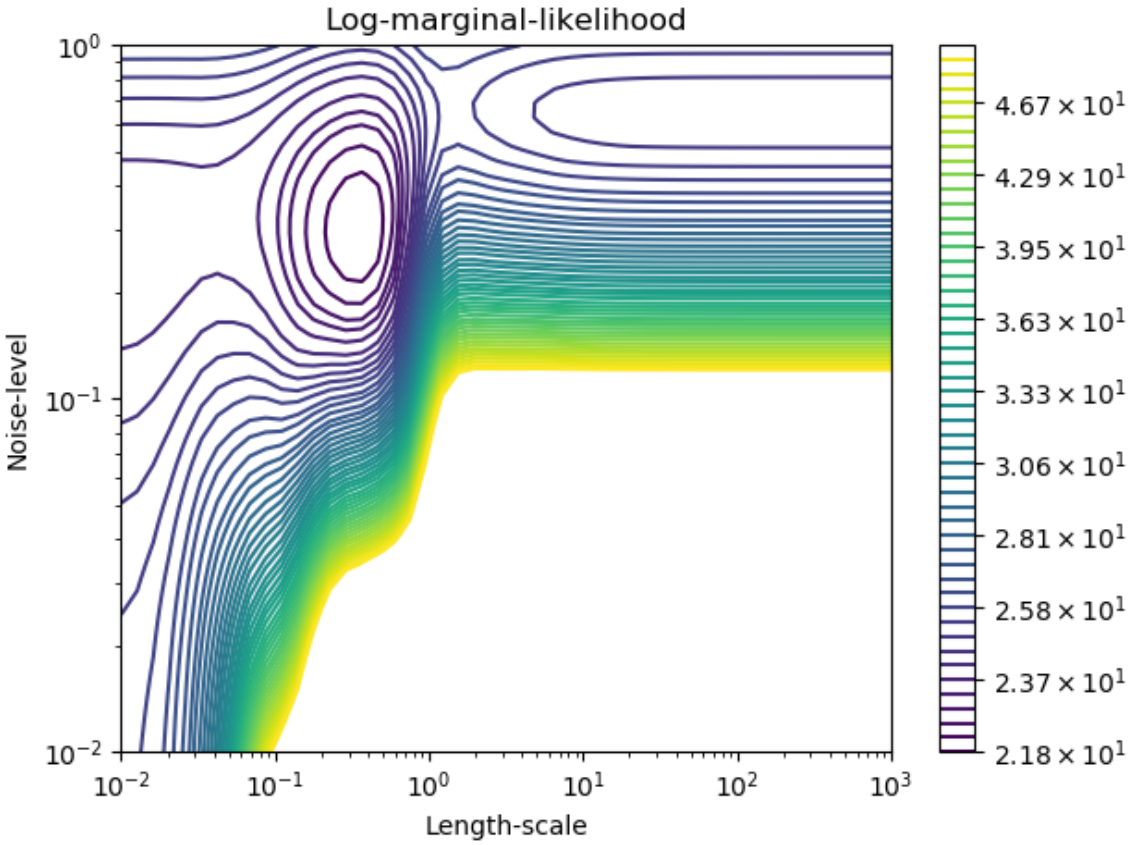
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.15.7 Gaussian process regression (GPR) with noise-level estimation

This example illustrates that GPR with a sum-kernel including a WhiteKernel can estimate the noise level of data. An illustration of the log-marginal-likelihood (LML) landscape shows that there exist two local maxima of LML. The first corresponds to a model with a high noise level and a large length scale, which explains all variations in the data by noise. The second one has a smaller noise level and shorter length scale, which explains most of the variation by the noise-free functional relationship. The second model has a higher likelihood; however, depending on the initial value for the hyperparameters, the gradient-based optimization might also converge to the high-noise solution. It is thus important to repeat the optimization several times for different initializations.


Initial: 1**2 * RBF(length_scale=100) + WhiteKernel(noise_level=1)
imum: 0.00316**2 * RBF(length_scale=109) + WhiteKernel(noise_level=0.6
Log-Marginal-Likelihood: -23.87233736198489
1.5
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5


Initial: 1**2 * RBF(length_scale=1) + WhiteKernel(noise_level=1e-05)
ɔtimum: 0.64**2 * RBF(length_scale=0.365) + WhiteKernel(noise_level=0.29
Log-Marginal-Likelihood: -21.805090890162035
1.5
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5



-

```
print(__doc__)

# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
#
# License: BSD 3 clause

import numpy as np

from matplotlib import pyplot as plt
from matplotlib.colors import LogNorm

from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel


rng = np.random.RandomState(0)
X = rng.uniform(0, 5, 20)[:, np.newaxis]
y = 0.5 * np.sin(3 * X[:, 0]) + rng.normal(0, 0.5, X.shape[0])

# First run
plt.figure()
kernel = 1.0 * RBF(length_scale=100.0, length_scale_bounds=(1e-2, 1e3)) \
    + WhiteKernel(noise_level=1, noise_level_bounds=(1e-10, 1e+1))
gp = GaussianProcessRegressor(kernel=kernel,
                              alpha=0.0).fit(X, y)
X_ = np.linspace(0, 5, 100)
y_mean, y_cov = gp.predict(X_[:, np.newaxis], return_cov=True)
```

(continues on next page)

(continued from previous page)

```
plt.plot(X_, y_mean, 'k', lw=3, zorder=9)
plt.fill_between(X_, y_mean - np.sqrt(np.diag(y_cov)),
                 y_mean + np.sqrt(np.diag(y_cov)),
                 alpha=0.5, color='k')
plt.plot(X_, 0.5*np.sin(3*X_), 'r', lw=3, zorder=9)
plt.scatter(X[:, 0], y, c='r', s=50, zorder=10, edgecolors=(0, 0, 0))
plt.title("Initial: %s\nOptimum: %s\nLog-Marginal-Likelihood: %s"
          % (kernel, gp.kernel_,
             gp.log_marginal_likelihood(gp.kernel_.theta)))
plt.tight_layout()

# Second run
plt.figure()
kernel = 1.0 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e3)) \
    + WhiteKernel(noise_level=1e-5, noise_level_bounds=(1e-10, 1e+1))
gp = GaussianProcessRegressor(kernel=kernel,
                              alpha=0.0).fit(X, y)
X_ = np.linspace(0, 5, 100)
y_mean, y_cov = gp.predict(X_[:, np.newaxis], return_cov=True)
plt.plot(X_, y_mean, 'k', lw=3, zorder=9)
plt.fill_between(X_, y_mean - np.sqrt(np.diag(y_cov)),
                 y_mean + np.sqrt(np.diag(y_cov)),
                 alpha=0.5, color='k')
plt.plot(X_, 0.5*np.sin(3*X_), 'r', lw=3, zorder=9)
plt.scatter(X[:, 0], y, c='r', s=50, zorder=10, edgecolors=(0, 0, 0))
plt.title("Initial: %s\nOptimum: %s\nLog-Marginal-Likelihood: %s"
          % (kernel, gp.kernel_,
             gp.log_marginal_likelihood(gp.kernel_.theta)))
plt.tight_layout()

# Plot LML landscape
plt.figure()
theta0 = np.logspace(-2, 3, 49)
theta1 = np.logspace(-2, 0, 50)
Theta0, Theta1 = np.meshgrid(theta0, theta1)
LML = [[gp.log_marginal_likelihood(np.log([0.36, Theta0[i, j], Theta1[i, j]]))
        for i in range(Theta0.shape[0])] for j in range(Theta0.shape[1])]
LML = np.array(LML).T

vmin, vmax = (-LML).min(), (-LML).max()
vmax = 50
level = np.around(np.logspace(np.log10(vmin), np.log10(vmax), 50), decimals=1)
plt.contour(Theta0, Theta1, -LML,
            levels=level, norm=LogNorm(vmin=vmin, vmax=vmax))
plt.colorbar()
plt.xscale("log")
plt.yscale("log")
plt.xlabel("Length-scale")
plt.ylabel("Noise-level")
plt.title("Log-marginal-likelihood")
plt.tight_layout()

plt.show()
```

Total running time of the script: (0 minutes 2.758 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.15.8 Gaussian Processes regression: basic introductory example

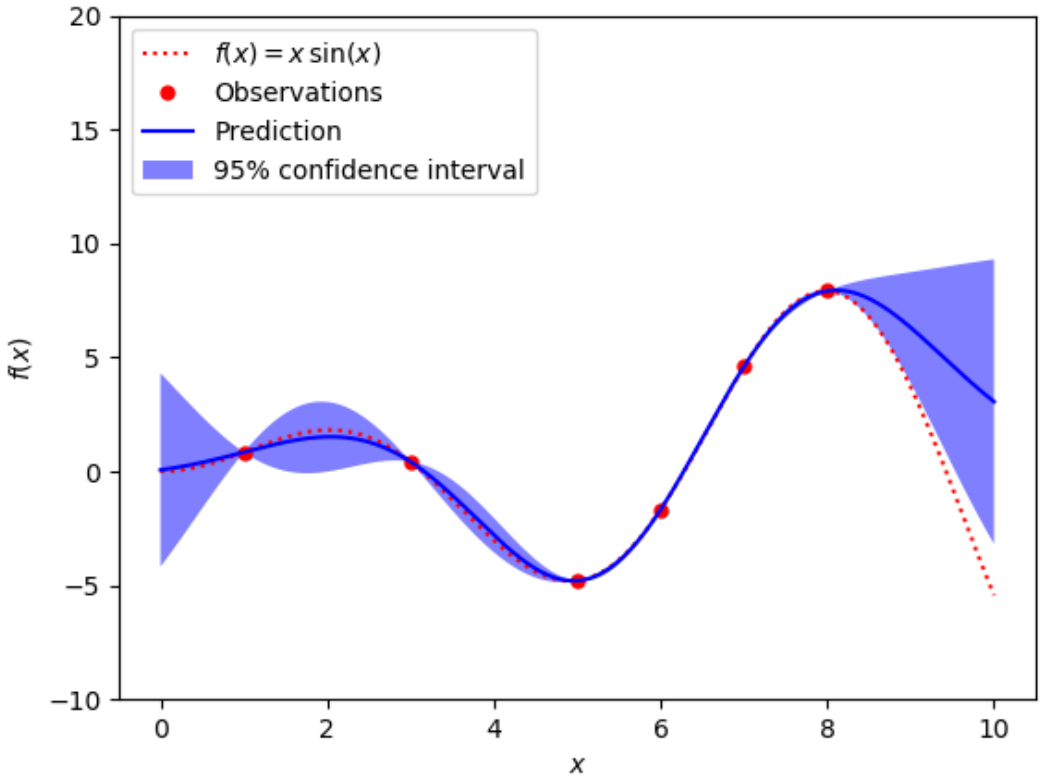
A simple one-dimensional regression example computed in two different ways:

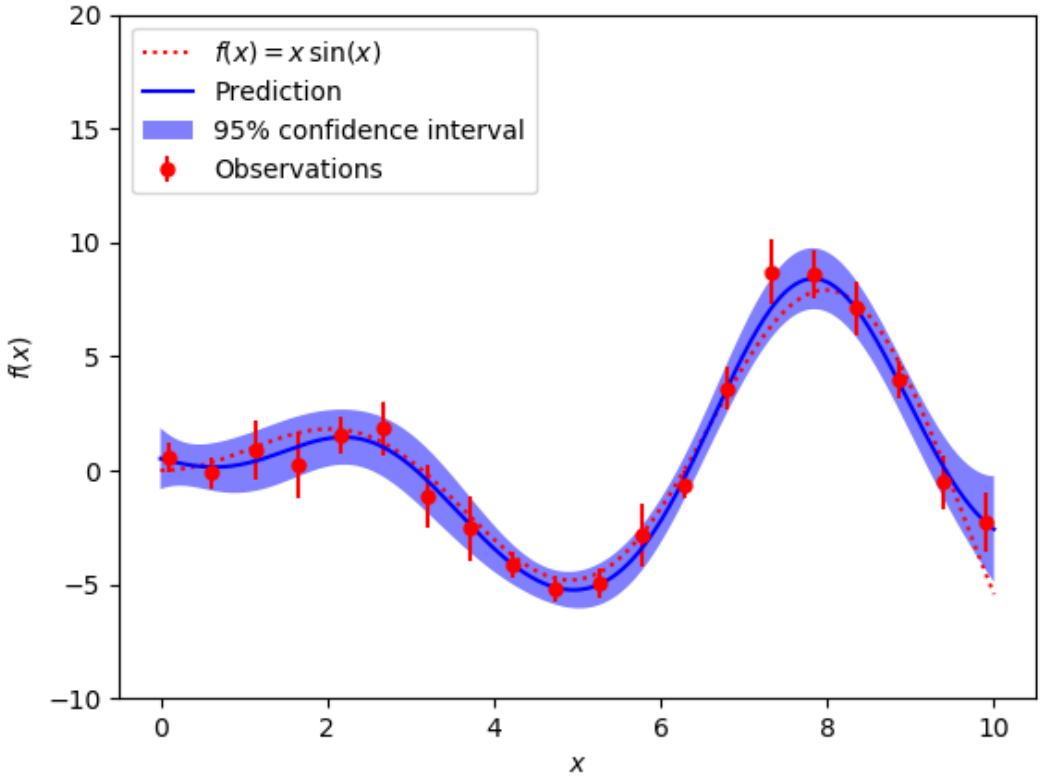
1. A noise-free case
2. A noisy case with known noise-level per datapoint

In both cases, the kernel's parameters are estimated using the maximum likelihood principle.

The figures illustrate the interpolating property of the Gaussian Process model as well as its probabilistic nature in the form of a pointwise 95% confidence interval.

Note that the parameter `alpha` is applied as a Tikhonov regularization of the assumed covariance between the training points.






```
print(__doc__)

# Author: Vincent Dubourg <vincent.dubourg@gmail.com>
#         Jake Vanderplas <vanderplas@astro.washington.edu>
#         Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>s
# License: BSD 3 clause

import numpy as np
from matplotlib import pyplot as plt

from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, ConstantKernel as C

np.random.seed(1)


def f(x):
    """The function to predict."""
    return x * np.sin(x)

# ----------------------------------------------------------------------
#  First the noiseless case
X = np.atleast_2d([1., 3., 5., 6., 7., 8.]).T

# Observations
y = f(X).ravel()
```

(continues on next page)

(continued from previous page)

```
# Mesh the input space for evaluations of the real function, the prediction and
# its MSE
x = np.atleast_2d(np.linspace(0, 10, 1000)).T

# Instantiate a Gaussian Process model
kernel = C(1.0, (1e-3, 1e3)) * RBF(10, (1e-2, 1e2))
gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)

# Fit to data using Maximum Likelihood Estimation of the parameters
gp.fit(X, y)

# Make the prediction on the meshed x-axis (ask for MSE as well)
y_pred, sigma = gp.predict(x, return_std=True)

# Plot the function, the prediction and the 95% confidence interval based on
# the MSE
plt.figure()
plt.plot(x, f(x), 'r:', label=r'$f(x) = x\,\sin(x)$')
plt.plot(X, y, 'r.', markersize=10, label='Observations')
plt.plot(x, y_pred, 'b-', label='Prediction')
plt.fill(np.concatenate([x, x[::-1]]),
         np.concatenate([y_pred - 1.9600 * sigma,
                        (y_pred + 1.9600 * sigma)[::-1]]),
         alpha=.5, fc='b', ec='None', label='95% confidence interval')
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
plt.ylim(-10, 20)
plt.legend(loc='upper left')

# ----------------------------------------------------------------------
# now the noisy case
X = np.linspace(0.1, 9.9, 20)
X = np.atleast_2d(X).T

# Observations and noise
y = f(X).ravel()
dy = 0.5 + 1.0 * np.random.random(y.shape)
noise = np.random.normal(0, dy)
y += noise

# Instantiate a Gaussian Process model
gp = GaussianProcessRegressor(kernel=kernel, alpha=dy ** 2,
                              n_restarts_optimizer=10)

# Fit to data using Maximum Likelihood Estimation of the parameters
gp.fit(X, y)

# Make the prediction on the meshed x-axis (ask for MSE as well)
y_pred, sigma = gp.predict(x, return_std=True)

# Plot the function, the prediction and the 95% confidence interval based on
# the MSE
plt.figure()
plt.plot(x, f(x), 'r:', label=r'$f(x) = x\,\sin(x)$')
plt.errorbar(X.ravel(), y, dy, fmt='r.', markersize=10, label='Observations')
plt.plot(x, y_pred, 'b-', label='Prediction')
plt.fill(np.concatenate([x, x[::-1]]),
```

(continues on next page)

(continued from previous page)

```
        np.concatenate([y_pred - 1.9600 * sigma,
                        (y_pred + 1.9600 * sigma)[::-1]]),
        alpha=.5, fc='b', ec='None', label='95% confidence interval')
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
plt.ylim(-10, 20)
plt.legend(loc='upper left')

plt.show()
```

Total running time of the script: (0 minutes 0.606 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.15.9 Gaussian process regression (GPR) on Mauna Loa CO2 data.

This example is based on Section 5.4.3 of "Gaussian Processes for Machine Learning" [RW2006]. It illustrates an example of complex kernel engineering and hyperparameter optimization using gradient ascent on the log-marginal-likelihood. The data consists of the monthly average atmospheric CO2 concentrations (in parts per million by volume (ppmv)) collected at the Mauna Loa Observatory in Hawaii, between 1958 and 2001. The objective is to model the CO2 concentration as a function of the time t.

The kernel is composed of several terms that are responsible for explaining different properties of the signal:

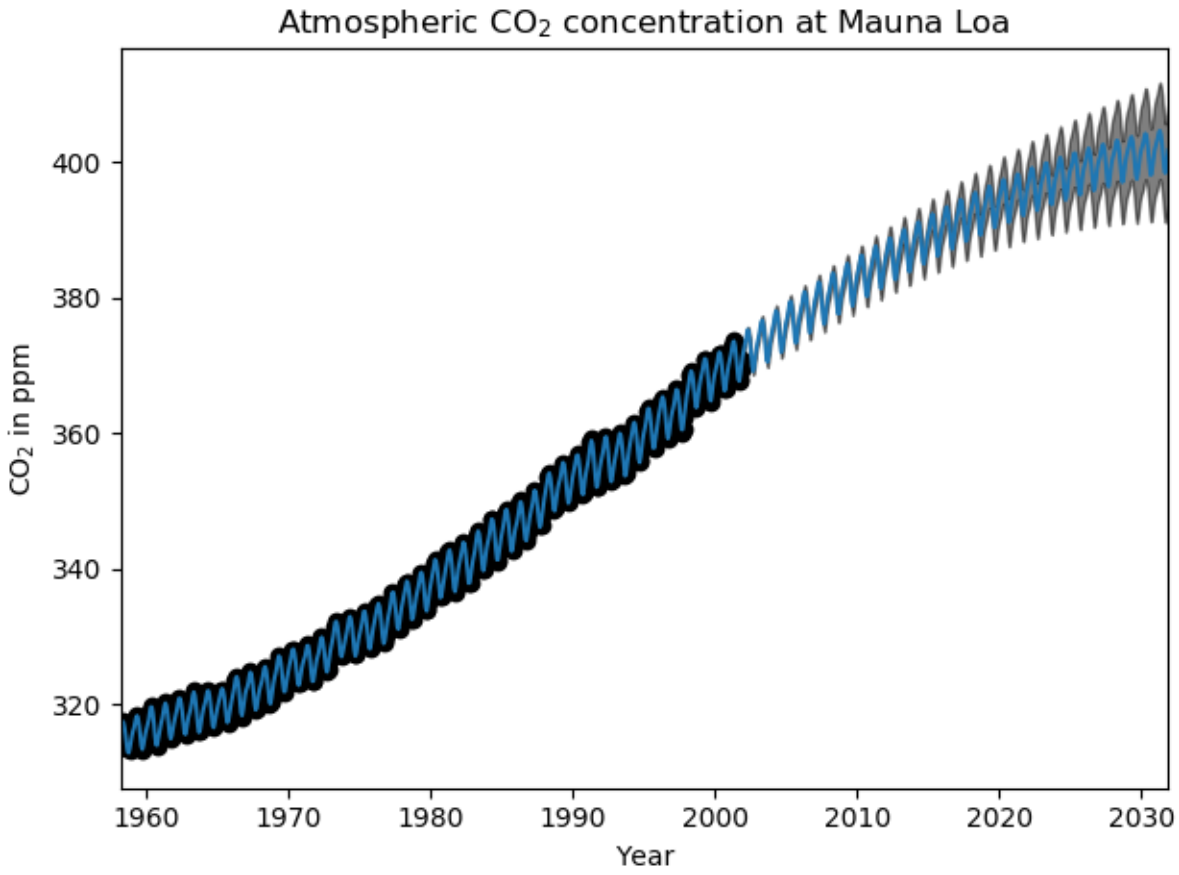
- a long term, smooth rising trend is to be explained by an RBF kernel. The RBF kernel with a large length-scale enforces this component to be smooth; it is not enforced that the trend is rising which leaves this choice to the GP. The specific length-scale and the amplitude are free hyperparameters.
- a seasonal component, which is to be explained by the periodic ExpSineSquared kernel with a fixed periodicity of 1 year. The length-scale of this periodic component, controlling its smoothness, is a free parameter. In order to allow decaying away from exact periodicity, the product with an RBF kernel is taken. The length-scale of this RBF component controls the decay time and is a further free parameter.
- smaller, medium term irregularities are to be explained by a RationalQuadratic kernel component, whose length-scale and alpha parameter, which determines the diffuseness of the length-scales, are to be determined. According to [RW2006], these irregularities can better be explained by a RationalQuadratic than an RBF kernel component, probably because it can accommodate several length-scales.
- a "noise" term, consisting of an RBF kernel contribution, which shall explain the correlated noise components such as local weather phenomena, and a WhiteKernel contribution for the white noise. The relative amplitudes and the RBF's length scale are further free parameters.

Maximizing the log-marginal-likelihood after subtracting the target's mean yields the following kernel with an LML of -83.214:

```
34.4**2 * RBF(length_scale=41.8)
+ 3.27**2 * RBF(length_scale=180) * ExpSineSquared(length_scale=1.44,
                                                   periodicity=1)
+ 0.446**2 * RationalQuadratic(alpha=17.7, length_scale=0.957)
+ 0.197**2 * RBF(length_scale=0.138) + WhiteKernel(noise_level=0.0336)
```

Thus, most of the target signal (34.4ppm) is explained by a long-term rising trend (length-scale 41.8 years). The periodic component has an amplitude of 3.27ppm, a decay time of 180 years and a length-scale of 1.44. The long

decay time indicates that we have a locally very close to periodic seasonal component. The correlated noise has an amplitude of 0.197ppm with a length scale of 0.138 years and a white-noise contribution of 0.197ppm. Thus, the overall noise level is very small, indicating that the data can be very well explained by the model. The figure shows also that the model makes very confident predictions until around 2015.



Out:

```
GPML kernel: 66**2 * RBF(length_scale=67) + 2.4**2 * RBF(length_scale=90) *␣
↪ExpSineSquared(length_scale=1.3, periodicity=1) + 0.66**2 *␣
↪RationalQuadratic(alpha=0.78, length_scale=1.2) + 0.18**2 * RBF(length_scale=0.134)␣
↪+ WhiteKernel(noise_level=0.0361)
Log-marginal-likelihood: -117.023

Learned kernel: 44.8**2 * RBF(length_scale=51.6) + 2.64**2 * RBF(length_scale=91.5) *␣
↪ExpSineSquared(length_scale=1.48, periodicity=1) + 0.536**2 *␣
↪RationalQuadratic(alpha=2.89, length_scale=0.968) + 0.188**2 * RBF(length_scale=0.
↪122) + WhiteKernel(noise_level=0.0367)
Log-marginal-likelihood: -115.050
```

```
# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
```

(continues on next page)

(continued from previous page)

```
#
# License: BSD 3 clause


import numpy as np

from matplotlib import pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels \
    import RBF, WhiteKernel, RationalQuadratic, ExpSineSquared

print(__doc__)


def load_mauna_loa_atmospheric_co2():
    ml_data = fetch_openml(data_id=41187)
    months = []
    ppmv_sums = []
    counts = []

    y = ml_data.data[:, 0]
    m = ml_data.data[:, 1]
    month_float = y + (m - 1) / 12
    ppmvs = ml_data.target

    for month, ppmv in zip(month_float, ppmvs):
        if not months or month != months[-1]:
            months.append(month)
            ppmv_sums.append(ppmv)
            counts.append(1)
        else:
            # aggregate monthly sum to produce average
            ppmv_sums[-1] += ppmv
            counts[-1] += 1

    months = np.asarray(months).reshape(-1, 1)
    avg_ppmvs = np.asarray(ppmv_sums) / counts
    return months, avg_ppmvs


X, y = load_mauna_loa_atmospheric_co2()

# Kernel with parameters given in GPML book
k1 = 66.0**2 * RBF(length_scale=67.0)  # long term smooth rising trend
k2 = 2.4**2 * RBF(length_scale=90.0) \
    * ExpSineSquared(length_scale=1.3, periodicity=1.0)  # seasonal component
# medium term irregularity
k3 = 0.66**2 \
    * RationalQuadratic(length_scale=1.2, alpha=0.78)
k4 = 0.18**2 * RBF(length_scale=0.134) \
    + WhiteKernel(noise_level=0.19**2)  # noise terms
kernel_gpml = k1 + k2 + k3 + k4

gp = GaussianProcessRegressor(kernel=kernel_gpml, alpha=0,
                              optimizer=None, normalize_y=True)
gp.fit(X, y)
```

(continues on next page)

(continued from previous page)

```
print("GPML kernel: %s" % gp.kernel_)
print("Log-marginal-likelihood: %.3f"
      % gp.log_marginal_likelihood(gp.kernel_.theta))

# Kernel with optimized parameters
k1 = 50.0**2 * RBF(length_scale=50.0)  # long term smooth rising trend
k2 = 2.0**2 * RBF(length_scale=100.0) \
    * ExpSineSquared(length_scale=1.0, periodicity=1.0,
                     periodicity_bounds="fixed")  # seasonal component
# medium term irregularities
k3 = 0.5**2 * RationalQuadratic(length_scale=1.0, alpha=1.0)
k4 = 0.1**2 * RBF(length_scale=0.1) \
    + WhiteKernel(noise_level=0.1**2,
                  noise_level_bounds=(1e-3, np.inf))  # noise terms
kernel = k1 + k2 + k3 + k4

gp = GaussianProcessRegressor(kernel=kernel, alpha=0,
                              normalize_y=True)
gp.fit(X, y)

print("\nLearned kernel: %s" % gp.kernel_)
print("Log-marginal-likelihood: %.3f"
      % gp.log_marginal_likelihood(gp.kernel_.theta))

X_ = np.linspace(X.min(), X.max() + 30, 1000)[:, np.newaxis]
y_pred, y_std = gp.predict(X_, return_std=True)

# Illustration
plt.scatter(X, y, c='k')
plt.plot(X_, y_pred)
plt.fill_between(X_[:, 0], y_pred - y_std, y_pred + y_std,
                 alpha=0.5, color='k')
plt.xlim(X_.min(), X_.max())
plt.xlabel("Year")
plt.ylabel(r"CO$_2$ in ppm")
plt.title(r"Atmospheric CO$_2$ concentration at Mauna Loa")
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 5.806 seconds)

Estimated memory usage: 37 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.15.10 Gaussian processes on discrete data structures

This example illustrates the use of Gaussian processes for regression and classification tasks on data that are not in fixed-length feature vector form. This is achieved through the use of kernel functions that operates directly on discrete structures such as variable-length sequences, trees, and graphs.

Specifically, here the input variables are some gene sequences stored as variable-length strings consisting of letters 'A', 'T', 'C', and 'G', while the output variables are floating point numbers and True/False labels in the regression and classification tasks, respectively.

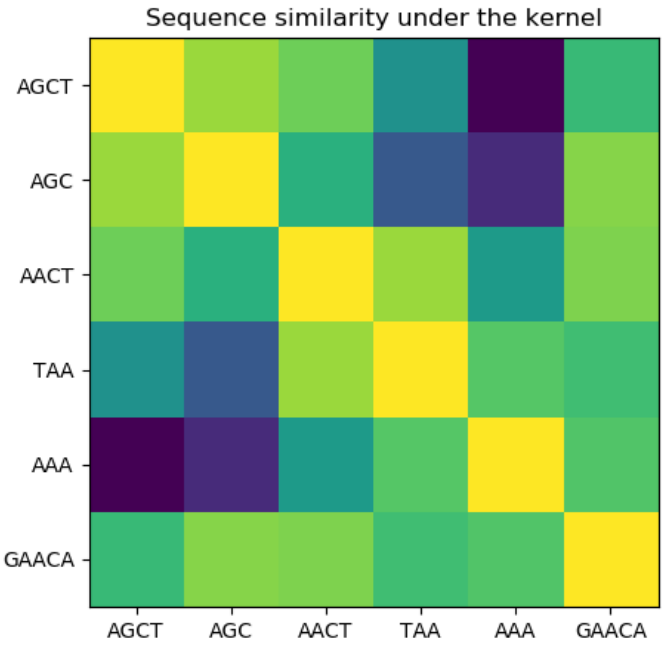
A kernel between the gene sequences is defined using R-convolution[1] by integrating a binary letter-wise kernel over all pairs of letters among a pair of strings.

This example will generate three figures.

In the first figure, we visualize the value of the kernel, i.e. the similarity of the sequences, using a colormap. Brighter color here indicates higher similarity.

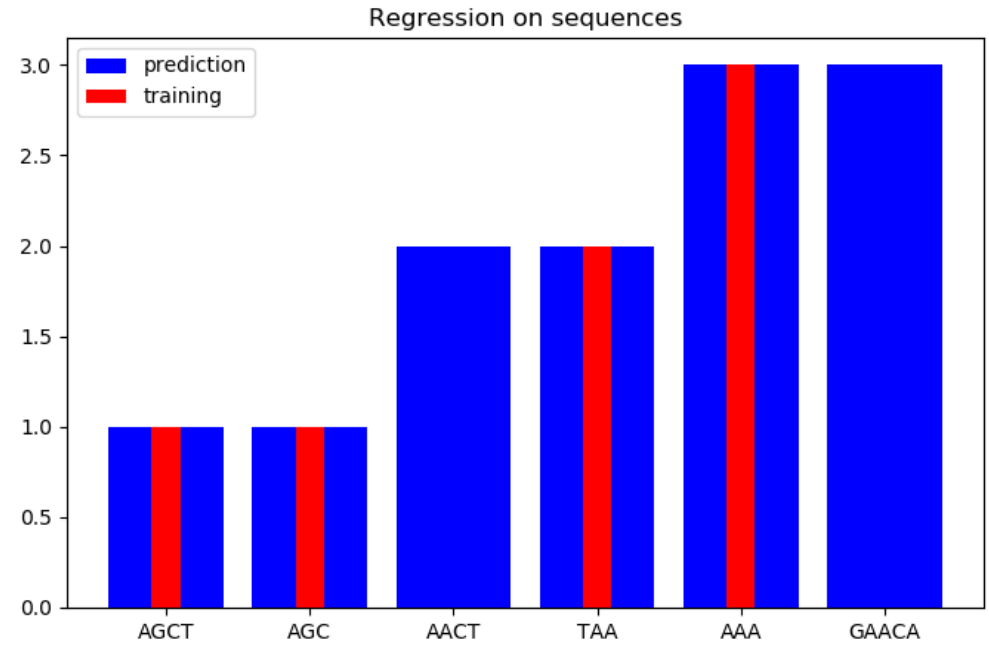
In the second figure, we show some regression result on a dataset of 6 sequences. Here we use the 1st, 2nd, 4th, and 5th sequences as the training set to make predictions on the 3rd and 6th sequences.

In the third figure, we demonstrate a classification model by training on 6 sequences and make predictions on another 5 sequences. The ground truth here is simply whether there is at least one 'A' in the sequence. Here the model makes four correct classifications and fails on one.

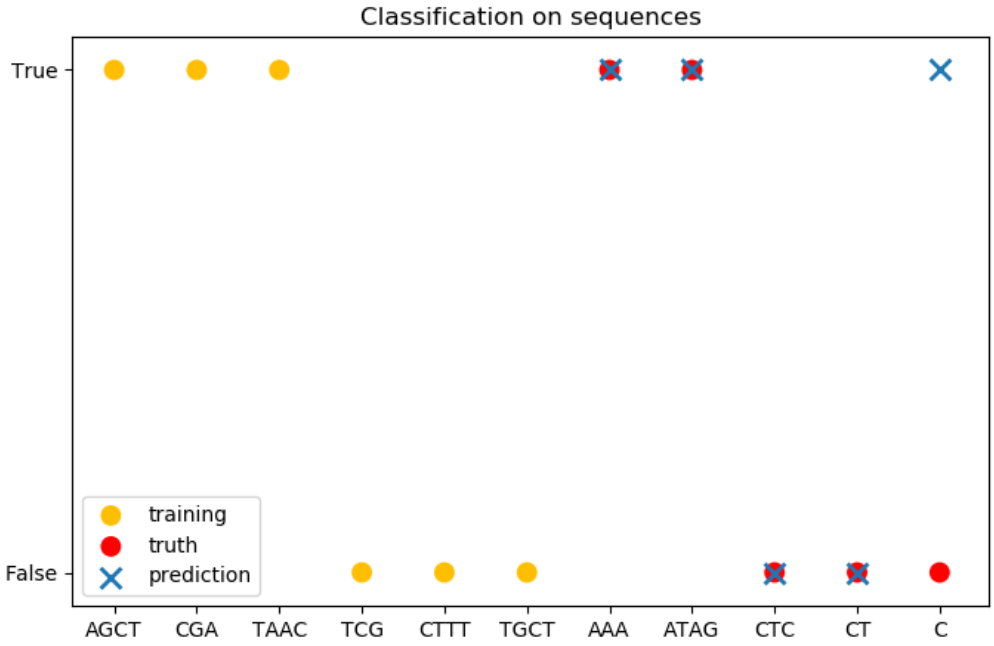


•

[1] Haussler, D. (1999). Convolution kernels on discrete structures (Vol. 646). Technical report, Department of Computer Science, University of California at Santa Cruz.



-



-

Out:

```
/home/circleci/project/sklearn/gaussian_process/_gpr.py:494: ConvergenceWarning:␣
↪lbfgs failed to converge (status=2):
ABNORMAL_TERMINATION_IN_LNSRCH.
```

(continues on next page)

(continued from previous page)

```
Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
  _check_optimize_result("lbfgs", opt_res)
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process.kernels import Kernel, Hyperparameter
from sklearn.gaussian_process.kernels import GenericKernelMixin
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.base import clone


class SequenceKernel(GenericKernelMixin, Kernel):
    '''
    A minimal (but valid) convolutional kernel for sequences of variable
    lengths.'''
    def __init__(self,
                 baseline_similarity=0.5,
                 baseline_similarity_bounds=(1e-5, 1)):
        self.baseline_similarity = baseline_similarity
        self.baseline_similarity_bounds = baseline_similarity_bounds

    @property
    def hyperparameter_baseline_similarity(self):
        return Hyperparameter("baseline_similarity",
                              "numeric",
                              self.baseline_similarity_bounds)

    def _f(self, s1, s2):
        '''
        kernel value between a pair of sequences
        '''
        return sum([1.0 if c1 == c2 else self.baseline_similarity
                    for c1 in s1
                    for c2 in s2])

    def _g(self, s1, s2):
        '''
        kernel derivative between a pair of sequences
        '''
        return sum([0.0 if c1 == c2 else 1.0
                    for c1 in s1
                    for c2 in s2])

    def __call__(self, X, Y=None, eval_gradient=False):
        if Y is None:
            Y = X
```

(continues on next page)

(continued from previous page)

```
        if eval_gradient:
            return (np.array([[self._f(x, y) for y in Y] for x in X]),
                    np.array([[[self._g(x, y)] for y in Y] for x in X]))
        else:
            return np.array([[self._f(x, y) for y in Y] for x in X])

    def diag(self, X):
        return np.array([self._f(x, x) for x in X])

    def is_stationary(self):
        return False

    def clone_with_theta(self, theta):
        cloned = clone(self)
        cloned.theta = theta
        return cloned


kernel = SequenceKernel()

'''
Sequence similarity matrix under the kernel
===========================================
'''

X = np.array(['AGCT', 'AGC', 'AACT', 'TAA', 'AAA', 'GAACA'])

K = kernel(X)
D = kernel.diag(X)

plt.figure(figsize=(8, 5))
plt.imshow(np.diag(D**-0.5).dot(K).dot(np.diag(D**-0.5)))
plt.xticks(np.arange(len(X)), X)
plt.yticks(np.arange(len(X)), X)
plt.title('Sequence similarity under the kernel')

'''
Regression
==========
'''

X = np.array(['AGCT', 'AGC', 'AACT', 'TAA', 'AAA', 'GAACA'])
Y = np.array([1.0, 1.0, 2.0, 2.0, 3.0, 3.0])

training_idx = [0, 1, 3, 4]
gp = GaussianProcessRegressor(kernel=kernel)
gp.fit(X[training_idx], Y[training_idx])

plt.figure(figsize=(8, 5))
plt.bar(np.arange(len(X)), gp.predict(X), color='b', label='prediction')
plt.bar(training_idx, Y[training_idx], width=0.2, color='r',
        alpha=1, label='training')
plt.xticks(np.arange(len(X)), X)
plt.title('Regression on sequences')
plt.legend()

'''
```

(continues on next page)

(continued from previous page)

```
Classification
==============
'''

X_train = np.array(['AGCT', 'CGA', 'TAAC', 'TCG', 'CTTT', 'TGCT'])
# whether there are 'A's in the sequence
Y_train = np.array([True, True, True, False, False, False])

gp = GaussianProcessClassifier(kernel)
gp.fit(X_train, Y_train)

X_test = ['AAA', 'ATAG', 'CTC', 'CT', 'C']
Y_test = [True, True, False, False, False]

plt.figure(figsize=(8, 5))
plt.scatter(np.arange(len(X_train)), [1.0 if c else -1.0 for c in Y_train],
            s=100, marker='o', edgecolor='none', facecolor=(1, 0.75, 0),
            label='training')
plt.scatter(len(X_train) + np.arange(len(X_test)),
            [1.0 if c else -1.0 for c in Y_test],
            s=100, marker='o', edgecolor='none', facecolor='r', label='truth')
plt.scatter(len(X_train) + np.arange(len(X_test)),
            [1.0 if c else -1.0 for c in gp.predict(X_test)],
            s=100, marker='x', edgecolor=(0, 1.0, 0.3), linewidth=2,
            label='prediction')
plt.xticks(np.arange(len(X_train) + len(X_test)),
           np.concatenate((X_train, X_test)))
plt.yticks([-1, 1], [False, True])
plt.title('Classification on sequences')
plt.legend()

plt.show()
```

Total running time of the script: (0 minutes 0.634 seconds)

Estimated memory usage: 8 MB

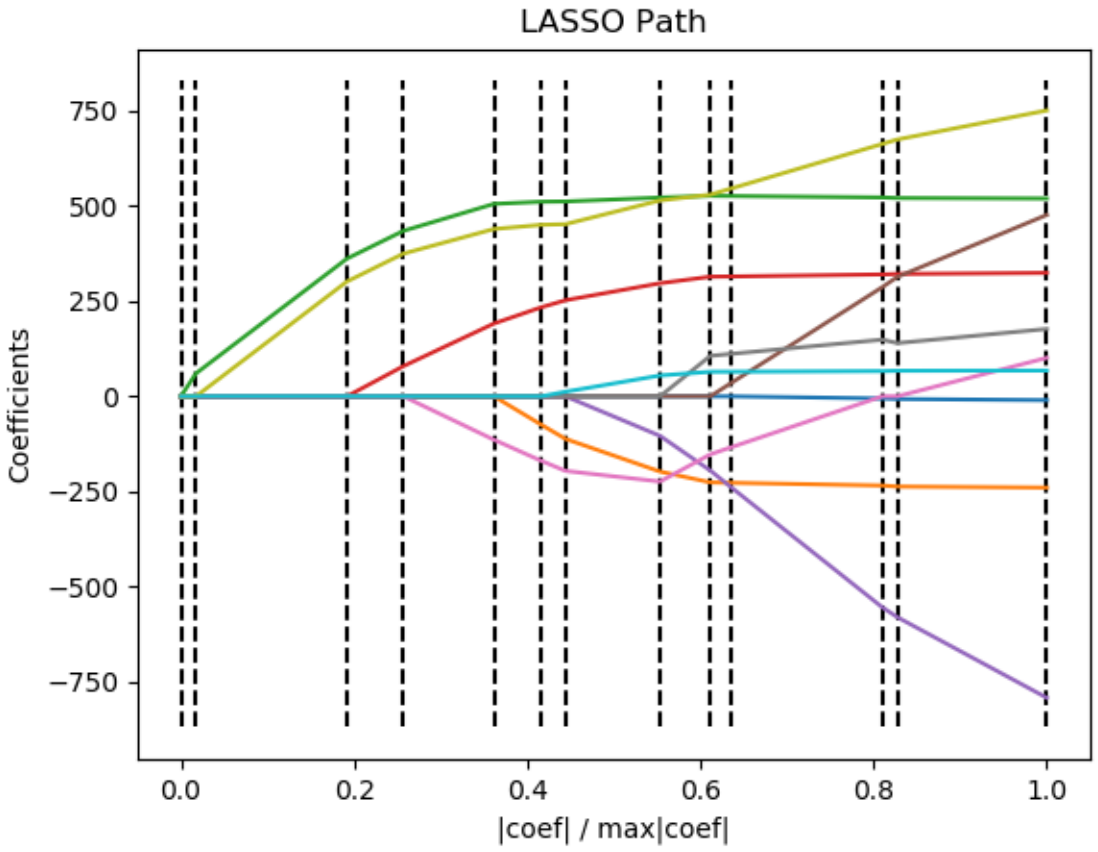
6.16 Generalized Linear Models

Examples concerning the *sklearn.linear_model* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.1 Lasso path using LARS

Computes Lasso Path along the regularization parameter using the LARS algorithm on the diabetes dataset. Each color represents a different feature of the coefficient vector, and this is displayed as a function of the regularization parameter.



Out:

```
Computing regularization path using the LARS ...
.
```

```
print(__doc__)

# Author: Fabian Pedregosa <fabian.pedregosa@inria.fr>
#         Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn import linear_model
from sklearn import datasets

X, y = datasets.load_diabetes(return_X_y=True)

print("Computing regularization path using the LARS ...")
```

(continues on next page)

(continued from previous page)

```
_, _, coefs = linear_model.lars_path(X, y, method='lasso', verbose=True)

xx = np.sum(np.abs(coefs.T), axis=1)
xx /= xx[-1]

plt.plot(xx, coefs.T)
ymin, ymax = plt.ylim()
plt.vlines(xx, ymin, ymax, linestyle='dashed')
plt.xlabel('|coef| / max|coef|')
plt.ylabel('Coefficients')
plt.title('LASSO Path')
plt.axis('tight')
plt.show()
```

Total running time of the script: (0 minutes 0.478 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

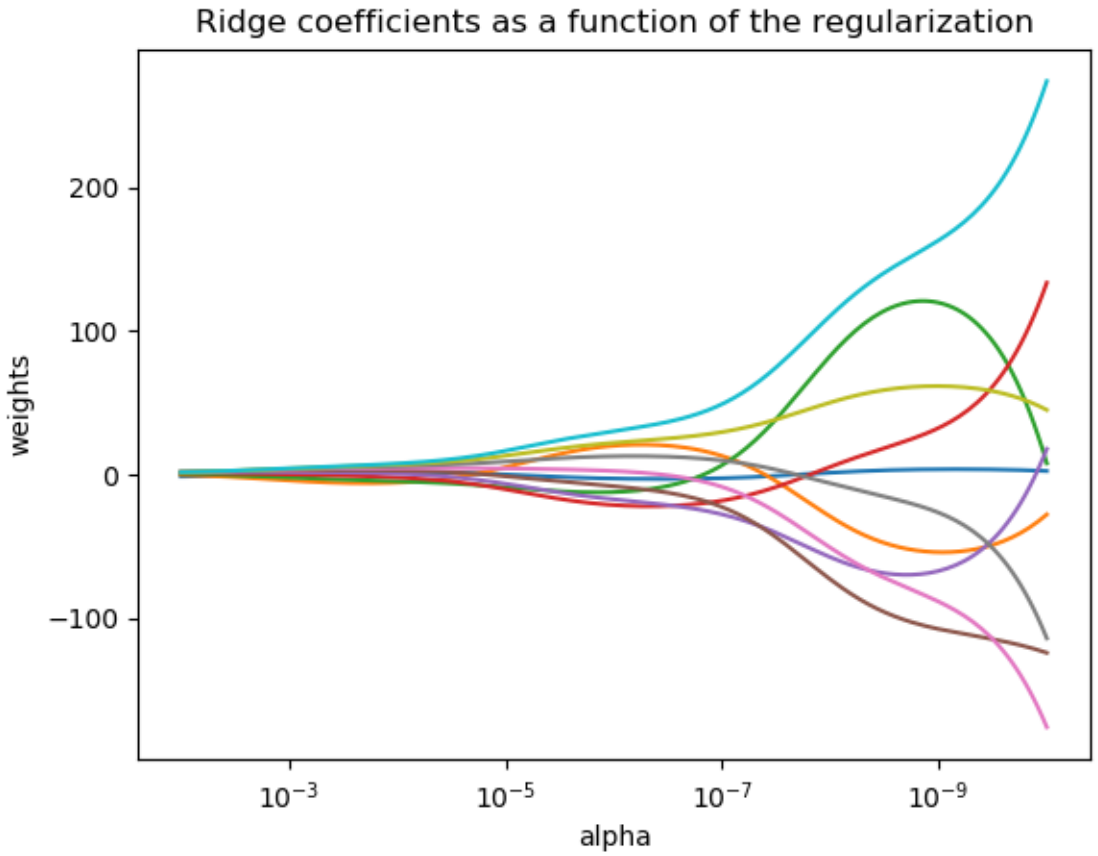
6.16.2 Plot Ridge coefficients as a function of the regularization

Shows the effect of collinearity in the coefficients of an estimator.

Ridge Regression is the estimator used in this example. Each color represents a different feature of the coefficient vector, and this is displayed as a function of the regularization parameter.

This example also shows the usefulness of applying Ridge regression to highly ill-conditioned matrices. For such matrices, a slight change in the target variable can cause huge variances in the calculated weights. In such cases, it is useful to set a certain regularization (alpha) to reduce this variation (noise).

When alpha is very large, the regularization effect dominates the squared loss function and the coefficients tend to zero. At the end of the path, as alpha tends toward zero and the solution tends towards the ordinary least squares, coefficients exhibit big oscillations. In practise it is necessary to tune alpha in such a way that a balance is maintained between both.




```
# Author: Fabian Pedregosa -- <fabian.pedregosa@inria.fr>
# License: BSD 3 clause

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model

# X is the 10x10 Hilbert matrix
X = 1. / (np.arange(1, 11) + np.arange(0, 10)[:, np.newaxis])
y = np.ones(10)

# #############################################################################
# Compute paths

n_alphas = 200
alphas = np.logspace(-10, -2, n_alphas)

coefs = []
for a in alphas:
    ridge = linear_model.Ridge(alpha=a, fit_intercept=False)
    ridge.fit(X, y)
    coefs.append(ridge.coef_)
```

(continues on next page)

(continued from previous page)

```
# ##############################################################################
# Display results

ax = plt.gca()

ax.plot(alphas, coefs)
ax.set_xscale('log')
ax.set_xlim(ax.get_xlim()[::-1])  # reverse axis
plt.xlabel('alpha')
plt.ylabel('weights')
plt.title('Ridge coefficients as a function of the regularization')
plt.axis('tight')
plt.show()
```

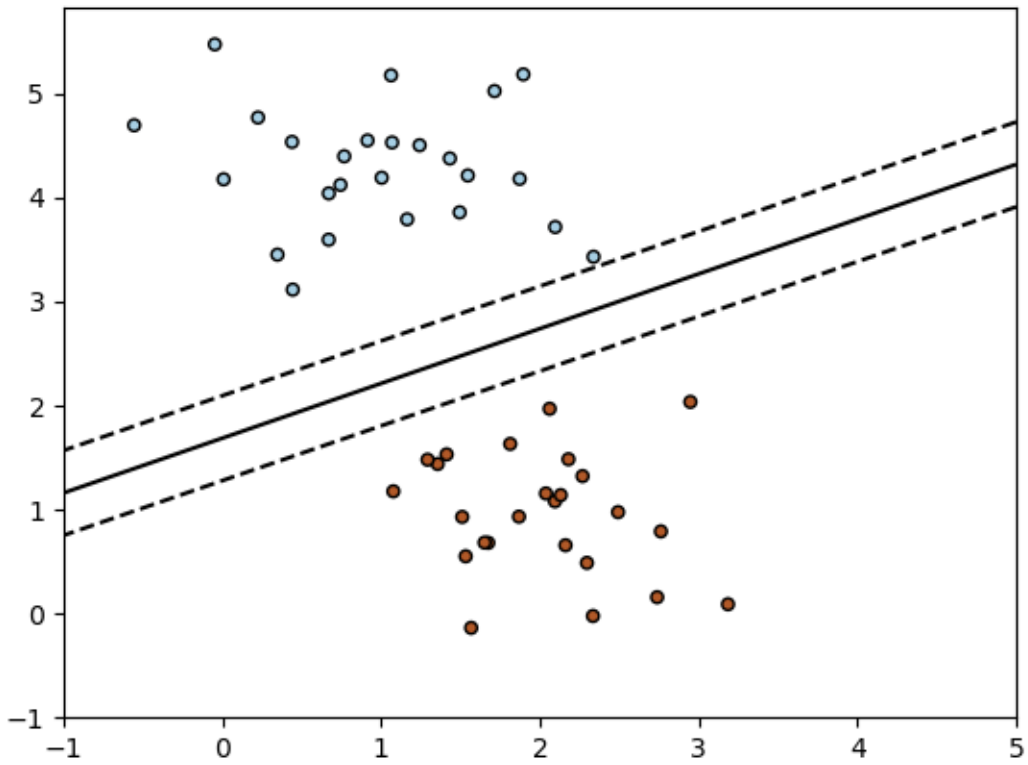
Total running time of the script: (0 minutes 0.535 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.3 SGD: Maximum margin separating hyperplane

Plot the maximum margin separating hyperplane within a two-class separable dataset using a linear Support Vector Machines classifier trained using SGD.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDClassifier
from sklearn.datasets import make_blobs

# we create 50 separable points
X, Y = make_blobs(n_samples=50, centers=2, random_state=0, cluster_std=0.60)

# fit the model
clf = SGDClassifier(loss="hinge", alpha=0.01, max_iter=200)

clf.fit(X, Y)

# plot the line, the points, and the nearest vectors to the plane
xx = np.linspace(-1, 5, 10)
yy = np.linspace(-1, 5, 10)

X1, X2 = np.meshgrid(xx, yy)
Z = np.empty(X1.shape)
for (i, j), val in np.ndenumerate(X1):
    x1 = val
    x2 = X2[i, j]
    p = clf.decision_function([[x1, x2]])
```

(continues on next page)

(continued from previous page)

```
    Z[i, j] = p[0]
levels = [-1.0, 0.0, 1.0]
linestyles = ['dashed', 'solid', 'dashed']
colors = 'k'
plt.contour(X1, X2, Z, levels, colors=colors, linestyles=linestyles)
plt.scatter(X[:, 0], X[:, 1], c=Y, cmap=plt.cm.Paired,
            edgecolor='black', s=20)

plt.axis('tight')
plt.show()
```

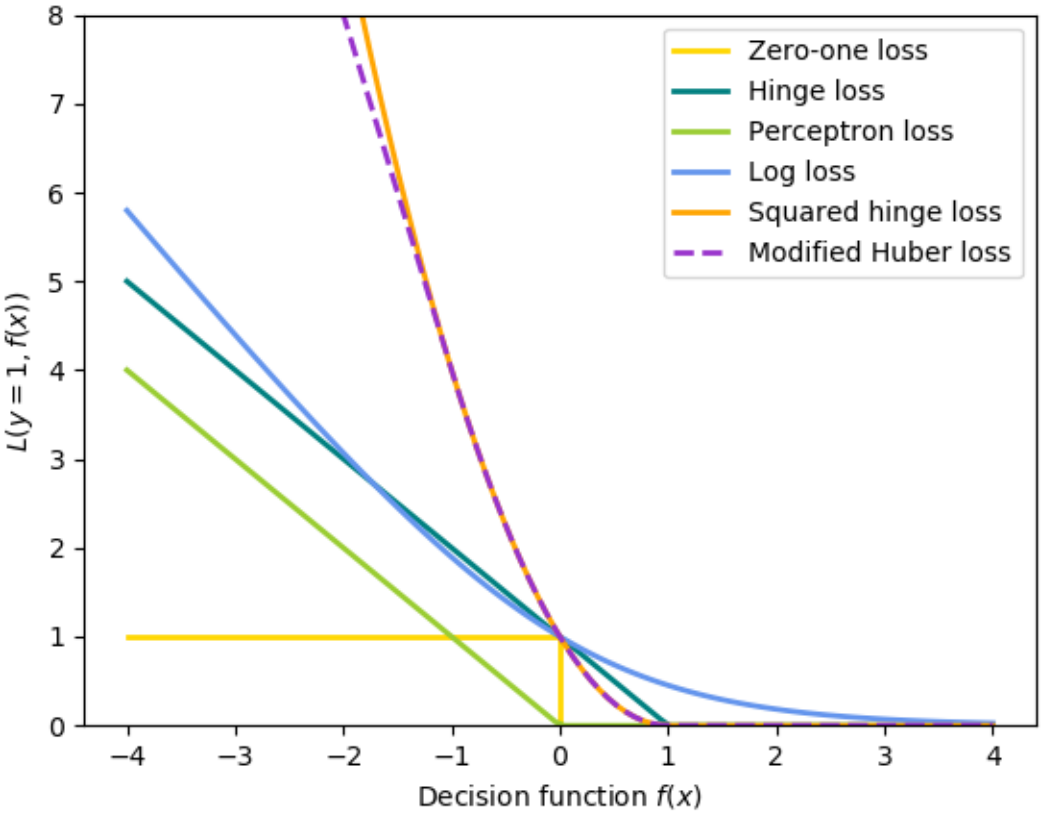
Total running time of the script: (0 minutes 0.464 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.4 SGD: convex loss functions

A plot that compares the various convex loss functions supported by *sklearn.linear_model.SGDClassifier*.



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt


def modified_huber_loss(y_true, y_pred):
    z = y_pred * y_true
    loss = -4 * z
    loss[z >= -1] = (1 - z[z >= -1]) ** 2
    loss[z >= 1.] = 0
    return loss


xmin, xmax = -4, 4
xx = np.linspace(xmin, xmax, 100)
lw = 2
plt.plot([xmin, 0, 0, xmax], [1, 1, 0, 0], color='gold', lw=lw,
         label="Zero-one loss")
plt.plot(xx, np.where(xx < 1, 1 - xx, 0), color='teal', lw=lw,
         label="Hinge loss")
plt.plot(xx, -np.minimum(xx, 0), color='yellowgreen', lw=lw,
         label="Perceptron loss")
plt.plot(xx, np.log2(1 + np.exp(-xx)), color='cornflowerblue', lw=lw,
         label="Log loss")
plt.plot(xx, np.where(xx < 1, 1 - xx, 0) ** 2, color='orange', lw=lw,
         label="Squared hinge loss")
plt.plot(xx, modified_huber_loss(xx, 1), color='darkorchid', lw=lw,
         linestyle='--', label="Modified Huber loss")
plt.ylim((0, 8))
plt.legend(loc="upper right")
plt.xlabel(r"Decision function $f(x)$")
plt.ylabel("$L(y=1, f(x))$")
plt.show()
```

Total running time of the script: (0 minutes 0.498 seconds)

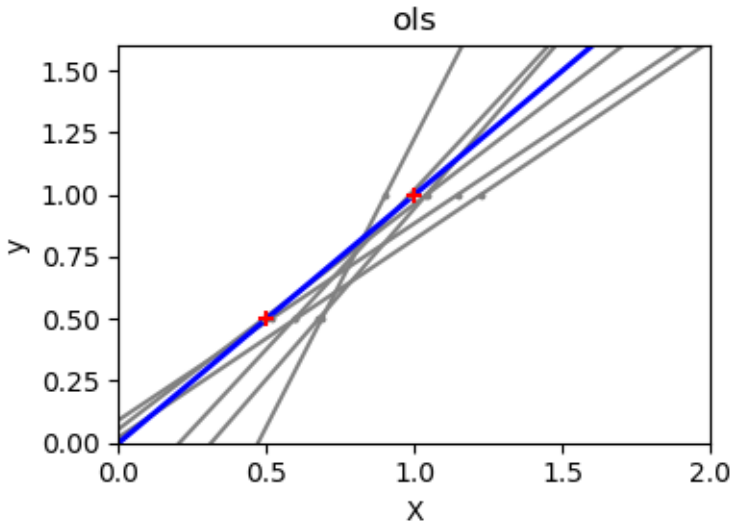
Estimated memory usage: 8 MB

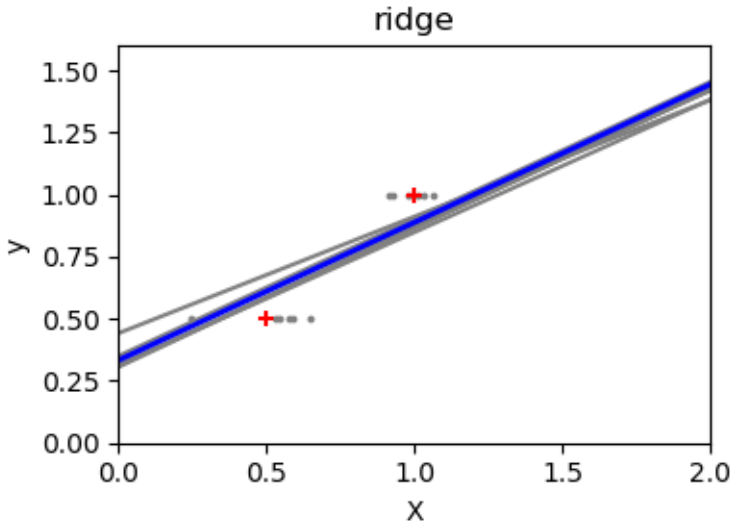
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.5 Ordinary Least Squares and Ridge Regression Variance

Due to the few points in each dimension and the straight line that linear regression uses to follow these points as well as it can, noise on the observations will cause great variance as shown in the first plot. Every line's slope can vary quite a bit for each prediction due to the noise induced in the observations.

Ridge regression is basically minimizing a penalised version of the least-squared function. The penalising `shrinks` the value of the regression coefficients. Despite the few data points in each dimension, the slope of the prediction is much more stable and the variance in the line itself is greatly reduced, in comparison to that of the standard linear regression



- 

-

```
print(__doc__)


# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause


import numpy as np
import matplotlib.pyplot as plt

from sklearn import linear_model

X_train = np.c_[.5, 1].T
y_train = [.5, 1]
X_test = np.c_[0, 2].T
```

(continues on next page)

(continued from previous page)

```
np.random.seed(0)

classifiers = dict(ols=linear_model.LinearRegression(),
                   ridge=linear_model.Ridge(alpha=.1))

for name, clf in classifiers.items():
    fig, ax = plt.subplots(figsize=(4, 3))

    for _ in range(6):
        this_X = .1 * np.random.normal(size=(2, 1)) + X_train
        clf.fit(this_X, y_train)

        ax.plot(X_test, clf.predict(X_test), color='gray')
        ax.scatter(this_X, y_train, s=3, c='gray', marker='o', zorder=10)

    clf.fit(X_train, y_train)
    ax.plot(X_test, clf.predict(X_test), linewidth=2, color='blue')
    ax.scatter(X_train, y_train, s=30, c='red', marker='+', zorder=10)

    ax.set_title(name)
    ax.set_xlim(0, 2)
    ax.set_ylim((0, 1.6))
    ax.set_xlabel('X')
    ax.set_ylabel('y')

    fig.tight_layout()

plt.show()
```

Total running time of the script: (0 minutes 0.566 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.6 Plot Ridge coefficients as a function of the L2 regularization

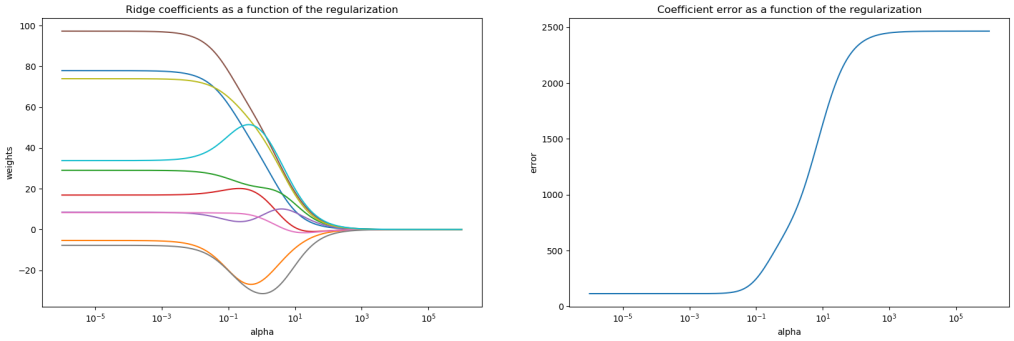
Ridge Regression is the estimator used in this example. Each color in the left plot represents one different dimension of the coefficient vector, and this is displayed as a function of the regularization parameter. The right plot shows how exact the solution is. This example illustrates how a well defined solution is found by Ridge regression and how regularization affects the coefficients and their values. The plot on the right shows how the difference of the coefficients from the estimator changes as a function of regularization.

In this example the dependent variable Y is set as a function of the input features: y = X*w + c. The coefficient vector w is randomly sampled from a normal distribution, whereas the bias term c is set to a constant.

As alpha tends toward zero the coefficients found by Ridge regression stabilize towards the randomly sampled vector w. For big alpha (strong regularisation) the coefficients are smaller (eventually converging at 0) leading to a simpler and biased solution. These dependencies can be observed on the left plot.

The right plot shows the mean squared error between the coefficients found by the model and the chosen vector w. Less regularised models retrieve the exact coefficients (error is equal to 0), stronger regularised models increase the error.

Please note that in this example the data is non-noisy, hence it is possible to extract the exact coefficients.




```
# Author: Kornel Kielczewski -- <kornel.k@plusnet.pl>

print(__doc__)

import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import make_regression
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error

clf = Ridge()

X, y, w = make_regression(n_samples=10, n_features=10, coef=True,
                          random_state=1, bias=3.5)

coefs = []
errors = []

alphas = np.logspace(-6, 6, 200)

# Train the model with different regularisation strengths
for a in alphas:
    clf.set_params(alpha=a)
    clf.fit(X, y)
    coefs.append(clf.coef_)
    errors.append(mean_squared_error(clf.coef_, w))

# Display results
plt.figure(figsize=(20, 6))

plt.subplot(121)
ax = plt.gca()
ax.plot(alphas, coefs)
ax.set_xscale('log')
plt.xlabel('alpha')
plt.ylabel('weights')
plt.title('Ridge coefficients as a function of the regularization')
plt.axis('tight')

plt.subplot(122)
ax = plt.gca()
ax.plot(alphas, errors)
ax.set_xscale('log')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('alpha')
plt.ylabel('error')
plt.title('Coefficient error as a function of the regularization')
plt.axis('tight')

plt.show()
```

Total running time of the script: (0 minutes 0.691 seconds)

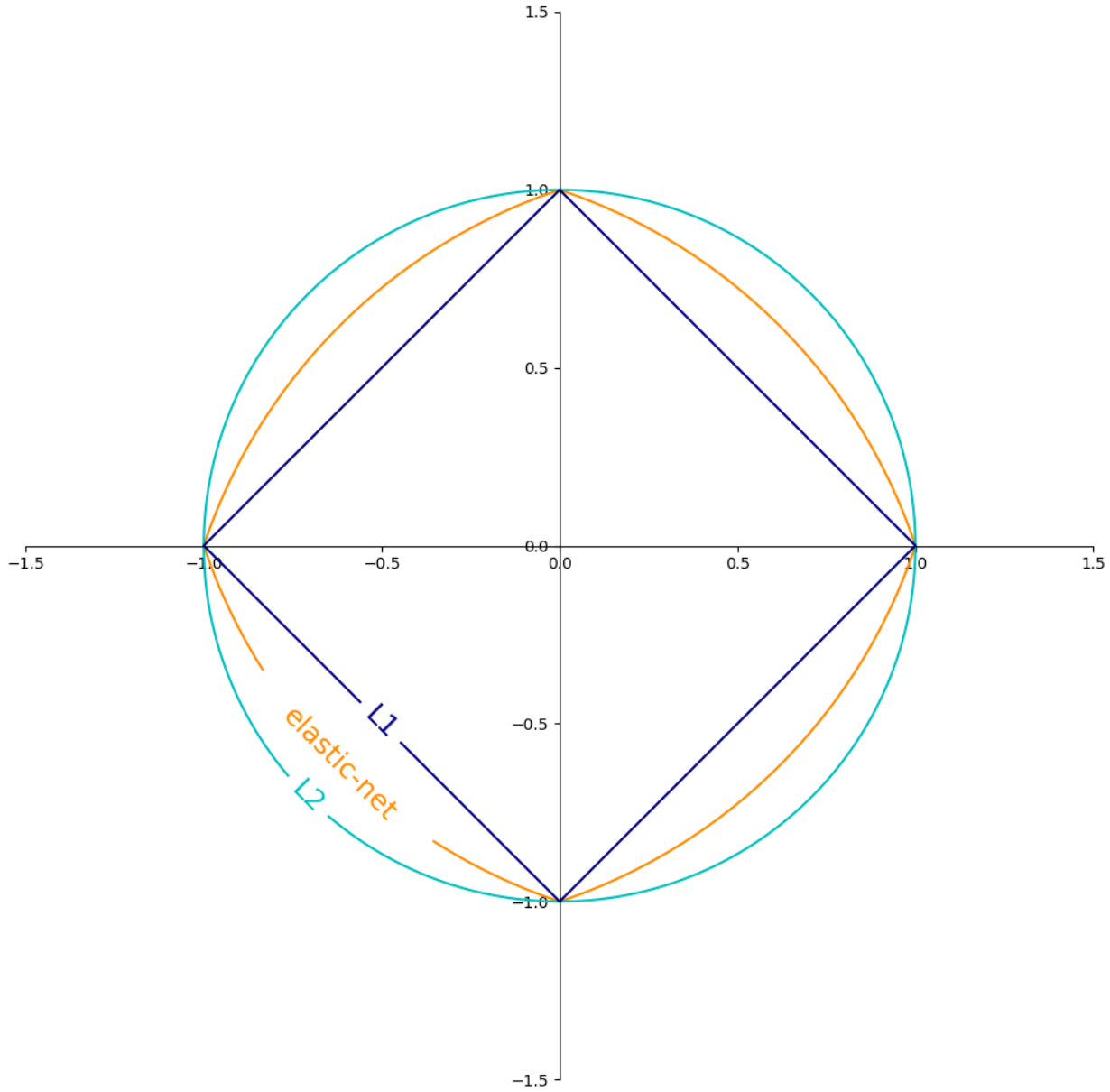
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.7 SGD: Penalties

Contours of where the penalty is equal to 1 for the three penalties L1, L2 and elastic-net.

All of the above are supported by *SGDClassifier* and *SGDRegressor*.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

l1_color = "navy"
l2_color = "c"
elastic_net_color = "darkorange"

line = np.linspace(-1.5, 1.5, 1001)
xx, yy = np.meshgrid(line, line)

l2 = xx ** 2 + yy ** 2
l1 = np.abs(xx) + np.abs(yy)
```

(continues on next page)

(continued from previous page)

```
rho = 0.5
elastic_net = rho * l1 + (1 - rho) * l2

plt.figure(figsize=(10, 10), dpi=100)
ax = plt.gca()

elastic_net_contour = plt.contour(xx, yy, elastic_net, levels=[1],
                                  colors=elastic_net_color)
l2_contour = plt.contour(xx, yy, l2, levels=[1], colors=l2_color)
l1_contour = plt.contour(xx, yy, l1, levels=[1], colors=l1_color)
ax.set_aspect("equal")
ax.spines['left'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('center')
ax.spines['top'].set_color('none')

plt.clabel(elastic_net_contour, inline=1, fontsize=18,
           fmt={1.0: 'elastic-net'}, manual=[(-1, -1)])
plt.clabel(l2_contour, inline=1, fontsize=18,
           fmt={1.0: 'L2'}, manual=[(-1, -1)])
plt.clabel(l1_contour, inline=1, fontsize=18,
           fmt={1.0: 'L1'}, manual=[(-1, -1)])

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.635 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

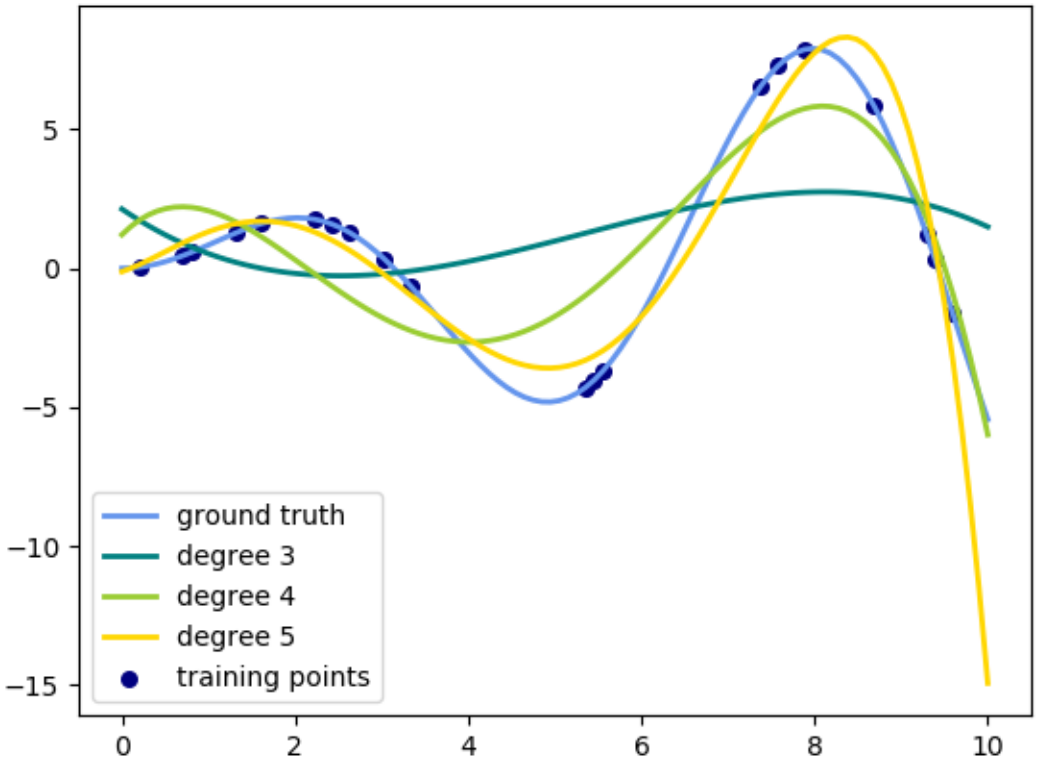
6.16.8 Polynomial interpolation

This example demonstrates how to approximate a function with a polynomial of degree n_degree by using ridge regression. Concretely, from n_samples 1d points, it suffices to build the Vandermonde matrix, which is n_samples x n_degree+1 and has the following form:

[[1, x_1, x_1 ** 2, x_1 ** 3, . . .], [1, x_2, x_2 ** 2, x_2 ** 3, . . .], . . .]

Intuitively, this matrix can be interpreted as a matrix of pseudo features (the points raised to some power). The matrix is akin to (but different from) the matrix induced by a polynomial kernel.

This example shows that you can do non-linear regression with a linear model, using a pipeline to add non-linear features. Kernel methods extend this idea and can induce very high (even infinite) dimensional feature spaces.




```
print(__doc__)

# Author: Mathieu Blondel
#         Jake Vanderplas
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline


def f(x):
    """ function to approximate by polynomial interpolation"""
    return x * np.sin(x)


# generate points used to plot
x_plot = np.linspace(0, 10, 100)

# generate points and keep a subset of them
x = np.linspace(0, 10, 100)
rng = np.random.RandomState(0)
```

(continues on next page)

(continued from previous page)

```
rng.shuffle(x)
x = np.sort(x[:20])
y = f(x)

# create matrix versions of these arrays
X = x[:, np.newaxis]
X_plot = x_plot[:, np.newaxis]

colors = ['teal', 'yellowgreen', 'gold']
lw = 2
plt.plot(x_plot, f(x_plot), color='cornflowerblue', linewidth=lw,
         label="ground truth")
plt.scatter(x, y, color='navy', s=30, marker='o', label="training points")

for count, degree in enumerate([3, 4, 5]):
    model = make_pipeline(PolynomialFeatures(degree), Ridge())
    model.fit(X, y)
    y_plot = model.predict(X_plot)
    plt.plot(x_plot, y_plot, color=colors[count], linewidth=lw,
             label="degree %d" % degree)

plt.legend(loc='lower left')

plt.show()
```

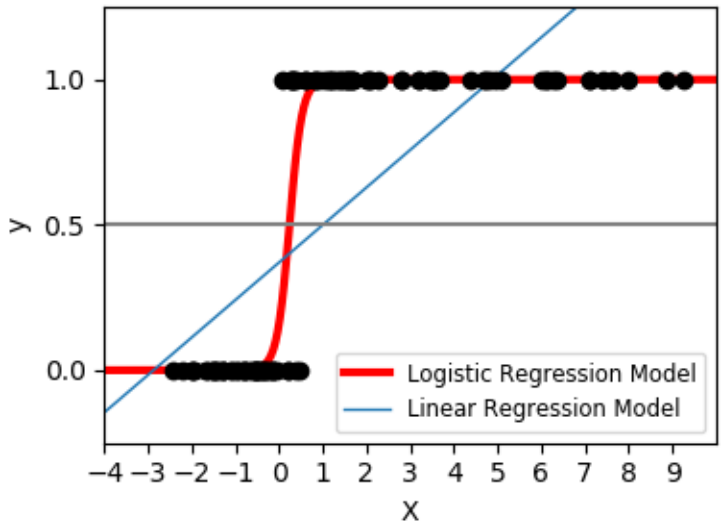
Total running time of the script: (0 minutes 0.472 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.9 Logistic function

Shown in the plot is how the logistic regression would, in this synthetic dataset, classify values as either 0 or 1, i.e. class one or two, using the logistic curve.




```
print(__doc__)


# Code source: Gael Varoquaux
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn import linear_model
from scipy.special import expit

# General a toy dataset:s it's just a straight line with some Gaussian noise:
xmin, xmax = -5, 5
n_samples = 100
np.random.seed(0)
X = np.random.normal(size=n_samples)
y = (X > 0).astype(np.float)
X[X > 0] *= 4
X += .3 * np.random.normal(size=n_samples)

X = X[:, np.newaxis]

# Fit the classifier
clf = linear_model.LogisticRegression(C=1e5)
clf.fit(X, y)

# and plot the result
plt.figure(1, figsize=(4, 3))
plt.clf()
plt.scatter(X.ravel(), y, color='black', zorder=20)
X_test = np.linspace(-5, 10, 300)

loss = expit(X_test * clf.coef_ + clf.intercept_).ravel()
plt.plot(X_test, loss, color='red', linewidth=3)

ols = linear_model.LinearRegression()
```

(continues on next page)

(continued from previous page)

```
ols.fit(X, y)
plt.plot(X_test, ols.coef_ * X_test + ols.intercept_, linewidth=1)
plt.axhline(.5, color='.5')

plt.ylabel('y')
plt.xlabel('X')
plt.xticks(range(-5, 10))
plt.yticks([0, 0.5, 1])
plt.ylim(-.25, 1.25)
plt.xlim(-4, 10)
plt.legend(('Logistic Regression Model', 'Linear Regression Model'),
           loc="lower right", fontsize='small')
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.491 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.10 Regularization path of L1- Logistic Regression

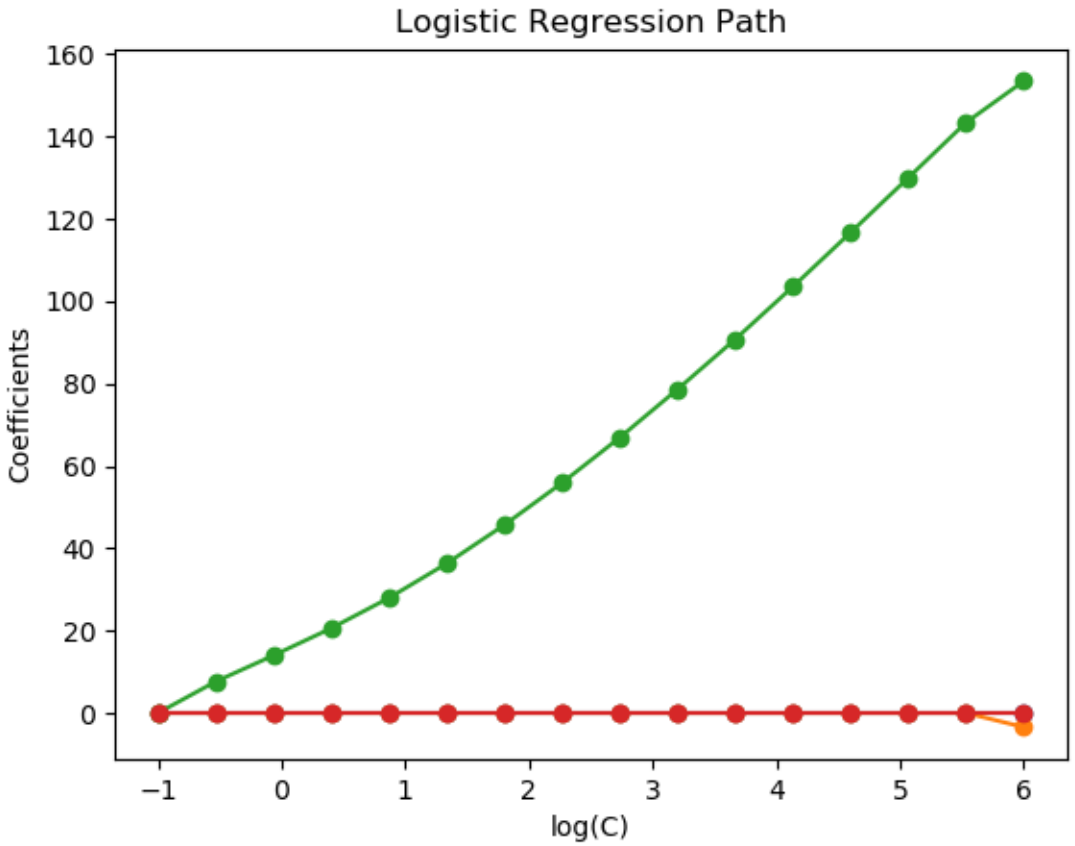
Train l1-penalized logistic regression models on a binary classification problem derived from the Iris dataset.

The models are ordered from strongest regularized to least regularized. The 4 coefficients of the models are collected and plotted as a "regularization path": on the left-hand side of the figure (strong regularizers), all the coefficients are exactly 0. When regularization gets progressively looser, coefficients can get non-zero values one after the other.

Here we choose the liblinear solver because it can efficiently optimize for the Logistic Regression loss with a non-smooth, sparsity inducing l1 penalty.

Also note that we set a low value for the tolerance to make sure that the model has converged before collecting the coefficients.

We also use warm_start=True which means that the coefficients of the models are reused to initialize the next model fit to speed-up the computation of the full-path.



Out:

```
Computing regularization path ...
This took 0.058s
```

```
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

from time import time
import numpy as np
import matplotlib.pyplot as plt

from sklearn import linear_model
from sklearn import datasets
from sklearn.svm import l1_min_c

iris = datasets.load_iris()
X = iris.data
```

(continues on next page)

(continued from previous page)

```
y = iris.target

X = X[y != 2]
y = y[y != 2]

X /= X.max()  # Normalize X to speed-up convergence

# #############################################################################
# Demo path functions

cs = l1_min_c(X, y, loss='log') * np.logspace(0, 7, 16)


print("Computing regularization path ...")
start = time()
clf = linear_model.LogisticRegression(penalty='l1', solver='liblinear',
                                      tol=1e-6, max_iter=int(1e6),
                                      warm_start=True,
                                      intercept_scaling=10000.)
coefs_ = []
for c in cs:
    clf.set_params(C=c)
    clf.fit(X, y)
    coefs_.append(clf.coef_.ravel().copy())
print("This took %0.3fs" % (time() - start))

coefs_ = np.array(coefs_)
plt.plot(np.log10(cs), coefs_, marker='o')
ymin, ymax = plt.ylim()
plt.xlabel('log(C)')
plt.ylabel('Coefficients')
plt.title('Logistic Regression Path')
plt.axis('tight')
plt.show()
```

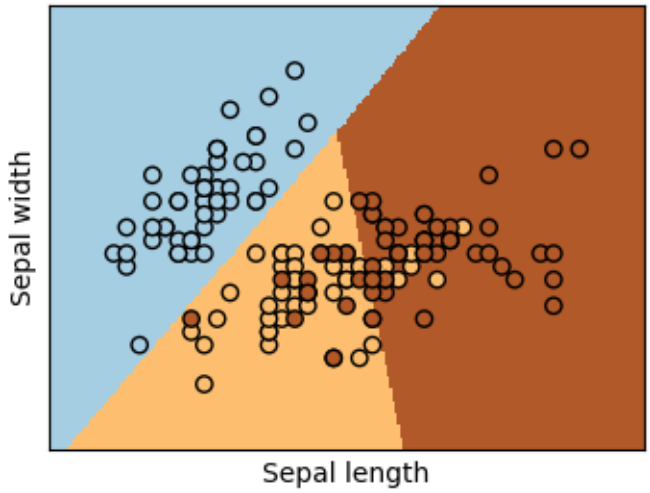
Total running time of the script: (0 minutes 0.528 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.11 Logistic Regression 3-class Classifier

Show below is a logistic-regression classifiers decision boundaries on the first two dimensions (sepal length and width) of the iris dataset. The datapoints are colored according to their labels.




```
print(__doc__)

# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn import datasets

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2]  # we only take the first two features.
Y = iris.target

logreg = LogisticRegression(C=1e5)

# Create an instance of Logistic Regression Classifier and fit the data.
logreg.fit(X, Y)

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
h = .02  # step size in the mesh
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = logreg.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure(1, figsize=(4, 3))
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=Y, edgecolors='k', cmap=plt.cm.Paired)
plt.xlabel('Sepal length')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Sepal width')

plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.xticks(())
plt.yticks(())

plt.show()
```

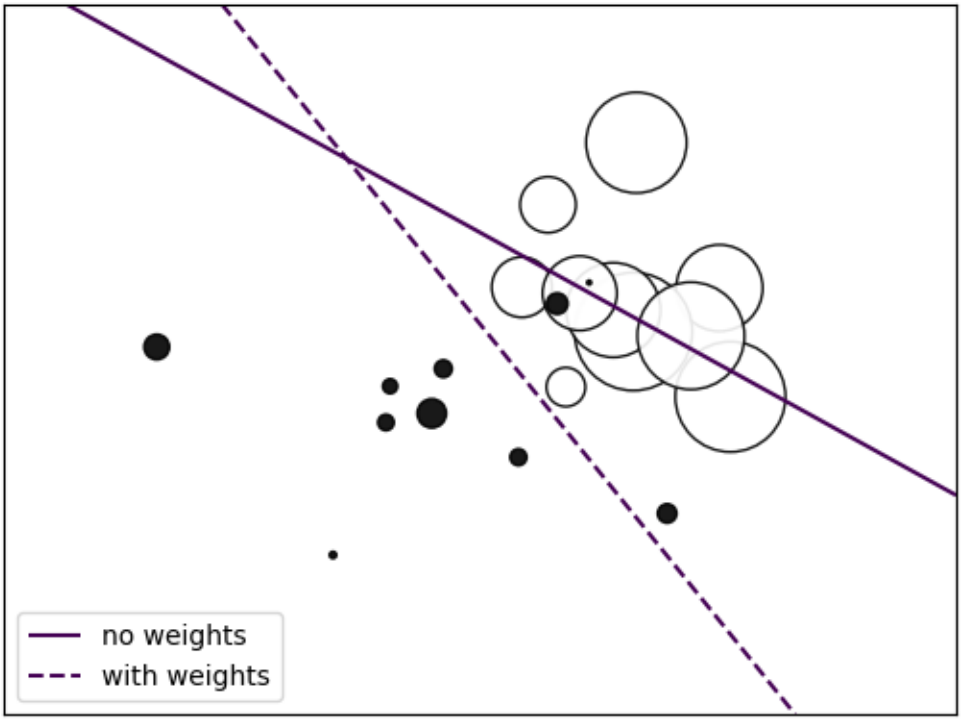
Total running time of the script: (0 minutes 0.539 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.12 SGD: Weighted samples

Plot decision function of a weighted dataset, where the size of points is proportional to its weight.




```
print(__doc__)
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model

# we create 20 points
np.random.seed(0)
X = np.r_[np.random.randn(10, 2) + [1, 1], np.random.randn(10, 2)]
y = [1] * 10 + [-1] * 10
sample_weight = 100 * np.abs(np.random.randn(20))
# and assign a bigger weight to the last 10 samples
sample_weight[:10] *= 10

# plot the weighted data points
xx, yy = np.meshgrid(np.linspace(-4, 5, 500), np.linspace(-4, 5, 500))
plt.figure()
plt.scatter(X[:, 0], X[:, 1], c=y, s=sample_weight, alpha=0.9,
            cmap=plt.cm.bone, edgecolor='black')

# fit the unweighted model
clf = linear_model.SGDClassifier(alpha=0.01, max_iter=100)
clf.fit(X, y)
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
no_weights = plt.contour(xx, yy, Z, levels=[0], linestyles=['solid'])

# fit the weighted model
clf = linear_model.SGDClassifier(alpha=0.01, max_iter=100)
clf.fit(X, y, sample_weight=sample_weight)
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
samples_weights = plt.contour(xx, yy, Z, levels=[0], linestyles=['dashed'])

plt.legend([no_weights.collections[0], samples_weights.collections[0]],
           ["no weights", "with weights"], loc="lower left")

plt.xticks(())
plt.yticks(())
plt.show()
```

Total running time of the script: (0 minutes 0.551 seconds)

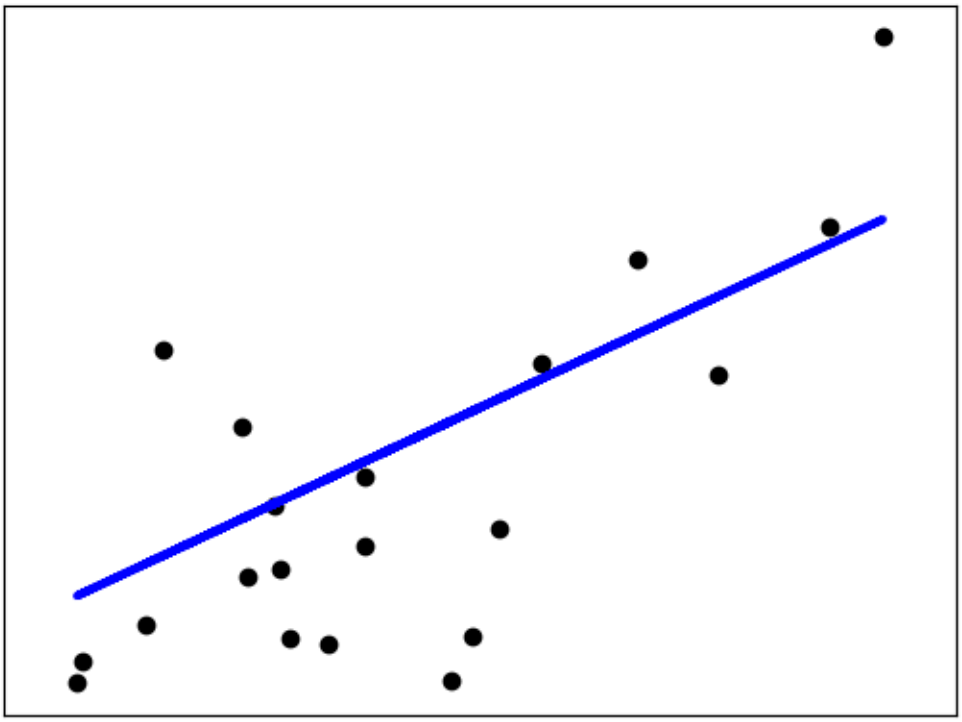
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.13 Linear Regression Example

This example uses the only the first feature of the `diabetes` dataset, in order to illustrate a two-dimensional plot of this regression technique. The straight line can be seen in the plot, showing how linear regression attempts to draw a straight line that will best minimize the residual sum of squares between the observed responses in the dataset, and the responses predicted by the linear approximation.

The coefficients, the residual sum of squares and the coefficient of determination are also calculated.



Out:

```
Coefficients:
 [938.23786125]
Mean squared error: 2548.07
Coefficient of determination: 0.47
```

```
print(__doc__)


# Code source: Jaques Grobler
# License: BSD 3 clause


import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model
from sklearn.metrics import mean_squared_error, r2_score

# Load the diabetes dataset
```

(continues on next page)

(continued from previous page)

```
diabetes_X, diabetes_y = datasets.load_diabetes(return_X_y=True)

# Use only one feature
diabetes_X = diabetes_X[:, np.newaxis, 2]

# Split the data into training/testing sets
diabetes_X_train = diabetes_X[:-20]
diabetes_X_test = diabetes_X[-20:]

# Split the targets into training/testing sets
diabetes_y_train = diabetes_y[:-20]
diabetes_y_test = diabetes_y[-20:]

# Create linear regression object
regr = linear_model.LinearRegression()

# Train the model using the training sets
regr.fit(diabetes_X_train, diabetes_y_train)

# Make predictions using the testing set
diabetes_y_pred = regr.predict(diabetes_X_test)

# The coefficients
print('Coefficients: \n', regr.coef_)
# The mean squared error
print('Mean squared error: %.2f'
      % mean_squared_error(diabetes_y_test, diabetes_y_pred))
# The coefficient of determination: 1 is perfect prediction
print('Coefficient of determination: %.2f'
      % r2_score(diabetes_y_test, diabetes_y_pred))

# Plot outputs
plt.scatter(diabetes_X_test, diabetes_y_test,  color='black')
plt.plot(diabetes_X_test, diabetes_y_pred, color='blue', linewidth=3)

plt.xticks(())
plt.yticks(())

plt.show()
```

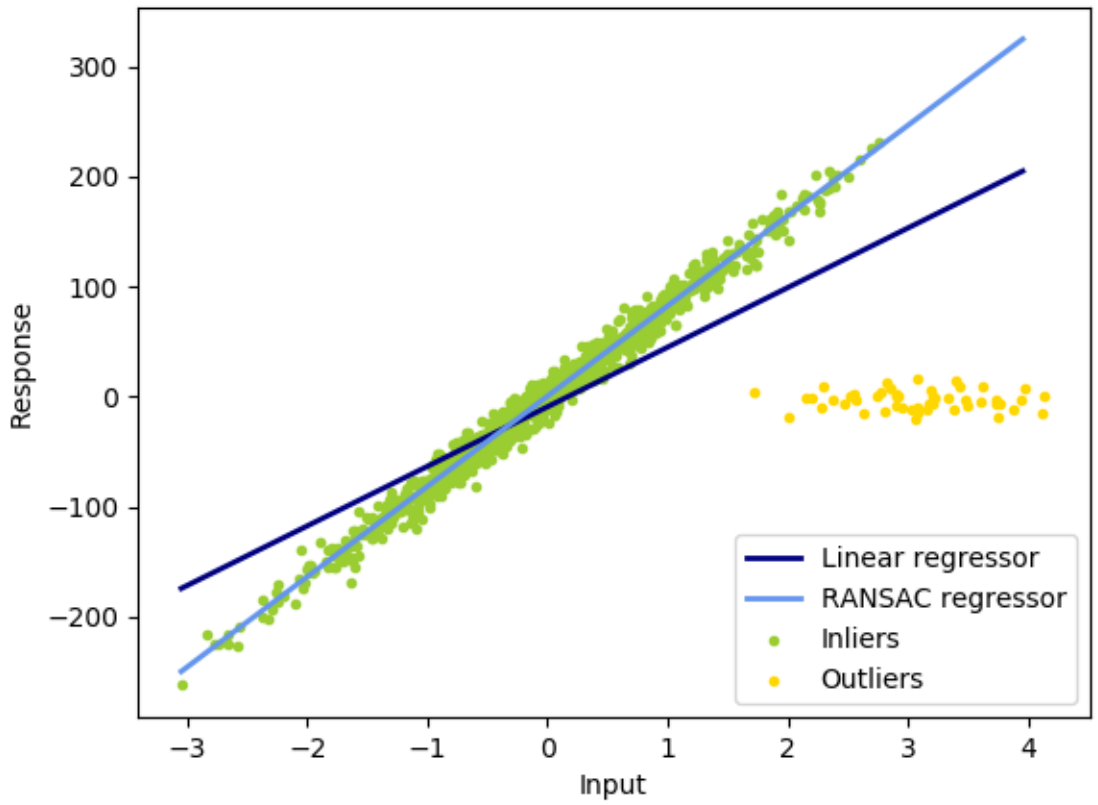
Total running time of the script: (0 minutes 0.498 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.14 Robust linear model estimation using RANSAC

In this example we see how to robustly fit a linear model to faulty data using the RANSAC algorithm.



Out:

```
Estimated coefficients (true, linear regression, RANSAC):
82.1903908407869 [54.17236387] [82.08533159]
```

```
import numpy as np
from matplotlib import pyplot as plt

from sklearn import linear_model, datasets


n_samples = 1000
n_outliers = 50


X, y, coef = datasets.make_regression(n_samples=n_samples, n_features=1,
                                      n_informative=1, noise=10,
                                      coef=True, random_state=0)

# Add outlier data
```

(continues on next page)

(continued from previous page)

```
np.random.seed(0)
X[:n_outliers] = 3 + 0.5 * np.random.normal(size=(n_outliers, 1))
y[:n_outliers] = -3 + 10 * np.random.normal(size=n_outliers)

# Fit line using all data
lr = linear_model.LinearRegression()
lr.fit(X, y)

# Robustly fit linear model with RANSAC algorithm
ransac = linear_model.RANSACRegressor()
ransac.fit(X, y)
inlier_mask = ransac.inlier_mask_
outlier_mask = np.logical_not(inlier_mask)

# Predict data of estimated models
line_X = np.arange(X.min(), X.max())[:, np.newaxis]
line_y = lr.predict(line_X)
line_y_ransac = ransac.predict(line_X)

# Compare estimated coefficients
print("Estimated coefficients (true, linear regression, RANSAC):")
print(coef, lr.coef_, ransac.estimator_.coef_)

lw = 2
plt.scatter(X[inlier_mask], y[inlier_mask], color='yellowgreen', marker='.',
            label='Inliers')
plt.scatter(X[outlier_mask], y[outlier_mask], color='gold', marker='.',
            label='Outliers')
plt.plot(line_X, line_y, color='navy', linewidth=lw, label='Linear regressor')
plt.plot(line_X, line_y_ransac, color='cornflowerblue', linewidth=lw,
         label='RANSAC regressor')
plt.legend(loc='lower right')
plt.xlabel("Input")
plt.ylabel("Response")
plt.show()
```

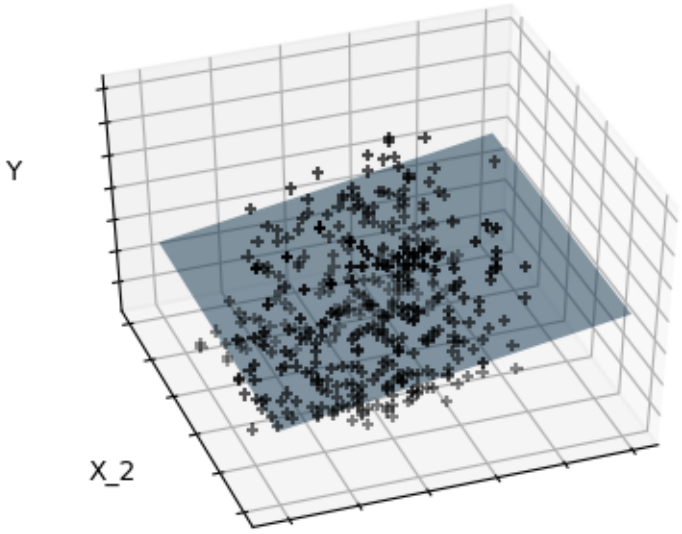
Total running time of the script: (0 minutes 0.579 seconds)

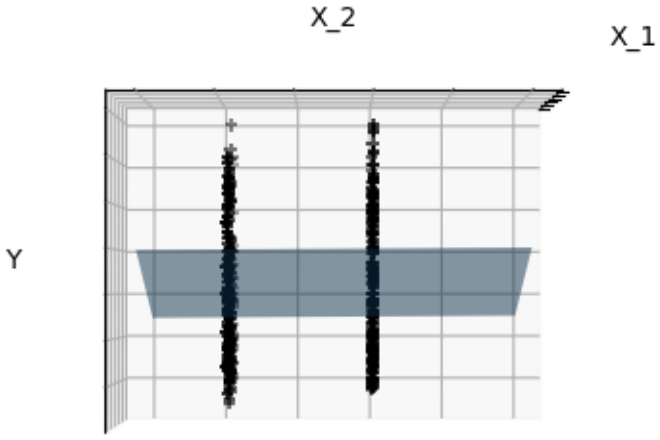
Estimated memory usage: 8 MB

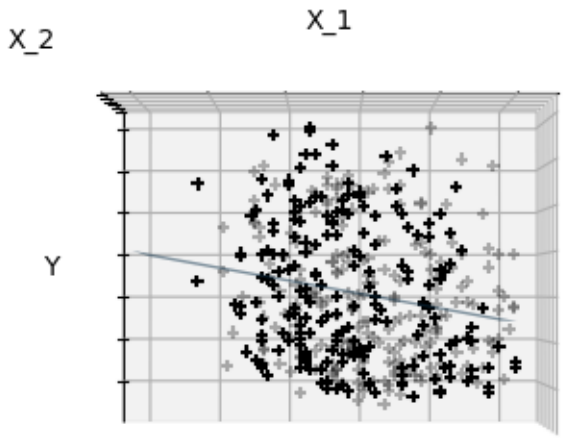
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.15 Sparsity Example: Fitting only features 1 and 2

Features 1 and 2 of the diabetes-dataset are fitted and plotted below. It illustrates that although feature 2 has a strong coefficient on the full model, it does not give us much regarding `y` when compared to just feature 1

- 
Y
X_2

- 
X_2
X_1
Y



-

```
print(__doc__)


# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

from sklearn import datasets, linear_model

X, y = datasets.load_diabetes(return_X_y=True)
indices = (0, 1)

X_train = X[:-20, indices]
X_test = X[-20:, indices]
y_train = y[:-20]
y_test = y[-20:]

ols = linear_model.LinearRegression()
ols.fit(X_train, y_train)


# ##############################################################################
# Plot the figure
def plot_figs(fig_num, elev, azim, X_train, clf):
    fig = plt.figure(fig_num, figsize=(4, 3))
    plt.clf()
    ax = Axes3D(fig, elev=elev, azim=azim)

    ax.scatter(X_train[:, 0], X_train[:, 1], y_train, c='k', marker='+')
    ax.plot_surface(np.array([[-.1, -.1], [.15, .15]]),
                    np.array([[-.1, .15], [-.1, .15]]),
                    clf.predict(np.array([[-.1, -.1, .15, .15],
                                          [-.1, .15, -.1, .15]]).T
```

(continues on next page)

(continued from previous page)

```
                                ).reshape((2, 2)),
                    alpha=.5)
    ax.set_xlabel('X_1')
    ax.set_ylabel('X_2')
    ax.set_zlabel('Y')
    ax.w_xaxis.set_ticklabels([])
    ax.w_yaxis.set_ticklabels([])
    ax.w_zaxis.set_ticklabels([])


# Generate the three different figures from different views
elev = 43.5
azim = -110
plot_figs(1, elev, azim, X_train, ols)

elev = -.5
azim = 0
plot_figs(2, elev, azim, X_train, ols)

elev = -.5
azim = 90
plot_figs(3, elev, azim, X_train, ols)

plt.show()
```

Total running time of the script: (0 minutes 0.652 seconds)

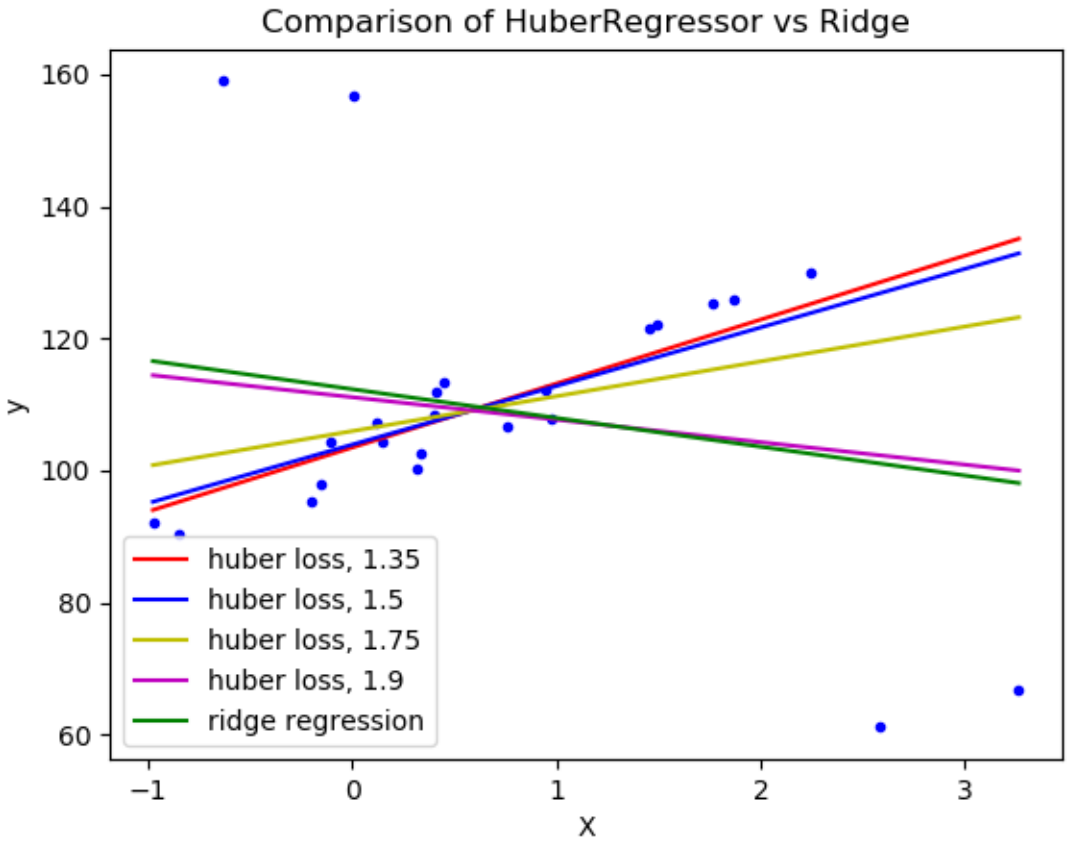
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.16 HuberRegressor vs Ridge on dataset with strong outliers

Fit Ridge and HuberRegressor on a dataset with outliers.

The example shows that the predictions in ridge are strongly influenced by the outliers present in the dataset. The Huber regressor is less influenced by the outliers since the model uses the linear loss for these. As the parameter epsilon is increased for the Huber regressor, the decision function approaches that of the ridge.




```
# Authors: Manoj Kumar mks542@nyu.edu
# License: BSD 3 clause

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_regression
from sklearn.linear_model import HuberRegressor, Ridge

# Generate toy data.
rng = np.random.RandomState(0)
X, y = make_regression(n_samples=20, n_features=1, random_state=0, noise=4.0,
                       bias=100.0)

# Add four strong outliers to the dataset.
X_outliers = rng.normal(0, 0.5, size=(4, 1))
y_outliers = rng.normal(0, 2.0, size=4)
X_outliers[:2, :] += X.max() + X.mean() / 4.
X_outliers[2:, :] += X.min() - X.mean() / 4.
y_outliers[:2] += y.min() - y.mean() / 4.
y_outliers[2:] += y.max() + y.mean() / 4.
X = np.vstack((X, X_outliers))
y = np.concatenate((y, y_outliers))
```

(continues on next page)

(continued from previous page)

```
plt.plot(X, y, 'b.')

# Fit the huber regressor over a series of epsilon values.
colors = ['r-', 'b-', 'y-', 'm-']

x = np.linspace(X.min(), X.max(), 7)
epsilon_values = [1.35, 1.5, 1.75, 1.9]
for k, epsilon in enumerate(epsilon_values):
    huber = HuberRegressor(alpha=0.0, epsilon=epsilon)
    huber.fit(X, y)
    coef_ = huber.coef_ * x + huber.intercept_
    plt.plot(x, coef_, colors[k], label="huber loss, %s" % epsilon)

# Fit a ridge regressor to compare it to huber regressor.
ridge = Ridge(alpha=0.0, random_state=0, normalize=True)
ridge.fit(X, y)
coef_ridge = ridge.coef_
coef_ = ridge.coef_ * x + ridge.intercept_
plt.plot(x, coef_, 'g-', label="ridge regression")

plt.title("Comparison of HuberRegressor vs Ridge")
plt.xlabel("X")
plt.ylabel("y")
plt.legend(loc=0)
plt.show()
```

Total running time of the script: (0 minutes 0.554 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.17 Lasso on dense and sparse data

We show that linear_model.Lasso provides the same results for dense and sparse data and that in the case of sparse data the speed is improved.

Out:

```
--- Dense matrices
Sparse Lasso done in 0.106757s
Dense Lasso done in 0.039164s
Distance between coefficients : 1.0705405751792344e-13
--- Sparse matrices
Matrix density : 0.6263000000000001 %
Sparse Lasso done in 0.193909s
Dense Lasso done in 0.866051s
Distance between coefficients : 7.928463765972842e-12
```

```
print(__doc__)

from time import time
from scipy import sparse
from scipy import linalg

from sklearn.datasets import make_regression
from sklearn.linear_model import Lasso


# #############################################################################
# The two Lasso implementations on Dense data
print("--- Dense matrices")

X, y = make_regression(n_samples=200, n_features=5000, random_state=0)
X_sp = sparse.coo_matrix(X)

alpha = 1
sparse_lasso = Lasso(alpha=alpha, fit_intercept=False, max_iter=1000)
dense_lasso = Lasso(alpha=alpha, fit_intercept=False, max_iter=1000)

t0 = time()
sparse_lasso.fit(X_sp, y)
print("Sparse Lasso done in %fs" % (time() - t0))

t0 = time()
dense_lasso.fit(X, y)
print("Dense Lasso done in %fs" % (time() - t0))

print("Distance between coefficients : %s"
      % linalg.norm(sparse_lasso.coef_ - dense_lasso.coef_))

# #############################################################################
# The two Lasso implementations on Sparse data
print("--- Sparse matrices")

Xs = X.copy()
Xs[Xs < 2.5] = 0.0
Xs = sparse.coo_matrix(Xs)
Xs = Xs.tocsc()

print("Matrix density : %s %%" % (Xs.nnz / float(X.size) * 100))

alpha = 0.1
sparse_lasso = Lasso(alpha=alpha, fit_intercept=False, max_iter=10000)
dense_lasso = Lasso(alpha=alpha, fit_intercept=False, max_iter=10000)

t0 = time()
sparse_lasso.fit(Xs, y)
print("Sparse Lasso done in %fs" % (time() - t0))

t0 = time()
dense_lasso.fit(Xs.toarray(), y)
print("Dense Lasso done in %fs" % (time() - t0))

print("Distance between coefficients : %s"
      % linalg.norm(sparse_lasso.coef_ - dense_lasso.coef_))
```

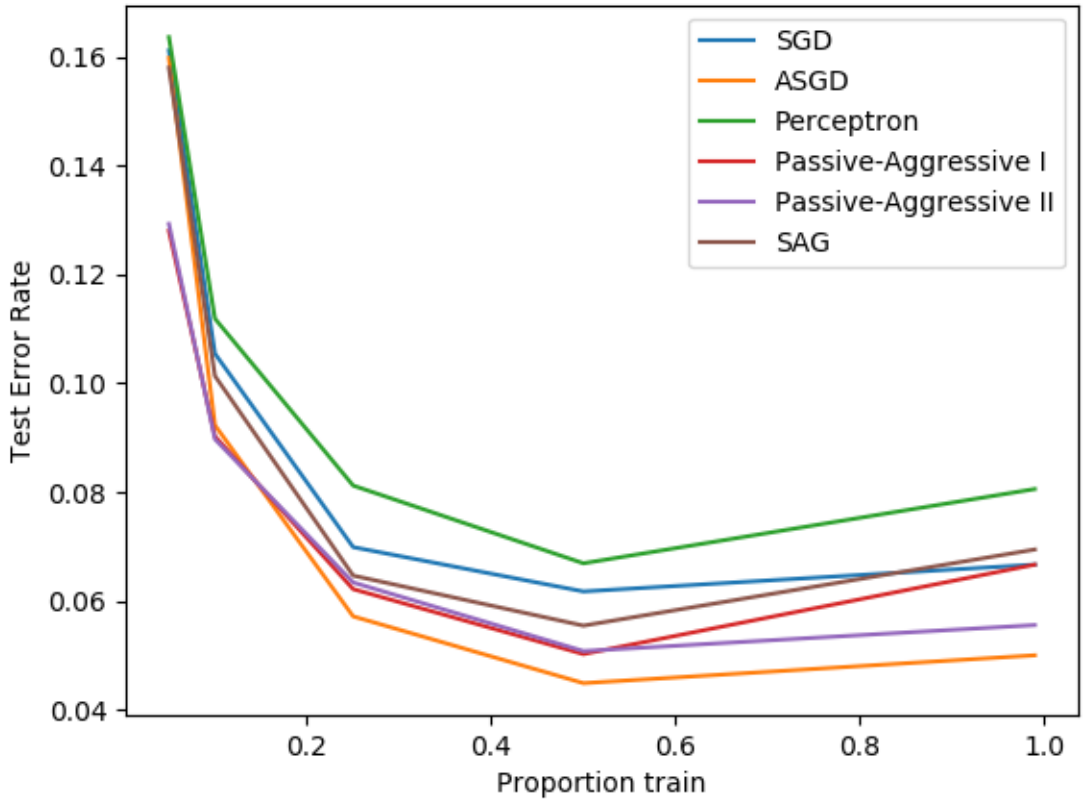
Total running time of the script: (0 minutes 1.579 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.18 Comparing various online solvers

An example showing how different online solvers perform on the hand-written digits dataset.



Out:

```
training SGD
training ASGD
training Perceptron
training Passive-Aggressive I
training Passive-Aggressive II
training SAG
```

```
# Author: Rob Zinkov <rob at zinkov dot com>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

from sklearn.model_selection import train_test_split
from sklearn.linear_model import SGDClassifier, Perceptron
from sklearn.linear_model import PassiveAggressiveClassifier
from sklearn.linear_model import LogisticRegression

heldout = [0.95, 0.90, 0.75, 0.50, 0.01]
rounds = 20
X, y = datasets.load_digits(return_X_y=True)

classifiers = [
    ("SGD", SGDClassifier(max_iter=100)),
    ("ASGD", SGDClassifier(average=True)),
    ("Perceptron", Perceptron()),
    ("Passive-Aggressive I", PassiveAggressiveClassifier(loss='hinge',
                                                         C=1.0, tol=1e-4)),
    ("Passive-Aggressive II", PassiveAggressiveClassifier(loss='squared_hinge',
                                                          C=1.0, tol=1e-4)),
    ("SAG", LogisticRegression(solver='sag', tol=1e-1, C=1.e4 / X.shape[0]))
]

xx = 1. - np.array(heldout)

for name, clf in classifiers:
    print("training %s" % name)
    rng = np.random.RandomState(42)
    yy = []
    for i in heldout:
        yy_ = []
        for r in range(rounds):
            X_train, X_test, y_train, y_test = \
                train_test_split(X, y, test_size=i, random_state=rng)
            clf.fit(X_train, y_train)
            y_pred = clf.predict(X_test)
            yy_.append(1 - np.mean(y_pred == y_test))
        yy.append(np.mean(yy_))
    plt.plot(xx, yy, label=name)

plt.legend(loc="upper right")
plt.xlabel("Proportion train")
plt.ylabel("Test Error Rate")
plt.show()
```

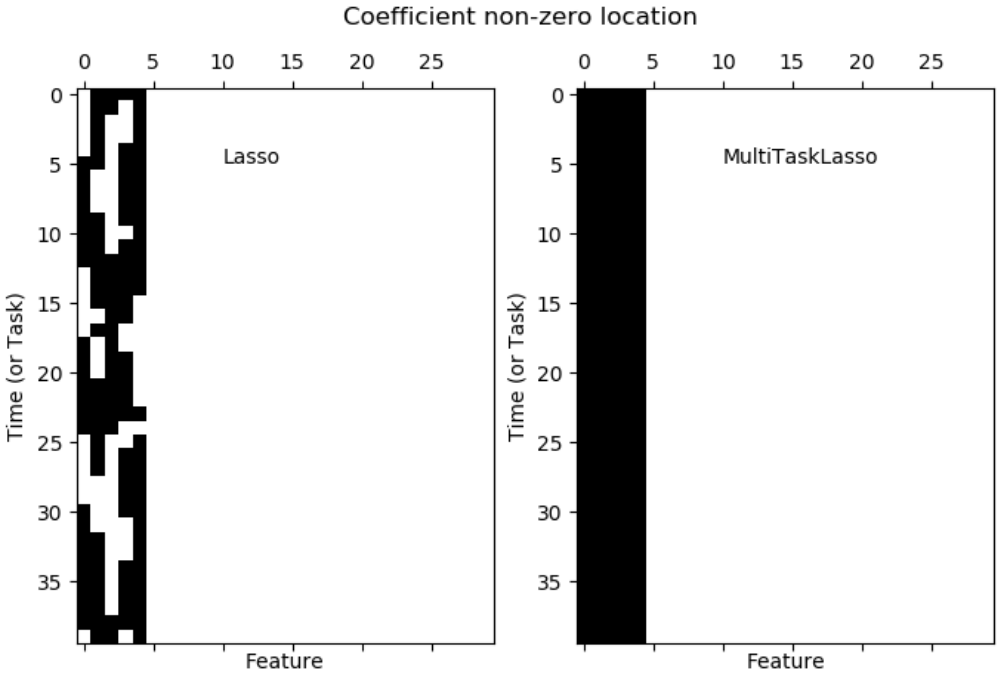
Total running time of the script: (0 minutes 17.905 seconds)

Estimated memory usage: 8 MB

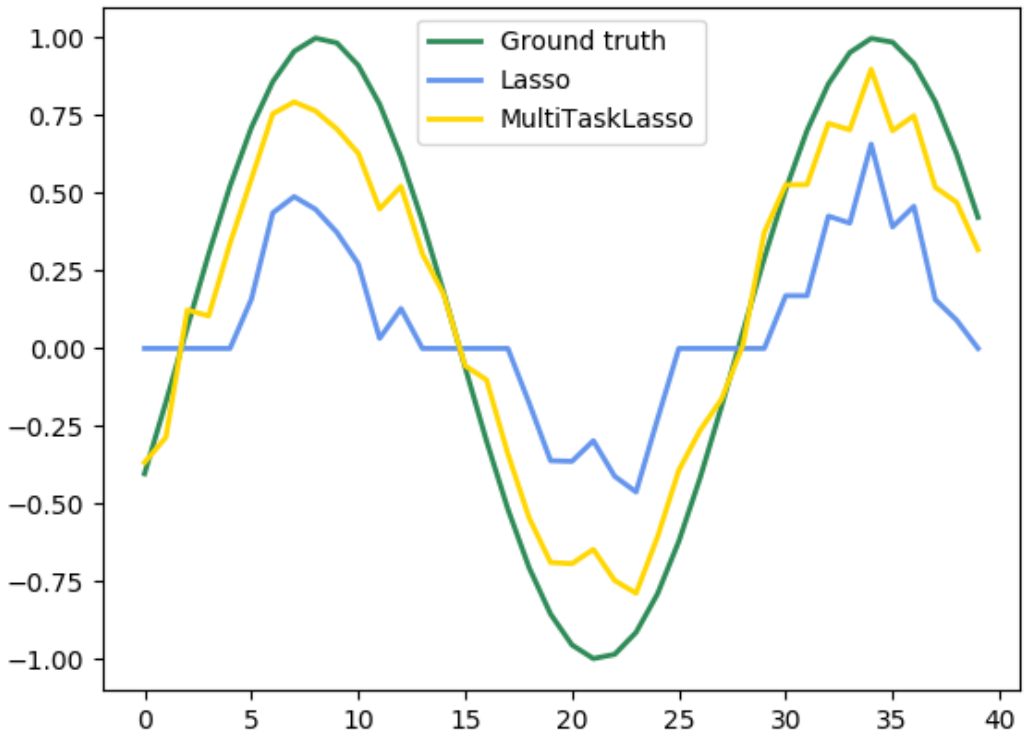
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.19 Joint feature selection with multi-task Lasso

The multi-task lasso allows to fit multiple regression problems jointly enforcing the selected features to be the same across tasks. This example simulates sequential measurements, each task is a time instant, and the relevant features vary in amplitude over time while being the same. The multi-task lasso imposes that features that are selected at one time point are select for all time point. This makes feature selection by the Lasso more stable.



-



-

```
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

import matplotlib.pyplot as plt
import numpy as np

from sklearn.linear_model import MultiTaskLasso, Lasso

rng = np.random.RandomState(42)

# Generate some 2D coefficients with sine waves with random frequency and phase
n_samples, n_features, n_tasks = 100, 30, 40
n_relevant_features = 5
coef = np.zeros((n_tasks, n_features))
times = np.linspace(0, 2 * np.pi, n_tasks)
for k in range(n_relevant_features):
    coef[:, k] = np.sin((1. + rng.randn(1)) * times + 3 * rng.randn(1))

X = rng.randn(n_samples, n_features)
Y = np.dot(X, coef.T) + rng.randn(n_samples, n_tasks)

coef_lasso_ = np.array([Lasso(alpha=0.5).fit(X, y).coef_ for y in Y.T])
coef_multi_task_lasso_ = MultiTaskLasso(alpha=1.).fit(X, Y).coef_

# ##############################################################################
```

(continues on next page)

(continued from previous page)

```
# Plot support and time series
fig = plt.figure(figsize=(8, 5))
plt.subplot(1, 2, 1)
plt.spy(coef_lasso_)
plt.xlabel('Feature')
plt.ylabel('Time (or Task)')
plt.text(10, 5, 'Lasso')
plt.subplot(1, 2, 2)
plt.spy(coef_multi_task_lasso_)
plt.xlabel('Feature')
plt.ylabel('Time (or Task)')
plt.text(10, 5, 'MultiTaskLasso')
fig.suptitle('Coefficient non-zero location')

feature_to_plot = 0
plt.figure()
lw = 2
plt.plot(coef[:, feature_to_plot], color='seagreen', linewidth=lw,
         label='Ground truth')
plt.plot(coef_lasso_[:, feature_to_plot], color='cornflowerblue', linewidth=lw,
         label='Lasso')
plt.plot(coef_multi_task_lasso_[:, feature_to_plot], color='gold', linewidth=lw,
         label='MultiTaskLasso')
plt.legend(loc='upper center')
plt.axis('tight')
plt.ylim([-1.1, 1.1])
plt.show()
```

Total running time of the script: (0 minutes 0.670 seconds)

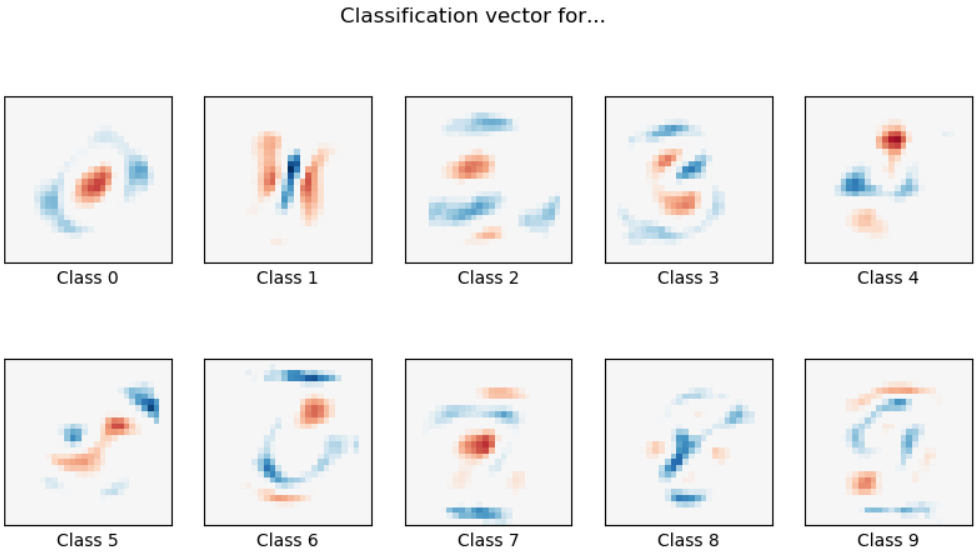
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.20 MNIST classification using multinomial logistic + L1

Here we fit a multinomial logistic regression with L1 penalty on a subset of the MNIST digits classification task. We use the SAGA algorithm for this purpose: this a solver that is fast when the number of samples is significantly larger than the number of features and is able to finely optimize non-smooth objective functions which is the case with the l1-penalty. Test accuracy reaches > 0.8, while weight vectors remains *sparse* and therefore more easily *interpretable*.

Note that this accuracy of this l1-penalized linear model is significantly below what can be reached by an l2-penalized linear model or a non-linear multi-layer perceptron model on this dataset.



Out:

```
Sparsity with L1 penalty: 79.95%
Test score with L1 penalty: 0.8322
Example run in 21.108 s
```

```
import time
import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import fetch_openml
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.utils import check_random_state

print(__doc__)

# Author: Arthur Mensch <arthur.mensch@m4x.org>
# License: BSD 3 clause

# Turn down for faster convergence
t0 = time.time()
train_samples = 5000

# Load data from https://www.openml.org/d/554
X, y = fetch_openml('mnist_784', version=1, return_X_y=True)

random_state = check_random_state(0)
permutation = random_state.permutation(X.shape[0])
X = X[permutation]
```

(continues on next page)

(continued from previous page)

```
y = y[permutation]
X = X.reshape((X.shape[0], -1))

X_train, X_test, y_train, y_test = train_test_split(
    X, y, train_size=train_samples, test_size=10000)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Turn up tolerance for faster convergence
clf = LogisticRegression(
    C=50. / train_samples, penalty='l1', solver='saga', tol=0.1
)
clf.fit(X_train, y_train)
sparsity = np.mean(clf.coef_ == 0) * 100
score = clf.score(X_test, y_test)
# print('Best C % .4f' % clf.C_)
print("Sparsity with L1 penalty: %.2f%%" % sparsity)
print("Test score with L1 penalty: %.4f" % score)

coef = clf.coef_.copy()
plt.figure(figsize=(10, 5))
scale = np.abs(coef).max()
for i in range(10):
    l1_plot = plt.subplot(2, 5, i + 1)
    l1_plot.imshow(coef[i].reshape(28, 28), interpolation='nearest',
                   cmap=plt.cm.RdBu, vmin=-scale, vmax=scale)
    l1_plot.set_xticks(())
    l1_plot.set_yticks(())
    l1_plot.set_xlabel('Class %i' % i)
plt.suptitle('Classification vector for...')

run_time = time.time() - t0
print('Example run in %.3f s' % run_time)
plt.show()
```

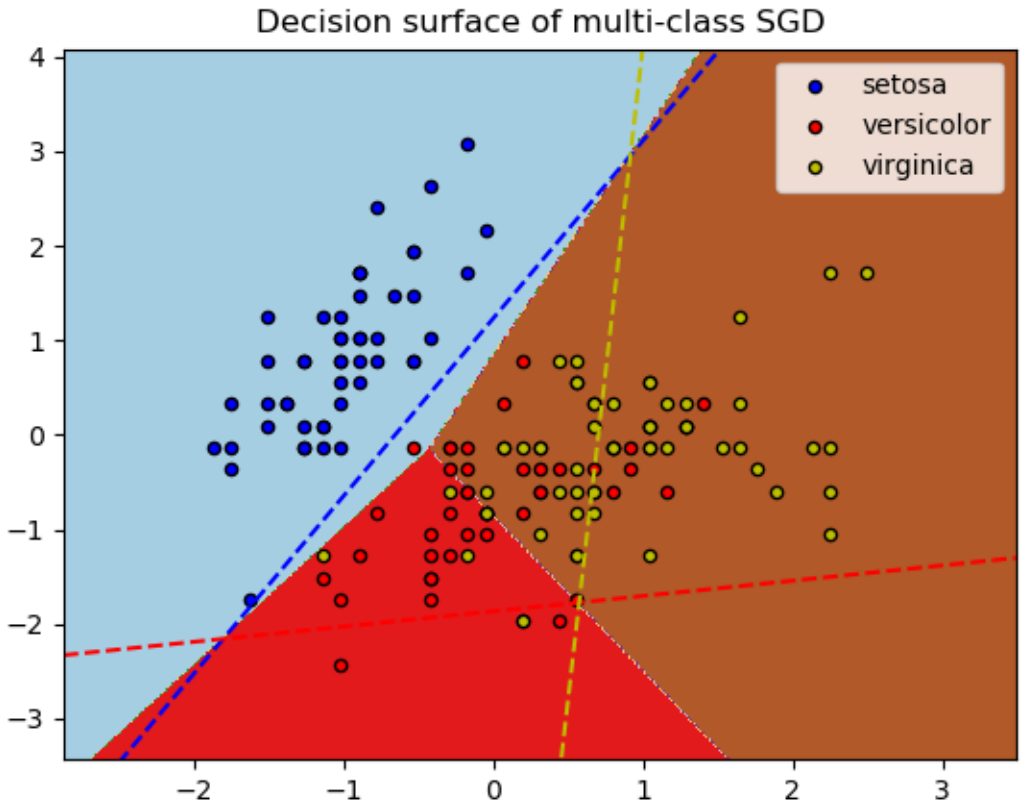
Total running time of the script: (0 minutes 21.470 seconds)

Estimated memory usage: 844 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.21 Plot multi-class SGD on the iris dataset

Plot decision surface of multi-class SGD on iris dataset. The hyperplanes corresponding to the three one-versus-all (OVA) classifiers are represented by the dashed lines.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.linear_model import SGDClassifier

# import some data to play with
iris = datasets.load_iris()

# we only take the first two features. We could
# avoid this ugly slicing by using a two-dim dataset
X = iris.data[:, :2]
y = iris.target
colors = "bry"

# shuffle
idx = np.arange(X.shape[0])
np.random.seed(13)
np.random.shuffle(idx)
X = X[idx]
y = y[idx]

# standardize
mean = X.mean(axis=0)
```

(continues on next page)

(continued from previous page)

```
std = X.std(axis=0)
X = (X - mean) / std

h = .02  # step size in the mesh

clf = SGDClassifier(alpha=0.001, max_iter=100).fit(X, y)

# create a mesh to plot in
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
# Put the result into a color plot
Z = Z.reshape(xx.shape)
cs = plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
plt.axis('tight')

# Plot also the training points
for i, color in zip(clf.classes_, colors):
    idx = np.where(y == i)
    plt.scatter(X[idx, 0], X[idx, 1], c=color, label=iris.target_names[i],
                cmap=plt.cm.Paired, edgecolor='black', s=20)
plt.title("Decision surface of multi-class SGD")
plt.axis('tight')

# Plot the three one-against-all classifiers
xmin, xmax = plt.xlim()
ymin, ymax = plt.ylim()
coef = clf.coef_
intercept = clf.intercept_


def plot_hyperplane(c, color):
    def line(x0):
        return (-(x0 * coef[c, 0]) - intercept[c]) / coef[c, 1]

    plt.plot([xmin, xmax], [line(xmin), line(xmax)],
             ls="--", color=color)


for i, color in zip(clf.classes_, colors):
    plot_hyperplane(i, color)
plt.legend()
plt.show()
```

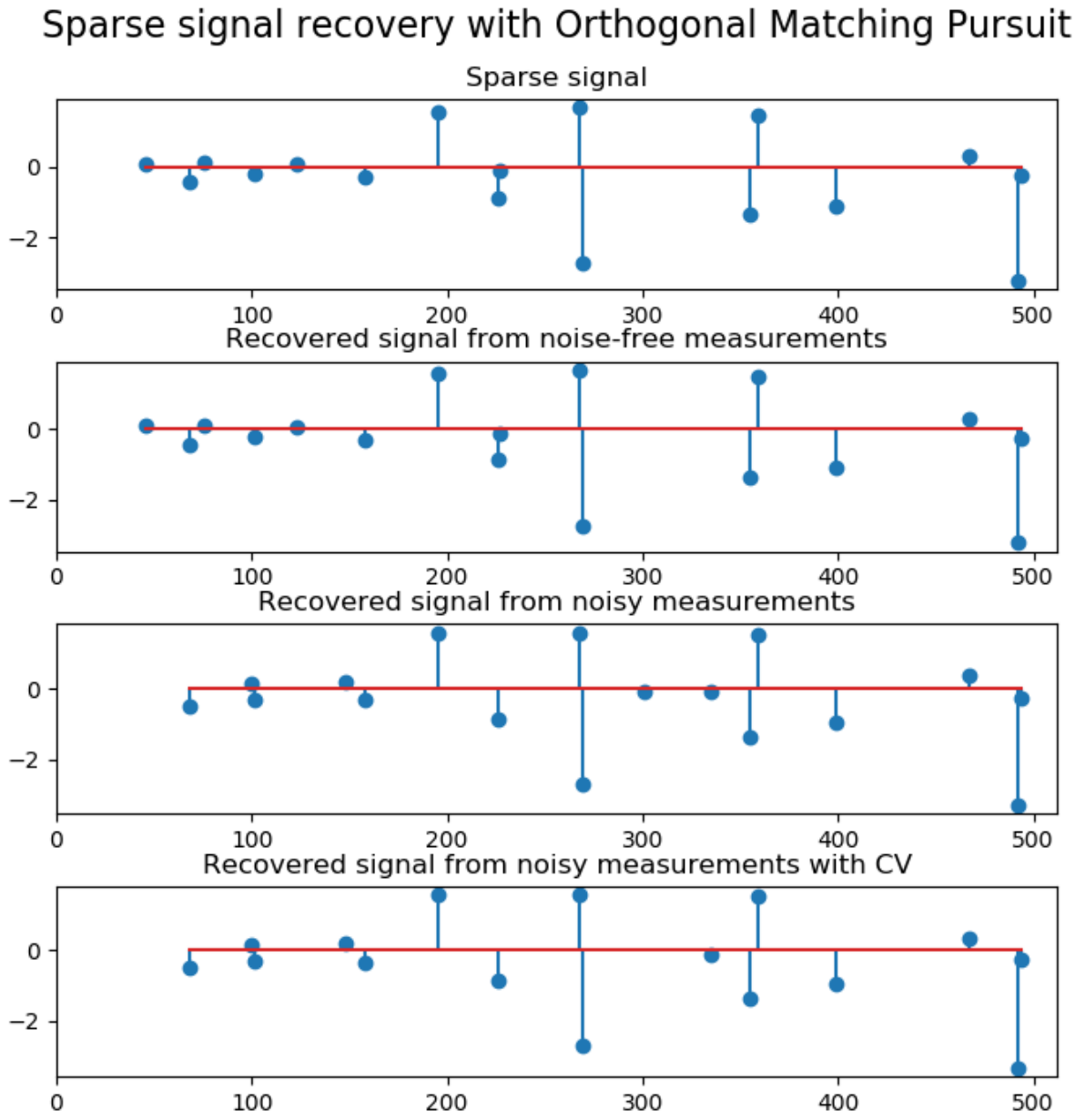
Total running time of the script: (0 minutes 0.585 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.22 Orthogonal Matching Pursuit

Using orthogonal matching pursuit for recovering a sparse signal from a noisy measurement encoded with a dictionary




```
print(__doc__)

import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import OrthogonalMatchingPursuit
from sklearn.linear_model import OrthogonalMatchingPursuitCV
from sklearn.datasets import make_sparse_coded_signal

n_components, n_features = 512, 100
n_nonzero_coefs = 17
```

(continues on next page)

(continued from previous page)

```
# generate the data

# y = Xw
# |x|_0 = n_nonzero_coefs

y, X, w = make_sparse_coded_signal(n_samples=1,
                                   n_components=n_components,
                                   n_features=n_features,
                                   n_nonzero_coefs=n_nonzero_coefs,
                                   random_state=0)

idx, = w.nonzero()

# distort the clean signal
y_noisy = y + 0.05 * np.random.randn(len(y))

# plot the sparse signal
plt.figure(figsize=(7, 7))
plt.subplot(4, 1, 1)
plt.xlim(0, 512)
plt.title("Sparse signal")
plt.stem(idx, w[idx], use_line_collection=True)

# plot the noise-free reconstruction
omp = OrthogonalMatchingPursuit(n_nonzero_coefs=n_nonzero_coefs)
omp.fit(X, y)
coef = omp.coef_
idx_r, = coef.nonzero()
plt.subplot(4, 1, 2)
plt.xlim(0, 512)
plt.title("Recovered signal from noise-free measurements")
plt.stem(idx_r, coef[idx_r], use_line_collection=True)

# plot the noisy reconstruction
omp.fit(X, y_noisy)
coef = omp.coef_
idx_r, = coef.nonzero()
plt.subplot(4, 1, 3)
plt.xlim(0, 512)
plt.title("Recovered signal from noisy measurements")
plt.stem(idx_r, coef[idx_r], use_line_collection=True)

# plot the noisy reconstruction with number of non-zeros set by CV
omp_cv = OrthogonalMatchingPursuitCV()
omp_cv.fit(X, y_noisy)
coef = omp_cv.coef_
idx_r, = coef.nonzero()
plt.subplot(4, 1, 4)
plt.xlim(0, 512)
plt.title("Recovered signal from noisy measurements with CV")
plt.stem(idx_r, coef[idx_r], use_line_collection=True)

plt.subplots_adjust(0.06, 0.04, 0.94, 0.90, 0.20, 0.38)
plt.suptitle('Sparse signal recovery with Orthogonal Matching Pursuit',
             fontsize=16)
plt.show()
```

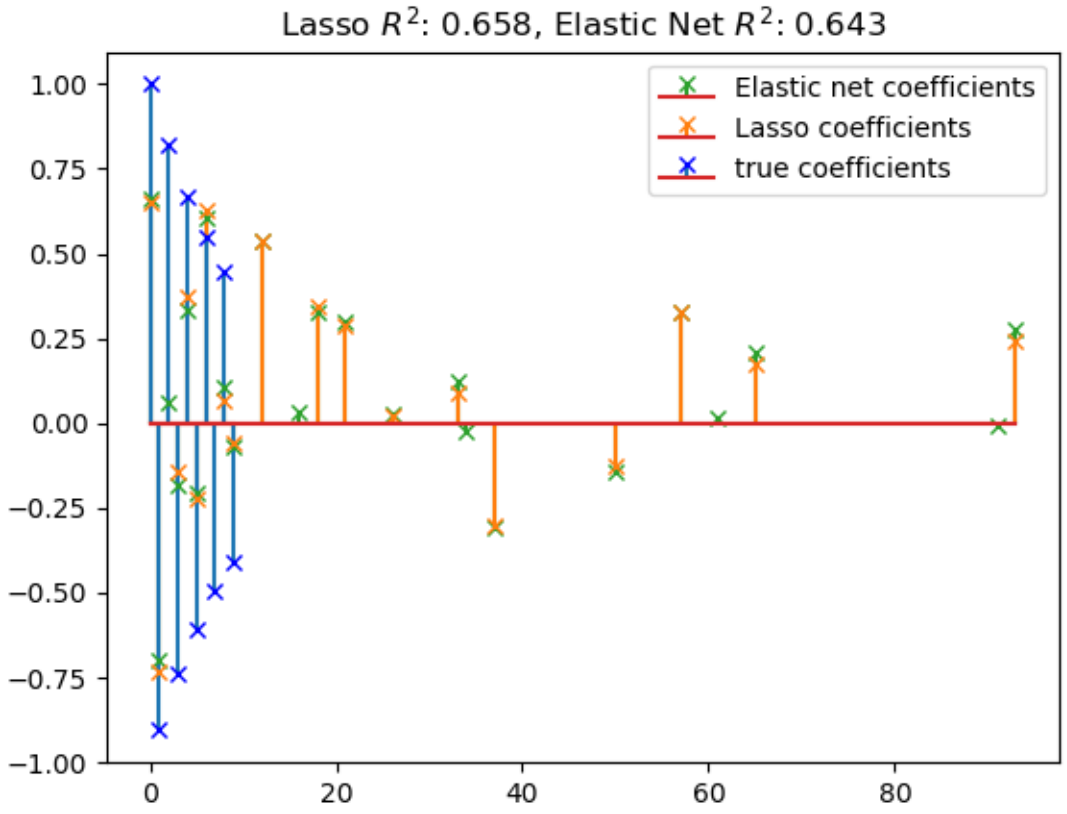
Total running time of the script: (0 minutes 0.686 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.23 Lasso and Elastic Net for Sparse Signals

Estimates Lasso and Elastic-Net regression models on a manually generated sparse signal corrupted with an additive noise. Estimated coefficients are compared with the ground-truth.



Out:

```
Lasso(alpha=0.1)
r^2 on test data : 0.658064
ElasticNet(alpha=0.1, l1_ratio=0.7)
r^2 on test data : 0.642515
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.metrics import r2_score

# ##############################################################################
# Generate some sparse data to play with
np.random.seed(42)

n_samples, n_features = 50, 100
X = np.random.randn(n_samples, n_features)

# Decreasing coef w. alternated signs for visualization
idx = np.arange(n_features)
coef = (-1) ** idx * np.exp(-idx / 10)
coef[10:] = 0  # sparsify coef
y = np.dot(X, coef)

# Add noise
y += 0.01 * np.random.normal(size=n_samples)

# Split data in train set and test set
n_samples = X.shape[0]
X_train, y_train = X[:n_samples // 2], y[:n_samples // 2]
X_test, y_test = X[n_samples // 2:], y[n_samples // 2:]

# ##############################################################################
# Lasso
from sklearn.linear_model import Lasso

alpha = 0.1
lasso = Lasso(alpha=alpha)

y_pred_lasso = lasso.fit(X_train, y_train).predict(X_test)
r2_score_lasso = r2_score(y_test, y_pred_lasso)
print(lasso)
print("r^2 on test data : %f" % r2_score_lasso)

# ##############################################################################
# ElasticNet
from sklearn.linear_model import ElasticNet

enet = ElasticNet(alpha=alpha, l1_ratio=0.7)

y_pred_enet = enet.fit(X_train, y_train).predict(X_test)
r2_score_enet = r2_score(y_test, y_pred_enet)
print(enet)
print("r^2 on test data : %f" % r2_score_enet)

m, s, _ = plt.stem(np.where(enet.coef_)[0], enet.coef_[enet.coef_ != 0],
                   markerfmt='x', label='Elastic net coefficients',
                   use_line_collection=True)
plt.setp([m, s], color="#2ca02c")
m, s, _ = plt.stem(np.where(lasso.coef_)[0], lasso.coef_[lasso.coef_ != 0],
                   markerfmt='x', label='Lasso coefficients',
```

(continues on next page)

(continued from previous page)

```
                    use_line_collection=True)
plt.setp([m, s], color='#ff7f0e')
plt.stem(np.where(coef)[0], coef[coef != 0], label='true coefficients',
         markerfmt='bx', use_line_collection=True)

plt.legend(loc='best')
plt.title("Lasso $R^2$: %.3f, Elastic Net $R^2$: %.3f"
          % (r2_score_lasso, r2_score_enet))
plt.show()
```

Total running time of the script: (0 minutes 0.552 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.24 Curve Fitting with Bayesian Ridge Regression

Computes a Bayesian Ridge Regression of Sinusoids.

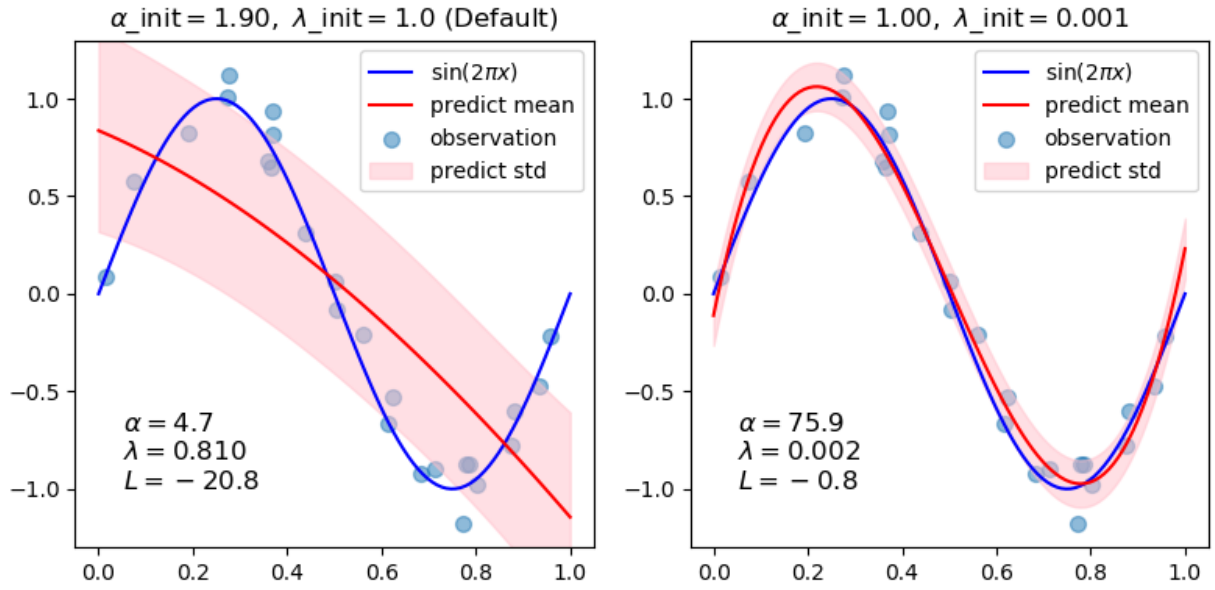
See *Bayesian Ridge Regression* for more information on the regressor.

In general, when fitting a curve with a polynomial by Bayesian ridge regression, the selection of initial values of the regularization parameters (alpha, lambda) may be important. This is because the regularization parameters are determined by an iterative procedure that depends on initial values.

In this example, the sinusoid is approximated by a polynomial using different pairs of initial values.

When starting from the default values (alpha_init = 1.90, lambda_init = 1.), the bias of the resulting curve is large, and the variance is small. So, lambda_init should be relatively small (1.e-3) so as to reduce the bias.

Also, by evaluating log marginal likelihood (L) of these models, we can determine which one is better. It can be concluded that the model with larger L is more likely.



```
print(__doc__)

# Author: Yoshihiro Uchida <nimbus1after2a1sun7shower@gmail.com>

import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import BayesianRidge


def func(x): return np.sin(2*np.pi*x)


# #############################################################################
# Generate sinusoidal data with noise
size = 25
rng = np.random.RandomState(1234)
x_train = rng.uniform(0., 1., size)
y_train = func(x_train) + rng.normal(scale=0.1, size=size)
x_test = np.linspace(0., 1., 100)


# #############################################################################
# Fit by cubic polynomial
n_order = 3
X_train = np.vander(x_train, n_order + 1, increasing=True)
X_test = np.vander(x_test, n_order + 1, increasing=True)

# #############################################################################
# Plot the true and predicted curves with log marginal likelihood (L)
reg = BayesianRidge(tol=1e-6, fit_intercept=False, compute_score=True)
fig, axes = plt.subplots(1, 2, figsize=(8, 4))
for i, ax in enumerate(axes):
    # Bayesian ridge regression with different initial value pairs
    if i == 0:
        init = [1 / np.var(y_train), 1.]  # Default values
    elif i == 1:
        init = [1., 1e-3]
        reg.set_params(alpha_init=init[0], lambda_init=init[1])
    reg.fit(X_train, y_train)
    ymean, ystd = reg.predict(X_test, return_std=True)

    ax.plot(x_test, func(x_test), color="blue", label="sin($2\\pi x$)")
    ax.scatter(x_train, y_train, s=50, alpha=0.5, label="observation")
    ax.plot(x_test, ymean, color="red", label="predict mean")
    ax.fill_between(x_test, ymean-ystd, ymean+ystd,
                    color="pink", alpha=0.5, label="predict std")
    ax.set_ylim(-1.3, 1.3)
    ax.legend()
    title = "$\\alpha$_init$={:.2f},\\ \\lambda$_init$={}$".format(
            init[0], init[1])
    if i == 0:
        title += " (Default)"
    ax.set_title(title, fontsize=12)
    text = "$\\alpha={:.1f}$\n$\\lambda={:.3f}$\n$L={:.1f}$".format(
           reg.alpha_, reg.lambda_, reg.scores_[-1])
    ax.text(0.05, -1.0, text, fontsize=12)
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.611 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.25 Theil-Sen Regression

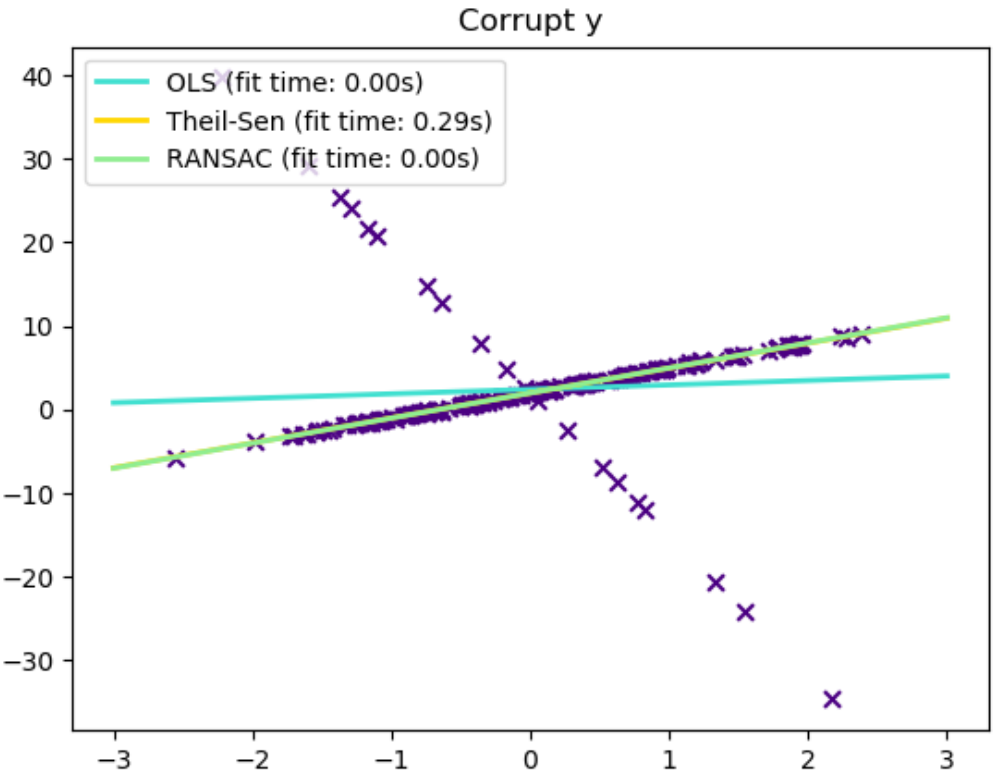
Computes a Theil-Sen Regression on a synthetic dataset.

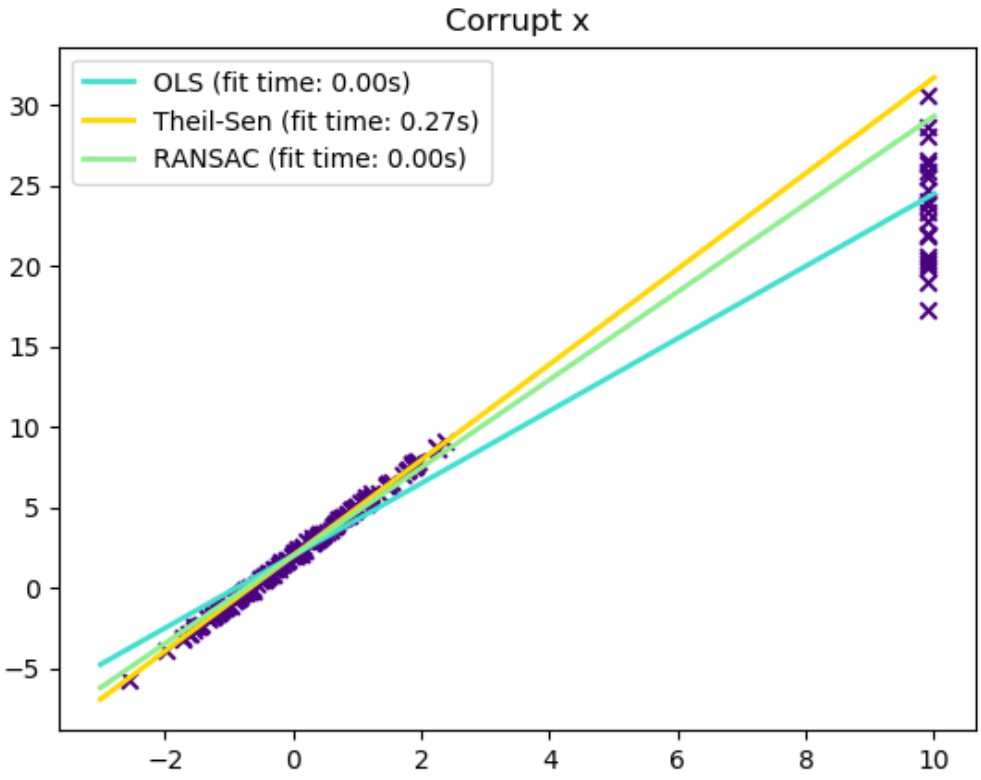
See *Theil-Sen estimator: generalized-median-based estimator* for more information on the regressor.

Compared to the OLS (ordinary least squares) estimator, the Theil-Sen estimator is robust against outliers. It has a breakdown point of about 29.3% in case of a simple linear regression which means that it can tolerate arbitrary corrupted data (outliers) of up to 29.3% in the two-dimensional case.

The estimation of the model is done by calculating the slopes and intercepts of a subpopulation of all possible combinations of p subsample points. If an intercept is fitted, p must be greater than or equal to n_features + 1. The final slope and intercept is then defined as the spatial median of these slopes and intercepts.

In certain cases Theil-Sen performs better than *RANSAC* which is also a robust method. This is illustrated in the second example below where outliers with respect to the x-axis perturb RANSAC. Tuning the `residual_threshold` parameter of RANSAC remedies this but in general a priori knowledge about the data and the nature of the outliers is needed. Due to the computational complexity of Theil-Sen it is recommended to use it only for small problems in terms of number of samples and features. For larger problems the `max_subpopulation` parameter restricts the magnitude of all possible combinations of p subsample points to a randomly chosen subset and therefore also limits the runtime. Therefore, Theil-Sen is applicable to larger problems with the drawback of losing some of its mathematical properties since it then works on a random subset.


Corrupt y
OLS (fit time: 0.00s)
Theil-Sen (fit time: 0.29s)
RANSAC (fit time: 0.00s)
40
30
20
10
0
−10
−20
−30
−3
−2
−1
0
1
2
3



-

```
# Author: Florian Wilhelm -- <florian.wilhelm@gmail.com>
# License: BSD 3 clause

import time
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, TheilSenRegressor
from sklearn.linear_model import RANSACRegressor

print(__doc__)

estimators = [('OLS', LinearRegression()),
              ('Theil-Sen', TheilSenRegressor(random_state=42)),
              ('RANSAC', RANSACRegressor(random_state=42)), ]
colors = {'OLS': 'turquoise', 'Theil-Sen': 'gold', 'RANSAC': 'lightgreen'}
lw = 2

# ############################################################################
# Outliers only in the y direction

np.random.seed(0)
n_samples = 200
# Linear model y = 3*x + N(2, 0.1**2)
x = np.random.randn(n_samples)
w = 3.
c = 2.
noise = 0.1 * np.random.randn(n_samples)
```

(continues on next page)

(continued from previous page)

```
y = w * x + c + noise
# 10% outliers
y[-20:] += -20 * x[-20:]
X = x[:, np.newaxis]

plt.scatter(x, y, color='indigo', marker='x', s=40)
line_x = np.array([-3, 3])
for name, estimator in estimators:
    t0 = time.time()
    estimator.fit(X, y)
    elapsed_time = time.time() - t0
    y_pred = estimator.predict(line_x.reshape(2, 1))
    plt.plot(line_x, y_pred, color=colors[name], linewidth=lw,
             label='%s (fit time: %.2fs)' % (name, elapsed_time))

plt.axis('tight')
plt.legend(loc='upper left')
plt.title("Corrupt y")

# ##############################################################################
# Outliers in the X direction

np.random.seed(0)
# Linear model y = 3*x + N(2, 0.1**2)
x = np.random.randn(n_samples)
noise = 0.1 * np.random.randn(n_samples)
y = 3 * x + 2 + noise
# 10% outliers
x[-20:] = 9.9
y[-20:] += 22
X = x[:, np.newaxis]

plt.figure()
plt.scatter(x, y, color='indigo', marker='x', s=40)

line_x = np.array([-3, 10])
for name, estimator in estimators:
    t0 = time.time()
    estimator.fit(X, y)
    elapsed_time = time.time() - t0
    y_pred = estimator.predict(line_x.reshape(2, 1))
    plt.plot(line_x, y_pred, color=colors[name], linewidth=lw,
             label='%s (fit time: %.2fs)' % (name, elapsed_time))

plt.axis('tight')
plt.legend(loc='upper left')
plt.title("Corrupt x")
plt.show()
```

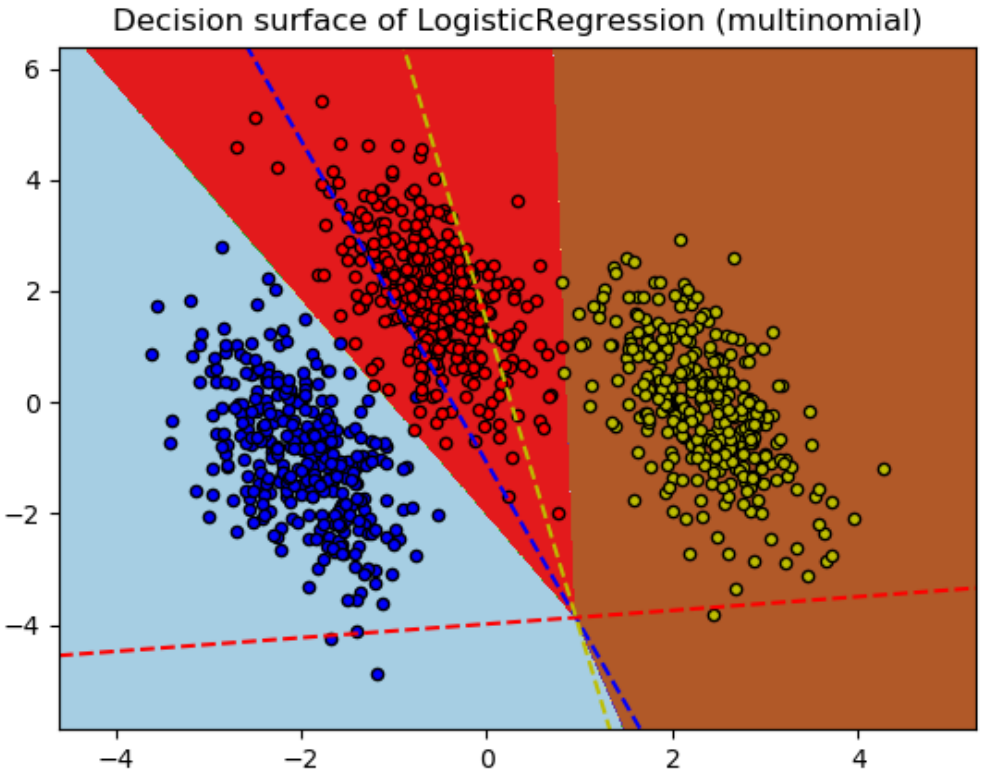
Total running time of the script: (0 minutes 1.024 seconds)

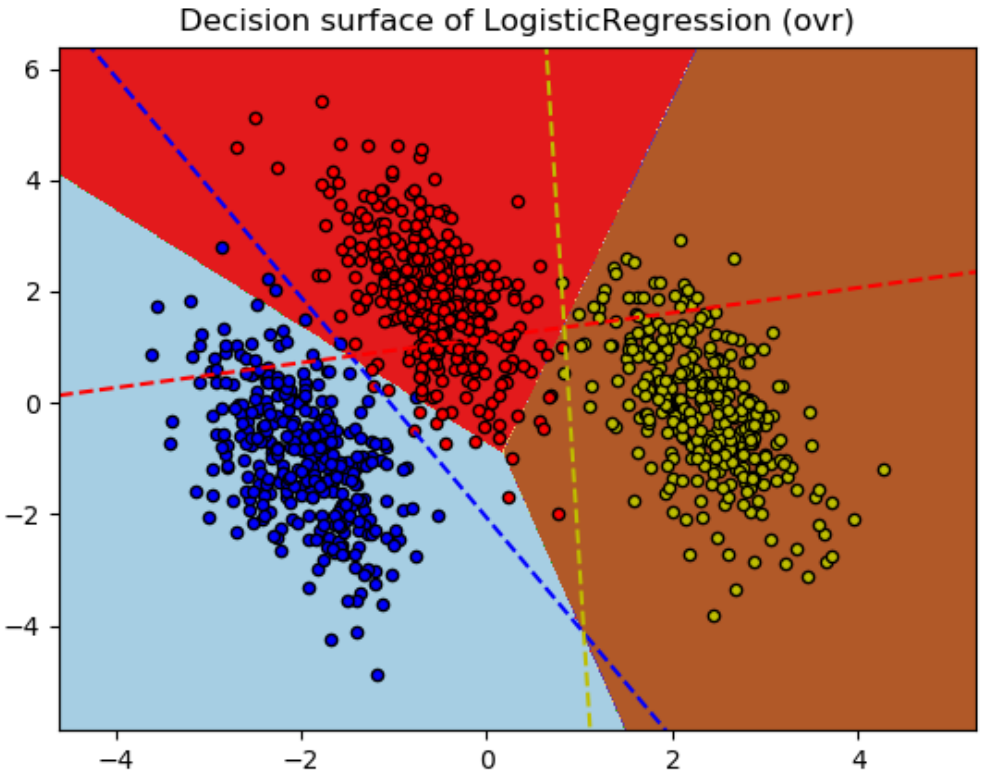
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.26 Plot multinomial and One-vs-Rest Logistic Regression

Plot decision surface of multinomial and One-vs-Rest Logistic Regression. The hyperplanes corresponding to the three One-vs-Rest (OVR) classifiers are represented by the dashed lines.





-

Out:

```
training score : 0.995 (multinomial)
training score : 0.976 (ovr)
```

```
print(__doc__)
# Authors: Tom Dupre la Tour <tom.dupre-la-tour@m4x.org>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.linear_model import LogisticRegression

# make 3-class dataset for classification
centers = [[-5, 0], [0, 1.5], [5, -1]]
X, y = make_blobs(n_samples=1000, centers=centers, random_state=40)
transformation = [[0.4, 0.2], [-0.4, 1.2]]
X = np.dot(X, transformation)

for multi_class in ('multinomial', 'ovr'):
    clf = LogisticRegression(solver='sag', max_iter=100, random_state=42,
```

(continues on next page)

(continued from previous page)

```
                               multi_class=multi_class).fit(X, y)

    # print the training scores
    print("training score : %.3f (%s)" % (clf.score(X, y), multi_class))

    # create a mesh to plot in
    h = .02  # step size in the mesh
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
    plt.title("Decision surface of LogisticRegression (%s)" % multi_class)
    plt.axis('tight')

    # Plot also the training points
    colors = "bry"
    for i, color in zip(clf.classes_, colors):
        idx = np.where(y == i)
        plt.scatter(X[idx, 0], X[idx, 1], c=color, cmap=plt.cm.Paired,
                    edgecolor='black', s=20)

    # Plot the three one-against-all classifiers
    xmin, xmax = plt.xlim()
    ymin, ymax = plt.ylim()
    coef = clf.coef_
    intercept = clf.intercept_

    def plot_hyperplane(c, color):
        def line(x0):
            return (-(x0 * coef[c, 0]) - intercept[c]) / coef[c, 1]
        plt.plot([xmin, xmax], [line(xmin), line(xmax)],
                 ls="--", color=color)

    for i, color in zip(clf.classes_, colors):
        plot_hyperplane(i, color)

plt.show()
```

Total running time of the script: (0 minutes 0.768 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.27 Robust linear estimator fitting

Here a sine function is fit with a polynomial of order 3, for values close to zero.

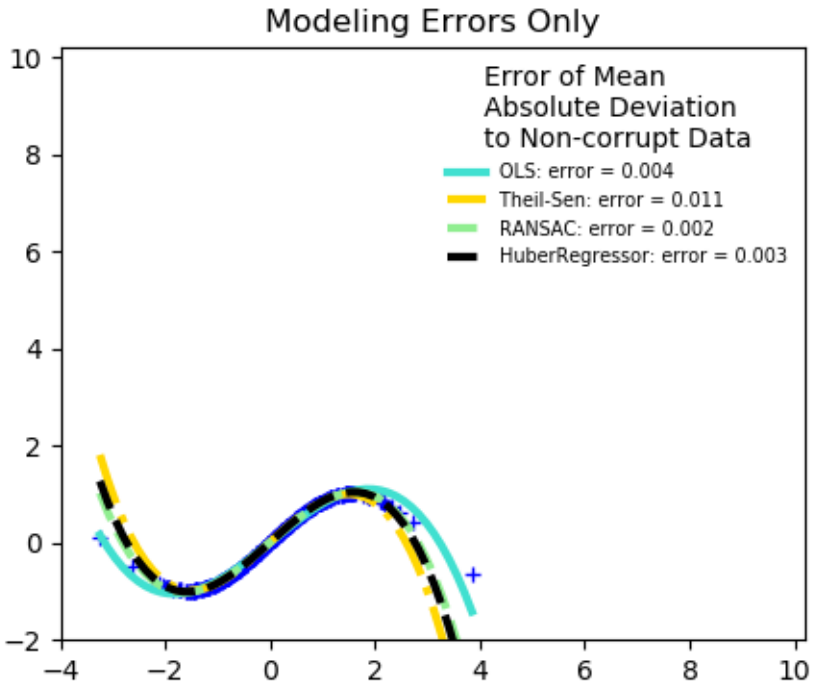
Robust fitting is demoed in different situations:

- No measurement errors, only modelling errors (fitting a sine with a polynomial)
- Measurement errors in X
- Measurement errors in y

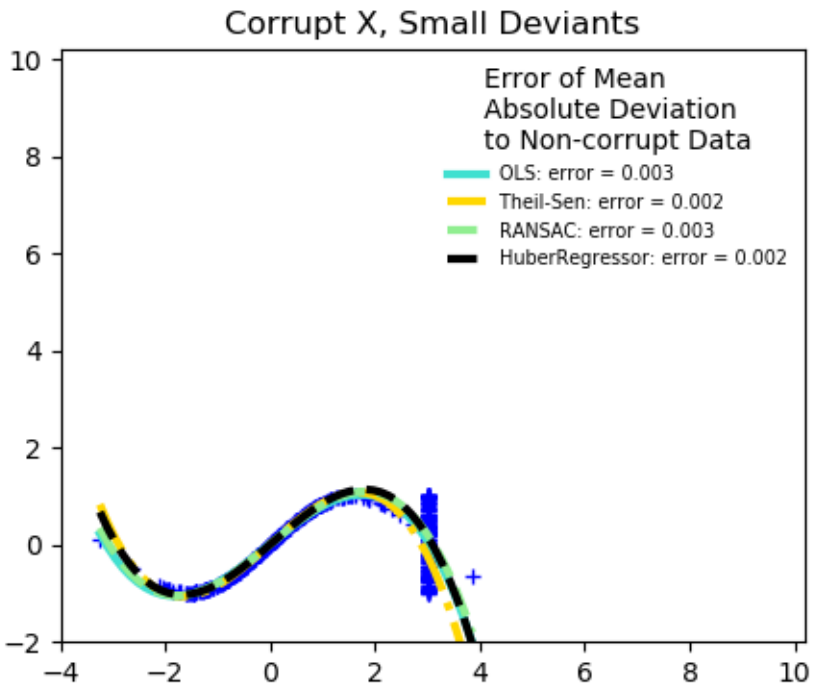
The median absolute deviation to non corrupt new data is used to judge the quality of the prediction.

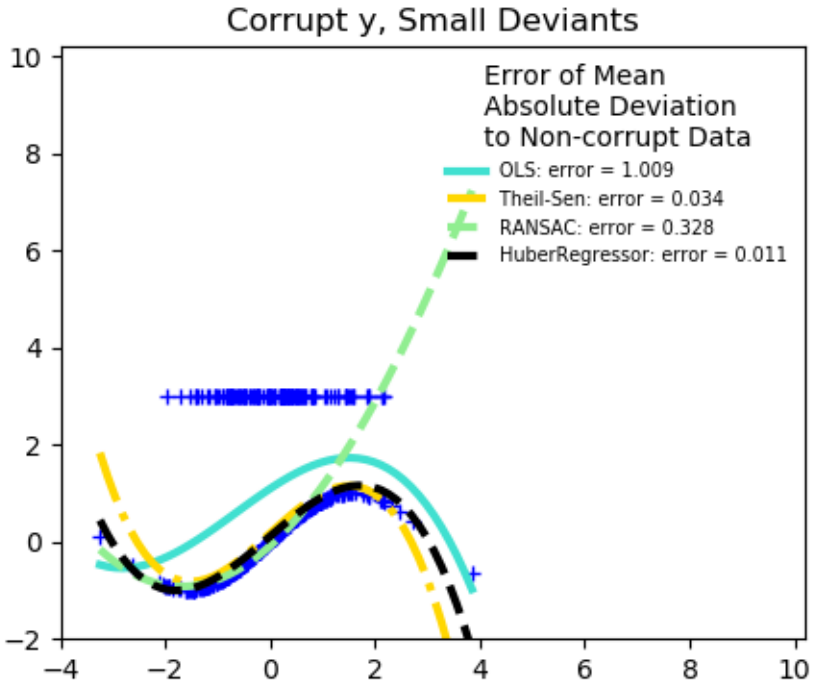
What we can see that:

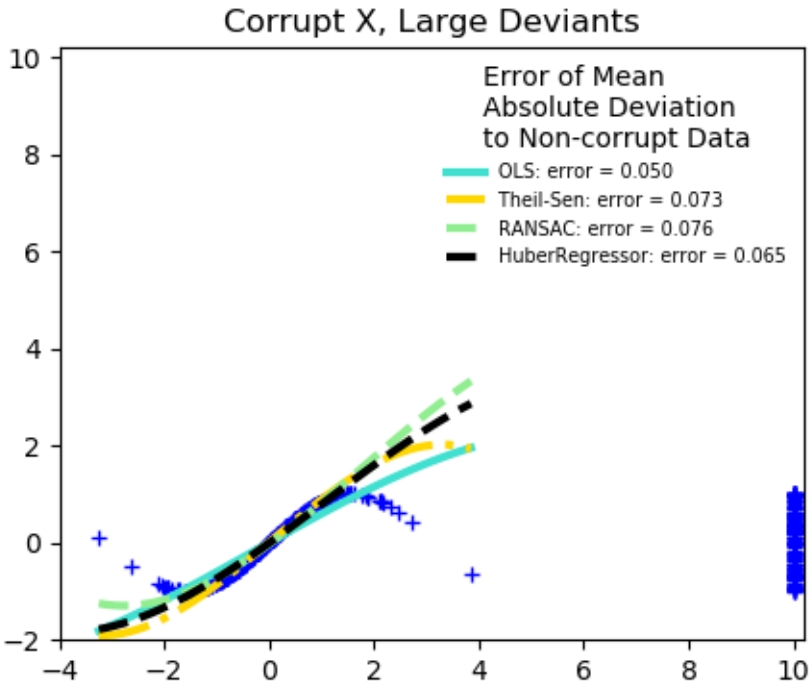
- RANSAC is good for strong outliers in the y direction
- TheilSen is good for small outliers, both in direction X and y, but has a break point above which it performs worse than OLS.
- The scores of HuberRegressor may not be compared directly to both TheilSen and RANSAC because it does not attempt to completely filter the outliers but lessen their effect.



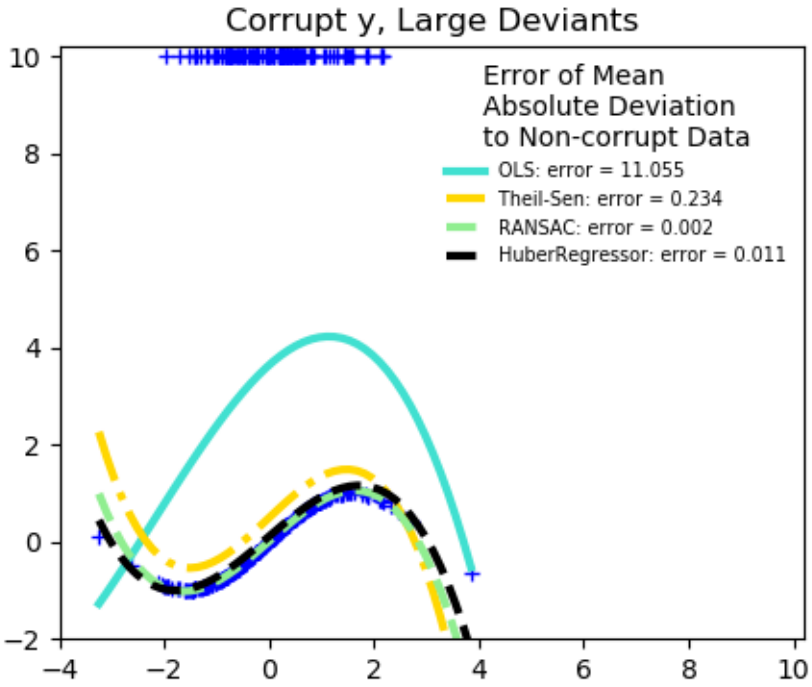
-

- 
Corrupt X, Small Deviants
Error of Mean
Absolute Deviation
to Non-corrupt Data
OLS: error = 0.003
Theil-Sen: error = 0.002
RANSAC: error = 0.003
HuberRegressor: error = 0.002

- 
Corrupt y, Small Deviants
Error of Mean
Absolute Deviation
to Non-corrupt Data
OLS: error = 1.009
Theil-Sen: error = 0.034
RANSAC: error = 0.328
HuberRegressor: error = 0.011



-



-

```
from matplotlib import pyplot as plt
import numpy as np

from sklearn.linear_model import (
```

(continues on next page)

(continued from previous page)

```
    LinearRegression, TheilSenRegressor, RANSACRegressor, HuberRegressor)
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

np.random.seed(42)

X = np.random.normal(size=400)
y = np.sin(X)
# Make sure that it X is 2D
X = X[:, np.newaxis]

X_test = np.random.normal(size=200)
y_test = np.sin(X_test)
X_test = X_test[:, np.newaxis]

y_errors = y.copy()
y_errors[::3] = 3

X_errors = X.copy()
X_errors[::3] = 3

y_errors_large = y.copy()
y_errors_large[::3] = 10

X_errors_large = X.copy()
X_errors_large[::3] = 10

estimators = [('OLS', LinearRegression()),
              ('Theil-Sen', TheilSenRegressor(random_state=42)),
              ('RANSAC', RANSACRegressor(random_state=42)),
              ('HuberRegressor', HuberRegressor())]
colors = {'OLS': 'turquoise', 'Theil-Sen': 'gold', 'RANSAC': 'lightgreen',
↪'HuberRegressor': 'black'}
linestyle = {'OLS': '-', 'Theil-Sen': '-.', 'RANSAC': '--', 'HuberRegressor': '--'}
lw = 3

x_plot = np.linspace(X.min(), X.max())
for title, this_X, this_y in [
        ('Modeling Errors Only', X, y),
        ('Corrupt X, Small Deviants', X_errors, y),
        ('Corrupt y, Small Deviants', X, y_errors),
        ('Corrupt X, Large Deviants', X_errors_large, y),
        ('Corrupt y, Large Deviants', X, y_errors_large)]:
    plt.figure(figsize=(5, 4))
    plt.plot(this_X[:, 0], this_y, 'b+')

    for name, estimator in estimators:
        model = make_pipeline(PolynomialFeatures(3), estimator)
        model.fit(this_X, this_y)
        mse = mean_squared_error(model.predict(X_test), y_test)
        y_plot = model.predict(x_plot[:, np.newaxis])
        plt.plot(x_plot, y_plot, color=colors[name], linestyle=linestyle[name],
                 linewidth=lw, label='%s: error = %.3f' % (name, mse))

    legend_title = 'Error of Mean\nAbsolute Deviation\nto Non-corrupt Data'
    legend = plt.legend(loc='upper right', frameon=False, title=legend_title,
```

(continues on next page)

(continued from previous page)

```
                        prop=dict(size='x-small'))
    plt.xlim(-4, 10.2)
    plt.ylim(-2, 10.2)
    plt.title(title)
plt.show()
```

Total running time of the script: (0 minutes 2.699 seconds)

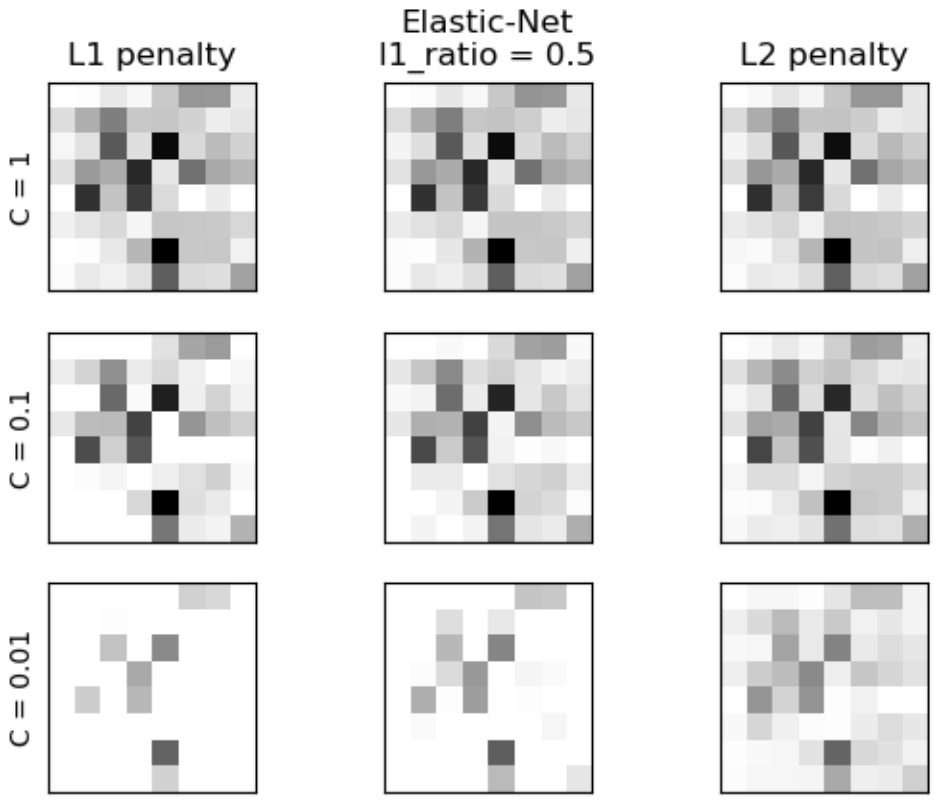
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.28 L1 Penalty and Sparsity in Logistic Regression

Comparison of the sparsity (percentage of zero coefficients) of solutions when L1, L2 and Elastic-Net penalty are used for different values of C. We can see that large values of C give more freedom to the model. Conversely, smaller values of C constrain the model more. In the L1 penalty case, this leads to sparser solutions. As expected, the Elastic-Net penalty sparsity is between that of L1 and L2.

We classify 8x8 images of digits into two classes: 0-4 against 5-9. The visualization shows coefficients of the models for varying C.



Out:

```
C=1.00
Sparsity with L1 penalty:                6.25%
Sparsity with Elastic-Net penalty:       4.69%
Sparsity with L2 penalty:                4.69%
Score with L1 penalty:                   0.90
Score with Elastic-Net penalty:          0.90
Score with L2 penalty:                   0.90
C=0.10
Sparsity with L1 penalty:                29.69%
Sparsity with Elastic-Net penalty:       12.50%
Sparsity with L2 penalty:                4.69%
Score with L1 penalty:                   0.90
Score with Elastic-Net penalty:          0.90
Score with L2 penalty:                   0.90
C=0.01
Sparsity with L1 penalty:                84.38%
Sparsity with Elastic-Net penalty:       68.75%
Sparsity with L2 penalty:                4.69%
Score with L1 penalty:                   0.86
Score with Elastic-Net penalty:          0.88
Score with L2 penalty:                   0.89
```

```
print(__doc__)

# Authors: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#          Mathieu Blondel <mathieu@mblondel.org>
#          Andreas Mueller <amueller@ais.uni-bonn.de>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

X, y = datasets.load_digits(return_X_y=True)

X = StandardScaler().fit_transform(X)

# classify small against large digits
y = (y > 4).astype(np.int)

l1_ratio = 0.5  # L1 weight in the Elastic-Net regularization

fig, axes = plt.subplots(3, 3)

# Set regularization parameter
for i, (C, axes_row) in enumerate(zip((1, 0.1, 0.01), axes)):
    # turn down tolerance for short training time
    clf_l1_LR = LogisticRegression(C=C, penalty='l1', tol=0.01, solver='saga')
    clf_l2_LR = LogisticRegression(C=C, penalty='l2', tol=0.01, solver='saga')
```

(continues on next page)

(continued from previous page)

```
    clf_en_LR = LogisticRegression(C=C, penalty='elasticnet', solver='saga',
                                   l1_ratio=l1_ratio, tol=0.01)
    clf_l1_LR.fit(X, y)
    clf_l2_LR.fit(X, y)
    clf_en_LR.fit(X, y)

    coef_l1_LR = clf_l1_LR.coef_.ravel()
    coef_l2_LR = clf_l2_LR.coef_.ravel()
    coef_en_LR = clf_en_LR.coef_.ravel()

    # coef_l1_LR contains zeros due to the
    # L1 sparsity inducing norm

    sparsity_l1_LR = np.mean(coef_l1_LR == 0) * 100
    sparsity_l2_LR = np.mean(coef_l2_LR == 0) * 100
    sparsity_en_LR = np.mean(coef_en_LR == 0) * 100

    print("C=%.2f" % C)
    print("{:<40} {:.2f}%".format("Sparsity with L1 penalty:", sparsity_l1_LR))
    print("{:<40} {:.2f}%".format("Sparsity with Elastic-Net penalty:",
                                  sparsity_en_LR))
    print("{:<40} {:.2f}%".format("Sparsity with L2 penalty:", sparsity_l2_LR))
    print("{:<40} {:.2f}".format("Score with L1 penalty:",
                                 clf_l1_LR.score(X, y)))
    print("{:<40} {:.2f}".format("Score with Elastic-Net penalty:",
                                 clf_en_LR.score(X, y)))
    print("{:<40} {:.2f}".format("Score with L2 penalty:",
                                 clf_l2_LR.score(X, y)))

    if i == 0:
        axes_row[0].set_title("L1 penalty")
        axes_row[1].set_title("Elastic-Net\nl1_ratio = %s" % l1_ratio)
        axes_row[2].set_title("L2 penalty")

    for ax, coefs in zip(axes_row, [coef_l1_LR, coef_en_LR, coef_l2_LR]):
        ax.imshow(np.abs(coefs.reshape(8, 8)), interpolation='nearest',
                  cmap='binary', vmax=1, vmin=0)
        ax.set_xticks(())
        ax.set_yticks(())

    axes_row[0].set_ylabel('C = %s' % C)

plt.show()
```

Total running time of the script: (0 minutes 0.787 seconds)

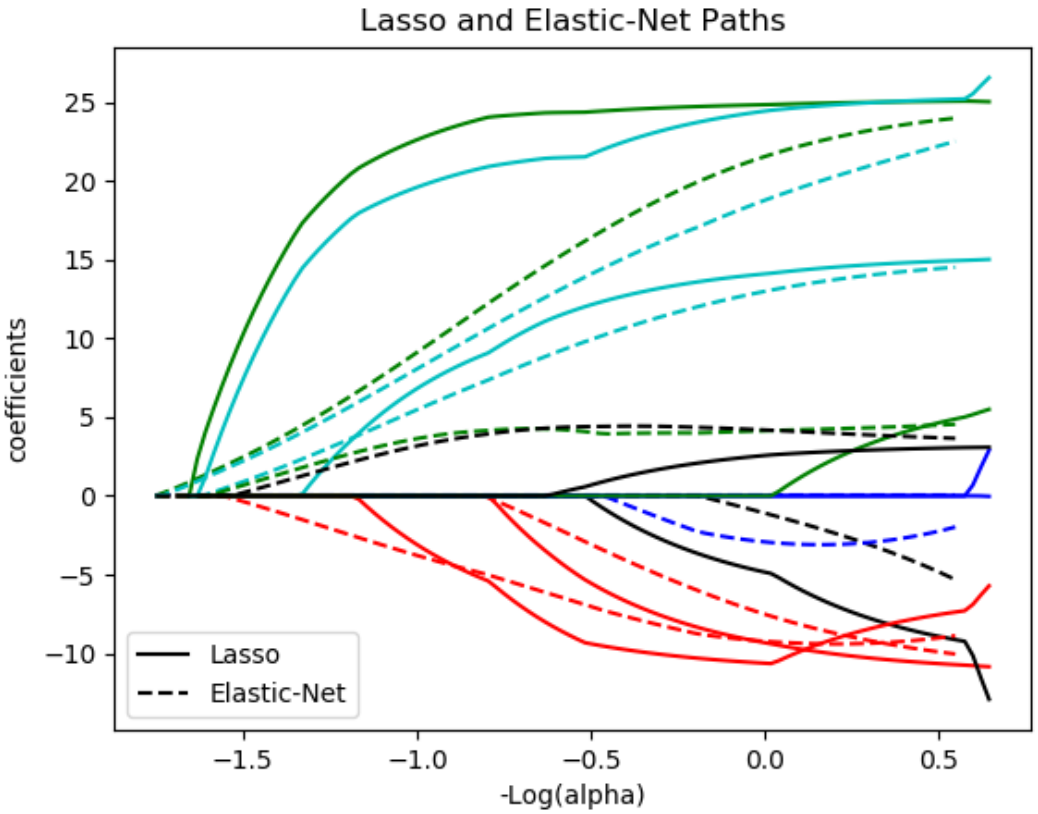
Estimated memory usage: 8 MB

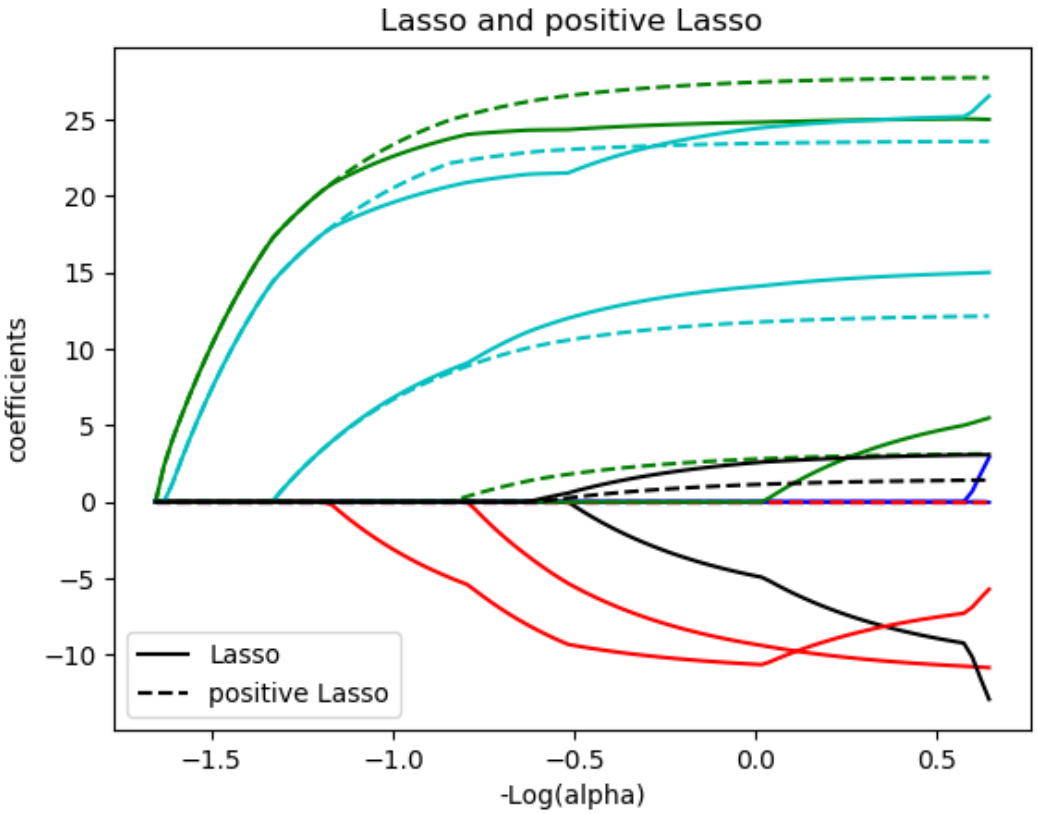
Note: Click *here* to download the full example code or to run this example in your browser via Binder

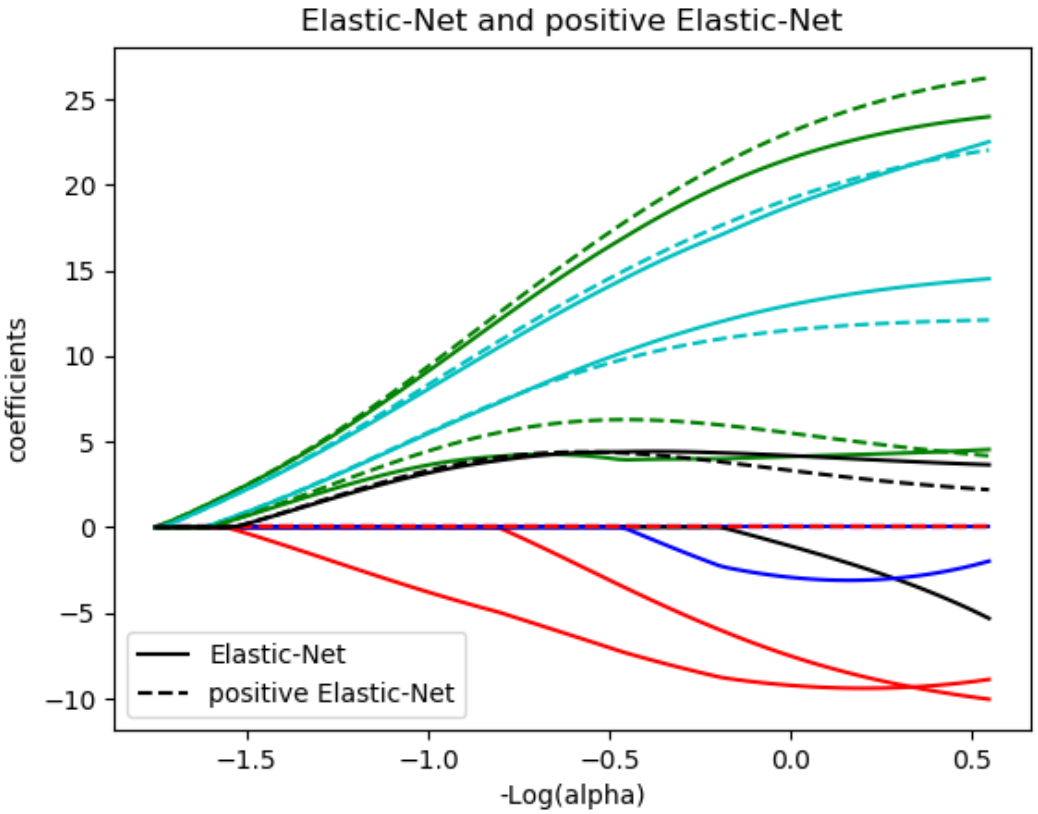
6.16.29 Lasso and Elastic Net

Lasso and elastic net (L1 and L2 penalisation) implemented using a coordinate descent.

The coefficients can be forced to be positive.


Lasso and Elastic-Net Paths
coefficients
-Log(alpha)
Lasso
Elastic-Net
25
20
15
10
5
0
−5
−10
−1.5
−1.0
−0.5
0.0
0.5


Lasso and positive Lasso
coefficients
25
20
15
10
5
0
−5
−10
−1.5
−1.0
−0.5
0.0
0.5
-Log(alpha)
Lasso
positive Lasso



Out:

```
Computing regularization path using the lasso...
Computing regularization path using the positive lasso...
Computing regularization path using the elastic net...
Computing regularization path using the positive elastic net...
```

```
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

from itertools import cycle
import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import lasso_path, enet_path
from sklearn import datasets


X, y = datasets.load_diabetes(return_X_y=True)
```

(continues on next page)

(continued from previous page)

```
X /= X.std(axis=0)  # Standardize data (easier to set the l1_ratio parameter)

# Compute paths

eps = 5e-3  # the smaller it is the longer is the path

print("Computing regularization path using the lasso...")
alphas_lasso, coefs_lasso, _ = lasso_path(X, y, eps, fit_intercept=False)

print("Computing regularization path using the positive lasso...")
alphas_positive_lasso, coefs_positive_lasso, _ = lasso_path(
    X, y, eps, positive=True, fit_intercept=False)
print("Computing regularization path using the elastic net...")
alphas_enet, coefs_enet, _ = enet_path(
    X, y, eps=eps, l1_ratio=0.8, fit_intercept=False)

print("Computing regularization path using the positive elastic net...")
alphas_positive_enet, coefs_positive_enet, _ = enet_path(
    X, y, eps=eps, l1_ratio=0.8, positive=True, fit_intercept=False)

# Display results

plt.figure(1)
colors = cycle(['b', 'r', 'g', 'c', 'k'])
neg_log_alphas_lasso = -np.log10(alphas_lasso)
neg_log_alphas_enet = -np.log10(alphas_enet)
for coef_l, coef_e, c in zip(coefs_lasso, coefs_enet, colors):
    l1 = plt.plot(neg_log_alphas_lasso, coef_l, c=c)
    l2 = plt.plot(neg_log_alphas_enet, coef_e, linestyle='--', c=c)

plt.xlabel('-Log(alpha)')
plt.ylabel('coefficients')
plt.title('Lasso and Elastic-Net Paths')
plt.legend((l1[-1], l2[-1]), ('Lasso', 'Elastic-Net'), loc='lower left')
plt.axis('tight')


plt.figure(2)
neg_log_alphas_positive_lasso = -np.log10(alphas_positive_lasso)
for coef_l, coef_pl, c in zip(coefs_lasso, coefs_positive_lasso, colors):
    l1 = plt.plot(neg_log_alphas_lasso, coef_l, c=c)
    l2 = plt.plot(neg_log_alphas_positive_lasso, coef_pl, linestyle='--', c=c)

plt.xlabel('-Log(alpha)')
plt.ylabel('coefficients')
plt.title('Lasso and positive Lasso')
plt.legend((l1[-1], l2[-1]), ('Lasso', 'positive Lasso'), loc='lower left')
plt.axis('tight')


plt.figure(3)
neg_log_alphas_positive_enet = -np.log10(alphas_positive_enet)
for (coef_e, coef_pe, c) in zip(coefs_enet, coefs_positive_enet, colors):
    l1 = plt.plot(neg_log_alphas_enet, coef_e, c=c)
    l2 = plt.plot(neg_log_alphas_positive_enet, coef_pe, linestyle='--', c=c)
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('-Log(alpha)')
plt.ylabel('coefficients')
plt.title('Elastic-Net and positive Elastic-Net')
plt.legend((l1[-1], l2[-1]), ('Elastic-Net', 'positive Elastic-Net'),
           loc='lower left')
plt.axis('tight')
plt.show()
```

Total running time of the script: (0 minutes 0.721 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.30 Automatic Relevance Determination Regression (ARD)

Fit regression model with Bayesian Ridge Regression.

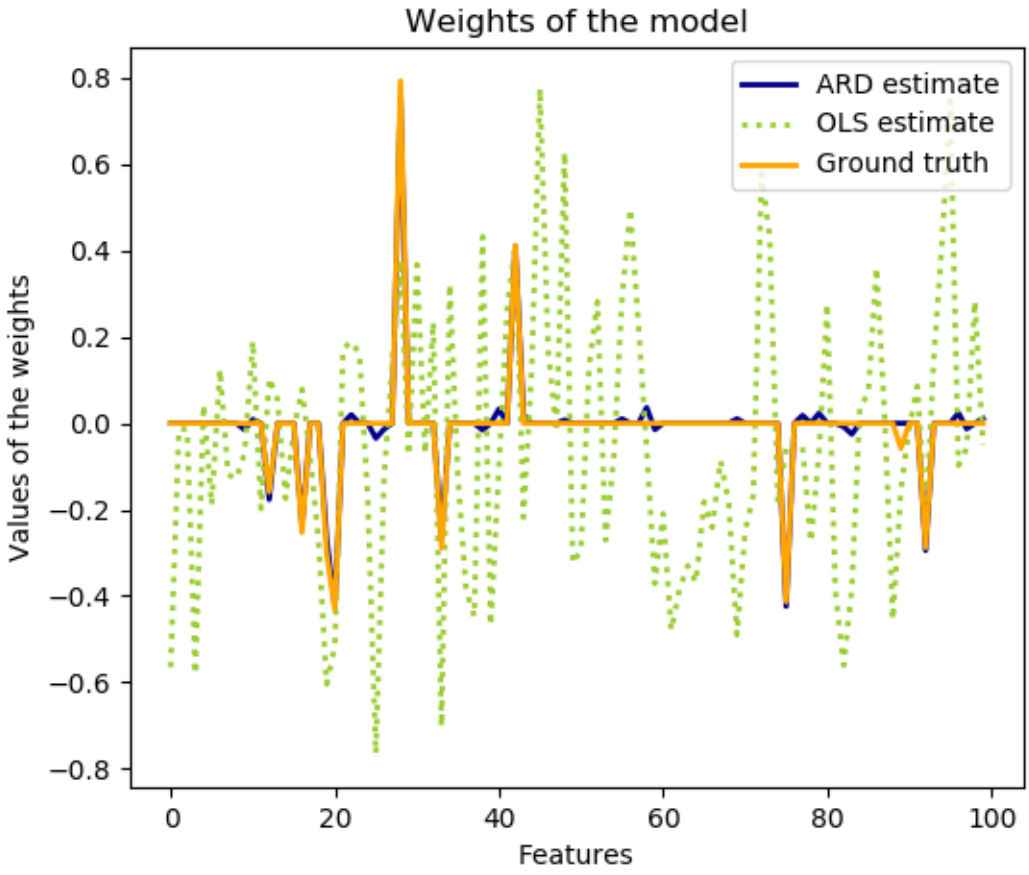
See *Bayesian Ridge Regression* for more information on the regressor.

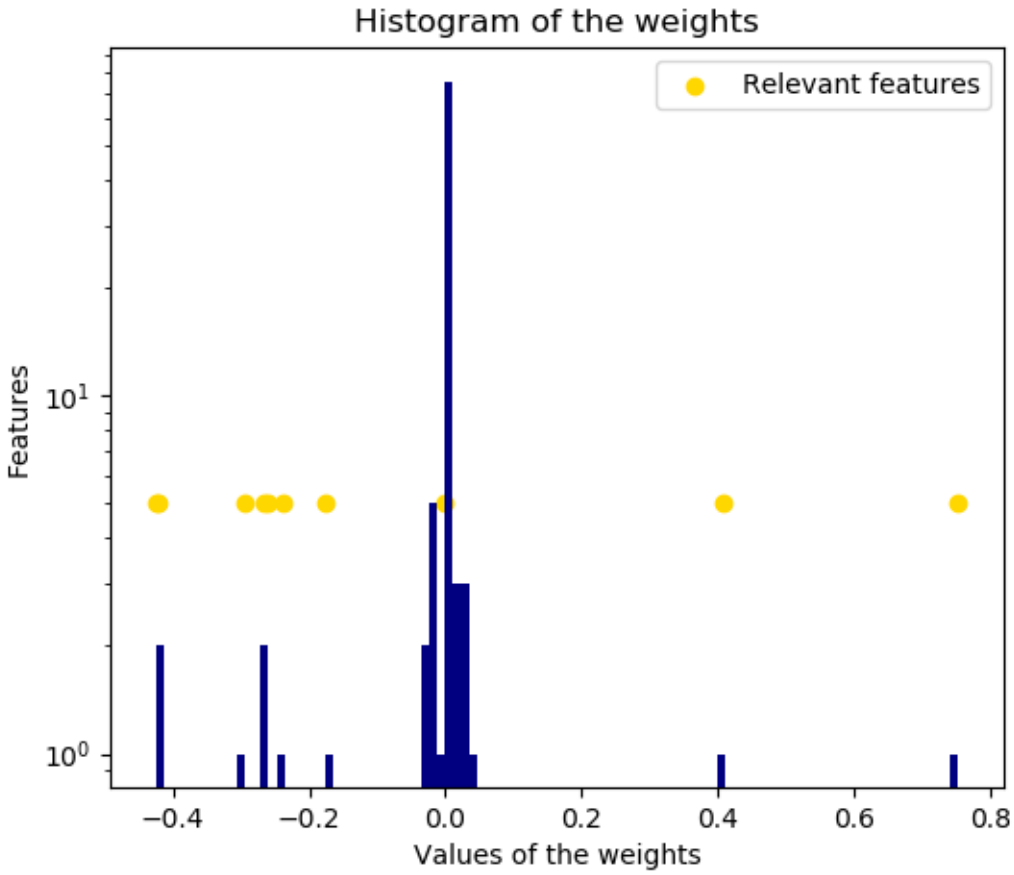
Compared to the OLS (ordinary least squares) estimator, the coefficient weights are slightly shifted toward zeros, which stabilises them.

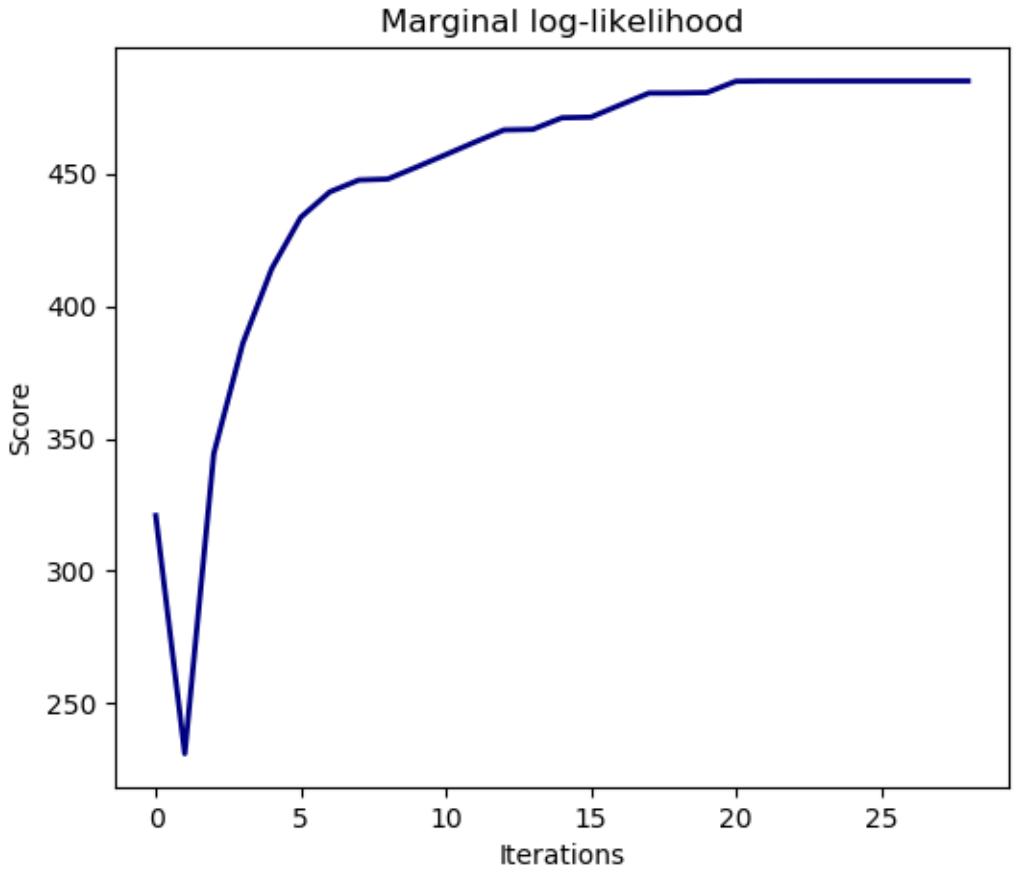
The histogram of the estimated weights is very peaked, as a sparsity-inducing prior is implied on the weights.

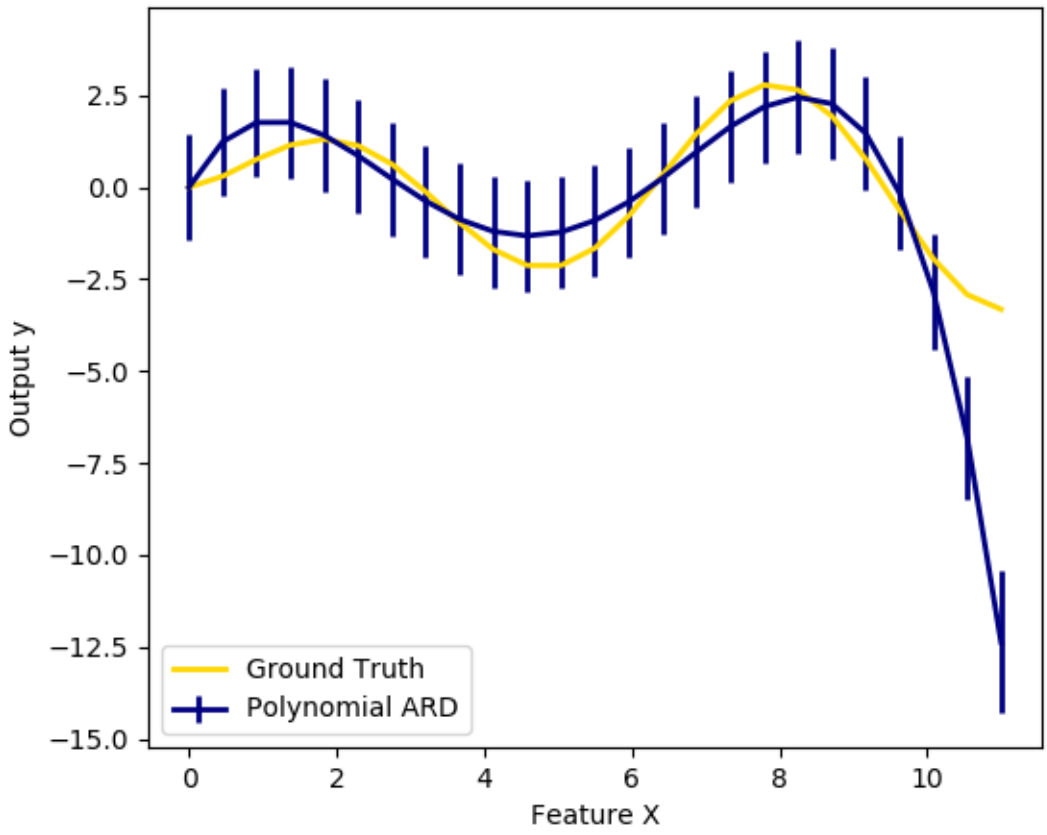
The estimation of the model is done by iteratively maximizing the marginal log-likelihood of the observations.

We also plot predictions and uncertainties for ARD for one dimensional regression using polynomial feature expansion. Note the uncertainty starts going up on the right side of the plot. This is because these test samples are outside of the range of the training samples.


Weights of the model
ARD estimate
OLS estimate
Ground truth
Values of the weights
Features
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
0
20
40
60
80
100


Histogram of the weights
Relevant features
Features
10^1
10^0
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8
Values of the weights


Marginal log-likelihood
Score
Iterations
450
400
350
300
250
0
5
10
15
20
25




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

from sklearn.linear_model import ARDRegression, LinearRegression

# ############################################################################
# Generating simulated data with Gaussian weights

# Parameters of the example
np.random.seed(0)
n_samples, n_features = 100, 100
# Create Gaussian data
X = np.random.randn(n_samples, n_features)
# Create weights with a precision lambda_ of 4.
lambda_ = 4.
w = np.zeros(n_features)
# Only keep 10 weights of interest
relevant_features = np.random.randint(0, n_features, 10)
for i in relevant_features:
    w[i] = stats.norm.rvs(loc=0, scale=1. / np.sqrt(lambda_))
# Create noise with a precision alpha of 50.
```

(continues on next page)

(continued from previous page)

```
alpha_ = 50.
noise = stats.norm.rvs(loc=0, scale=1. / np.sqrt(alpha_), size=n_samples)
# Create the target
y = np.dot(X, w) + noise

# ############################################################################
# Fit the ARD Regression
clf = ARDRegression(compute_score=True)
clf.fit(X, y)

ols = LinearRegression()
ols.fit(X, y)

# ############################################################################
# Plot the true weights, the estimated weights, the histogram of the
# weights, and predictions with standard deviations
plt.figure(figsize=(6, 5))
plt.title("Weights of the model")
plt.plot(clf.coef_, color='darkblue', linestyle='-', linewidth=2,
         label="ARD estimate")
plt.plot(ols.coef_, color='yellowgreen', linestyle=':', linewidth=2,
         label="OLS estimate")
plt.plot(w, color='orange', linestyle='-', linewidth=2, label="Ground truth")
plt.xlabel("Features")
plt.ylabel("Values of the weights")
plt.legend(loc=1)

plt.figure(figsize=(6, 5))
plt.title("Histogram of the weights")
plt.hist(clf.coef_, bins=n_features, color='navy', log=True)
plt.scatter(clf.coef_[relevant_features], np.full(len(relevant_features), 5.),
            color='gold', marker='o', label="Relevant features")
plt.ylabel("Features")
plt.xlabel("Values of the weights")
plt.legend(loc=1)

plt.figure(figsize=(6, 5))
plt.title("Marginal log-likelihood")
plt.plot(clf.scores_, color='navy', linewidth=2)
plt.ylabel("Score")
plt.xlabel("Iterations")


# Plotting some predictions for polynomial regression
def f(x, noise_amount):
    y = np.sqrt(x) * np.sin(x)
    noise = np.random.normal(0, 1, len(x))
    return y + noise_amount * noise


degree = 10
X = np.linspace(0, 10, 100)
y = f(X, noise_amount=1)
clf_poly = ARDRegression(threshold_lambda=1e5)
clf_poly.fit(np.vander(X, degree), y)

X_plot = np.linspace(0, 11, 25)
```

(continues on next page)

(continued from previous page)

```
y_plot = f(X_plot, noise_amount=0)
y_mean, y_std = clf_poly.predict(np.vander(X_plot, degree), return_std=True)
plt.figure(figsize=(6, 5))
plt.errorbar(X_plot, y_mean, y_std, color='navy',
             label="Polynomial ARD", linewidth=2)
plt.plot(X_plot, y_plot, color='gold', linewidth=2,
         label="Ground Truth")
plt.ylabel("Output y")
plt.xlabel("Feature X")
plt.legend(loc="lower left")
plt.show()
```

Total running time of the script: (0 minutes 0.866 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.31 Bayesian Ridge Regression

Computes a Bayesian Ridge Regression on a synthetic dataset.

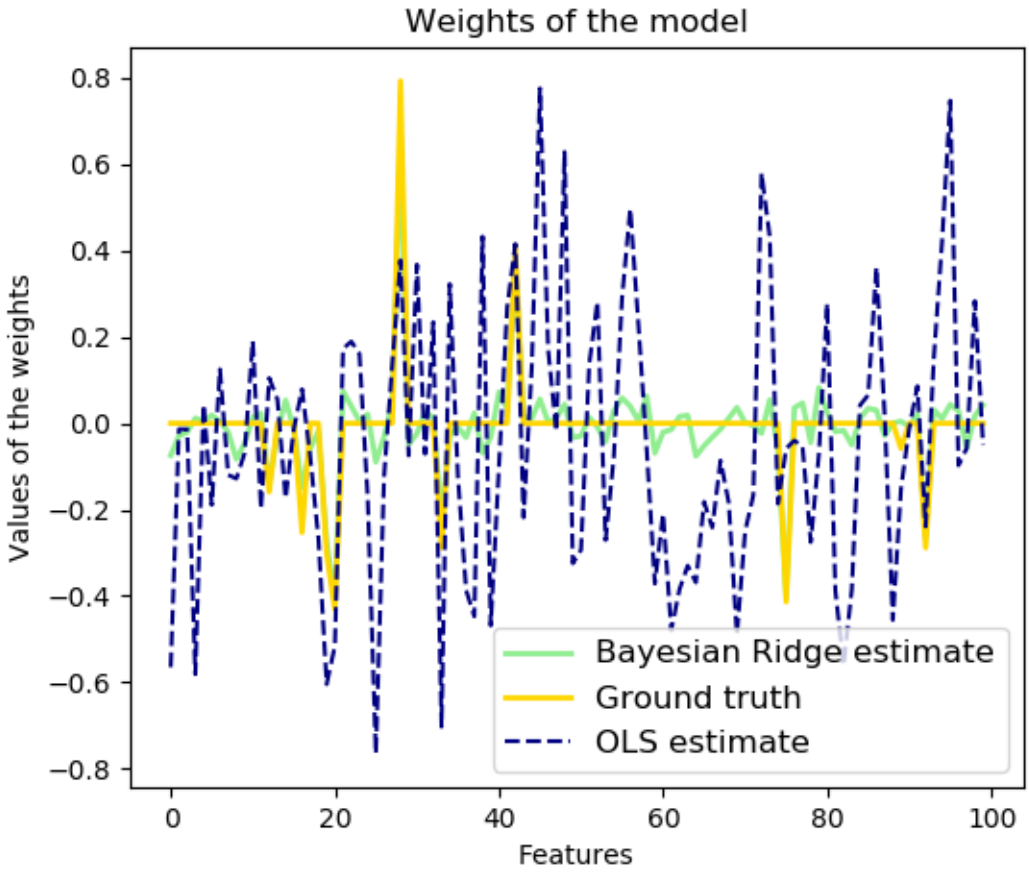
See *Bayesian Ridge Regression* for more information on the regressor.

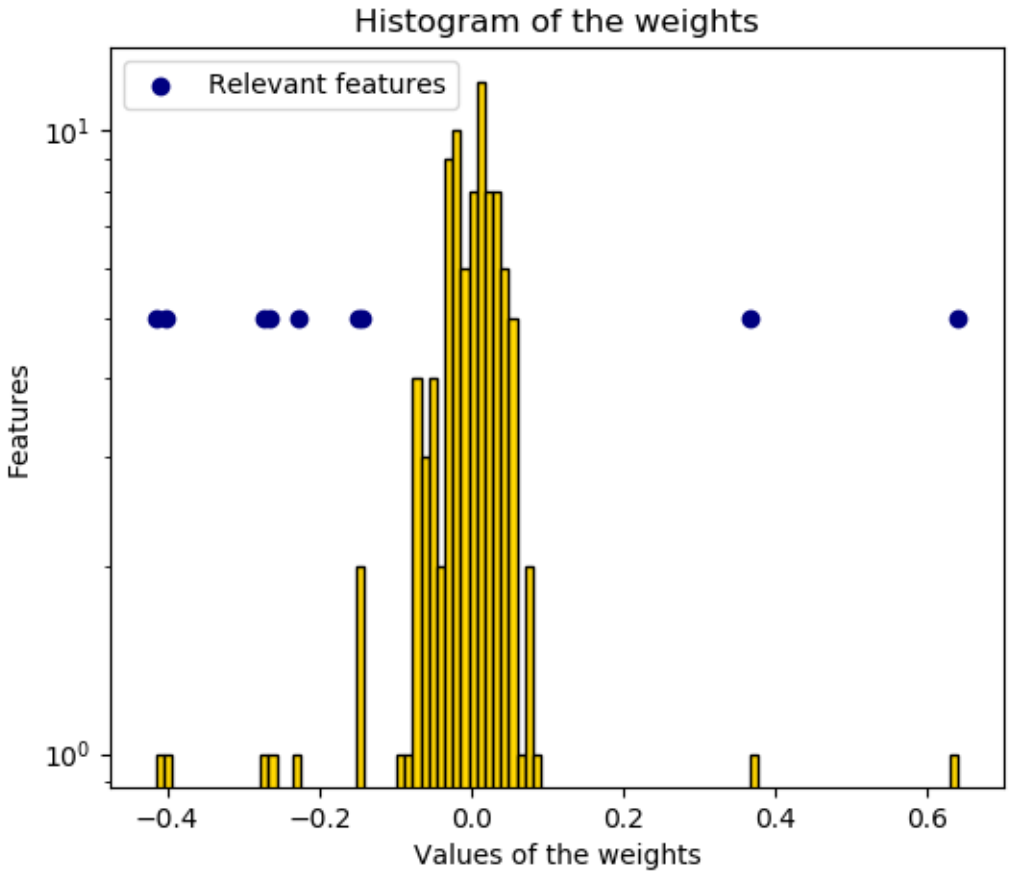
Compared to the OLS (ordinary least squares) estimator, the coefficient weights are slightly shifted toward zeros, which stabilises them.

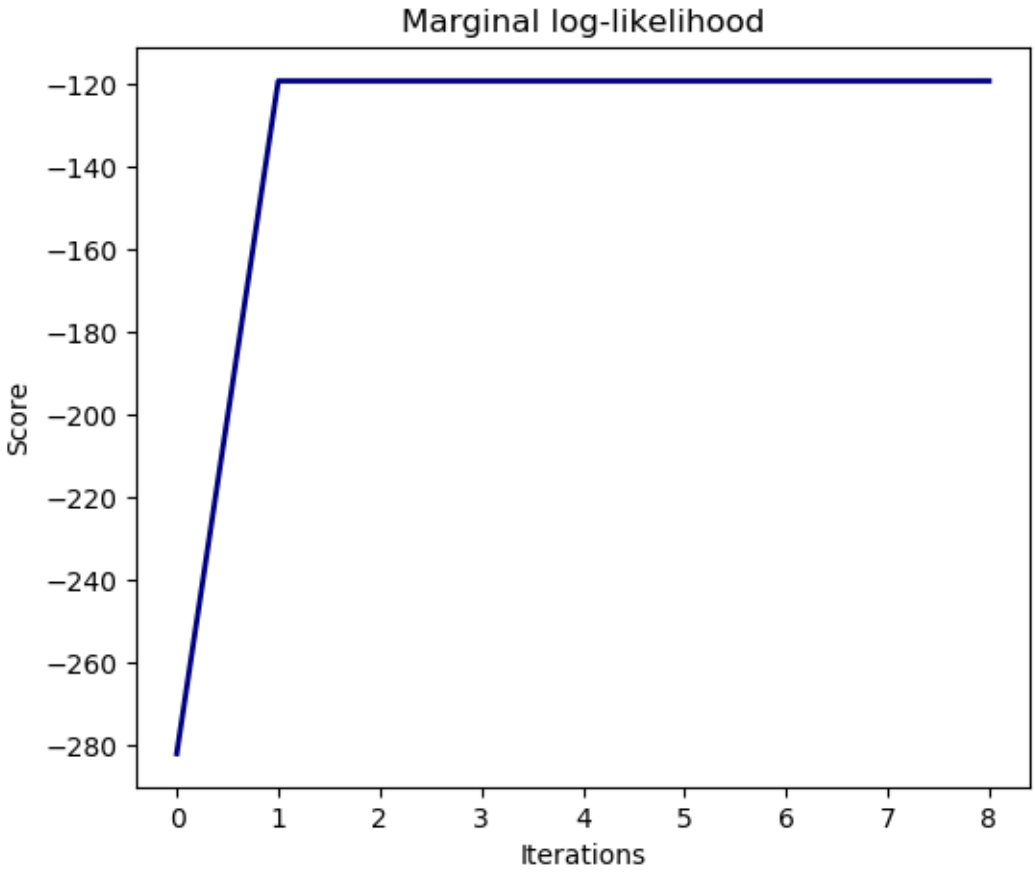
As the prior on the weights is a Gaussian prior, the histogram of the estimated weights is Gaussian.

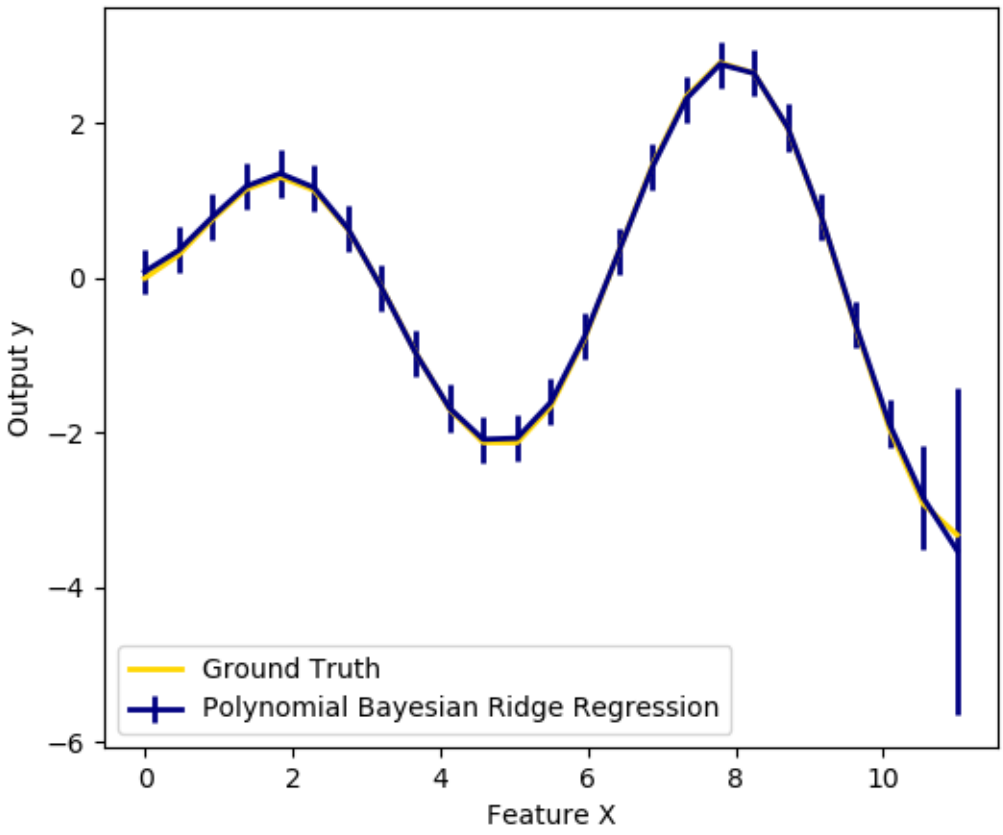
The estimation of the model is done by iteratively maximizing the marginal log-likelihood of the observations.

We also plot predictions and uncertainties for Bayesian Ridge Regression for one dimensional regression using polynomial feature expansion. Note the uncertainty starts going up on the right side of the plot. This is because these test samples are outside of the range of the training samples.


Weights of the model
Values of the weights
Features
Bayesian Ridge estimate
Ground truth
OLS estimate
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
0
20
40
60
80
100


Histogram of the weights
Relevant features
10^1
10^0
Features
−0.4
−0.2
0.0
0.2
0.4
0.6
Values of the weights


Marginal log-likelihood
−120
−140
−160
−180
−200
−220
−240
−260
−280
Score
0
1
2
3
4
5
6
7
8
Iterations




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

from sklearn.linear_model import BayesianRidge, LinearRegression

# ##############################################################################
# Generating simulated data with Gaussian weights
np.random.seed(0)
n_samples, n_features = 100, 100
X = np.random.randn(n_samples, n_features)  # Create Gaussian data
# Create weights with a precision lambda_ of 4.
lambda_ = 4.
w = np.zeros(n_features)
# Only keep 10 weights of interest
relevant_features = np.random.randint(0, n_features, 10)
for i in relevant_features:
    w[i] = stats.norm.rvs(loc=0, scale=1. / np.sqrt(lambda_))
# Create noise with a precision alpha of 50.
alpha_ = 50.
noise = stats.norm.rvs(loc=0, scale=1. / np.sqrt(alpha_), size=n_samples)
# Create the target
```

(continues on next page)

(continued from previous page)

```
y = np.dot(X, w) + noise

# ############################################################################
# Fit the Bayesian Ridge Regression and an OLS for comparison
clf = BayesianRidge(compute_score=True)
clf.fit(X, y)

ols = LinearRegression()
ols.fit(X, y)

# ############################################################################
# Plot true weights, estimated weights, histogram of the weights, and
# predictions with standard deviations
lw = 2
plt.figure(figsize=(6, 5))
plt.title("Weights of the model")
plt.plot(clf.coef_, color='lightgreen', linewidth=lw,
         label="Bayesian Ridge estimate")
plt.plot(w, color='gold', linewidth=lw, label="Ground truth")
plt.plot(ols.coef_, color='navy', linestyle='--', label="OLS estimate")
plt.xlabel("Features")
plt.ylabel("Values of the weights")
plt.legend(loc="best", prop=dict(size=12))

plt.figure(figsize=(6, 5))
plt.title("Histogram of the weights")
plt.hist(clf.coef_, bins=n_features, color='gold', log=True,
         edgecolor='black')
plt.scatter(clf.coef_[relevant_features], np.full(len(relevant_features), 5.),
            color='navy', label="Relevant features")
plt.ylabel("Features")
plt.xlabel("Values of the weights")
plt.legend(loc="upper left")

plt.figure(figsize=(6, 5))
plt.title("Marginal log-likelihood")
plt.plot(clf.scores_, color='navy', linewidth=lw)
plt.ylabel("Score")
plt.xlabel("Iterations")


# Plotting some predictions for polynomial regression
def f(x, noise_amount):
    y = np.sqrt(x) * np.sin(x)
    noise = np.random.normal(0, 1, len(x))
    return y + noise_amount * noise


degree = 10
X = np.linspace(0, 10, 100)
y = f(X, noise_amount=0.1)
clf_poly = BayesianRidge()
clf_poly.fit(np.vander(X, degree), y)

X_plot = np.linspace(0, 11, 25)
y_plot = f(X_plot, noise_amount=0)
y_mean, y_std = clf_poly.predict(np.vander(X_plot, degree), return_std=True)
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(6, 5))
plt.errorbar(X_plot, y_mean, y_std, color='navy',
             label="Polynomial Bayesian Ridge Regression", linewidth=lw)
plt.plot(X_plot, y_plot, color='gold', linewidth=lw,
         label="Ground Truth")
plt.ylabel("Output y")
plt.xlabel("Feature X")
plt.legend(loc="lower left")
plt.show()
```

Total running time of the script: (0 minutes 0.789 seconds)

Estimated memory usage: 8 MB

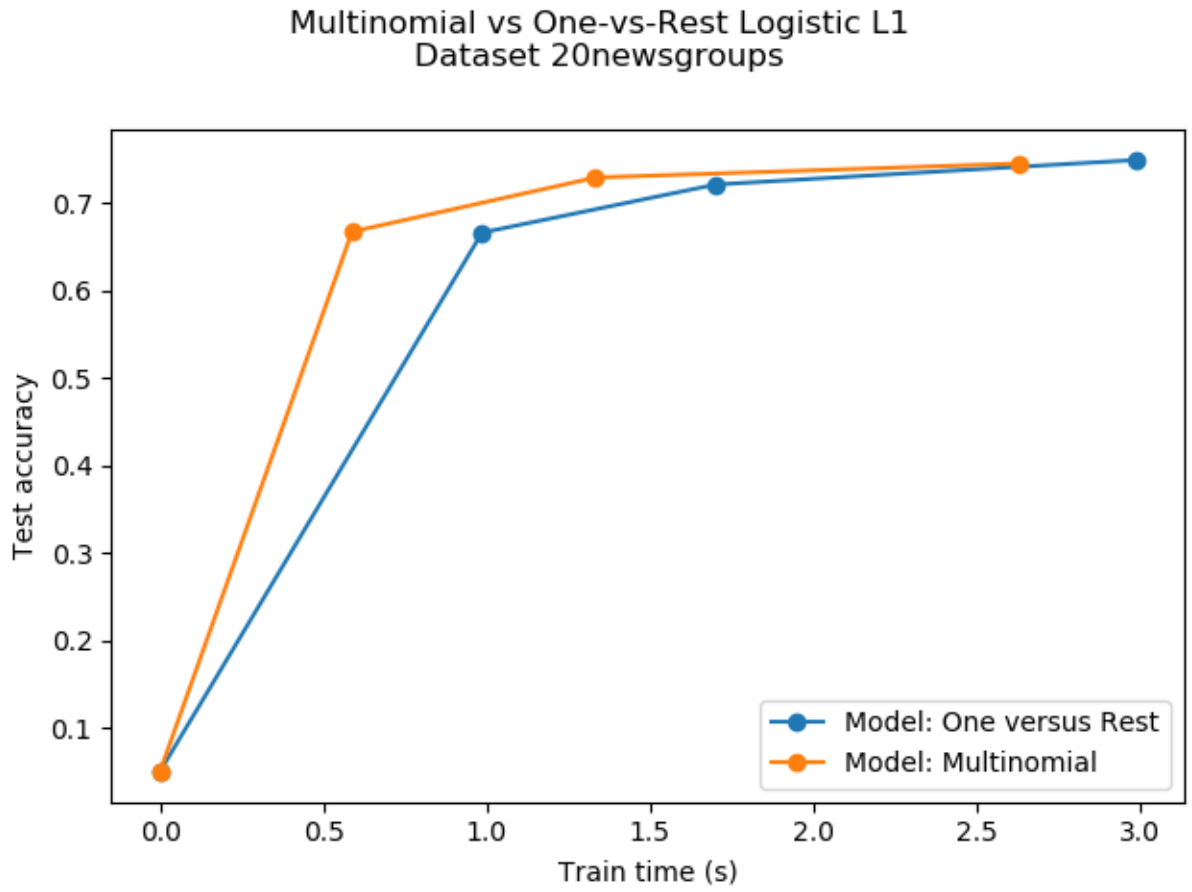
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.32 Multiclass sparse logistic regression on 20newgroups

Comparison of multinomial logistic L1 vs one-versus-rest L1 logistic regression to classify documents from the newsgroups20 dataset. Multinomial logistic regression yields more accurate results and is faster to train on the larger scale dataset.

Here we use the l1 sparsity that trims the weights of not informative features to zero. This is good if the goal is to extract the strongly discriminative vocabulary of each class. If the goal is to get the best predictive accuracy, it is better to use the non sparsity-inducing l2 penalty instead.

A more traditional (and possibly better) way to predict on a sparse subset of input features would be to use univariate feature selection followed by a traditional (l2-penalised) logistic regression model.



Out:

```
Dataset 20newsgroup, train_samples=9000, n_features=130107, n_classes=20
[model=One versus Rest, solver=saga] Number of epochs: 1
[model=One versus Rest, solver=saga] Number of epochs: 2
[model=One versus Rest, solver=saga] Number of epochs: 4
Test accuracy for model ovr: 0.7490
% non-zero coefficients for model ovr, per class:
 [0.31743104 0.36815852 0.4181174  0.46115889 0.24595141 0.41350581
 0.31281945 0.27054655 0.58720899 0.32972861 0.4158116  0.3312658
 0.41888599 0.41120001 0.59643217 0.31666244 0.34279478 0.28130692
 0.35278655 0.24748861]
Run time (4 epochs) for model ovr:2.99
[model=Multinomial, solver=saga] Number of epochs: 1
[model=Multinomial, solver=saga] Number of epochs: 3
[model=Multinomial, solver=saga] Number of epochs: 7
Test accuracy for model multinomial: 0.7450
% non-zero coefficients for model multinomial, per class:
 [0.13219888 0.11452112 0.13066169 0.13681047 0.12066991 0.15909982
 0.13450468 0.09146318 0.07916561 0.12143851 0.13911627 0.10760374
 0.18984374 0.12143851 0.17524038 0.22289346 0.11605832 0.07916561
 0.07301682 0.15141384]
Run time (7 epochs) for model multinomial:2.63
Example run in 11.120 s
```

```
import timeit
import warnings

import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import fetch_20newsgroups_vectorized
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.exceptions import ConvergenceWarning

print(__doc__)
# Author: Arthur Mensch

warnings.filterwarnings("ignore", category=ConvergenceWarning,
                        module="sklearn")
t0 = timeit.default_timer()

# We use SAGA solver
solver = 'saga'

# Turn down for faster run time
n_samples = 10000

X, y = fetch_20newsgroups_vectorized('all', return_X_y=True)
X = X[:n_samples]
y = y[:n_samples]

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    random_state=42,
                                                    stratify=y,
                                                    test_size=0.1)
train_samples, n_features = X_train.shape
n_classes = np.unique(y).shape[0]

print('Dataset 20newsgroup, train_samples=%i, n_features=%i, n_classes=%i'
      % (train_samples, n_features, n_classes))

models = {'ovr': {'name': 'One versus Rest', 'iters': [1, 2, 4]},
          'multinomial': {'name': 'Multinomial', 'iters': [1, 3, 7]}}

for model in models:
    # Add initial chance-level values for plotting purpose
    accuracies = [1 / n_classes]
    times = [0]
    densities = [1]

    model_params = models[model]

    # Small number of epochs for fast runtime
    for this_max_iter in model_params['iters']:
        print('[model=%s, solver=%s] Number of epochs: %s' %
              (model_params['name'], solver, this_max_iter))
        lr = LogisticRegression(solver=solver,
```

(continues on next page)

(continued from previous page)

```
                                multi_class=model,
                                penalty='l1',
                                max_iter=this_max_iter,
                                random_state=42,
                                )
            t1 = timeit.default_timer()
            lr.fit(X_train, y_train)
            train_time = timeit.default_timer() - t1

            y_pred = lr.predict(X_test)
            accuracy = np.sum(y_pred == y_test) / y_test.shape[0]
            density = np.mean(lr.coef_ != 0, axis=1) * 100
            accuracies.append(accuracy)
            densities.append(density)
            times.append(train_time)
        models[model]['times'] = times
        models[model]['densities'] = densities
        models[model]['accuracies'] = accuracies
        print('Test accuracy for model %s: %.4f' % (model, accuracies[-1]))
        print('%% non-zero coefficients for model %s, '
              'per class:\n %s' % (model, densities[-1]))
        print('Run time (%i epochs) for model %s:'
              '%.2f' % (model_params['iters'][-1], model, times[-1]))

fig = plt.figure()
ax = fig.add_subplot(111)

for model in models:
    name = models[model]['name']
    times = models[model]['times']
    accuracies = models[model]['accuracies']
    ax.plot(times, accuracies, marker='o',
            label='Model: %s' % name)
    ax.set_xlabel('Train time (s)')
    ax.set_ylabel('Test accuracy')
ax.legend()
fig.suptitle('Multinomial vs One-vs-Rest Logistic L1\n'
             'Dataset %s' % '20newsgroups')
fig.tight_layout()
fig.subplots_adjust(top=0.85)
run_time = timeit.default_timer() - t0
print('Example run in %.3f s' % run_time)
plt.show()
```

Total running time of the script: (0 minutes 11.405 seconds)

Estimated memory usage: 51 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.33 Lasso model selection: Cross-Validation / AIC / BIC

Use the Akaike information criterion (AIC), the Bayes Information criterion (BIC) and cross-validation to select an optimal value of the regularization parameter alpha of the *Lasso* estimator.

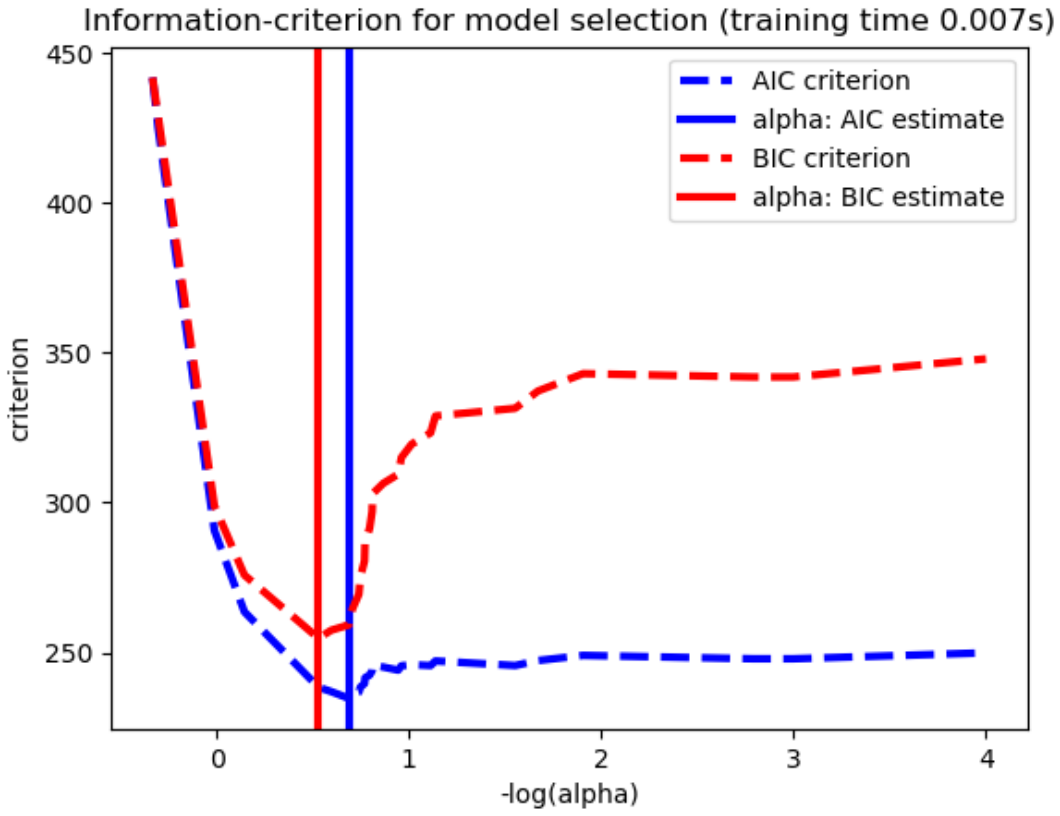
Results obtained with LassoLarsIC are based on AIC/BIC criteria.

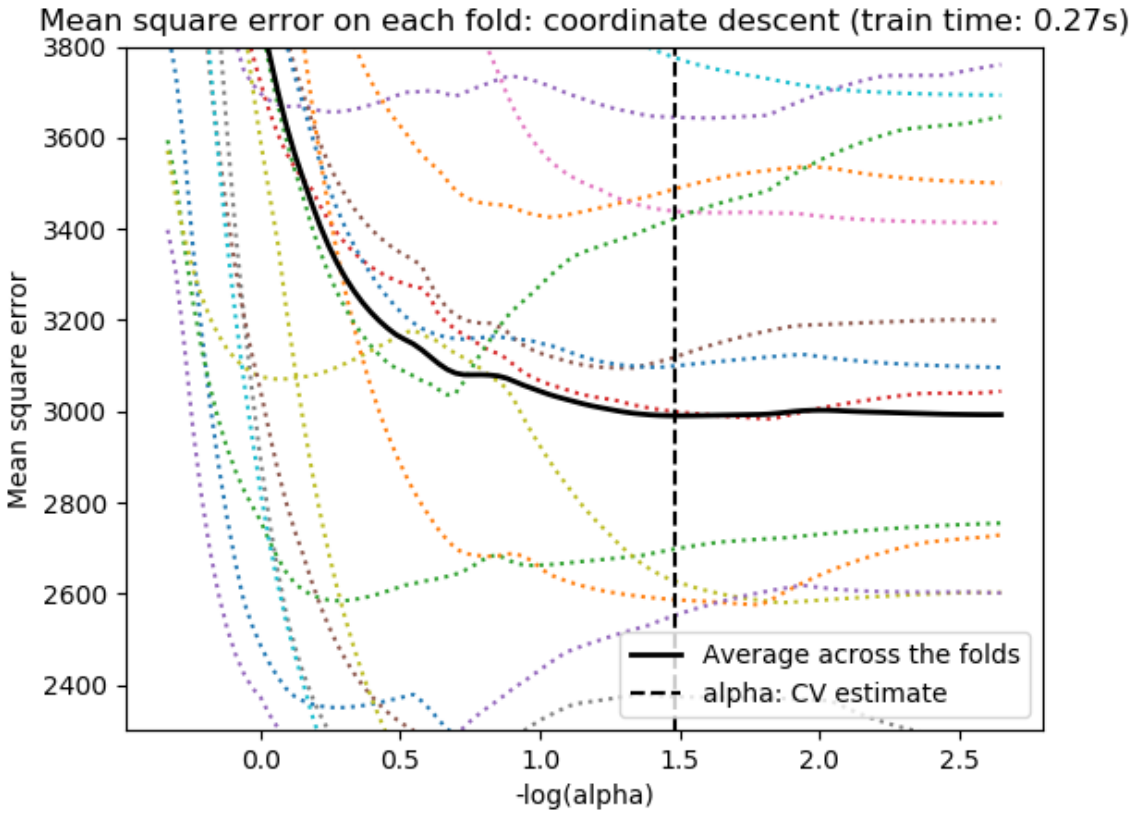
Information-criterion based model selection is very fast, but it relies on a proper estimation of degrees of freedom, are derived for large samples (asymptotic results) and assume the model is correct, i.e. that the data are actually generated by this model. They also tend to break when the problem is badly conditioned (more features than samples).

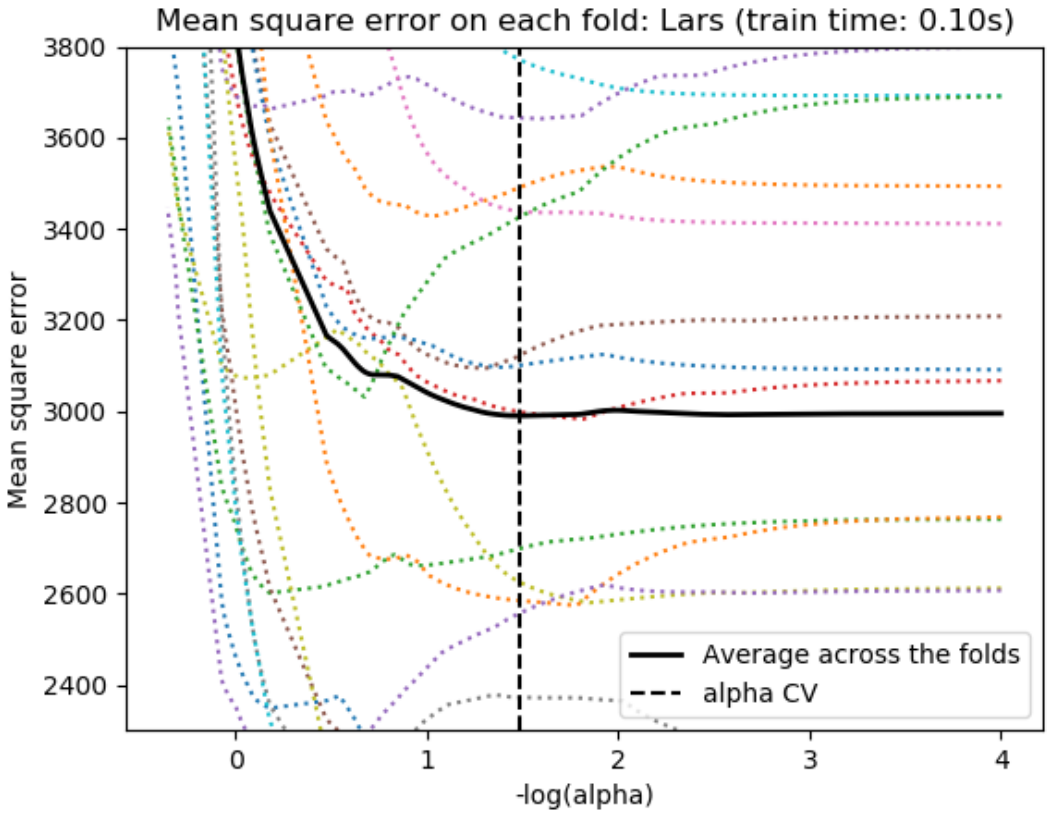
For cross-validation, we use 20-fold with 2 algorithms to compute the Lasso path: coordinate descent, as implemented by the LassoCV class, and Lars (least angle regression) as implemented by the LassoLarsCV class. Both algorithms give roughly the same results. They differ with regards to their execution speed and sources of numerical errors.

Lars computes a path solution only for each kink in the path. As a result, it is very efficient when there are only of few kinks, which is the case if there are few features or samples. Also, it is able to compute the full path without setting any meta parameter. On the opposite, coordinate descent compute the path points on a pre-specified grid (here we use the default). Thus it is more efficient if the number of grid points is smaller than the number of kinks in the path. Such a strategy can be interesting if the number of features is really large and there are enough samples to select a large amount. In terms of numerical errors, for heavily correlated variables, Lars will accumulate more errors, while the coordinate descent algorithm will only sample the path on a grid.

Note how the optimal value of alpha varies for each fold. This illustrates why nested-cross validation is necessary when trying to evaluate the performance of a method for which a parameter is chosen by cross-validation: this choice of parameter may not be optimal for unseen data.




Mean square error on each fold: coordinate descent (train time: 0.27s)
Mean square error
3800
3600
3400
3200
3000
2800
2600
2400
0.0
0.5
1.0
1.5
2.0
2.5
-log(alpha)
Average across the folds
alpha: CV estimate



Out:

```
Computing regularization path using the coordinate descent lasso...
Computing regularization path using the Lars lasso...
```

```
print(__doc__)

# Author: Olivier Grisel, Gael Varoquaux, Alexandre Gramfort
# License: BSD 3 clause

import time

import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import LassoCV, LassoLarsCV, LassoLarsIC
from sklearn import datasets

# This is to avoid division by zero while doing np.log10
EPSILON = 1e-4

X, y = datasets.load_diabetes(return_X_y=True)
```

(continues on next page)

(continued from previous page)

```
rng = np.random.RandomState(42)
X = np.c_[X, rng.randn(X.shape[0], 14)]  # add some bad features

# normalize data as done by Lars to allow for comparison
X /= np.sqrt(np.sum(X ** 2, axis=0))

# ##############################################################################
# LassoLarsIC: least angle regression with BIC/AIC criterion

model_bic = LassoLarsIC(criterion='bic')
t1 = time.time()
model_bic.fit(X, y)
t_bic = time.time() - t1
alpha_bic_ = model_bic.alpha_

model_aic = LassoLarsIC(criterion='aic')
model_aic.fit(X, y)
alpha_aic_ = model_aic.alpha_


def plot_ic_criterion(model, name, color):
    alpha_ = model.alpha_ + EPSILON
    alphas_ = model.alphas_ + EPSILON
    criterion_ = model.criterion_
    plt.plot(-np.log10(alphas_), criterion_, '--', color=color,
             linewidth=3, label='%s criterion' % name)
    plt.axvline(-np.log10(alpha_), color=color, linewidth=3,
                label='alpha: %s estimate' % name)
    plt.xlabel('-log(alpha)')
    plt.ylabel('criterion')


plt.figure()
plot_ic_criterion(model_aic, 'AIC', 'b')
plot_ic_criterion(model_bic, 'BIC', 'r')
plt.legend()
plt.title('Information-criterion for model selection (training time %.3fs)'
          % t_bic)

# ##############################################################################
# LassoCV: coordinate descent

# Compute paths
print("Computing regularization path using the coordinate descent lasso...")
t1 = time.time()
model = LassoCV(cv=20).fit(X, y)
t_lasso_cv = time.time() - t1

# Display results
m_log_alphas = -np.log10(model.alphas_ + EPSILON)

plt.figure()
ymin, ymax = 2300, 3800
plt.plot(m_log_alphas, model.mse_path_, ':')
plt.plot(m_log_alphas, model.mse_path_.mean(axis=-1), 'k',
         label='Average across the folds', linewidth=2)
```

(continues on next page)

(continued from previous page)

```
plt.axvline(-np.log10(model.alpha_ + EPSILON), linestyle='--', color='k',
            label='alpha: CV estimate')

plt.legend()

plt.xlabel('-log(alpha)')
plt.ylabel('Mean square error')
plt.title('Mean square error on each fold: coordinate descent '
          '(train time: %.2fs)' % t_lasso_cv)
plt.axis('tight')
plt.ylim(ymin, ymax)

# #############################################################################
# LassoLarsCV: least angle regression

# Compute paths
print("Computing regularization path using the Lars lasso...")
t1 = time.time()
model = LassoLarsCV(cv=20).fit(X, y)
t_lasso_lars_cv = time.time() - t1

# Display results
m_log_alphas = -np.log10(model.cv_alphas_ + EPSILON)

plt.figure()
plt.plot(m_log_alphas, model.mse_path_, ':')
plt.plot(m_log_alphas, model.mse_path_.mean(axis=-1), 'k',
         label='Average across the folds', linewidth=2)
plt.axvline(-np.log10(model.alpha_), linestyle='--', color='k',
            label='alpha CV')
plt.legend()

plt.xlabel('-log(alpha)')
plt.ylabel('Mean square error')
plt.title('Mean square error on each fold: Lars (train time: %.2fs)'
          % t_lasso_lars_cv)
plt.axis('tight')
plt.ylim(ymin, ymax)

plt.show()
```

Total running time of the script: (0 minutes 0.872 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.16.34 Early stopping of Stochastic Gradient Descent

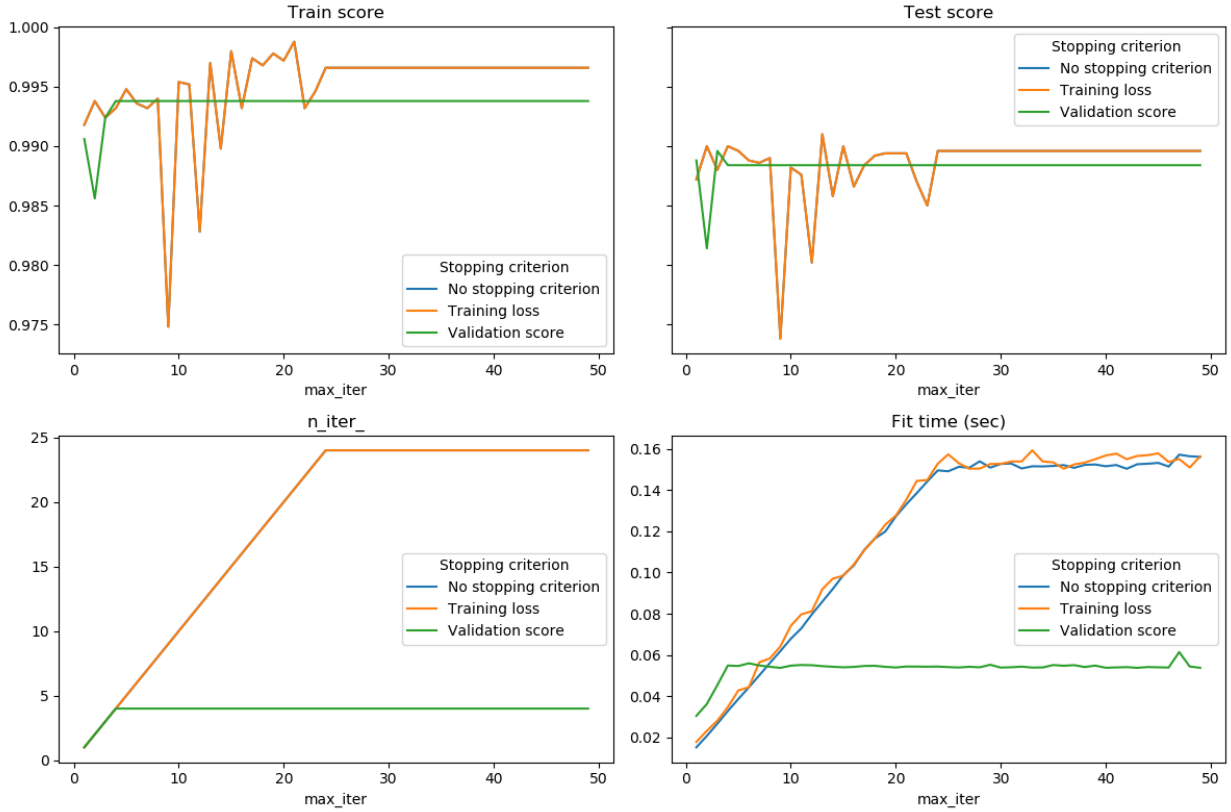
Stochastic Gradient Descent is an optimization technique which minimizes a loss function in a stochastic fashion, performing a gradient descent step sample by sample. In particular, it is a very efficient method to fit linear models.

As a stochastic method, the loss function is not necessarily decreasing at each iteration, and convergence is only guaranteed in expectation. For this reason, monitoring the convergence on the loss function can be difficult.

Another approach is to monitor convergence on a validation score. In this case, the input data is split into a training set and a validation set. The model is then fitted on the training set and the stopping criterion is based on the prediction score computed on the validation set. This enables us to find the least number of iterations which is sufficient to build a model that generalizes well to unseen data and reduces the chance of over-fitting the training data.

This early stopping strategy is activated if `early_stopping=True`; otherwise the stopping criterion only uses the training loss on the entire input data. To better control the early stopping strategy, we can specify a parameter `validation_fraction` which set the fraction of the input dataset that we keep aside to compute the validation score. The optimization will continue until the validation score did not improve by at least `tol` during the last `n_iter_no_change` iterations. The actual number of iterations is available at the attribute `n_iter_`.

This example illustrates how the early stopping can used in the *`sklearn.linear_model.SGDClassifier`* model to achieve almost the same accuracy as compared to a model built without early stopping. This can significantly reduce training time. Note that scores differ between the stopping criteria even from early iterations because some of the training data is held out with the validation stopping criterion.



Out:

```
No stopping criterion: ..................................................
Training loss: ..........................................................
Validation score: .......................................................
```

```
# Authors: Tom Dupre la Tour
#
```

(continues on next page)

(continued from previous page)

```
# License: BSD 3 clause
import time
import sys

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn import linear_model
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.utils._testing import ignore_warnings
from sklearn.exceptions import ConvergenceWarning
from sklearn.utils import shuffle

print(__doc__)


def load_mnist(n_samples=None, class_0='0', class_1='8'):
    """Load MNIST, select two classes, shuffle and return only n_samples."""
    # Load data from http://openml.org/d/554
    mnist = fetch_openml('mnist_784', version=1)

    # take only two classes for binary classification
    mask = np.logical_or(mnist.target == class_0, mnist.target == class_1)

    X, y = shuffle(mnist.data[mask], mnist.target[mask], random_state=42)
    if n_samples is not None:
        X, y = X[:n_samples], y[:n_samples]
    return X, y


@ignore_warnings(category=ConvergenceWarning)
def fit_and_score(estimator, max_iter, X_train, X_test, y_train, y_test):
    """Fit the estimator on the train set and score it on both sets"""
    estimator.set_params(max_iter=max_iter)
    estimator.set_params(random_state=0)

    start = time.time()
    estimator.fit(X_train, y_train)

    fit_time = time.time() - start
    n_iter = estimator.n_iter_
    train_score = estimator.score(X_train, y_train)
    test_score = estimator.score(X_test, y_test)

    return fit_time, n_iter, train_score, test_score


# Define the estimators to compare
estimator_dict = {
    'No stopping criterion':
    linear_model.SGDClassifier(n_iter_no_change=3),
    'Training loss':
    linear_model.SGDClassifier(early_stopping=False, n_iter_no_change=3,
                               tol=0.1),
    'Validation score':
```

(continues on next page)

(continued from previous page)

```
    linear_model.SGDClassifier(early_stopping=True, n_iter_no_change=3,
                               tol=0.0001, validation_fraction=0.2)
}

# Load the dataset
X, y = load_mnist(n_samples=10000)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,
                                                    random_state=0)

results = []
for estimator_name, estimator in estimator_dict.items():
    print(estimator_name + ': ', end='')
    for max_iter in range(1, 50):
        print('.', end='')
        sys.stdout.flush()

        fit_time, n_iter, train_score, test_score = fit_and_score(
            estimator, max_iter, X_train, X_test, y_train, y_test)

        results.append((estimator_name, max_iter, fit_time, n_iter,
                        train_score, test_score))
    print('')

# Transform the results in a pandas dataframe for easy plotting
columns = [
    'Stopping criterion', 'max_iter', 'Fit time (sec)', 'n_iter_',
    'Train score', 'Test score'
]
results_df = pd.DataFrame(results, columns=columns)

# Define what to plot (x_axis, y_axis)
lines = 'Stopping criterion'
plot_list = [
    ('max_iter', 'Train score'),
    ('max_iter', 'Test score'),
    ('max_iter', 'n_iter_'),
    ('max_iter', 'Fit time (sec)'),
]

nrows = 2
ncols = int(np.ceil(len(plot_list) / 2.))
fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=(6 * ncols,
                                                            4 * nrows))
axes[0, 0].get_shared_y_axes().join(axes[0, 0], axes[0, 1])

for ax, (x_axis, y_axis) in zip(axes.ravel(), plot_list):
    for criterion, group_df in results_df.groupby(lines):
        group_df.plot(x=x_axis, y=y_axis, label=criterion, ax=ax)
    ax.set_title(y_axis)
    ax.legend(title=lines)

fig.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 34.642 seconds)

Estimated memory usage: 778 MB

6.17 Inspection

Examples related to the *sklearn.inspection* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.17.1 Permutation Importance with Multicollinear or Correlated Features

In this example, we compute the permutation importance on the Wisconsin breast cancer dataset using *permutation_importance*. The *RandomForestClassifier* can easily get about 97% accuracy on a test dataset. Because this dataset contains multicollinear features, the permutation importance will show that none of the features are important. One approach to handling multicollinearity is by performing hierarchical clustering on the features' Spearman rank-order correlations, picking a threshold, and keeping a single feature from each cluster.

Note: See also *Permutation Importance vs Random Forest Feature Importance (MDI)*

```
print(__doc__)
from collections import defaultdict

import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import spearmanr
from scipy.cluster import hierarchy

from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import RandomForestClassifier
from sklearn.inspection import permutation_importance
from sklearn.model_selection import train_test_split
```

Random Forest Feature Importance on Breast Cancer Data

First, we train a random forest on the breast cancer dataset and evaluate its accuracy on a test set:

```
data = load_breast_cancer()
X, y = data.data, data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

clf = RandomForestClassifier(n_estimators=100, random_state=42)
clf.fit(X_train, y_train)
print("Accuracy on test data: {:.2f}".format(clf.score(X_test, y_test)))
```

Out:

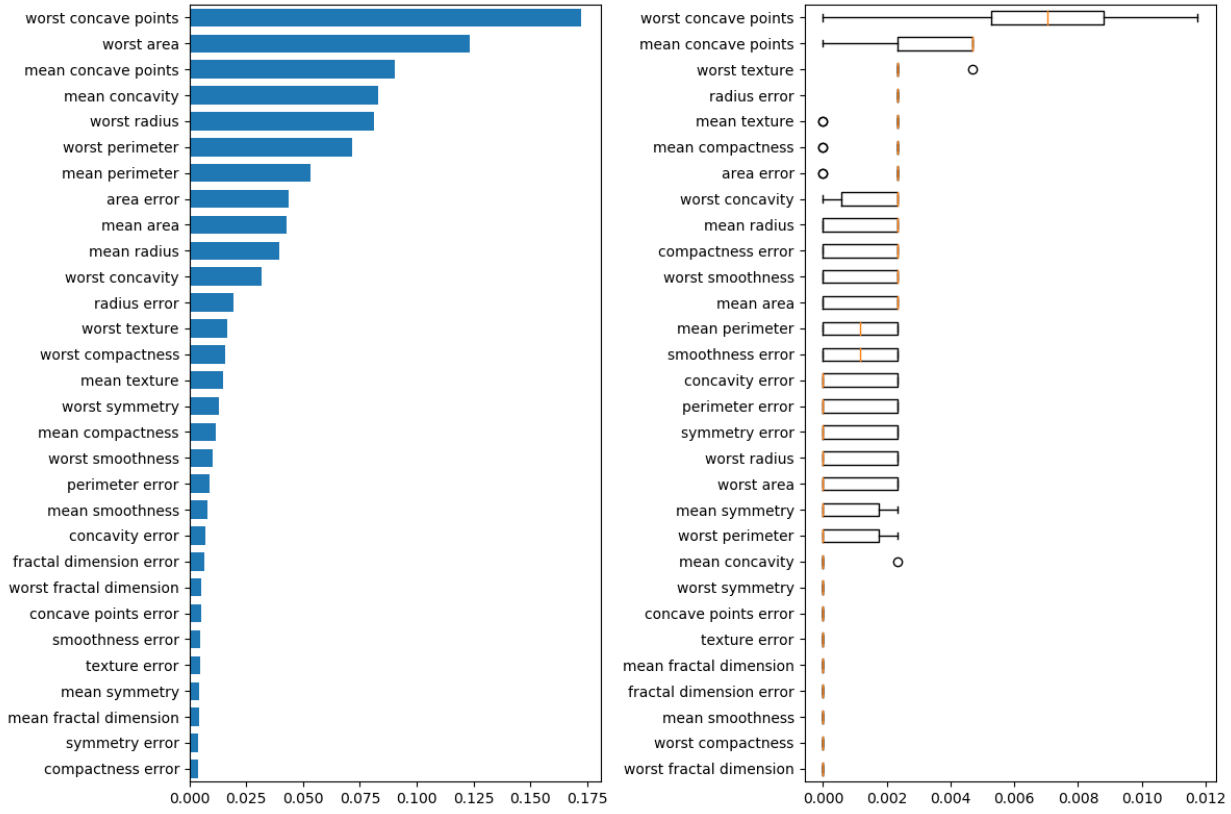
```
Accuracy on test data: 0.97
```

Next, we plot the tree based feature importance and the permutation importance. The permutation importance plot shows that permuting a feature drops the accuracy by at most `0.012`, which would suggest that none of the features are important. This is in contradiction with the high test accuracy computed above: some feature must be important. The permutation importance is calculated on the training set to show how much the model relies on each feature during training.

```
result = permutation_importance(clf, X_train, y_train, n_repeats=10,
                                random_state=42)
perm_sorted_idx = result.importances_mean.argsort()

tree_importance_sorted_idx = np.argsort(clf.feature_importances_)
tree_indices = np.arange(0, len(clf.feature_importances_)) + 0.5

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 8))
ax1.barh(tree_indices,
         clf.feature_importances_[tree_importance_sorted_idx], height=0.7)
ax1.set_yticklabels(data.feature_names[tree_importance_sorted_idx])
ax1.set_yticks(tree_indices)
ax1.set_ylim((0, len(clf.feature_importances_)))
ax2.boxplot(result.importances[perm_sorted_idx].T, vert=False,
            labels=data.feature_names[perm_sorted_idx])
fig.tight_layout()
plt.show()
```

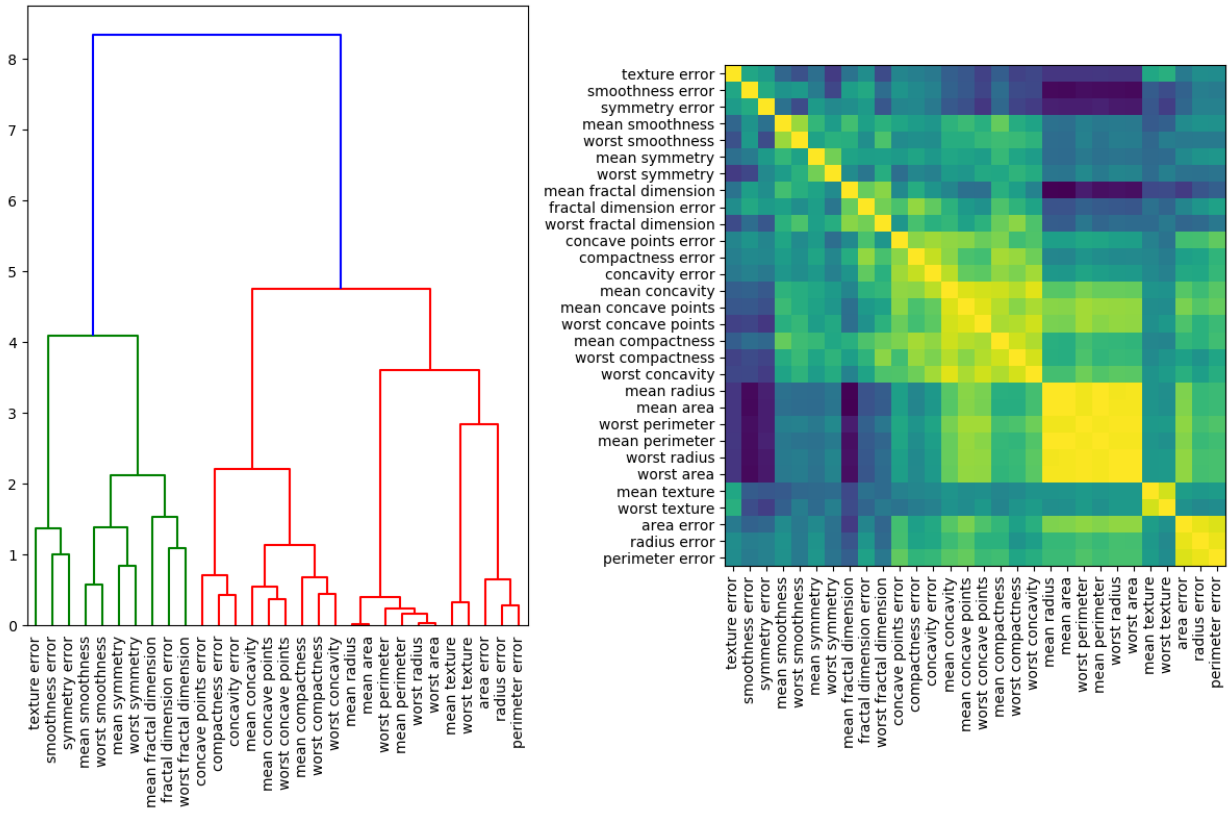


Handling Multicollinear Features

When features are collinear, permutating one feature will have little effect on the models performance because it can get the same information from a correlated feature. One way to handle multicollinear features is by performing hierarchical clustering on the Spearman rank-order correlations, picking a threshold, and keeping a single feature from each cluster. First, we plot a heatmap of the correlated features:

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 8))
corr = spearmanr(X).correlation
corr_linkage = hierarchy.ward(corr)
dendro = hierarchy.dendrogram(corr_linkage, labels=data.feature_names, ax=ax1,
                              leaf_rotation=90)
dendro_idx = np.arange(0, len(dendro['ivl']))

ax2.imshow(corr[dendro['leaves'], :][:, dendro['leaves']])
ax2.set_xticks(dendro_idx)
ax2.set_yticks(dendro_idx)
ax2.set_xticklabels(dendro['ivl'], rotation='vertical')
ax2.set_yticklabels(dendro['ivl'])
fig.tight_layout()
plt.show()
```



Next, we manually pick a threshold by visual inspection of the dendrogram to group our features into clusters and choose a feature from each cluster to keep, select those features from our dataset, and train a new random forest. The test accuracy of the new random forest did not change much compared to the random forest trained on the complete dataset.

```
cluster_ids = hierarchy.fcluster(corr_linkage, 1, criterion='distance')
cluster_id_to_feature_ids = defaultdict(list)
for idx, cluster_id in enumerate(cluster_ids):
    cluster_id_to_feature_ids[cluster_id].append(idx)
selected_features = [v[0] for v in cluster_id_to_feature_ids.values()]

X_train_sel = X_train[:, selected_features]
X_test_sel = X_test[:, selected_features]
```

(continues on next page)

(continued from previous page)

```
clf_sel = RandomForestClassifier(n_estimators=100, random_state=42)
clf_sel.fit(X_train_sel, y_train)
print("Accuracy on test data with features removed: {:.2f}".format(
      clf_sel.score(X_test_sel, y_test)))
```

Out:

```
Accuracy on test data with features removed: 0.97
```

Total running time of the script: (0 minutes 5.175 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.17.2 Permutation Importance vs Random Forest Feature Importance (MDI)

In this example, we will compare the impurity-based feature importance of `RandomForestClassifier` with the permutation importance on the titanic dataset using `permutation_importance`. We will show that the impurity-based feature importance can inflate the importance of numerical features.

Furthermore, the impurity-based feature importance of random forests suffers from being computed on statistics derived from the training dataset: the importances can be high even for features that are not predictive of the target variable, as long as the model has the capacity to use them to overfit.

This example shows how to use Permutation Importances as an alternative that can mitigate those limitations.

References:

[1] L. Breiman, "Random Forests", Machine Learning, 45(1), 5-32,

2001. https://doi.org/10.1023/A:1010933404324

```
print(__doc__)
import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import fetch_openml
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.inspection import permutation_importance
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
```

Data Loading and Feature Engineering

Let's use pandas to load a copy of the titanic dataset. The following shows how to apply separate preprocessing on numerical and categorical features.

We further include two random variables that are not correlated in any way with the target variable (`survived`):

- `random_num` is a high cardinality numerical variable (as many unique values as records).
- `random_cat` is a low cardinality categorical variable (3 possible values).

```
X, y = fetch_openml("titanic", version=1, as_frame=True, return_X_y=True)
rng = np.random.RandomState(seed=42)
X['random_cat'] = rng.randint(3, size=X.shape[0])
X['random_num'] = rng.randn(X.shape[0])

categorical_columns = ['pclass', 'sex', 'embarked', 'random_cat']
numerical_columns = ['age', 'sibsp', 'parch', 'fare', 'random_num']

X = X[categorical_columns + numerical_columns]

X_train, X_test, y_train, y_test = train_test_split(
    X, y, stratify=y, random_state=42)

categorical_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])
numerical_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='mean'))
])

preprocessing = ColumnTransformer(
    [('cat', categorical_pipe, categorical_columns),
     ('num', numerical_pipe, numerical_columns)])

rf = Pipeline([
    ('preprocess', preprocessing),
    ('classifier', RandomForestClassifier(random_state=42))
])
rf.fit(X_train, y_train)
```

Out:

```
Pipeline(steps=[('preprocess',
                 ColumnTransformer(transformers=[('cat',
                                                  Pipeline(steps=[('imputer',
                                                                   SimpleImputer(fill_
↪value='missing',
                                                                                 
↪strategy='constant')),
                                                                  ('onehot',
                                                                   
↪OneHotEncoder(handle_unknown='ignore'))]),
                                                  ['pclass', 'sex', 'embarked',
                                                   'random_cat']),
                                                 ('num',
                                                  Pipeline(steps=[('imputer',
                                                                   SimpleImputer())]),
                                                  ['age', 'sibsp', 'parch',
                                                   'fare', 'random_num'])])),
                ('classifier', RandomForestClassifier(random_state=42))])
```

Accuracy of the Model

Prior to inspecting the feature importances, it is important to check that the model predictive performance is high enough. Indeed there would be little interest of inspecting the important features of a non-predictive model.

Here one can observe that the train accuracy is very high (the forest model has enough capacity to completely memorize the training set) but it can still generalize well enough to the test set thanks to the built-in bagging of random forests.

It might be possible to trade some accuracy on the training set for a slightly better accuracy on the test set by limiting the capacity of the trees (for instance by setting `min_samples_leaf=5` or `min_samples_leaf=10`) so as to limit overfitting while not introducing too much underfitting.

However let's keep our high capacity random forest model for now so as to illustrate some pitfalls with feature importance on variables with many unique values.

```
print("RF train accuracy: %0.3f" % rf.score(X_train, y_train))
print("RF test accuracy: %0.3f" % rf.score(X_test, y_test))
```

Out:

```
RF train accuracy: 1.000
RF test accuracy: 0.817
```

Tree's Feature Importance from Mean Decrease in Impurity (MDI)

The impurity-based feature importance ranks the numerical features to be the most important features. As a result, the non-predictive `random_num` variable is ranked the most important!

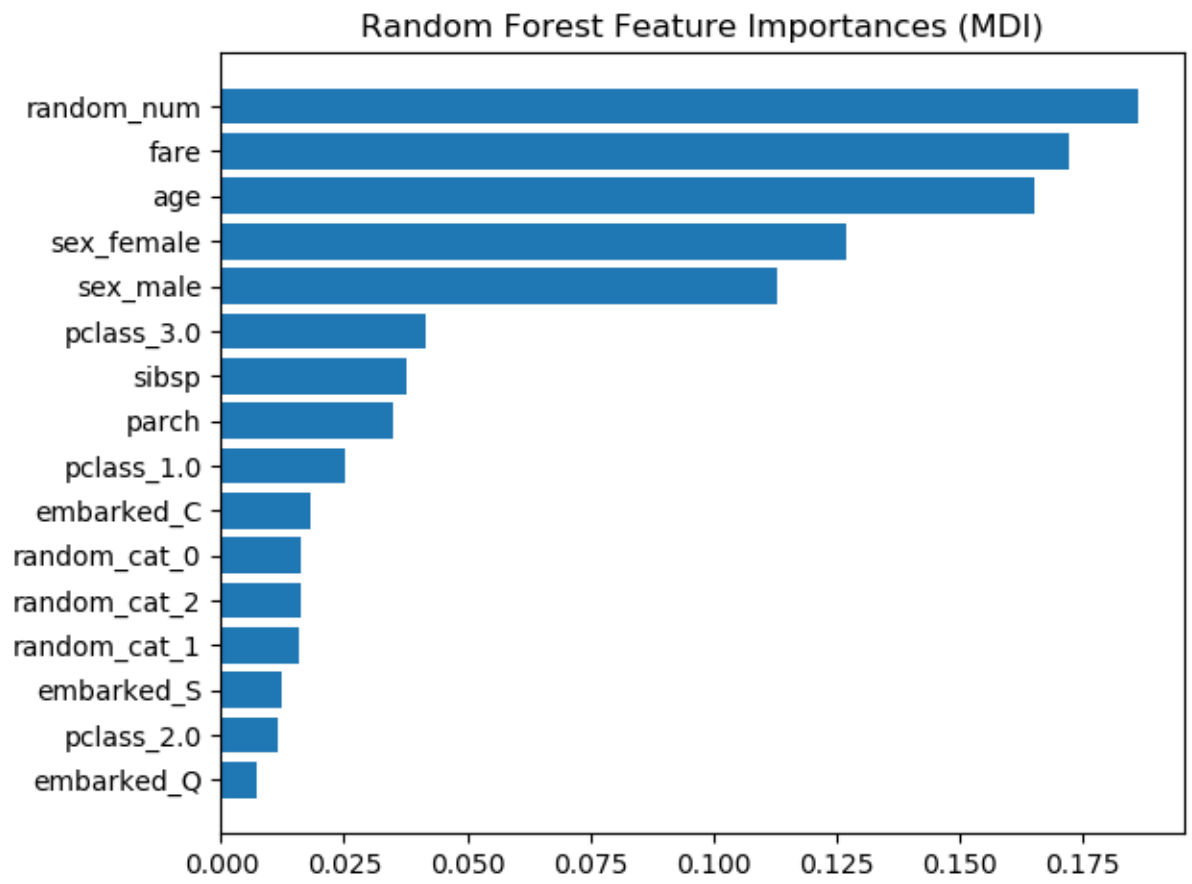
This problem stems from two limitations of impurity-based feature importances:

- impurity-based importances are biased towards high cardinality features;
- impurity-based importances are computed on training set statistics and therefore do not reflect the ability of feature to be useful to make predictions that generalize to the test set (when the model has enough capacity).

```
ohe = (rf.named_steps['preprocess']
         .named_transformers_['cat']
         .named_steps['onehot'])
feature_names = ohe.get_feature_names(input_features=categorical_columns)
feature_names = np.r_[feature_names, numerical_columns]

tree_feature_importances = (
    rf.named_steps['classifier'].feature_importances_)
sorted_idx = tree_feature_importances.argsort()

y_ticks = np.arange(0, len(feature_names))
fig, ax = plt.subplots()
ax.barh(y_ticks, tree_feature_importances[sorted_idx])
ax.set_yticklabels(feature_names[sorted_idx])
ax.set_yticks(y_ticks)
ax.set_title("Random Forest Feature Importances (MDI)")
fig.tight_layout()
plt.show()
```

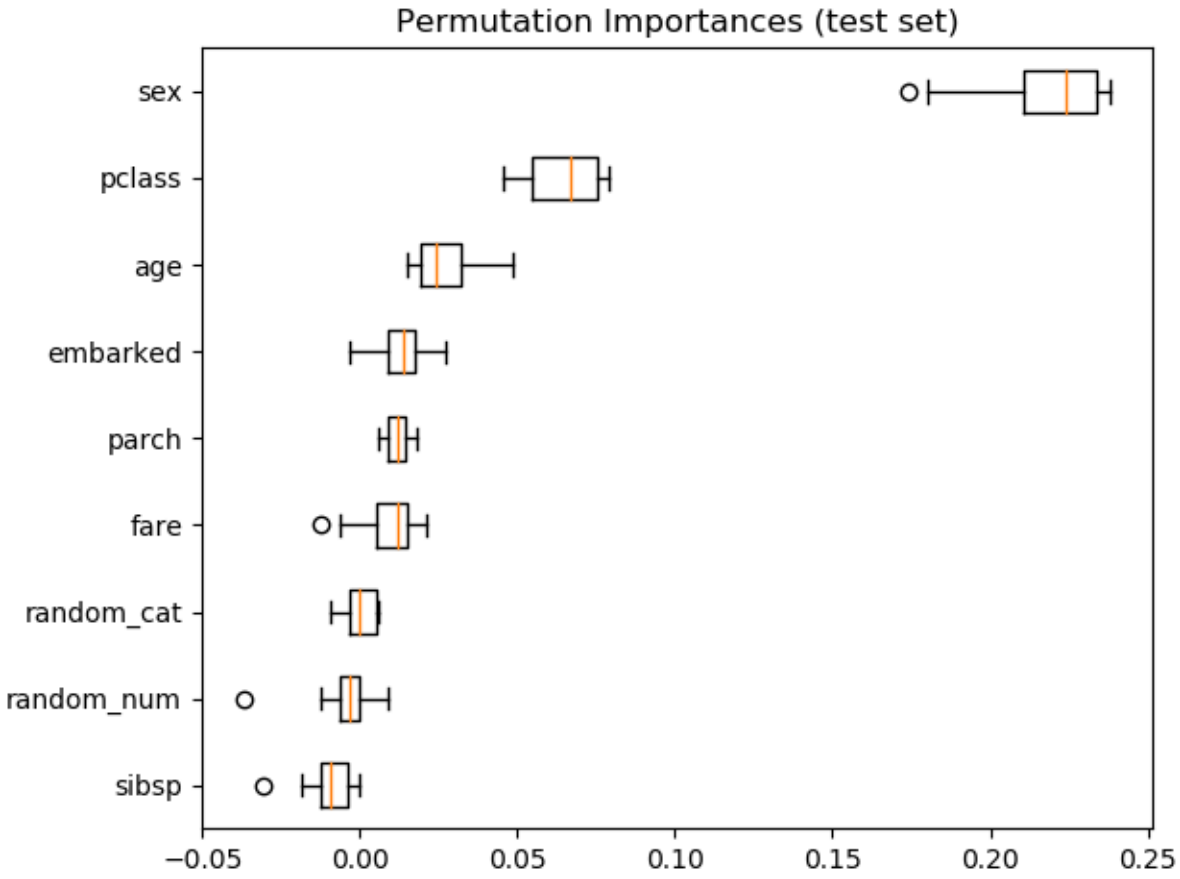


As an alternative, the permutation importances of `rf` are computed on a held out test set. This shows that the low cardinality categorical feature, `sex` is the most important feature.

Also note that both random features have very low importances (close to 0) as expected.

```
result = permutation_importance(rf, X_test, y_test, n_repeats=10,
                                random_state=42, n_jobs=2)
sorted_idx = result.importances_mean.argsort()

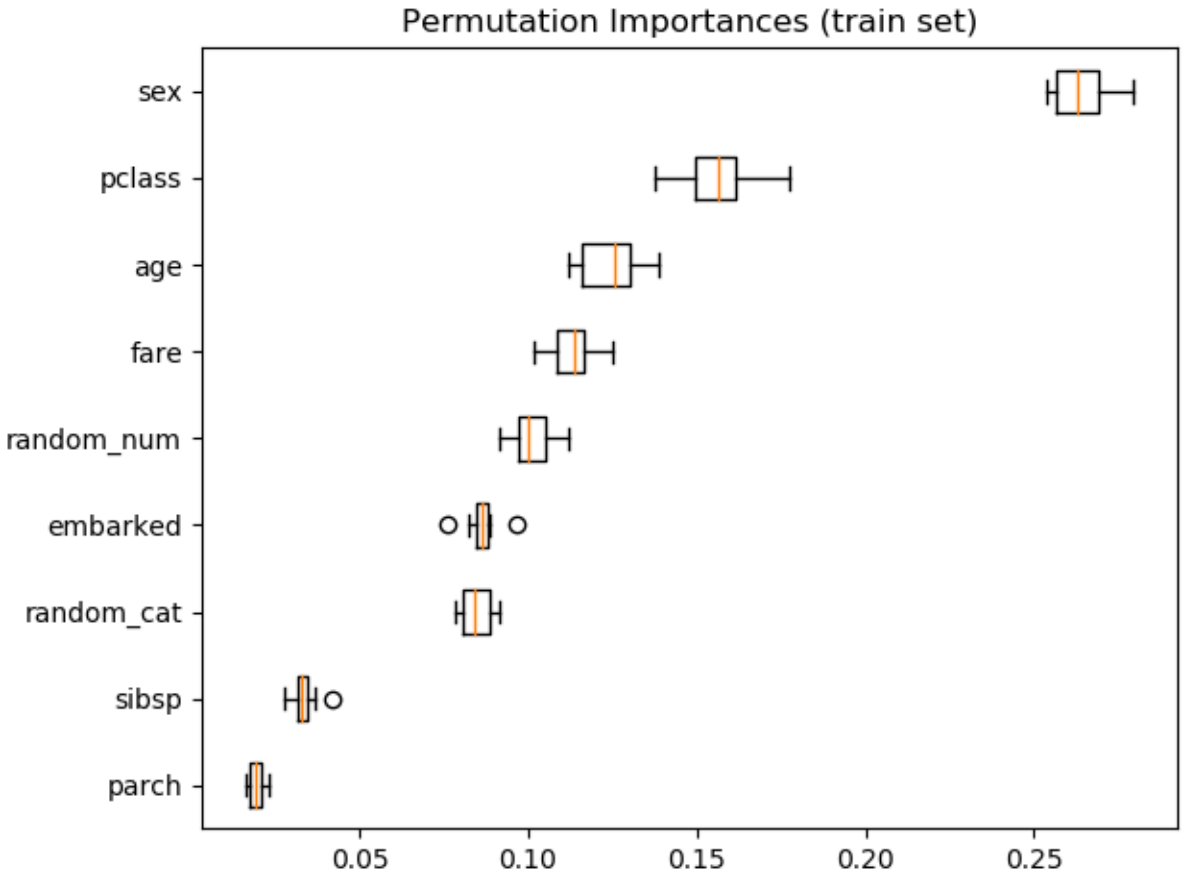
fig, ax = plt.subplots()
ax.boxplot(result.importances[sorted_idx].T,
           vert=False, labels=X_test.columns[sorted_idx])
ax.set_title("Permutation Importances (test set)")
fig.tight_layout()
plt.show()
```



It is also possible to compute the permutation importances on the training set. This reveals that `random_num` gets a significantly higher importance ranking than when computed on the test set. The difference between those two plots is a confirmation that the RF model has enough capacity to use that random numerical feature to overfit. You can further confirm this by re-running this example with constrained RF with min_samples_leaf=10.

```
result = permutation_importance(rf, X_train, y_train, n_repeats=10,
                                random_state=42, n_jobs=2)
sorted_idx = result.importances_mean.argsort()

fig, ax = plt.subplots()
ax.boxplot(result.importances[sorted_idx].T,
           vert=False, labels=X_train.columns[sorted_idx])
ax.set_title("Permutation Importances (train set)")
fig.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 5.346 seconds)

Estimated memory usage: 74 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.17.3 Partial Dependence Plots

Partial dependence plots show the dependence between the target function[2] and a set of 'target' features, marginalizing over the values of all other features (the complement features). Due to the limits of human perception, the size of the target feature set must be small (usually, one or two) thus the target features are usually chosen among the most important features.

This example shows how to obtain partial dependence plots from a `MLPRegressor` and a `HistGradientBoostingRegressor` trained on the California housing dataset. The example is taken from[1].

The plots show four 1-way and two 1-way partial dependence plots (omitted for `MLPRegressor` due to computation time). The target variables for the one-way PDP are: median income (`MedInc`), average occupants per household (`AvgOccup`), median house age (`HouseAge`), and average rooms per household (`AveRooms`).

[2] For classification you can think of it as the regression score before the link function.

[1] T. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning Ed. 2", Springer, 2009.

```
print(__doc__)

from time import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import QuantileTransformer
from sklearn.pipeline import make_pipeline

from sklearn.inspection import partial_dependence
from sklearn.inspection import plot_partial_dependence
from sklearn.experimental import enable_hist_gradient_boosting  # noqa
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.datasets import fetch_california_housing
```

California Housing data preprocessing

Center target to avoid gradient boosting init bias: gradient boosting with the 'recursion' method does not account for the initial estimator (here the average target, by default)

```
cal_housing = fetch_california_housing()
X = pd.DataFrame(cal_housing.data, columns=cal_housing.feature_names)
y = cal_housing.target

y -= y.mean()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1,
                                                    random_state=0)
```

Partial Dependence computation for multi-layer perceptron

Let's fit a MLPRegressor and compute single-variable partial dependence plots

```
print("Training MLPRegressor...")
tic = time()
est = make_pipeline(QuantileTransformer(),
                    MLPRegressor(hidden_layer_sizes=(50, 50),
                                 learning_rate_init=0.01,
                                 early_stopping=True))
est.fit(X_train, y_train)
print("done in {:.3f}s".format(time() - tic))
print("Test R2 score: {:.2f}".format(est.score(X_test, y_test)))
```

Out:

```
Training MLPRegressor...
done in 4.017s
Test R2 score: 0.81
```

We configured a pipeline to scale the numerical input features and tuned the neural network size and learning rate to get a reasonable compromise between training time and predictive performance on a test set.

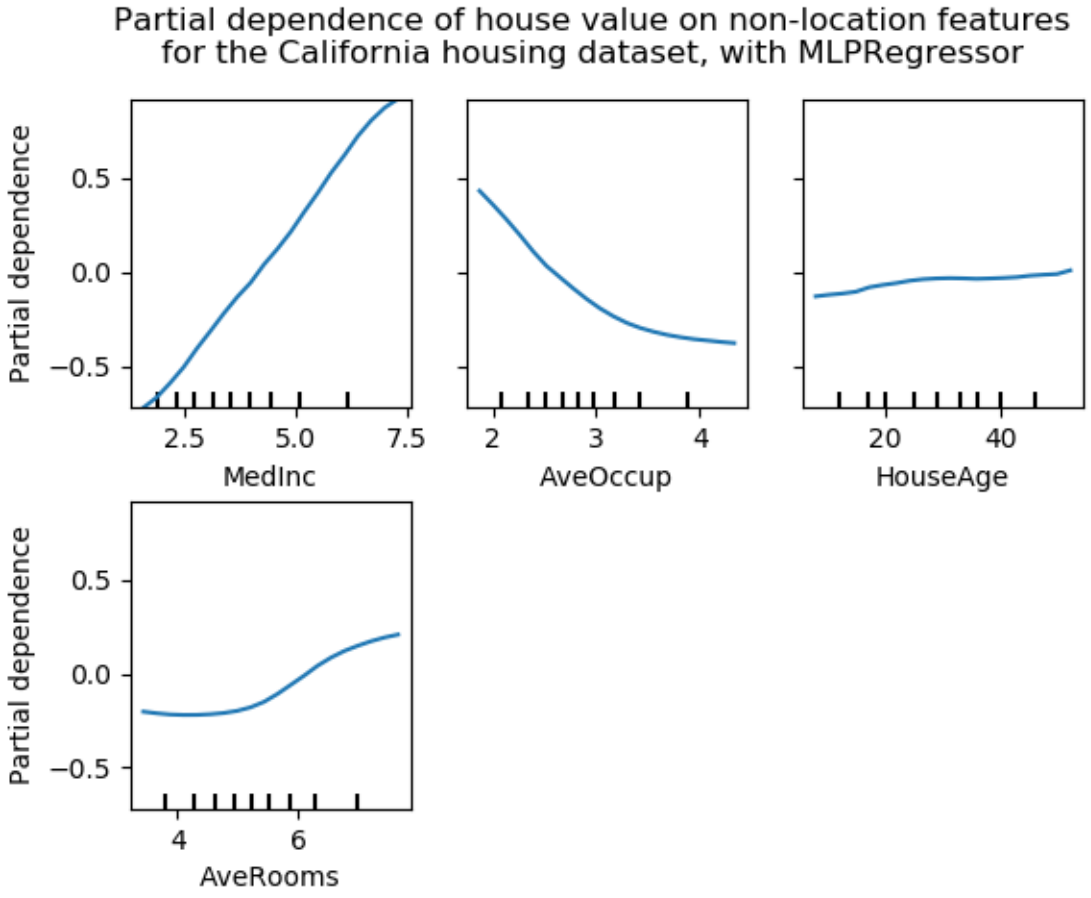
Importantly, this tabular dataset has very different dynamic ranges for its features. Neural networks tend to be very sensitive to features with varying scales and forgetting to preprocess the numeric feature would lead to a very poor model.

It would be possible to get even higher predictive performance with a larger neural network but the training would also be significantly more expensive.

Note that it is important to check that the model is accurate enough on a test set before plotting the partial dependence since there would be little use in explaining the impact of a given feature on the prediction function of a poor model.

Let's now compute the partial dependence plots for this neural network using the model-agnostic (brute-force) method:

```
print('Computing partial dependence plots...')
tic = time()
# We don't compute the 2-way PDP (5, 1) here, because it is a lot slower
# with the brute method.
features = ['MedInc', 'AveOccup', 'HouseAge', 'AveRooms']
plot_partial_dependence(est, X_train, features,
                        n_jobs=3, grid_resolution=20)
print("done in {:.3f}s".format(time() - tic))
fig = plt.gcf()
fig.suptitle('Partial dependence of house value on non-location features\n'
             'for the California housing dataset, with MLPRegressor')
fig.subplots_adjust(hspace=0.3)
```



Out:

```
Computing partial dependence plots...
done in 2.134s
```

Partial Dependence computation for Gradient Boosting

Let's now fit a GradientBoostingRegressor and compute the partial dependence plots either or one or two variables at a time.

```
print("Training GradientBoostingRegressor...")
tic = time()
est = HistGradientBoostingRegressor()
est.fit(X_train, y_train)
print("done in {:.3f}s".format(time() - tic))
print("Test R2 score: {:.2f}".format(est.score(X_test, y_test)))
```

Out:

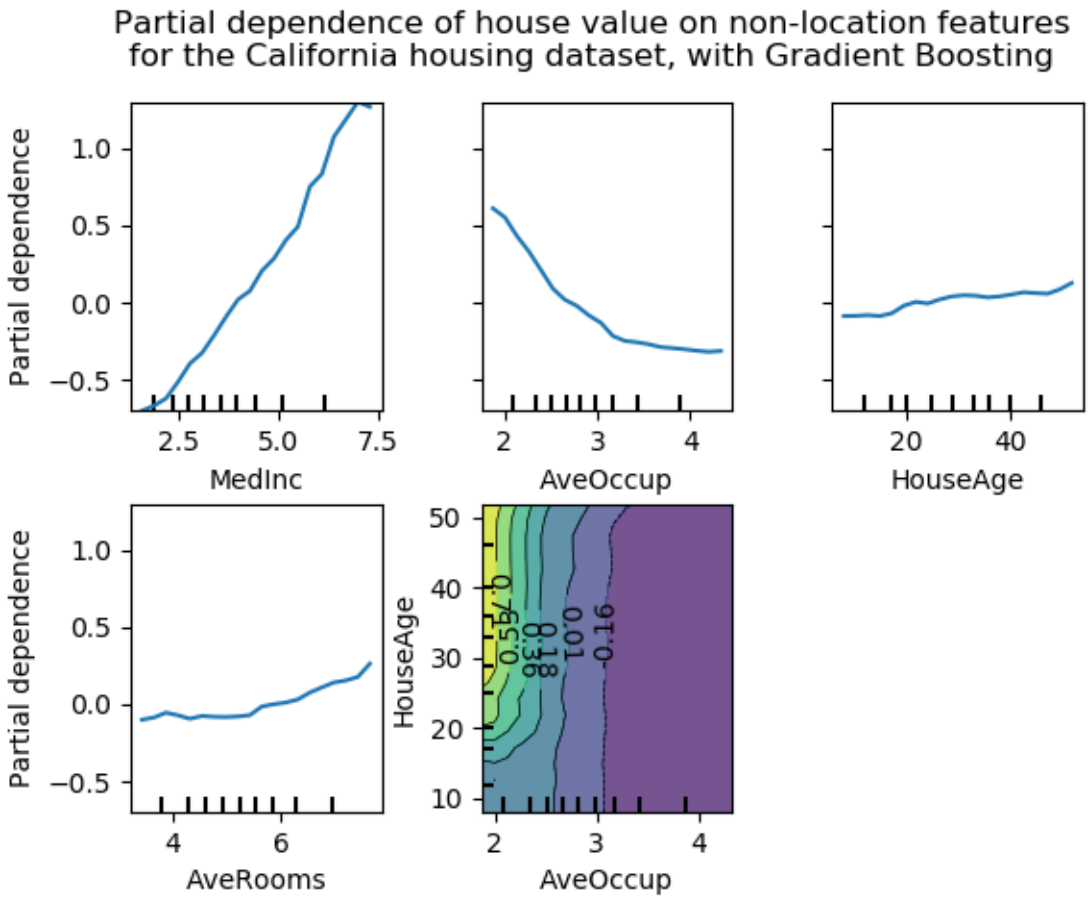
```
Training GradientBoostingRegressor...
done in 0.388s
Test R2 score: 0.85
```

Here, we used the default hyperparameters for the gradient boosting model without any preprocessing as tree-based models are naturally robust to monotonic transformations of numerical features.

Note that on this tabular dataset, Gradient Boosting Machines are both significantly faster to train and more accurate than neural networks. It is also significantly cheaper to tune their hyperparameters (the default tend to work well while this is not often the case for neural networks).

Finally, as we will see next, computing partial dependence plots tree-based models is also orders of magnitude faster making it cheap to compute partial dependence plots for pairs of interacting features:

```
print('Computing partial dependence plots...')
tic = time()
features = ['MedInc', 'AveOccup', 'HouseAge', 'AveRooms',
            ('AveOccup', 'HouseAge')]
plot_partial_dependence(est, X_train, features,
                        n_jobs=3, grid_resolution=20)
print("done in {:.3f}s".format(time() - tic))
fig = plt.gcf()
fig.suptitle('Partial dependence of house value on non-location features\n'
             'for the California housing dataset, with Gradient Boosting')
fig.subplots_adjust(wspace=0.4, hspace=0.3)
```



Out:

```
Computing partial dependence plots...
done in 0.130s
```

Analysis of the plots

We can clearly see that the median house price shows a linear relationship with the median income (top left) and that the house price drops when the average occupants per household increases (top middle). The top right plot shows that the house age in a district does not have a strong influence on the (median) house price; so does the average rooms per household. The tick marks on the x-axis represent the deciles of the feature values in the training data.

We also observe that *MLPRegressor* has much smoother predictions than *HistGradientBoostingRegressor*. For the plots to be comparable, it is necessary to subtract the average value of the target `y`: The 'recursion' method, used by default for *HistGradientBoostingRegressor*, does not account for the initial predictor (in our case the average target). Setting the target average to 0 avoids this bias.

Partial dependence plots with two target features enable us to visualize interactions among them. The two-way partial dependence plot shows the dependence of median house price on joint values of house age and average occupants per household. We can clearly see an interaction between the two features: for an average occupancy greater than two, the house price is nearly independent of the house age, whereas for values less than two there is a strong dependence on age.

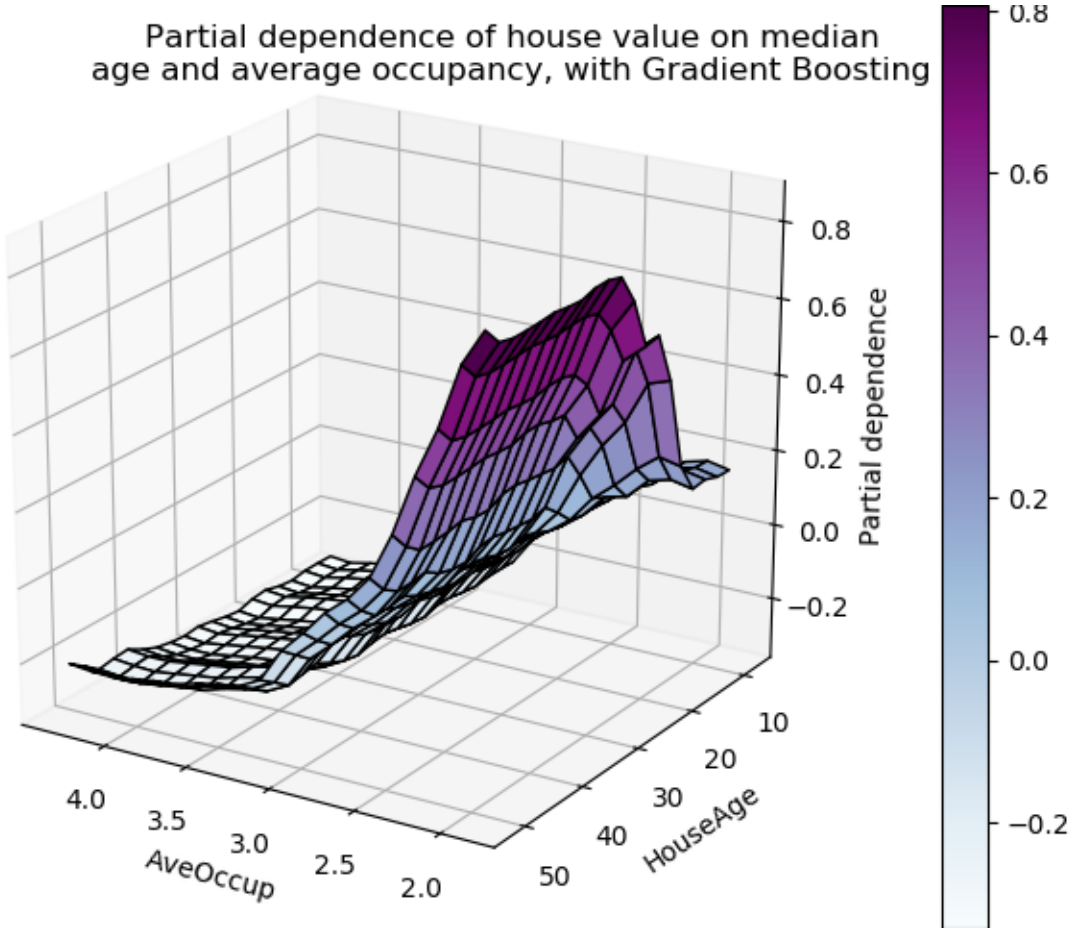
3D interaction plots

Let's make the same partial dependence plot for the 2 features interaction, this time in 3 dimensions.

```
fig = plt.figure()

features = ('AveOccup', 'HouseAge')
pdp, axes = partial_dependence(est, X_train, features=features,
                               grid_resolution=20)
XX, YY = np.meshgrid(axes[0], axes[1])
Z = pdp[0].T
ax = Axes3D(fig)
surf = ax.plot_surface(XX, YY, Z, rstride=1, cstride=1,
                       cmap=plt.cm.BuPu, edgecolor='k')
ax.set_xlabel(features[0])
ax.set_ylabel(features[1])
ax.set_zlabel('Partial dependence')
#  pretty init view
ax.view_init(elev=22, azim=122)
plt.colorbar(surf)
plt.suptitle('Partial dependence of house value on median\n'
             'age and average occupancy, with Gradient Boosting')
plt.subplots_adjust(top=0.9)

plt.show()
```



Total running time of the script: (0 minutes 8.994 seconds)

Estimated memory usage: 9 MB

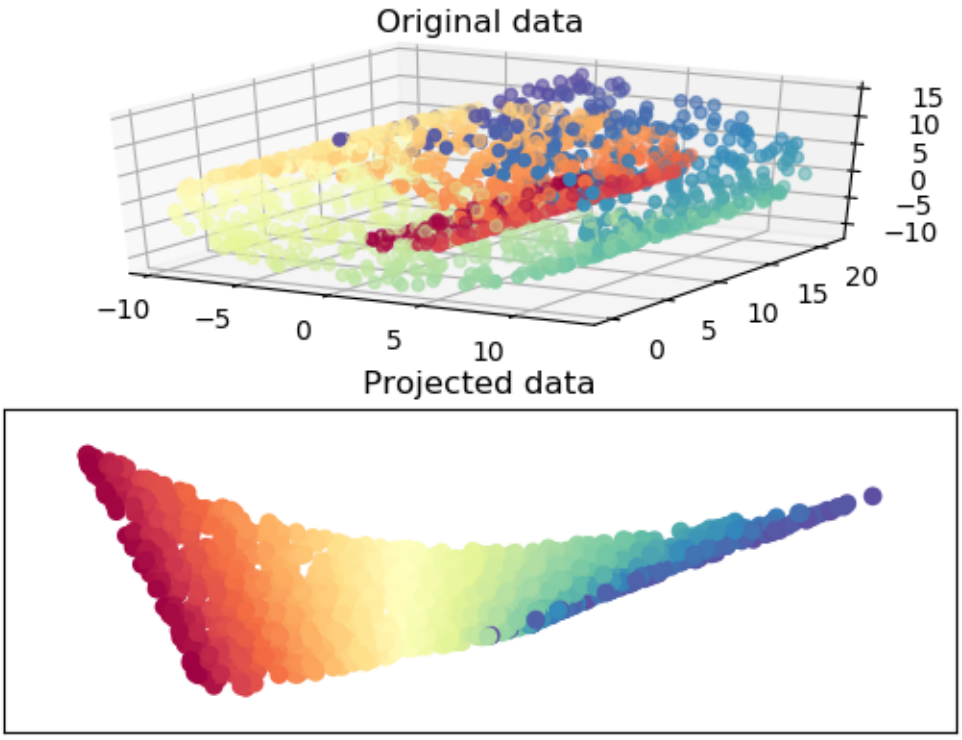
6.18 Manifold learning

Examples concerning the `sklearn.manifold` module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.18.1 Swiss Roll reduction with LLE

An illustration of Swiss Roll reduction with locally linear embedding



Out:

```
Computing LLE embedding
Done. Reconstruction error: 1.27445e-07
```

```
# Author: Fabian Pedregosa -- <fabian.pedregosa@inria.fr>
# License: BSD 3 clause (C) INRIA 2011

print(__doc__)

import matplotlib.pyplot as plt

# This import is needed to modify the way figure behaves
from mpl_toolkits.mplot3d import Axes3D
Axes3D

#----------------------------------------------------------------------
# Locally linear embedding of the swiss roll

from sklearn import manifold, datasets
X, color = datasets.make_swiss_roll(n_samples=1500)

print("Computing LLE embedding")
X_r, err = manifold.locally_linear_embedding(X, n_neighbors=12,
                                             n_components=2)
print("Done. Reconstruction error: %g" % err)

#----------------------------------------------------------------------
# Plot result

fig = plt.figure()

ax = fig.add_subplot(211, projection='3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral)

ax.set_title("Original data")
ax = fig.add_subplot(212)
ax.scatter(X_r[:, 0], X_r[:, 1], c=color, cmap=plt.cm.Spectral)
plt.axis('tight')
plt.xticks([]), plt.yticks([])
plt.title('Projected data')
plt.show()
```

Total running time of the script: (0 minutes 0.723 seconds)

Estimated memory usage: 9 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

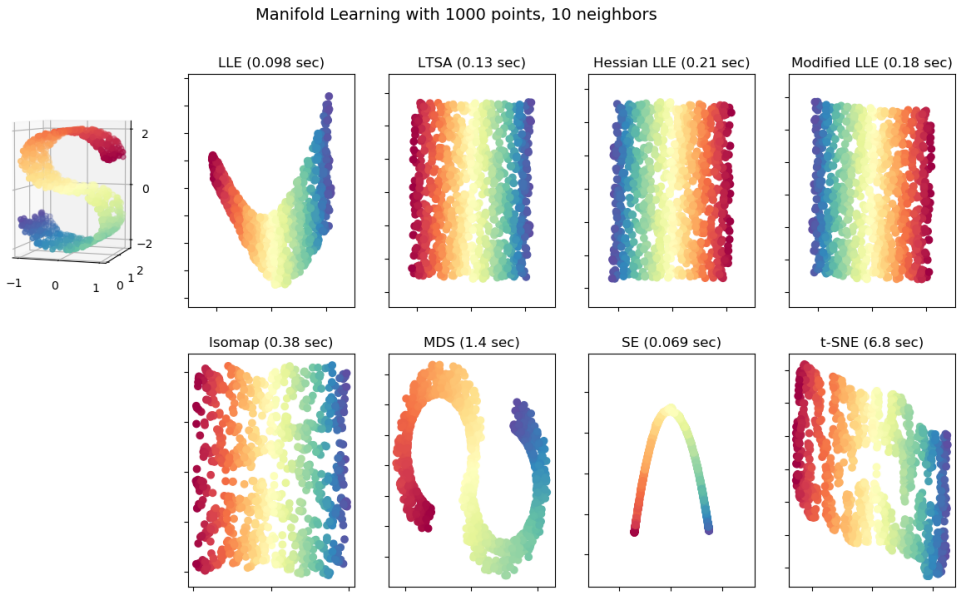
6.18.2 Comparison of Manifold Learning methods

An illustration of dimensionality reduction on the S-curve dataset with various manifold learning methods.

For a discussion and comparison of these algorithms, see the *manifold module page*

For a similar example, where the methods are applied to a sphere dataset, see *Manifold Learning methods on a severed sphere*

Note that the purpose of the MDS is to find a low-dimensional representation of the data (here 2D) in which the distances respect well the distances in the original high-dimensional space, unlike other manifold-learning algorithms, it does not seeks an isotropic representation of the data in the low-dimensional space.



Out:

```
LLE: 0.098 sec
LTSA: 0.13 sec
Hessian LLE: 0.21 sec
Modified LLE: 0.18 sec
Isomap: 0.38 sec
MDS: 1.4 sec
SE: 0.069 sec
t-SNE: 6.8 sec
```

```
# Author: Jake Vanderplas -- <vanderplas@astro.washington.edu>

print(__doc__)

from collections import OrderedDict
from functools import partial
from time import time

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import NullFormatter

from sklearn import manifold, datasets

# Next line to silence pyflakes. This import is needed.
Axes3D

n_points = 1000
```

(continues on next page)

(continued from previous page)

```
X, color = datasets.make_s_curve(n_points, random_state=0)
n_neighbors = 10
n_components = 2

# Create figure
fig = plt.figure(figsize=(15, 8))
fig.suptitle("Manifold Learning with %i points, %i neighbors"
             % (1000, n_neighbors), fontsize=14)

# Add 3d scatter plot
ax = fig.add_subplot(251, projection='3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral)
ax.view_init(4, -72)

# Set-up manifold methods
LLE = partial(manifold.LocallyLinearEmbedding,
              n_neighbors, n_components, eigen_solver='auto')

methods = OrderedDict()
methods['LLE'] = LLE(method='standard')
methods['LTSA'] = LLE(method='ltsa')
methods['Hessian LLE'] = LLE(method='hessian')
methods['Modified LLE'] = LLE(method='modified')
methods['Isomap'] = manifold.Isomap(n_neighbors, n_components)
methods['MDS'] = manifold.MDS(n_components, max_iter=100, n_init=1)
methods['SE'] = manifold.SpectralEmbedding(n_components=n_components,
                                           n_neighbors=n_neighbors)
methods['t-SNE'] = manifold.TSNE(n_components=n_components, init='pca',
                                 random_state=0)

# Plot results
for i, (label, method) in enumerate(methods.items()):
    t0 = time()
    Y = method.fit_transform(X)
    t1 = time()
    print("%s: %.2g sec" % (label, t1 - t0))
    ax = fig.add_subplot(2, 5, 2 + i + (i > 3))
    ax.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
    ax.set_title("%s (%.2g sec)" % (label, t1 - t0))
    ax.xaxis.set_major_formatter(NullFormatter())
    ax.yaxis.set_major_formatter(NullFormatter())
    ax.axis('tight')

plt.show()
```

Total running time of the script: (0 minutes 9.983 seconds)

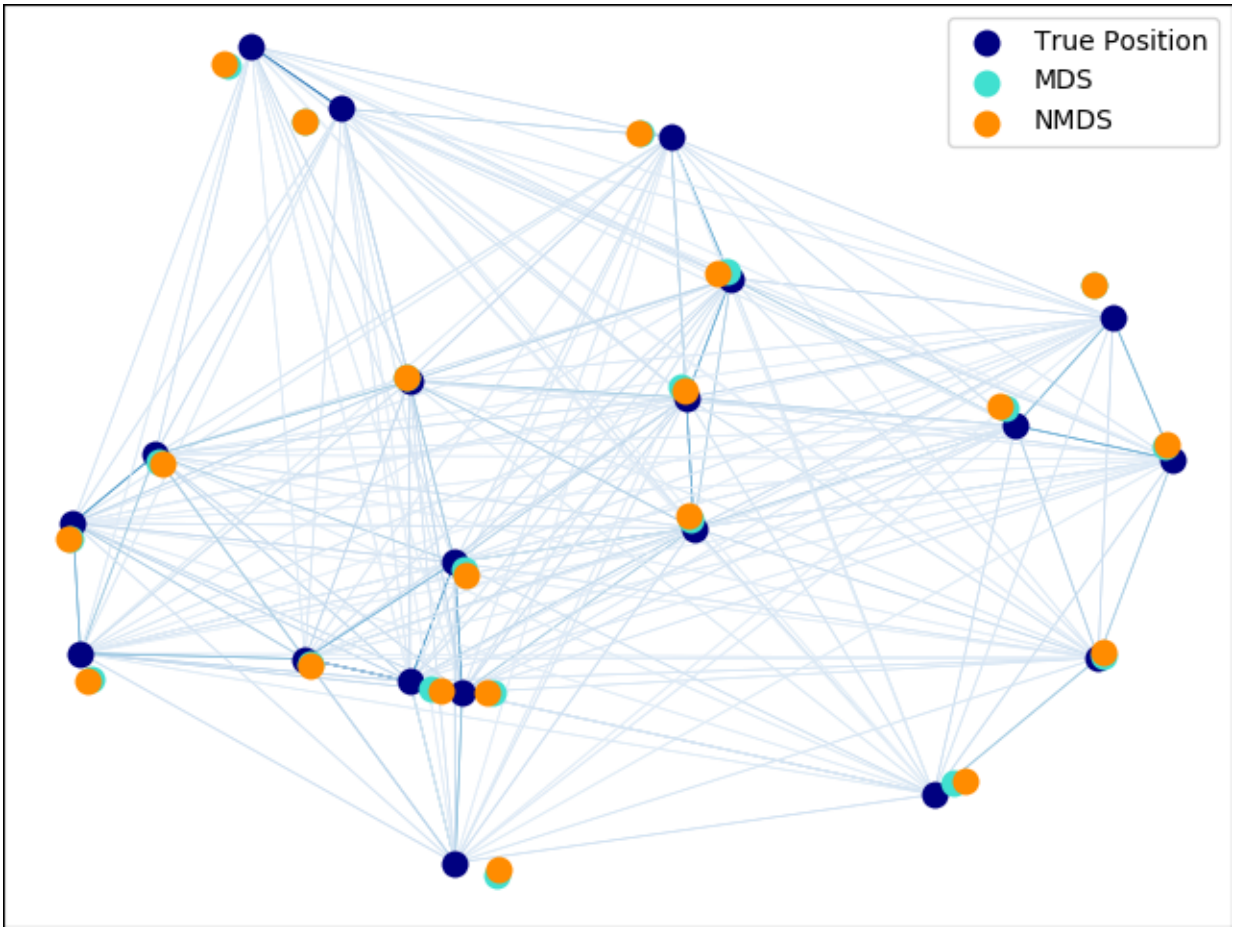
Estimated memory usage: 53 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.18.3 Multi-dimensional scaling

An illustration of the metric and non-metric MDS on generated noisy data.

The reconstructed points using the metric MDS and non metric MDS are slightly shifted to avoid overlapping.




```
# Author: Nelle Varoquaux <nelle.varoquaux@gmail.com>
# License: BSD

print(__doc__)
import numpy as np

from matplotlib import pyplot as plt
from matplotlib.collections import LineCollection

from sklearn import manifold
from sklearn.metrics import euclidean_distances
from sklearn.decomposition import PCA

EPSILON = np.finfo(np.float32).eps
n_samples = 20
seed = np.random.RandomState(seed=3)
X_true = seed.randint(0, 20, 2 * n_samples).astype(np.float)
X_true = X_true.reshape((n_samples, 2))
# Center the data
X_true -= X_true.mean()

similarities = euclidean_distances(X_true)

# Add noise to the similarities
```

(continues on next page)

(continued from previous page)

```
noise = np.random.rand(n_samples, n_samples)
noise = noise + noise.T
noise[np.arange(noise.shape[0]), np.arange(noise.shape[0])] = 0
similarities += noise

mds = manifold.MDS(n_components=2, max_iter=3000, eps=1e-9, random_state=seed,
                   dissimilarity="precomputed", n_jobs=1)
pos = mds.fit(similarities).embedding_

nmds = manifold.MDS(n_components=2, metric=False, max_iter=3000, eps=1e-12,
                    dissimilarity="precomputed", random_state=seed, n_jobs=1,
                    n_init=1)
npos = nmds.fit_transform(similarities, init=pos)

# Rescale the data
pos *= np.sqrt((X_true ** 2).sum()) / np.sqrt((pos ** 2).sum())
npos *= np.sqrt((X_true ** 2).sum()) / np.sqrt((npos ** 2).sum())

# Rotate the data
clf = PCA(n_components=2)
X_true = clf.fit_transform(X_true)

pos = clf.fit_transform(pos)

npos = clf.fit_transform(npos)

fig = plt.figure(1)
ax = plt.axes([0., 0., 1., 1.])

s = 100
plt.scatter(X_true[:, 0], X_true[:, 1], color='navy', s=s, lw=0,
            label='True Position')
plt.scatter(pos[:, 0], pos[:, 1], color='turquoise', s=s, lw=0, label='MDS')
plt.scatter(npos[:, 0], npos[:, 1], color='darkorange', s=s, lw=0, label='NMDS')
plt.legend(scatterpoints=1, loc='best', shadow=False)

similarities = similarities.max() / (similarities + EPSILON) * 100
np.fill_diagonal(similarities, 0)
# Plot the edges
start_idx, end_idx = np.where(pos)
# a sequence of (*line0*, *line1*, *line2*), where::
#            linen = (x0, y0), (x1, y1), ... (xm, ym)
segments = [[X_true[i, :], X_true[j, :]]
            for i in range(len(pos)) for j in range(len(pos))]
values = np.abs(similarities)
lc = LineCollection(segments,
                    zorder=0, cmap=plt.cm.Blues,
                    norm=plt.Normalize(0, values.max()))
lc.set_array(similarities.flatten())
lc.set_linewidths(np.full(len(segments), 0.5))
ax.add_collection(lc)

plt.show()
```

Total running time of the script: (0 minutes 0.570 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.18.4 t-SNE: The effect of various perplexity values on the shape

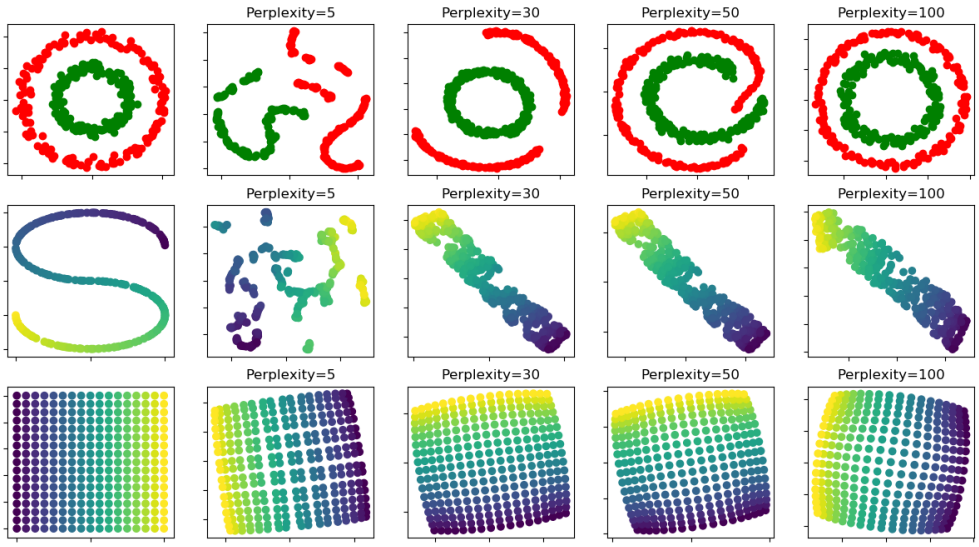
An illustration of t-SNE on the two concentric circles and the S-curve datasets for different perplexity values.

We observe a tendency towards clearer shapes as the perplexity value increases.

The size, the distance and the shape of clusters may vary upon initialization, perplexity values and does not always convey a meaning.

As shown below, t-SNE for higher perplexities finds meaningful topology of two concentric circles, however the size and the distance of the circles varies slightly from the original. Contrary to the two circles dataset, the shapes visually diverge from S-curve topology on the S-curve dataset even for larger perplexity values.

For further details, "How to Use t-SNE Effectively" https://distill.pub/2016/misread-tsne/ provides a good discussion of the effects of various parameters, as well as interactive plots to explore those effects.



Out:

```
circles, perplexity=5 in 0.95 sec
circles, perplexity=30 in 1.4 sec
circles, perplexity=50 in 1.6 sec
circles, perplexity=100 in 2 sec
S-curve, perplexity=5 in 1 sec
S-curve, perplexity=30 in 1.3 sec
S-curve, perplexity=50 in 1.1 sec
S-curve, perplexity=100 in 1.6 sec
uniform grid, perplexity=5 in 0.9 sec
uniform grid, perplexity=30 in 1.2 sec
uniform grid, perplexity=50 in 1.1 sec
uniform grid, perplexity=100 in 1.5 sec
```

```
# Author: Narine Kokhlikyan <narine@slice.com>
# License: BSD

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from matplotlib.ticker import NullFormatter
from sklearn import manifold, datasets
from time import time

n_samples = 300
n_components = 2
(fig, subplots) = plt.subplots(3, 5, figsize=(15, 8))
perplexities = [5, 30, 50, 100]

X, y = datasets.make_circles(n_samples=n_samples, factor=.5, noise=.05)

red = y == 0
green = y == 1

ax = subplots[0][0]
ax.scatter(X[red, 0], X[red, 1], c="r")
ax.scatter(X[green, 0], X[green, 1], c="g")
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

for i, perplexity in enumerate(perplexities):
    ax = subplots[0][i + 1]

    t0 = time()
    tsne = manifold.TSNE(n_components=n_components, init='random',
                         random_state=0, perplexity=perplexity)
    Y = tsne.fit_transform(X)
    t1 = time()
    print("circles, perplexity=%d in %.2g sec" % (perplexity, t1 - t0))
    ax.set_title("Perplexity=%d" % perplexity)
    ax.scatter(Y[red, 0], Y[red, 1], c="r")
    ax.scatter(Y[green, 0], Y[green, 1], c="g")
    ax.xaxis.set_major_formatter(NullFormatter())
    ax.yaxis.set_major_formatter(NullFormatter())
    ax.axis('tight')

# Another example using s-curve
X, color = datasets.make_s_curve(n_samples, random_state=0)

ax = subplots[1][0]
ax.scatter(X[:, 0], X[:, 2], c=color)
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())

for i, perplexity in enumerate(perplexities):
```

(continues on next page)

(continued from previous page)

```
    ax = subplots[1][i + 1]

    t0 = time()
    tsne = manifold.TSNE(n_components=n_components, init='random',
                         random_state=0, perplexity=perplexity)
    Y = tsne.fit_transform(X)
    t1 = time()
    print("S-curve, perplexity=%d in %.2g sec" % (perplexity, t1 - t0))

    ax.set_title("Perplexity=%d" % perplexity)
    ax.scatter(Y[:, 0], Y[:, 1], c=color)
    ax.xaxis.set_major_formatter(NullFormatter())
    ax.yaxis.set_major_formatter(NullFormatter())
    ax.axis('tight')


# Another example using a 2D uniform grid
x = np.linspace(0, 1, int(np.sqrt(n_samples)))
xx, yy = np.meshgrid(x, x)
X = np.hstack([
    xx.ravel().reshape(-1, 1),
    yy.ravel().reshape(-1, 1),
])
color = xx.ravel()
ax = subplots[2][0]
ax.scatter(X[:, 0], X[:, 1], c=color)
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())

for i, perplexity in enumerate(perplexities):
    ax = subplots[2][i + 1]

    t0 = time()
    tsne = manifold.TSNE(n_components=n_components, init='random',
                         random_state=0, perplexity=perplexity)
    Y = tsne.fit_transform(X)
    t1 = time()
    print("uniform grid, perplexity=%d in %.2g sec" % (perplexity, t1 - t0))

    ax.set_title("Perplexity=%d" % perplexity)
    ax.scatter(Y[:, 0], Y[:, 1], c=color)
    ax.xaxis.set_major_formatter(NullFormatter())
    ax.yaxis.set_major_formatter(NullFormatter())
    ax.axis('tight')


plt.show()
```

Total running time of the script: (0 minutes 16.344 seconds)

Estimated memory usage: 8 MB

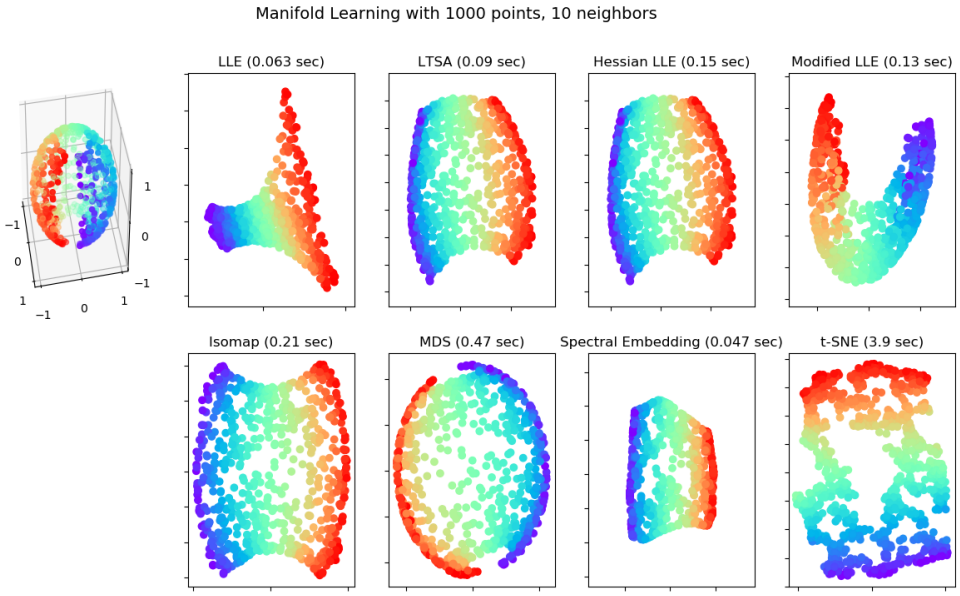
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.18.5 Manifold Learning methods on a severed sphere

An application of the different *Manifold learning* techniques on a spherical data-set. Here one can see the use of dimensionality reduction in order to gain some intuition regarding the manifold learning methods. Regarding the dataset, the poles are cut from the sphere, as well as a thin slice down its side. This enables the manifold learning techniques to 'spread it open' whilst projecting it onto two dimensions.

For a similar example, where the methods are applied to the S-curve dataset, see *Comparison of Manifold Learning methods*

Note that the purpose of the *MDS* is to find a low-dimensional representation of the data (here 2D) in which the distances respect well the distances in the original high-dimensional space, unlike other manifold-learning algorithms, it does not seeks an isotropic representation of the data in the low-dimensional space. Here the manifold problem matches fairly that of representing a flat map of the Earth, as with map projection



Out:

```
standard: 0.063 sec
ltsa: 0.09 sec
hessian: 0.15 sec
modified: 0.13 sec
ISO: 0.21 sec
MDS: 0.47 sec
Spectral Embedding: 0.047 sec
t-SNE: 3.9 sec
```

```
# Author: Jaques Grobler <jaques.grobler@inria.fr>
# License: BSD 3 clause
```

(continues on next page)

(continued from previous page)

```
print(__doc__)

from time import time

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import NullFormatter

from sklearn import manifold
from sklearn.utils import check_random_state

# Next line to silence pyflakes.
Axes3D

# Variables for manifold learning.
n_neighbors = 10
n_samples = 1000

# Create our sphere.
random_state = check_random_state(0)
p = random_state.rand(n_samples) * (2 * np.pi - 0.55)
t = random_state.rand(n_samples) * np.pi

# Sever the poles from the sphere.
indices = ((t < (np.pi - (np.pi / 8))) & (t > ((np.pi / 8))))
colors = p[indices]
x, y, z = np.sin(t[indices]) * np.cos(p[indices]), \
    np.sin(t[indices]) * np.sin(p[indices]), \
    np.cos(t[indices])

# Plot our dataset.
fig = plt.figure(figsize=(15, 8))
plt.suptitle("Manifold Learning with %i points, %i neighbors"
             % (1000, n_neighbors), fontsize=14)

ax = fig.add_subplot(251, projection='3d')
ax.scatter(x, y, z, c=p[indices], cmap=plt.cm.rainbow)
ax.view_init(40, -10)

sphere_data = np.array([x, y, z]).T

# Perform Locally Linear Embedding Manifold learning
methods = ['standard', 'ltsa', 'hessian', 'modified']
labels = ['LLE', 'LTSA', 'Hessian LLE', 'Modified LLE']

for i, method in enumerate(methods):
    t0 = time()
    trans_data = manifold\
        .LocallyLinearEmbedding(n_neighbors, 2,
                                method=method).fit_transform(sphere_data).T
    t1 = time()
    print("%s: %.2g sec" % (methods[i], t1 - t0))

    ax = fig.add_subplot(252 + i)
    plt.scatter(trans_data[0], trans_data[1], c=colors, cmap=plt.cm.rainbow)
    plt.title("%s (%.2g sec)" % (labels[i], t1 - t0))
```

(continues on next page)

(continued from previous page)

```
    ax.xaxis.set_major_formatter(NullFormatter())
    ax.yaxis.set_major_formatter(NullFormatter())
    plt.axis('tight')

# Perform Isomap Manifold learning.
t0 = time()
trans_data = manifold.Isomap(n_neighbors, n_components=2)\
    .fit_transform(sphere_data).T
t1 = time()
print("%s: %.2g sec" % ('ISO', t1 - t0))

ax = fig.add_subplot(257)
plt.scatter(trans_data[0], trans_data[1], c=colors, cmap=plt.cm.rainbow)
plt.title("%s (%.2g sec)" % ('Isomap', t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

# Perform Multi-dimensional scaling.
t0 = time()
mds = manifold.MDS(2, max_iter=100, n_init=1)
trans_data = mds.fit_transform(sphere_data).T
t1 = time()
print("MDS: %.2g sec" % (t1 - t0))

ax = fig.add_subplot(258)
plt.scatter(trans_data[0], trans_data[1], c=colors, cmap=plt.cm.rainbow)
plt.title("MDS (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

# Perform Spectral Embedding.
t0 = time()
se = manifold.SpectralEmbedding(n_components=2,
                                n_neighbors=n_neighbors)
trans_data = se.fit_transform(sphere_data).T
t1 = time()
print("Spectral Embedding: %.2g sec" % (t1 - t0))

ax = fig.add_subplot(259)
plt.scatter(trans_data[0], trans_data[1], c=colors, cmap=plt.cm.rainbow)
plt.title("Spectral Embedding (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

# Perform t-distributed stochastic neighbor embedding.
t0 = time()
tsne = manifold.TSNE(n_components=2, init='pca', random_state=0)
trans_data = tsne.fit_transform(sphere_data).T
t1 = time()
print("t-SNE: %.2g sec" % (t1 - t0))

ax = fig.add_subplot(2, 5, 10)
plt.scatter(trans_data[0], trans_data[1], c=colors, cmap=plt.cm.rainbow)
plt.title("t-SNE (%.2g sec)" % (t1 - t0))
```

(continues on next page)

(continued from previous page)

```
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

plt.show()
```

Total running time of the script: (0 minutes 5.904 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.18.6 Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...

An illustration of various embeddings on the digits dataset.

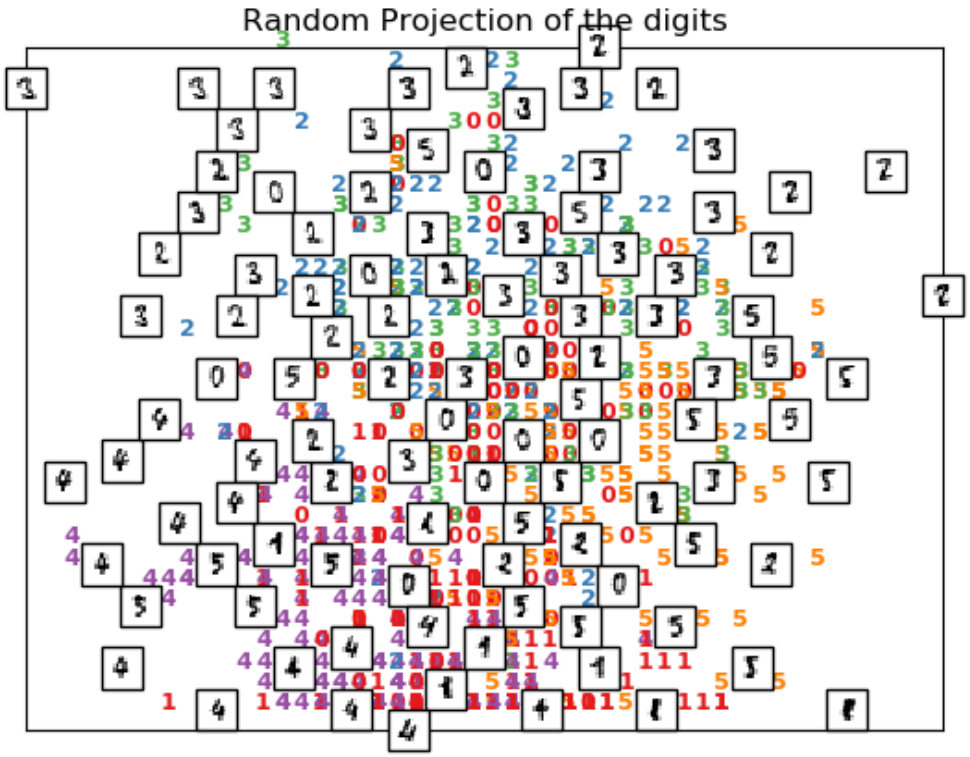
The RandomTreesEmbedding, from the *sklearn.ensemble* module, is not technically a manifold embedding method, as it learn a high-dimensional representation on which we apply a dimensionality reduction method. However, it is often useful to cast a dataset into a representation in which the classes are linearly-separable.

t-SNE will be initialized with the embedding that is generated by PCA in this example, which is not the default setting. It ensures global stability of the embedding, i.e., the embedding does not depend on random initialization.

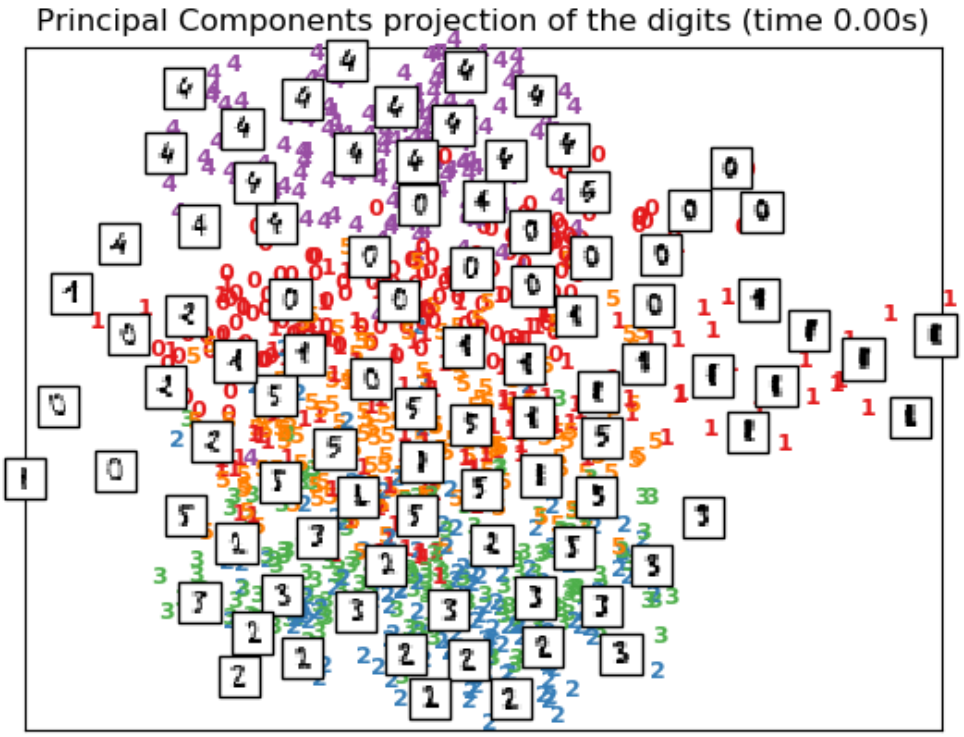
Linear Discriminant Analysis, from the *sklearn.discriminant_analysis* module, and Neighborhood Components Analysis, from the *sklearn.neighbors* module, are supervised dimensionality reduction method, i.e. they make use of the provided labels, contrary to other methods.

A selection from the 64-dimensional digits dataset

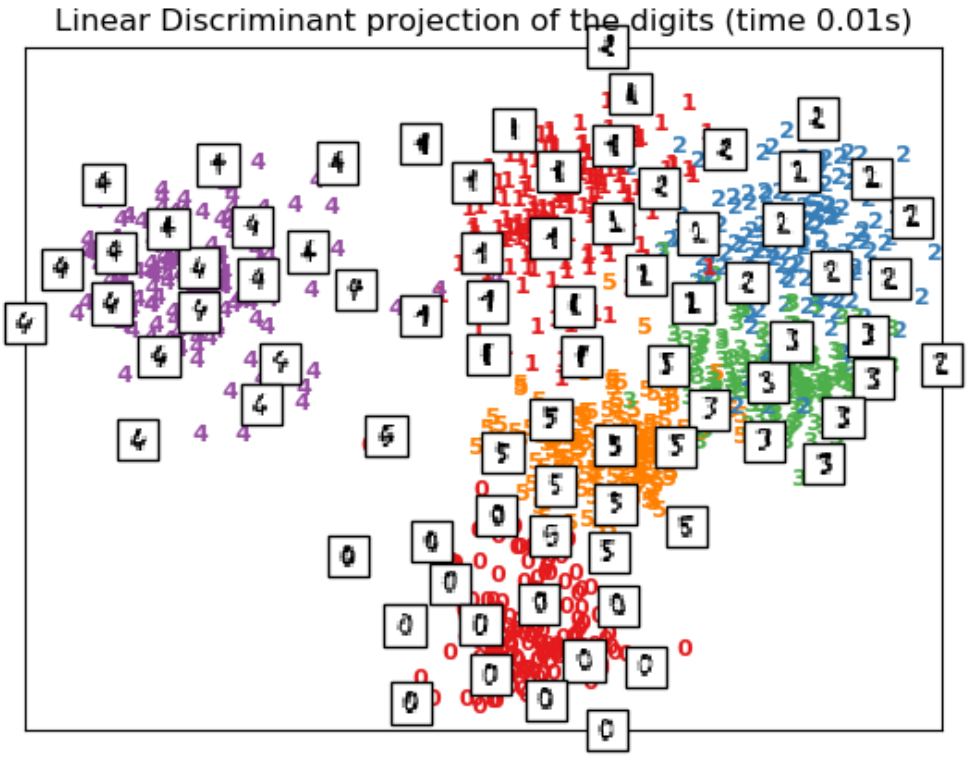
-


Random Projection of the digits

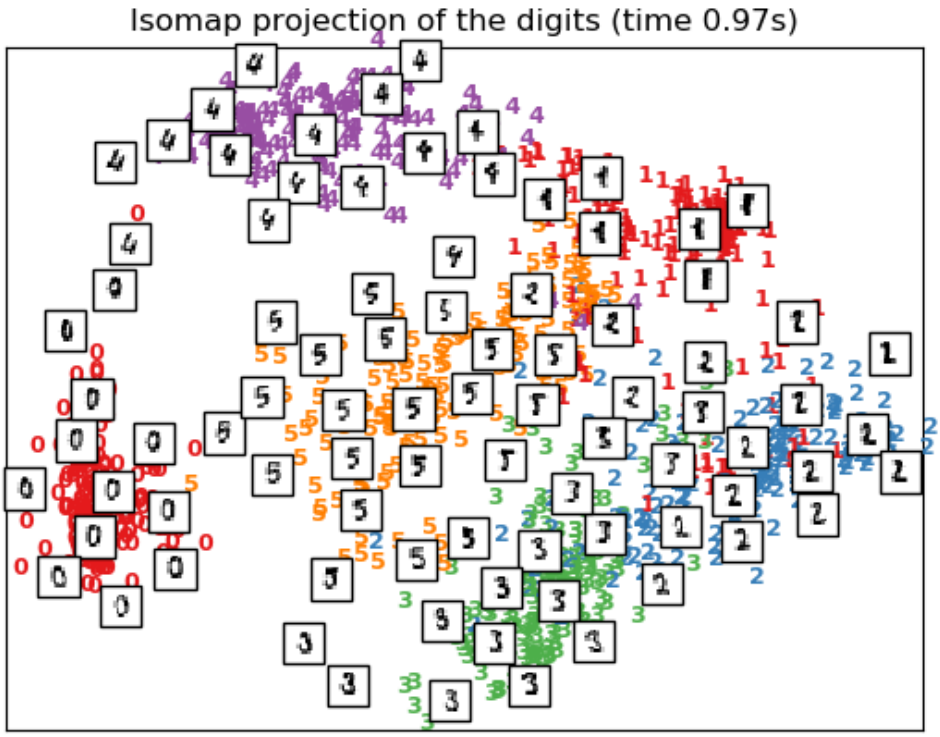
-


Principal Components projection of the digits (time 0.00s)

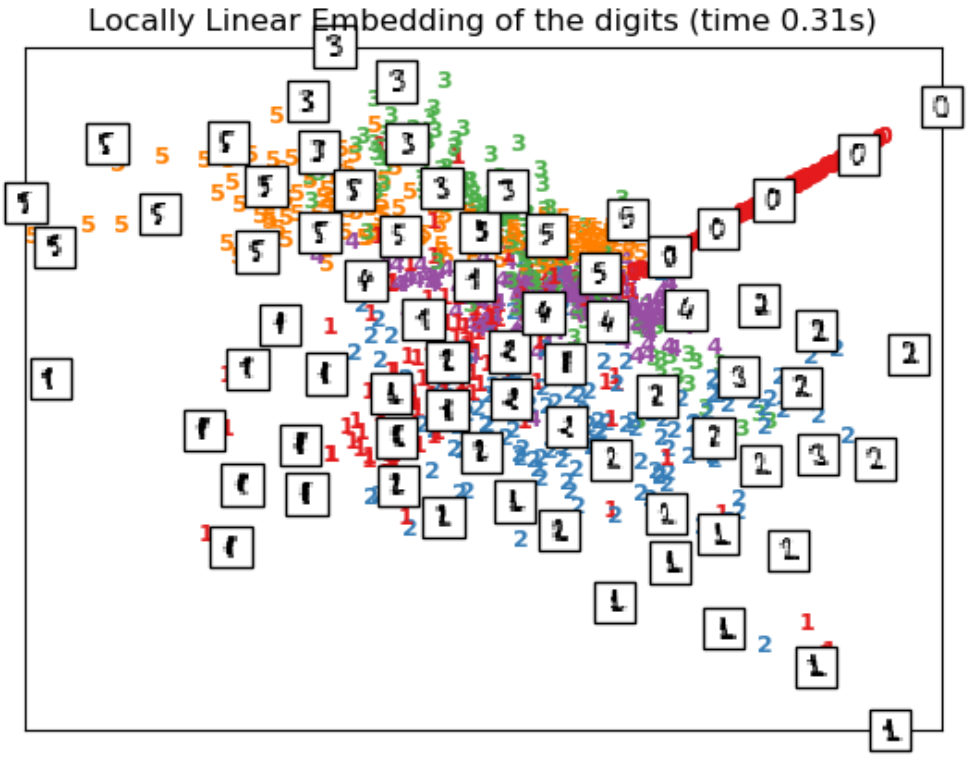
-


Linear Discriminant projection of the digits (time 0.01s)

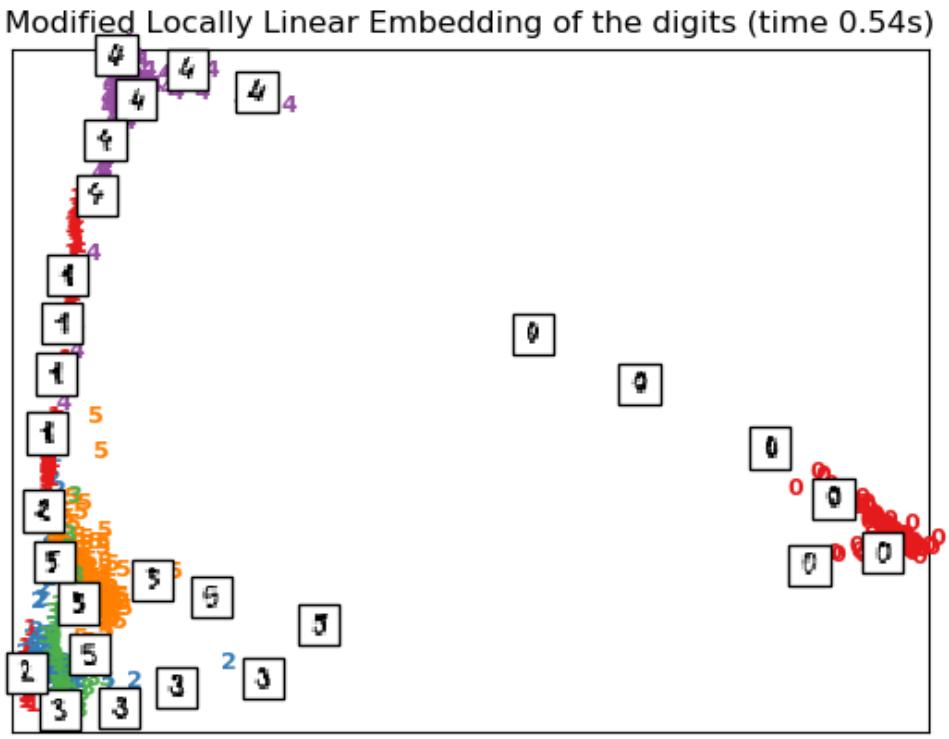
-


Isomap projection of the digits (time 0.97s)

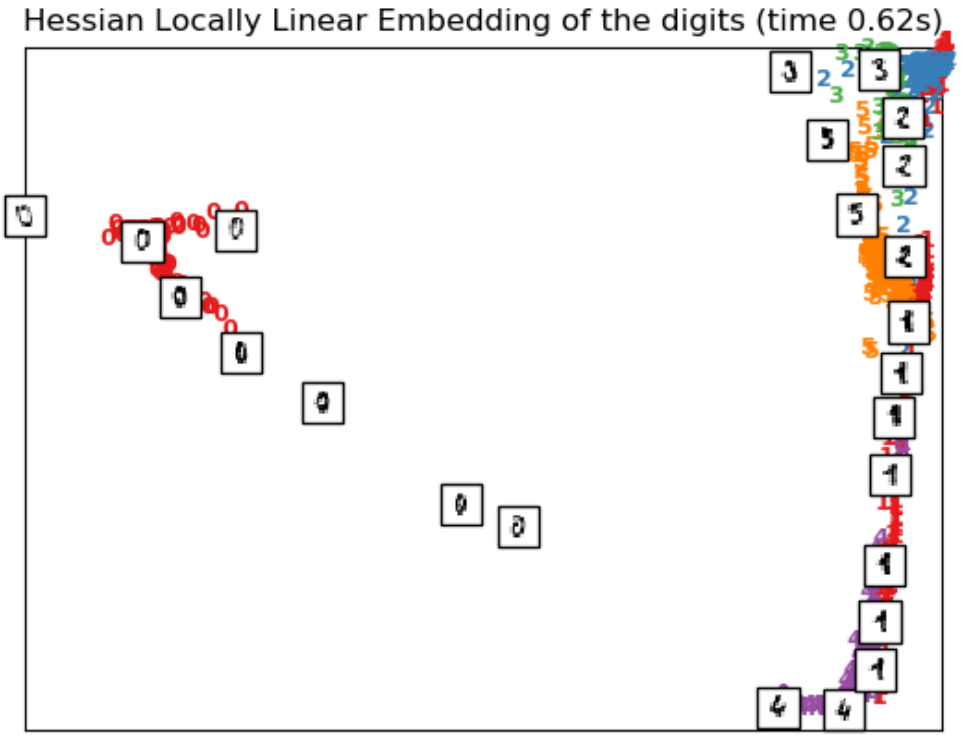
-


Locally Linear Embedding of the digits (time 0.31s)

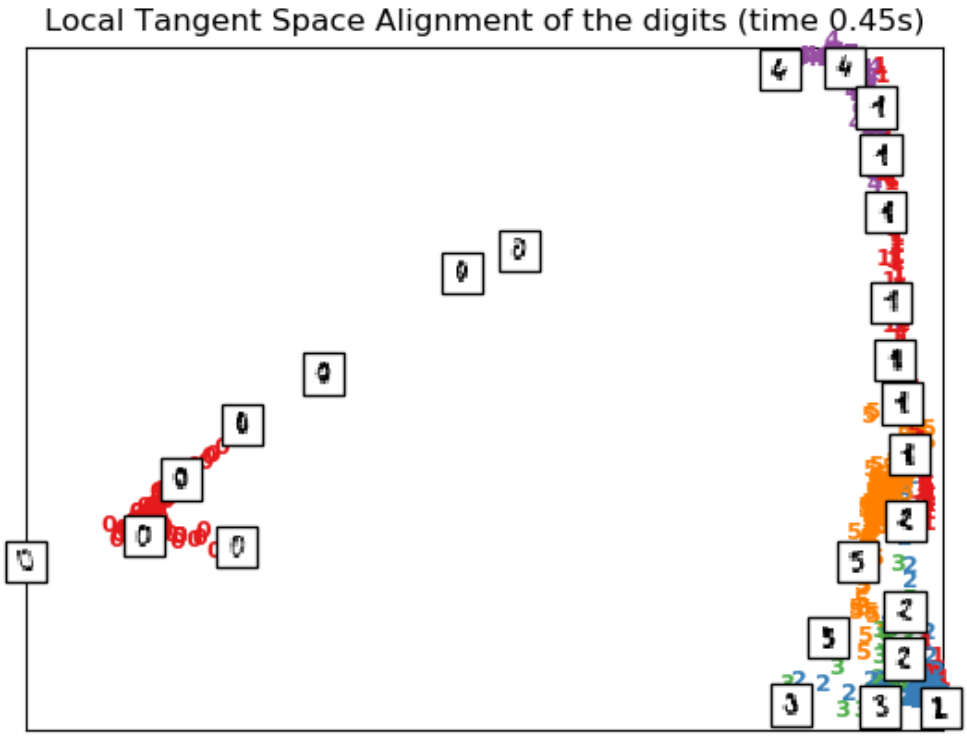
-


Modified Locally Linear Embedding of the digits (time 0.54s)

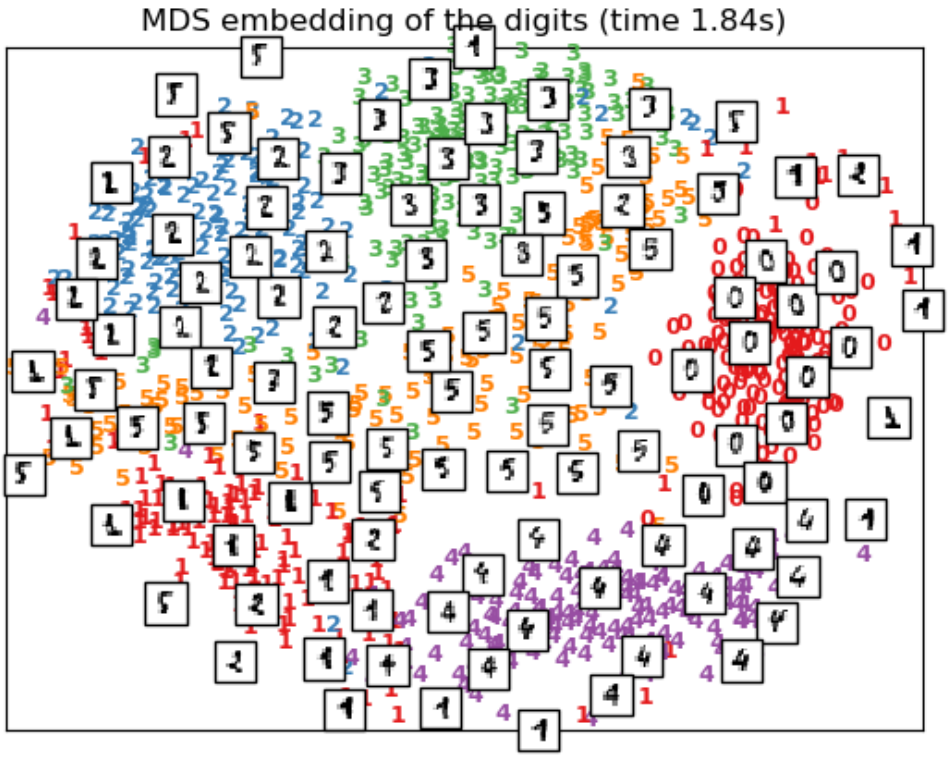
-


Hessian Locally Linear Embedding of the digits (time 0.62s)

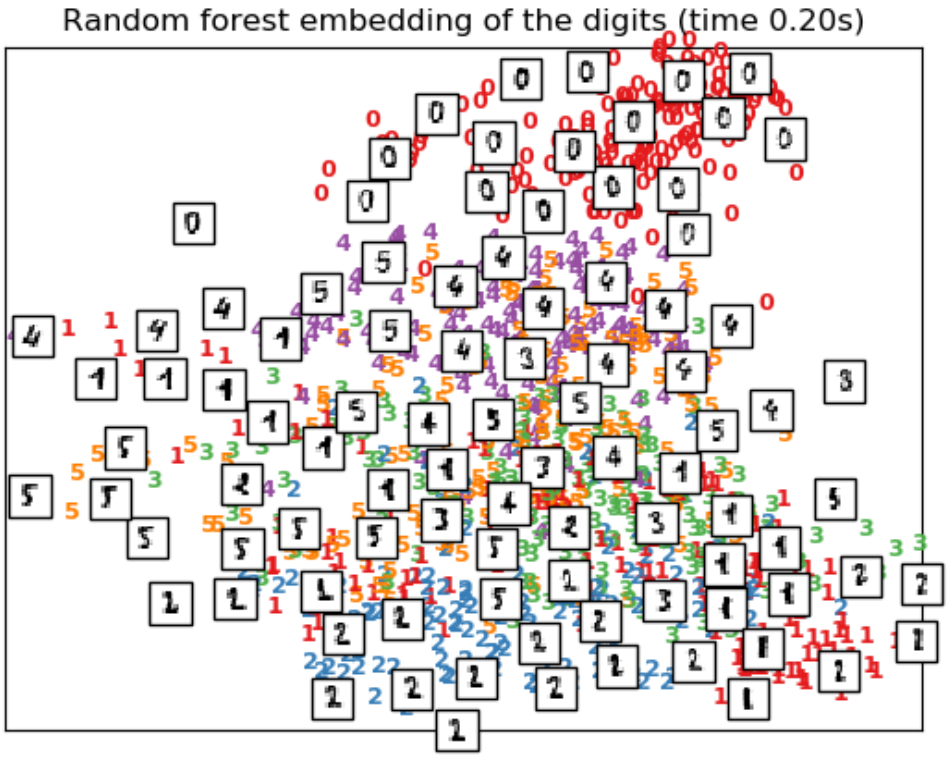
-


Local Tangent Space Alignment of the digits (time 0.45s)

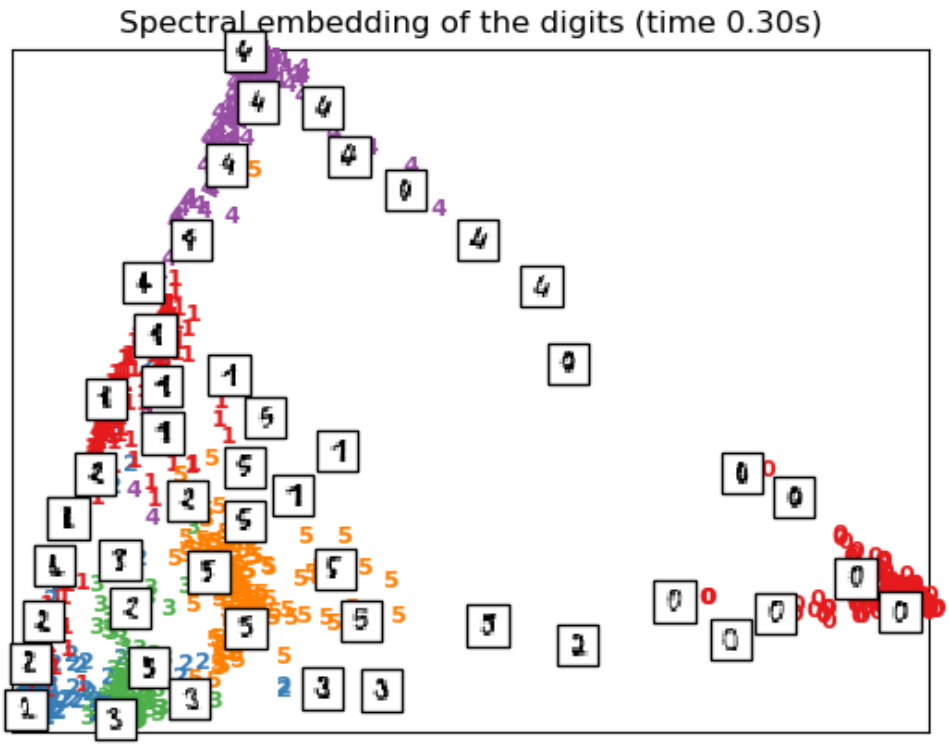
-


MDS embedding of the digits (time 1.84s)

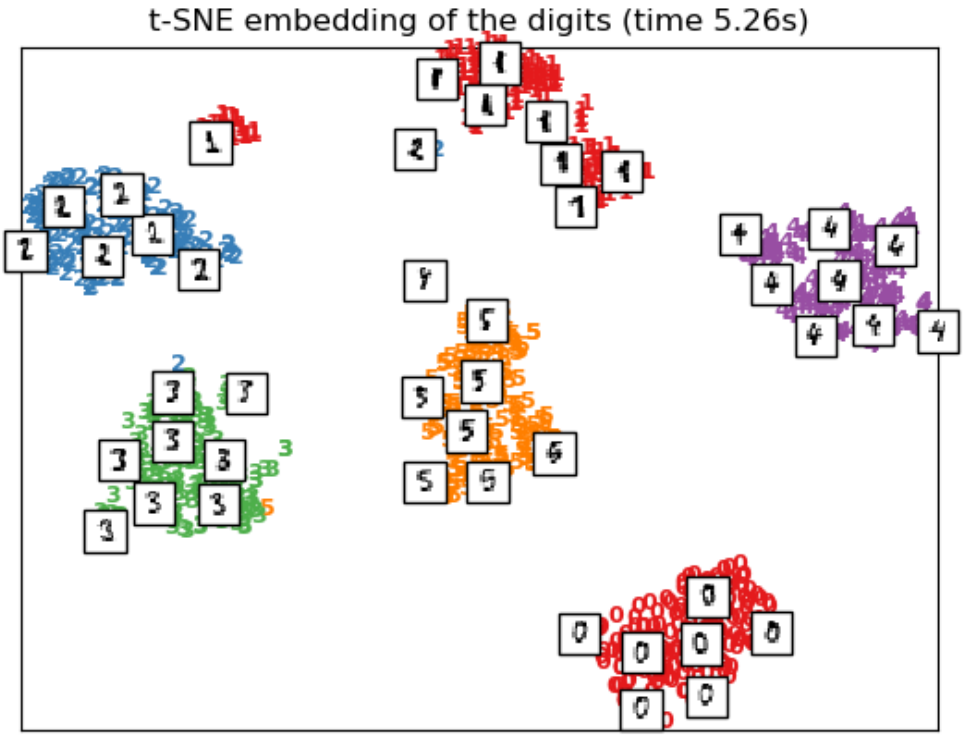
-


Random forest embedding of the digits (time 0.20s)

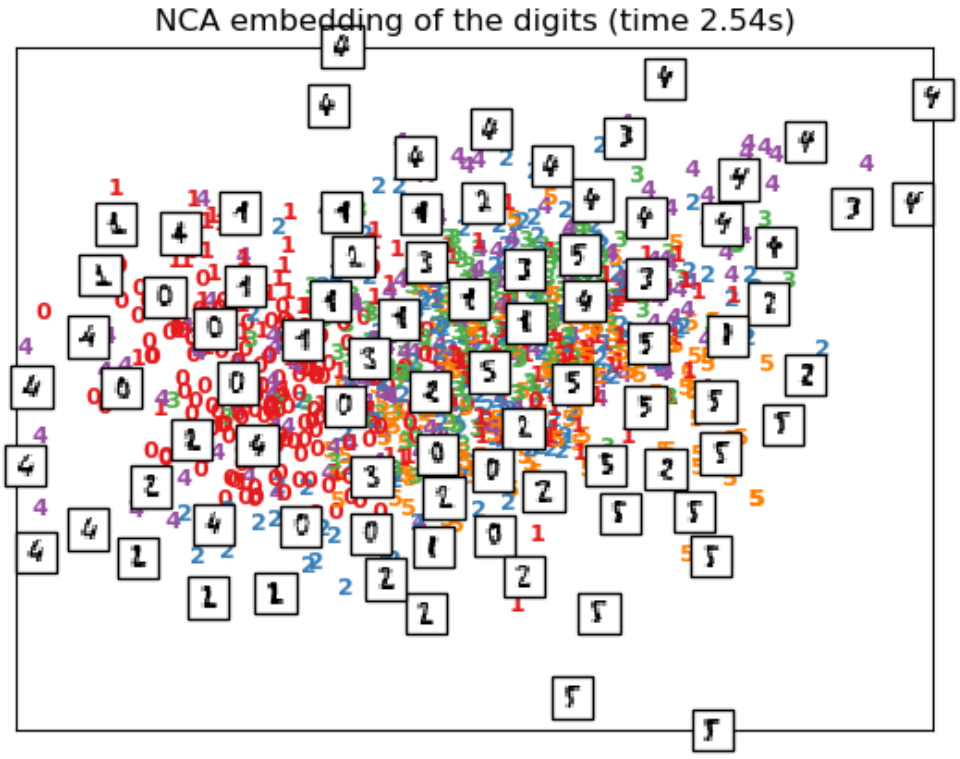
-


Spectral embedding of the digits (time 0.30s)

-


t-SNE embedding of the digits (time 5.26s)

-



-

Out:

```
Computing random projection
Computing PCA projection
Computing Linear Discriminant Analysis projection
Computing Isomap projection
Done.
Computing LLE embedding
Done. Reconstruction error: 1.63544e-06
Computing modified LLE embedding
Done. Reconstruction error: 0.36067
Computing Hessian LLE embedding
Done. Reconstruction error: 0.212806
Computing LTSA embedding
Done. Reconstruction error: 0.212804
Computing MDS embedding
Done. Stress: 137396079.407684
Computing Totally Random Trees embedding
Computing Spectral embedding
Computing t-SNE embedding
Computing NCA projection
```

```
# Authors: Fabian Pedregosa <fabian.pedregosa@inria.fr>
#          Olivier Grisel <olivier.grisel@ensta.org>
#          Mathieu Blondel <mathieu@mblondel.org>
#          Gael Varoquaux
# License: BSD 3 clause (C) INRIA 2011

from time import time
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import offsetbox
from sklearn import (manifold, datasets, decomposition, ensemble,
                     discriminant_analysis, random_projection, neighbors)
print(__doc__)

digits = datasets.load_digits(n_class=6)
X = digits.data
y = digits.target
n_samples, n_features = X.shape
n_neighbors = 30


# ----------------------------------------------------------------------
# Scale and visualize the embedding vectors
def plot_embedding(X, title=None):
    x_min, x_max = np.min(X, 0), np.max(X, 0)
    X = (X - x_min) / (x_max - x_min)

    plt.figure()
    ax = plt.subplot(111)
    for i in range(X.shape[0]):
        plt.text(X[i, 0], X[i, 1], str(y[i]),
                 color=plt.cm.Set1(y[i] / 10.),
                 fontdict={'weight': 'bold', 'size': 9})

    if hasattr(offsetbox, 'AnnotationBbox'):
        # only print thumbnails with matplotlib > 1.0
        shown_images = np.array([[1., 1.]])  # just something big
        for i in range(X.shape[0]):
            dist = np.sum((X[i] - shown_images) ** 2, 1)
            if np.min(dist) < 4e-3:
                # don't show points that are too close
                continue
            shown_images = np.r_[shown_images, [X[i]]]
            imagebox = offsetbox.AnnotationBbox(
                offsetbox.OffsetImage(digits.images[i], cmap=plt.cm.gray_r),
                X[i])
            ax.add_artist(imagebox)
    plt.xticks([]), plt.yticks([])
    if title is not None:
        plt.title(title)


# ----------------------------------------------------------------------
# Plot images of the digits
n_img_per_row = 20
img = np.zeros((10 * n_img_per_row, 10 * n_img_per_row))
for i in range(n_img_per_row):
```

(continues on next page)

(continued from previous page)

```
    ix = 10 * i + 1
    for j in range(n_img_per_row):
        iy = 10 * j + 1
        img[ix:ix + 8, iy:iy + 8] = X[i * n_img_per_row + j].reshape((8, 8))

plt.imshow(img, cmap=plt.cm.binary)
plt.xticks([])
plt.yticks([])
plt.title('A selection from the 64-dimensional digits dataset')


# ----------------------------------------------------------------------
# Random 2D projection using a random unitary matrix
print("Computing random projection")
rp = random_projection.SparseRandomProjection(n_components=2, random_state=42)
X_projected = rp.fit_transform(X)
plot_embedding(X_projected, "Random Projection of the digits")


# ----------------------------------------------------------------------
# Projection on to the first 2 principal components

print("Computing PCA projection")
t0 = time()
X_pca = decomposition.TruncatedSVD(n_components=2).fit_transform(X)
plot_embedding(X_pca,
               "Principal Components projection of the digits (time %.2fs)" %
               (time() - t0))

# ----------------------------------------------------------------------
# Projection on to the first 2 linear discriminant components

print("Computing Linear Discriminant Analysis projection")
X2 = X.copy()
X2.flat[::X.shape[1] + 1] += 0.01  # Make X invertible
t0 = time()
X_lda = discriminant_analysis.LinearDiscriminantAnalysis(n_components=2
                                                         ).fit_transform(X2, y)
plot_embedding(X_lda,
               "Linear Discriminant projection of the digits (time %.2fs)" %
               (time() - t0))


# ----------------------------------------------------------------------
# Isomap projection of the digits dataset
print("Computing Isomap projection")
t0 = time()
X_iso = manifold.Isomap(n_neighbors, n_components=2).fit_transform(X)
print("Done.")
plot_embedding(X_iso,
               "Isomap projection of the digits (time %.2fs)" %
               (time() - t0))


# ----------------------------------------------------------------------
# Locally linear embedding of the digits dataset
print("Computing LLE embedding")
```

(continues on next page)

(continued from previous page)

```
clf = manifold.LocallyLinearEmbedding(n_neighbors, n_components=2,
                                      method='standard')
t0 = time()
X_lle = clf.fit_transform(X)
print("Done. Reconstruction error: %g" % clf.reconstruction_error_)
plot_embedding(X_lle,
               "Locally Linear Embedding of the digits (time %.2fs)" %
               (time() - t0))


# ----------------------------------------------------------------------
# Modified Locally linear embedding of the digits dataset
print("Computing modified LLE embedding")
clf = manifold.LocallyLinearEmbedding(n_neighbors, n_components=2,
                                      method='modified')
t0 = time()
X_mlle = clf.fit_transform(X)
print("Done. Reconstruction error: %g" % clf.reconstruction_error_)
plot_embedding(X_mlle,
               "Modified Locally Linear Embedding of the digits (time %.2fs)" %
               (time() - t0))


# ----------------------------------------------------------------------
# HLLE embedding of the digits dataset
print("Computing Hessian LLE embedding")
clf = manifold.LocallyLinearEmbedding(n_neighbors, n_components=2,
                                      method='hessian')
t0 = time()
X_hlle = clf.fit_transform(X)
print("Done. Reconstruction error: %g" % clf.reconstruction_error_)
plot_embedding(X_hlle,
               "Hessian Locally Linear Embedding of the digits (time %.2fs)" %
               (time() - t0))


# ----------------------------------------------------------------------
# LTSA embedding of the digits dataset
print("Computing LTSA embedding")
clf = manifold.LocallyLinearEmbedding(n_neighbors, n_components=2,
                                      method='ltsa')
t0 = time()
X_ltsa = clf.fit_transform(X)
print("Done. Reconstruction error: %g" % clf.reconstruction_error_)
plot_embedding(X_ltsa,
               "Local Tangent Space Alignment of the digits (time %.2fs)" %
               (time() - t0))

# ----------------------------------------------------------------------
# MDS  embedding of the digits dataset
print("Computing MDS embedding")
clf = manifold.MDS(n_components=2, n_init=1, max_iter=100)
t0 = time()
X_mds = clf.fit_transform(X)
print("Done. Stress: %f" % clf.stress_)
plot_embedding(X_mds,
               "MDS embedding of the digits (time %.2fs)" %
```

(continues on next page)

(continued from previous page)

```
               (time() - t0))

# ----------------------------------------------------------------------
# Random Trees embedding of the digits dataset
print("Computing Totally Random Trees embedding")
hasher = ensemble.RandomTreesEmbedding(n_estimators=200, random_state=0,
                                       max_depth=5)
t0 = time()
X_transformed = hasher.fit_transform(X)
pca = decomposition.TruncatedSVD(n_components=2)
X_reduced = pca.fit_transform(X_transformed)

plot_embedding(X_reduced,
               "Random forest embedding of the digits (time %.2fs)" %
               (time() - t0))

# ----------------------------------------------------------------------
# Spectral embedding of the digits dataset
print("Computing Spectral embedding")
embedder = manifold.SpectralEmbedding(n_components=2, random_state=0,
                                      eigen_solver="arpack")
t0 = time()
X_se = embedder.fit_transform(X)

plot_embedding(X_se,
               "Spectral embedding of the digits (time %.2fs)" %
               (time() - t0))

# ----------------------------------------------------------------------
# t-SNE embedding of the digits dataset
print("Computing t-SNE embedding")
tsne = manifold.TSNE(n_components=2, init='pca', random_state=0)
t0 = time()
X_tsne = tsne.fit_transform(X)

plot_embedding(X_tsne,
               "t-SNE embedding of the digits (time %.2fs)" %
               (time() - t0))

# ----------------------------------------------------------------------
# NCA projection of the digits dataset
print("Computing NCA projection")
nca = neighbors.NeighborhoodComponentsAnalysis(init='random',
                                               n_components=2, random_state=0)
t0 = time()
X_nca = nca.fit_transform(X, y)

plot_embedding(X_nca,
               "NCA embedding of the digits (time %.2fs)" %
               (time() - t0))

plt.show()
```

Total running time of the script: (0 minutes 24.409 seconds)

Estimated memory usage: 94 MB

6.19 Missing Value Imputation

Examples concerning the *sklearn.impute* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.19.1 Imputing missing values with variants of IterativeImputer

The *sklearn.impute.IterativeImputer* class is very flexible - it can be used with a variety of estimators to do round-robin regression, treating every variable as an output in turn.

In this example we compare some estimators for the purpose of missing feature imputation with *sklearn.impute.IterativeImputer*:

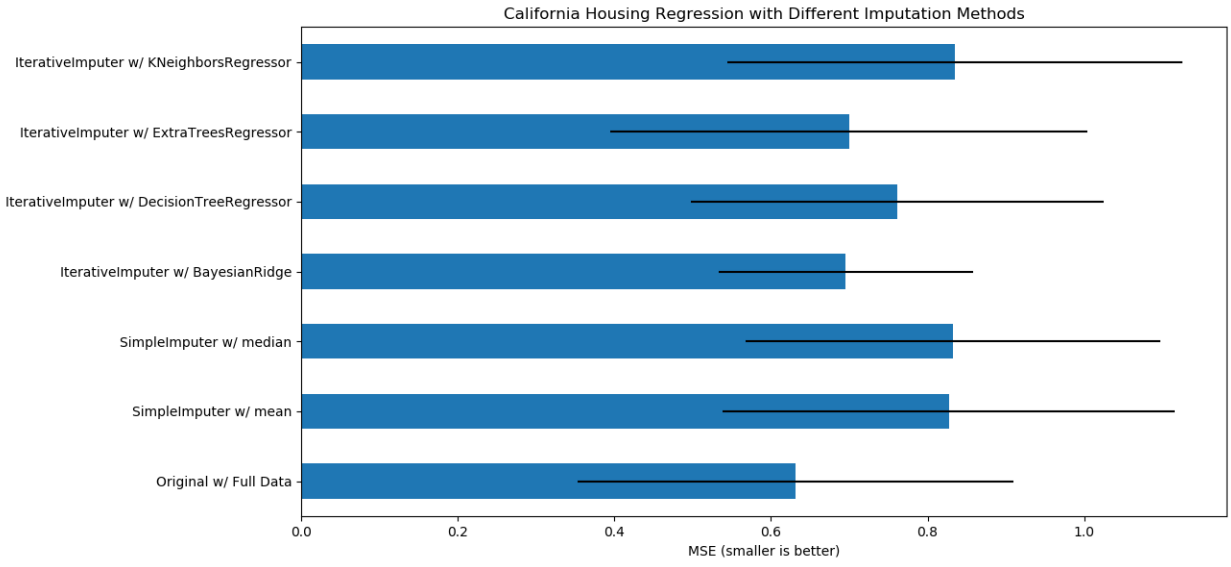
- *BayesianRidge*: regularized linear regression
- *DecisionTreeRegressor*: non-linear regression
- *ExtraTreesRegressor*: similar to missForest in R
- *KNeighborsRegressor*: comparable to other KNN imputation approaches

Of particular interest is the ability of *sklearn.impute.IterativeImputer* to mimic the behavior of missForest, a popular imputation package for R. In this example, we have chosen to use *sklearn.ensemble.ExtraTreesRegressor* instead of *sklearn.ensemble.RandomForestRegressor* (as in missForest) due to its increased speed.

Note that *sklearn.neighbors.KNeighborsRegressor* is different from KNN imputation, which learns from samples with missing values by using a distance metric that accounts for missing values, rather than imputing them.

The goal is to compare different estimators to see which one is best for the *sklearn.impute.IterativeImputer* when using a *sklearn.linear_model.BayesianRidge* estimator on the California housing dataset with a single value randomly removed from each row.

For this particular pattern of missing values we see that *sklearn.ensemble.ExtraTreesRegressor* and *sklearn.linear_model.BayesianRidge* give the best results.



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# To use this experimental feature, we need to explicitly ask for it:
from sklearn.experimental import enable_iterative_imputer  # noqa
from sklearn.datasets import fetch_california_housing
from sklearn.impute import SimpleImputer
from sklearn.impute import IterativeImputer
from sklearn.linear_model import BayesianRidge
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score

N_SPLITS = 5

rng = np.random.RandomState(0)

X_full, y_full = fetch_california_housing(return_X_y=True)
# ~2k samples is enough for the purpose of the example.
# Remove the following two lines for a slower run with different error bars.
X_full = X_full[::10]
y_full = y_full[::10]
n_samples, n_features = X_full.shape

# Estimate the score on the entire dataset, with no missing values
br_estimator = BayesianRidge()
score_full_data = pd.DataFrame(
    cross_val_score(
        br_estimator, X_full, y_full, scoring='neg_mean_squared_error',
        cv=N_SPLITS
    ),
    columns=['Full Data']
)

# Add a single missing value to each row
X_missing = X_full.copy()
y_missing = y_full
missing_samples = np.arange(n_samples)
missing_features = rng.choice(n_features, n_samples, replace=True)
X_missing[missing_samples, missing_features] = np.nan

# Estimate the score after imputation (mean and median strategies)
score_simple_imputer = pd.DataFrame()
for strategy in ('mean', 'median'):
    estimator = make_pipeline(
        SimpleImputer(missing_values=np.nan, strategy=strategy),
        br_estimator
    )
    score_simple_imputer[strategy] = cross_val_score(
        estimator, X_missing, y_missing, scoring='neg_mean_squared_error',
        cv=N_SPLITS
    )
```

(continues on next page)

(continued from previous page)

```
# Estimate the score after iterative imputation of the missing values
# with different estimators
estimators = [
    BayesianRidge(),
    DecisionTreeRegressor(max_features='sqrt', random_state=0),
    ExtraTreesRegressor(n_estimators=10, random_state=0),
    KNeighborsRegressor(n_neighbors=15)
]
score_iterative_imputer = pd.DataFrame()
for impute_estimator in estimators:
    estimator = make_pipeline(
        IterativeImputer(random_state=0, estimator=impute_estimator),
        br_estimator
    )
    score_iterative_imputer[impute_estimator.__class__.__name__] = \
        cross_val_score(
            estimator, X_missing, y_missing, scoring='neg_mean_squared_error',
            cv=N_SPLITS
        )

scores = pd.concat(
    [score_full_data, score_simple_imputer, score_iterative_imputer],
    keys=['Original', 'SimpleImputer', 'IterativeImputer'], axis=1
)

# plot boston results
fig, ax = plt.subplots(figsize=(13, 6))
means = -scores.mean()
errors = scores.std()
means.plot.barh(xerr=errors, ax=ax)
ax.set_title('California Housing Regression with Different Imputation Methods')
ax.set_xlabel('MSE (smaller is better)')
ax.set_yticks(np.arange(means.shape[0]))
ax.set_yticklabels([" w/ ".join(label) for label in means.index.tolist()])
plt.tight_layout(pad=1)
plt.show()
```

Total running time of the script: (0 minutes 17.209 seconds)

Estimated memory usage: 60 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.19.2 Imputing missing values before building an estimator

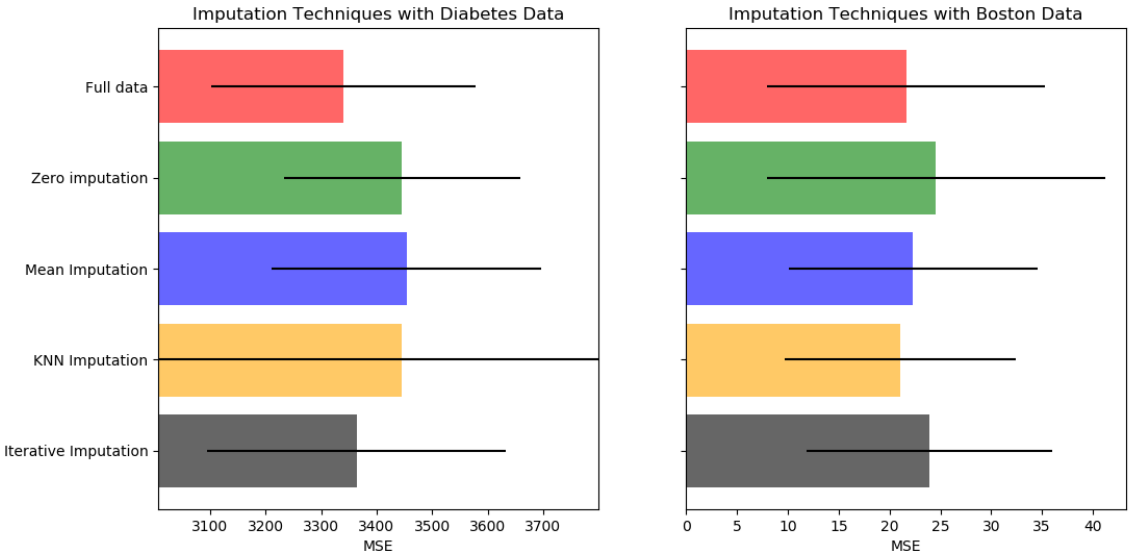
Missing values can be replaced by the mean, the median or the most frequent value using the basic *sklearn.impute.SimpleImputer*. The median is a more robust estimator for data with high magnitude variables which could dominate results (otherwise known as a 'long tail').

With KNNImputer, missing values can be imputed using the weighted or unweighted mean of the desired number of nearest neighbors.

Another option is the *sklearn.impute.IterativeImputer*. This uses round-robin linear regression, treating every variable as an output in turn. The version implemented assumes Gaussian (output) variables. If your features are

obviously non-Normal, consider transforming them to look more Normal so as to potentially improve performance.

In addition of using an imputing method, we can also keep an indication of the missing information using *sklearn.impute.MissingIndicator* which might carry some information.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

# To use the experimental IterativeImputer, we need to explicitly ask for it:
from sklearn.experimental import enable_iterative_imputer  # noqa
from sklearn.datasets import load_diabetes
from sklearn.datasets import load_boston
from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import make_pipeline, make_union
from sklearn.impute import (
    SimpleImputer, KNNImputer, IterativeImputer, MissingIndicator)
from sklearn.model_selection import cross_val_score

rng = np.random.RandomState(0)

N_SPLITS = 5
REGRESSOR = RandomForestRegressor(random_state=0)


def get_scores_for_imputer(imputer, X_missing, y_missing):
    estimator = make_pipeline(
        make_union(imputer, MissingIndicator(missing_values=0)),
        REGRESSOR)
    impute_scores = cross_val_score(estimator, X_missing, y_missing,
                                    scoring='neg_mean_squared_error',
                                    cv=N_SPLITS)
    return impute_scores
```

(continues on next page)

(continued from previous page)

```
def get_results(dataset):
    X_full, y_full = dataset.data, dataset.target
    n_samples = X_full.shape[0]
    n_features = X_full.shape[1]

    # Estimate the score on the entire dataset, with no missing values
    full_scores = cross_val_score(REGRESSOR, X_full, y_full,
                                  scoring='neg_mean_squared_error',
                                  cv=N_SPLITS)

    # Add missing values in 75% of the lines
    missing_rate = 0.75
    n_missing_samples = int(np.floor(n_samples * missing_rate))
    missing_samples = np.hstack((np.zeros(n_samples - n_missing_samples,
                                          dtype=np.bool),
                                 np.ones(n_missing_samples,
                                         dtype=np.bool)))
    rng.shuffle(missing_samples)
    missing_features = rng.randint(0, n_features, n_missing_samples)
    X_missing = X_full.copy()
    X_missing[np.where(missing_samples)[0], missing_features] = 0
    y_missing = y_full.copy()

    # Estimate the score after replacing missing values by 0
    imputer = SimpleImputer(missing_values=0,
                            strategy='constant',
                            fill_value=0)
    zero_impute_scores = get_scores_for_imputer(imputer, X_missing, y_missing)

    # Estimate the score after imputation (mean strategy) of the missing values
    imputer = SimpleImputer(missing_values=0, strategy="mean")
    mean_impute_scores = get_scores_for_imputer(imputer, X_missing, y_missing)

    # Estimate the score after kNN-imputation of the missing values
    imputer = KNNImputer(missing_values=0)
    knn_impute_scores = get_scores_for_imputer(imputer, X_missing, y_missing)

    # Estimate the score after iterative imputation of the missing values
    imputer = IterativeImputer(missing_values=0,
                               random_state=0,
                               n_nearest_features=5,
                               sample_posterior=True)
    iterative_impute_scores = get_scores_for_imputer(imputer,
                                                     X_missing,
                                                     y_missing)

    return ((full_scores.mean(), full_scores.std()),
            (zero_impute_scores.mean(), zero_impute_scores.std()),
            (mean_impute_scores.mean(), mean_impute_scores.std()),
            (knn_impute_scores.mean(), knn_impute_scores.std()),
            (iterative_impute_scores.mean(), iterative_impute_scores.std()))


results_diabetes = np.array(get_results(load_diabetes()))
mses_diabetes = results_diabetes[:, 0] * -1
stds_diabetes = results_diabetes[:, 1]
```

(continues on next page)

(continued from previous page)

```
results_boston = np.array(get_results(load_boston()))
mses_boston = results_boston[:, 0] * -1
stds_boston = results_boston[:, 1]

n_bars = len(mses_diabetes)
xval = np.arange(n_bars)

x_labels = ['Full data',
            'Zero imputation',
            'Mean Imputation',
            'KNN Imputation',
            'Iterative Imputation']
colors = ['r', 'g', 'b', 'orange', 'black']

# plot diabetes results
plt.figure(figsize=(12, 6))
ax1 = plt.subplot(121)
for j in xval:
    ax1.barh(j, mses_diabetes[j], xerr=stds_diabetes[j],
             color=colors[j], alpha=0.6, align='center')

ax1.set_title('Imputation Techniques with Diabetes Data')
ax1.set_xlim(left=np.min(mses_diabetes) * 0.9,
             right=np.max(mses_diabetes) * 1.1)
ax1.set_yticks(xval)
ax1.set_xlabel('MSE')
ax1.invert_yaxis()
ax1.set_yticklabels(x_labels)

# plot boston results
ax2 = plt.subplot(122)
for j in xval:
    ax2.barh(j, mses_boston[j], xerr=stds_boston[j],
             color=colors[j], alpha=0.6, align='center')

ax2.set_title('Imputation Techniques with Boston Data')
ax2.set_yticks(xval)
ax2.set_xlabel('MSE')
ax2.invert_yaxis()
ax2.set_yticklabels([''] * n_bars)

plt.show()
```

Total running time of the script: (0 minutes 16.465 seconds)

Estimated memory usage: 8 MB

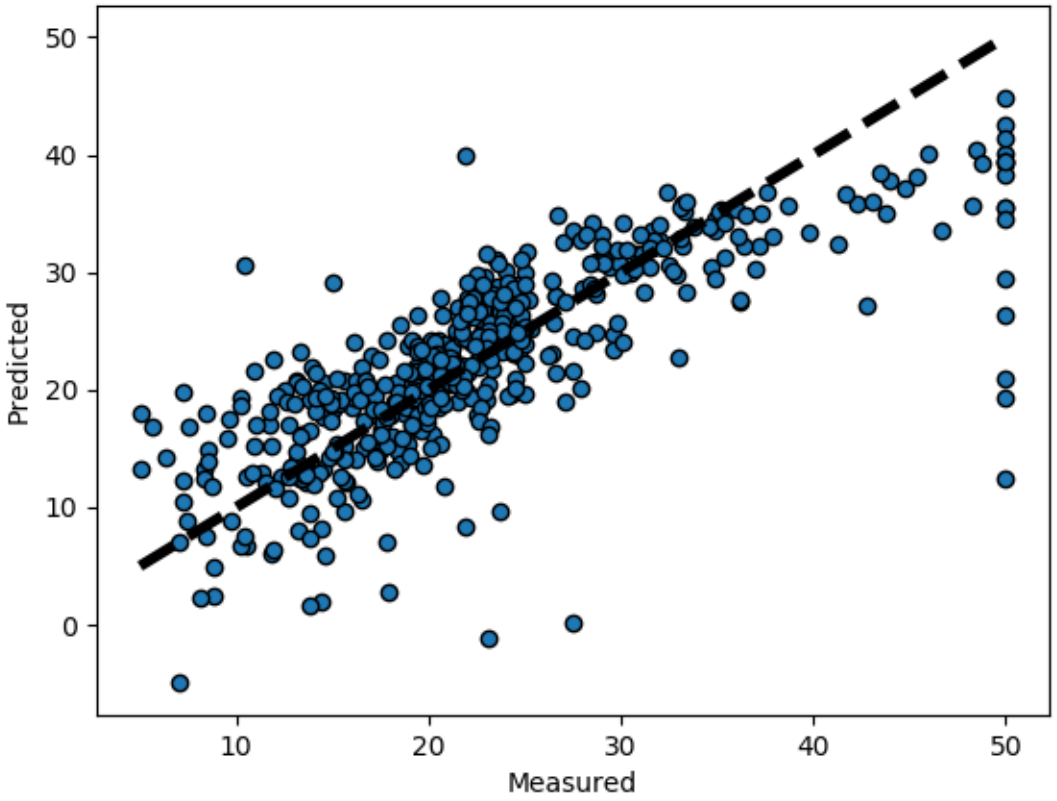
6.20 Model Selection

Examples related to the *sklearn.model_selection* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.1 Plotting Cross-Validated Predictions

This example shows how to use *cross_val_predict* to visualize prediction errors.




```
from sklearn import datasets
from sklearn.model_selection import cross_val_predict
from sklearn import linear_model
import matplotlib.pyplot as plt

lr = linear_model.LinearRegression()
X, y = datasets.load_boston(return_X_y=True)

# cross_val_predict returns an array of the same size as `y` where each entry
# is a prediction obtained by cross validation:
predicted = cross_val_predict(lr, X, y, cv=10)

fig, ax = plt.subplots()
ax.scatter(y, predicted, edgecolors=(0, 0, 0))
ax.plot([y.min(), y.max()], [y.min(), y.max()], 'k--', lw=4)
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
plt.show()
```

Total running time of the script: (0 minutes 0.482 seconds)

Estimated memory usage: 8 MB

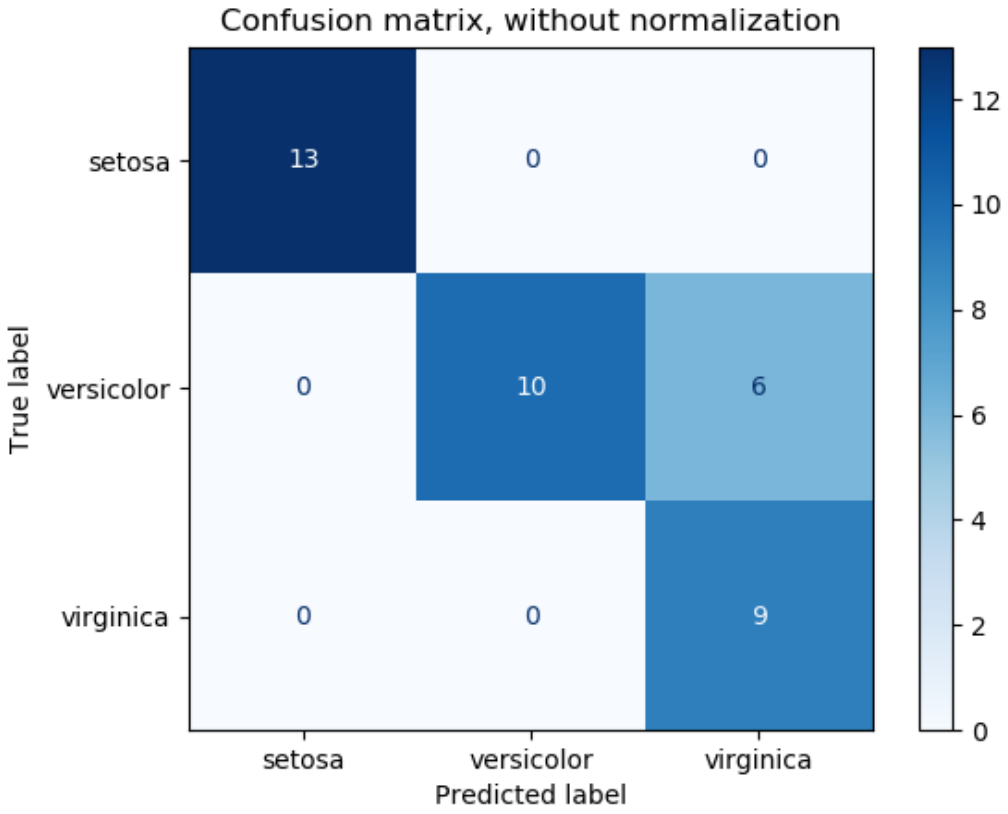
Note: Click *here* to download the full example code or to run this example in your browser via Binder

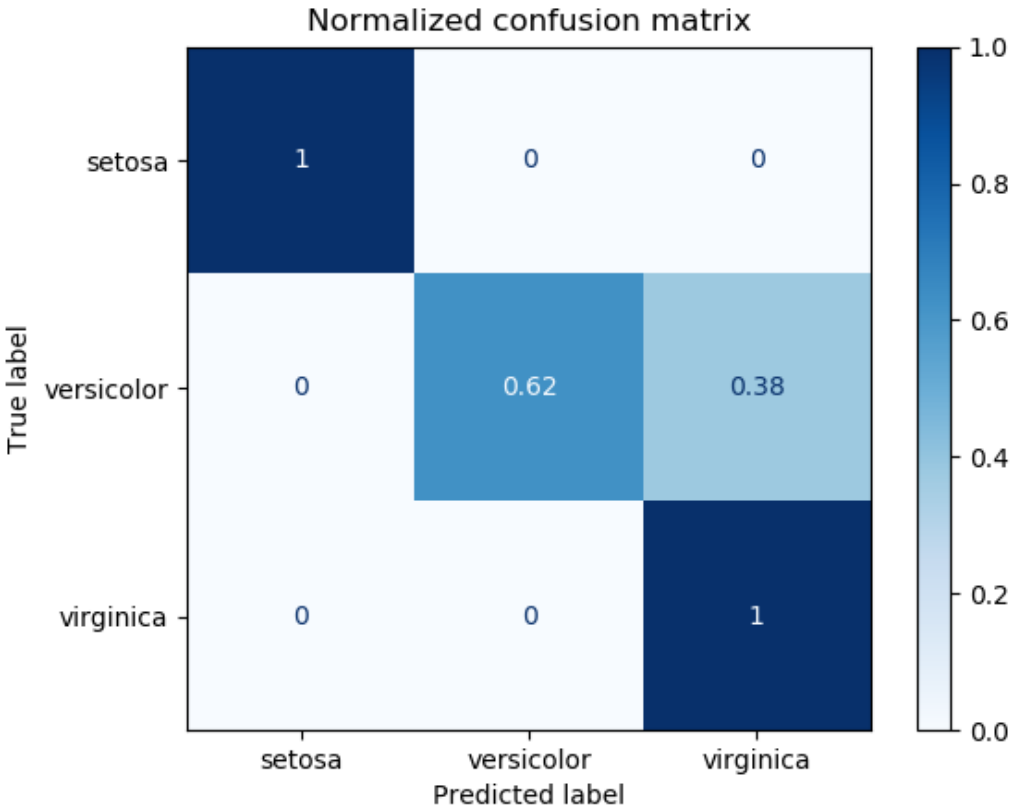
6.20.2 Confusion matrix

Example of confusion matrix usage to evaluate the quality of the output of a classifier on the iris data set. The diagonal elements represent the number of points for which the predicted label is equal to the true label, while off-diagonal elements are those that are mislabeled by the classifier. The higher the diagonal values of the confusion matrix the better, indicating many correct predictions.

The figures show the confusion matrix with and without normalization by class support size (number of elements in each class). This kind of normalization can be interesting in case of class imbalance to have a more visual interpretation of which class is being misclassified.

Here the results are not as good as they could be as our choice for the regularization parameter C was not the best. In real life applications this parameter is usually chosen using *Tuning the hyper-parameters of an estimator*.





Out:

```
Confusion matrix, without normalization
[[13  0  0]
 [ 0 10  6]
 [ 0  0  9]]
Normalized confusion matrix
[[1.   0.   0.  ]
 [0.   0.62 0.38]
 [0.   0.   1.  ]]
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import plot_confusion_matrix

# import some data to play with
iris = datasets.load_iris()
```

(continues on next page)

(continued from previous page)

```
X = iris.data
y = iris.target
class_names = iris.target_names

# Split the data into a training set and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# Run classifier, using a model that is too regularized (C too low) to see
# the impact on the results
classifier = svm.SVC(kernel='linear', C=0.01).fit(X_train, y_train)

np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
titles_options = [("Confusion matrix, without normalization", None),
                  ("Normalized confusion matrix", 'true')]
for title, normalize in titles_options:
    disp = plot_confusion_matrix(classifier, X_test, y_test,
                                 display_labels=class_names,
                                 cmap=plt.cm.Blues,
                                 normalize=normalize)
    disp.ax_.set_title(title)

    print(title)
    print(disp.confusion_matrix)

plt.show()
```

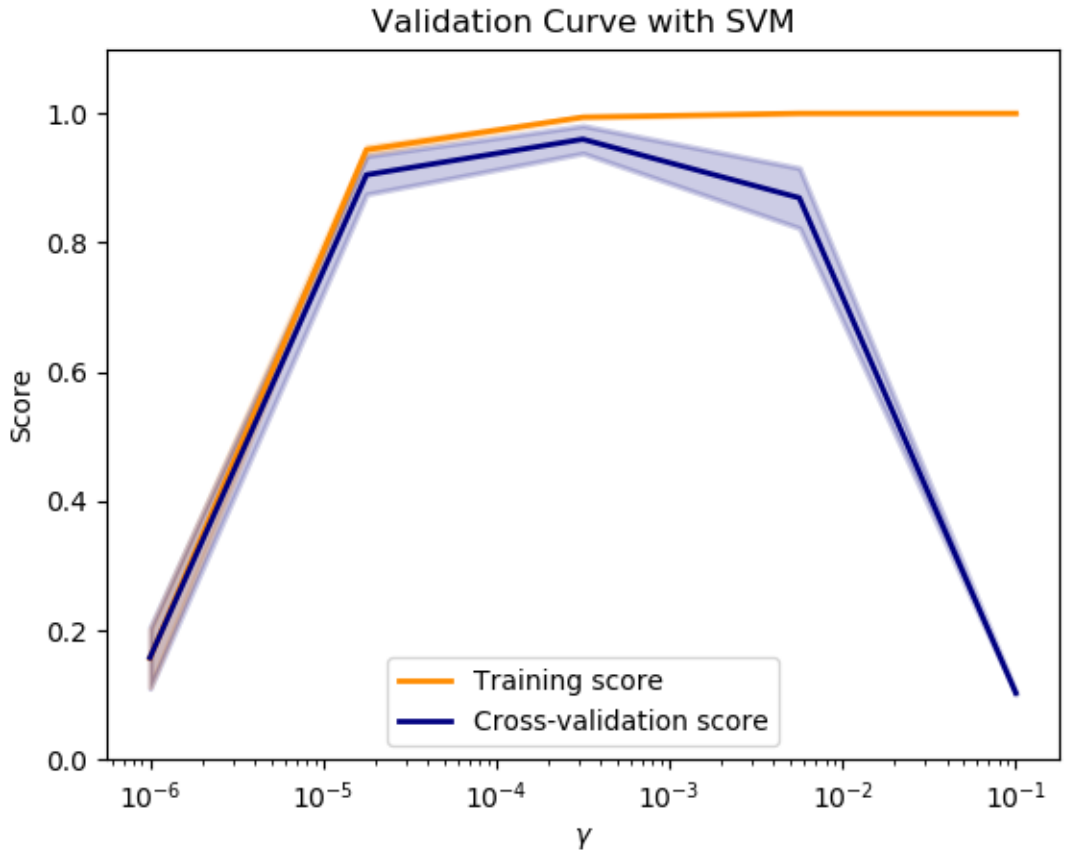
Total running time of the script: (0 minutes 0.577 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.3 Plotting Validation Curves

In this plot you can see the training scores and validation scores of an SVM for different values of the kernel parameter gamma. For very low values of gamma, you can see that both the training score and the validation score are low. This is called underfitting. Medium values of gamma will result in high values for both scores, i.e. the classifier is performing fairly well. If gamma is too high, the classifier will overfit, which means that the training score is good but the validation score is poor.




```
print(__doc__)

import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import load_digits
from sklearn.svm import SVC
from sklearn.model_selection import validation_curve

X, y = load_digits(return_X_y=True)

param_range = np.logspace(-6, -1, 5)
train_scores, test_scores = validation_curve(
    SVC(), X, y, param_name="gamma", param_range=param_range,
    scoring="accuracy", n_jobs=1)
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

plt.title("Validation Curve with SVM")
plt.xlabel(r"$\gamma$")
plt.ylabel("Score")
plt.ylim(0.0, 1.1)
lw = 2
```

(continues on next page)

(continued from previous page)

```
plt.semilogx(param_range, train_scores_mean, label="Training score",
             color="darkorange", lw=lw)
plt.fill_between(param_range, train_scores_mean - train_scores_std,
                 train_scores_mean + train_scores_std, alpha=0.2,
                 color="darkorange", lw=lw)
plt.semilogx(param_range, test_scores_mean, label="Cross-validation score",
             color="navy", lw=lw)
plt.fill_between(param_range, test_scores_mean - test_scores_std,
                 test_scores_mean + test_scores_std, alpha=0.2,
                 color="navy", lw=lw)
plt.legend(loc="best")
plt.show()
```

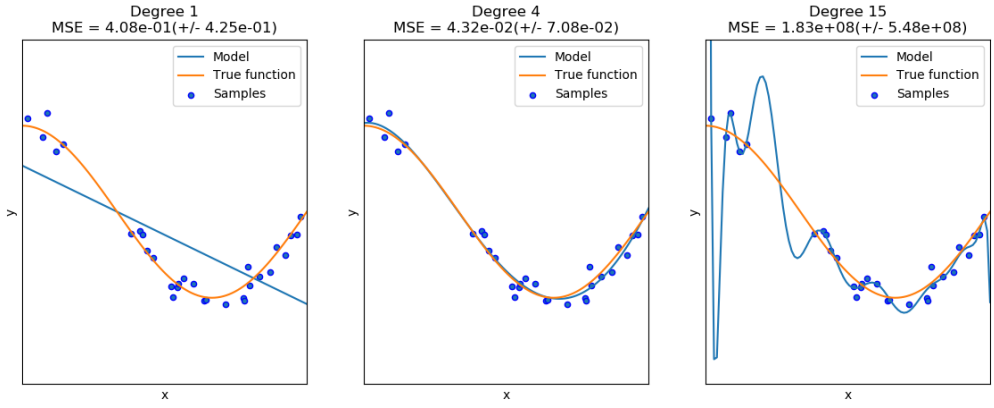
Total running time of the script: (0 minutes 13.827 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.4 Underfitting vs. Overfitting

This example demonstrates the problems of underfitting and overfitting and how we can use linear regression with polynomial features to approximate nonlinear functions. The plot shows the function that we want to approximate, which is a part of the cosine function. In addition, the samples from the real function and the approximations of different models are displayed. The models have polynomial features of different degrees. We can see that a linear function (polynomial with degree 1) is not sufficient to fit the training samples. This is called **underfitting**. A polynomial of degree 4 approximates the true function almost perfectly. However, for higher degrees the model will **overfit** the training data, i.e. it learns the noise of the training data. We evaluate quantitatively **overfitting / underfitting** by using cross-validation. We calculate the mean squared error (MSE) on the validation set, the higher, the less likely the model generalizes correctly from the training data.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
```

(continues on next page)

(continued from previous page)

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score


def true_fun(X):
    return np.cos(1.5 * np.pi * X)

np.random.seed(0)

n_samples = 30
degrees = [1, 4, 15]

X = np.sort(np.random.rand(n_samples))
y = true_fun(X) + np.random.randn(n_samples) * 0.1

plt.figure(figsize=(14, 5))
for i in range(len(degrees)):
    ax = plt.subplot(1, len(degrees), i + 1)
    plt.setp(ax, xticks=(), yticks=())

    polynomial_features = PolynomialFeatures(degree=degrees[i],
                                             include_bias=False)
    linear_regression = LinearRegression()
    pipeline = Pipeline([("polynomial_features", polynomial_features),
                         ("linear_regression", linear_regression)])
    pipeline.fit(X[:, np.newaxis], y)

    # Evaluate the models using crossvalidation
    scores = cross_val_score(pipeline, X[:, np.newaxis], y,
                             scoring="neg_mean_squared_error", cv=10)

    X_test = np.linspace(0, 1, 100)
    plt.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label="Model")
    plt.plot(X_test, true_fun(X_test), label="True function")
    plt.scatter(X, y, edgecolor='b', s=20, label="Samples")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.xlim((0, 1))
    plt.ylim((-2, 2))
    plt.legend(loc="best")
    plt.title("Degree {}\nMSE = {:.2e}(+/- {:.2e})".format(
        degrees[i], -scores.mean(), scores.std()))
plt.show()
```

Total running time of the script: (0 minutes 0.582 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.5 Parameter estimation using grid search with cross-validation

This examples shows how a classifier is optimized by cross-validation, which is done using the *sklearn.model_selection.GridSearchCV* object on a development set that comprises only half of the available labeled

data.

The performance of the selected hyper-parameters and trained model is then measured on a dedicated evaluation set that was not used during the model selection step.

More details on tools available for model selection can be found in the sections on *Cross-validation: evaluating estimator performance* and *Tuning the hyper-parameters of an estimator*.

Out:

```
# Tuning hyper-parameters for precision

Best parameters set found on development set:

{'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}

Grid scores on development set:

0.986 (+/-0.016) for {'C': 1, 'gamma': 0.001, 'kernel': 'rbf'}
0.959 (+/-0.028) for {'C': 1, 'gamma': 0.0001, 'kernel': 'rbf'}
0.988 (+/-0.017) for {'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
0.982 (+/-0.026) for {'C': 10, 'gamma': 0.0001, 'kernel': 'rbf'}
0.988 (+/-0.017) for {'C': 100, 'gamma': 0.001, 'kernel': 'rbf'}
0.983 (+/-0.026) for {'C': 100, 'gamma': 0.0001, 'kernel': 'rbf'}
0.988 (+/-0.017) for {'C': 1000, 'gamma': 0.001, 'kernel': 'rbf'}
0.983 (+/-0.026) for {'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}
0.974 (+/-0.012) for {'C': 1, 'kernel': 'linear'}
0.974 (+/-0.012) for {'C': 10, 'kernel': 'linear'}
0.974 (+/-0.012) for {'C': 100, 'kernel': 'linear'}
0.974 (+/-0.012) for {'C': 1000, 'kernel': 'linear'}

Detailed classification report:

The model is trained on the full development set.
The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

           0       1.00      1.00      1.00        89
           1       0.97      1.00      0.98        90
           2       0.99      0.98      0.98        92
           3       1.00      0.99      0.99        93
           4       1.00      1.00      1.00        76
           5       0.99      0.98      0.99       108
           6       0.99      1.00      0.99        89
           7       0.99      1.00      0.99        78
           8       1.00      0.98      0.99        92
           9       0.99      0.99      0.99        92

    accuracy                           0.99       899
   macro avg       0.99      0.99      0.99       899
weighted avg       0.99      0.99      0.99       899


# Tuning hyper-parameters for recall

Best parameters set found on development set:

{'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
```

(continues on next page)

(continued from previous page)

```
Grid scores on development set:

0.986 (+/-0.019) for {'C': 1, 'gamma': 0.001, 'kernel': 'rbf'}
0.957 (+/-0.028) for {'C': 1, 'gamma': 0.0001, 'kernel': 'rbf'}
0.987 (+/-0.019) for {'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
0.981 (+/-0.028) for {'C': 10, 'gamma': 0.0001, 'kernel': 'rbf'}
0.987 (+/-0.019) for {'C': 100, 'gamma': 0.001, 'kernel': 'rbf'}
0.982 (+/-0.026) for {'C': 100, 'gamma': 0.0001, 'kernel': 'rbf'}
0.987 (+/-0.019) for {'C': 1000, 'gamma': 0.001, 'kernel': 'rbf'}
0.982 (+/-0.026) for {'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}
0.971 (+/-0.010) for {'C': 1, 'kernel': 'linear'}
0.971 (+/-0.010) for {'C': 10, 'kernel': 'linear'}
0.971 (+/-0.010) for {'C': 100, 'kernel': 'linear'}
0.971 (+/-0.010) for {'C': 1000, 'kernel': 'linear'}

Detailed classification report:

The model is trained on the full development set.
The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

           0       1.00      1.00      1.00        89
           1       0.97      1.00      0.98        90
           2       0.99      0.98      0.98        92
           3       1.00      0.99      0.99        93
           4       1.00      1.00      1.00        76
           5       0.99      0.98      0.99       108
           6       0.99      1.00      0.99        89
           7       0.99      1.00      0.99        78
           8       1.00      0.98      0.99        92
           9       0.99      0.99      0.99        92

    accuracy                           0.99       899
   macro avg       0.99      0.99      0.99       899
weighted avg       0.99      0.99      0.99       899
```

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.svm import SVC

print(__doc__)

# Loading the Digits dataset
digits = datasets.load_digits()

# To apply an classifier on this data, we need to flatten the image, to
# turn the data in a (samples, feature) matrix:
```

(continues on next page)

(continued from previous page)

```
n_samples = len(digits.images)
X = digits.images.reshape((n_samples, -1))
y = digits.target

# Split the dataset in two equal parts
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.5, random_state=0)

# Set the parameters by cross-validation
tuned_parameters = [{'kernel': ['rbf'], 'gamma': [1e-3, 1e-4],
                     'C': [1, 10, 100, 1000]},
                    {'kernel': ['linear'], 'C': [1, 10, 100, 1000]}]

scores = ['precision', 'recall']

for score in scores:
    print("# Tuning hyper-parameters for %s" % score)
    print()

    clf = GridSearchCV(
        SVC(), tuned_parameters, scoring='%s_macro' % score
    )
    clf.fit(X_train, y_train)

    print("Best parameters set found on development set:")
    print()
    print(clf.best_params_)
    print()
    print("Grid scores on development set:")
    print()
    means = clf.cv_results_['mean_test_score']
    stds = clf.cv_results_['std_test_score']
    for mean, std, params in zip(means, stds, clf.cv_results_['params']):
        print("%0.3f (+/-%0.03f) for %r"
              % (mean, std * 2, params))
    print()

    print("Detailed classification report:")
    print()
    print("The model is trained on the full development set.")
    print("The scores are computed on the full evaluation set.")
    print()
    y_true, y_pred = y_test, clf.predict(X_test)
    print(classification_report(y_true, y_pred))
    print()

# Note the problem is too easy: the hyperparameter plateau is too flat and the
# output model is the same for precision and recall with ties in quality.
```

Total running time of the script: (0 minutes 4.512 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.6 Comparing randomized search and grid search for hyperparameter estimation

Compare randomized search and grid search for optimizing hyperparameters of a random forest. All parameters that influence the learning are searched simultaneously (except for the number of estimators, which poses a time / quality tradeoff).

The randomized search and the grid search explore exactly the same space of parameters. The result in parameter settings is quite similar, while the run time for randomized search is drastically lower.

The performance is may slightly worse for the randomized search, and is likely due to a noise effect and would not carry over to a held-out test set.

Note that in practice, one would not search over this many different parameters simultaneously using grid search, but pick only the ones deemed most important.

Out:

```
RandomizedSearchCV took 19.80 seconds for 20 candidates parameter settings.
Model with rank: 1
Mean validation score: 0.920 (std: 0.028)
Parameters: {'alpha': 0.07316411520495676, 'average': False, 'l1_ratio': 0.
↪29007760721044407}

Model with rank: 2
Mean validation score: 0.920 (std: 0.029)
Parameters: {'alpha': 0.0005223493320259539, 'average': True, 'l1_ratio': 0.
↪7936977033574206}

Model with rank: 3
Mean validation score: 0.918 (std: 0.031)
Parameters: {'alpha': 0.00025790124268693137, 'average': True, 'l1_ratio': 0.
↪5699649107012649}

GridSearchCV took 113.86 seconds for 100 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.931 (std: 0.026)
Parameters: {'alpha': 0.0001, 'average': True, 'l1_ratio': 0.0}

Model with rank: 2
Mean validation score: 0.928 (std: 0.030)
Parameters: {'alpha': 0.0001, 'average': True, 'l1_ratio': 0.1111111111111111}

Model with rank: 3
Mean validation score: 0.927 (std: 0.026)
Parameters: {'alpha': 0.0001, 'average': True, 'l1_ratio': 0.5555555555555556}
```

```
print(__doc__)

import numpy as np

from time import time
import scipy.stats as stats
```

(continues on next page)

(continued from previous page)

```
from sklearn.utils.fixes import loguniform

from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.datasets import load_digits
from sklearn.linear_model import SGDClassifier

# get some data
X, y = load_digits(return_X_y=True)

# build a classifier
clf = SGDClassifier(loss='hinge', penalty='elasticnet',
                    fit_intercept=True)


# Utility function to report best scores
def report(results, n_top=3):
    for i in range(1, n_top + 1):
        candidates = np.flatnonzero(results['rank_test_score'] == i)
        for candidate in candidates:
            print("Model with rank: {0}".format(i))
            print("Mean validation score: {0:.3f} (std: {1:.3f})"
                  .format(results['mean_test_score'][candidate],
                          results['std_test_score'][candidate]))
            print("Parameters: {0}".format(results['params'][candidate]))
            print("")


# specify parameters and distributions to sample from
param_dist = {'average': [True, False],
              'l1_ratio': stats.uniform(0, 1),
              'alpha': loguniform(1e-4, 1e0)}

# run randomized search
n_iter_search = 20
random_search = RandomizedSearchCV(clf, param_distributions=param_dist,
                                   n_iter=n_iter_search)

start = time()
random_search.fit(X, y)
print("RandomizedSearchCV took %.2f seconds for %d candidates"
      " parameter settings." % ((time() - start), n_iter_search))
report(random_search.cv_results_)

# use a full grid over all parameters
param_grid = {'average': [True, False],
              'l1_ratio': np.linspace(0, 1, num=10),
              'alpha': np.power(10, np.arange(-4, 1, dtype=float))}

# run grid search
grid_search = GridSearchCV(clf, param_grid=param_grid)
start = time()
grid_search.fit(X, y)

print("GridSearchCV took %.2f seconds for %d candidate parameter settings."
      % (time() - start, len(grid_search.cv_results_['params'])))
report(grid_search.cv_results_)
```

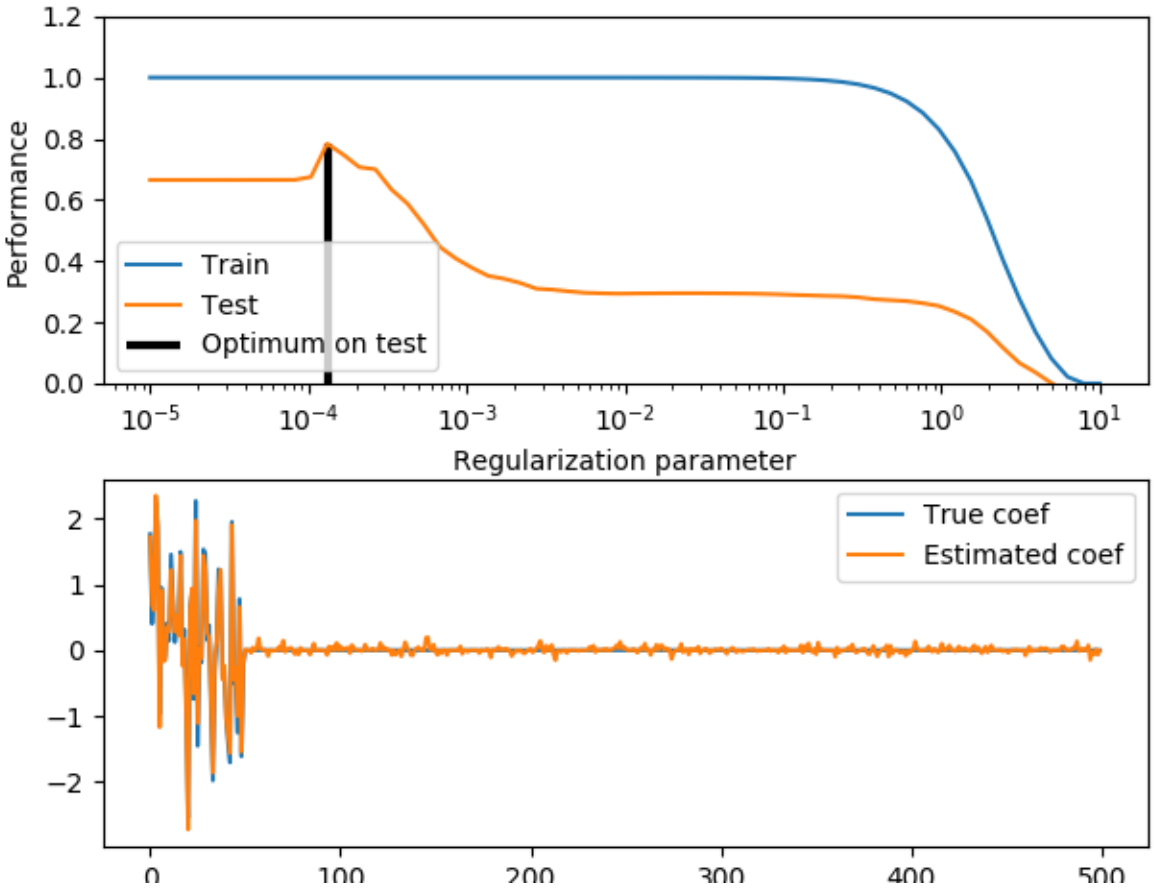
Total running time of the script: (2 minutes 13.989 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.7 Train error vs Test error

Illustration of how the performance of an estimator on unseen data (test data) is not the same as the performance on training data. As the regularization increases the performance on train decreases while the performance on test is optimal within a range of values of the regularization parameter. The example with an Elastic-Net regression model and the performance is measured using the explained variance a.k.a. R^2.



Out:

```
Optimal regularization parameter : 0.00013141473626117567
```

```
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

import numpy as np
from sklearn import linear_model

# #############################################################################
# Generate sample data
n_samples_train, n_samples_test, n_features = 75, 150, 500
np.random.seed(0)
coef = np.random.randn(n_features)
coef[50:] = 0.0  # only the top 10 features are impacting the model
X = np.random.randn(n_samples_train + n_samples_test, n_features)
y = np.dot(X, coef)

# Split train and test data
X_train, X_test = X[:n_samples_train], X[n_samples_train:]
y_train, y_test = y[:n_samples_train], y[n_samples_train:]

# #############################################################################
# Compute train and test errors
alphas = np.logspace(-5, 1, 60)
enet = linear_model.ElasticNet(l1_ratio=0.7, max_iter=10000)
train_errors = list()
test_errors = list()
for alpha in alphas:
    enet.set_params(alpha=alpha)
    enet.fit(X_train, y_train)
    train_errors.append(enet.score(X_train, y_train))
    test_errors.append(enet.score(X_test, y_test))

i_alpha_optim = np.argmax(test_errors)
alpha_optim = alphas[i_alpha_optim]
print("Optimal regularization parameter : %s" % alpha_optim)

# Estimate the coef_ on full data with optimal regularization parameter
enet.set_params(alpha=alpha_optim)
coef_ = enet.fit(X, y).coef_

# #############################################################################
# Plot results functions

import matplotlib.pyplot as plt
plt.subplot(2, 1, 1)
plt.semilogx(alphas, train_errors, label='Train')
plt.semilogx(alphas, test_errors, label='Test')
plt.vlines(alpha_optim, plt.ylim()[0], np.max(test_errors), color='k',
           linewidth=3, label='Optimum on test')
plt.legend(loc='lower left')
plt.ylim([0, 1.2])
plt.xlabel('Regularization parameter')
plt.ylabel('Performance')

# Show estimated coef_ vs true coef
plt.subplot(2, 1, 2)
```

(continues on next page)

(continued from previous page)

```
plt.plot(coef, label='True coef')
plt.plot(coef_, label='Estimated coef')
plt.legend()
plt.subplots_adjust(0.09, 0.04, 0.94, 0.94, 0.26, 0.26)
plt.show()
```

Total running time of the script: (0 minutes 2.775 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.8 Receiver Operating Characteristic (ROC) with cross validation

Example of Receiver Operating Characteristic (ROC) metric to evaluate classifier output quality using cross-validation.

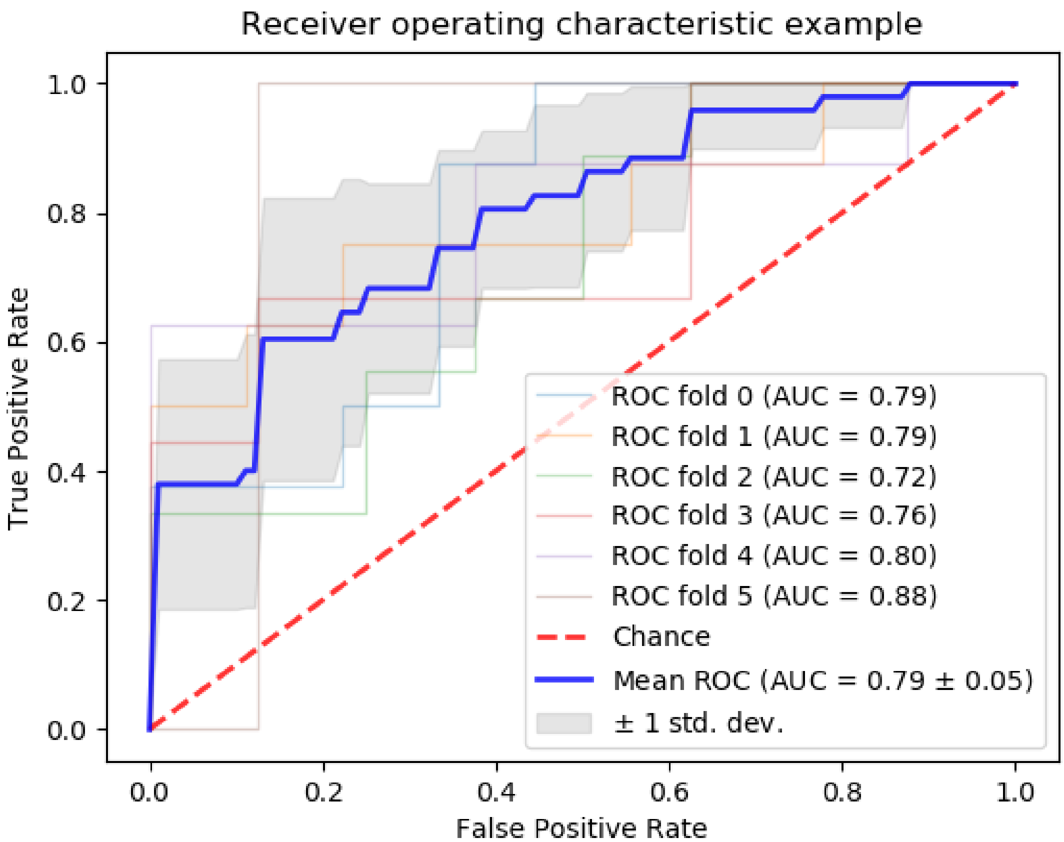
ROC curves typically feature true positive rate on the Y axis, and false positive rate on the X axis. This means that the top left corner of the plot is the "ideal" point - a false positive rate of zero, and a true positive rate of one. This is not very realistic, but it does mean that a larger area under the curve (AUC) is usually better.

The "steepness" of ROC curves is also important, since it is ideal to maximize the true positive rate while minimizing the false positive rate.

This example shows the ROC response of different datasets, created from K-fold cross-validation. Taking all of these curves, it is possible to calculate the mean area under curve, and see the variance of the curve when the training set is split into different subsets. This roughly shows how the classifier output is affected by changes in the training data, and how different the splits generated by K-fold cross-validation are from one another.

Note:

See also ***sklearn.metrics.roc_auc_score***, *sklearn.model_selection.cross_val_score*, *Receiver Operating Characteristic (ROC)*,



Out:

```
/home/circleci/project/examples/model_selection/plot_roc_crossval.py:75:␣
↪DeprecationWarning: scipy.interp is deprecated and will be removed in SciPy 2.0.0,␣
↪use numpy.interp instead
  interp_tpr = interp(mean_fpr, viz.fpr, viz.tpr)
/home/circleci/project/examples/model_selection/plot_roc_crossval.py:75:␣
↪DeprecationWarning: scipy.interp is deprecated and will be removed in SciPy 2.0.0,␣
↪use numpy.interp instead
  interp_tpr = interp(mean_fpr, viz.fpr, viz.tpr)
/home/circleci/project/examples/model_selection/plot_roc_crossval.py:75:␣
↪DeprecationWarning: scipy.interp is deprecated and will be removed in SciPy 2.0.0,␣
↪use numpy.interp instead
  interp_tpr = interp(mean_fpr, viz.fpr, viz.tpr)
/home/circleci/project/examples/model_selection/plot_roc_crossval.py:75:␣
↪DeprecationWarning: scipy.interp is deprecated and will be removed in SciPy 2.0.0,␣
↪use numpy.interp instead
  interp_tpr = interp(mean_fpr, viz.fpr, viz.tpr)
/home/circleci/project/examples/model_selection/plot_roc_crossval.py:75:␣
↪DeprecationWarning: scipy.interp is deprecated and will be removed in SciPy 2.0.0,␣
↪use numpy.interp instead
  interp_tpr = interp(mean_fpr, viz.fpr, viz.tpr)
/home/circleci/project/examples/model_selection/plot_roc_crossval.py:75:␣
↪DeprecationWarning: scipy.interp is deprecated and will be removed in SciPy 2.0.0,␣
↪use numpy.interp instead
  interp_tpr = interp(mean_fpr, viz.fpr, viz.tpr)
```

```
print(__doc__)

import numpy as np
from scipy import interp
import matplotlib.pyplot as plt

from sklearn import svm, datasets
from sklearn.metrics import auc
from sklearn.metrics import plot_roc_curve
from sklearn.model_selection import StratifiedKFold

# #############################################################################
# Data IO and generation

# Import some data to play with
iris = datasets.load_iris()
X = iris.data
y = iris.target
X, y = X[y != 2], y[y != 2]
n_samples, n_features = X.shape

# Add noisy features
random_state = np.random.RandomState(0)
X = np.c_[X, random_state.randn(n_samples, 200 * n_features)]

# #############################################################################
# Classification and ROC analysis

# Run classifier with cross-validation and plot ROC curves
cv = StratifiedKFold(n_splits=6)
classifier = svm.SVC(kernel='linear', probability=True,
                     random_state=random_state)

tprs = []
aucs = []
mean_fpr = np.linspace(0, 1, 100)

fig, ax = plt.subplots()
for i, (train, test) in enumerate(cv.split(X, y)):
    classifier.fit(X[train], y[train])
    viz = plot_roc_curve(classifier, X[test], y[test],
                         name='ROC fold {}'.format(i),
                         alpha=0.3, lw=1, ax=ax)
    interp_tpr = interp(mean_fpr, viz.fpr, viz.tpr)
    interp_tpr[0] = 0.0
    tprs.append(interp_tpr)
    aucs.append(viz.roc_auc)

ax.plot([0, 1], [0, 1], linestyle='--', lw=2, color='r',
        label='Chance', alpha=.8)

mean_tpr = np.mean(tprs, axis=0)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
```

(continues on next page)

(continued from previous page)

```
std_auc = np.std(aucs)
ax.plot(mean_fpr, mean_tpr, color='b',
        label=r'Mean ROC (AUC = %0.2f $\pm$ %0.2f)' % (mean_auc, std_auc),
        lw=2, alpha=.8)

std_tpr = np.std(tprs, axis=0)
tprs_upper = np.minimum(mean_tpr + std_tpr, 1)
tprs_lower = np.maximum(mean_tpr - std_tpr, 0)
ax.fill_between(mean_fpr, tprs_lower, tprs_upper, color='grey', alpha=.2,
                label=r'$\pm$ 1 std. dev.')

ax.set(xlim=[-0.05, 1.05], ylim=[-0.05, 1.05],
       title="Receiver operating characteristic example")
ax.legend(loc="lower right")
plt.show()
```

Total running time of the script: (0 minutes 0.674 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.9 Nested versus non-nested cross-validation

This example compares non-nested and nested cross-validation strategies on a classifier of the iris data set. Nested cross-validation (CV) is often used to train a model in which hyperparameters also need to be optimized. Nested CV estimates the generalization error of the underlying model and its (hyper)parameter search. Choosing the parameters that maximize non-nested CV biases the model to the dataset, yielding an overly-optimistic score.

Model selection without nested CV uses the same data to tune model parameters and evaluate model performance. Information may thus "leak" into the model and overfit the data. The magnitude of this effect is primarily dependent on the size of the dataset and the stability of the model. See Cawley and Talbot[1] for an analysis of these issues.

To avoid this problem, nested CV effectively uses a series of train/validation/test set splits. In the inner loop (here executed by `GridSearchCV`), the score is approximately maximized by fitting a model to each training set, and then directly maximized in selecting (hyper)parameters over the validation set. In the outer loop (here in `cross_val_score`), generalization error is estimated by averaging test set scores over several dataset splits.

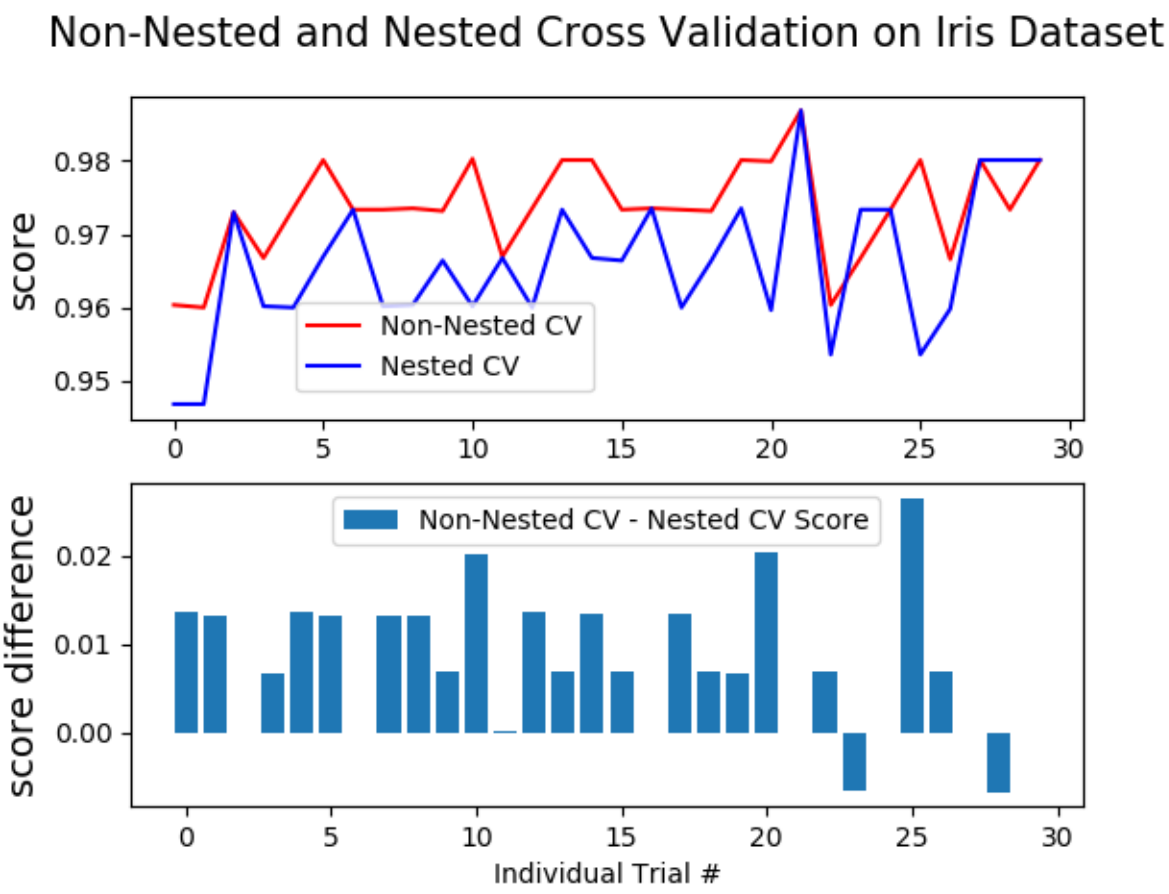
The example below uses a support vector classifier with a non-linear kernel to build a model with optimized hyperparameters by grid search. We compare the performance of non-nested and nested CV strategies by taking the difference between their scores.

See Also:

- *Cross-validation: evaluating estimator performance*
- *Tuning the hyper-parameters of an estimator*

[1] Cawley, G.C.; Talbot, N.L.C. On over-fitting in model selection and subsequent selection bias in performance evaluation. J. Mach. Learn. Res 2010,11, 2079-2107.

References:



Out:

```
Average difference of 0.007581 with std. dev. of 0.007833.
```

```
from sklearn.datasets import load_iris
from matplotlib import pyplot as plt
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV, cross_val_score, KFold
import numpy as np

print(__doc__)

# Number of random trials
NUM_TRIALS = 30

# Load the dataset
iris = load_iris()
```

(continues on next page)

(continued from previous page)

```
X_iris = iris.data
y_iris = iris.target

# Set up possible values of parameters to optimize over
p_grid = {"C": [1, 10, 100],
          "gamma": [.01, .1]}

# We will use a Support Vector Classifier with "rbf" kernel
svm = SVC(kernel="rbf")

# Arrays to store scores
non_nested_scores = np.zeros(NUM_TRIALS)
nested_scores = np.zeros(NUM_TRIALS)

# Loop for each trial
for i in range(NUM_TRIALS):

    # Choose cross-validation techniques for the inner and outer loops,
    # independently of the dataset.
    # E.g "GroupKFold", "LeaveOneOut", "LeaveOneGroupOut", etc.
    inner_cv = KFold(n_splits=4, shuffle=True, random_state=i)
    outer_cv = KFold(n_splits=4, shuffle=True, random_state=i)

    # Non_nested parameter search and scoring
    clf = GridSearchCV(estimator=svm, param_grid=p_grid, cv=inner_cv)
    clf.fit(X_iris, y_iris)
    non_nested_scores[i] = clf.best_score_

    # Nested CV with parameter optimization
    nested_score = cross_val_score(clf, X=X_iris, y=y_iris, cv=outer_cv)
    nested_scores[i] = nested_score.mean()

score_difference = non_nested_scores - nested_scores

print("Average difference of {:6f} with std. dev. of {:6f}."
      .format(score_difference.mean(), score_difference.std()))

# Plot scores on each trial for nested and non-nested CV
plt.figure()
plt.subplot(211)
non_nested_scores_line, = plt.plot(non_nested_scores, color='r')
nested_line, = plt.plot(nested_scores, color='b')
plt.ylabel("score", fontsize="14")
plt.legend([non_nested_scores_line, nested_line],
           ["Non-Nested CV", "Nested CV"],
           bbox_to_anchor=(0, .4, .5, 0))
plt.title("Non-Nested and Nested Cross Validation on Iris Dataset",
          x=.5, y=1.1, fontsize="15")

# Plot bar chart of the difference.
plt.subplot(212)
difference_plot = plt.bar(range(NUM_TRIALS), score_difference)
plt.xlabel("Individual Trial #")
plt.legend([difference_plot],
           ["Non-Nested CV - Nested CV Score"],
           bbox_to_anchor=(0, 1, .8, 0))
plt.ylabel("score difference", fontsize="14")
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

Total running time of the script: (0 minutes 5.144 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.10 Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV

Multiple metric parameter search can be done by setting the `scoring` parameter to a list of metric scorer names or a dict mapping the scorer names to the scorer callables.

The scores of all the scorers are available in the `cv_results_` dict at keys ending in `'_<scorer_name>'` (`'mean_test_precision'`, `'rank_test_precision'`, etc...)

The `best_estimator_`, `best_index_`, `best_score_` and `best_params_` correspond to the scorer (key) that is set to the `refit` attribute.

```
# Author: Raghav RV <rvraghav93@gmail.com>
# License: BSD

import numpy as np
from matplotlib import pyplot as plt

from sklearn.datasets import make_hastie_10_2
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier

print(__doc__)
```

Running `GridSearchCV` using multiple evaluation metrics

```
X, y = make_hastie_10_2(n_samples=8000, random_state=42)

# The scorers can be either be one of the predefined metric strings or a scorer
# callable, like the one returned by make_scorer
scoring = {'AUC': 'roc_auc', 'Accuracy': make_scorer(accuracy_score)}

# Setting refit='AUC', refits an estimator on the whole dataset with the
# parameter setting that has the best cross-validated AUC score.
# That estimator is made available at ``gs.best_estimator_`` along with
# parameters like ``gs.best_score_``, ``gs.best_params_`` and
# ``gs.best_index_``
gs = GridSearchCV(DecisionTreeClassifier(random_state=42),
                  param_grid={'min_samples_split': range(2, 403, 10)},
                  scoring=scoring, refit='AUC', return_train_score=True)
gs.fit(X, y)
results = gs.cv_results_
```

Plotting the result

```
plt.figure(figsize=(13, 13))
plt.title("GridSearchCV evaluating using multiple scorers simultaneously",
          fontsize=16)

plt.xlabel("min_samples_split")
plt.ylabel("Score")

ax = plt.gca()
ax.set_xlim(0, 402)
ax.set_ylim(0.73, 1)

# Get the regular numpy array from the MaskedArray
X_axis = np.array(results['param_min_samples_split'].data, dtype=float)

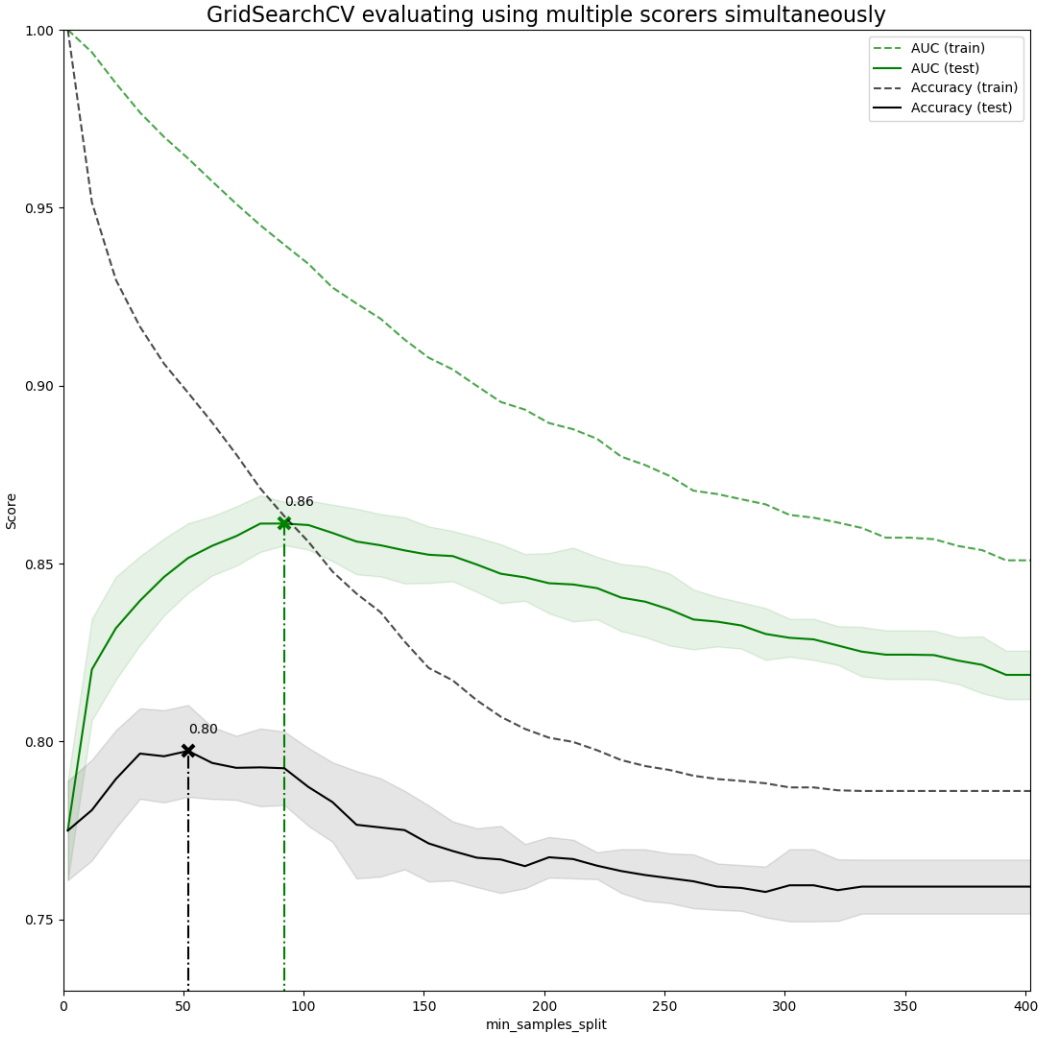
for scorer, color in zip(sorted(scoring), ['g', 'k']):
    for sample, style in (('train', '--'), ('test', '-')):
        sample_score_mean = results['mean_%s_%s' % (sample, scorer)]
        sample_score_std = results['std_%s_%s' % (sample, scorer)]
        ax.fill_between(X_axis, sample_score_mean - sample_score_std,
                        sample_score_mean + sample_score_std,
                        alpha=0.1 if sample == 'test' else 0, color=color)
        ax.plot(X_axis, sample_score_mean, style, color=color,
                alpha=1 if sample == 'test' else 0.7,
                label="%s (%s)" % (scorer, sample))

    best_index = np.nonzero(results['rank_test_%s' % scorer] == 1)[0][0]
    best_score = results['mean_test_%s' % scorer][best_index]

    # Plot a dotted vertical line at the best score for that scorer marked by x
    ax.plot([X_axis[best_index], ] * 2, [0, best_score],
            linestyle='-.', color=color, marker='x', markeredgewidth=3, ms=8)

    # Annotate the best score for that scorer
    ax.annotate("%0.2f" % best_score,
                (X_axis[best_index], best_score + 0.005))

plt.legend(loc="best")
plt.grid(False)
plt.show()
```



Total running time of the script: (0 minutes 24.387 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.11 Sample pipeline for text feature extraction and evaluation

The dataset used in this example is the 20 newsgroups dataset which will be automatically downloaded and then cached and reused for the document classification example.

You can adjust the number of categories by giving their names to the dataset loader or setting them to None to get the 20 of them.

Here is a sample output of a run on a quad-core machine:

```
Loading 20 newsgroups dataset for categories:
['alt.atheism', 'talk.religion.misc']
1427 documents
2 categories

Performing grid search...
pipeline: ['vect', 'tfidf', 'clf']
parameters:
{'clf__alpha': (1.0000000000000001e-05, 9.9999999999999995e-07),
 'clf__max_iter': (10, 50, 80),
 'clf__penalty': ('l2', 'elasticnet'),
 'tfidf__use_idf': (True, False),
 'vect__max_n': (1, 2),
 'vect__max_df': (0.5, 0.75, 1.0),
 'vect__max_features': (None, 5000, 10000, 50000)}
done in 1737.030s

Best score: 0.940
Best parameters set:
    clf__alpha: 9.9999999999999995e-07
    clf__max_iter: 50
    clf__penalty: 'elasticnet'
    tfidf__use_idf: True
    vect__max_n: 2
    vect__max_df: 0.75
    vect__max_features: 50000
```

```
# Author: Olivier Grisel <olivier.grisel@ensta.org>
#         Peter Prettenhofer <peter.prettenhofer@gmail.com>
#         Mathieu Blondel <mathieu@mblondel.org>
# License: BSD 3 clause
from pprint import pprint
from time import time
import logging

from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline

print(__doc__)

# Display progress logs on stdout
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s %(message)s')


# ##############################################################################
# Load some categories from the training set
categories = [
    'alt.atheism',
    'talk.religion.misc',
]
# Uncomment the following to do the analysis on all the categories
```

(continues on next page)

(continued from previous page)

```
#categories = None

print("Loading 20 newsgroups dataset for categories:")
print(categories)

data = fetch_20newsgroups(subset='train', categories=categories)
print("%d documents" % len(data.filenames))
print("%d categories" % len(data.target_names))
print()

# ##############################################################################
# Define a pipeline combining a text feature extractor with a simple
# classifier
pipeline = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('clf', SGDClassifier()),
])

# uncommenting more parameters will give better exploring power but will
# increase processing time in a combinatorial way
parameters = {
    'vect__max_df': (0.5, 0.75, 1.0),
    # 'vect__max_features': (None, 5000, 10000, 50000),
    'vect__ngram_range': ((1, 1), (1, 2)),  # unigrams or bigrams
    # 'tfidf__use_idf': (True, False),
    # 'tfidf__norm': ('l1', 'l2'),
    'clf__max_iter': (20,),
    'clf__alpha': (0.00001, 0.000001),
    'clf__penalty': ('l2', 'elasticnet'),
    # 'clf__max_iter': (10, 50, 80),
}

if __name__ == "__main__":
    # multiprocessing requires the fork to happen in a __main__ protected
    # block

    # find the best parameters for both the feature extraction and the
    # classifier
    grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1, verbose=1)

    print("Performing grid search...")
    print("pipeline:", [name for name, _ in pipeline.steps])
    print("parameters:")
    pprint(parameters)
    t0 = time()
    grid_search.fit(data.data, data.target)
    print("done in %0.3fs" % (time() - t0))
    print()

    print("Best score: %0.3f" % grid_search.best_score_)
    print("Best parameters set:")
    best_parameters = grid_search.best_estimator_.get_params()
    for param_name in sorted(parameters.keys()):
        print("\t%s: %r" % (param_name, best_parameters[param_name]))
```

Total running time of the script: (0 minutes 0.000 seconds)

Estimated memory usage: 0 MB

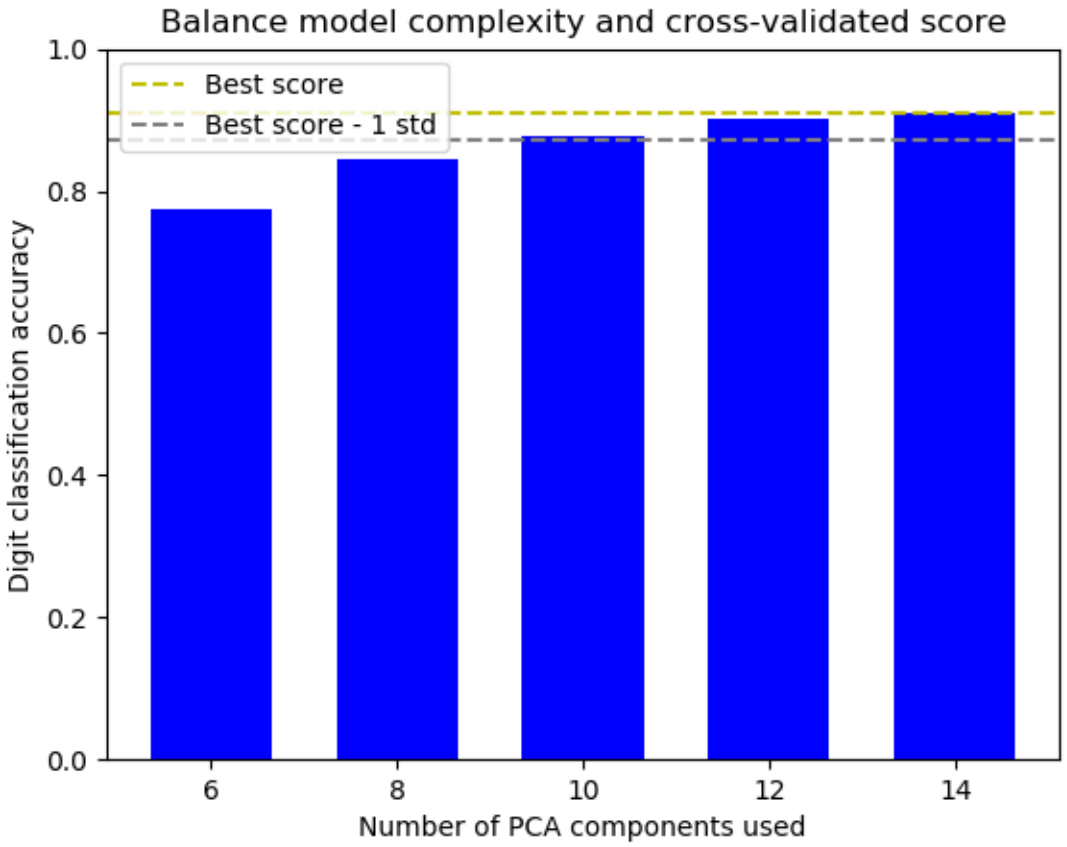
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.12 Balance model complexity and cross-validated score

This example balances model complexity and cross-validated score by finding a decent accuracy within 1 standard deviation of the best accuracy score while minimising the number of PCA components [1].

The figure shows the trade-off between cross-validated score and the number of PCA components. The balanced case is when n_components=10 and accuracy=0.88, which falls into the range within 1 standard deviation of the best accuracy score.

[1] Hastie, T., Tibshirani, R.,, Friedman, J. (2001). Model Assessment and Selection. The Elements of Statistical Learning (pp. 219-260). New York, NY, USA: Springer New York Inc..



Out:

```
The best_index_ is 2
The n_components selected is 10
The corresponding accuracy score is 0.88
```

```
# Author: Wenhao Zhang <wenhaoz@ucla.edu>

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC


def lower_bound(cv_results):
    """
    Calculate the lower bound within 1 standard deviation
    of the best `mean_test_scores`.

    Parameters
    ----------
    cv_results : dict of numpy(masked) ndarrays
        See attribute cv_results_ of `GridSearchCV`

    Returns
    -------
    float
        Lower bound within 1 standard deviation of the
        best `mean_test_score`.
    """
    best_score_idx = np.argmax(cv_results['mean_test_score'])

    return (cv_results['mean_test_score'][best_score_idx]
            - cv_results['std_test_score'][best_score_idx])


def best_low_complexity(cv_results):
    """
    Balance model complexity with cross-validated score.

    Parameters
    ----------
    cv_results : dict of numpy(masked) ndarrays
        See attribute cv_results_ of `GridSearchCV`.

    Return
    ------
    int
        Index of a model that has the fewest PCA components
        while has its test score within 1 standard deviation of the best
        `mean_test_score`.
    """
    threshold = lower_bound(cv_results)
    candidate_idx = np.flatnonzero(cv_results['mean_test_score'] >= threshold)
```

(continues on next page)

(continued from previous page)

```
    best_idx = candidate_idx[cv_results['param_reduce_dim__n_components']
                             [candidate_idx].argmin()]
    return best_idx


pipe = Pipeline([
        ('reduce_dim', PCA(random_state=42)),
        ('classify', LinearSVC(random_state=42, C=0.01)),
])

param_grid = {
    'reduce_dim__n_components': [6, 8, 10, 12, 14]
}

grid = GridSearchCV(pipe, cv=10, n_jobs=1, param_grid=param_grid,
                    scoring='accuracy', refit=best_low_complexity)
X, y = load_digits(return_X_y=True)
grid.fit(X, y)

n_components = grid.cv_results_['param_reduce_dim__n_components']
test_scores = grid.cv_results_['mean_test_score']

plt.figure()
plt.bar(n_components, test_scores, width=1.3, color='b')

lower = lower_bound(grid.cv_results_)
plt.axhline(np.max(test_scores), linestyle='--', color='y',
            label='Best score')
plt.axhline(lower, linestyle='--', color='.5', label='Best score - 1 std')

plt.title("Balance model complexity and cross-validated score")
plt.xlabel('Number of PCA components used')
plt.ylabel('Digit classification accuracy')
plt.xticks(n_components.tolist())
plt.ylim((0, 1.0))
plt.legend(loc='upper left')

best_index_ = grid.best_index_

print("The best_index_ is %d" % best_index_)
print("The n_components selected is %d" % n_components[best_index_])
print("The corresponding accuracy score is %.2f"
      % grid.cv_results_['mean_test_score'][best_index_])
plt.show()
```

Total running time of the script: (0 minutes 5.575 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.13 Visualizing cross-validation behavior in scikit-learn

Choosing the right cross-validation object is a crucial part of fitting a model properly. There are many ways to split data into training and test sets in order to avoid model overfitting, to standardize the number of groups in test sets, etc.

This example visualizes the behavior of several common scikit-learn objects for comparison.

```
from sklearn.model_selection import (TimeSeriesSplit, KFold, ShuffleSplit,
                                     StratifiedKFold, GroupShuffleSplit,
                                     GroupKFold, StratifiedShuffleSplit)
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Patch
np.random.seed(1338)
cmap_data = plt.cm.Paired
cmap_cv = plt.cm.coolwarm
n_splits = 4
```

Visualize our data

First, we must understand the structure of our data. It has 100 randomly generated input datapoints, 3 classes split unevenly across datapoints, and 10 "groups" split evenly across datapoints.

As we'll see, some cross-validation objects do specific things with labeled data, others behave differently with grouped data, and others do not use this information.

To begin, we'll visualize our data.

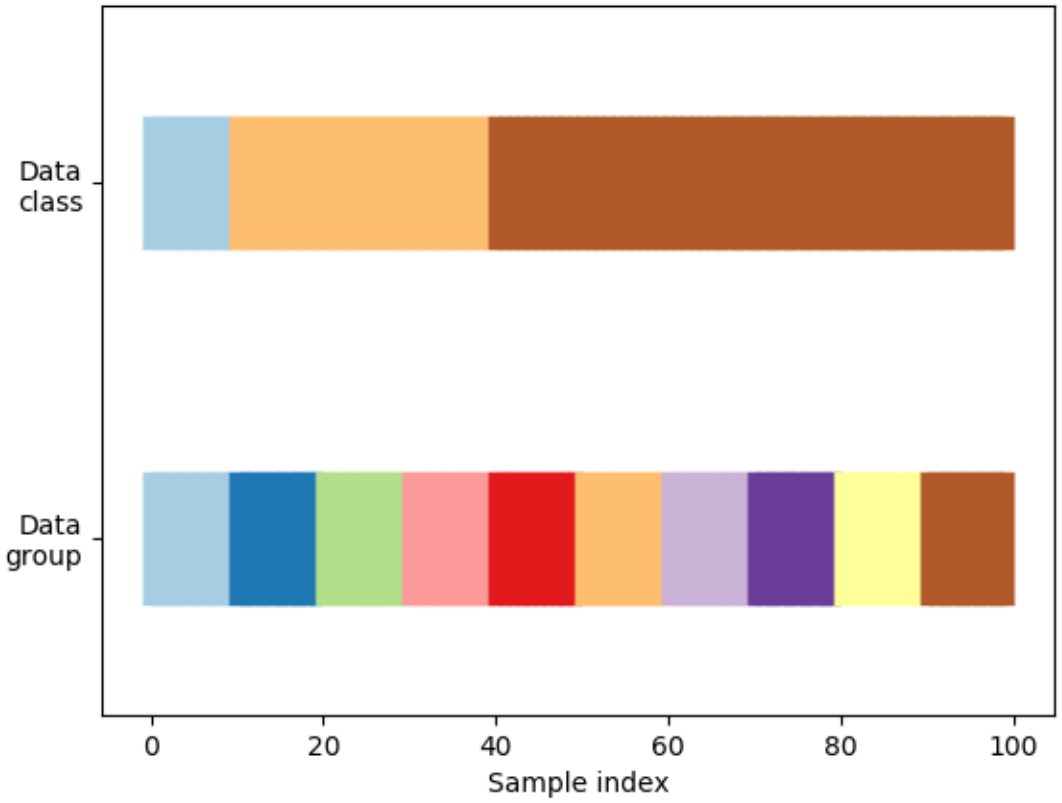
```
# Generate the class/group data
n_points = 100
X = np.random.randn(100, 10)

percentiles_classes = [.1, .3, .6]
y = np.hstack([[ii] * int(100 * perc)
               for ii, perc in enumerate(percentiles_classes)])

# Evenly spaced groups repeated once
groups = np.hstack([[ii] * 10 for ii in range(10)])


def visualize_groups(classes, groups, name):
    # Visualize dataset groups
    fig, ax = plt.subplots()
    ax.scatter(range(len(groups)),  [.5] * len(groups), c=groups, marker='_',
               lw=50, cmap=cmap_data)
    ax.scatter(range(len(groups)),  [3.5] * len(groups), c=classes, marker='_',
               lw=50, cmap=cmap_data)
    ax.set(ylim=[-1, 5], yticks=[.5, 3.5],
           yticklabels=['Data\ngroup', 'Data\nclass'], xlabel="Sample index")


visualize_groups(y, groups, 'no groups')
```



Define a function to visualize cross-validation behavior

We'll define a function that lets us visualize the behavior of each cross-validation object. We'll perform 4 splits of the data. On each split, we'll visualize the indices chosen for the training set (in blue) and the test set (in red).

```
def plot_cv_indices(cv, X, y, group, ax, n_splits, lw=10):
    """Create a sample plot for indices of a cross-validation object."""

    # Generate the training/testing visualizations for each CV split
    for ii, (tr, tt) in enumerate(cv.split(X=X, y=y, groups=group)):
        # Fill in indices with the training/test groups
        indices = np.array([np.nan] * len(X))
        indices[tt] = 1
        indices[tr] = 0

        # Visualize the results
        ax.scatter(range(len(indices)), [ii + .5] * len(indices),
                   c=indices, marker='_', lw=lw, cmap=cmap_cv,
                   vmin=-.2, vmax=1.2)

    # Plot the data classes and groups at the end
    ax.scatter(range(len(X)), [ii + 1.5] * len(X),
               c=y, marker='_', lw=lw, cmap=cmap_data)
```

(continues on next page)

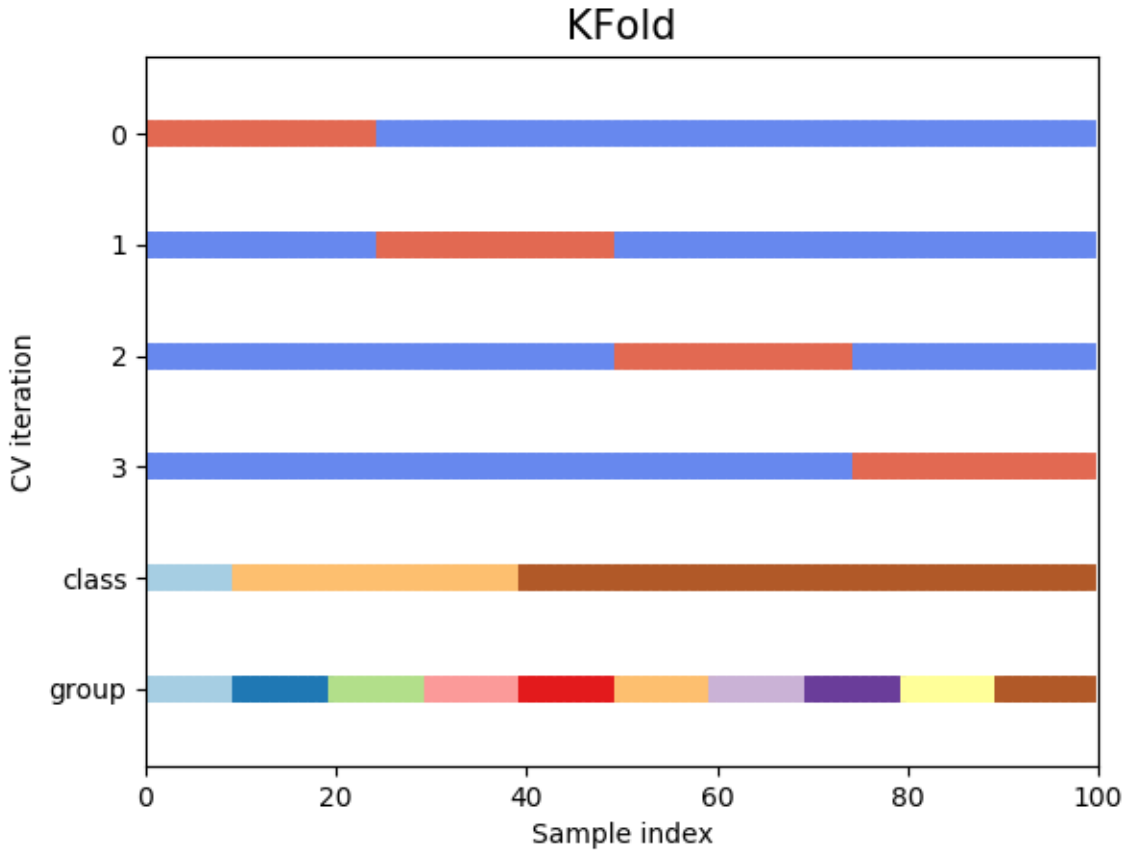
(continued from previous page)

```
    ax.scatter(range(len(X)), [ii + 2.5] * len(X),
               c=group, marker='_', lw=lw, cmap=cmap_data)

    # Formatting
    yticklabels = list(range(n_splits)) + ['class', 'group']
    ax.set(yticks=np.arange(n_splits+2) + .5, yticklabels=yticklabels,
           xlabel='Sample index', ylabel="CV iteration",
           ylim=[n_splits+2.2, -.2], xlim=[0, 100])
    ax.set_title('{}'.format(type(cv).__name__), fontsize=15)
    return ax
```

Let's see how it looks for the *KFold* cross-validation object:

```
fig, ax = plt.subplots()
cv = KFold(n_splits)
plot_cv_indices(cv, X, y, groups, ax, n_splits)
```

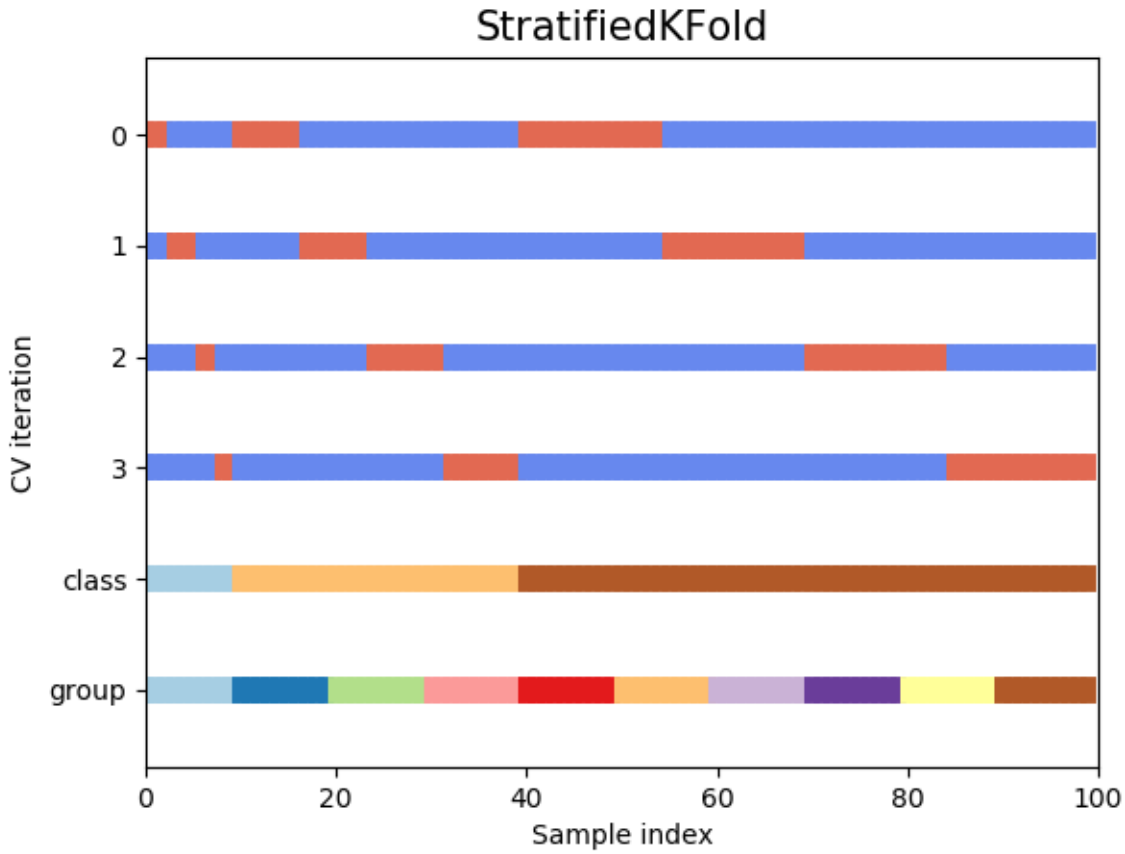


Out:

```
<matplotlib.axes._subplots.AxesSubplot object at 0x7ffa289f4b20>
```

As you can see, by default the KFold cross-validation iterator does not take either datapoint class or group into consideration. We can change this by using the `StratifiedKFold` like so.

```
fig, ax = plt.subplots()
cv = StratifiedKFold(n_splits)
plot_cv_indices(cv, X, y, groups, ax, n_splits)
```



Out:

```
<matplotlib.axes._subplots.AxesSubplot object at 0x7ffa269c2310>
```

In this case, the cross-validation retained the same ratio of classes across each CV split. Next we'll visualize this behavior for a number of CV iterators.

Visualize cross-validation indices for many CV objects

Let's visually compare the cross validation behavior for many scikit-learn cross-validation objects. Below we will loop through several common cross-validation objects, visualizing the behavior of each.

Note how some use the group/class information while others do not.

```
cvs = [KFold, GroupKFold, ShuffleSplit, StratifiedKFold,
       GroupShuffleSplit, StratifiedShuffleSplit, TimeSeriesSplit]

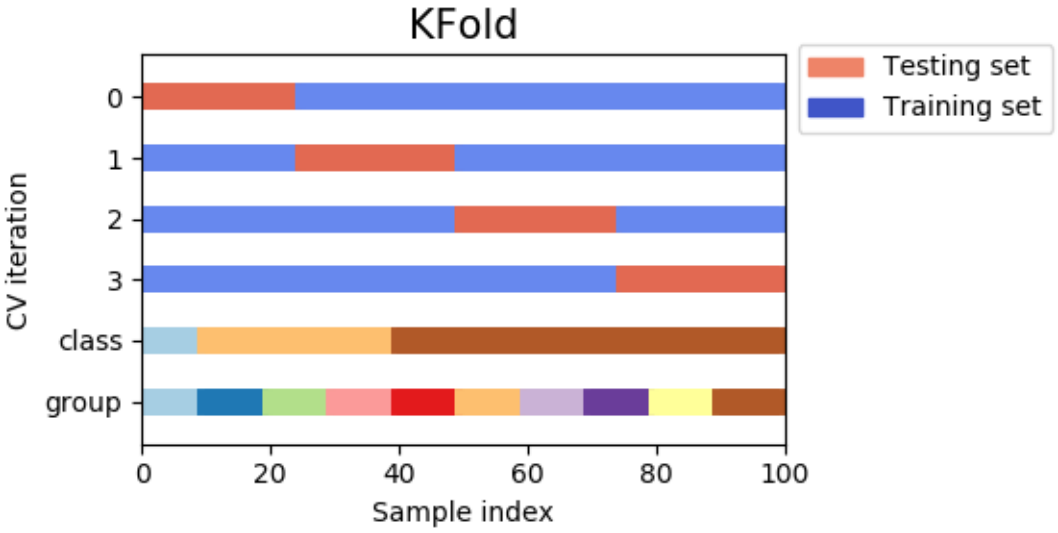

for cv in cvs:
    this_cv = cv(n_splits=n_splits)
```

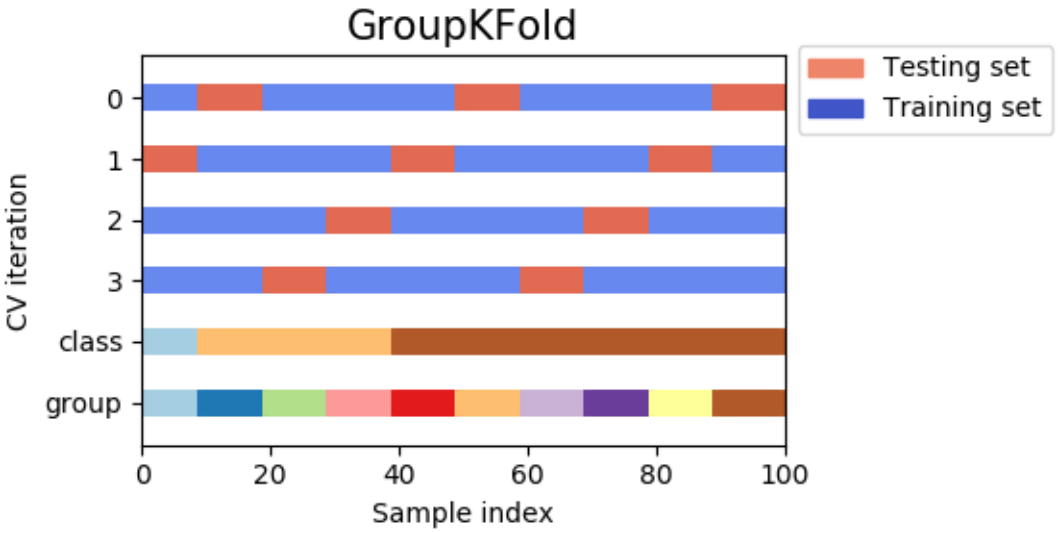
(continues on next page)

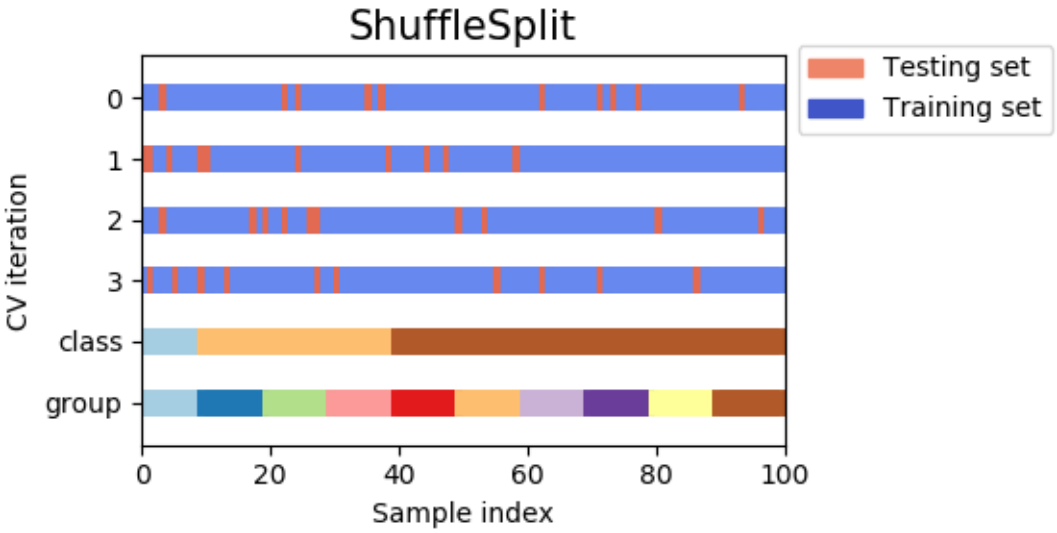
(continued from previous page)

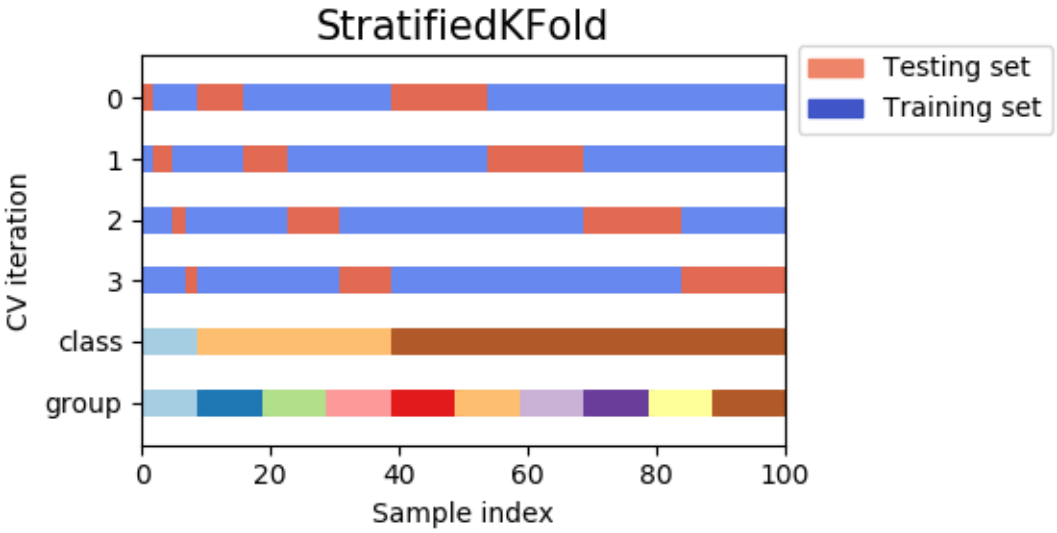
```
    fig, ax = plt.subplots(figsize=(6, 3))
    plot_cv_indices(this_cv, X, y, groups, ax, n_splits)

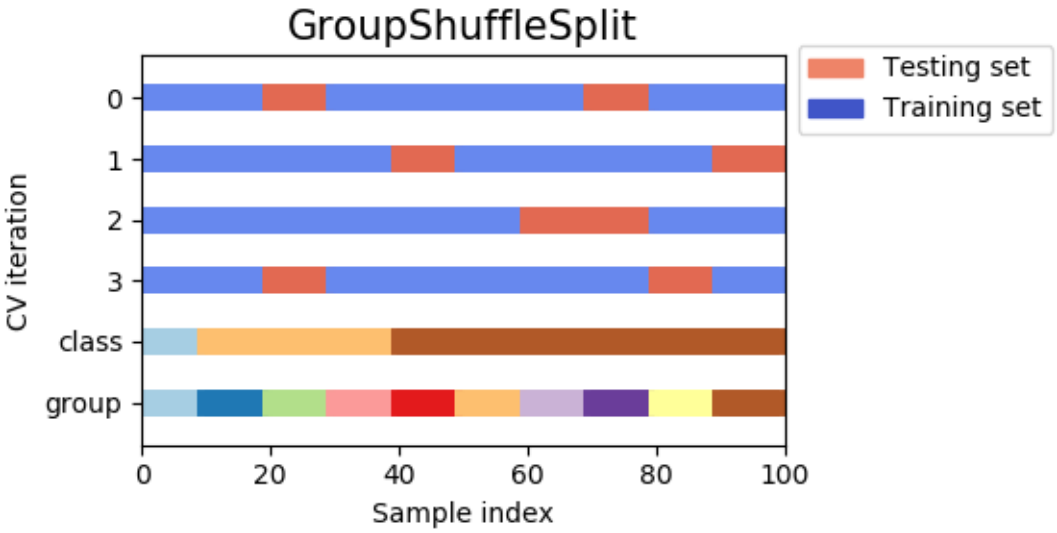
    ax.legend([Patch(color=cmap_cv(.8)), Patch(color=cmap_cv(.02))],
              ['Testing set', 'Training set'], loc=(1.02, .8))
    # Make the legend fit
    plt.tight_layout()
    fig.subplots_adjust(right=.7)
plt.show()
```

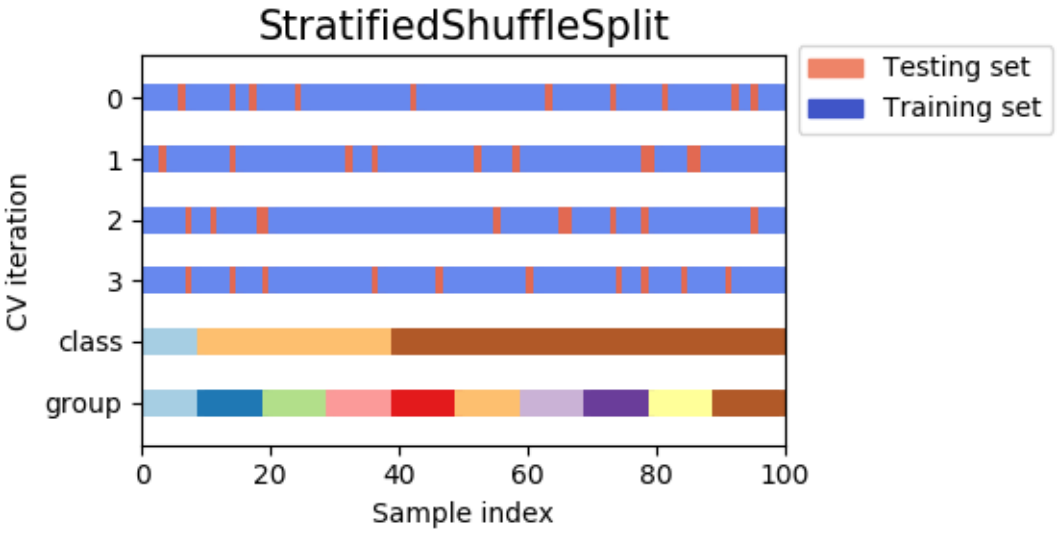
- 

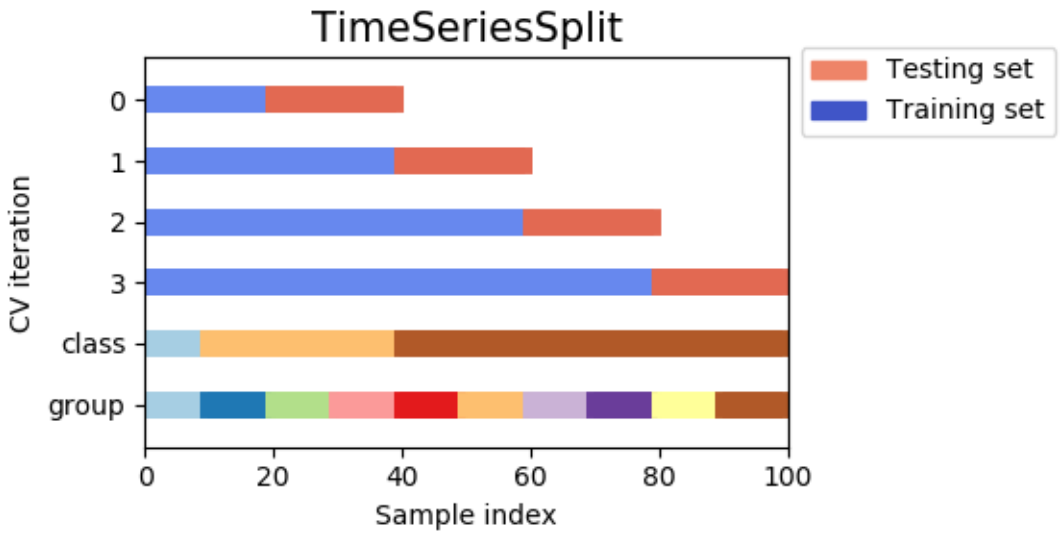
- 

- 
ShuffleSplit
Testing set
Training set
CV iteration
0
1
2
3
class
group
0
20
40
60
80
100
Sample index

- 
StratifiedKFold
Testing set
Training set
CV iteration
0
1
2
3
class
group
0
20
40
60
80
100
Sample index

- 
GroupShuffleSplit
Testing set
Training set
CV iteration
0
1
2
3
class
group
0
20
40
60
80
100
Sample index

- 
StratifiedShuffleSplit
Testing set
Training set
CV iteration
0
1
2
3
class
group
0
20
40
60
80
100
Sample index



-

Total running time of the script: (0 minutes 2.661 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.14 Receiver Operating Characteristic (ROC)

Example of Receiver Operating Characteristic (ROC) metric to evaluate classifier output quality.

ROC curves typically feature true positive rate on the Y axis, and false positive rate on the X axis. This means that the top left corner of the plot is the "ideal" point - a false positive rate of zero, and a true positive rate of one. This is not very realistic, but it does mean that a larger area under the curve (AUC) is usually better.

The "steepness" of ROC curves is also important, since it is ideal to maximize the true positive rate while minimizing the false positive rate.

ROC curves are typically used in binary classification to study the output of a classifier. In order to extend ROC curve and ROC area to multi-label classification, it is necessary to binarize the output. One ROC curve can be drawn per label, but one can also draw a ROC curve by considering each element of the label indicator matrix as a binary prediction (micro-averaging).

Another evaluation measure for multi-label classification is macro-averaging, which gives equal weight to the classification of each label.

Note:

See also ***sklearn.metrics.roc_auc_score***, *Receiver Operating Characteristic (ROC) with cross validation*

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from itertools import cycle
```

(continues on next page)

(continued from previous page)

```
from sklearn import svm, datasets
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import label_binarize
from sklearn.multiclass import OneVsRestClassifier
from scipy import interp
from sklearn.metrics import roc_auc_score

# Import some data to play with
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Binarize the output
y = label_binarize(y, classes=[0, 1, 2])
n_classes = y.shape[1]

# Add noisy features to make the problem harder
random_state = np.random.RandomState(0)
n_samples, n_features = X.shape
X = np.c_[X, random_state.randn(n_samples, 200 * n_features)]

# shuffle and split training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.5,
                                                    random_state=0)

# Learn to predict each class against the other
classifier = OneVsRestClassifier(svm.SVC(kernel='linear', probability=True,
                                 random_state=random_state))
y_score = classifier.fit(X_train, y_train).decision_function(X_test)

# Compute ROC curve and ROC area for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test.ravel(), y_score.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])
```

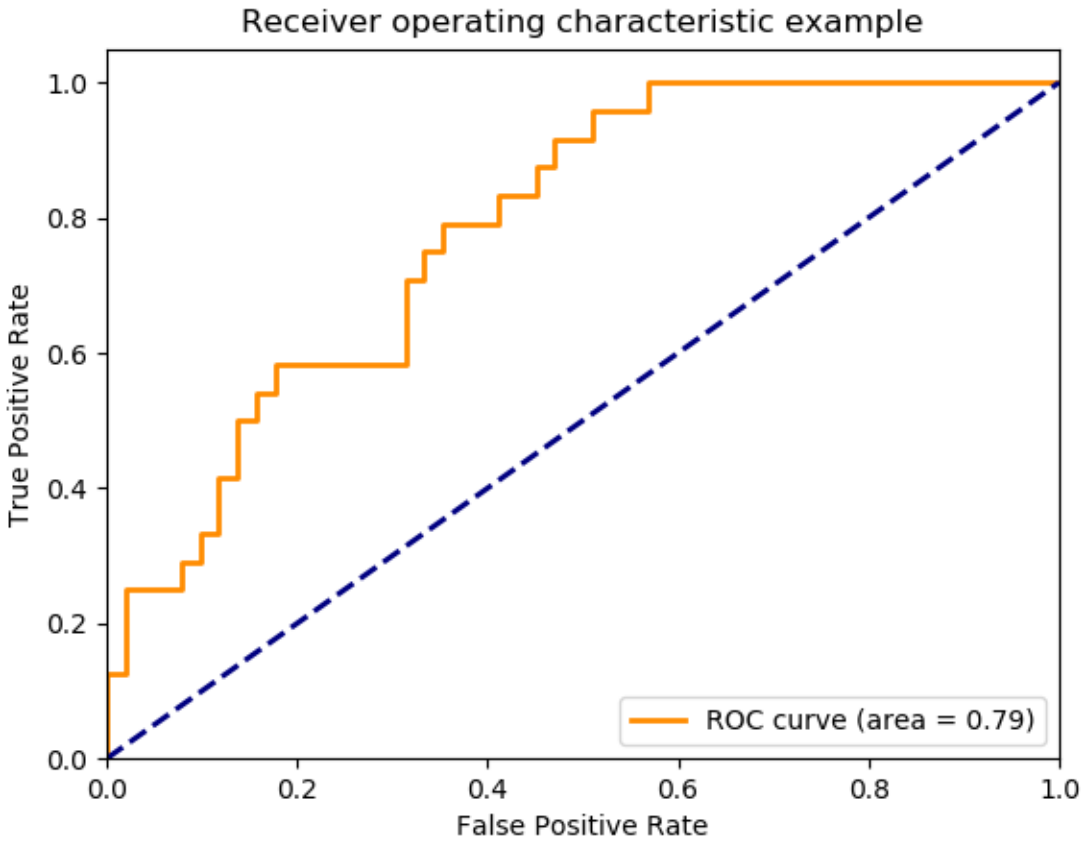
Plot of a ROC curve for a specific class

```
plt.figure()
lw = 2
plt.plot(fpr[2], tpr[2], color='darkorange',
         lw=lw, label='ROC curve (area = %0.2f)' % roc_auc[2])
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



Plot ROC curves for the multilabel problem

Compute macro-average ROC curve and ROC area

```
# First aggregate all false positive rates
all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

# Then interpolate all ROC curves at this points
mean_tpr = np.zeros_like(all_fpr)
for i in range(n_classes):
    mean_tpr += interp(all_fpr, fpr[i], tpr[i])

# Finally average it and compute AUC
mean_tpr /= n_classes

fpr["macro"] = all_fpr
tpr["macro"] = mean_tpr
roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

# Plot all ROC curves
plt.figure()
```

(continues on next page)

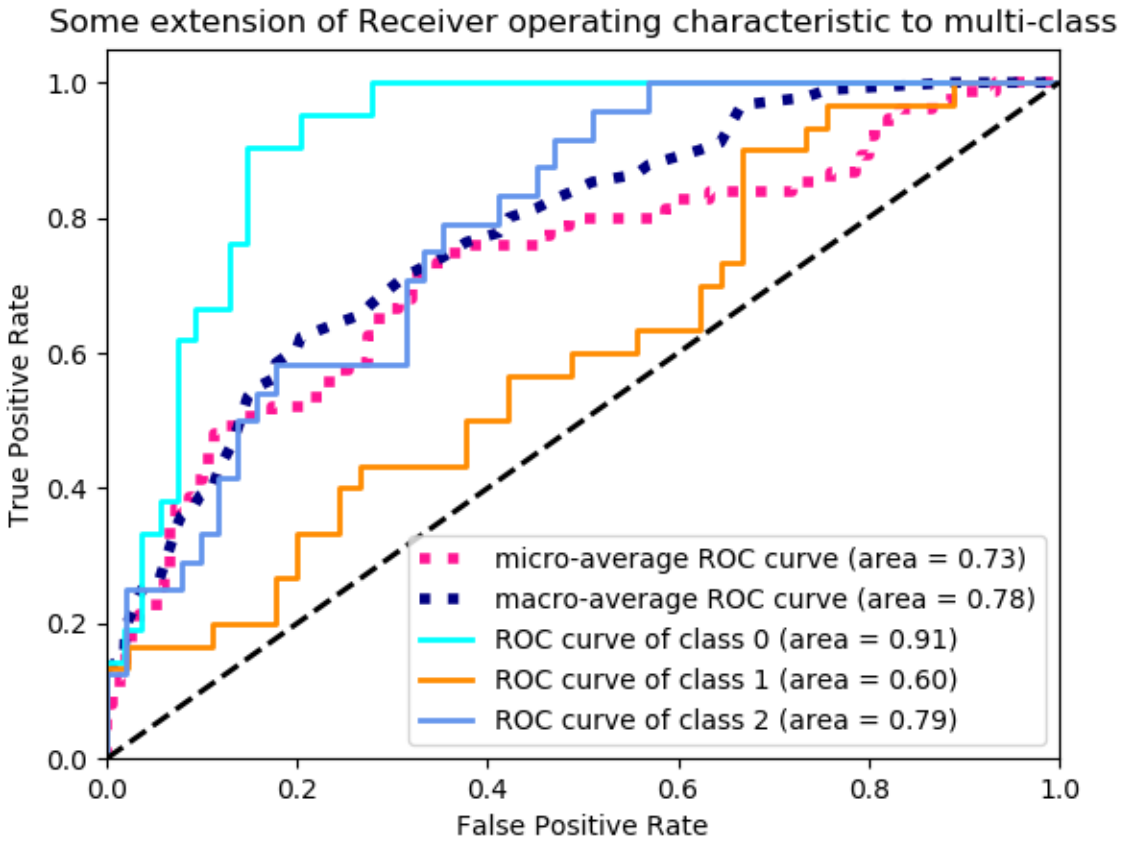
(continued from previous page)

```
plt.plot(fpr["micro"], tpr["micro"],
         label='micro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["micro"]),
         color='deeppink', linestyle=':', linewidth=4)

plt.plot(fpr["macro"], tpr["macro"],
         label='macro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["macro"]),
         color='navy', linestyle=':', linewidth=4)

colors = cycle(['aqua', 'darkorange', 'cornflowerblue'])
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=lw,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Some extension of Receiver operating characteristic to multi-class')
plt.legend(loc="lower right")
plt.show()
```



Out:

```
/home/circleci/project/examples/model_selection/plot_roc.py:112: DeprecationWarning:␣
↪scipy.interp is deprecated and will be removed in SciPy 2.0.0, use numpy.interp␣
↪instead
  mean_tpr += interp(all_fpr, fpr[i], tpr[i])
```

Area under ROC for the multiclass problem

The *sklearn.metrics.roc_auc_score* function can be used for multi-class classification. The multi-class One-vs-One scheme compares every unique pairwise combination of classes. In this section, we calculate the AUC using the OvR and OvO schemes. We report a macro average, and a prevalence-weighted average.

```
y_prob = classifier.predict_proba(X_test)

macro_roc_auc_ovo = roc_auc_score(y_test, y_prob, multi_class="ovo",
                                  average="macro")
weighted_roc_auc_ovo = roc_auc_score(y_test, y_prob, multi_class="ovo",
                                     average="weighted")
macro_roc_auc_ovr = roc_auc_score(y_test, y_prob, multi_class="ovr",
                                  average="macro")
weighted_roc_auc_ovr = roc_auc_score(y_test, y_prob, multi_class="ovr",
                                     average="weighted")
print("One-vs-One ROC AUC scores:\n{:.6f} (macro),\n{:.6f} "
      "(weighted by prevalence)"
      .format(macro_roc_auc_ovo, weighted_roc_auc_ovo))
print("One-vs-Rest ROC AUC scores:\n{:.6f} (macro),\n{:.6f} "
      "(weighted by prevalence)"
      .format(macro_roc_auc_ovr, weighted_roc_auc_ovr))
```

Out:

```
One-vs-One ROC AUC scores:
0.767722 (macro),
0.748802 (weighted by prevalence)
One-vs-Rest ROC AUC scores:
0.767722 (macro),
0.748802 (weighted by prevalence)
```

Total running time of the script: (0 minutes 1.787 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.15 Precision-Recall

Example of Precision-Recall metric to evaluate classifier output quality.

Precision-Recall is a useful measure of success of prediction when the classes are very imbalanced. In information retrieval, precision is a measure of result relevancy, while recall is a measure of how many truly relevant results are returned.

The precision-recall curve shows the tradeoff between precision and recall for different threshold. A high area under the curve represents both high recall and high precision, where high precision relates to a low false positive rate, and

high recall relates to a low false negative rate. High scores for both show that the classifier is returning accurate results (high precision), as well as returning a majority of all positive results (high recall).

A system with high recall but low precision returns many results, but most of its predicted labels are incorrect when compared to the training labels. A system with high precision but low recall is just the opposite, returning very few results, but most of its predicted labels are correct when compared to the training labels. An ideal system with high precision and high recall will return many results, with all results labeled correctly.

Precision (P) is defined as the number of true positives (T_p) over the number of true positives plus the number of false positives (F_p).

$P = \frac{T_p}{T_p+F_p}$

Recall (R) is defined as the number of true positives (T_p) over the number of true positives plus the number of false negatives (F_n).

$R = \frac{T_p}{T_p+F_n}$

These quantities are also related to the (F_1) score, which is defined as the harmonic mean of precision and recall.

$F1 = 2\frac{P \times R}{P+R}$

Note that the precision may not decrease with recall. The definition of precision ($\frac{T_p}{T_p+F_p}$) shows that lowering the threshold of a classifier may increase the denominator, by increasing the number of results returned. If the threshold was previously set too high, the new results may all be true positives, which will increase precision. If the previous threshold was about right or too low, further lowering the threshold will introduce false positives, decreasing precision.

Recall is defined as $\frac{T_p}{T_p+F_n}$, where $T_p + F_n$ does not depend on the classifier threshold. This means that lowering the classifier threshold may increase recall, by increasing the number of true positive results. It is also possible that lowering the threshold may leave recall unchanged, while the precision fluctuates.

The relationship between recall and precision can be observed in the stairstep area of the plot - at the edges of these steps a small change in the threshold considerably reduces precision, with only a minor gain in recall.

Average precision (AP) summarizes such a plot as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight:

$\text{AP} = \sum_n (R_n - R_{n-1}) P_n$

where P_n and R_n are the precision and recall at the nth threshold. A pair (R_k, P_k) is referred to as an *operating point*.

AP and the trapezoidal area under the operating points (`sklearn.metrics.auc`) are common ways to summarize a precision-recall curve that lead to different results. Read more in the *User Guide*.

Precision-recall curves are typically used in binary classification to study the output of a classifier. In order to extend the precision-recall curve and average precision to multi-class or multi-label classification, it is necessary to binarize the output. One curve can be drawn per label, but one can also draw a precision-recall curve by considering each element of the label indicator matrix as a binary prediction (micro-averaging).

Note:

See also **`sklearn.metrics.average_precision_score`**, `sklearn.metrics.recall_score`, `sklearn.metrics.precision_score`, `sklearn.metrics.f1_score`

In binary classification settings

Create simple data

Try to differentiate the two first classes of the iris data

```
from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
import numpy as np

iris = datasets.load_iris()
X = iris.data
y = iris.target

# Add noisy features
random_state = np.random.RandomState(0)
n_samples, n_features = X.shape
X = np.c_[X, random_state.randn(n_samples, 200 * n_features)]

# Limit to the two first classes, and split into training and test
X_train, X_test, y_train, y_test = train_test_split(X[y < 2], y[y < 2],
                                                    test_size=.5,
                                                    random_state=random_state)

# Create a simple classifier
classifier = svm.LinearSVC(random_state=random_state)
classifier.fit(X_train, y_train)
y_score = classifier.decision_function(X_test)
```

Compute the average precision score

```
from sklearn.metrics import average_precision_score
average_precision = average_precision_score(y_test, y_score)

print('Average precision-recall score: {0:0.2f}'.format(
      average_precision))
```

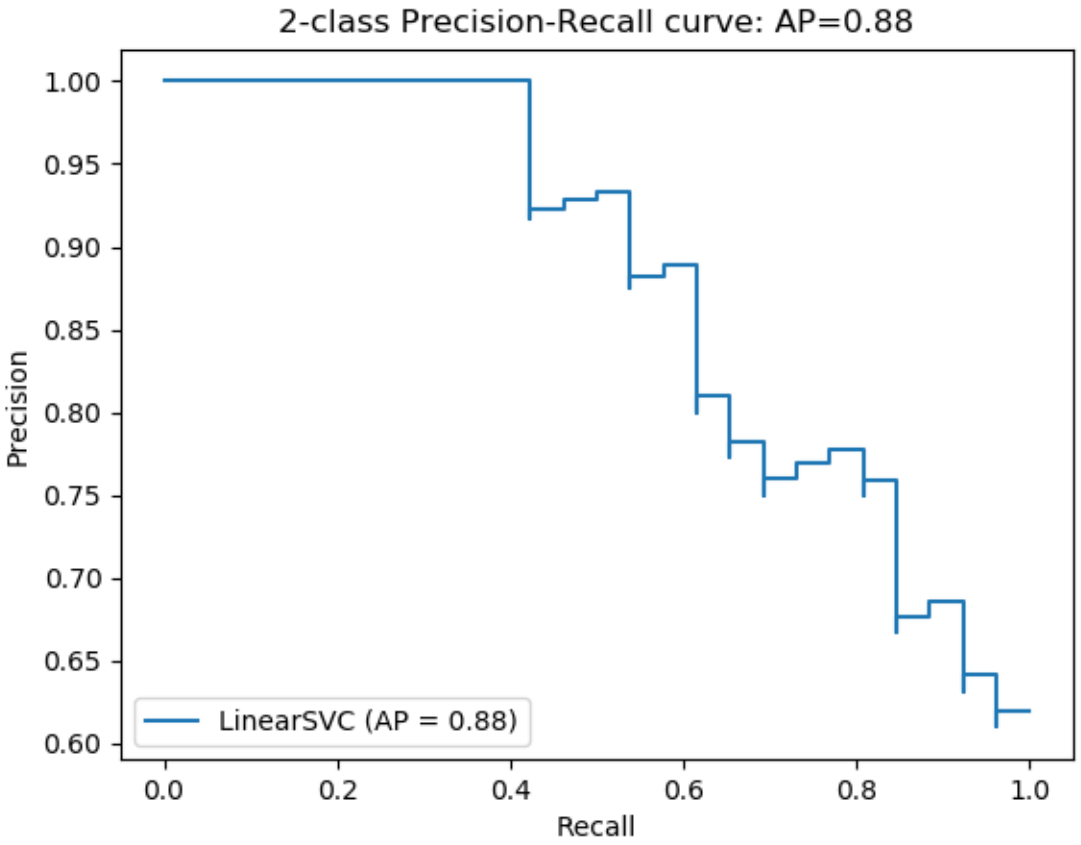
Out:

```
Average precision-recall score: 0.88
```

Plot the Precision-Recall curve

```
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import plot_precision_recall_curve
import matplotlib.pyplot as plt

disp = plot_precision_recall_curve(classifier, X_test, y_test)
disp.ax_.set_title('2-class Precision-Recall curve: '
                   'AP={0:0.2f}'.format(average_precision))
```



Out:

```
Text(0.5, 1.0, '2-class Precision-Recall curve: AP=0.88')
```

In multi-label settings

Create multi-label data, fit, and predict

We create a multi-label dataset, to illustrate the precision-recall in multi-label settings

```
from sklearn.preprocessing import label_binarize

# Use label_binarize to be multi-label like settings
Y = label_binarize(y, classes=[0, 1, 2])
n_classes = Y.shape[1]

# Split into training and test
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.5,
                                                    random_state=random_state)

# We use OneVsRestClassifier for multi-label prediction
from sklearn.multiclass import OneVsRestClassifier

# Run classifier
```

(continues on next page)

(continued from previous page)

```
classifier = OneVsRestClassifier(svm.LinearSVC(random_state=random_state))
classifier.fit(X_train, Y_train)
y_score = classifier.decision_function(X_test)
```

The average precision score in multi-label settings

```
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import average_precision_score

# For each class
precision = dict()
recall = dict()
average_precision = dict()
for i in range(n_classes):
    precision[i], recall[i], _ = precision_recall_curve(Y_test[:, i],
                                                        y_score[:, i])
    average_precision[i] = average_precision_score(Y_test[:, i], y_score[:, i])

# A "micro-average": quantifying score on all classes jointly
precision["micro"], recall["micro"], _ = precision_recall_curve(Y_test.ravel(),
    y_score.ravel())
average_precision["micro"] = average_precision_score(Y_test, y_score,
                                                     average="micro")
print('Average precision score, micro-averaged over all classes: {0:0.2f}'
      .format(average_precision["micro"]))
```

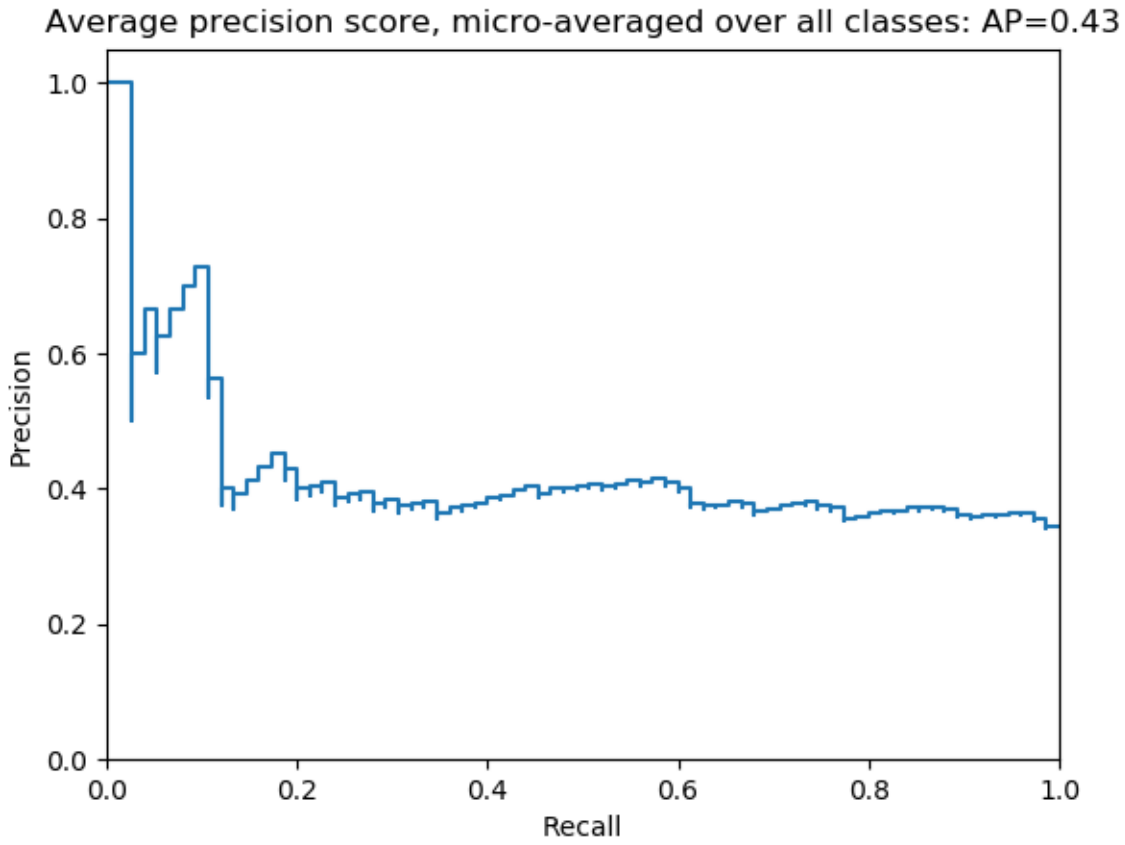
Out:

```
Average precision score, micro-averaged over all classes: 0.43
```

Plot the micro-averaged Precision-Recall curve

```
plt.figure()
plt.step(recall['micro'], precision['micro'], where='post')

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title(
    'Average precision score, micro-averaged over all classes: AP={0:0.2f}'
    .format(average_precision["micro"]))
```



Out:

```
Text(0.5, 1.0, 'Average precision score, micro-averaged over all classes: AP=0.43')
```

Plot Precision-Recall curve for each class and iso-f1 curves

```
from itertools import cycle
# setup plot details
colors = cycle(['navy', 'turquoise', 'darkorange', 'cornflowerblue', 'teal'])

plt.figure(figsize=(7, 8))
f_scores = np.linspace(0.2, 0.8, num=4)
lines = []
labels = []
for f_score in f_scores:
    x = np.linspace(0.01, 1)
    y = f_score * x / (2 * x - f_score)
    l, = plt.plot(x[y >= 0], y[y >= 0], color='gray', alpha=0.2)
    plt.annotate('f1={0:0.1f}'.format(f_score), xy=(0.9, y[45] + 0.02))

lines.append(l)
labels.append('iso-f1 curves')
l, = plt.plot(recall["micro"], precision["micro"], color='gold', lw=2)
lines.append(l)
```

(continues on next page)

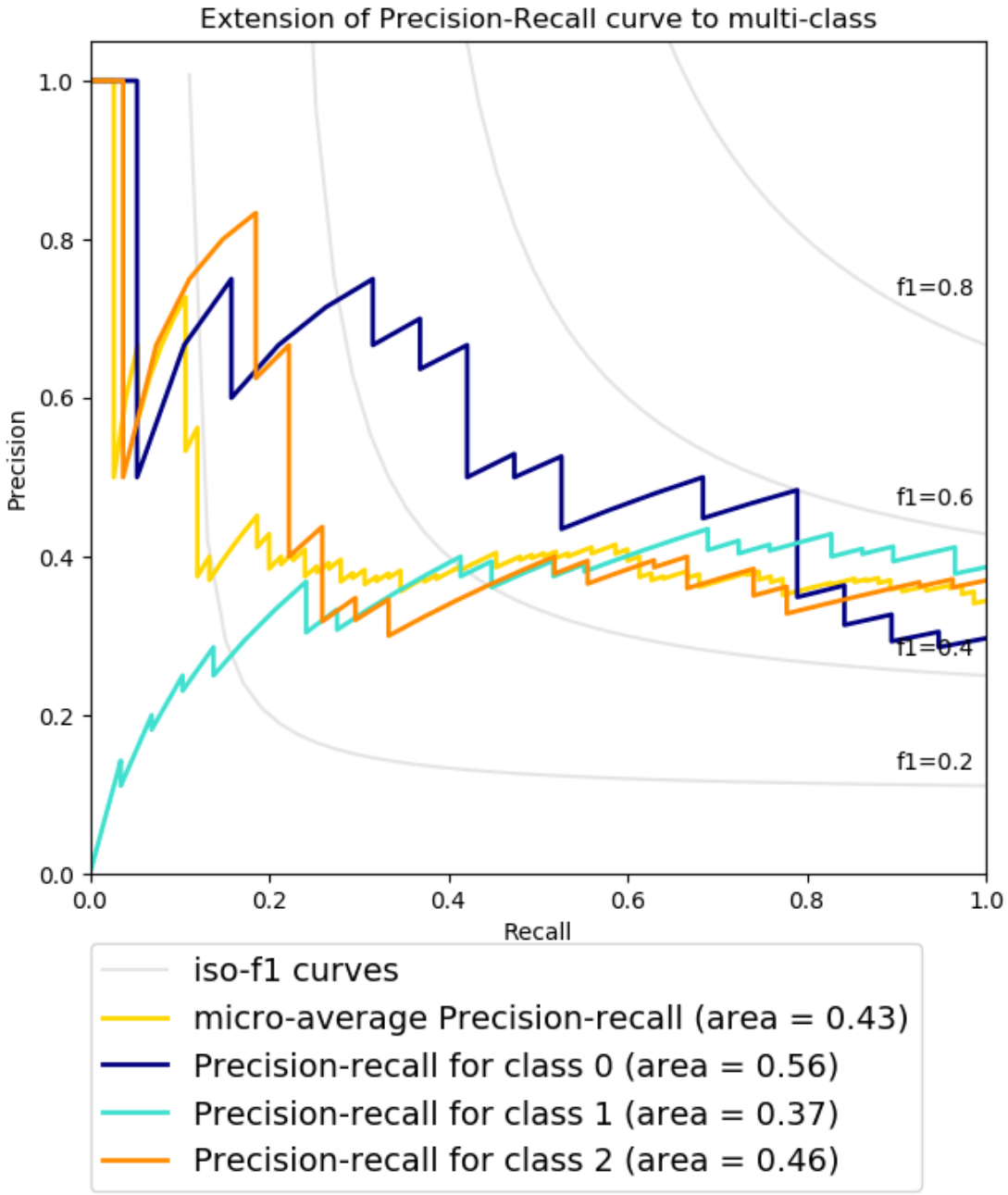
(continued from previous page)

```
labels.append('micro-average Precision-recall (area = {0:0.2f})'
              ''.format(average_precision["micro"]))

for i, color in zip(range(n_classes), colors):
    l, = plt.plot(recall[i], precision[i], color=color, lw=2)
    lines.append(l)
    labels.append('Precision-recall for class {0} (area = {1:0.2f})'
                  ''.format(i, average_precision[i]))

fig = plt.gcf()
fig.subplots_adjust(bottom=0.25)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Extension of Precision-Recall curve to multi-class')
plt.legend(lines, labels, loc=(0, -.38), prop=dict(size=14))


plt.show()
```



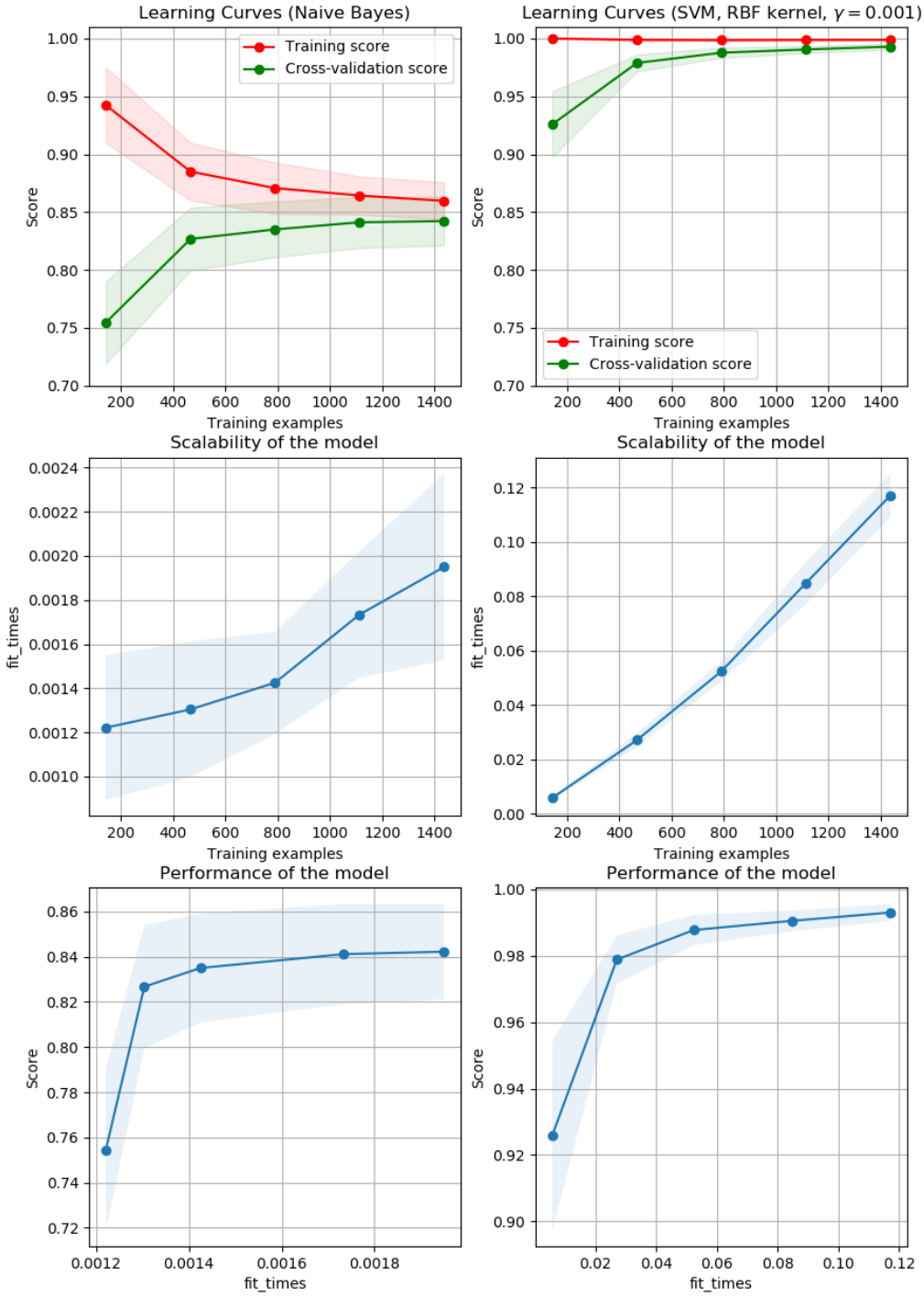
Total running time of the script: (0 minutes 2.755 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.20.16 Plotting Learning Curves

In the first column, first row the learning curve of a naive Bayes classifier is shown for the digits dataset. Note that the training score and the cross-validation score are both not very good at the end. However, the shape of the curve can be found in more complex datasets very often: the training score is very high at the beginning and decreases and the cross-validation score is very low at the beginning and increases. In the second column, first row we see the learning curve of an SVM with RBF kernel. We can see clearly that the training score is still around the maximum and the validation score could be increased with more training samples. The plots in the second row show the times required by the models to train with various sizes of training dataset. The plots in the third row show how much time was required to train the models for each training sizes.


Learning Curves (Naive Bayes)
Training score
Cross-validation score
Score
Training examples
Learning Curves (SVM, RBF kernel, $\gamma = 0.001$)
Training score
Cross-validation score
Score
Training examples
Scalability of the model
fit_times
Training examples
Scalability of the model
fit_times
Training examples
Performance of the model
Score
fit_times
Performance of the model
Score
fit_times

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.datasets import load_digits
from sklearn.model_selection import learning_curve
from sklearn.model_selection import ShuffleSplit


def plot_learning_curve(estimator, title, X, y, axes=None, ylim=None, cv=None,
                        n_jobs=None, train_sizes=np.linspace(.1, 1.0, 5)):
    """
    Generate 3 plots: the test and training learning curve, the training
    samples vs fit times curve, the fit times vs score curve.

    Parameters
    ----------
    estimator : object type that implements the "fit" and "predict" methods
        An object of that type which is cloned for each validation.

    title : string
        Title for the chart.

    X : array-like, shape (n_samples, n_features)
        Training vector, where n_samples is the number of samples and
        n_features is the number of features.

    y : array-like, shape (n_samples) or (n_samples, n_features), optional
        Target relative to X for classification or regression;
        None for unsupervised learning.

    axes : array of 3 axes, optional (default=None)
        Axes to use for plotting the curves.

    ylim : tuple, shape (ymin, ymax), optional
        Defines minimum and maximum yvalues plotted.

    cv : int, cross-validation generator or an iterable, optional
        Determines the cross-validation splitting strategy.
        Possible inputs for cv are:
          - None, to use the default 5-fold cross-validation,
          - integer, to specify the number of folds.
          - :term:`CV splitter`,
          - An iterable yielding (train, test) splits as arrays of indices.

        For integer/None inputs, if ``y`` is binary or multiclass,
        :class:`StratifiedKFold` used. If the estimator is not a classifier
        or if ``y`` is neither binary nor multiclass, :class:`KFold` is used.

        Refer :ref:`User Guide <cross_validation>` for the various
        cross-validators that can be used here.

    n_jobs : int or None, optional (default=None)
        Number of jobs to run in parallel.
        ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
```

(continues on next page)

(continued from previous page)

```
        ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
        for more details.

    train_sizes : array-like, shape (n_ticks,), dtype float or int
        Relative or absolute numbers of training examples that will be used to
        generate the learning curve. If the dtype is float, it is regarded as a
        fraction of the maximum size of the training set (that is determined
        by the selected validation method), i.e. it has to be within (0, 1].
        Otherwise it is interpreted as absolute sizes of the training sets.
        Note that for classification the number of samples usually have to
        be big enough to contain at least one sample from each class.
        (default: np.linspace(0.1, 1.0, 5))
    """
    if axes is None:
        _, axes = plt.subplots(1, 3, figsize=(20, 5))

    axes[0].set_title(title)
    if ylim is not None:
        axes[0].set_ylim(*ylim)
    axes[0].set_xlabel("Training examples")
    axes[0].set_ylabel("Score")

    train_sizes, train_scores, test_scores, fit_times, _ = \
        learning_curve(estimator, X, y, cv=cv, n_jobs=n_jobs,
                       train_sizes=train_sizes,
                       return_times=True)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    fit_times_mean = np.mean(fit_times, axis=1)
    fit_times_std = np.std(fit_times, axis=1)

    # Plot learning curve
    axes[0].grid()
    axes[0].fill_between(train_sizes, train_scores_mean - train_scores_std,
                         train_scores_mean + train_scores_std, alpha=0.1,
                         color="r")
    axes[0].fill_between(train_sizes, test_scores_mean - test_scores_std,
                         test_scores_mean + test_scores_std, alpha=0.1,
                         color="g")
    axes[0].plot(train_sizes, train_scores_mean, 'o-', color="r",
                 label="Training score")
    axes[0].plot(train_sizes, test_scores_mean, 'o-', color="g",
                 label="Cross-validation score")
    axes[0].legend(loc="best")

    # Plot n_samples vs fit_times
    axes[1].grid()
    axes[1].plot(train_sizes, fit_times_mean, 'o-')
    axes[1].fill_between(train_sizes, fit_times_mean - fit_times_std,
                         fit_times_mean + fit_times_std, alpha=0.1)
    axes[1].set_xlabel("Training examples")
    axes[1].set_ylabel("fit_times")
    axes[1].set_title("Scalability of the model")

    # Plot fit_time vs score
```

(continues on next page)

(continued from previous page)

```
    axes[2].grid()
    axes[2].plot(fit_times_mean, test_scores_mean, 'o-')
    axes[2].fill_between(fit_times_mean, test_scores_mean - test_scores_std,
                         test_scores_mean + test_scores_std, alpha=0.1)
    axes[2].set_xlabel("fit_times")
    axes[2].set_ylabel("Score")
    axes[2].set_title("Performance of the model")

    return plt


fig, axes = plt.subplots(3, 2, figsize=(10, 15))

X, y = load_digits(return_X_y=True)

title = "Learning Curves (Naive Bayes)"
# Cross validation with 100 iterations to get smoother mean test and train
# score curves, each time with 20% data randomly selected as a validation set.
cv = ShuffleSplit(n_splits=100, test_size=0.2, random_state=0)

estimator = GaussianNB()
plot_learning_curve(estimator, title, X, y, axes=axes[:, 0], ylim=(0.7, 1.01),
                    cv=cv, n_jobs=4)

title = r"Learning Curves (SVM, RBF kernel, $\gamma=0.001$)"
# SVC is more expensive so we do a lower number of CV iterations:
cv = ShuffleSplit(n_splits=10, test_size=0.2, random_state=0)
estimator = SVC(gamma=0.001)
plot_learning_curve(estimator, title, X, y, axes=axes[:, 1], ylim=(0.7, 1.01),
                    cv=cv, n_jobs=4)

plt.show()
```

Total running time of the script: (0 minutes 3.994 seconds)

Estimated memory usage: 172 MB

6.21 Multioutput methods

Examples concerning the *sklearn.multioutput* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

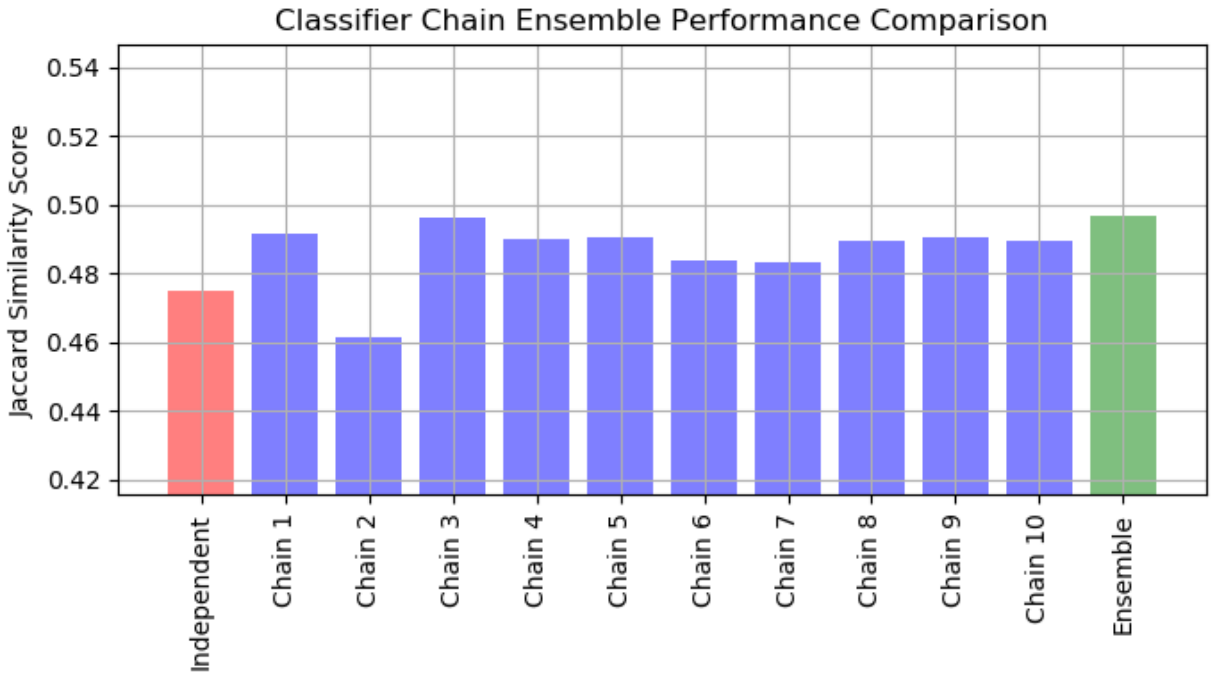
6.21.1 Classifier Chain

Example of using classifier chain on a multilabel dataset.

For this example we will use the yeast dataset which contains 2417 datapoints each with 103 features and 14 possible labels. Each data point has at least one label. As a baseline we first train a logistic regression classifier for each of the 14 labels. To evaluate the performance of these classifiers we predict on a held-out test set and calculate the *jaccard score* for each sample.

Next we create 10 classifier chains. Each classifier chain contains a logistic regression model for each of the 14 labels. The models in each chain are ordered randomly. In addition to the 103 features in the dataset, each model gets the predictions of the preceding models in the chain as features (note that by default at training time each model gets the true labels as features). These additional features allow each chain to exploit correlations among the classes. The Jaccard similarity score for each chain tends to be greater than that of the set independent logistic models.

Because the models in each chain are arranged randomly there is significant variation in performance among the chains. Presumably there is an optimal ordering of the classes in a chain that will yield the best performance. However we do not know that ordering a priori. Instead we can construct an voting ensemble of classifier chains by averaging the binary predictions of the chains and apply a threshold of 0.5. The Jaccard similarity score of the ensemble is greater than that of the independent models and tends to exceed the score of each chain in the ensemble (although this is not guaranteed with randomly ordered chains).




```
# Author: Adam Kleczewski
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.multioutput import ClassifierChain
from sklearn.model_selection import train_test_split
from sklearn.multiclass import OneVsRestClassifier
from sklearn.metrics import jaccard_score
from sklearn.linear_model import LogisticRegression

print(__doc__)

# Load a multi-label dataset from https://www.openml.org/d/40597
X, Y = fetch_openml('yeast', version=4, return_X_y=True)
Y = Y == 'TRUE'
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.2,
                                                    random_state=0)
```

(continues on next page)

(continued from previous page)

```
# Fit an independent logistic regression model for each class using the
# OneVsRestClassifier wrapper.
base_lr = LogisticRegression()
ovr = OneVsRestClassifier(base_lr)
ovr.fit(X_train, Y_train)
Y_pred_ovr = ovr.predict(X_test)
ovr_jaccard_score = jaccard_score(Y_test, Y_pred_ovr, average='samples')

# Fit an ensemble of logistic regression classifier chains and take the
# take the average prediction of all the chains.
chains = [ClassifierChain(base_lr, order='random', random_state=i)
          for i in range(10)]
for chain in chains:
    chain.fit(X_train, Y_train)

Y_pred_chains = np.array([chain.predict(X_test) for chain in
                          chains])
chain_jaccard_scores = [jaccard_score(Y_test, Y_pred_chain >= .5,
                                      average='samples')
                        for Y_pred_chain in Y_pred_chains]

Y_pred_ensemble = Y_pred_chains.mean(axis=0)
ensemble_jaccard_score = jaccard_score(Y_test,
                                       Y_pred_ensemble >= .5,
                                       average='samples')

model_scores = [ovr_jaccard_score] + chain_jaccard_scores
model_scores.append(ensemble_jaccard_score)

model_names = ('Independent',
               'Chain 1',
               'Chain 2',
               'Chain 3',
               'Chain 4',
               'Chain 5',
               'Chain 6',
               'Chain 7',
               'Chain 8',
               'Chain 9',
               'Chain 10',
               'Ensemble')

x_pos = np.arange(len(model_names))

# Plot the Jaccard similarity scores for the independent model, each of the
# chains, and the ensemble (note that the vertical axis on this plot does
# not begin at 0).

fig, ax = plt.subplots(figsize=(7, 4))
ax.grid(True)
ax.set_title('Classifier Chain Ensemble Performance Comparison')
ax.set_xticks(x_pos)
ax.set_xticklabels(model_names, rotation='vertical')
ax.set_ylabel('Jaccard Similarity Score')
ax.set_ylim([min(model_scores) * .9, max(model_scores) * 1.1])
colors = ['r'] + ['b'] * len(chain_jaccard_scores) + ['g']
```

(continues on next page)

(continued from previous page)

```
ax.bar(x_pos, model_scores, alpha=0.5, color=colors)
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 3.661 seconds)

Estimated memory usage: 8 MB

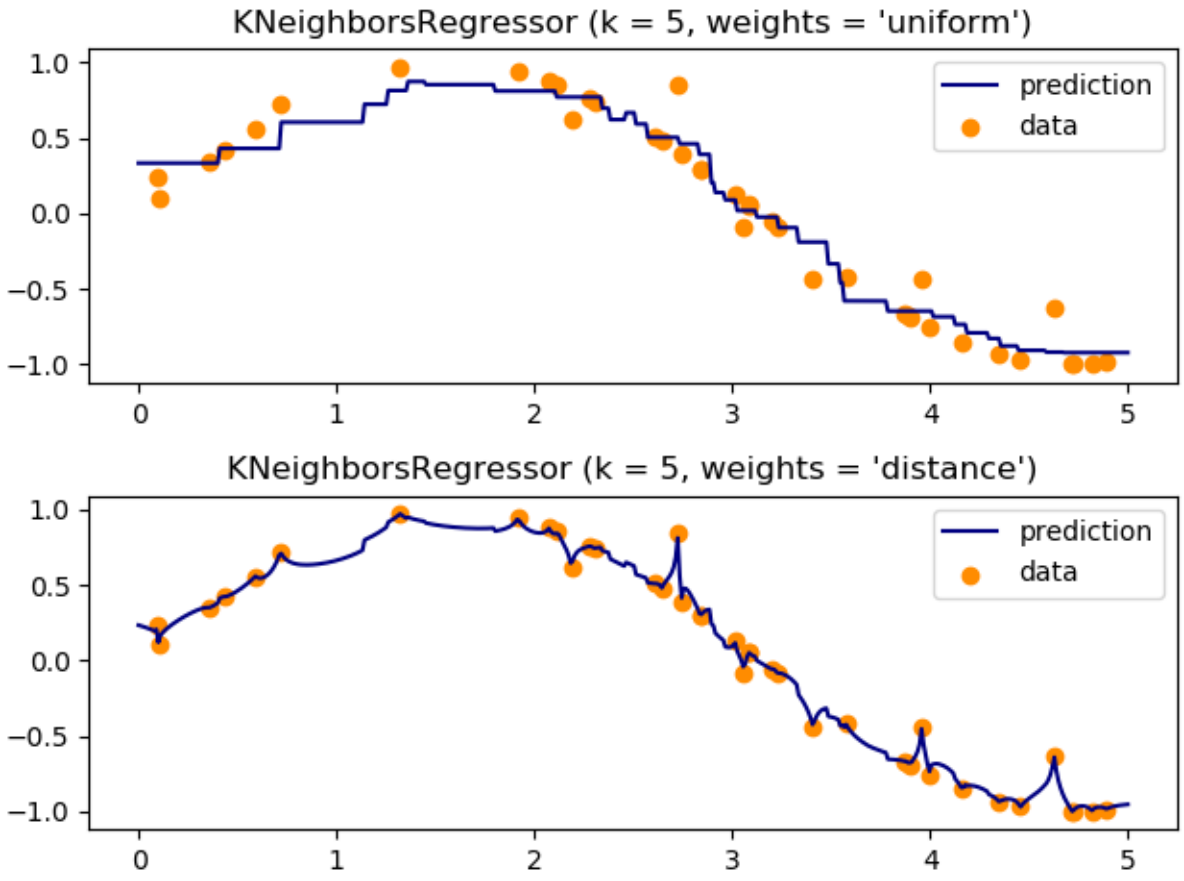
6.22 Nearest Neighbors

Examples concerning the *sklearn.neighbors* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.22.1 Nearest Neighbors regression

Demonstrate the resolution of a regression problem using a k-Nearest Neighbor and the interpolation of the target using both barycenter and constant weights.



```
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#         Fabian Pedregosa <fabian.pedregosa@inria.fr>
#
# License: BSD 3 clause (C) INRIA


# #############################################################################
# Generate sample data
import numpy as np
import matplotlib.pyplot as plt
from sklearn import neighbors

np.random.seed(0)
X = np.sort(5 * np.random.rand(40, 1), axis=0)
T = np.linspace(0, 5, 500)[:, np.newaxis]
y = np.sin(X).ravel()

# Add noise to targets
y[::5] += 1 * (0.5 - np.random.rand(8))

# #############################################################################
# Fit regression model
n_neighbors = 5

for i, weights in enumerate(['uniform', 'distance']):
    knn = neighbors.KNeighborsRegressor(n_neighbors, weights=weights)
    y_ = knn.fit(X, y).predict(T)

    plt.subplot(2, 1, i + 1)
    plt.scatter(X, y, color='darkorange', label='data')
    plt.plot(T, y_, color='navy', label='prediction')
    plt.axis('tight')
    plt.legend()
    plt.title("KNeighborsRegressor (k = %i, weights = '%s')" % (n_neighbors,
                                                                weights))

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.610 seconds)

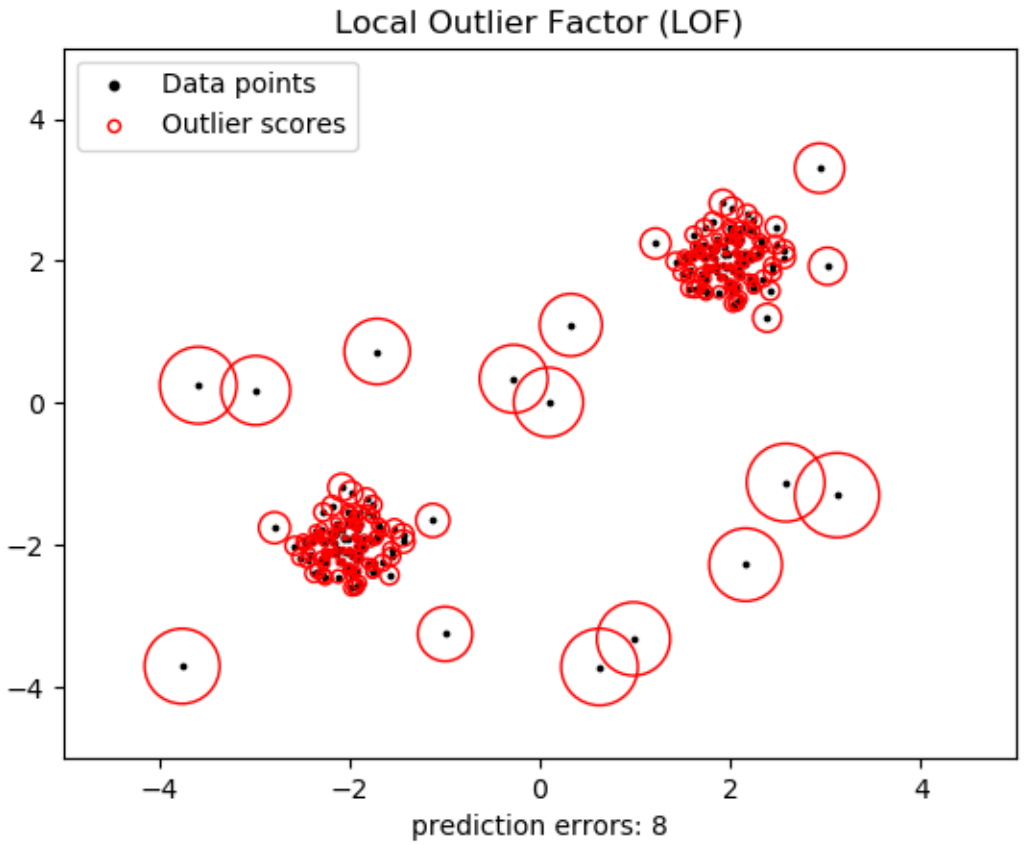
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.22.2 Outlier detection with Local Outlier Factor (LOF)

The Local Outlier Factor (LOF) algorithm is an unsupervised anomaly detection method which computes the local density deviation of a given data point with respect to its neighbors. It considers as outliers the samples that have a substantially lower density than their neighbors. This example shows how to use LOF for outlier detection which is the default use case of this estimator in scikit-learn. Note that when LOF is used for outlier detection it has no predict, decision_function and score_samples methods. See *User Guide*: for details on the difference between outlier detection and novelty detection and how to use LOF for novelty detection.

The number of neighbors considered (parameter n_neighbors) is typically set 1) greater than the minimum number of samples a cluster has to contain, so that other samples can be local outliers relative to this cluster, and 2) smaller than the maximum number of close by samples that can potentially be local outliers. In practice, such informations are generally not available, and taking n_neighbors=20 appears to work well in general.




```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import LocalOutlierFactor

print(__doc__)

np.random.seed(42)

# Generate train data
X_inliers = 0.3 * np.random.randn(100, 2)
X_inliers = np.r_[X_inliers + 2, X_inliers - 2]

# Generate some outliers
X_outliers = np.random.uniform(low=-4, high=4, size=(20, 2))
X = np.r_[X_inliers, X_outliers]

n_outliers = len(X_outliers)
ground_truth = np.ones(len(X), dtype=int)
ground_truth[-n_outliers:] = -1
```

(continues on next page)

(continued from previous page)

```
# fit the model for outlier detection (default)
clf = LocalOutlierFactor(n_neighbors=20, contamination=0.1)
# use fit_predict to compute the predicted labels of the training samples
# (when LOF is used for outlier detection, the estimator has no predict,
# decision_function and score_samples methods).
y_pred = clf.fit_predict(X)
n_errors = (y_pred != ground_truth).sum()
X_scores = clf.negative_outlier_factor_

plt.title("Local Outlier Factor (LOF)")
plt.scatter(X[:, 0], X[:, 1], color='k', s=3., label='Data points')
# plot circles with radius proportional to the outlier scores
radius = (X_scores.max() - X_scores) / (X_scores.max() - X_scores.min())
plt.scatter(X[:, 0], X[:, 1], s=1000 * radius, edgecolors='r',
            facecolors='none', label='Outlier scores')
plt.axis('tight')
plt.xlim((-5, 5))
plt.ylim((-5, 5))
plt.xlabel("prediction errors: %d" % (n_errors))
legend = plt.legend(loc='upper left')
legend.legendHandles[0]._sizes = [10]
legend.legendHandles[1]._sizes = [20]
plt.show()
```

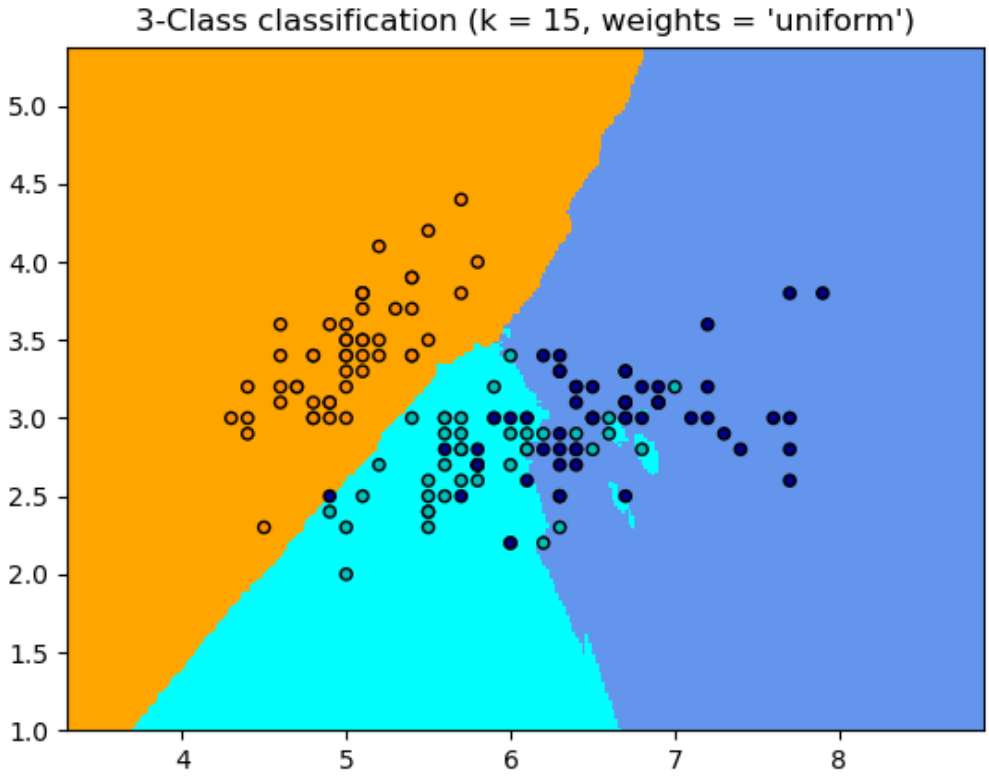
Total running time of the script: (0 minutes 0.575 seconds)

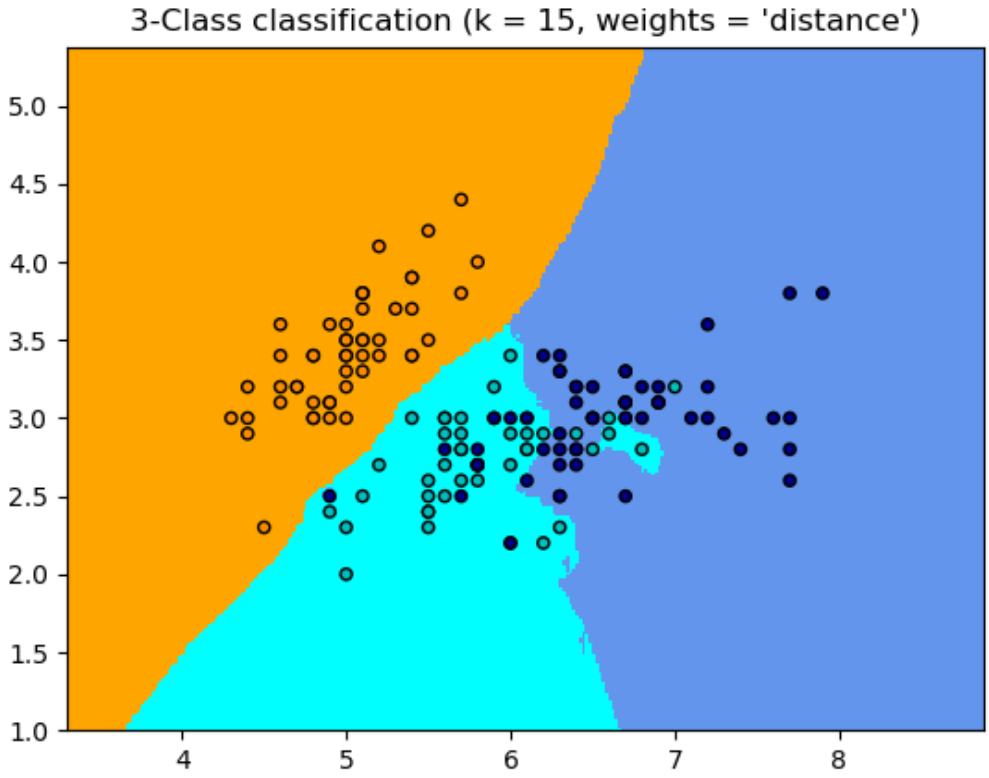
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.22.3 Nearest Neighbors Classification

Sample usage of Nearest Neighbors classification. It will plot the decision boundaries for each class.


3-Class classification (k = 15, weights = 'uniform')
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
8



-

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets

n_neighbors = 15

# import some data to play with
iris = datasets.load_iris()

# we only take the first two features. We could avoid this ugly
# slicing by using a two-dim dataset
X = iris.data[:, :2]
y = iris.target

h = .02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['orange', 'cyan', 'cornflowerblue'])
cmap_bold = ListedColormap(['darkorange', 'c', 'darkblue'])

for weights in ['uniform', 'distance']:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(n_neighbors, weights=weights)
    clf.fit(X, y)
```

(continues on next page)

(continued from previous page)

```
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold,
                edgecolor='k', s=20)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("3-Class classification (k = %i, weights = '%s')"
              % (n_neighbors, weights))

plt.show()
```

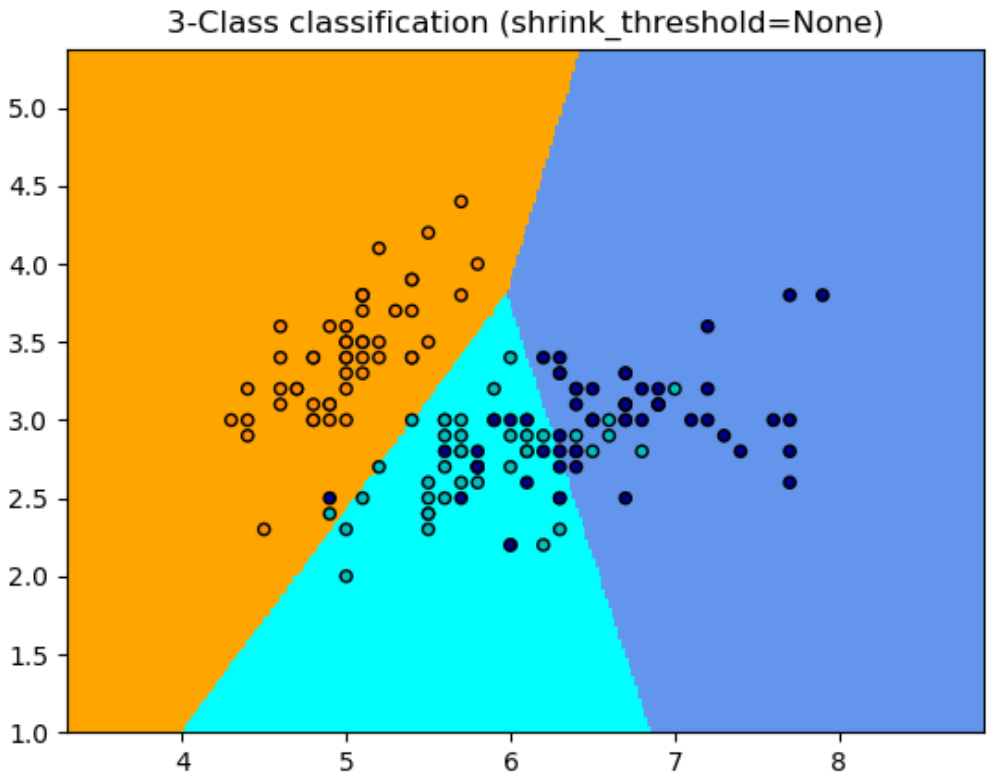
Total running time of the script: (0 minutes 2.022 seconds)

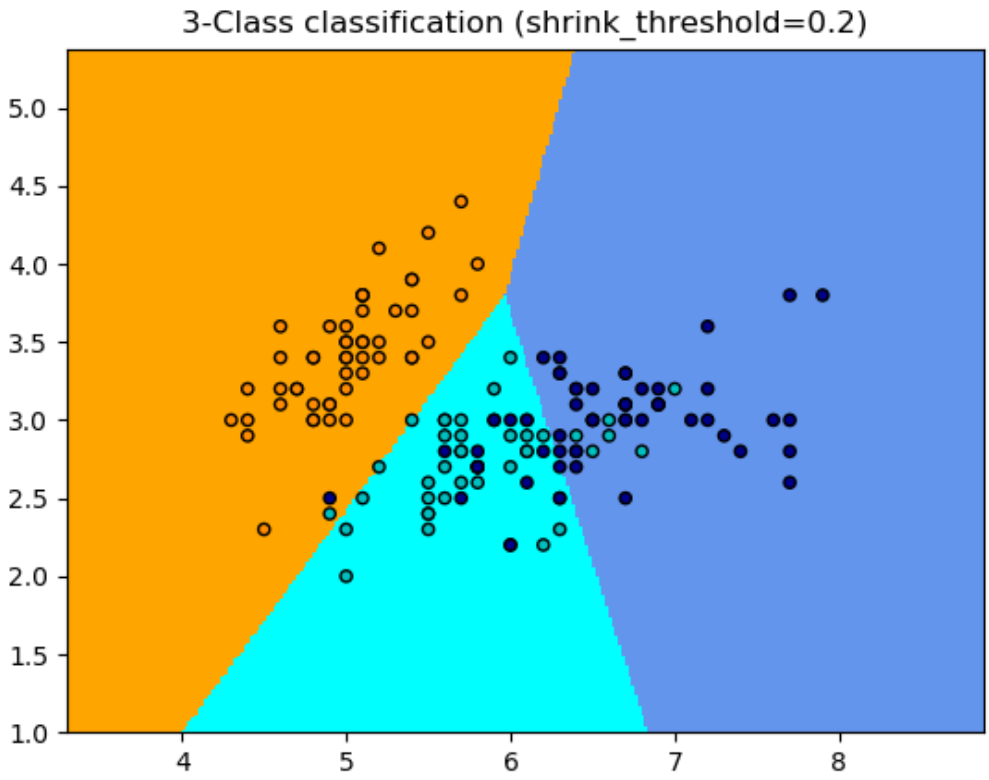
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.22.4 Nearest Centroid Classification

Sample usage of Nearest Centroid classification. It will plot the decision boundaries for each class.


3-Class classification (shrink_threshold=None)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
8



-

Out:

```
None 0.8133333333333334
0.2 0.82
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from sklearn.neighbors import NearestCentroid

n_neighbors = 15

# import some data to play with
iris = datasets.load_iris()
# we only take the first two features. We could avoid this ugly
# slicing by using a two-dim dataset
X = iris.data[:, :2]
y = iris.target
```

(continues on next page)

(continued from previous page)

```
h = .02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['orange', 'cyan', 'cornflowerblue'])
cmap_bold = ListedColormap(['darkorange', 'c', 'darkblue'])

for shrinkage in [None, .2]:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = NearestCentroid(shrink_threshold=shrinkage)
    clf.fit(X, y)
    y_pred = clf.predict(X)
    print(shrinkage, np.mean(y == y_pred))
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold,
                edgecolor='k', s=20)
    plt.title("3-Class classification (shrink_threshold=%r)"
              % shrinkage)
    plt.axis('tight')

plt.show()
```

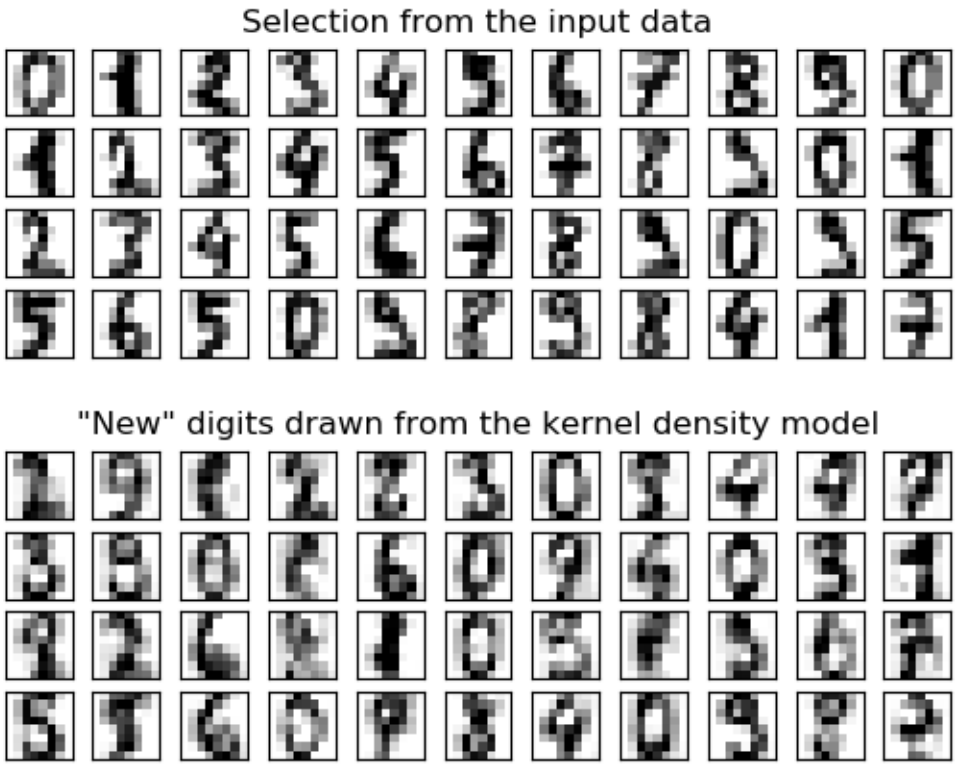
Total running time of the script: (0 minutes 0.690 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.22.5 Kernel Density Estimation

This example shows how kernel density estimation (KDE), a powerful non-parametric density estimation technique, can be used to learn a generative model for a dataset. With this generative model in place, new samples can be drawn. These new samples reflect the underlying model of the data.



Out:

```
best bandwidth: 3.79269019073225
```

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_digits
from sklearn.neighbors import KernelDensity
from sklearn.decomposition import PCA
from sklearn.model_selection import GridSearchCV

# load the data
digits = load_digits()

# project the 64-dimensional data to a lower dimension
pca = PCA(n_components=15, whiten=False)
data = pca.fit_transform(digits.data)

# use grid search cross-validation to optimize the bandwidth
```

(continues on next page)

(continued from previous page)

```
params = {'bandwidth': np.logspace(-1, 1, 20)}
grid = GridSearchCV(KernelDensity(), params)
grid.fit(data)

print("best bandwidth: {0}".format(grid.best_estimator_.bandwidth))

# use the best estimator to compute the kernel density estimate
kde = grid.best_estimator_

# sample 44 new points from the data
new_data = kde.sample(44, random_state=0)
new_data = pca.inverse_transform(new_data)

# turn data into a 4x11 grid
new_data = new_data.reshape((4, 11, -1))
real_data = digits.data[:44].reshape((4, 11, -1))

# plot real digits and resampled digits
fig, ax = plt.subplots(9, 11, subplot_kw=dict(xticks=[], yticks=[]))
for j in range(11):
    ax[4, j].set_visible(False)
    for i in range(4):
        im = ax[i, j].imshow(real_data[i, j].reshape((8, 8)),
                             cmap=plt.cm.binary, interpolation='nearest')
        im.set_clim(0, 16)
        im = ax[i + 5, j].imshow(new_data[i, j].reshape((8, 8)),
                                 cmap=plt.cm.binary, interpolation='nearest')
        im.set_clim(0, 16)

ax[0, 5].set_title('Selection from the input data')
ax[5, 5].set_title('"New" digits drawn from the kernel density model')

plt.show()
```

Total running time of the script: (0 minutes 4.927 seconds)

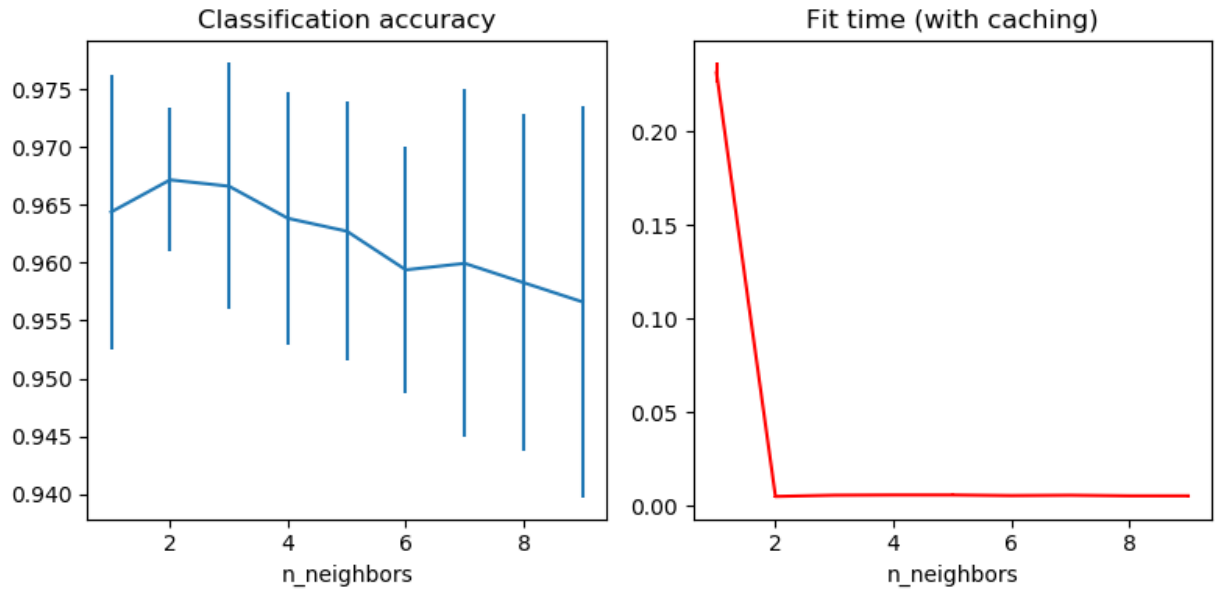
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.22.6 Caching nearest neighbors

This examples demonstrates how to precompute the k nearest neighbors before using them in KNeighborsClassifier. KNeighborsClassifier can compute the nearest neighbors internally, but precomputing them can have several benefits, such as finer parameter control, caching for multiple use, or custom implementations.

Here we use the caching property of pipelines to cache the nearest neighbors graph between multiple fits of KNeighborsClassifier. The first call is slow since it computes the neighbors graph, while subsequent call are faster as they do not need to recompute the graph. Here the durations are small since the dataset is small, but the gain can be more substantial when the dataset grows larger, or when the grid of parameter to search is large.




```
# Author: Tom Dupre la Tour
#
# License: BSD 3 clause
from tempfile import TemporaryDirectory
import matplotlib.pyplot as plt

from sklearn.neighbors import KNeighborsTransformer, KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.datasets import load_digits
from sklearn.pipeline import Pipeline

print(__doc__)

X, y = load_digits(return_X_y=True)
n_neighbors_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# The transformer computes the nearest neighbors graph using the maximum number
# of neighbors necessary in the grid search. The classifier model filters the
# nearest neighbors graph as required by its own n_neighbors parameter.
graph_model = KNeighborsTransformer(n_neighbors=max(n_neighbors_list),
                                    mode='distance')
classifier_model = KNeighborsClassifier(metric='precomputed')

# Note that we give `memory` a directory to cache the graph computation
# that will be used several times when tuning the hyperparameters of the
# classifier.
with TemporaryDirectory(prefix="sklearn_graph_cache_") as tmpdir:
    full_model = Pipeline(
        steps=[('graph', graph_model), ('classifier', classifier_model)],
        memory=tmpdir)

    param_grid = {'classifier__n_neighbors': n_neighbors_list}
    grid_model = GridSearchCV(full_model, param_grid)
    grid_model.fit(X, y)

# Plot the results of the grid search.
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(1, 2, figsize=(8, 4))
axes[0].errorbar(x=n_neighbors_list,
                 y=grid_model.cv_results_['mean_test_score'],
                 yerr=grid_model.cv_results_['std_test_score'])
axes[0].set(xlabel='n_neighbors', title='Classification accuracy')
axes[1].errorbar(x=n_neighbors_list, y=grid_model.cv_results_['mean_fit_time'],
                 yerr=grid_model.cv_results_['std_fit_time'], color='r')
axes[1].set(xlabel='n_neighbors', title='Fit time (with caching)')
fig.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 5.303 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.22.7 Neighborhood Components Analysis Illustration

This example illustrates a learned distance metric that maximizes the nearest neighbors classification accuracy. It provides a visual representation of this metric compared to the original point space. Please refer to the *User Guide* for more information.

```
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.neighbors import NeighborhoodComponentsAnalysis
from matplotlib import cm
from sklearn.utils.fixes import logsumexp

print(__doc__)
```

Original points

First we create a data set of 9 samples from 3 classes, and plot the points in the original space. For this example, we focus on the classification of point no. 3. The thickness of a link between point no. 3 and another point is proportional to their distance.

```
X, y = make_classification(n_samples=9, n_features=2, n_informative=2,
                           n_redundant=0, n_classes=3, n_clusters_per_class=1,
                           class_sep=1.0, random_state=0)

plt.figure(1)
ax = plt.gca()
for i in range(X.shape[0]):
    ax.text(X[i, 0], X[i, 1], str(i), va='center', ha='center')
    ax.scatter(X[i, 0], X[i, 1], s=300, c=cm.Set1(y[[i]]), alpha=0.4)

ax.set_title("Original points")
ax.axes.get_xaxis().set_visible(False)
```

(continues on next page)

(continued from previous page)

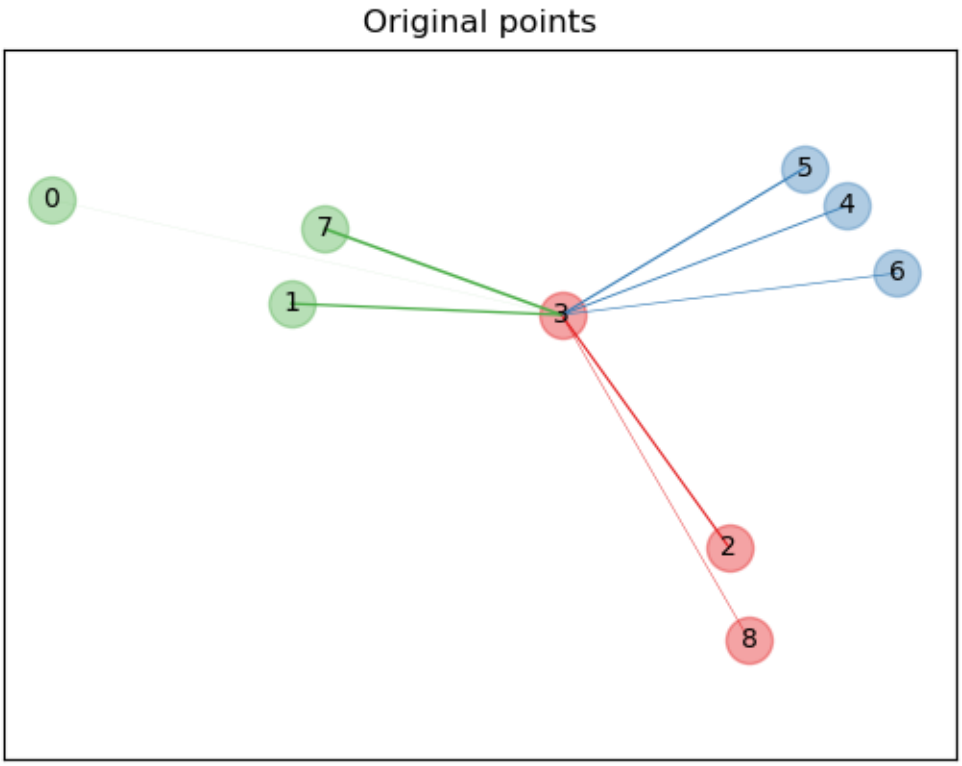
```
ax.axes.get_yaxis().set_visible(False)
ax.axis('equal')  # so that boundaries are displayed correctly as circles


def link_thickness_i(X, i):
    diff_embedded = X[i] - X
    dist_embedded = np.einsum('ij,ij->i', diff_embedded,
                              diff_embedded)
    dist_embedded[i] = np.inf

    # compute exponentiated distances (use the log-sum-exp trick to
    # avoid numerical instabilities
    exp_dist_embedded = np.exp(-dist_embedded -
                               logsumexp(-dist_embedded))
    return exp_dist_embedded


def relate_point(X, i, ax):
    pt_i = X[i]
    for j, pt_j in enumerate(X):
        thickness = link_thickness_i(X, i)
        if i != j:
            line = ([pt_i[0], pt_j[0]], [pt_i[1], pt_j[1]])
            ax.plot(*line, c=cm.Set1(y[j]),
                    linewidth=5*thickness[j])


i = 3
relate_point(X, i, ax)
plt.show()
```



Learning an embedding

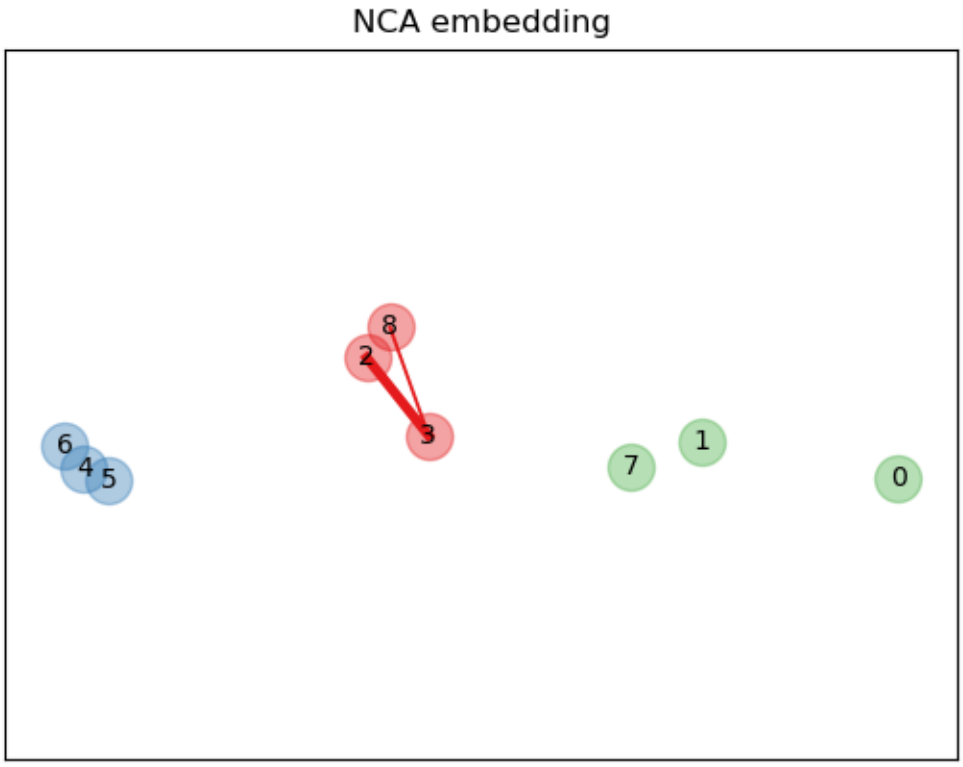
We use *NeighborhoodComponentsAnalysis* to learn an embedding and plot the points after the transformation. We then take the embedding and find the nearest neighbors.

```
nca = NeighborhoodComponentsAnalysis(max_iter=30, random_state=0)
nca = nca.fit(X, y)

plt.figure(2)
ax2 = plt.gca()
X_embedded = nca.transform(X)
relate_point(X_embedded, i, ax2)

for i in range(len(X)):
    ax2.text(X_embedded[i, 0], X_embedded[i, 1], str(i),
             va='center', ha='center')
    ax2.scatter(X_embedded[i, 0], X_embedded[i, 1], s=300, c=cm.Set1(y[[i]]),
                alpha=0.4)

ax2.set_title("NCA embedding")
ax2.axes.get_xaxis().set_visible(False)
ax2.axes.get_yaxis().set_visible(False)
ax2.axis('equal')
plt.show()
```



Total running time of the script: (0 minutes 1.758 seconds)

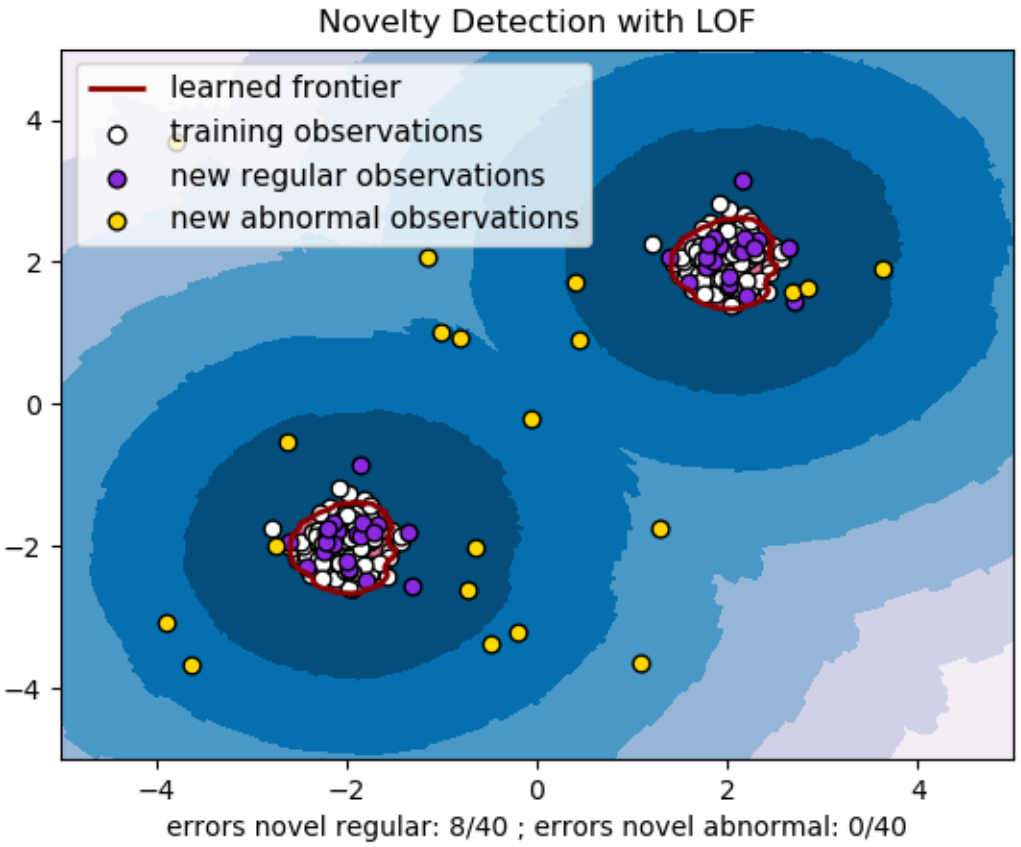
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.22.8 Novelty detection with Local Outlier Factor (LOF)

The Local Outlier Factor (LOF) algorithm is an unsupervised anomaly detection method which computes the local density deviation of a given data point with respect to its neighbors. It considers as outliers the samples that have a substantially lower density than their neighbors. This example shows how to use LOF for novelty detection. Note that when LOF is used for novelty detection you MUST not use predict, decision_function and score_samples on the training set as this would lead to wrong results. You must only use these methods on new unseen data (which are not in the training set). See *User Guide*: for details on the difference between outlier detection and novelty detection and how to use LOF for outlier detection.

The number of neighbors considered, (parameter n_neighbors) is typically set 1) greater than the minimum number of samples a cluster has to contain, so that other samples can be local outliers relative to this cluster, and 2) smaller than the maximum number of close by samples that can potentially be local outliers. In practice, such informations are generally not available, and taking n_neighbors=20 appears to work well in general.




```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from sklearn.neighbors import LocalOutlierFactor

print(__doc__)

np.random.seed(42)

xx, yy = np.meshgrid(np.linspace(-5, 5, 500), np.linspace(-5, 5, 500))
# Generate normal (not abnormal) training observations
X = 0.3 * np.random.randn(100, 2)
X_train = np.r_[X + 2, X - 2]
# Generate new normal (not abnormal) observations
X = 0.3 * np.random.randn(20, 2)
X_test = np.r_[X + 2, X - 2]
# Generate some abnormal novel observations
X_outliers = np.random.uniform(low=-4, high=4, size=(20, 2))

# fit the model for novelty detection (novelty=True)
clf = LocalOutlierFactor(n_neighbors=20, novelty=True, contamination=0.1)
clf.fit(X_train)
# DO NOT use predict, decision_function and score_samples on X_train as this
# would give wrong results but only on new unseen data (not used in X_train),
# e.g. X_test, X_outliers or the meshgrid
```

(continues on next page)

(continued from previous page)

```
y_pred_test = clf.predict(X_test)
y_pred_outliers = clf.predict(X_outliers)
n_error_test = y_pred_test[y_pred_test == -1].size
n_error_outliers = y_pred_outliers[y_pred_outliers == 1].size

# plot the learned frontier, the points, and the nearest vectors to the plane
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.title("Novelty Detection with LOF")
plt.contourf(xx, yy, Z, levels=np.linspace(Z.min(), 0, 7), cmap=plt.cm.PuBu)
a = plt.contour(xx, yy, Z, levels=[0], linewidths=2, colors='darkred')
plt.contourf(xx, yy, Z, levels=[0, Z.max()], colors='palevioletred')

s = 40
b1 = plt.scatter(X_train[:, 0], X_train[:, 1], c='white', s=s, edgecolors='k')
b2 = plt.scatter(X_test[:, 0], X_test[:, 1], c='blueviolet', s=s,
                 edgecolors='k')
c = plt.scatter(X_outliers[:, 0], X_outliers[:, 1], c='gold', s=s,
                edgecolors='k')
plt.axis('tight')
plt.xlim((-5, 5))
plt.ylim((-5, 5))
plt.legend([a.collections[0], b1, b2, c],
           ["learned frontier", "training observations",
            "new regular observations", "new abnormal observations"],
           loc="upper left",
           prop=matplotlib.font_manager.FontProperties(size=11))
plt.xlabel(
    "errors novel regular: %d/40 ; errors novel abnormal: %d/40"
    % (n_error_test, n_error_outliers))
plt.show()
```

Total running time of the script: (0 minutes 0.984 seconds)

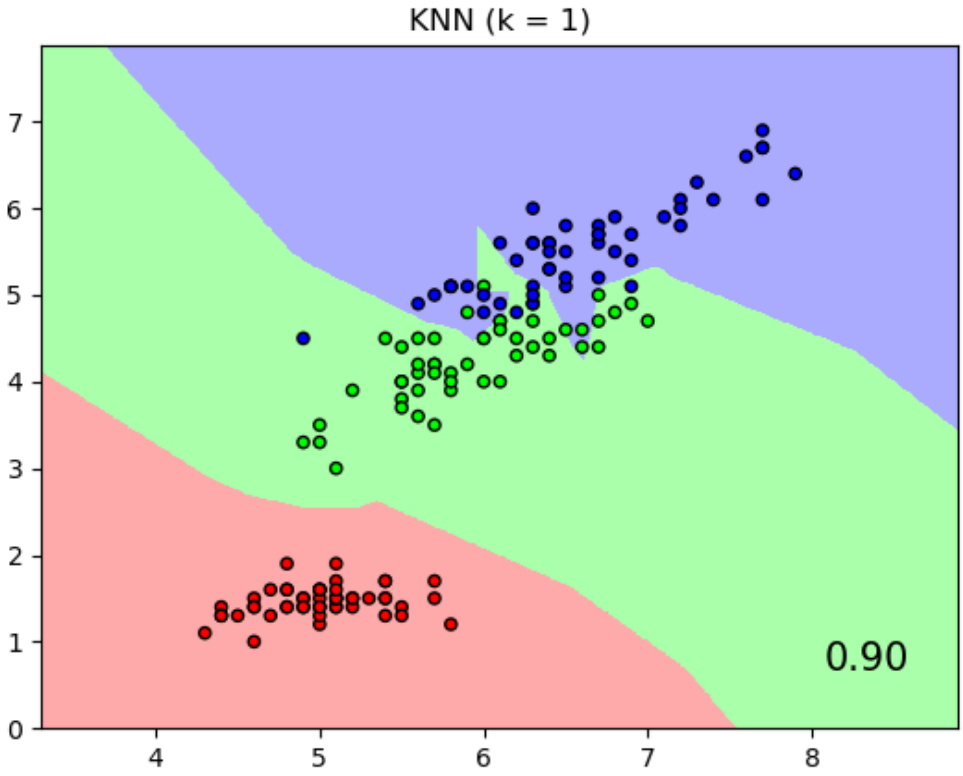
Estimated memory usage: 157 MB

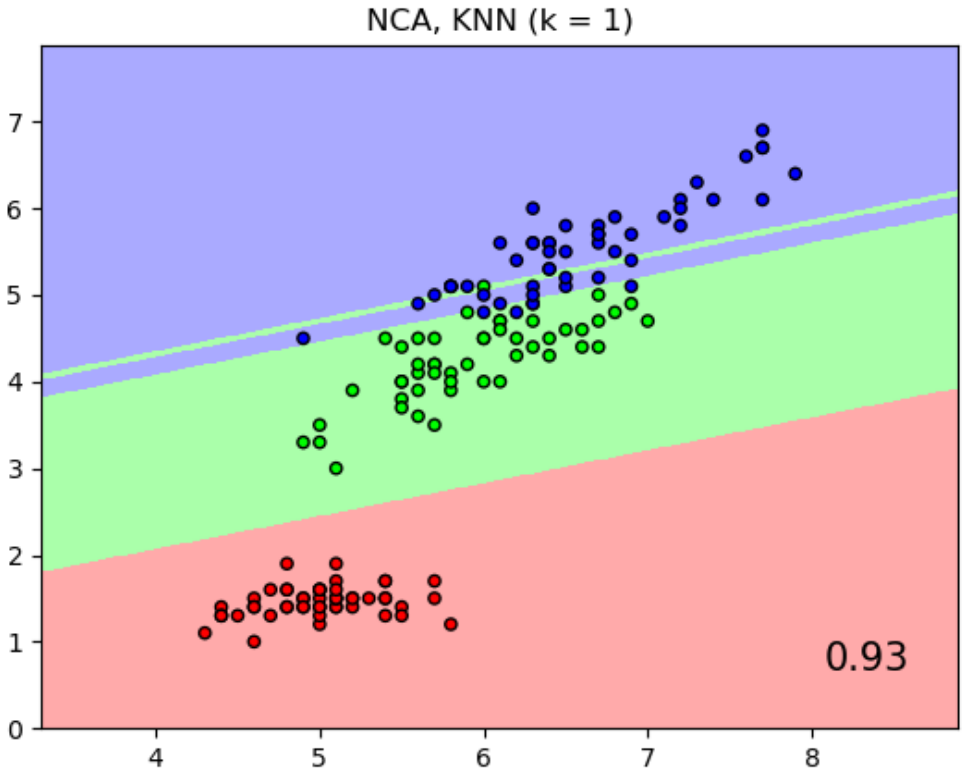
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.22.9 Comparing Nearest Neighbors with and without Neighborhood Components Analysis

An example comparing nearest neighbors classification with and without Neighborhood Components Analysis.

It will plot the class decision boundaries given by a Nearest Neighbors classifier when using the Euclidean distance on the original features, versus using the Euclidean distance after the transformation learned by Neighborhood Components Analysis. The latter aims to find a linear transformation that maximises the (stochastic) nearest neighbor classification accuracy on the training set.


KNN (k = 1)
0.90
0
1
2
3
4
5
6
7
4
5
6
7
8



-

```
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import (KNeighborsClassifier,
                               NeighborhoodComponentsAnalysis)
from sklearn.pipeline import Pipeline


print(__doc__)

n_neighbors = 1

dataset = datasets.load_iris()
X, y = dataset.data, dataset.target

# we only take two features. We could avoid this ugly
# slicing by using a two-dim dataset
X = X[:, [0, 2]]

X_train, X_test, y_train, y_test = \
    train_test_split(X, y, stratify=y, test_size=0.7, random_state=42)
```

(continues on next page)

(continued from previous page)

```
h = .01  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

names = ['KNN', 'NCA, KNN']

classifiers = [Pipeline([('scaler', StandardScaler()),
                         ('knn', KNeighborsClassifier(n_neighbors=n_neighbors))
                         ]),
               Pipeline([('scaler', StandardScaler()),
                         ('nca', NeighborhoodComponentsAnalysis()),
                         ('knn', KNeighborsClassifier(n_neighbors=n_neighbors))
                         ])
               ]

x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

for name, clf in zip(names, classifiers):

    clf.fit(X_train, y_train)
    score = clf.score(X_test, y_test)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light, alpha=.8)

    # Plot also the training and testing points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold, edgecolor='k', s=20)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("{} (k = {})".format(name, n_neighbors))
    plt.text(0.9, 0.1, '{:.2f}'.format(score), size=15,
             ha='center', va='center', transform=plt.gca().transAxes)

plt.show()
```

Total running time of the script: (0 minutes 18.150 seconds)

Estimated memory usage: 12 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.22.10 Dimensionality Reduction with Neighborhood Components Analysis

Sample usage of Neighborhood Components Analysis for dimensionality reduction.

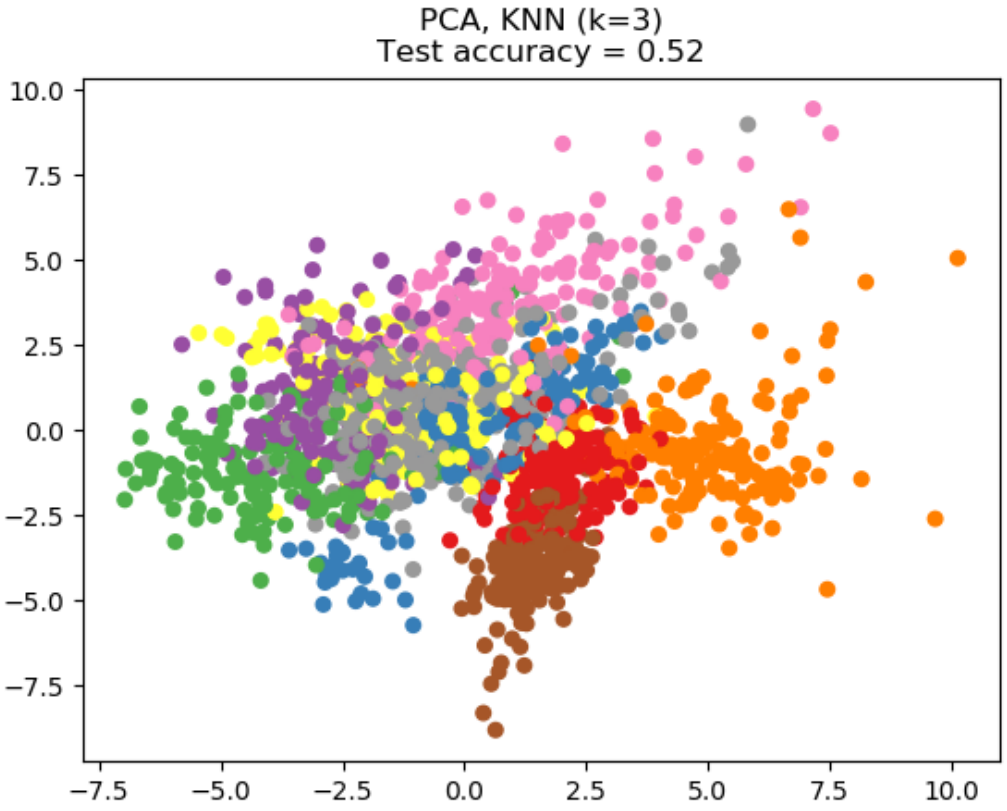
This example compares different (linear) dimensionality reduction methods applied on the Digits data set. The data set contains images of digits from 0 to 9 with approximately 180 samples of each class. Each image is of dimension 8x8 = 64, and is reduced to a two-dimensional data point.

Principal Component Analysis (PCA) applied to this data identifies the combination of attributes (principal components, or directions in the feature space) that account for the most variance in the data. Here we plot the different samples on the 2 first principal components.

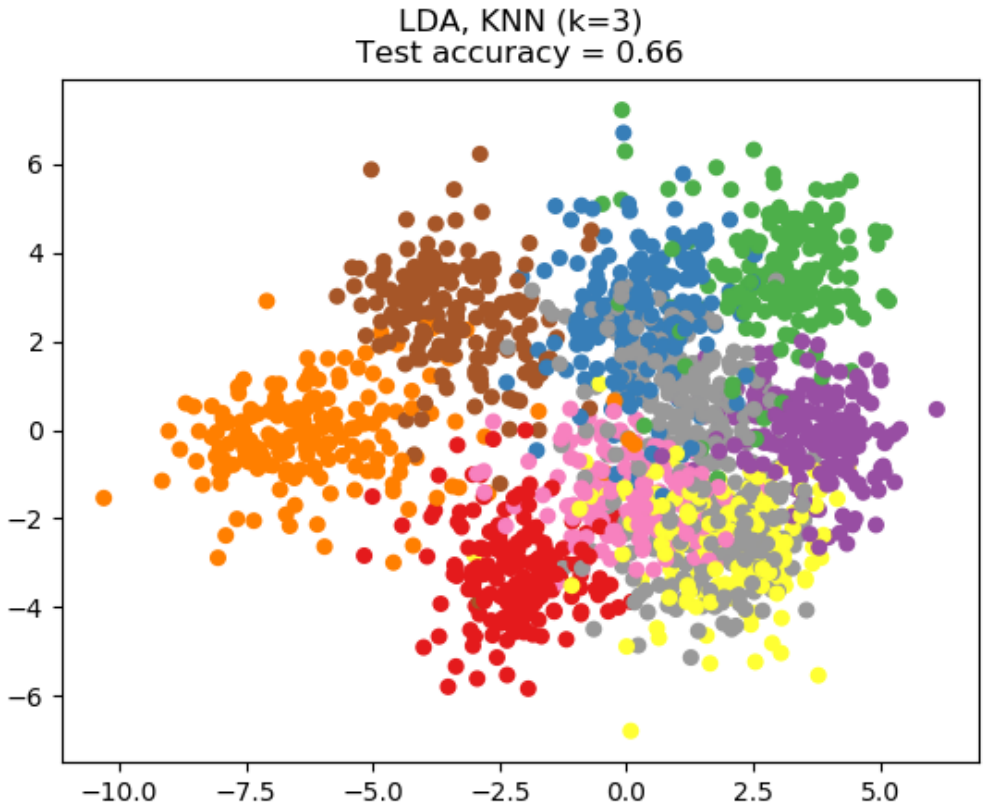
Linear Discriminant Analysis (LDA) tries to identify attributes that account for the most variance *between classes*. In particular, LDA, in contrast to PCA, is a supervised method, using known class labels.

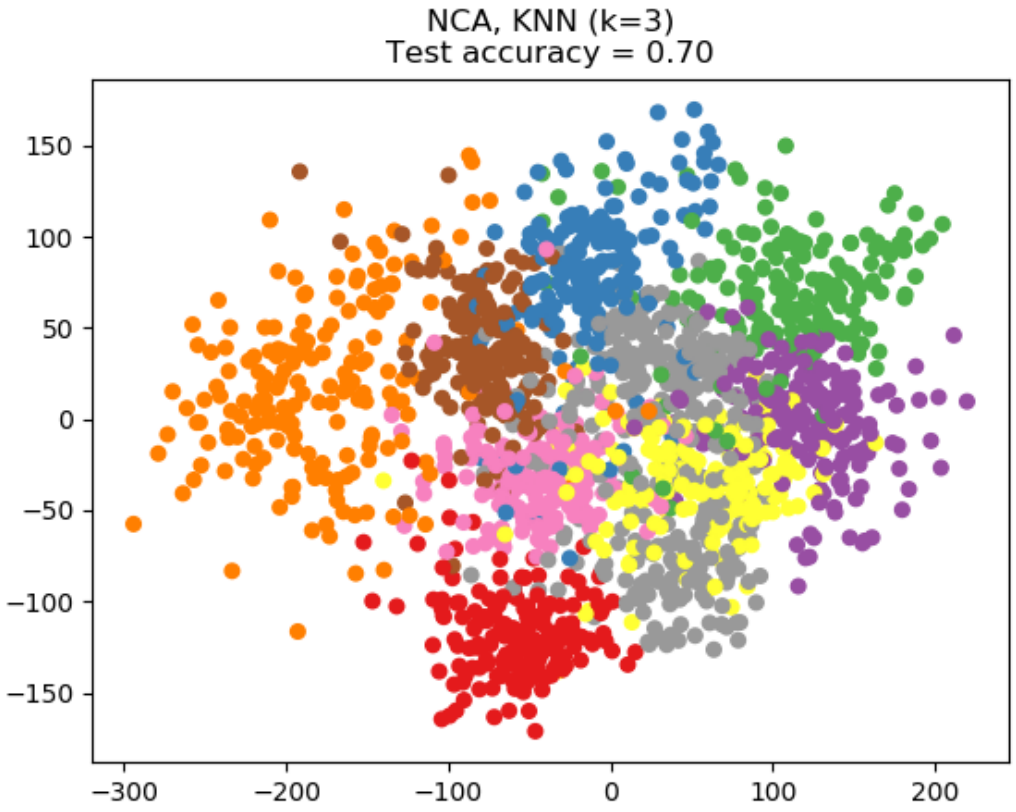
Neighborhood Components Analysis (NCA) tries to find a feature space such that a stochastic nearest neighbor algorithm will give the best accuracy. Like LDA, it is a supervised method.

One can see that NCA enforces a clustering of the data that is visually meaningful despite the large reduction in dimension.



•


LDA, KNN (k=3)
Test accuracy = 0.66
6
4
2
0
−2
−4
−6
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0



•

```
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neighbors import (KNeighborsClassifier,
                               NeighborhoodComponentsAnalysis)
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

print(__doc__)

n_neighbors = 3
random_state = 0

# Load Digits dataset
X, y = datasets.load_digits(return_X_y=True)

# Split into train/test
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.5, stratify=y,
                     random_state=random_state)

dim = len(X[0])
```

(continues on next page)

(continued from previous page)

```
n_classes = len(np.unique(y))

# Reduce dimension to 2 with PCA
pca = make_pipeline(StandardScaler(),
                    PCA(n_components=2, random_state=random_state))

# Reduce dimension to 2 with LinearDiscriminantAnalysis
lda = make_pipeline(StandardScaler(),
                    LinearDiscriminantAnalysis(n_components=2))

# Reduce dimension to 2 with NeighborhoodComponentAnalysis
nca = make_pipeline(StandardScaler(),
                    NeighborhoodComponentsAnalysis(n_components=2,
                                                   random_state=random_state))

# Use a nearest neighbor classifier to evaluate the methods
knn = KNeighborsClassifier(n_neighbors=n_neighbors)

# Make a list of the methods to be compared
dim_reduction_methods = [('PCA', pca), ('LDA', lda), ('NCA', nca)]

# plt.figure()
for i, (name, model) in enumerate(dim_reduction_methods):
    plt.figure()
    # plt.subplot(1, 3, i + 1, aspect=1)

    # Fit the method's model
    model.fit(X_train, y_train)

    # Fit a nearest neighbor classifier on the embedded training set
    knn.fit(model.transform(X_train), y_train)

    # Compute the nearest neighbor accuracy on the embedded test set
    acc_knn = knn.score(model.transform(X_test), y_test)

    # Embed the data set in 2 dimensions using the fitted model
    X_embedded = model.transform(X)

    # Plot the projected points and show the evaluation score
    plt.scatter(X_embedded[:, 0], X_embedded[:, 1], c=y, s=30, cmap='Set1')
    plt.title("{}, KNN (k={})\nTest accuracy = {:.2f}".format(name,
                                                              n_neighbors,
                                                              acc_knn))
plt.show()
```

Total running time of the script: (0 minutes 2.726 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.22.11 Kernel Density Estimate of Species Distributions

This shows an example of a neighbors-based query (in particular a kernel density estimate) on geospatial data, using a Ball Tree built upon the Haversine distance metric – i.e. distances over points in latitude/longitude. The dataset is provided by Phillips et. al. (2006). If available, the example uses basemap to plot the coast lines and national boundaries of South America.

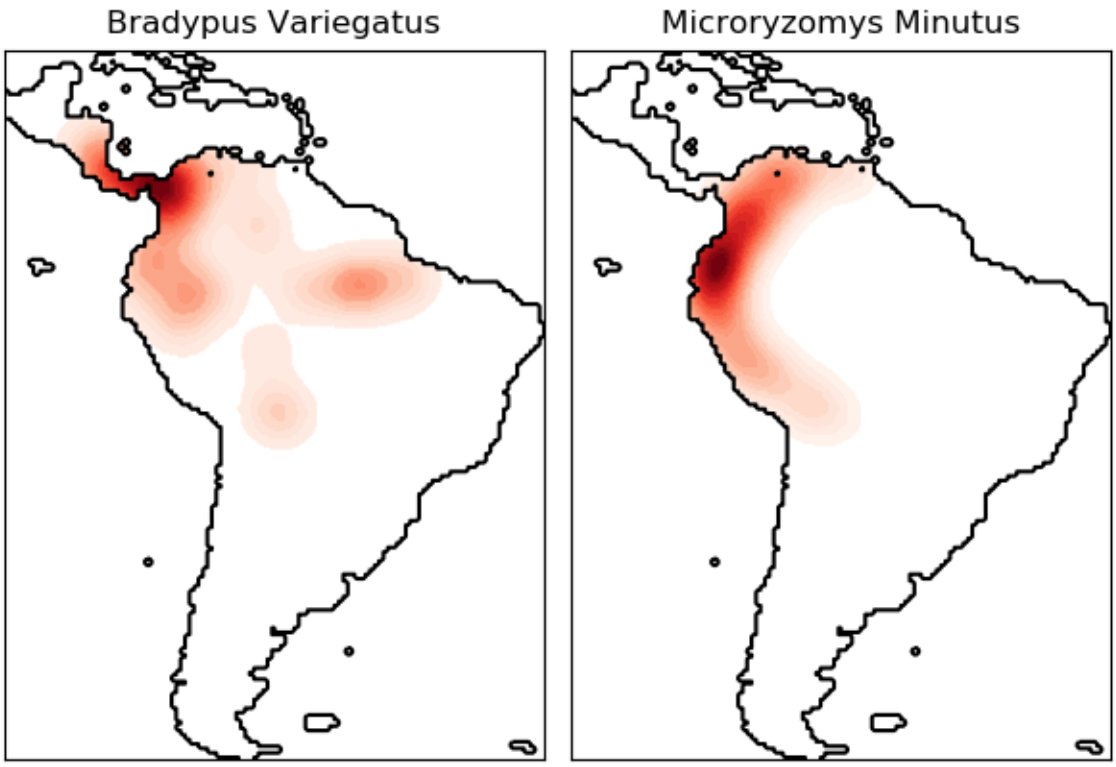
This example does not perform any learning over the data (see *Species distribution modeling* for an example of classification based on the attributes in this dataset). It simply shows the kernel density estimate of observed data points in geospatial coordinates.

The two species are:

- "Bradypus variegatus" , the Brown-throated Sloth.
- "Microryzomys minutus" , also known as the Forest Small Rice Rat, a rodent that lives in Peru, Colombia, Ecuador, Peru, and Venezuela.

References

- "Maximum entropy modeling of species geographic distributions" S. J. Phillips, R. P. Anderson, R. E. Schapire - Ecological Modelling, 190:231-259, 2006.



Out:

```
- computing KDE in spherical coordinates
- plot coastlines from coverage
- computing KDE in spherical coordinates
- plot coastlines from coverage
```

```
# Author: Jake Vanderplas <jakevdp@cs.washington.edu>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_species_distributions
from sklearn.neighbors import KernelDensity

# if basemap is available, we'll use it.
# otherwise, we'll improvise later...
try:
    from mpl_toolkits.basemap import Basemap
    basemap = True
except ImportError:
    basemap = False


def construct_grids(batch):
    """Construct the map grid from the batch object

    Parameters
    ----------
    batch : Batch object
        The object returned by :func:`fetch_species_distributions`

    Returns
    -------
    (xgrid, ygrid) : 1-D arrays
        The grid corresponding to the values in batch.coverages
    """
    # x,y coordinates for corner cells
    xmin = batch.x_left_lower_corner + batch.grid_size
    xmax = xmin + (batch.Nx * batch.grid_size)
    ymin = batch.y_left_lower_corner + batch.grid_size
    ymax = ymin + (batch.Ny * batch.grid_size)

    # x coordinates of the grid cells
    xgrid = np.arange(xmin, xmax, batch.grid_size)
    # y coordinates of the grid cells
    ygrid = np.arange(ymin, ymax, batch.grid_size)

    return (xgrid, ygrid)


# Get matrices/arrays of species IDs and locations
data = fetch_species_distributions()
```

(continues on next page)

(continued from previous page)

```
species_names = ['Bradypus Variegatus', 'Microryzomys Minutus']

Xtrain = np.vstack([data['train']['dd lat'],
                    data['train']['dd long']]).T
ytrain = np.array([d.decode('ascii').startswith('micro')
                   for d in data['train']['species']], dtype='int')
Xtrain *= np.pi / 180.  # Convert lat/long to radians

# Set up the data grid for the contour plot
xgrid, ygrid = construct_grids(data)
X, Y = np.meshgrid(xgrid[::5], ygrid[::5][::-1])
land_reference = data.coverages[6][::5, ::5]
land_mask = (land_reference > -9999).ravel()

xy = np.vstack([Y.ravel(), X.ravel()]).T
xy = xy[land_mask]
xy *= np.pi / 180.

# Plot map of South America with distributions of each species
fig = plt.figure()
fig.subplots_adjust(left=0.05, right=0.95, wspace=0.05)

for i in range(2):
    plt.subplot(1, 2, i + 1)

    # construct a kernel density estimate of the distribution
    print(" - computing KDE in spherical coordinates")
    kde = KernelDensity(bandwidth=0.04, metric='haversine',
                        kernel='gaussian', algorithm='ball_tree')
    kde.fit(Xtrain[ytrain == i])

    # evaluate only on the land: -9999 indicates ocean
    Z = np.full(land_mask.shape[0], -9999, dtype='int')
    Z[land_mask] = np.exp(kde.score_samples(xy))
    Z = Z.reshape(X.shape)

    # plot contours of the density
    levels = np.linspace(0, Z.max(), 25)
    plt.contourf(X, Y, Z, levels=levels, cmap=plt.cm.Reds)

    if basemap:
        print(" - plot coastlines using basemap")
        m = Basemap(projection='cyl', llcrnrlat=Y.min(),
                    urcrnrlat=Y.max(), llcrnrlon=X.min(),
                    urcrnrlon=X.max(), resolution='c')
        m.drawcoastlines()
        m.drawcountries()
    else:
        print(" - plot coastlines from coverage")
        plt.contour(X, Y, land_reference,
                    levels=[-9998], colors="k",
                    linestyles="solid")
        plt.xticks([])
        plt.yticks([])

    plt.title(species_names[i])
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

Total running time of the script: (0 minutes 6.553 seconds)

Estimated memory usage: 60 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

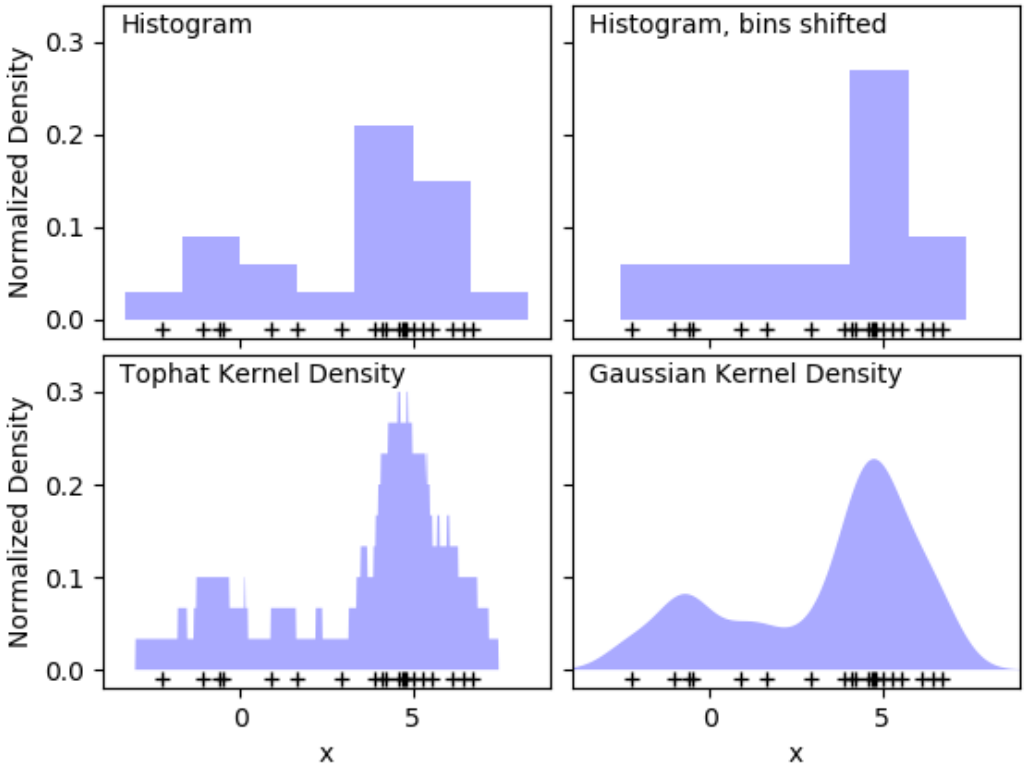
6.22.12 Simple 1D Kernel Density Estimation

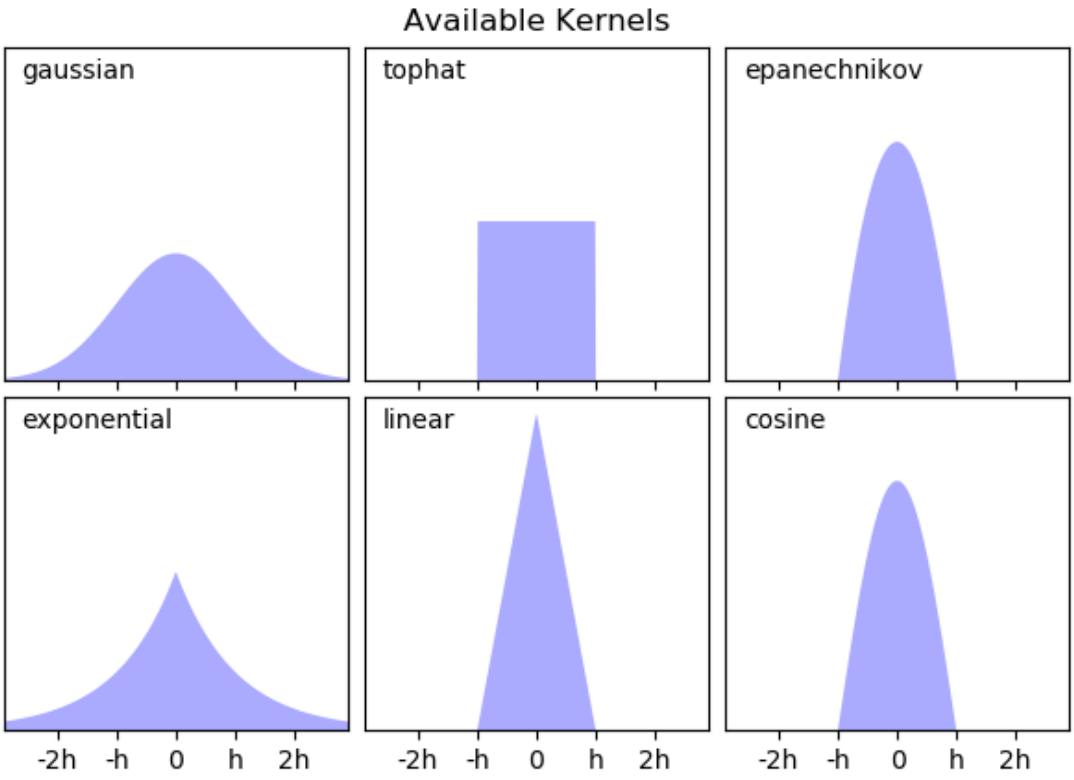
This example uses the `sklearn.neighbors.KernelDensity` class to demonstrate the principles of Kernel Density Estimation in one dimension.

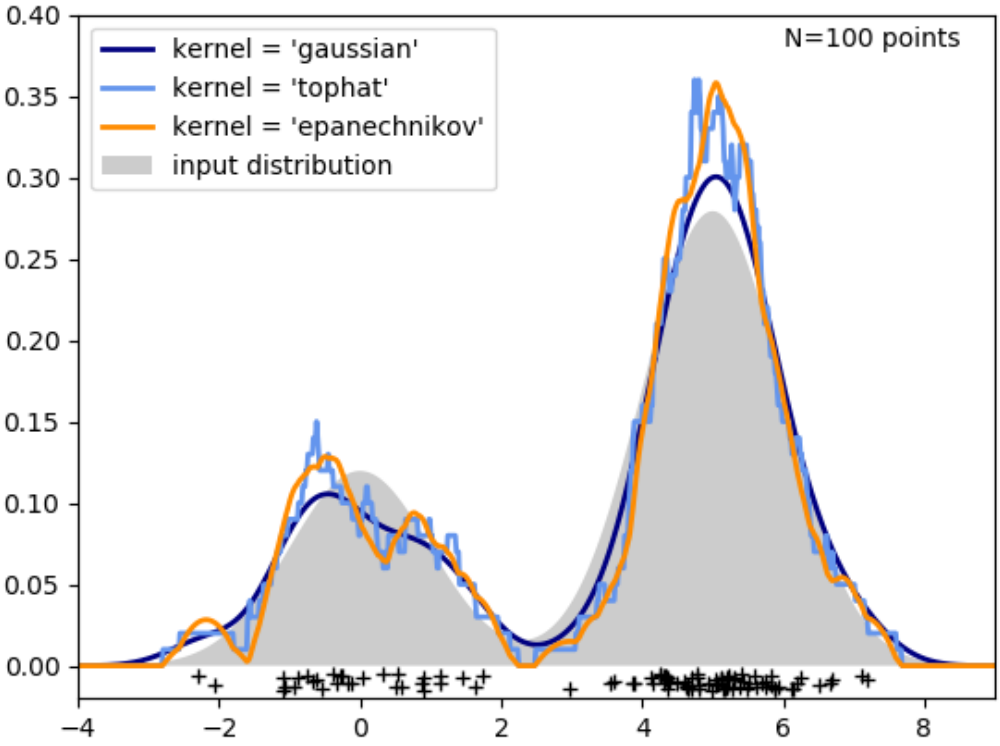
The first plot shows one of the problems with using histograms to visualize the density of points in 1D. Intuitively, a histogram can be thought of as a scheme in which a unit "block" is stacked above each point on a regular grid. As the top two panels show, however, the choice of gridding for these blocks can lead to wildly divergent ideas about the underlying shape of the density distribution. If we instead center each block on the point it represents, we get the estimate shown in the bottom left panel. This is a kernel density estimation with a "top hat" kernel. This idea can be generalized to other kernel shapes: the bottom-right panel of the first figure shows a Gaussian kernel density estimate over the same distribution.

Scikit-learn implements efficient kernel density estimation using either a Ball Tree or KD Tree structure, through the `sklearn.neighbors.KernelDensity` estimator. The available kernels are shown in the second figure of this example.

The third figure compares kernel density estimates for a distribution of 100 samples in 1 dimension. Though this example uses 1D distributions, kernel density estimation is easily and efficiently extensible to higher dimensions as well.


Histogram
Histogram, bins shifted
Tophat Kernel Density
Gaussian Kernel Density
Normalized Density
Normalized Density
0.0
0.1
0.2
0.3
0
5
x
x


Available Kernels
gaussian
tophat
epanechnikov
exponential
linear
cosine
-2h
-h
0
h
2h




```
# Author: Jake Vanderplas <jakevdp@cs.washington.edu>
#
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from distutils.version import LooseVersion
from scipy.stats import norm
from sklearn.neighbors import KernelDensity

# `normed` is being deprecated in favor of `density` in histograms
if LooseVersion(matplotlib.__version__) >= '2.1':
    density_param = {'density': True}
else:
    density_param = {'normed': True}

# ----------------------------------------------------------------------
# Plot the progression of histograms to kernels
np.random.seed(1)
N = 20
X = np.concatenate((np.random.normal(0, 1, int(0.3 * N)),
                    np.random.normal(5, 1, int(0.7 * N))))[:, np.newaxis]
X_plot = np.linspace(-5, 10, 1000)[:, np.newaxis]
bins = np.linspace(-5, 10, 10)

fig, ax = plt.subplots(2, 2, sharex=True, sharey=True)
fig.subplots_adjust(hspace=0.05, wspace=0.05)
```

(continues on next page)

(continued from previous page)

```
# histogram 1
ax[0, 0].hist(X[:, 0], bins=bins, fc='#AAAAFF', **density_param)
ax[0, 0].text(-3.5, 0.31, "Histogram")

# histogram 2
ax[0, 1].hist(X[:, 0], bins=bins + 0.75, fc='#AAAAFF', **density_param)
ax[0, 1].text(-3.5, 0.31, "Histogram, bins shifted")

# tophat KDE
kde = KernelDensity(kernel='tophat', bandwidth=0.75).fit(X)
log_dens = kde.score_samples(X_plot)
ax[1, 0].fill(X_plot[:, 0], np.exp(log_dens), fc='#AAAAFF')
ax[1, 0].text(-3.5, 0.31, "Tophat Kernel Density")

# Gaussian KDE
kde = KernelDensity(kernel='gaussian', bandwidth=0.75).fit(X)
log_dens = kde.score_samples(X_plot)
ax[1, 1].fill(X_plot[:, 0], np.exp(log_dens), fc='#AAAAFF')
ax[1, 1].text(-3.5, 0.31, "Gaussian Kernel Density")

for axi in ax.ravel():
    axi.plot(X[:, 0], np.full(X.shape[0], -0.01), '+k')
    axi.set_xlim(-4, 9)
    axi.set_ylim(-0.02, 0.34)

for axi in ax[:, 0]:
    axi.set_ylabel('Normalized Density')

for axi in ax[1, :]:
    axi.set_xlabel('x')

# ----------------------------------------------------------------------
# Plot all available kernels
X_plot = np.linspace(-6, 6, 1000)[:, None]
X_src = np.zeros((1, 1))

fig, ax = plt.subplots(2, 3, sharex=True, sharey=True)
fig.subplots_adjust(left=0.05, right=0.95, hspace=0.05, wspace=0.05)


def format_func(x, loc):
    if x == 0:
        return '0'
    elif x == 1:
        return 'h'
    elif x == -1:
        return '-h'
    else:
        return '%ih' % x

for i, kernel in enumerate(['gaussian', 'tophat', 'epanechnikov',
                            'exponential', 'linear', 'cosine']):
    axi = ax.ravel()[i]
    log_dens = KernelDensity(kernel=kernel).fit(X_src).score_samples(X_plot)
    axi.fill(X_plot[:, 0], np.exp(log_dens), '-k', fc='#AAAAFF')
    axi.text(-2.6, 0.95, kernel)
```

(continues on next page)

(continued from previous page)

```
    axi.xaxis.set_major_formatter(plt.FuncFormatter(format_func))
    axi.xaxis.set_major_locator(plt.MultipleLocator(1))
    axi.yaxis.set_major_locator(plt.NullLocator())

    axi.set_ylim(0, 1.05)
    axi.set_xlim(-2.9, 2.9)

ax[0, 1].set_title('Available Kernels')

# ----------------------------------------------------------------------
# Plot a 1D density example
N = 100
np.random.seed(1)
X = np.concatenate((np.random.normal(0, 1, int(0.3 * N)),
                    np.random.normal(5, 1, int(0.7 * N))))[:, np.newaxis]

X_plot = np.linspace(-5, 10, 1000)[:, np.newaxis]

true_dens = (0.3 * norm(0, 1).pdf(X_plot[:, 0])
             + 0.7 * norm(5, 1).pdf(X_plot[:, 0]))

fig, ax = plt.subplots()
ax.fill(X_plot[:, 0], true_dens, fc='black', alpha=0.2,
        label='input distribution')
colors = ['navy', 'cornflowerblue', 'darkorange']
kernels = ['gaussian', 'tophat', 'epanechnikov']
lw = 2

for color, kernel in zip(colors, kernels):
    kde = KernelDensity(kernel=kernel, bandwidth=0.5).fit(X)
    log_dens = kde.score_samples(X_plot)
    ax.plot(X_plot[:, 0], np.exp(log_dens), color=color, lw=lw,
            linestyle='-', label="kernel = '{0}'".format(kernel))

ax.text(6, 0.38, "N={0} points".format(N))

ax.legend(loc='upper left')
ax.plot(X[:, 0], -0.005 - 0.01 * np.random.random(X.shape[0]), '+k')

ax.set_xlim(-4, 9)
ax.set_ylim(-0.02, 0.4)
plt.show()
```

Total running time of the script: (0 minutes 0.880 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.22.13 Approximate nearest neighbors in TSNE

This example presents how to chain KNeighborsTransformer and TSNE in a pipeline. It also shows how to wrap the packages `annoy` and `nmslib` to replace KNeighborsTransformer and perform approximate nearest neighbors. These packages can be installed with `pip install annoy nmslib`.

Note: Currently `TSNE(metric='precomputed')` does not modify the precomputed distances, and thus assumes that precomputed euclidean distances are squared. In future versions, a parameter in TSNE will control the optional squaring of precomputed distances (see #12401).

Note: In KNeighborsTransformer we use the definition which includes each training point as its own neighbor in the count of `n_neighbors`, and for compatibility reasons, one extra neighbor is computed when `mode == 'distance'`. Please note that we do the same in the proposed wrappers.

Sample output:

```
Benchmarking on MNIST_2000:
---------------------------
AnnoyTransformer:                     0.583 sec
NMSlibTransformer:                    0.321 sec
KNeighborsTransformer:                1.225 sec
TSNE with AnnoyTransformer:           4.903 sec
TSNE with NMSlibTransformer:          5.009 sec
TSNE with KNeighborsTransformer:      6.210 sec
TSNE with internal NearestNeighbors: 6.365 sec

Benchmarking on MNIST_10000:
----------------------------
AnnoyTransformer:                     4.457 sec
NMSlibTransformer:                    2.080 sec
KNeighborsTransformer:                30.680 sec
TSNE with AnnoyTransformer:           30.225 sec
TSNE with NMSlibTransformer:          43.295 sec
TSNE with KNeighborsTransformer:      64.845 sec
TSNE with internal NearestNeighbors: 64.984 sec
```

```
# Author: Tom Dupre la Tour
#
# License: BSD 3 clause
import time
import sys

try:
    import annoy
except ImportError:
    print("The package 'annoy' is required to run this example.")
    sys.exit()

try:
    import nmslib
except ImportError:
    print("The package 'nmslib' is required to run this example.")
    sys.exit()

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import NullFormatter
from scipy.sparse import csr_matrix

from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.neighbors import KNeighborsTransformer
from sklearn.utils._testing import assert_array_almost_equal
from sklearn.datasets import fetch_openml
from sklearn.pipeline import make_pipeline
```

(continues on next page)

(continued from previous page)

```
from sklearn.manifold import TSNE
from sklearn.utils import shuffle

print(__doc__)


class NMSlibTransformer(TransformerMixin, BaseEstimator):
    """Wrapper for using nmslib as sklearn's KNeighborsTransformer"""

    def __init__(self, n_neighbors=5, metric='euclidean', method='sw-graph',
                 n_jobs=1):
        self.n_neighbors = n_neighbors
        self.method = method
        self.metric = metric
        self.n_jobs = n_jobs

    def fit(self, X):
        self.n_samples_fit_ = X.shape[0]

        # see more metric in the manual
        # https://github.com/nmslib/nmslib/tree/master/manual
        space = {
            'sqeuclidean': 'l2',
            'euclidean': 'l2',
            'cosine': 'cosinesimil',
            'l1': 'l1',
            'l2': 'l2',
        }[self.metric]

        self.nmslib_ = nmslib.init(method=self.method, space=space)
        self.nmslib_.addDataPointBatch(X)
        self.nmslib_.createIndex()
        return self

    def transform(self, X):
        n_samples_transform = X.shape[0]

        # For compatibility reasons, as each sample is considered as its own
        # neighbor, one extra neighbor will be computed.
        n_neighbors = self.n_neighbors + 1

        results = self.nmslib_.knnQueryBatch(X, k=n_neighbors,
                                             num_threads=self.n_jobs)
        indices, distances = zip(*results)
        indices, distances = np.vstack(indices), np.vstack(distances)

        if self.metric == 'sqeuclidean':
            distances **= 2

        indptr = np.arange(0, n_samples_transform * n_neighbors + 1,
                           n_neighbors)
        kneighbors_graph = csr_matrix((distances.ravel(), indices.ravel(),
                                       indptr), shape=(n_samples_transform,
                                                       self.n_samples_fit_))

        return kneighbors_graph
```

(continues on next page)

(continued from previous page)

```
class AnnoyTransformer(TransformerMixin, BaseEstimator):
    """Wrapper for using annoy.AnnoyIndex as sklearn's KNeighborsTransformer"""

    def __init__(self, n_neighbors=5, metric='euclidean', n_trees=10,
                 search_k=-1):
        self.n_neighbors = n_neighbors
        self.n_trees = n_trees
        self.search_k = search_k
        self.metric = metric

    def fit(self, X):
        self.n_samples_fit_ = X.shape[0]
        metric = self.metric if self.metric != 'sqeuclidean' else 'euclidean'
        self.annoy_ = annoy.AnnoyIndex(X.shape[1], metric=metric)
        for i, x in enumerate(X):
            self.annoy_.add_item(i, x.tolist())
        self.annoy_.build(self.n_trees)
        return self

    def transform(self, X):
        return self._transform(X)

    def fit_transform(self, X, y=None):
        return self.fit(X)._transform(X=None)

    def _transform(self, X):
        """As `transform`, but handles X is None for faster `fit_transform`."""

        n_samples_transform = self.n_samples_fit_ if X is None else X.shape[0]

        # For compatibility reasons, as each sample is considered as its own
        # neighbor, one extra neighbor will be computed.
        n_neighbors = self.n_neighbors + 1

        indices = np.empty((n_samples_transform, n_neighbors),
                           dtype=np.int)
        distances = np.empty((n_samples_transform, n_neighbors))

        if X is None:
            for i in range(self.annoy_.get_n_items()):
                ind, dist = self.annoy_.get_nns_by_item(
                    i, n_neighbors, self.search_k, include_distances=True)

                indices[i], distances[i] = ind, dist
        else:
            for i, x in enumerate(X):
                indices[i], distances[i] = self.annoy_.get_nns_by_vector(
                    x.tolist(), n_neighbors, self.search_k,
                    include_distances=True)

        if self.metric == 'sqeuclidean':
            distances **= 2

        indptr = np.arange(0, n_samples_transform * n_neighbors + 1,
                           n_neighbors)
        kneighbors_graph = csr_matrix((distances.ravel(), indices.ravel(),
```

(continues on next page)

(continued from previous page)

```
                                       indptr), shape=(n_samples_transform,
                                                       self.n_samples_fit_))

        return kneighbors_graph


def test_transformers():
    """Test that AnnoyTransformer and KNeighborsTransformer give same results
    """
    X = np.random.RandomState(42).randn(10, 2)

    knn = KNeighborsTransformer()
    Xt0 = knn.fit_transform(X)

    ann = AnnoyTransformer()
    Xt1 = ann.fit_transform(X)

    nms = NMSlibTransformer()
    Xt2 = nms.fit_transform(X)

    assert_array_almost_equal(Xt0.toarray(), Xt1.toarray(), decimal=5)
    assert_array_almost_equal(Xt0.toarray(), Xt2.toarray(), decimal=5)


def load_mnist(n_samples):
    """Load MNIST, shuffle the data, and return only n_samples."""
    mnist = fetch_openml(data_id=41063)
    X, y = shuffle(mnist.data, mnist.target, random_state=42)
    return X[:n_samples], y[:n_samples]


def run_benchmark():
    datasets = [
        ('MNIST_2000', load_mnist(n_samples=2000)),
        ('MNIST_10000', load_mnist(n_samples=10000)),
    ]

    n_iter = 500
    perplexity = 30
    # TSNE requires a certain number of neighbors which depends on the
    # perplexity parameter.
    # Add one since we include each sample as its own neighbor.
    n_neighbors = int(3. * perplexity + 1) + 1

    transformers = [
        ('AnnoyTransformer', AnnoyTransformer(n_neighbors=n_neighbors,
                                              metric='sqeuclidean')),
        ('NMSlibTransformer', NMSlibTransformer(n_neighbors=n_neighbors,
                                                metric='sqeuclidean')),
        ('KNeighborsTransformer', KNeighborsTransformer(
            n_neighbors=n_neighbors, mode='distance', metric='sqeuclidean')),
        ('TSNE with AnnoyTransformer', make_pipeline(
            AnnoyTransformer(n_neighbors=n_neighbors, metric='sqeuclidean'),
            TSNE(metric='precomputed', perplexity=perplexity,
                 method="barnes_hut", random_state=42, n_iter=n_iter), )),
        ('TSNE with NMSlibTransformer', make_pipeline(
            NMSlibTransformer(n_neighbors=n_neighbors, metric='sqeuclidean'),
```

(continues on next page)

(continued from previous page)

```
            TSNE(metric='precomputed', perplexity=perplexity,
                 method="barnes_hut", random_state=42, n_iter=n_iter), )),
        ('TSNE with KNeighborsTransformer', make_pipeline(
            KNeighborsTransformer(n_neighbors=n_neighbors, mode='distance',
                                  metric='sqeuclidean'),
            TSNE(metric='precomputed', perplexity=perplexity,
                 method="barnes_hut", random_state=42, n_iter=n_iter), )),
        ('TSNE with internal NearestNeighbors',
         TSNE(metric='sqeuclidean', perplexity=perplexity, method="barnes_hut",
              random_state=42, n_iter=n_iter)),
    ]

    # init the plot
    nrows = len(datasets)
    ncols = np.sum([1 for name, model in transformers if 'TSNE' in name])
    fig, axes = plt.subplots(nrows=nrows, ncols=ncols, squeeze=False,
                             figsize=(5 * ncols, 4 * nrows))
    axes = axes.ravel()
    i_ax = 0

    for dataset_name, (X, y) in datasets:

        msg = 'Benchmarking on %s:' % dataset_name
        print('\n%s\n%s' % (msg, '-' * len(msg)))

        for transformer_name, transformer in transformers:
            start = time.time()
            Xt = transformer.fit_transform(X)
            duration = time.time() - start

            # print the duration report
            longest = np.max([len(name) for name, model in transformers])
            whitespaces = ' ' * (longest - len(transformer_name))
            print('%s: %s%.3f sec' % (transformer_name, whitespaces, duration))

            # plot TSNE embedding which should be very similar across methods
            if 'TSNE' in transformer_name:
                axes[i_ax].set_title(transformer_name + '\non ' + dataset_name)
                axes[i_ax].scatter(Xt[:, 0], Xt[:, 1], c=y, alpha=0.2,
                                   cmap=plt.cm.viridis)
                axes[i_ax].xaxis.set_major_formatter(NullFormatter())
                axes[i_ax].yaxis.set_major_formatter(NullFormatter())
                axes[i_ax].axis('tight')
                i_ax += 1

    fig.tight_layout()
    plt.show()


if __name__ == '__main__':
    test_transformers()
    run_benchmark()
```

Total running time of the script: (0 minutes 0.000 seconds)

Estimated memory usage: 0 MB

6.23 Neural Networks

Examples concerning the *sklearn.neural_network* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

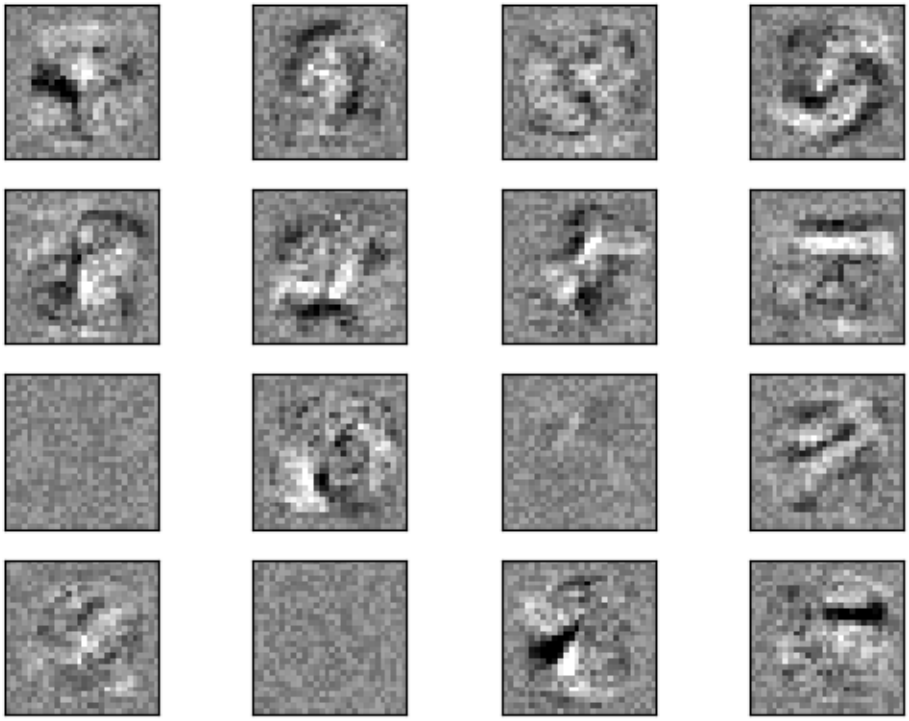
6.23.1 Visualization of MLP weights on MNIST

Sometimes looking at the learned coefficients of a neural network can provide insight into the learning behavior. For example if weights look unstructured, maybe some were not used at all, or if very large coefficients exist, maybe regularization was too low or the learning rate too high.

This example shows how to plot some of the first layer weights in a MLPClassifier trained on the MNIST dataset.

The input data consists of 28x28 pixel handwritten digits, leading to 784 features in the dataset. Therefore the first layer weight matrix have the shape (784, hidden_layer_sizes[0]). We can therefore visualize a single column of the weight matrix as a 28x28 pixel image.

To make the example run faster, we use very few hidden units, and train only for a very short time. Training longer would result in weights with a much smoother spatial appearance.



Out:

```
Iteration 1, loss = 0.32009978
Iteration 2, loss = 0.15347534
Iteration 3, loss = 0.11544755
Iteration 4, loss = 0.09279764
Iteration 5, loss = 0.07889367
Iteration 6, loss = 0.07170497
Iteration 7, loss = 0.06282111
Iteration 8, loss = 0.05530788
Iteration 9, loss = 0.04960484
Iteration 10, loss = 0.04645355
Training set score: 0.986800
Test set score: 0.970000
```

```
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.neural_network import MLPClassifier

print(__doc__)

# Load data from https://www.openml.org/d/554
X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
X = X / 255.

# rescale the data, use the traditional train/test split
X_train, X_test = X[:60000], X[60000:]
y_train, y_test = y[:60000], y[60000:]

mlp = MLPClassifier(hidden_layer_sizes=(50,), max_iter=10, alpha=1e-4,
                    solver='sgd', verbose=10, random_state=1,
                    learning_rate_init=.1)

mlp.fit(X_train, y_train)
print("Training set score: %f" % mlp.score(X_train, y_train))
print("Test set score: %f" % mlp.score(X_test, y_test))

fig, axes = plt.subplots(4, 4)
# use global min / max to ensure all weights are shown on the same scale
vmin, vmax = mlp.coefs_[0].min(), mlp.coefs_[0].max()
for coef, ax in zip(mlp.coefs_[0].T, axes.ravel()):
    ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray, vmin=.5 * vmin,
               vmax=.5 * vmax)
    ax.set_xticks(())
    ax.set_yticks(())

plt.show()
```

Total running time of the script: (0 minutes 50.627 seconds)

Estimated memory usage: 1148 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

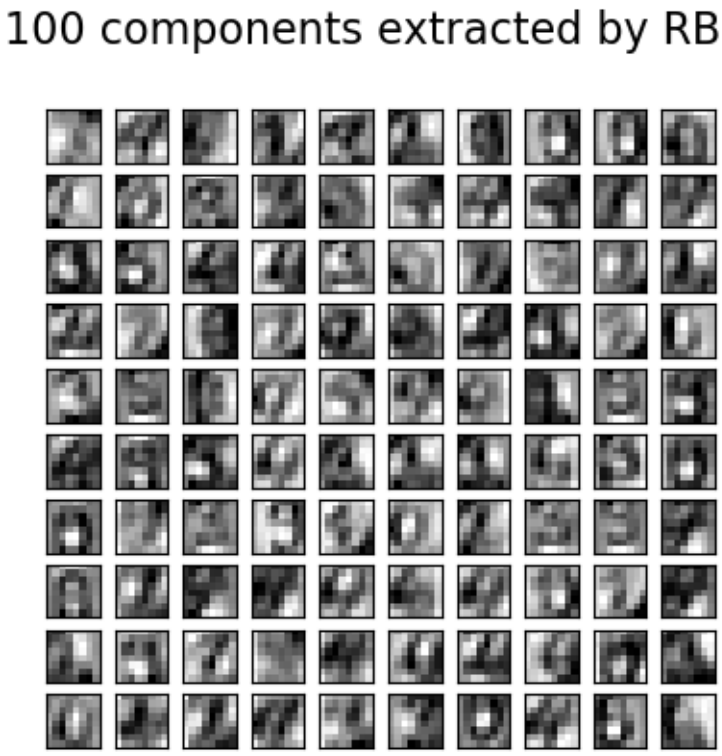
6.23.2 Restricted Boltzmann Machine features for digit classification

For greyscale image data where pixel values can be interpreted as degrees of blackness on a white background, like handwritten digit recognition, the Bernoulli Restricted Boltzmann machine model (*BernoulliRBM*) can perform effective non-linear feature extraction.

In order to learn good latent representations from a small dataset, we artificially generate more labeled data by perturbing the training data with linear shifts of 1 pixel in each direction.

This example shows how to build a classification pipeline with a BernoulliRBM feature extractor and a *LogisticRegression* classifier. The hyperparameters of the entire model (learning rate, hidden layer size, regularization) were optimized by grid search, but the search is not reproduced here because of runtime constraints.

Logistic regression on raw pixel values is presented for comparison. The example shows that the features extracted by the BernoulliRBM help improve the classification accuracy.



Out:

```
[BernoulliRBM] Iteration 1, pseudo-likelihood = -25.39, time = 0.15s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -23.77, time = 0.22s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -22.94, time = 0.22s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -21.91, time = 0.21s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -21.69, time = 0.21s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -21.06, time = 0.22s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -20.89, time = 0.20s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -20.64, time = 0.22s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -20.36, time = 0.23s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -20.09, time = 0.28s
Logistic regression using RBM features:
              precision    recall  f1-score   support

           0       0.99      0.98      0.99       174
```

(continues on next page)

(continued from previous page)

```
              1       0.91      0.93      0.92       184
              2       0.94      0.96      0.95       166
              3       0.96      0.90      0.93       194
              4       0.97      0.94      0.96       186
              5       0.91      0.92      0.92       181
              6       0.98      0.97      0.97       207
              7       0.94      0.98      0.96       154
              8       0.91      0.90      0.90       182
              9       0.87      0.91      0.89       169

    accuracy                           0.94      1797
   macro avg       0.94      0.94      0.94      1797
weighted avg       0.94      0.94      0.94      1797


Logistic regression using raw pixel features:
              precision    recall  f1-score   support

              0       0.90      0.92      0.91       174
              1       0.60      0.58      0.59       184
              2       0.76      0.85      0.80       166
              3       0.78      0.79      0.78       194
              4       0.82      0.84      0.83       186
              5       0.76      0.76      0.76       181
              6       0.90      0.87      0.89       207
              7       0.85      0.88      0.87       154
              8       0.67      0.58      0.62       182
              9       0.75      0.76      0.75       169

    accuracy                           0.78      1797
   macro avg       0.78      0.78      0.78      1797
weighted avg       0.78      0.78      0.78      1797
```

```
print(__doc__)

# Authors: Yann N. Dauphin, Vlad Niculae, Gabriel Synnaeve
# License: BSD

import numpy as np
import matplotlib.pyplot as plt

from scipy.ndimage import convolve
from sklearn import linear_model, datasets, metrics
from sklearn.model_selection import train_test_split
from sklearn.neural_network import BernoulliRBM
from sklearn.pipeline import Pipeline
from sklearn.base import clone


# #############################################################################
# Setting up
```

(continues on next page)

(continued from previous page)

```
def nudge_dataset(X, Y):
    """
    This produces a dataset 5 times bigger than the original one,
    by moving the 8x8 images in X around by 1px to left, right, down, up
    """
    direction_vectors = [
        [[0, 1, 0],
         [0, 0, 0],
         [0, 0, 0]],

        [[0, 0, 0],
         [1, 0, 0],
         [0, 0, 0]],

        [[0, 0, 0],
         [0, 0, 1],
         [0, 0, 0]],

        [[0, 0, 0],
         [0, 0, 0],
         [0, 1, 0]]]

    def shift(x, w):
        return convolve(x.reshape((8, 8)), mode='constant', weights=w).ravel()

    X = np.concatenate([X] +
                       [np.apply_along_axis(shift, 1, X, vector)
                        for vector in direction_vectors])
    Y = np.concatenate([Y for _ in range(5)], axis=0)
    return X, Y


# Load Data
X, y = datasets.load_digits(return_X_y=True)
X = np.asarray(X, 'float32')
X, Y = nudge_dataset(X, y)
X = (X - np.min(X, 0)) / (np.max(X, 0) + 0.0001)  # 0-1 scaling

X_train, X_test, Y_train, Y_test = train_test_split(
    X, Y, test_size=0.2, random_state=0)

# Models we will use
logistic = linear_model.LogisticRegression(solver='newton-cg', tol=1)
rbm = BernoulliRBM(random_state=0, verbose=True)

rbm_features_classifier = Pipeline(
    steps=[('rbm', rbm), ('logistic', logistic)])

# #############################################################################
# Training

# Hyper-parameters. These were set by cross-validation,
# using a GridSearchCV. Here we are not performing cross-validation to
# save time.
rbm.learning_rate = 0.06
rbm.n_iter = 10
```

(continues on next page)

(continued from previous page)

```
# More components tend to give better prediction performance, but larger
# fitting time
rbm.n_components = 100
logistic.C = 6000

# Training RBM-Logistic Pipeline
rbm_features_classifier.fit(X_train, Y_train)

# Training the Logistic regression classifier directly on the pixel
raw_pixel_classifier = clone(logistic)
raw_pixel_classifier.C = 100.
raw_pixel_classifier.fit(X_train, Y_train)

# ##############################################################################
# Evaluation

Y_pred = rbm_features_classifier.predict(X_test)
print("Logistic regression using RBM features:\n%s\n" % (
    metrics.classification_report(Y_test, Y_pred)))

Y_pred = raw_pixel_classifier.predict(X_test)
print("Logistic regression using raw pixel features:\n%s\n" % (
    metrics.classification_report(Y_test, Y_pred)))

# ##############################################################################
# Plotting

plt.figure(figsize=(4.2, 4))
for i, comp in enumerate(rbm.components_):
    plt.subplot(10, 10, i + 1)
    plt.imshow(comp.reshape((8, 8)), cmap=plt.cm.gray_r,
               interpolation='nearest')
    plt.xticks(())
    plt.yticks(())
plt.suptitle('100 components extracted by RBM', fontsize=16)
plt.subplots_adjust(0.08, 0.02, 0.92, 0.85, 0.08, 0.23)

plt.show()
```

Total running time of the script: (0 minutes 6.556 seconds)

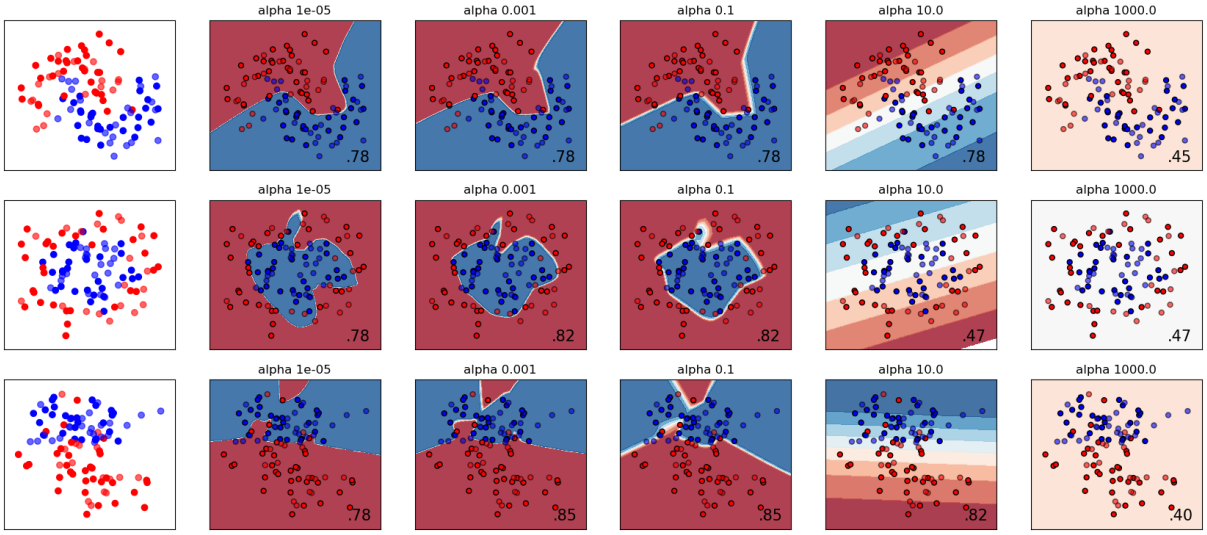
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.23.3 Varying regularization in Multi-layer Perceptron

A comparison of different values for regularization parameter 'alpha' on synthetic datasets. The plot shows that different alphas yield different decision functions.

Alpha is a parameter for regularization term, aka penalty term, that combats overfitting by constraining the size of the weights. Increasing alpha may fix high variance (a sign of overfitting) by encouraging smaller weights, resulting in a decision boundary plot that appears with lesser curvatures. Similarly, decreasing alpha may fix high bias (a sign of underfitting) by encouraging larger weights, potentially resulting in a more complicated decision boundary.




```
print(__doc__)


# Author: Issam H. Laradji
# License: BSD 3 clause

import numpy as np
from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neural_network import MLPClassifier

h = .02  # step size in the mesh

alphas = np.logspace(-5, 3, 5)
names = ['alpha ' + str(i) for i in alphas]

classifiers = []
for i in alphas:
    classifiers.append(MLPClassifier(solver='lbfgs', alpha=i, random_state=1,
                                     hidden_layer_sizes=[100, 100]))

X, y = make_classification(n_features=2, n_redundant=0, n_informative=2,
                           random_state=0, n_clusters_per_class=1)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [make_moons(noise=0.3, random_state=0),
            make_circles(noise=0.2, factor=0.5, random_state=1),
            linearly_separable]
```

(continues on next page)

(continued from previous page)

```
figure = plt.figure(figsize=(17, 9))
i = 1
# iterate over datasets
for X, y in datasets:
    # preprocess dataset, split into training and test part
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.4)

    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # just plot the dataset first
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])
    ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
    # Plot the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright)
    # and testing points
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.6)
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    i += 1

    # iterate over classifiers
    for name, clf in zip(names, classifiers):
        ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
        clf.fit(X_train, y_train)
        score = clf.score(X_test, y_test)

        # Plot the decision boundary. For that, we will assign a color to each
        # point in the mesh [x_min, x_max]x[y_min, y_max].
        if hasattr(clf, "decision_function"):
            Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
        else:
            Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        ax.contourf(xx, yy, Z, cmap=cm, alpha=.8)

        # Plot also the training points
        ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
                   edgecolors='black', s=25)
        # and testing points
        ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
                   alpha=0.6, edgecolors='black', s=25)

        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
        ax.set_xticks(())
        ax.set_yticks(())
        ax.set_title(name)
        ax.text(xx.max() - .3, yy.min() + .3, ('%.2f' % score).lstrip('0'),
```

(continues on next page)

(continued from previous page)

```
                size=15, horizontalalignment='right')
        i += 1

figure.subplots_adjust(left=.02, right=.98)
plt.show()
```

Total running time of the script: (0 minutes 4.626 seconds)

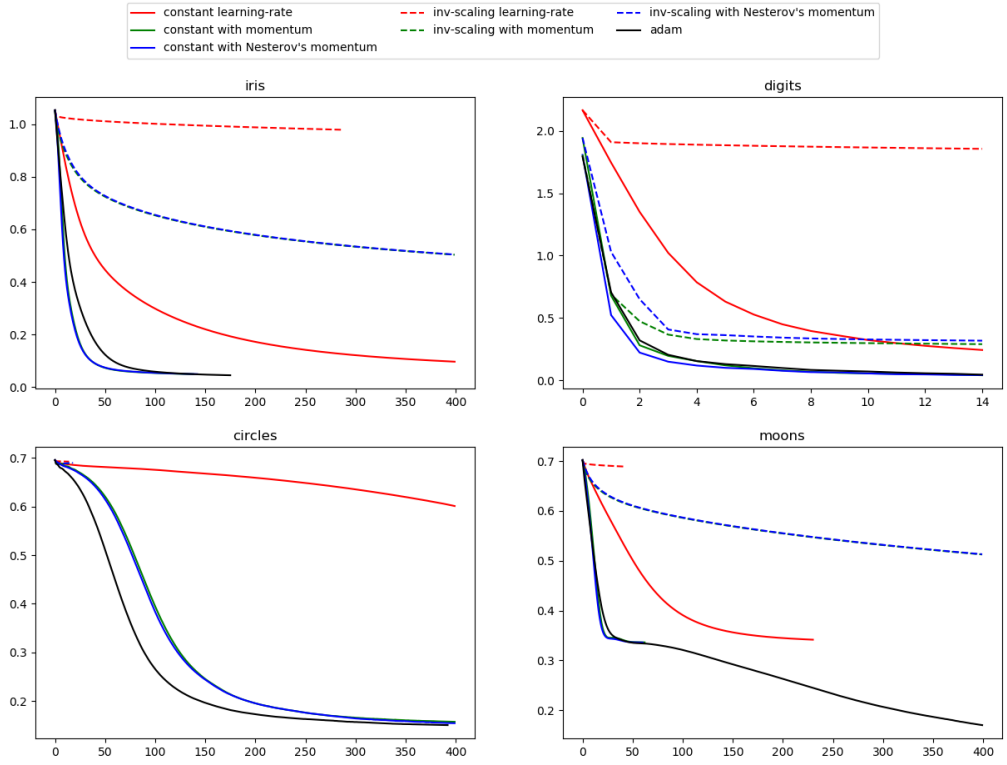
Estimated memory usage: 65 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.23.4 Compare Stochastic learning strategies for MLPClassifier

This example visualizes some training loss curves for different stochastic learning strategies, including SGD and Adam. Because of time-constraints, we use several small datasets, for which L-BFGS might be more suitable. The general trend shown in these examples seems to carry over to larger datasets, however.

Note that those results can be highly dependent on the value of `learning_rate_init`.



Out:

```
learning on dataset iris
training: constant learning-rate
Training set score: 0.980000
```

(continues on next page)

(continued from previous page)

```
Training set loss: 0.096950
training: constant with momentum
Training set score: 0.980000
Training set loss: 0.049530
training: constant with Nesterov's momentum
Training set score: 0.980000
Training set loss: 0.049540
training: inv-scaling learning-rate
Training set score: 0.360000
Training set loss: 0.978444
training: inv-scaling with momentum
Training set score: 0.860000
Training set loss: 0.503452
training: inv-scaling with Nesterov's momentum
Training set score: 0.860000
Training set loss: 0.504185
training: adam
Training set score: 0.980000
Training set loss: 0.045311

learning on dataset digits
training: constant learning-rate
Training set score: 0.956038
Training set loss: 0.243802
training: constant with momentum
Training set score: 0.992766
Training set loss: 0.041297
training: constant with Nesterov's momentum
Training set score: 0.993879
Training set loss: 0.042898
training: inv-scaling learning-rate
Training set score: 0.638843
Training set loss: 1.855465
training: inv-scaling with momentum
Training set score: 0.912632
Training set loss: 0.290584
training: inv-scaling with Nesterov's momentum
Training set score: 0.909293
Training set loss: 0.318387
training: adam
Training set score: 0.991653
Training set loss: 0.045934

learning on dataset circles
training: constant learning-rate
Training set score: 0.840000
Training set loss: 0.601052
training: constant with momentum
Training set score: 0.940000
Training set loss: 0.157334
training: constant with Nesterov's momentum
Training set score: 0.940000
Training set loss: 0.154453
training: inv-scaling learning-rate
Training set score: 0.500000
Training set loss: 0.692470
training: inv-scaling with momentum
```

(continues on next page)

(continued from previous page)

```
Training set score: 0.500000
Training set loss: 0.689143
training: inv-scaling with Nesterov's momentum
Training set score: 0.500000
Training set loss: 0.689751
training: adam
Training set score: 0.940000
Training set loss: 0.150527

learning on dataset moons
training: constant learning-rate
Training set score: 0.850000
Training set loss: 0.341523
training: constant with momentum
Training set score: 0.850000
Training set loss: 0.336188
training: constant with Nesterov's momentum
Training set score: 0.850000
Training set loss: 0.335919
training: inv-scaling learning-rate
Training set score: 0.500000
Training set loss: 0.689015
training: inv-scaling with momentum
Training set score: 0.830000
Training set loss: 0.512595
training: inv-scaling with Nesterov's momentum
Training set score: 0.830000
Training set loss: 0.513034
training: adam
Training set score: 0.930000
Training set loss: 0.170087
```

```
print(__doc__)

import warnings

import matplotlib.pyplot as plt

from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import MinMaxScaler
from sklearn import datasets
from sklearn.exceptions import ConvergenceWarning

# different learning rate schedules and momentum parameters
params = [{'solver': 'sgd', 'learning_rate': 'constant', 'momentum': 0,
           'learning_rate_init': 0.2},
          {'solver': 'sgd', 'learning_rate': 'constant', 'momentum': .9,
           'nesterovs_momentum': False, 'learning_rate_init': 0.2},
          {'solver': 'sgd', 'learning_rate': 'constant', 'momentum': .9,
           'nesterovs_momentum': True, 'learning_rate_init': 0.2},
          {'solver': 'sgd', 'learning_rate': 'invscaling', 'momentum': 0,
```

(continues on next page)

(continued from previous page)

```
          'learning_rate_init': 0.2},
         {'solver': 'sgd', 'learning_rate': 'invscaling', 'momentum': .9,
          'nesterovs_momentum': True, 'learning_rate_init': 0.2},
         {'solver': 'sgd', 'learning_rate': 'invscaling', 'momentum': .9,
          'nesterovs_momentum': False, 'learning_rate_init': 0.2},
         {'solver': 'adam', 'learning_rate_init': 0.01}]

labels = ["constant learning-rate", "constant with momentum",
          "constant with Nesterov's momentum",
          "inv-scaling learning-rate", "inv-scaling with momentum",
          "inv-scaling with Nesterov's momentum", "adam"]

plot_args = [{'c': 'red', 'linestyle': '-'},
             {'c': 'green', 'linestyle': '-'},
             {'c': 'blue', 'linestyle': '-'},
             {'c': 'red', 'linestyle': '--'},
             {'c': 'green', 'linestyle': '--'},
             {'c': 'blue', 'linestyle': '--'},
             {'c': 'black', 'linestyle': '-'}]


def plot_on_dataset(X, y, ax, name):
    # for each dataset, plot learning for each learning strategy
    print("\nlearning on dataset %s" % name)
    ax.set_title(name)

    X = MinMaxScaler().fit_transform(X)
    mlps = []
    if name == "digits":
        # digits is larger but converges fairly quickly
        max_iter = 15
    else:
        max_iter = 400

    for label, param in zip(labels, params):
        print("training: %s" % label)
        mlp = MLPClassifier(random_state=0,
                            max_iter=max_iter, **param)

        # some parameter combinations will not converge as can be seen on the
        # plots so they are ignored here
        with warnings.catch_warnings():
            warnings.filterwarnings("ignore", category=ConvergenceWarning,
                                    module="sklearn")
            mlp.fit(X, y)

        mlps.append(mlp)
        print("Training set score: %f" % mlp.score(X, y))
        print("Training set loss: %f" % mlp.loss_)
    for mlp, label, args in zip(mlps, labels, plot_args):
        ax.plot(mlp.loss_curve_, label=label, **args)


fig, axes = plt.subplots(2, 2, figsize=(15, 10))
# load / generate some toy datasets
iris = datasets.load_iris()
X_digits, y_digits = datasets.load_digits(return_X_y=True)
```

(continues on next page)

(continued from previous page)

```
data_sets = [(iris.data, iris.target),
             (X_digits, y_digits),
             datasets.make_circles(noise=0.2, factor=0.5, random_state=1),
             datasets.make_moons(noise=0.3, random_state=0)]

for ax, data, name in zip(axes.ravel(), data_sets, ['iris', 'digits',
                                                    'circles', 'moons']):
    plot_on_dataset(*data, ax=ax, name=name)

fig.legend(ax.get_lines(), labels, ncol=3, loc="upper center")
plt.show()
```

Total running time of the script: (0 minutes 4.017 seconds)

Estimated memory usage: 8 MB

6.24 Pipelines and composite estimators

Examples of how to compose transformers and pipelines from other estimators. See the *User Guide*.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.24.1 Concatenating multiple feature extraction methods

In many real-world examples, there are many ways to extract features from a dataset. Often it is beneficial to combine several methods to obtain good performance. This example shows how to use `FeatureUnion` to combine features obtained by PCA and univariate selection.

Combining features using this transformer has the benefit that it allows cross validation and grid searches over the whole process.

The combination used in this example is not particularly helpful on this dataset and is only used to illustrate the usage of FeatureUnion.

Out:

```
Combined space has 3 features
Fitting 5 folds for each of 18 candidates, totalling 90 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=0.1
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=0.1, score=0.
↪933, total=   0.0s
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=0.1
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=0.1, score=0.
↪933, total=   0.0s
[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.0s remaining:    0.0s
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=0.1
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=0.1, score=0.
↪867, total=   0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    0.0s remaining:    0.0s
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=0.1
```

(continues on next page)

(continued from previous page)

```
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=0.1, score=0.
↪933, total=   0.0s
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    0.0s remaining:    0.0s
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=0.1
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=0.1, score=1.
↪000, total=   0.0s
[Parallel(n_jobs=1)]: Done   5 out of   5 | elapsed:    0.0s remaining:    0.0s
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=1
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=1, score=0.
↪900, total=   0.0s
[Parallel(n_jobs=1)]: Done   6 out of   6 | elapsed:    0.0s remaining:    0.0s
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=1
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=1, score=1.
↪000, total=   0.0s
[Parallel(n_jobs=1)]: Done   7 out of   7 | elapsed:    0.0s remaining:    0.0s
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=1
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=1, score=0.
↪867, total=   0.0s
[Parallel(n_jobs=1)]: Done   8 out of   8 | elapsed:    0.0s remaining:    0.0s
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=1
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=1, score=0.
↪933, total=   0.0s
[Parallel(n_jobs=1)]: Done   9 out of   9 | elapsed:    0.0s remaining:    0.0s
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=1
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=10, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=10, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=10, score=0.
↪900, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=10, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=1, features__univ_select__k=1, svm__C=10, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=0.1
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=0.1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=0.1
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=0.1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=0.1
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=0.1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=0.1
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=0.1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=0.1
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=0.1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=1
```

(continues on next page)

(continued from previous page)

```
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=1
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=1
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=1
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=1
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=10
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=10, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=10
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=10, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=10
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=10, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=10
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=10, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=1, features__univ_select__k=2, svm__C=10
[CV]  features__pca__n_components=1, features__univ_select__k=2, svm__C=10, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=0.1
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=0.1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=0.1
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=0.1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=0.1
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=0.1, score=0.
↪867, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=0.1
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=0.1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=0.1
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=0.1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=1
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=1
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=1
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=1
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=1
```

(continues on next page)

(continued from previous page)

```
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=10, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=10, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=10, score=0.
↪900, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=10, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=2, features__univ_select__k=1, svm__C=10, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=0.1
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=0.1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=0.1
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=0.1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=0.1
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=0.1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=0.1
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=0.1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=0.1
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=0.1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=1
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=1
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=1
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=1
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=1
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=10
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=10, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=10
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=10, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=10
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=10, score=0.
↪900, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=10
```

(continues on next page)

(continued from previous page)

```
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=10, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=2, features__univ_select__k=2, svm__C=10
[CV]  features__pca__n_components=2, features__univ_select__k=2, svm__C=10, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=0.1
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=0.1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=0.1
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=0.1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=0.1
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=0.1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=0.1
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=0.1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=0.1
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=0.1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=1
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=1
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=1
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=1
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=1
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=10, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=10, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=10, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=10, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=1, svm__C=10
[CV]  features__pca__n_components=3, features__univ_select__k=1, svm__C=10, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=0.1
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=0.1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=0.1
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=0.1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=0.1
```

(continues on next page)

(continued from previous page)

```
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=0.1, score=0.
↪933, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=0.1
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=0.1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=0.1
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=0.1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=1
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=1
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=1
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=1
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=1, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=1
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=1, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=10
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=10, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=10
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=10, score=1.
↪000, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=10
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=10, score=0.
↪900, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=10
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=10, score=0.
↪967, total=   0.0s
[CV] features__pca__n_components=3, features__univ_select__k=2, svm__C=10
[CV]  features__pca__n_components=3, features__univ_select__k=2, svm__C=10, score=1.
↪000, total=   0.0s
[Parallel(n_jobs=1)]: Done  90 out of  90 | elapsed:    0.3s finished
Pipeline(steps=[('features',
                 FeatureUnion(transformer_list=[('pca', PCA(n_components=3)),
                                                ('univ_select',
                                                 SelectKBest(k=1))])),
                ('svm', SVC(C=10, kernel='linear'))])
```

```
# Author: Andreas Mueller <amueller@ais.uni-bonn.de>
#
# License: BSD 3 clause

from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.model_selection import GridSearchCV
```

(continues on next page)

(continued from previous page)

```
from sklearn.svm import SVC
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest

iris = load_iris()

X, y = iris.data, iris.target

# This dataset is way too high-dimensional. Better do PCA:
pca = PCA(n_components=2)

# Maybe some original features where good, too?
selection = SelectKBest(k=1)

# Build estimator from PCA and Univariate selection:

combined_features = FeatureUnion([("pca", pca), ("univ_select", selection)])

# Use combined features to transform dataset:
X_features = combined_features.fit(X, y).transform(X)
print("Combined space has", X_features.shape[1], "features")

svm = SVC(kernel="linear")

# Do grid search over k, n_components and C:

pipeline = Pipeline([("features", combined_features), ("svm", svm)])

param_grid = dict(features__pca__n_components=[1, 2, 3],
                  features__univ_select__k=[1, 2],
                  svm__C=[0.1, 1, 10])

grid_search = GridSearchCV(pipeline, param_grid=param_grid, verbose=10)
grid_search.fit(X, y)
print(grid_search.best_estimator_)
```

Total running time of the script: (0 minutes 0.621 seconds)

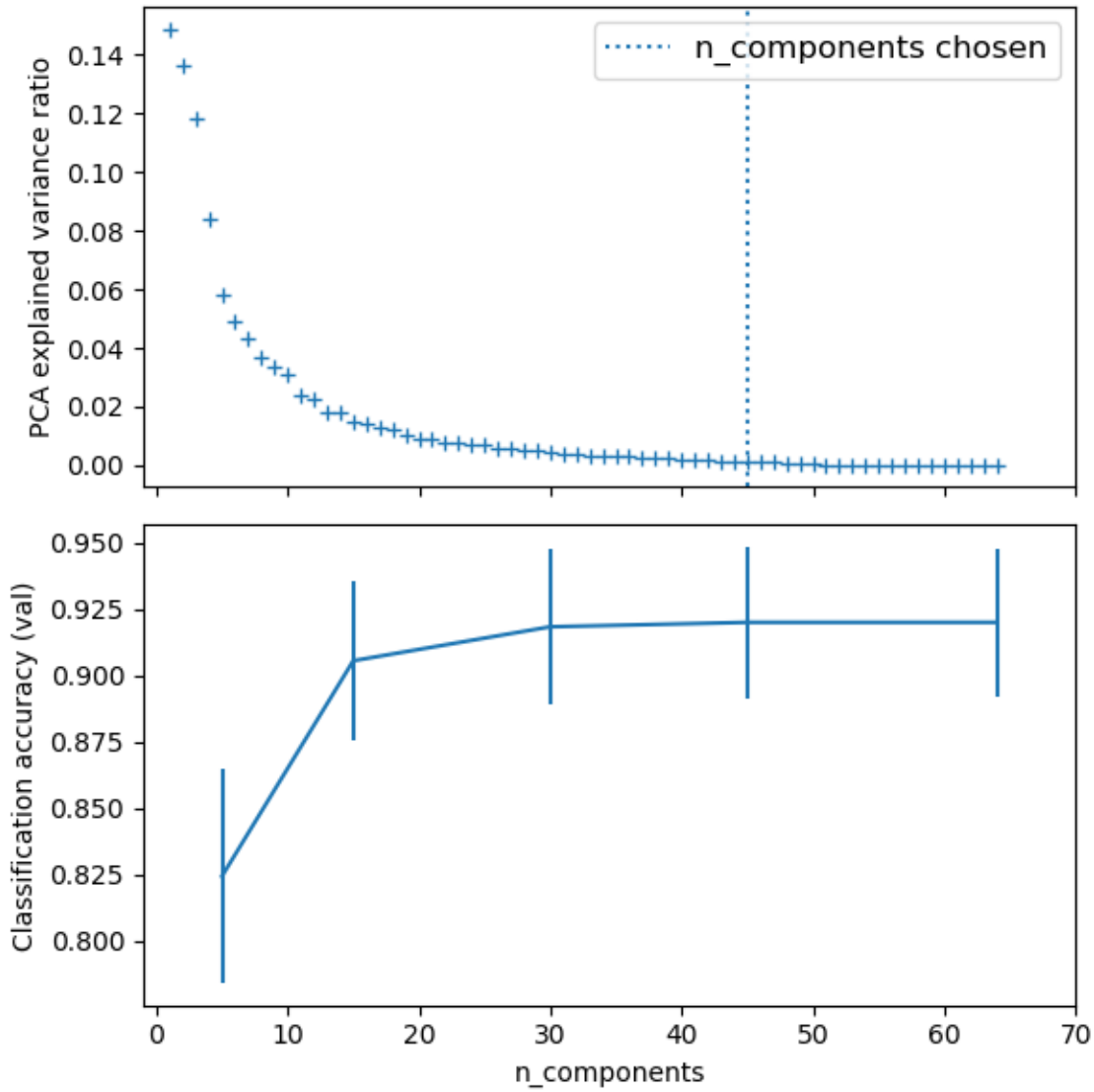
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.24.2 Pipelining: chaining a PCA and a logistic regression

The PCA does an unsupervised dimensionality reduction, while the logistic regression does the prediction.

We use a GridSearchCV to set the dimensionality of the PCA



Out:

```
Best parameter (CV score=0.920):
{'logistic__C': 0.046415888336127774, 'pca__n_components': 45}
```

```
print(__doc__)


# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV


# Define a pipeline to search for the best combination of PCA truncation
# and classifier regularization.
pca = PCA()
# set the tolerance to a large value to make the example faster
logistic = LogisticRegression(max_iter=10000, tol=0.1)
pipe = Pipeline(steps=[('pca', pca), ('logistic', logistic)])

X_digits, y_digits = datasets.load_digits(return_X_y=True)

# Parameters of pipelines can be set using '__' separated parameter names:
param_grid = {
    'pca__n_components': [5, 15, 30, 45, 64],
    'logistic__C': np.logspace(-4, 4, 4),
}
search = GridSearchCV(pipe, param_grid, n_jobs=-1)
search.fit(X_digits, y_digits)
print("Best parameter (CV score=%0.3f):" % search.best_score_)
print(search.best_params_)

# Plot the PCA spectrum
pca.fit(X_digits)

fig, (ax0, ax1) = plt.subplots(nrows=2, sharex=True, figsize=(6, 6))
ax0.plot(np.arange(1, pca.n_components_ + 1),
         pca.explained_variance_ratio_, '+', linewidth=2)
ax0.set_ylabel('PCA explained variance ratio')

ax0.axvline(search.best_estimator_.named_steps['pca'].n_components,
            linestyle=':', label='n_components chosen')
ax0.legend(prop=dict(size=12))

# For each number of components, find the best classifier results
results = pd.DataFrame(search.cv_results_)
components_col = 'param_pca__n_components'
best_clfs = results.groupby(components_col).apply(
    lambda g: g.nlargest(1, 'mean_test_score'))

best_clfs.plot(x=components_col, y='mean_test_score', yerr='std_test_score',
               legend=False, ax=ax1)
ax1.set_ylabel('Classification accuracy (val)')
ax1.set_xlabel('n_components')

plt.xlim(-1, 70)

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 12.707 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.24.3 Column Transformer with Mixed Types

This example illustrates how to apply different preprocessing and feature extraction pipelines to different subsets of features, using *sklearn.compose.ColumnTransformer*. This is particularly handy for the case of datasets that contain heterogeneous data types, since we may want to scale the numeric features and one-hot encode the categorical ones.

In this example, the numeric data is standard-scaled after mean-imputation, while the categorical data is one-hot encoded after imputing missing values with a new category (`'missing'`).

Finally, the preprocessing pipeline is integrated in a full prediction pipeline using *sklearn.pipeline.Pipeline*, together with a simple classification model.

```
# Author: Pedro Morales <part.morales@gmail.com>
#
# License: BSD 3 clause

import numpy as np

from sklearn.compose import ColumnTransformer
from sklearn.datasets import fetch_openml
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV

np.random.seed(0)

# Load data from https://www.openml.org/d/40945
X, y = fetch_openml("titanic", version=1, as_frame=True, return_X_y=True)

# Alternatively X and y can be obtained directly from the frame attribute:
# X = titanic.frame.drop('survived', axis=1)
# y = titanic.frame['survived']

# We will train our classifier with the following features:
# Numeric Features:
# - age: float.
# - fare: float.
# Categorical Features:
# - embarked: categories encoded as strings {'C', 'S', 'Q'}.
# - sex: categories encoded as strings {'female', 'male'}.
# - pclass: ordinal integers {1, 2, 3}.

# We create the preprocessing pipelines for both numeric and categorical data.
numeric_features = ['age', 'fare']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])
```

(continues on next page)

(continued from previous page)

```
categorical_features = ['embarked', 'sex', 'pclass']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', LogisticRegression())])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

clf.fit(X_train, y_train)
print("model score: %.3f" % clf.score(X_test, y_test))
```

Out:

```
model score: 0.790
```

Using the prediction pipeline in a grid search

Grid search can also be performed on the different preprocessing steps defined in the `ColumnTransformer` object, together with the classifier's hyperparameters as part of the `Pipeline`. We will search for both the imputer strategy of the numeric preprocessing and the regularization parameter of the logistic regression using *sklearn.model_selection.GridSearchCV*.

```
param_grid = {
    'preprocessor__num__imputer__strategy': ['mean', 'median'],
    'classifier__C': [0.1, 1.0, 10, 100],
}

grid_search = GridSearchCV(clf, param_grid, cv=10)
grid_search.fit(X_train, y_train)

print(("best logistic regression from grid search: %.3f"
       % grid_search.score(X_test, y_test)))
```

Out:

```
best logistic regression from grid search: 0.798
```

Total running time of the script: (0 minutes 2.795 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.24.4 Selecting dimensionality reduction with Pipeline and GridSearchCV

This example constructs a pipeline that does dimensionality reduction followed by prediction with a support vector classifier. It demonstrates the use of `GridSearchCV` and `Pipeline` to optimize over different classes of estimators in a single CV run – unsupervised `PCA` and `NMF` dimensionality reductions are compared to univariate feature selection during the grid search.

Additionally, `Pipeline` can be instantiated with the `memory` argument to memoize the transformers within the pipeline, avoiding to fit again the same transformers over and over.

Note that the use of `memory` to enable caching becomes interesting when the fitting of a transformer is costly.

Illustration of `Pipeline` and `GridSearchCV`

This section illustrates the use of a `Pipeline` with `GridSearchCV`

```
# Authors: Robert McGibbon, Joel Nothman, Guillaume Lemaitre


import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC
from sklearn.decomposition import PCA, NMF
from sklearn.feature_selection import SelectKBest, chi2

print(__doc__)

pipe = Pipeline([
    # the reduce_dim stage is populated by the param_grid
    ('reduce_dim', 'passthrough'),
    ('classify', LinearSVC(dual=False, max_iter=10000))
])

N_FEATURES_OPTIONS = [2, 4, 8]
C_OPTIONS = [1, 10, 100, 1000]
param_grid = [
    {
        'reduce_dim': [PCA(iterated_power=7), NMF()],
        'reduce_dim__n_components': N_FEATURES_OPTIONS,
        'classify__C': C_OPTIONS
    },
    {
        'reduce_dim': [SelectKBest(chi2)],
        'reduce_dim__k': N_FEATURES_OPTIONS,
        'classify__C': C_OPTIONS
    },
]
reducer_labels = ['PCA', 'NMF', 'KBest(chi2)']

grid = GridSearchCV(pipe, n_jobs=1, param_grid=param_grid)
X, y = load_digits(return_X_y=True)
grid.fit(X, y)

mean_scores = np.array(grid.cv_results_['mean_test_score'])
```

(continues on next page)

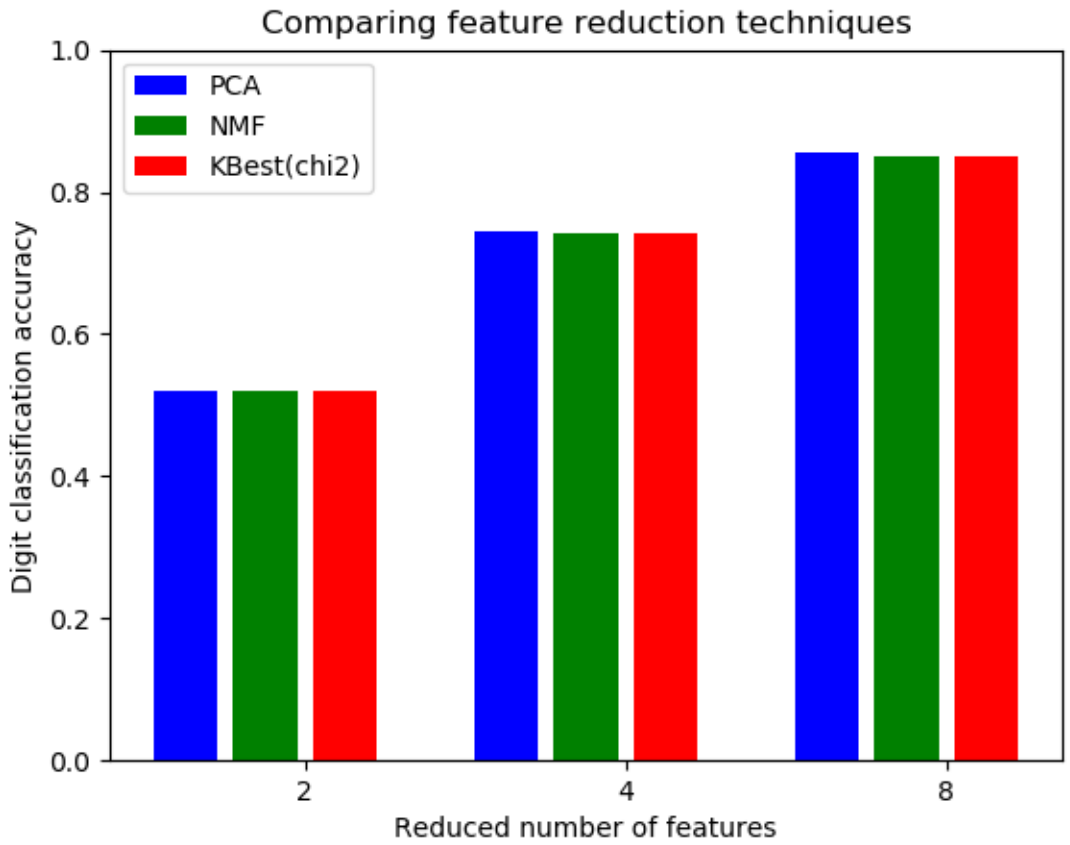
(continued from previous page)

```
# scores are in the order of param_grid iteration, which is alphabetical
mean_scores = mean_scores.reshape(len(C_OPTIONS), -1, len(N_FEATURES_OPTIONS))
# select score for best C
mean_scores = mean_scores.max(axis=0)
bar_offsets = (np.arange(len(N_FEATURES_OPTIONS)) *
               (len(reducer_labels) + 1) + .5)

plt.figure()
COLORS = 'bgrcmyk'
for i, (label, reducer_scores) in enumerate(zip(reducer_labels, mean_scores)):
    plt.bar(bar_offsets + i, reducer_scores, label=label, color=COLORS[i])

plt.title("Comparing feature reduction techniques")
plt.xlabel('Reduced number of features')
plt.xticks(bar_offsets + len(reducer_labels) / 2, N_FEATURES_OPTIONS)
plt.ylabel('Digit classification accuracy')
plt.ylim((0, 1))
plt.legend(loc='upper left')

plt.show()
```



Caching transformers within a `Pipeline`

It is sometimes worthwhile storing the state of a specific transformer since it could be used again. Using a pipeline in `GridSearchCV` triggers such situations. Therefore, we use the argument `memory` to enable caching.

> **Warning:** Note that this example is, however, only an illustration since for this specific case fitting PCA is not necessarily slower than loading the cache. Hence, use the `memory` constructor parameter when the fitting of a transformer is costly.

```
from joblib import Memory
from shutil import rmtree

# Create a temporary folder to store the transformers of the pipeline
location = 'cachedir'
memory = Memory(location=location, verbose=10)
cached_pipe = Pipeline([('reduce_dim', PCA()),
                        ('classify', LinearSVC(dual=False, max_iter=10000))],
                       memory=memory)

# This time, a cached pipeline will be used within the grid search


# Delete the temporary cache before exiting
memory.clear(warn=False)
rmtree(location)
```

The `PCA` fitting is only computed at the evaluation of the first configuration of the `C` parameter of the `LinearSVC` classifier. The other configurations of `C` will trigger the loading of the cached `PCA` estimator data, leading to save processing time. Therefore, the use of caching the pipeline using `memory` is highly beneficial when fitting a transformer is costly.

Total running time of the script: (0 minutes 5.232 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.24.5 Column Transformer with Heterogeneous Data Sources

Datasets can often contain components of that require different feature extraction and processing pipelines. This scenario might occur when:

1. Your dataset consists of heterogeneous data types (e.g. raster images and text captions)
2. Your dataset is stored in a Pandas DataFrame and different columns require different processing pipelines.

This example demonstrates how to use *`sklearn.compose.ColumnTransformer`* on a dataset containing different types of features. We use the 20-newsgroups dataset and compute standard bag-of-words features for the subject line and body in separate pipelines as well as ad hoc features on the body. We combine them (with weights) using a ColumnTransformer and finally train a classifier on the combined set of features.

The choice of features is not particularly helpful, but serves to illustrate the technique.

Out:

```
[Pipeline] ....... (step 1 of 3) Processing subjectbody, total=   0.0s
[Pipeline] ............. (step 2 of 3) Processing union, total=   0.3s
[Pipeline] ............... (step 3 of 3) Processing svc, total=   0.0s
              precision    recall  f1-score   support

           0       0.78      0.71      0.74       319
           1       0.67      0.75      0.71       251

    accuracy                           0.73       570
   macro avg       0.73      0.73      0.72       570
weighted avg       0.73      0.73      0.73       570
```

```
# Author: Matt Terry <matt.terry@gmail.com>
#
# License: BSD 3 clause

import numpy as np

from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.datasets import fetch_20newsgroups
from sklearn.decomposition import TruncatedSVD
from sklearn.feature_extraction import DictVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import classification_report
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.svm import LinearSVC


class TextStats(TransformerMixin, BaseEstimator):
    """Extract features from each document for DictVectorizer"""

    def fit(self, x, y=None):
        return self

    def transform(self, posts):
        return [{'length': len(text),
                 'num_sentences': text.count('.')}
                for text in posts]


class SubjectBodyExtractor(TransformerMixin, BaseEstimator):
    """Extract the subject & body from a usenet post in a single pass.

    Takes a sequence of strings and produces a dict of sequences.  Keys are
    `subject` and `body`.
    """
    def fit(self, x, y=None):
        return self

    def transform(self, posts):
```

(continues on next page)

(continued from previous page)

```
        # construct object dtype array with two columns
        # first column = 'subject' and second column = 'body'
        features = np.empty(shape=(len(posts), 2), dtype=object)
        for i, text in enumerate(posts):
            headers, _, bod = text.partition('\n\n')
            features[i, 1] = bod

            prefix = 'Subject:'
            sub = ''
            for line in headers.split('\n'):
                if line.startswith(prefix):
                    sub = line[len(prefix):]
                    break
            features[i, 0] = sub

        return features


pipeline = Pipeline([
    # Extract the subject & body
    ('subjectbody', SubjectBodyExtractor()),

    # Use ColumnTransformer to combine the features from subject and body
    ('union', ColumnTransformer(
        [
            # Pulling features from the post's subject line (first column)
            ('subject', TfidfVectorizer(min_df=50), 0),

            # Pipeline for standard bag-of-words model for body (second column)
            ('body_bow', Pipeline([
                ('tfidf', TfidfVectorizer()),
                ('best', TruncatedSVD(n_components=50)),
            ]), 1),

            # Pipeline for pulling ad hoc features from post's body
            ('body_stats', Pipeline([
                ('stats', TextStats()),  # returns a list of dicts
                ('vect', DictVectorizer()),  # list of dicts -> feature matrix
            ]), 1),
        ],

        # weight components in ColumnTransformer
        transformer_weights={
            'subject': 0.8,
            'body_bow': 0.5,
            'body_stats': 1.0,
        }
    )),

    # Use a SVC classifier on the combined features
    ('svc', LinearSVC(dual=False)),
], verbose=True)

# limit the list of categories to make running this example faster.
categories = ['alt.atheism', 'talk.religion.misc']
X_train, y_train = fetch_20newsgroups(random_state=1,
                                      subset='train',
```

(continues on next page)

(continued from previous page)

```
                                      categories=categories,
                                      remove=('footers', 'quotes'),
                                      return_X_y=True)
X_test, y_test = fetch_20newsgroups(random_state=1,
                                    subset='test',
                                    categories=categories,
                                    remove=('footers', 'quotes'),
                                    return_X_y=True)

pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)
print(classification_report(y_test, y_pred))
```

Total running time of the script: (0 minutes 2.683 seconds)

Estimated memory usage: 50 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.24.6 Effect of transforming the targets in regression model

In this example, we give an overview of the *sklearn.compose.TransformedTargetRegressor*. Two examples illustrate the benefit of transforming the targets before learning a linear regression model. The first example uses synthetic data while the second example is based on the Boston housing data set.

```
# Author: Guillaume Lemaitre <guillaume.lemaitre@inria.fr>
# License: BSD 3 clause


import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from distutils.version import LooseVersion

print(__doc__)
```

Synthetic example

```
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.linear_model import RidgeCV
from sklearn.compose import TransformedTargetRegressor
from sklearn.metrics import median_absolute_error, r2_score


# `normed` is being deprecated in favor of `density` in histograms
if LooseVersion(matplotlib.__version__) >= '2.1':
    density_param = {'density': True}
else:
    density_param = {'normed': True}
```

A synthetic random regression problem is generated. The targets `y` are modified by: (i) translating all targets such that all entries are non-negative and (ii) applying an exponential function to obtain non-linear targets which cannot be fitted using a simple linear model.

Therefore, a logarithmic (`np.log1p`) and an exponential function (`np.expm1`) will be used to transform the targets before training a linear regression model and using it for prediction.

```
X, y = make_regression(n_samples=10000, noise=100, random_state=0)
y = np.exp((y + abs(y.min())) / 200)
y_trans = np.log1p(y)
```

The following illustrate the probability density functions of the target before and after applying the logarithmic functions.

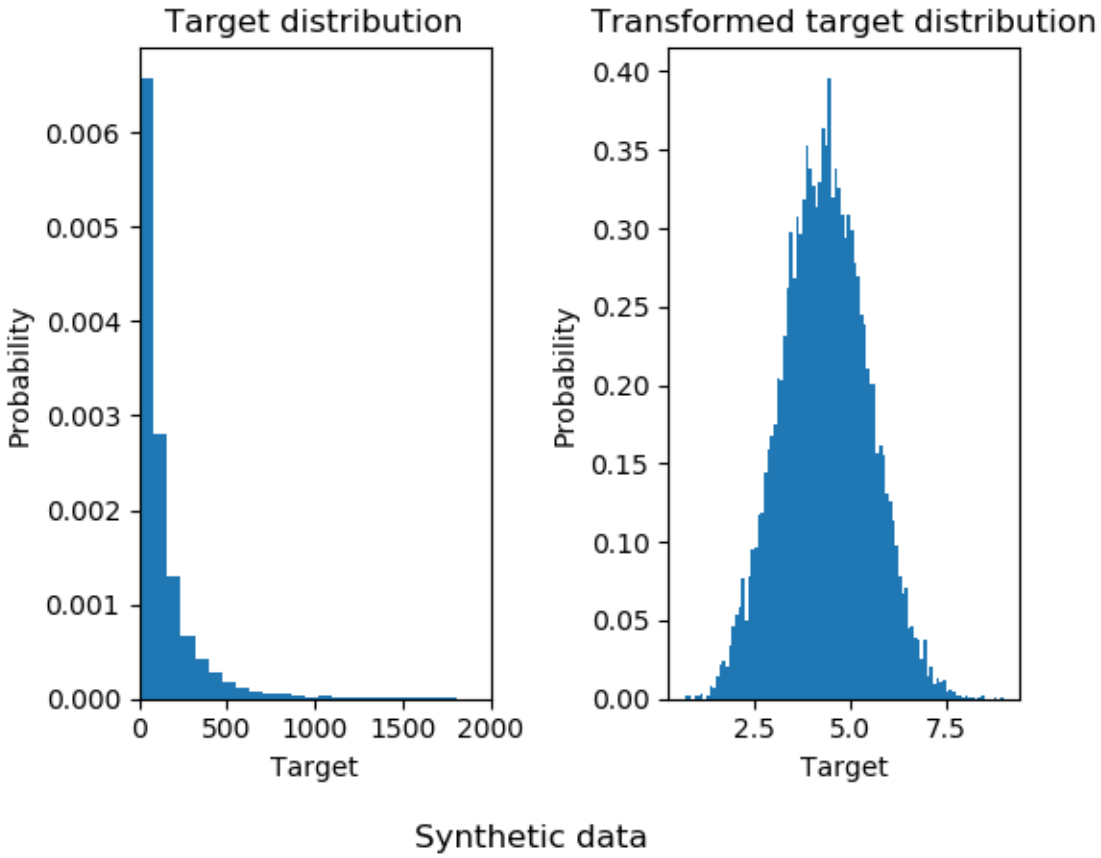
```
f, (ax0, ax1) = plt.subplots(1, 2)

ax0.hist(y, bins=100, **density_param)
ax0.set_xlim([0, 2000])
ax0.set_ylabel('Probability')
ax0.set_xlabel('Target')
ax0.set_title('Target distribution')

ax1.hist(y_trans, bins=100, **density_param)
ax1.set_ylabel('Probability')
ax1.set_xlabel('Target')
ax1.set_title('Transformed target distribution')

f.suptitle("Synthetic data", y=0.035)
f.tight_layout(rect=[0.05, 0.05, 0.95, 0.95])

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```



At first, a linear model will be applied on the original targets. Due to the non-linearity, the model trained will not be precise during the prediction. Subsequently, a logarithmic function is used to linearize the targets, allowing better prediction even with a similar linear model as reported by the median absolute error (MAE).

```
f, (ax0, ax1) = plt.subplots(1, 2, sharey=True)

regr = RidgeCV()
regr.fit(X_train, y_train)
y_pred = regr.predict(X_test)

ax0.scatter(y_test, y_pred)
ax0.plot([0, 2000], [0, 2000], '--k')
ax0.set_ylabel('Target predicted')
ax0.set_xlabel('True Target')
ax0.set_title('Ridge regression \n without target transformation')
ax0.text(100, 1750, r'$R^2$=%.2f, MAE=%.2f' % (
    r2_score(y_test, y_pred), median_absolute_error(y_test, y_pred)))
ax0.set_xlim([0, 2000])
ax0.set_ylim([0, 2000])

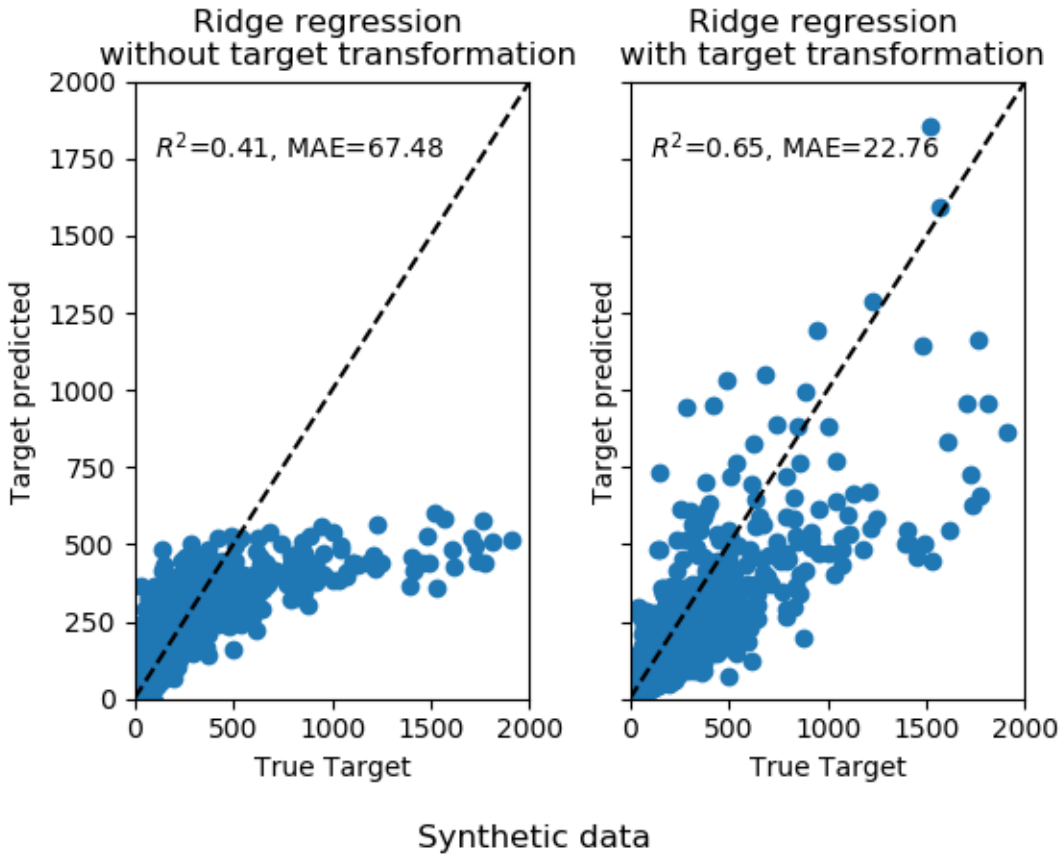
regr_trans = TransformedTargetRegressor(regressor=RidgeCV(),
                                        func=np.log1p,
                                        inverse_func=np.expm1)
regr_trans.fit(X_train, y_train)
y_pred = regr_trans.predict(X_test)
```

(continues on next page)

(continued from previous page)

```
ax1.scatter(y_test, y_pred)
ax1.plot([0, 2000], [0, 2000], '--k')
ax1.set_ylabel('Target predicted')
ax1.set_xlabel('True Target')
ax1.set_title('Ridge regression \n with target transformation')
ax1.text(100, 1750, r'$R^2$=%.2f, MAE=%.2f' % (
    r2_score(y_test, y_pred), median_absolute_error(y_test, y_pred)))
ax1.set_xlim([0, 2000])
ax1.set_ylim([0, 2000])

f.suptitle("Synthetic data", y=0.035)
f.tight_layout(rect=[0.05, 0.05, 0.95, 0.95])
```



Real-world data set

In a similar manner, the boston housing data set is used to show the impact of transforming the targets before learning a model. In this example, the targets to be predicted corresponds to the weighted distances to the five Boston employment centers.

```
from sklearn.datasets import load_boston
from sklearn.preprocessing import QuantileTransformer, quantile_transform

dataset = load_boston()
```

(continues on next page)

(continued from previous page)

```
target = np.array(dataset.feature_names) == "DIS"
X = dataset.data[:, np.logical_not(target)]
y = dataset.data[:, target].squeeze()
y_trans = quantile_transform(dataset.data[:, target],
                             n_quantiles=300,
                             output_distribution='normal',
                             copy=True).squeeze()
```

A *sklearn.preprocessing.QuantileTransformer* is used such that the targets follows a normal distribution before applying a *sklearn.linear_model.RidgeCV* model.

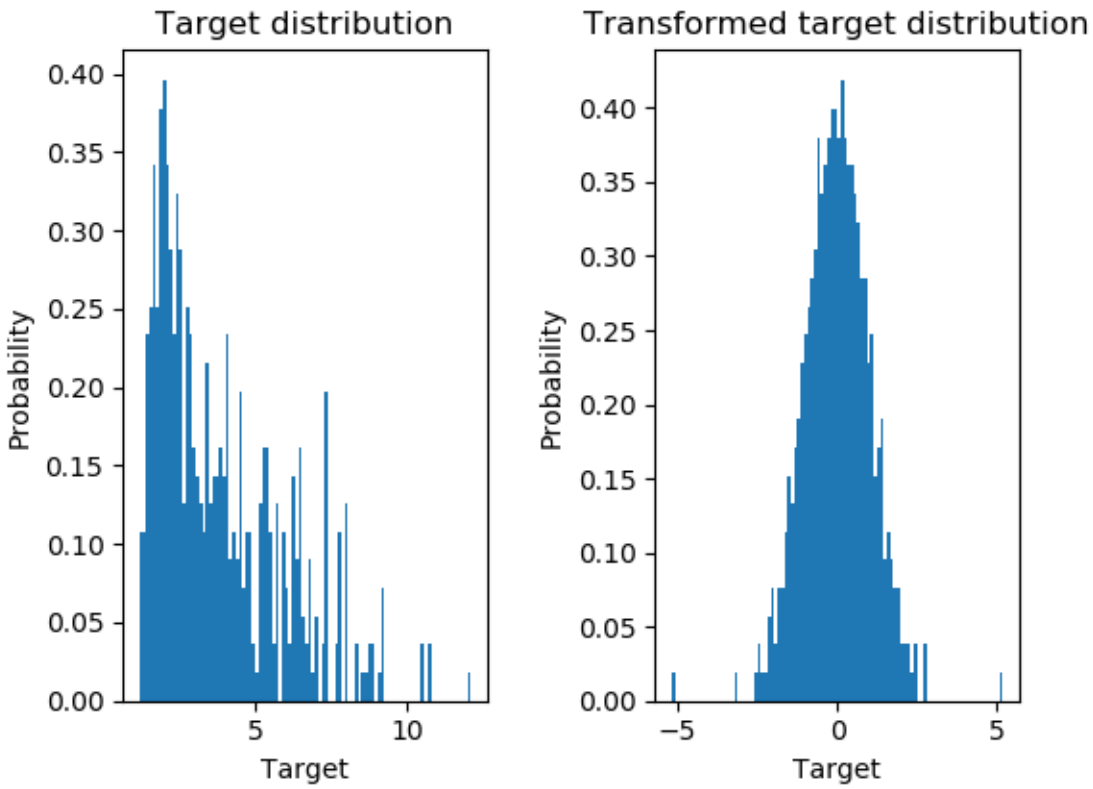
```
f, (ax0, ax1) = plt.subplots(1, 2)

ax0.hist(y, bins=100, **density_param)
ax0.set_ylabel('Probability')
ax0.set_xlabel('Target')
ax0.set_title('Target distribution')

ax1.hist(y_trans, bins=100, **density_param)
ax1.set_ylabel('Probability')
ax1.set_xlabel('Target')
ax1.set_title('Transformed target distribution')

f.suptitle("Boston housing data: distance to employment centers", y=0.035)
f.tight_layout(rect=[0.05, 0.05, 0.95, 0.95])

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
```



The effect of the transformer is weaker than on the synthetic data. However, the transform induces a decrease of the MAE.

```
f, (ax0, ax1) = plt.subplots(1, 2, sharey=True)

regr = RidgeCV()
regr.fit(X_train, y_train)
y_pred = regr.predict(X_test)

ax0.scatter(y_test, y_pred)
ax0.plot([0, 10], [0, 10], '--k')
ax0.set_ylabel('Target predicted')
ax0.set_xlabel('True Target')
ax0.set_title('Ridge regression \n without target transformation')
ax0.text(1, 9, r'$R^2$=%.2f, MAE=%.2f' % (
    r2_score(y_test, y_pred), median_absolute_error(y_test, y_pred)))
ax0.set_xlim([0, 10])
ax0.set_ylim([0, 10])

regr_trans = TransformedTargetRegressor(
    regressor=RidgeCV(),
    transformer=QuantileTransformer(n_quantiles=300,
                                    output_distribution='normal'))
regr_trans.fit(X_train, y_train)
y_pred = regr_trans.predict(X_test)
```

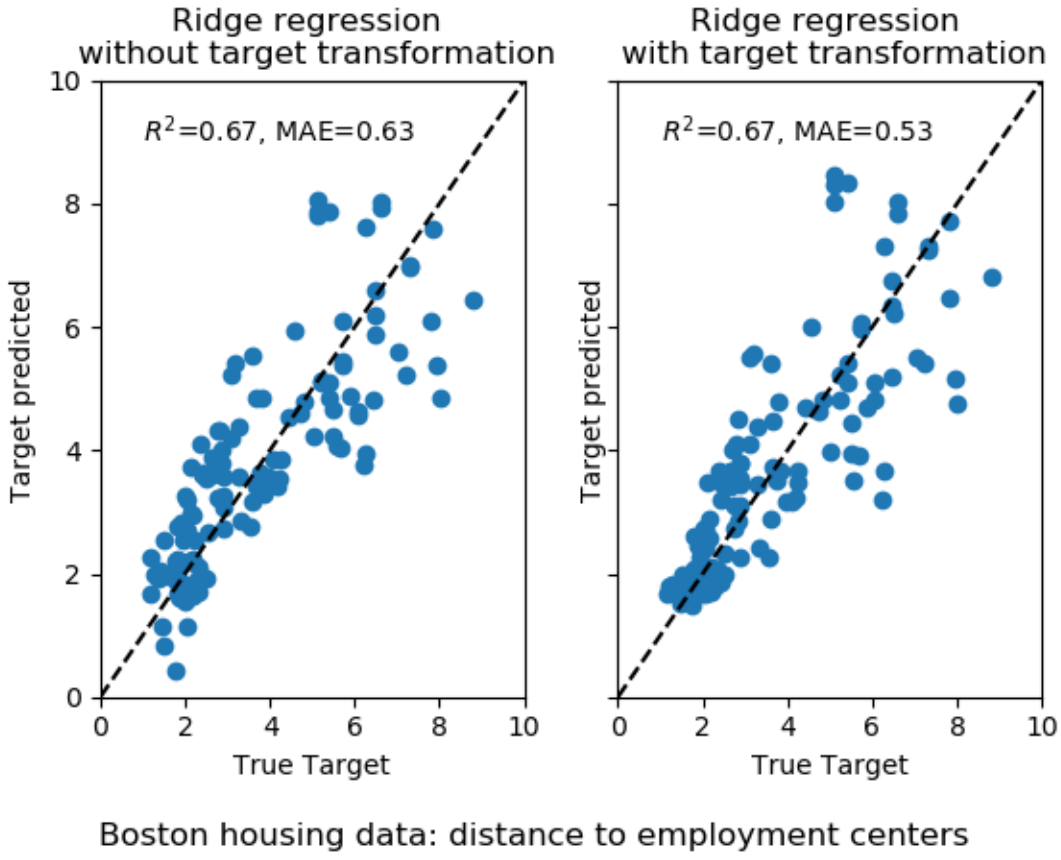
(continues on next page)

(continued from previous page)

```
ax1.scatter(y_test, y_pred)
ax1.plot([0, 10], [0, 10], '--k')
ax1.set_ylabel('Target predicted')
ax1.set_xlabel('True Target')
ax1.set_title('Ridge regression \n with target transformation')
ax1.text(1, 9, r'$R^2$=%.2f, MAE=%.2f' % (
    r2_score(y_test, y_pred), median_absolute_error(y_test, y_pred)))
ax1.set_xlim([0, 10])
ax1.set_ylim([0, 10])

f.suptitle("Boston housing data: distance to employment centers", y=0.035)
f.tight_layout(rect=[0.05, 0.05, 0.95, 0.95])

plt.show()
```



Total running time of the script: (0 minutes 3.346 seconds)

Estimated memory usage: 10 MB

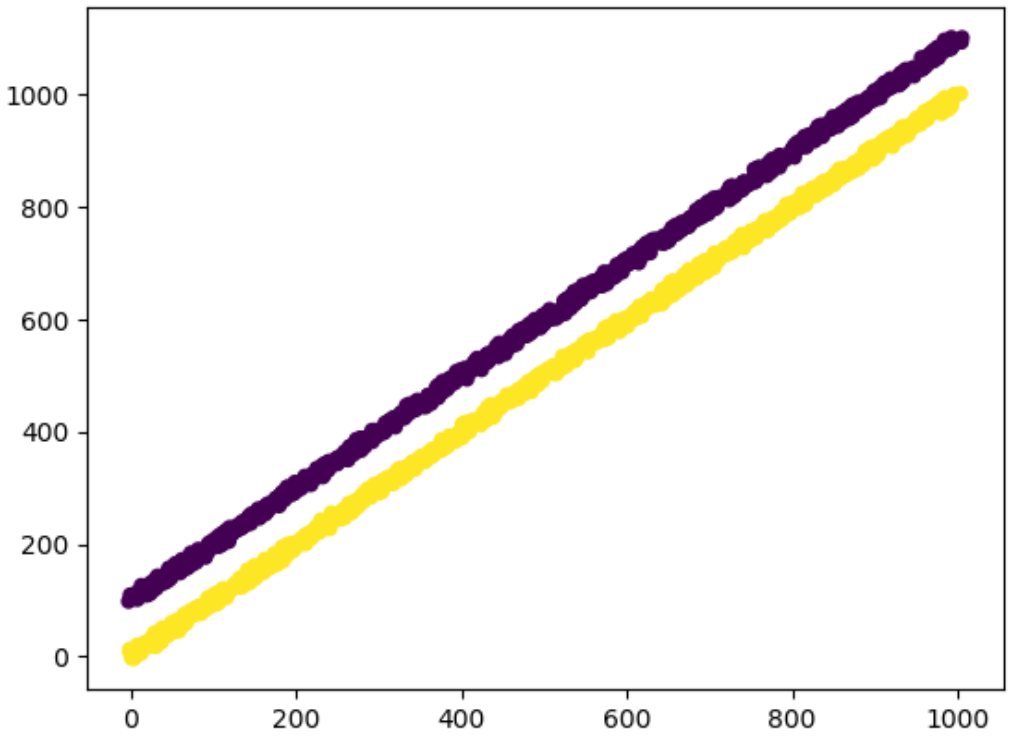
6.25 Preprocessing

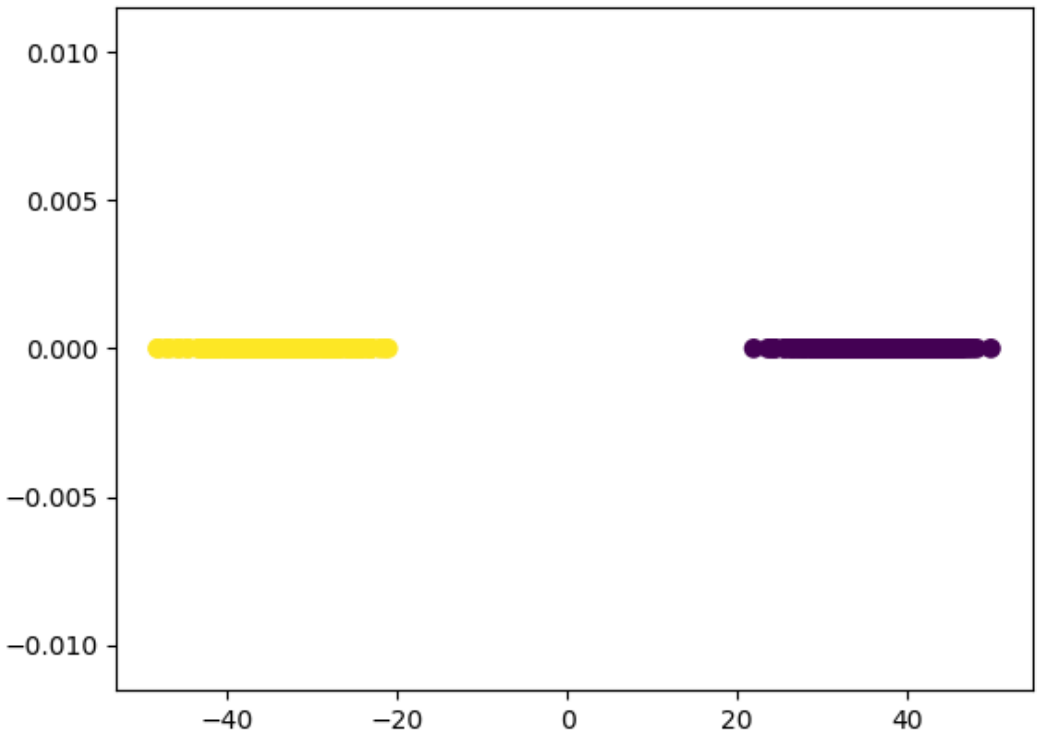
Examples concerning the *sklearn.preprocessing* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.25.1 Using FunctionTransformer to select columns

Shows how to use a function transformer in a pipeline. If you know your dataset's first principle component is irrelevant for a classification task, you can use the FunctionTransformer to select all but the first column of the PCA transformed data.





-

```
import matplotlib.pyplot as plt
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import FunctionTransformer


def _generate_vector(shift=0.5, noise=15):
    return np.arange(1000) + (np.random.rand(1000) - shift) * noise


def generate_dataset():
    """
    This dataset is two lines with a slope ~ 1, where one has
    a y offset of ~100
    """
    return np.vstack((
        np.vstack((
            _generate_vector(),
            _generate_vector() + 100,
        )).T,
        np.vstack((
            _generate_vector(),
            _generate_vector(),
        )).T,
```

(continues on next page)

(continued from previous page)

```
    )), np.hstack((np.zeros(1000), np.ones(1000)))


def all_but_first_column(X):
    return X[:, 1:]


def drop_first_component(X, y):
    """
    Create a pipeline with PCA and the column selector and use it to
    transform the dataset.
    """
    pipeline = make_pipeline(
        PCA(), FunctionTransformer(all_but_first_column),
    )
    X_train, X_test, y_train, y_test = train_test_split(X, y)
    pipeline.fit(X_train, y_train)
    return pipeline.transform(X_test), y_test


if __name__ == '__main__':
    X, y = generate_dataset()
    lw = 0
    plt.figure()
    plt.scatter(X[:, 0], X[:, 1], c=y, lw=lw)
    plt.figure()
    X_transformed, y_transformed = drop_first_component(*generate_dataset())
    plt.scatter(
        X_transformed[:, 0],
        np.zeros(len(X_transformed)),
        c=y_transformed,
        lw=lw,
        s=60
    )
    plt.show()
```

Total running time of the script: (0 minutes 0.397 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

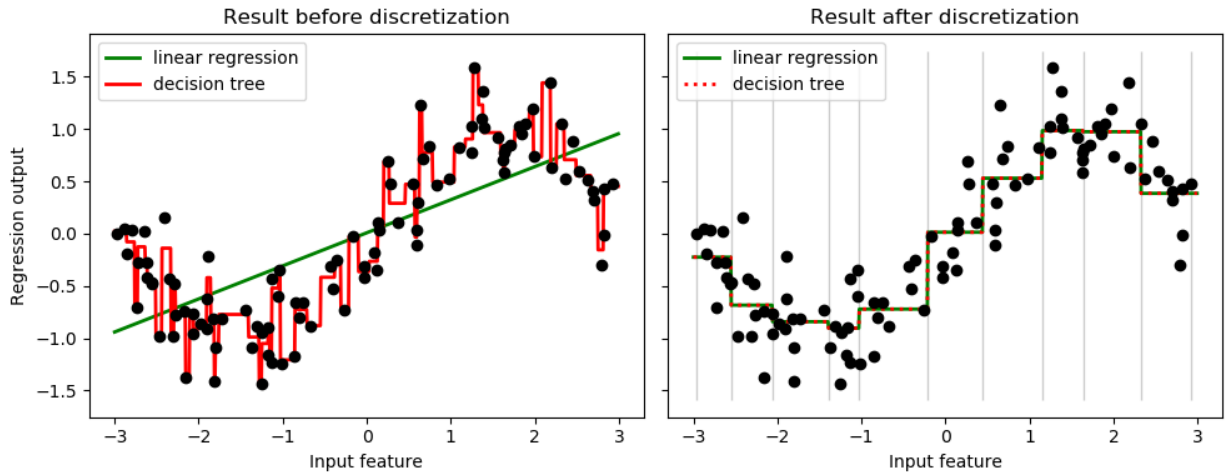
6.25.2 Using KBinsDiscretizer to discretize continuous features

The example compares prediction result of linear regression (linear model) and decision tree (tree based model) with and without discretization of real-valued features.

As is shown in the result before discretization, linear model is fast to build and relatively straightforward to interpret, but can only model linear relationships, while decision tree can build a much more complex model of the data. One way to make linear model more powerful on continuous data is to use discretization (also known as binning). In the example, we discretize the feature and one-hot encode the transformed data. Note that if the bins are not reasonably wide, there would appear to be a substantially increased risk of overfitting, so the discretizer parameters should usually be tuned under cross validation.

After discretization, linear regression and decision tree make exactly the same prediction. As features are constant

within each bin, any model must predict the same value for all points within a bin. Compared with the result before discretization, linear model become much more flexible while decision tree gets much less flexible. Note that binning features generally has no beneficial effect for tree-based models, as these models can learn to split up the data anywhere.




```
# Author: Andreas Müller
#         Hanmin Qin <qinhanmin2005@sina.com>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.tree import DecisionTreeRegressor

print(__doc__)

# construct the dataset
rnd = np.random.RandomState(42)
X = rnd.uniform(-3, 3, size=100)
y = np.sin(X) + rnd.normal(size=len(X)) / 3
X = X.reshape(-1, 1)

# transform the dataset with KBinsDiscretizer
enc = KBinsDiscretizer(n_bins=10, encode='onehot')
X_binned = enc.fit_transform(X)

# predict with original dataset
fig, (ax1, ax2) = plt.subplots(ncols=2, sharey=True, figsize=(10, 4))
line = np.linspace(-3, 3, 1000, endpoint=False).reshape(-1, 1)
reg = LinearRegression().fit(X, y)
ax1.plot(line, reg.predict(line), linewidth=2, color='green',
         label="linear regression")
reg = DecisionTreeRegressor(min_samples_split=3, random_state=0).fit(X, y)
ax1.plot(line, reg.predict(line), linewidth=2, color='red',
         label="decision tree")
ax1.plot(X[:, 0], y, 'o', c='k')
ax1.legend(loc="best")
ax1.set_ylabel("Regression output")
```

(continues on next page)

(continued from previous page)

```
ax1.set_xlabel("Input feature")
ax1.set_title("Result before discretization")

# predict with transformed dataset
line_binned = enc.transform(line)
reg = LinearRegression().fit(X_binned, y)
ax2.plot(line, reg.predict(line_binned), linewidth=2, color='green',
         linestyle='-', label='linear regression')
reg = DecisionTreeRegressor(min_samples_split=3,
                            random_state=0).fit(X_binned, y)
ax2.plot(line, reg.predict(line_binned), linewidth=2, color='red',
         linestyle=':', label='decision tree')
ax2.plot(X[:, 0], y, 'o', c='k')
ax2.vlines(enc.bin_edges_[0], *plt.gca().get_ylim(), linewidth=1, alpha=.2)
ax2.legend(loc="best")
ax2.set_xlabel("Input feature")
ax2.set_title("Result after discretization")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.614 seconds)

Estimated memory usage: 8 MB

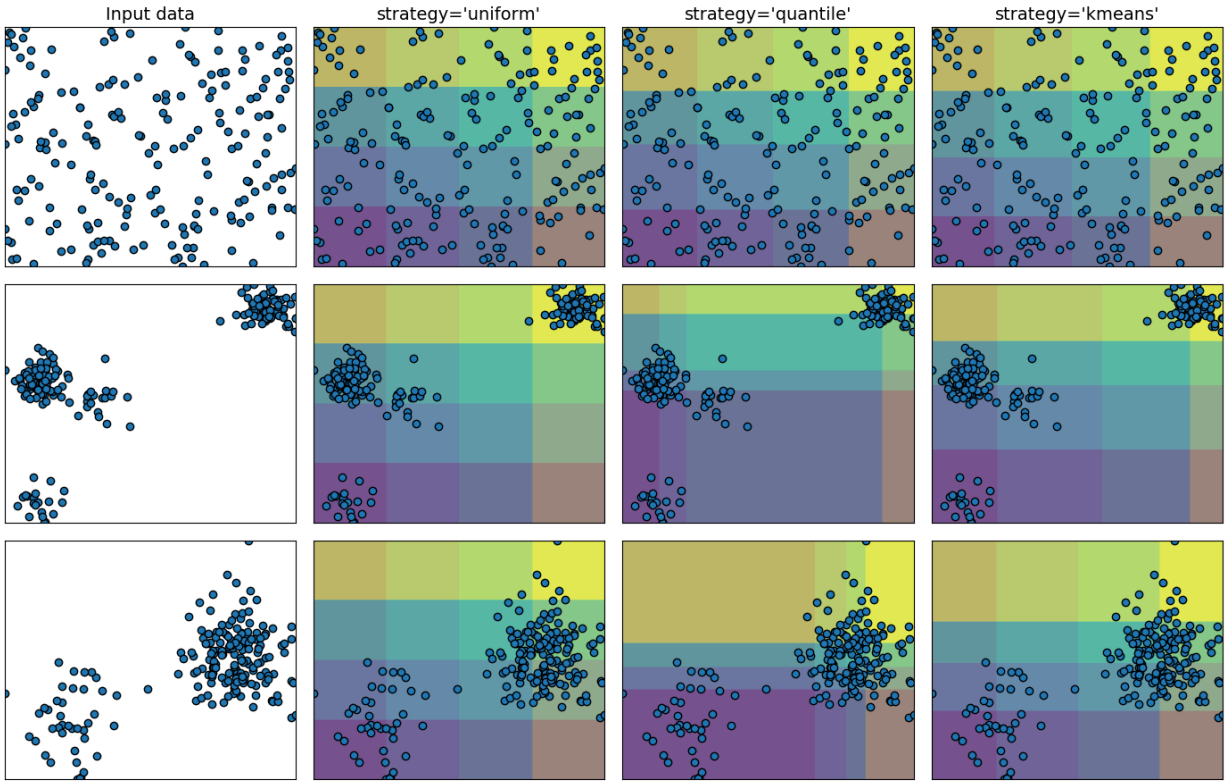
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.25.3 Demonstrating the different strategies of KBinsDiscretizer

This example presents the different strategies implemented in KBinsDiscretizer:

- 'uniform': The discretization is uniform in each feature, which means that the bin widths are constant in each dimension.
- quantile': The discretization is done on the quantiled values, which means that each bin has approximately the same number of samples.
- 'kmeans': The discretization is based on the centroids of a KMeans clustering procedure.

The plot shows the regions where the discretized encoding is constant.




```
# Author: Tom Dupré la Tour
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.preprocessing import KBinsDiscretizer
from sklearn.datasets import make_blobs

print(__doc__)

strategies = ['uniform', 'quantile', 'kmeans']

n_samples = 200
centers_0 = np.array([[0, 0], [0, 5], [2, 4], [8, 8]])
centers_1 = np.array([[0, 0], [3, 1]])

# construct the datasets
random_state = 42
X_list = [
    np.random.RandomState(random_state).uniform(-3, 3, size=(n_samples, 2)),
    make_blobs(n_samples=[n_samples // 10, n_samples * 4 // 10,
                          n_samples // 10, n_samples * 4 // 10],
               cluster_std=0.5, centers=centers_0,
               random_state=random_state)[0],
    make_blobs(n_samples=[n_samples // 5, n_samples * 4 // 5],
               cluster_std=0.5, centers=centers_1,
               random_state=random_state)[0],
]
```

(continues on next page)

(continued from previous page)

```
figure = plt.figure(figsize=(14, 9))
i = 1
for ds_cnt, X in enumerate(X_list):

    ax = plt.subplot(len(X_list), len(strategies) + 1, i)
    ax.scatter(X[:, 0], X[:, 1], edgecolors='k')
    if ds_cnt == 0:
        ax.set_title("Input data", size=14)

    xx, yy = np.meshgrid(
        np.linspace(X[:, 0].min(), X[:, 0].max(), 300),
        np.linspace(X[:, 1].min(), X[:, 1].max(), 300))
    grid = np.c_[xx.ravel(), yy.ravel()]

    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())

    i += 1
    # transform the dataset with KBinsDiscretizer
    for strategy in strategies:
        enc = KBinsDiscretizer(n_bins=4, encode='ordinal', strategy=strategy)
        enc.fit(X)
        grid_encoded = enc.transform(grid)

        ax = plt.subplot(len(X_list), len(strategies) + 1, i)

        # horizontal stripes
        horizontal = grid_encoded[:, 0].reshape(xx.shape)
        ax.contourf(xx, yy, horizontal, alpha=.5)
        # vertical stripes
        vertical = grid_encoded[:, 1].reshape(xx.shape)
        ax.contourf(xx, yy, vertical, alpha=.5)

        ax.scatter(X[:, 0], X[:, 1], edgecolors='k')
        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
        ax.set_xticks(())
        ax.set_yticks(())
        if ds_cnt == 0:
            ax.set_title("strategy='%s'" % (strategy, ), size=14)

        i += 1

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.939 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.25.4 Importance of Feature Scaling

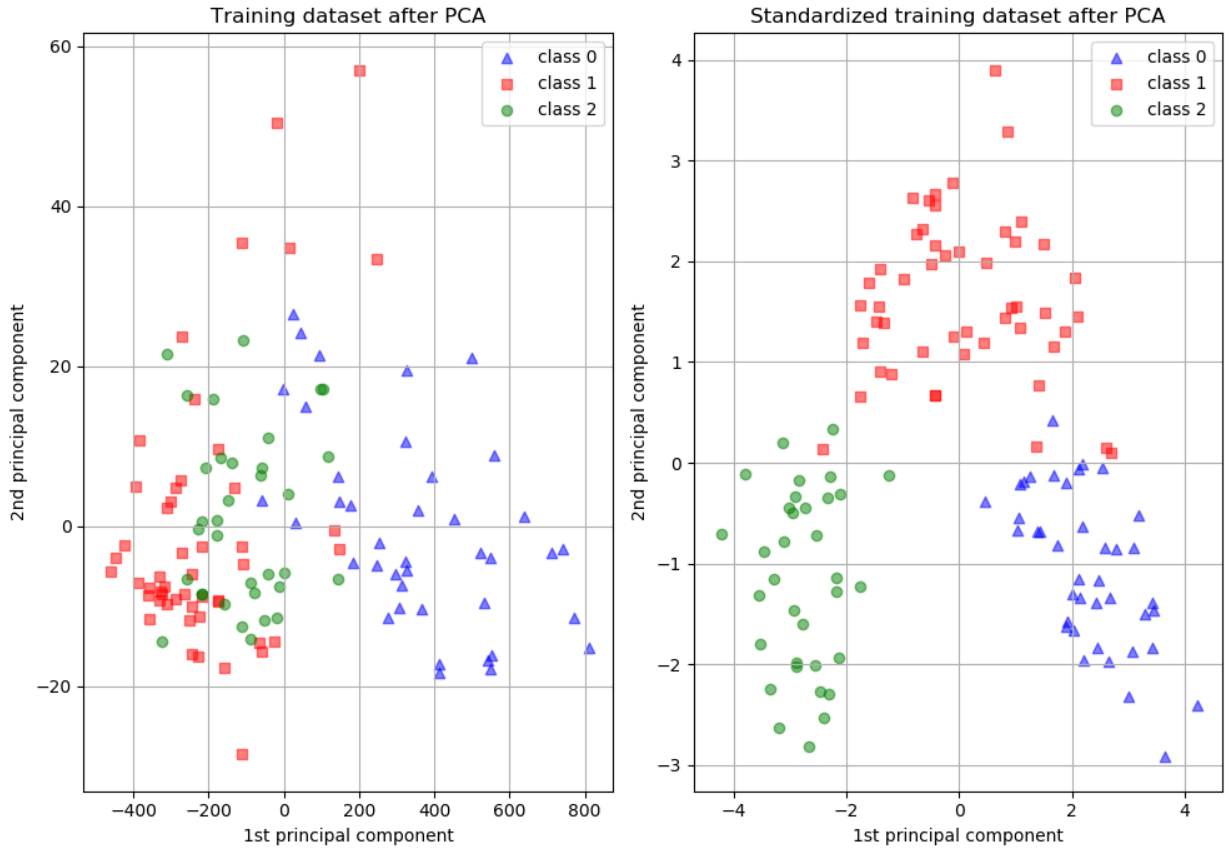
Feature scaling through standardization (or Z-score normalization) can be an important preprocessing step for many machine learning algorithms. Standardization involves rescaling the features such that they have the properties of a standard normal distribution with a mean of zero and a standard deviation of one.

While many algorithms (such as SVM, K-nearest neighbors, and logistic regression) require features to be normalized, intuitively we can think of Principle Component Analysis (PCA) as being a prime example of when normalization is important. In PCA we are interested in the components that maximize the variance. If one component (e.g. human height) varies less than another (e.g. weight) because of their respective scales (meters vs. kilos), PCA might determine that the direction of maximal variance more closely corresponds with the 'weight' axis, if those features are not scaled. As a change in height of one meter can be considered much more important than the change in weight of one kilogram, this is clearly incorrect.

To illustrate this, PCA is performed comparing the use of data with `StandardScaler` applied, to unscaled data. The results are visualized and a clear difference noted. The 1st principal component in the unscaled set can be seen. It can be seen that feature #13 dominates the direction, being a whole two orders of magnitude above the other features. This is contrasted when observing the principal component for the scaled version of the data. In the scaled version, the orders of magnitude are roughly the same across all the features.

The dataset used is the Wine Dataset available at UCI. This dataset has continuous features that are heterogeneous in scale due to differing properties that they measure (i.e alcohol content, and malic acid).

The transformed data is then used to train a naive Bayes classifier, and a clear difference in prediction accuracies is observed wherein the dataset which is scaled before PCA vastly outperforms the unscaled version.



Out:

```
Prediction accuracy for the normal test dataset with PCA
81.48%


Prediction accuracy for the standardized test dataset with PCA
98.15%


PC 1 without scaling:
 [ 1.76e-03 -8.36e-04  1.55e-04 -5.31e-03  2.02e-02  1.02e-03  1.53e-03
 -1.12e-04  6.31e-04  2.33e-03  1.54e-04  7.43e-04  1.00e+00]

PC 1 with scaling:
 [ 0.13 -0.26 -0.01 -0.23  0.16  0.39  0.42 -0.28  0.33 -0.11  0.3   0.38
  0.28]
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.naive_bayes import GaussianNB
from sklearn import metrics
import matplotlib.pyplot as plt
from sklearn.datasets import load_wine
from sklearn.pipeline import make_pipeline
print(__doc__)

# Code source: Tyler Lanigan <tylerlanigan@gmail.com>
#              Sebastian Raschka <mail@sebastianraschka.com>

# License: BSD 3 clause

RANDOM_STATE = 42
FIG_SIZE = (10, 7)


features, target = load_wine(return_X_y=True)

# Make a train/test split using 30% test size
X_train, X_test, y_train, y_test = train_test_split(features, target,
                                                    test_size=0.30,
                                                    random_state=RANDOM_STATE)

# Fit to data and predict using pipelined GNB and PCA.
unscaled_clf = make_pipeline(PCA(n_components=2), GaussianNB())
unscaled_clf.fit(X_train, y_train)
pred_test = unscaled_clf.predict(X_test)

# Fit to data and predict using pipelined scaling, GNB and PCA.
std_clf = make_pipeline(StandardScaler(), PCA(n_components=2), GaussianNB())
std_clf.fit(X_train, y_train)
pred_test_std = std_clf.predict(X_test)
```

(continues on next page)

(continued from previous page)

```
# Show prediction accuracies in scaled and unscaled data.
print('\nPrediction accuracy for the normal test dataset with PCA')
print('{:.2%}\n'.format(metrics.accuracy_score(y_test, pred_test)))

print('\nPrediction accuracy for the standardized test dataset with PCA')
print('{:.2%}\n'.format(metrics.accuracy_score(y_test, pred_test_std)))

# Extract PCA from pipeline
pca = unscaled_clf.named_steps['pca']
pca_std = std_clf.named_steps['pca']

# Show first principal components
print('\nPC 1 without scaling:\n', pca.components_[0])
print('\nPC 1 with scaling:\n', pca_std.components_[0])

# Use PCA without and with scale on X_train data for visualization.
X_train_transformed = pca.transform(X_train)
scaler = std_clf.named_steps['standardscaler']
X_train_std_transformed = pca_std.transform(scaler.transform(X_train))

# visualize standardized vs. untouched dataset with PCA performed
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=FIG_SIZE)


for l, c, m in zip(range(0, 3), ('blue', 'red', 'green'), ('^', 's', 'o')):
    ax1.scatter(X_train_transformed[y_train == l, 0],
                X_train_transformed[y_train == l, 1],
                color=c,
                label='class %s' % l,
                alpha=0.5,
                marker=m
                )

for l, c, m in zip(range(0, 3), ('blue', 'red', 'green'), ('^', 's', 'o')):
    ax2.scatter(X_train_std_transformed[y_train == l, 0],
                X_train_std_transformed[y_train == l, 1],
                color=c,
                label='class %s' % l,
                alpha=0.5,
                marker=m
                )

ax1.set_title('Training dataset after PCA')
ax2.set_title('Standardized training dataset after PCA')

for ax in (ax1, ax2):
    ax.set_xlabel('1st principal component')
    ax.set_ylabel('2nd principal component')
    ax.legend(loc='upper right')
    ax.grid()

plt.tight_layout()

plt.show()
```

Total running time of the script: (0 minutes 0.669 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.25.5 Map data to a normal distribution

This example demonstrates the use of the Box-Cox and Yeo-Johnson transforms through `PowerTransformer` to map data from various distributions to a normal distribution.

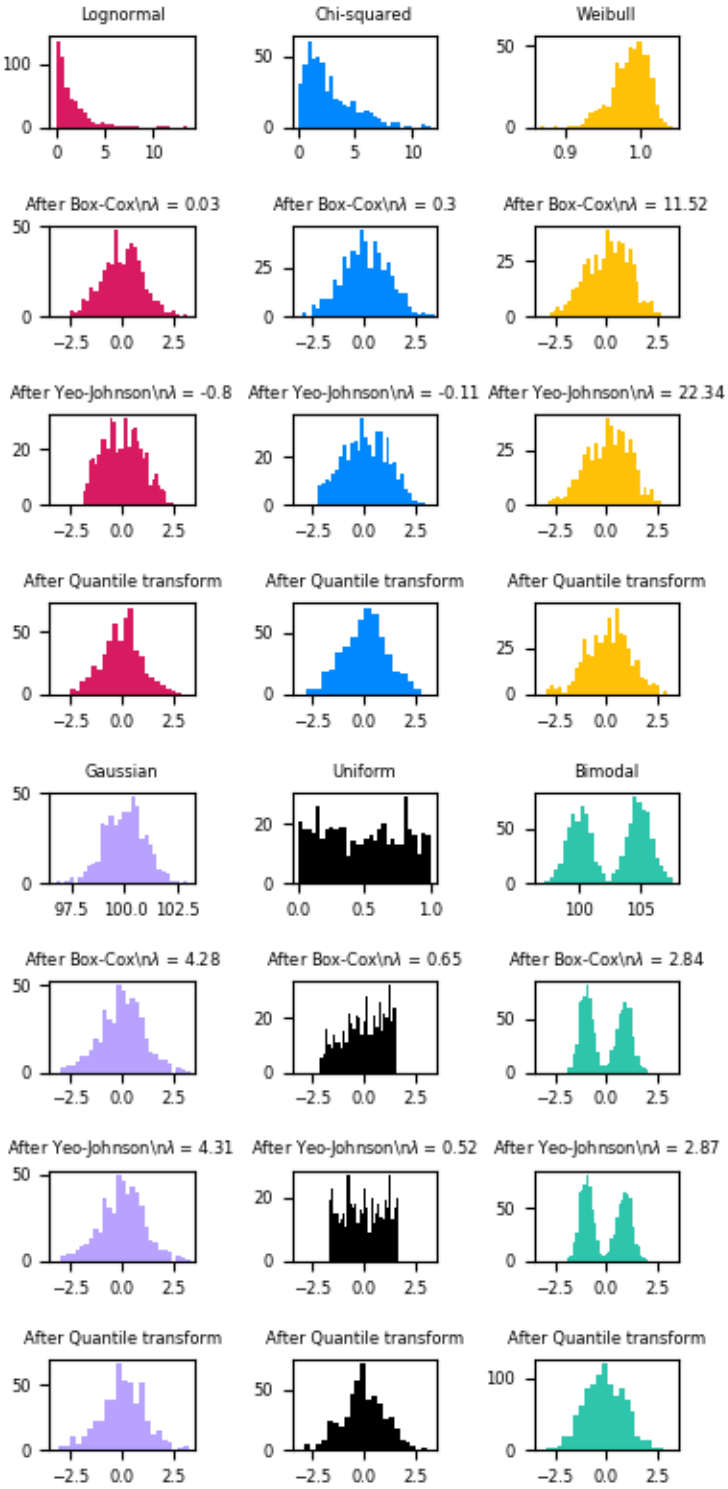
The power transform is useful as a transformation in modeling problems where homoscedasticity and normality are desired. Below are examples of Box-Cox and Yeo-Johnwon applied to six different probability distributions: Lognormal, Chi-squared, Weibull, Gaussian, Uniform, and Bimodal.

Note that the transformations successfully map the data to a normal distribution when applied to certain datasets, but are ineffective with others. This highlights the importance of visualizing the data before and after transformation.

Also note that even though Box-Cox seems to perform better than Yeo-Johnson for lognormal and chi-squared distributions, keep in mind that Box-Cox does not support inputs with negative values.

For comparison, we also add the output from `QuantileTransformer`. It can force any arbitrary distribution into a gaussian, provided that there are enough training samples (thousands). Because it is a non-parametric method, it is harder to interpret than the parametric ones (Box-Cox and Yeo-Johnson).

On "small" datasets (less than a few hundred points), the quantile transformer is prone to overfitting. The use of the power transform is then recommended.




```
# Author: Eric Chang <ericchang2017@u.northwestern.edu>
#         Nicolas Hug <contact@nicolas-hug.com>
# License: BSD 3 clause
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import QuantileTransformer
from sklearn.model_selection import train_test_split

print(__doc__)


N_SAMPLES = 1000
FONT_SIZE = 6
BINS = 30


rng = np.random.RandomState(304)
bc = PowerTransformer(method='box-cox')
yj = PowerTransformer(method='yeo-johnson')
# n_quantiles is set to the training set size rather than the default value
# to avoid a warning being raised by this example
qt = QuantileTransformer(n_quantiles=500, output_distribution='normal',
                         random_state=rng)
size = (N_SAMPLES, 1)


# lognormal distribution
X_lognormal = rng.lognormal(size=size)

# chi-squared distribution
df = 3
X_chisq = rng.chisquare(df=df, size=size)

# weibull distribution
a = 50
X_weibull = rng.weibull(a=a, size=size)

# gaussian distribution
loc = 100
X_gaussian = rng.normal(loc=loc, size=size)

# uniform distribution
X_uniform = rng.uniform(low=0, high=1, size=size)

# bimodal distribution
loc_a, loc_b = 100, 105
X_a, X_b = rng.normal(loc=loc_a, size=size), rng.normal(loc=loc_b, size=size)
X_bimodal = np.concatenate([X_a, X_b], axis=0)


# create plots
distributions = [
    ('Lognormal', X_lognormal),
    ('Chi-squared', X_chisq),
    ('Weibull', X_weibull),
    ('Gaussian', X_gaussian),
    ('Uniform', X_uniform),
    ('Bimodal', X_bimodal)
```

(continues on next page)

(continued from previous page)

```
]

colors = ['#D81B60', '#0188FF', '#FFC107',
          '#B7A2FF', '#000000', '#2EC5AC']

fig, axes = plt.subplots(nrows=8, ncols=3, figsize=plt.figaspect(2))
axes = axes.flatten()
axes_idxs = [(0, 3, 6, 9), (1, 4, 7, 10), (2, 5, 8, 11), (12, 15, 18, 21),
             (13, 16, 19, 22), (14, 17, 20, 23)]
axes_list = [(axes[i], axes[j], axes[k], axes[l])
             for (i, j, k, l) in axes_idxs]


for distribution, color, axes in zip(distributions, colors, axes_list):
    name, X = distribution
    X_train, X_test = train_test_split(X, test_size=.5)

    # perform power transforms and quantile transform
    X_trans_bc = bc.fit(X_train).transform(X_test)
    lmbda_bc = round(bc.lambdas_[0], 2)
    X_trans_yj = yj.fit(X_train).transform(X_test)
    lmbda_yj = round(yj.lambdas_[0], 2)
    X_trans_qt = qt.fit(X_train).transform(X_test)

    ax_original, ax_bc, ax_yj, ax_qt = axes

    ax_original.hist(X_train, color=color, bins=BINS)
    ax_original.set_title(name, fontsize=FONT_SIZE)
    ax_original.tick_params(axis='both', which='major', labelsize=FONT_SIZE)

    for ax, X_trans, meth_name, lmbda in zip(
            (ax_bc, ax_yj, ax_qt),
            (X_trans_bc, X_trans_yj, X_trans_qt),
            ('Box-Cox', 'Yeo-Johnson', 'Quantile transform'),
            (lmbda_bc, lmbda_yj, None)):
        ax.hist(X_trans, color=color, bins=BINS)
        title = 'After {}'.format(meth_name)
        if lmbda is not None:
            title += r'\n$\lambda$ = {}'.format(lmbda)
        ax.set_title(title, fontsize=FONT_SIZE)
        ax.tick_params(axis='both', which='major', labelsize=FONT_SIZE)
        ax.set_xlim([-3.5, 3.5])


plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 1.839 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

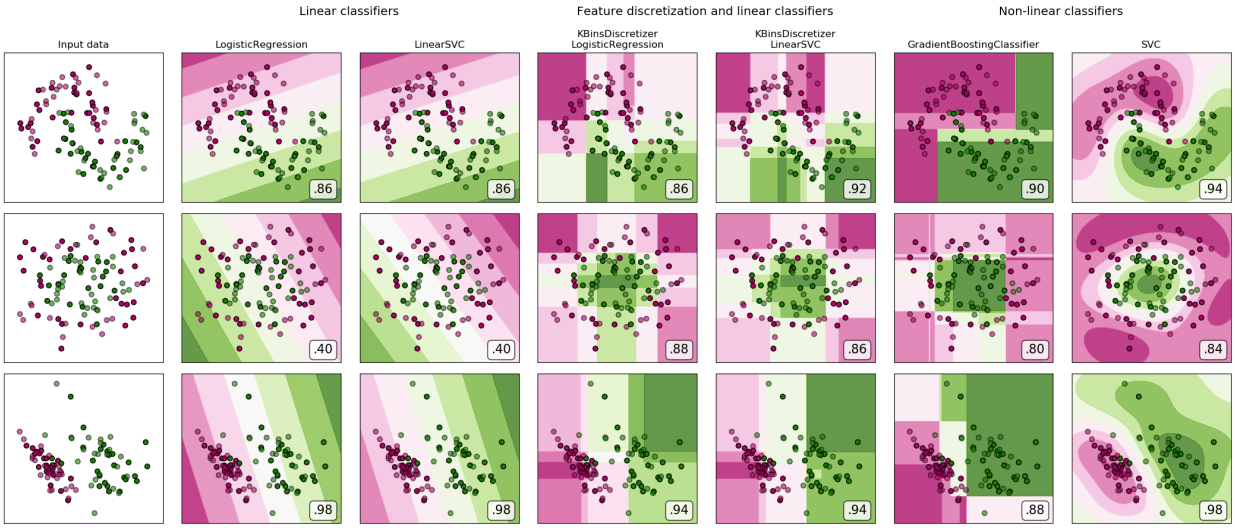
6.25.6 Feature discretization

A demonstration of feature discretization on synthetic classification datasets. Feature discretization decomposes each feature into a set of bins, here equally distributed in width. The discrete values are then one-hot encoded, and given to a linear classifier. This preprocessing enables a non-linear behavior even though the classifier is linear.

On this example, the first two rows represent linearly non-separable datasets (moons and concentric circles) while the third is approximately linearly separable. On the two linearly non-separable datasets, feature discretization largely increases the performance of linear classifiers. On the linearly separable dataset, feature discretization decreases the performance of linear classifiers. Two non-linear classifiers are also shown for comparison.

This example should be taken with a grain of salt, as the intuition conveyed does not necessarily carry over to real datasets. Particularly in high-dimensional spaces, data can more easily be separated linearly. Moreover, using feature discretization and one-hot encoding increases the number of features, which easily lead to overfitting when the number of samples is small.

The plots show training points in solid colors and testing points semi-transparent. The lower right shows the classification accuracy on the test set.



Out:

```
dataset 0
---------
LogisticRegression: 0.86
LinearSVC: 0.86
KBinsDiscretizer + LogisticRegression: 0.86
KBinsDiscretizer + LinearSVC: 0.92
GradientBoostingClassifier: 0.90
SVC: 0.94

dataset 1
---------
LogisticRegression: 0.40
LinearSVC: 0.40
KBinsDiscretizer + LogisticRegression: 0.88
KBinsDiscretizer + LinearSVC: 0.86
GradientBoostingClassifier: 0.80
SVC: 0.84
```

(continues on next page)

(continued from previous page)

```
dataset 2
---------
LogisticRegression: 0.98
LinearSVC: 0.98
KBinsDiscretizer + LogisticRegression: 0.94
KBinsDiscretizer + LinearSVC: 0.94
GradientBoostingClassifier: 0.88
SVC: 0.98
```

```
# Code source: Tom Dupré la Tour
# Adapted from plot_classifier_comparison by Gaël Varoquaux and Andreas Müller
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.svm import SVC, LinearSVC
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.utils._testing import ignore_warnings
from sklearn.exceptions import ConvergenceWarning

print(__doc__)

h = .02  # step size in the mesh


def get_name(estimator):
    name = estimator.__class__.__name__
    if name == 'Pipeline':
        name = [get_name(est[1]) for est in estimator.steps]
        name = ' + '.join(name)
    return name


# list of (estimator, param_grid), where param_grid is used in GridSearchCV
classifiers = [
    (LogisticRegression(random_state=0), {
        'C': np.logspace(-2, 7, 10)
    }),
    (LinearSVC(random_state=0), {
        'C': np.logspace(-2, 7, 10)
    }),
    (make_pipeline(
```

(continues on next page)

(continued from previous page)

```
        KBinsDiscretizer(encode='onehot'),
        LogisticRegression(random_state=0)), {
            'kbinsdiscretizer__n_bins': np.arange(2, 10),
            'logisticregression__C': np.logspace(-2, 7, 10),
        }),
    (make_pipeline(
        KBinsDiscretizer(encode='onehot'), LinearSVC(random_state=0)), {
            'kbinsdiscretizer__n_bins': np.arange(2, 10),
            'linearsvc__C': np.logspace(-2, 7, 10),
        }),
    (GradientBoostingClassifier(n_estimators=50, random_state=0), {
        'learning_rate': np.logspace(-4, 0, 10)
    }),
    (SVC(random_state=0), {
        'C': np.logspace(-2, 7, 10)
    }),
]

names = [get_name(e) for e, g in classifiers]

n_samples = 100
datasets = [
    make_moons(n_samples=n_samples, noise=0.2, random_state=0),
    make_circles(n_samples=n_samples, noise=0.2, factor=0.5, random_state=1),
    make_classification(n_samples=n_samples, n_features=2, n_redundant=0,
                        n_informative=2, random_state=2,
                        n_clusters_per_class=1)
]

fig, axes = plt.subplots(nrows=len(datasets), ncols=len(classifiers) + 1,
                         figsize=(21, 9))

cm = plt.cm.PiYG
cm_bright = ListedColormap(['#b30065', '#178000'])

# iterate over datasets
for ds_cnt, (X, y) in enumerate(datasets):
    print('\ndataset %d\n---------' % ds_cnt)

    # preprocess dataset, split into training and test part
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=.5, random_state=42)

    # create the grid for background colors
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(
        np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # plot the dataset first
    ax = axes[ds_cnt, 0]
    if ds_cnt == 0:
        ax.set_title("Input data")
    # plot the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
               edgecolors='k')
```

(continues on next page)

(continued from previous page)

```
    # and testing points
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.6,
               edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())

    # iterate over classifiers
    for est_idx, (name, (estimator, param_grid)) in \
            enumerate(zip(names, classifiers)):
        ax = axes[ds_cnt, est_idx + 1]

        clf = GridSearchCV(estimator=estimator, param_grid=param_grid)
        with ignore_warnings(category=ConvergenceWarning):
            clf.fit(X_train, y_train)
        score = clf.score(X_test, y_test)
        print('%s: %.2f' % (name, score))

        # plot the decision boundary. For that, we will assign a color to each
        # point in the mesh [x_min, x_max]*[y_min, y_max].
        if hasattr(clf, "decision_function"):
            Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
        else:
            Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

        # put the result into a color plot
        Z = Z.reshape(xx.shape)
        ax.contourf(xx, yy, Z, cmap=cm, alpha=.8)

        # plot the training points
        ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
                   edgecolors='k')
        # and testing points
        ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
                   edgecolors='k', alpha=0.6)
        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
        ax.set_xticks(())
        ax.set_yticks(())

        if ds_cnt == 0:
            ax.set_title(name.replace(' + ', '\n'))
        ax.text(0.95, 0.06, ('%.2f' % score).lstrip('0'), size=15,
                bbox=dict(boxstyle='round', alpha=0.8, facecolor='white'),
                transform=ax.transAxes, horizontalalignment='right')


plt.tight_layout()

# Add suptitles above the figure
plt.subplots_adjust(top=0.90)
suptitles = [
    'Linear classifiers',
    'Feature discretization and linear classifiers',
    'Non-linear classifiers',
]
```

(continues on next page)

(continued from previous page)

```
for i, suptitle in zip([1, 3, 5], suptitles):
    ax = axes[0, i]
    ax.text(1.05, 1.25, suptitle, transform=ax.transAxes,
            horizontalalignment='center', size='x-large')
plt.show()
```

Total running time of the script: (0 minutes 18.809 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.25.7 Compare the effect of different scalers on data with outliers

Feature 0 (median income in a block) and feature 5 (number of households) of the California housing dataset have very different scales and contain some very large outliers. These two characteristics lead to difficulties to visualize the data and, more importantly, they can degrade the predictive performance of many machine learning algorithms. Unscaled data can also slow down or even prevent the convergence of many gradient-based estimators.

Indeed many estimators are designed with the assumption that each feature takes values close to zero or more importantly that all features vary on comparable scales. In particular, metric-based and gradient-based estimators often assume approximately standardized data (centered features with unit variances). A notable exception are decision tree-based estimators that are robust to arbitrary scaling of the data.

This example uses different scalers, transformers, and normalizers to bring the data within a pre-defined range.

Scalers are linear (or more precisely affine) transformers and differ from each other in the way to estimate the parameters used to shift and scale each feature.

`QuantileTransformer` provides non-linear transformations in which distances between marginal outliers and inliers are shrunk. `PowerTransformer` provides non-linear transformations in which data is mapped to a normal distribution to stabilize variance and minimize skewness.

Unlike the previous transformations, normalization refers to a per sample transformation instead of a per feature transformation.

The following code is a bit verbose, feel free to jump directly to the analysis of the *results*.

```
# Author:  Raghav RV <rvraghav93@gmail.com>
#          Guillaume Lemaitre <g.lemaitre58@gmail.com>
#          Thomas Unterthiner
# License: BSD 3 clause

import numpy as np

import matplotlib as mpl
from matplotlib import pyplot as plt
from matplotlib import cm

from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import minmax_scale
from sklearn.preprocessing import MaxAbsScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import Normalizer
```

(continues on next page)

(continued from previous page)

```
from sklearn.preprocessing import QuantileTransformer
from sklearn.preprocessing import PowerTransformer

from sklearn.datasets import fetch_california_housing

print(__doc__)

dataset = fetch_california_housing()
X_full, y_full = dataset.data, dataset.target

# Take only 2 features to make visualization easier
# Feature of 0 has a long tail distribution.
# Feature 5 has a few but very large outliers.

X = X_full[:, [0, 5]]

distributions = [
    ('Unscaled data', X),
    ('Data after standard scaling',
        StandardScaler().fit_transform(X)),
    ('Data after min-max scaling',
        MinMaxScaler().fit_transform(X)),
    ('Data after max-abs scaling',
        MaxAbsScaler().fit_transform(X)),
    ('Data after robust scaling',
        RobustScaler(quantile_range=(25, 75)).fit_transform(X)),
    ('Data after power transformation (Yeo-Johnson)',
     PowerTransformer(method='yeo-johnson').fit_transform(X)),
    ('Data after power transformation (Box-Cox)',
     PowerTransformer(method='box-cox').fit_transform(X)),
    ('Data after quantile transformation (gaussian pdf)',
        QuantileTransformer(output_distribution='normal')
        .fit_transform(X)),
    ('Data after quantile transformation (uniform pdf)',
        QuantileTransformer(output_distribution='uniform')
        .fit_transform(X)),
    ('Data after sample-wise L2 normalizing',
        Normalizer().fit_transform(X)),
]

# scale the output between 0 and 1 for the colorbar
y = minmax_scale(y_full)

# plasma does not exist in matplotlib < 1.5
cmap = getattr(cm, 'plasma_r', cm.hot_r)

def create_axes(title, figsize=(16, 6)):
    fig = plt.figure(figsize=figsize)
    fig.suptitle(title)

    # define the axis for the first plot
    left, width = 0.1, 0.22
    bottom, height = 0.1, 0.7
    bottom_h = height + 0.15
    left_h = left + width + 0.02

    rect_scatter = [left, bottom, width, height]
```

(continues on next page)

(continued from previous page)

```
    rect_histx = [left, bottom_h, width, 0.1]
    rect_histy = [left_h, bottom, 0.05, height]

    ax_scatter = plt.axes(rect_scatter)
    ax_histx = plt.axes(rect_histx)
    ax_histy = plt.axes(rect_histy)

    # define the axis for the zoomed-in plot
    left = width + left + 0.2
    left_h = left + width + 0.02

    rect_scatter = [left, bottom, width, height]
    rect_histx = [left, bottom_h, width, 0.1]
    rect_histy = [left_h, bottom, 0.05, height]

    ax_scatter_zoom = plt.axes(rect_scatter)
    ax_histx_zoom = plt.axes(rect_histx)
    ax_histy_zoom = plt.axes(rect_histy)

    # define the axis for the colorbar
    left, width = width + left + 0.13, 0.01

    rect_colorbar = [left, bottom, width, height]
    ax_colorbar = plt.axes(rect_colorbar)

    return ((ax_scatter, ax_histy, ax_histx),
            (ax_scatter_zoom, ax_histy_zoom, ax_histx_zoom),
            ax_colorbar)


def plot_distribution(axes, X, y, hist_nbins=50, title="",
                      x0_label="", x1_label=""):
    ax, hist_X1, hist_X0 = axes

    ax.set_title(title)
    ax.set_xlabel(x0_label)
    ax.set_ylabel(x1_label)

    # The scatter plot
    colors = cmap(y)
    ax.scatter(X[:, 0], X[:, 1], alpha=0.5, marker='o', s=5, lw=0, c=colors)

    # Removing the top and the right spine for aesthetics
    # make nice axis layout
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()
    ax.spines['left'].set_position(('outward', 10))
    ax.spines['bottom'].set_position(('outward', 10))

    # Histogram for axis X1 (feature 5)
    hist_X1.set_ylim(ax.get_ylim())
    hist_X1.hist(X[:, 1], bins=hist_nbins, orientation='horizontal',
                 color='grey', ec='grey')
    hist_X1.axis('off')
```

(continues on next page)

(continued from previous page)

```
    # Histogram for axis X0 (feature 0)
    hist_X0.set_xlim(ax.get_xlim())
    hist_X0.hist(X[:, 0], bins=hist_nbins, orientation='vertical',
                 color='grey', ec='grey')
    hist_X0.axis('off')
```

Two plots will be shown for each scaler/normalizer/transformer. The left figure will show a scatter plot of the full data set while the right figure will exclude the extreme values considering only 99 % of the data set, excluding marginal outliers. In addition, the marginal distributions for each feature will be shown on the side of the scatter plot.

```
def make_plot(item_idx):
    title, X = distributions[item_idx]
    ax_zoom_out, ax_zoom_in, ax_colorbar = create_axes(title)
    axarr = (ax_zoom_out, ax_zoom_in)
    plot_distribution(axarr[0], X, y, hist_nbins=200,
                      x0_label="Median Income",
                      x1_label="Number of households",
                      title="Full data")

    # zoom-in
    zoom_in_percentile_range = (0, 99)
    cutoffs_X0 = np.percentile(X[:, 0], zoom_in_percentile_range)
    cutoffs_X1 = np.percentile(X[:, 1], zoom_in_percentile_range)

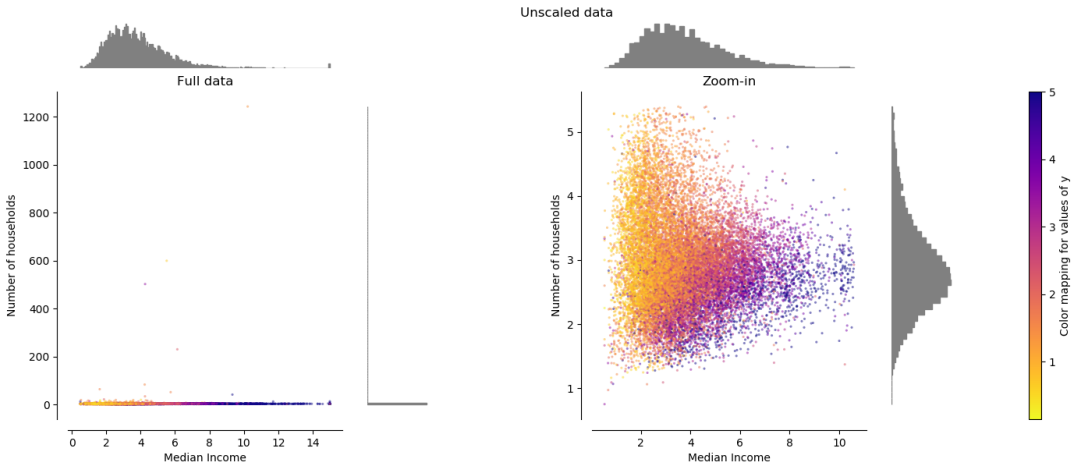
    non_outliers_mask = (
        np.all(X > [cutoffs_X0[0], cutoffs_X1[0]], axis=1) &
        np.all(X < [cutoffs_X0[1], cutoffs_X1[1]], axis=1))
    plot_distribution(axarr[1], X[non_outliers_mask], y[non_outliers_mask],
                      hist_nbins=50,
                      x0_label="Median Income",
                      x1_label="Number of households",
                      title="Zoom-in")

    norm = mpl.colors.Normalize(y_full.min(), y_full.max())
    mpl.colorbar.ColorbarBase(ax_colorbar, cmap=cmap,
                              norm=norm, orientation='vertical',
                              label='Color mapping for values of y')
```

Original data

Each transformation is plotted showing two transformed features, with the left plot showing the entire dataset, and the right zoomed-in to show the dataset without the marginal outliers. A large majority of the samples are compacted to a specific range, [0, 10] for the median income and [0, 6] for the number of households. Note that there are some marginal outliers (some blocks have more than 1200 households). Therefore, a specific pre-processing can be very beneficial depending of the application. In the following, we present some insights and behaviors of those pre-processing methods in the presence of marginal outliers.

```
make_plot(0)
```

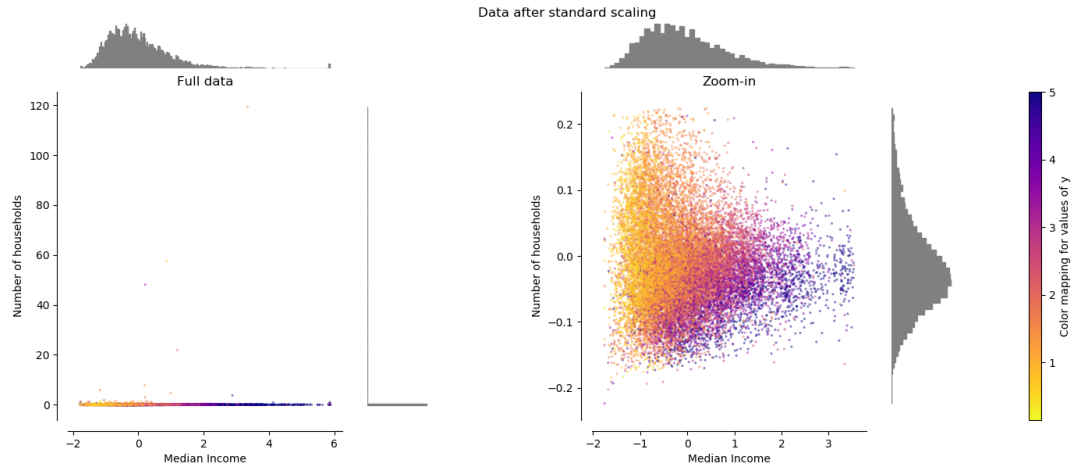


StandardScaler

`StandardScaler` removes the mean and scales the data to unit variance. However, the outliers have an influence when computing the empirical mean and standard deviation which shrink the range of the feature values as shown in the left figure below. Note in particular that because the outliers on each feature have different magnitudes, the spread of the transformed data on each feature is very different: most of the data lie in the [-2, 4] range for the transformed median income feature while the same data is squeezed in the smaller [-0.2, 0.2] range for the transformed number of households.

`StandardScaler` therefore cannot guarantee balanced feature scales in the presence of outliers.

```
make_plot(1)
```

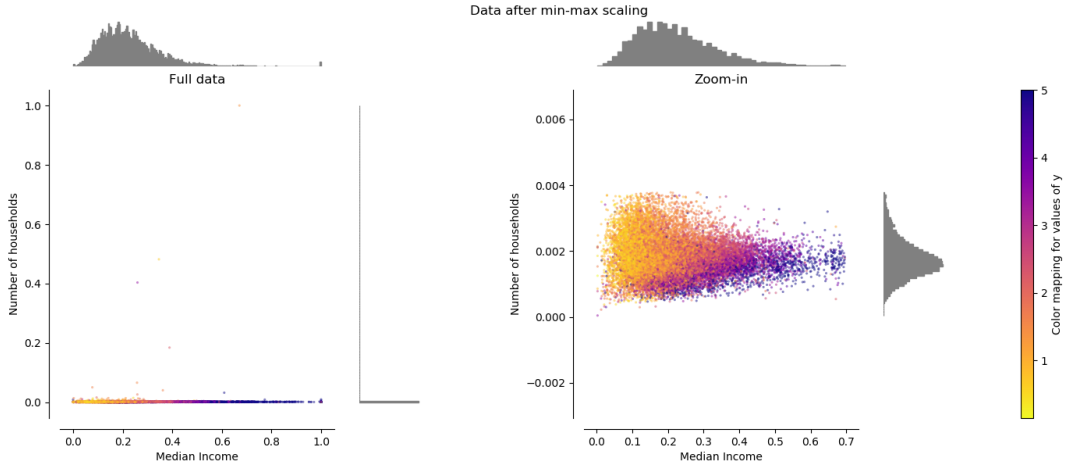


MinMaxScaler

`MinMaxScaler` rescales the data set such that all feature values are in the range [0, 1] as shown in the right panel below. However, this scaling compress all inliers in the narrow range [0, 0.005] for the transformed number of households.

As `StandardScaler`, `MinMaxScaler` is very sensitive to the presence of outliers.

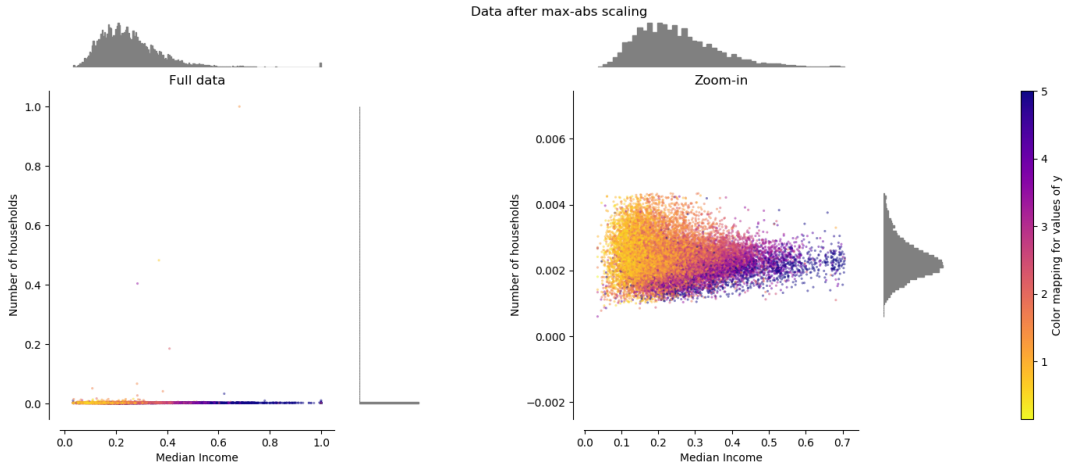
```
make_plot(2)
```



MaxAbsScaler

`MaxAbsScaler` differs from the previous scaler such that the absolute values are mapped in the range [0, 1]. On positive only data, this scaler behaves similarly to `MinMaxScaler` and therefore also suffers from the presence of large outliers.

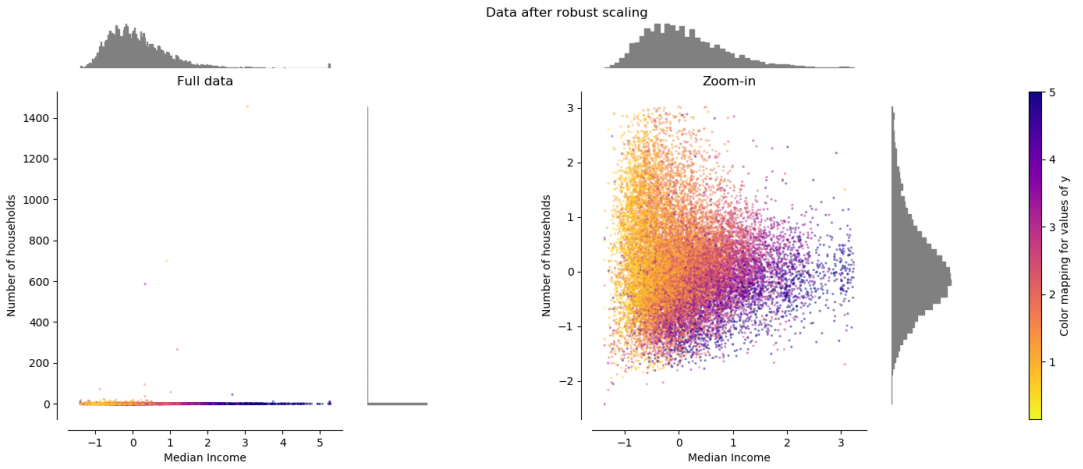
```
make_plot(3)
```



RobustScaler

Unlike the previous scalers, the centering and scaling statistics of this scaler are based on percentiles and are therefore not influenced by a few number of very large marginal outliers. Consequently, the resulting range of the transformed feature values is larger than for the previous scalers and, more importantly, are approximately similar: for both features most of the transformed values lie in a [-2, 3] range as seen in the zoomed-in figure. Note that the outliers themselves are still present in the transformed data. If a separate outlier clipping is desirable, a non-linear transformation is required (see below).

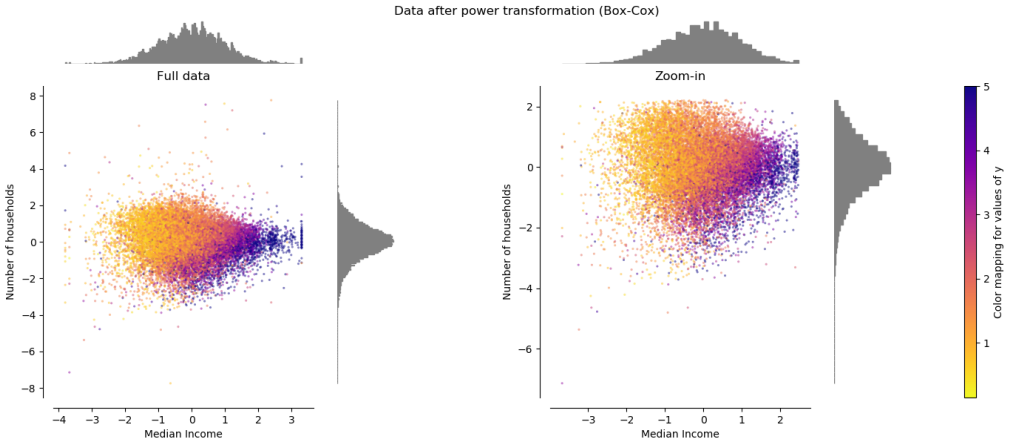
```
make_plot(4)
```



PowerTransformer

`PowerTransformer` applies a power transformation to each feature to make the data more Gaussian-like. Currently, `PowerTransformer` implements the Yeo-Johnson and Box-Cox transforms. The power transform finds the optimal scaling factor to stabilize variance and mimimize skewness through maximum likelihood estimation. By default, `PowerTransformer` also applies zero-mean, unit variance normalization to the transformed output. Note that Box-Cox can only be applied to strictly positive data. Income and number of households happen to be strictly positive, but if negative values are present the Yeo-Johnson transformed is to be preferred.

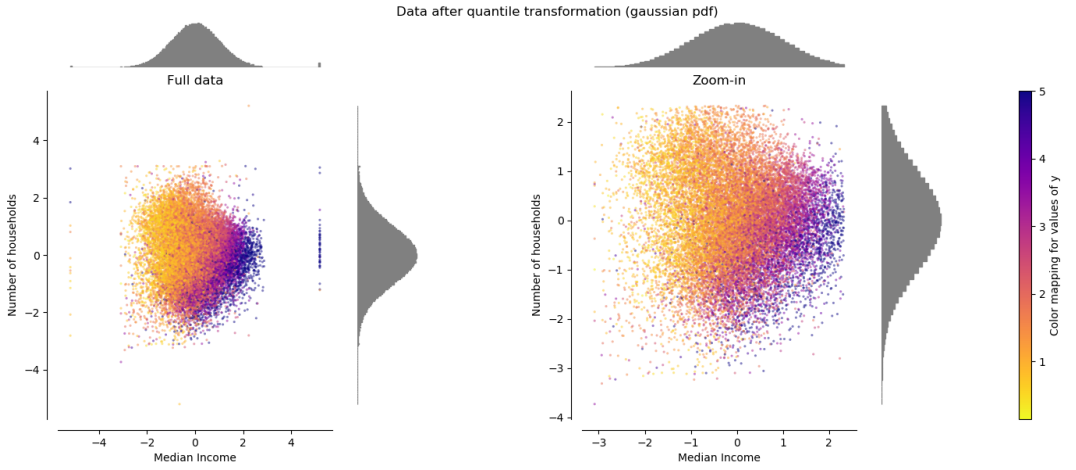
```
make_plot(5)
make_plot(6)
```



QuantileTransformer (Gaussian output)

`QuantileTransformer` has an additional `output_distribution` parameter allowing to match a Gaussian distribution instead of a uniform distribution. Note that this non-parametetric transformer introduces saturation artifacts for extreme values.

```
make_plot(7)
```

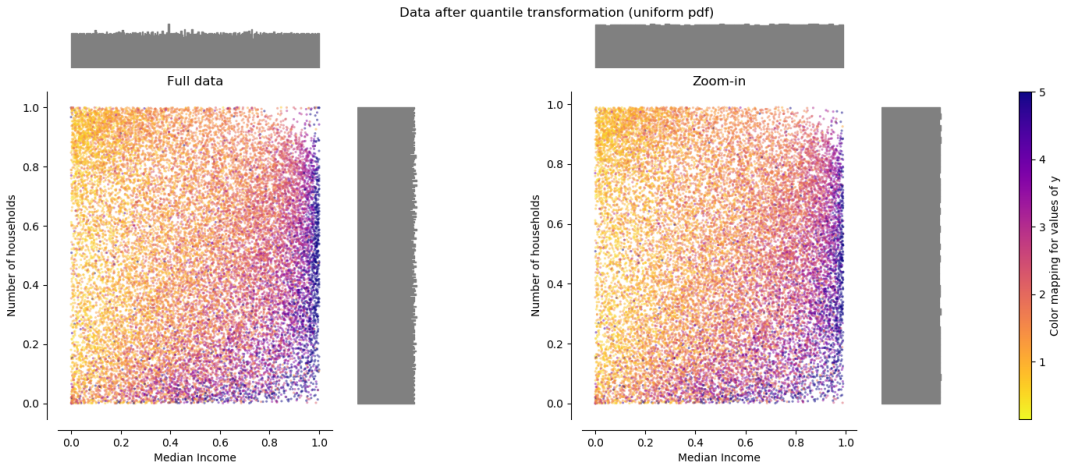


QuantileTransformer (uniform output)

`QuantileTransformer` applies a non-linear transformation such that the probability density function of each feature will be mapped to a uniform distribution. In this case, all the data will be mapped in the range [0, 1], even the outliers which cannot be distinguished anymore from the inliers.

As `RobustScaler`, `QuantileTransformer` is robust to outliers in the sense that adding or removing outliers in the training set will yield approximately the same transformation on held out data. But contrary to `RobustScaler`, `QuantileTransformer` will also automatically collapse any outlier by setting them to the a priori defined range boundaries (0 and 1).

```
make_plot(8)
```

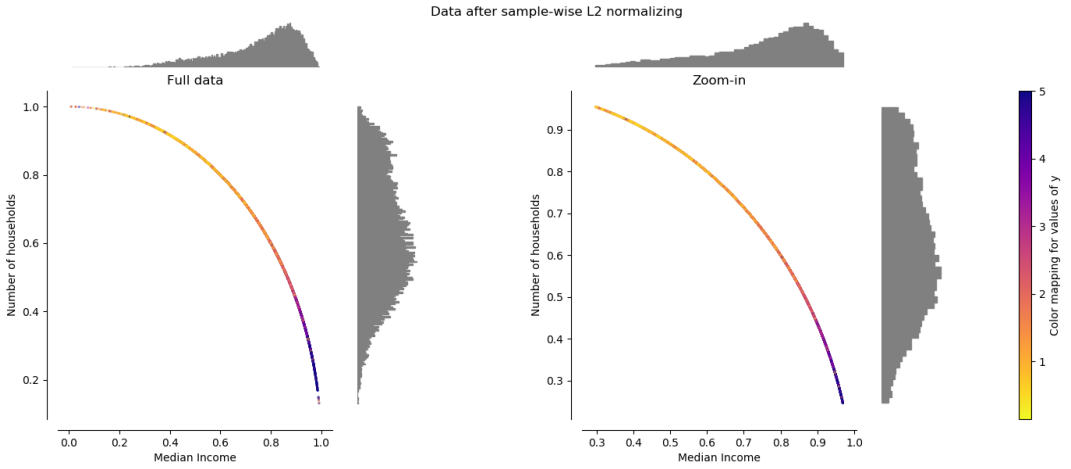


Normalizer

The `Normalizer` rescales the vector for each sample to have unit norm, independently of the distribution of the samples. It can be seen on both figures below where all samples are mapped onto the unit circle. In our example the two selected features have only positive values; therefore the transformed data only lie in the positive quadrant. This would not be the case if some original features had a mix of positive and negative values.

```
make_plot(9)

plt.show()
```



Total running time of the script: (0 minutes 11.455 seconds)

Estimated memory usage: 8 MB

6.26 Release Highlights

These examples illustrate the main features of the releases of scikit-learn.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.26.1 Release Highlights for scikit-learn 0.22

We are pleased to announce the release of scikit-learn 0.22, which comes with many bug fixes and new features! We detail below a few of the major features of this release. For an exhaustive list of all the changes, please refer to the *release notes*.

To install the latest version (with pip):

```
pip install --upgrade scikit-learn
```

or with conda:

```
conda install scikit-learn
```

New plotting API

A new plotting API is available for creating visualizations. This new API allows for quickly adjusting the visuals of a plot without involving any recomputation. It is also possible to add different plots to the same figure. The following example illustrates `plot_roc_curve`, but other plots utilities are supported like `plot_partial_dependence`, `plot_precision_recall_curve`, and `plot_confusion_matrix`. Read more about this new API in the *User Guide*.

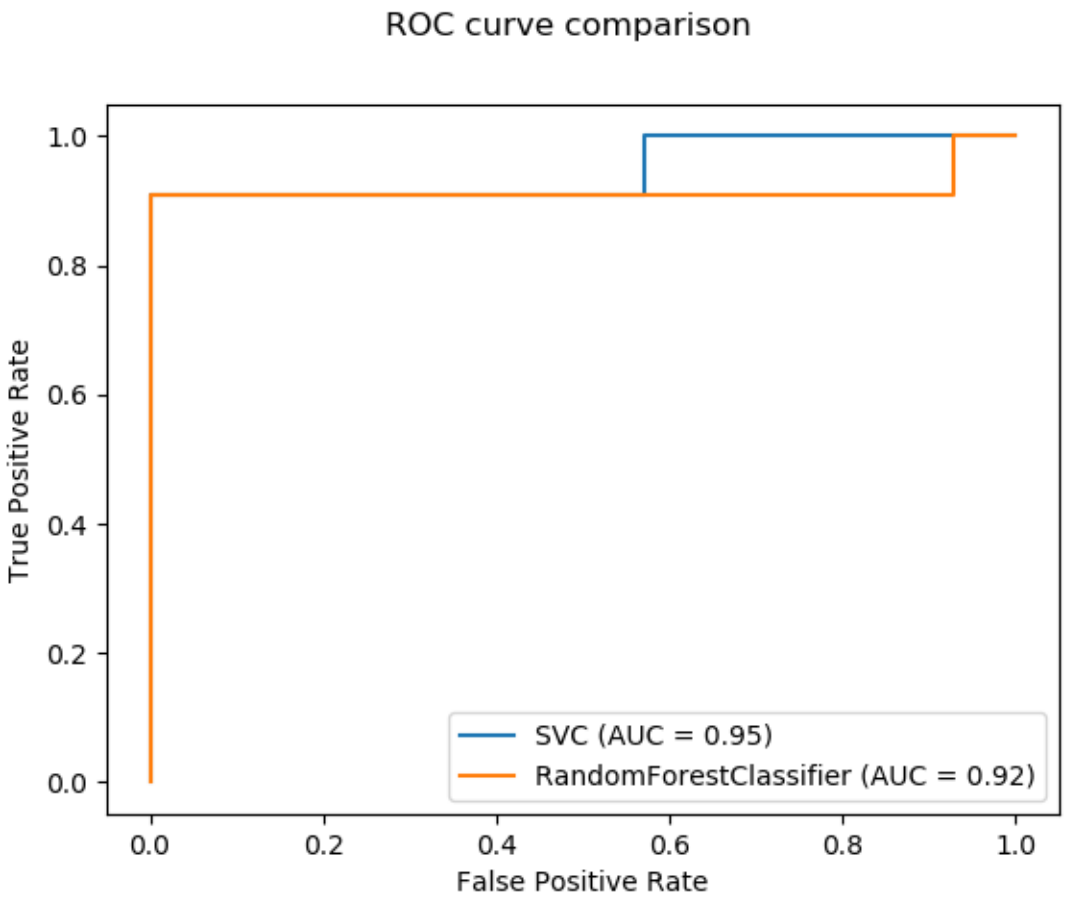
```
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import plot_roc_curve
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt

X, y = make_classification(random_state=0)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

svc = SVC(random_state=42)
svc.fit(X_train, y_train)
rfc = RandomForestClassifier(random_state=42)
rfc.fit(X_train, y_train)

svc_disp = plot_roc_curve(svc, X_test, y_test)
rfc_disp = plot_roc_curve(rfc, X_test, y_test, ax=svc_disp.ax_)
rfc_disp.figure_.suptitle("ROC curve comparison")

plt.show()
```



Stacking Classifier and Regressor

StackingClassifier and *StackingRegressor* allow you to have a stack of estimators with a final classifier or a regressor. Stacked generalization consists in stacking the output of individual estimators and use a classifier to compute the final prediction. Stacking allows to use the strength of each individual estimator by using their output as input of a final estimator. Base estimators are fitted on the full `X` while the final estimator is trained using cross-validated predictions of the base estimators using `cross_val_predict`.

Read more in the *User Guide*.

```
from sklearn.datasets import load_iris
from sklearn.svm import LinearSVC
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.ensemble import StackingClassifier
from sklearn.model_selection import train_test_split

X, y = load_iris(return_X_y=True)
estimators = [
    ('rf', RandomForestClassifier(n_estimators=10, random_state=42)),
    ('svr', make_pipeline(StandardScaler(),
                          LinearSVC(random_state=42)))
]
```

(continues on next page)

(continued from previous page)

```
clf = StackingClassifier(
    estimators=estimators, final_estimator=LogisticRegression()
)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, stratify=y, random_state=42
)
clf.fit(X_train, y_train).score(X_test, y_test)
```

Out:

```
0.9473684210526315
```

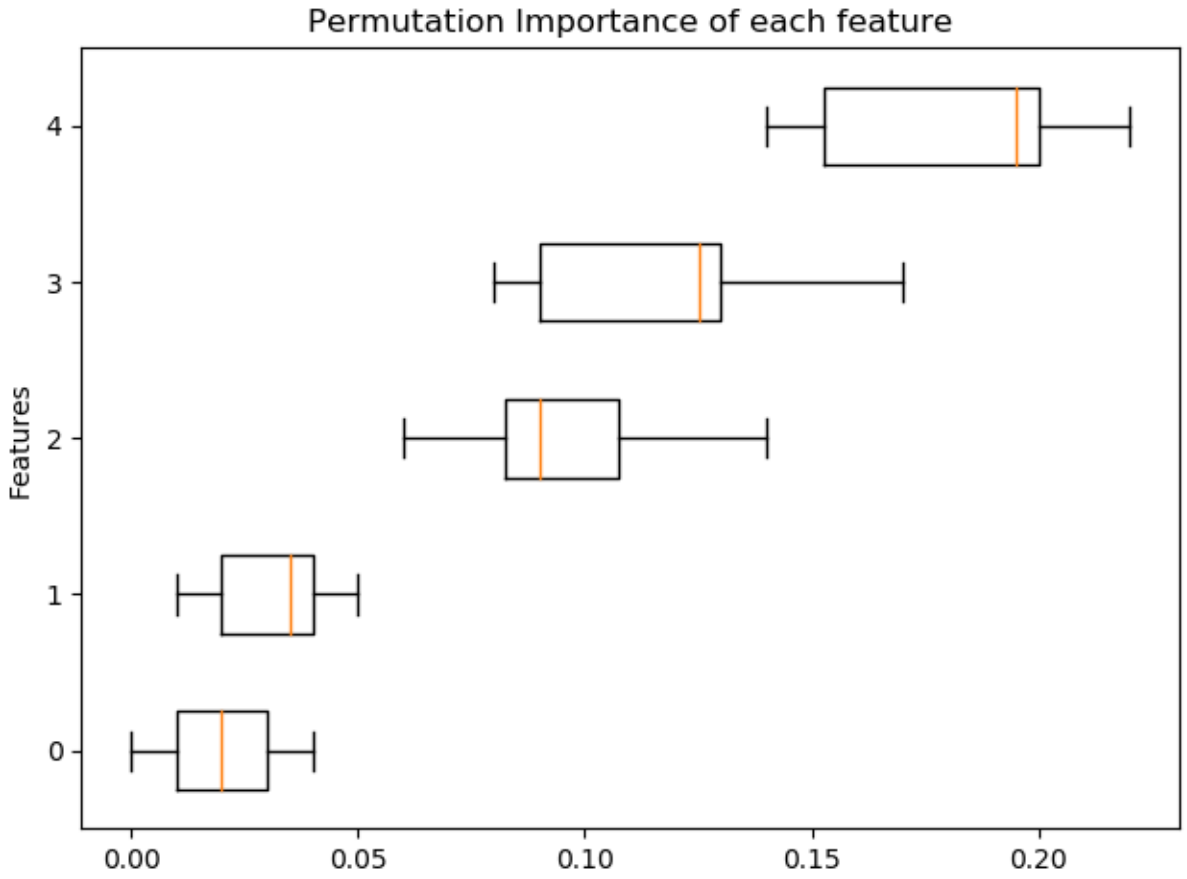
Permutation-based feature importance

The *inspection.permutation_importance* can be used to get an estimate of the importance of each feature, for any fitted estimator:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.inspection import permutation_importance

X, y = make_classification(random_state=0, n_features=5, n_informative=3)
rf = RandomForestClassifier(random_state=0).fit(X, y)
result = permutation_importance(rf, X, y, n_repeats=10, random_state=0,
                                n_jobs=-1)

fig, ax = plt.subplots()
sorted_idx = result.importances_mean.argsort()
ax.boxplot(result.importances[sorted_idx].T,
           vert=False, labels=range(X.shape[1]))
ax.set_title("Permutation Importance of each feature")
ax.set_ylabel("Features")
fig.tight_layout()
plt.show()
```



Native support for missing values for gradient boosting

The *ensemble.HistGradientBoostingClassifier* and *ensemble.HistGradientBoostingRegressor* now have native support for missing values (NaNs). This means that there is no need for imputing data when training or predicting.

```
from sklearn.experimental import enable_hist_gradient_boosting  # noqa
from sklearn.ensemble import HistGradientBoostingClassifier
import numpy as np

X = np.array([0, 1, 2, np.nan]).reshape(-1, 1)
y = [0, 0, 1, 1]

gbdt = HistGradientBoostingClassifier(min_samples_leaf=1).fit(X, y)
print(gbdt.predict(X))
```

Out:

```
[0 0 1 1]
```

Precomputed sparse nearest neighbors graph

Most estimators based on nearest neighbors graphs now accept precomputed sparse graphs as input, to reuse the same graph for multiple estimator fits. To use this feature in a pipeline, one can use the `memory` parameter, along with one of the two new transformers, *neighbors.KNeighborsTransformer* and *neighbors.RadiusNeighborsTransformer*. The precomputation can also be performed by custom estimators to use alternative implementations, such as approximate nearest neighbors methods. See more details in the *User Guide*.

```
from tempfile import TemporaryDirectory
from sklearn.neighbors import KNeighborsTransformer
from sklearn.manifold import Isomap
from sklearn.pipeline import make_pipeline

X, y = make_classification(random_state=0)

with TemporaryDirectory(prefix="sklearn_cache_") as tmpdir:
    estimator = make_pipeline(
        KNeighborsTransformer(n_neighbors=10, mode='distance'),
        Isomap(n_neighbors=10, metric='precomputed'),
        memory=tmpdir)
    estimator.fit(X)

    # We can decrease the number of neighbors and the graph will not be
    # recomputed.
    estimator.set_params(isomap__n_neighbors=5)
    estimator.fit(X)
```

KNN Based Imputation

We now support imputation for completing missing values using k-Nearest Neighbors.

Each sample's missing values are imputed using the mean value from `n_neighbors` nearest neighbors found in the training set. Two samples are close if the features that neither is missing are close. By default, a euclidean distance metric that supports missing values, `nan_euclidean_distances`, is used to find the nearest neighbors.

Read more in the *User Guide*.

```
import numpy as np
from sklearn.impute import KNNImputer

X = [[1, 2, np.nan], [3, 4, 3], [np.nan, 6, 5], [8, 8, 7]]
imputer = KNNImputer(n_neighbors=2)
print(imputer.fit_transform(X))
```

Out:

```
[[1.  2.  4. ]
 [3.  4.  3. ]
 [5.5 6.  5. ]
 [8.  8.  7. ]]
```

Tree pruning

It is now possible to prune most tree-based estimators once the trees are built. The pruning is based on minimal cost-complexity. Read more in the *User Guide* for details.

```
X, y = make_classification(random_state=0)

rf = RandomForestClassifier(random_state=0, ccp_alpha=0).fit(X, y)
print("Average number of nodes without pruning {:.1f}".format(
    np.mean([e.tree_.node_count for e in rf.estimators_])))

rf = RandomForestClassifier(random_state=0, ccp_alpha=0.05).fit(X, y)
print("Average number of nodes with pruning {:.1f}".format(
    np.mean([e.tree_.node_count for e in rf.estimators_])))
```

Out:

```
Average number of nodes without pruning 22.3
Average number of nodes with pruning 6.4
```

Retrieve dataframes from OpenML

datasets.fetch_openml can now return pandas dataframe and thus properly handle datasets with heterogeneous data:

```
from sklearn.datasets import fetch_openml

titanic = fetch_openml('titanic', version=1, as_frame=True)
print(titanic.data.head()[['pclass', 'embarked']])
```

Out:

```
   pclass embarked
0     1.0        S
1     1.0        S
2     1.0        S
3     1.0        S
4     1.0        S
```

Checking scikit-learn compatibility of an estimator

Developers can check the compatibility of their scikit-learn compatible estimators using *check_estimator*. For instance, the `check_estimator(LinearSVC)` passes.

We now provide a `pytest` specific decorator which allows `pytest` to run all checks independently and report the checks that are failing.

```
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.utils.estimator_checks import parametrize_with_checks


@parametrize_with_checks([LogisticRegression, DecisionTreeRegressor])
def test_sklearn_compatible_estimator(estimator, check):
    check(estimator)
```

ROC AUC now supports multiclass classification

The `roc_auc_score` function can also be used in multi-class classification. Two averaging strategies are currently supported: the one-vs-one algorithm computes the average of the pairwise ROC AUC scores, and the one-vs-rest algorithm computes the average of the ROC AUC scores for each class against all other classes. In both cases, the multiclass ROC AUC scores are computed from the probability estimates that a sample belongs to a particular class according to the model. The OvO and OvR algorithms support weighting uniformly (`average='macro'`) and weighting by the prevalence (`average='weighted'`).

Read more in the *User Guide*.

```
from sklearn.datasets import make_classification
from sklearn.svm import SVC
from sklearn.metrics import roc_auc_score

X, y = make_classification(n_classes=4, n_informative=16)
clf = SVC(decision_function_shape='ovo', probability=True).fit(X, y)
print(roc_auc_score(y, clf.predict_proba(X), multi_class='ovo'))
```

Out:

```
0.9957333333333332
```

Total running time of the script: (0 minutes 4.730 seconds)

Estimated memory usage: 8 MB

6.27 Semi Supervised Classification

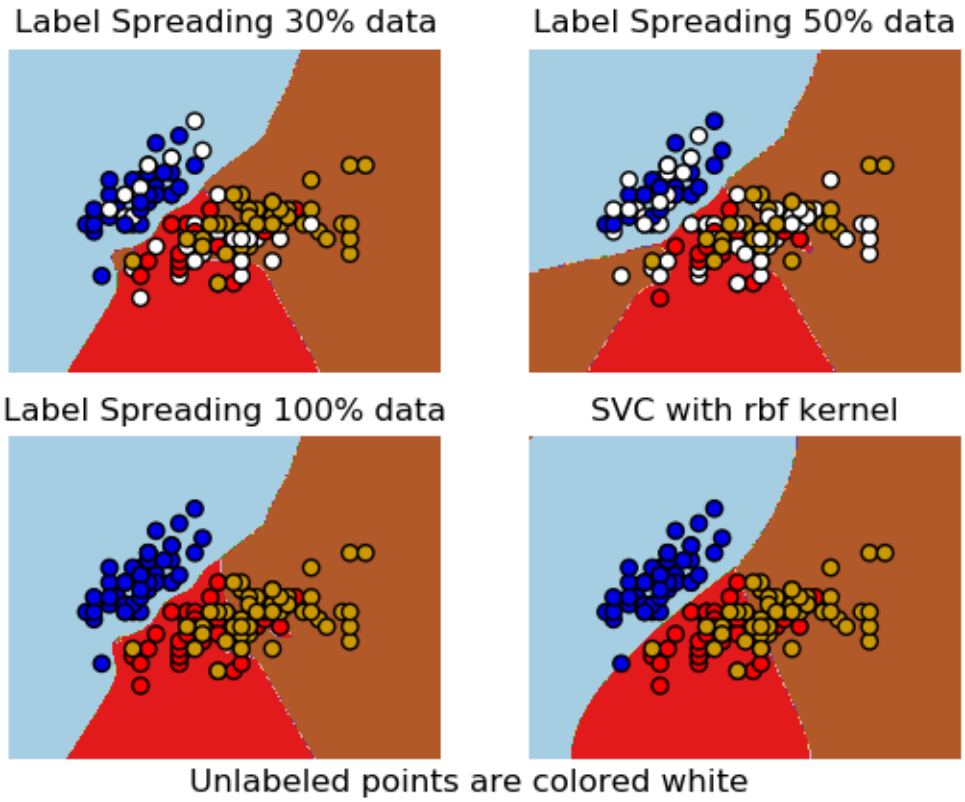
Examples concerning the *sklearn.semi_supervised* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.27.1 Decision boundary of label propagation versus SVM on the Iris dataset

Comparison for decision boundary generated on iris dataset between Label Propagation and SVM.

This demonstrates Label Propagation learning a good boundary even with a small amount of labeled data.




```
print(__doc__)

# Authors: Clay Woolam <clay@woolam.org>
# License: BSD

import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn import svm
from sklearn.semi_supervised import LabelSpreading

rng = np.random.RandomState(0)

iris = datasets.load_iris()

X = iris.data[:, :2]
y = iris.target

# step size in the mesh
h = .02

y_30 = np.copy(y)
y_30[rng.rand(len(y)) < 0.3] = -1
y_50 = np.copy(y)
y_50[rng.rand(len(y)) < 0.5] = -1
```

(continues on next page)

(continued from previous page)

```
# we create an instance of SVM and fit out data. We do not scale our
# data since we want to plot the support vectors
ls30 = (LabelSpreading().fit(X, y_30), y_30)
ls50 = (LabelSpreading().fit(X, y_50), y_50)
ls100 = (LabelSpreading().fit(X, y), y)
rbf_svc = (svm.SVC(kernel='rbf', gamma=.5).fit(X, y), y)

# create a mesh to plot in
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

# title for the plots
titles = ['Label Spreading 30% data',
          'Label Spreading 50% data',
          'Label Spreading 100% data',
          'SVC with rbf kernel']

color_map = {-1: (1, 1, 1), 0: (0, 0, .9), 1: (1, 0, 0), 2: (.8, .6, 0)}

for i, (clf, y_train) in enumerate((ls30, ls50, ls100, rbf_svc)):
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    plt.subplot(2, 2, i + 1)
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
    plt.axis('off')

    # Plot also the training points
    colors = [color_map[y] for y in y_train]
    plt.scatter(X[:, 0], X[:, 1], c=colors, edgecolors='black')

    plt.title(titles[i])

plt.suptitle("Unlabeled points are colored white", y=0.1)
plt.show()
```

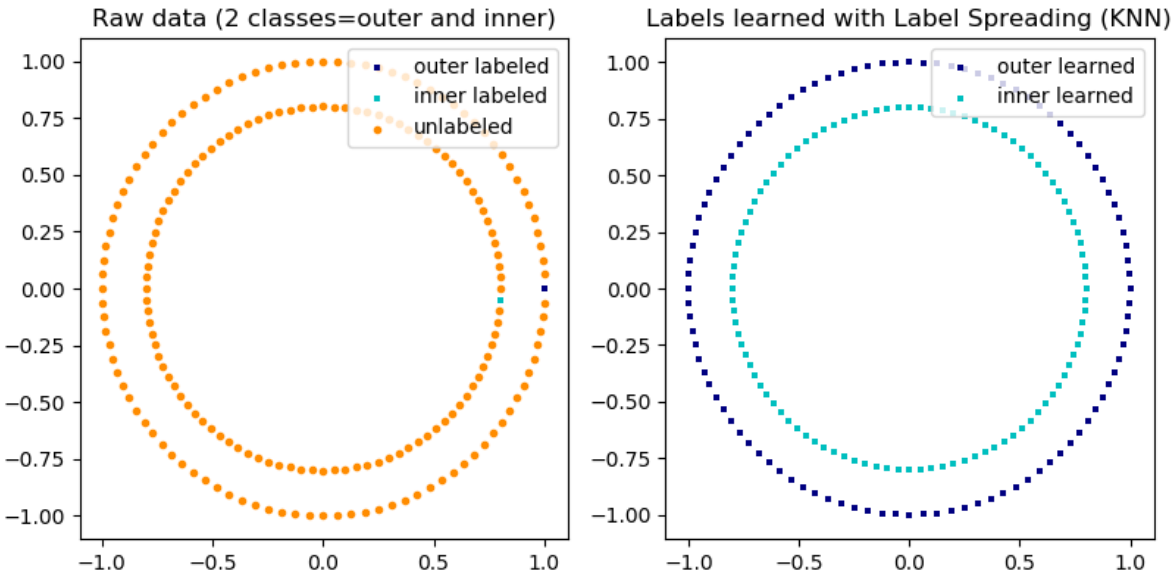
Total running time of the script: (0 minutes 1.188 seconds)

Estimated memory usage: 79 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.27.2 Label Propagation learning a complex structure

Example of LabelPropagation learning a complex internal structure to demonstrate “manifold learning”. The outer circle should be labeled “red” and the inner circle “blue”. Because both label groups lie inside their own distinct shape, we can see that the labels propagate correctly around the circle.




```
print(__doc__)

# Authors: Clay Woolam <clay@woolam.org>
#          Andreas Mueller <amueller@ais.uni-bonn.de>
# License: BSD

import numpy as np
import matplotlib.pyplot as plt
from sklearn.semi_supervised import LabelSpreading
from sklearn.datasets import make_circles

# generate ring with inner box
n_samples = 200
X, y = make_circles(n_samples=n_samples, shuffle=False)
outer, inner = 0, 1
labels = np.full(n_samples, -1.)
labels[0] = outer
labels[-1] = inner

# #############################################################################
# Learn with LabelSpreading
label_spread = LabelSpreading(kernel='knn', alpha=0.8)
label_spread.fit(X, labels)

# #############################################################################
# Plot output labels
output_labels = label_spread.transduction_
plt.figure(figsize=(8.5, 4))
plt.subplot(1, 2, 1)
plt.scatter(X[labels == outer, 0], X[labels == outer, 1], color='navy',
            marker='s', lw=0, label="outer labeled", s=10)
plt.scatter(X[labels == inner, 0], X[labels == inner, 1], color='c',
            marker='s', lw=0, label='inner labeled', s=10)
plt.scatter(X[labels == -1, 0], X[labels == -1, 1], color='darkorange',
            marker='.', label='unlabeled')
plt.legend(scatterpoints=1, shadow=False, loc='upper right')
plt.title("Raw data (2 classes=outer and inner)")
```

(continues on next page)

(continued from previous page)

```
plt.subplot(1, 2, 2)
output_label_array = np.asarray(output_labels)
outer_numbers = np.where(output_label_array == outer)[0]
inner_numbers = np.where(output_label_array == inner)[0]
plt.scatter(X[outer_numbers, 0], X[outer_numbers, 1], color='navy',
            marker='s', lw=0, s=10, label="outer learned")
plt.scatter(X[inner_numbers, 0], X[inner_numbers, 1], color='c',
            marker='s', lw=0, s=10, label="inner learned")
plt.legend(scatterpoints=1, shadow=False, loc='upper right')
plt.title("Labels learned with Label Spreading (KNN)")

plt.subplots_adjust(left=0.07, bottom=0.07, right=0.93, top=0.92)
plt.show()
```

Total running time of the script: (0 minutes 0.621 seconds)

Estimated memory usage: 8 MB

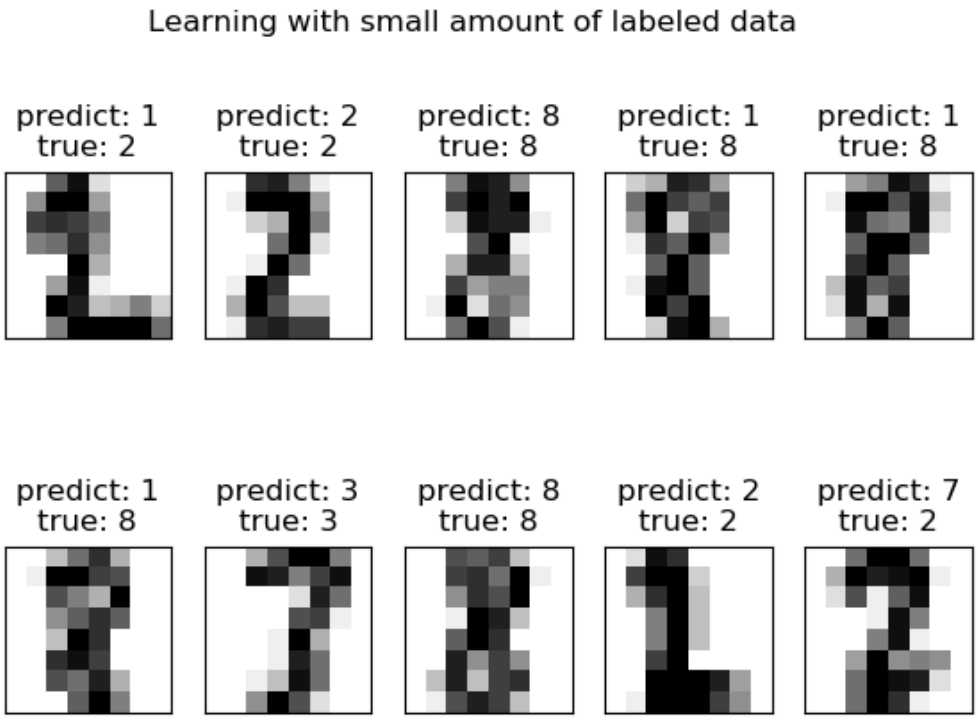
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.27.3 Label Propagation digits: Demonstrating performance

This example demonstrates the power of semisupervised learning by training a Label Spreading model to classify handwritten digits with sets of very few labels.

The handwritten digit dataset has 1797 total points. The model will be trained using all points, but only 30 will be labeled. Results in the form of a confusion matrix and a series of metrics over each class will be very good.

At the end, the top 10 most uncertain predictions will be shown.



Out:

```
Label Spreading model: 40 labeled & 300 unlabeled points (340 total)
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        27
           1       0.82      1.00      0.90        37
           2       1.00      0.86      0.92        28
           3       1.00      0.80      0.89        35
           4       0.92      1.00      0.96        24
           5       0.74      0.94      0.83        34
           6       0.89      0.96      0.92        25
           7       0.94      0.89      0.91        35
           8       1.00      0.68      0.81        31
           9       0.81      0.88      0.84        24

    accuracy                           0.90       300
   macro avg       0.91      0.90      0.90       300
weighted avg       0.91      0.90      0.90       300

Confusion matrix
[[27  0  0  0  0  0  0  0  0  0]
 [ 0 37  0  0  0  0  0  0  0  0]
 [ 0  1 24  0  0  0  2  1  0  0]
 [ 0  0  0 28  0  5  0  1  0  1]
 [ 0  0  0  0 24  0  0  0  0  0]
 [ 0  0  0  0  0 32  0  0  0  2]
```

(continues on next page)

(continued from previous page)

```
 [ 0  0  0  0  0  1 24  0  0  0]
 [ 0  0  0  0  1  3  0 31  0  0]
 [ 0  7  0  0  0  0  1  0 21  2]
 [ 0  0  0  0  1  2  0  0  0 21]]
```

```
print(__doc__)

# Authors: Clay Woolam <clay@woolam.org>
# License: BSD

import numpy as np
import matplotlib.pyplot as plt

from scipy import stats

from sklearn import datasets
from sklearn.semi_supervised import LabelSpreading

from sklearn.metrics import confusion_matrix, classification_report

digits = datasets.load_digits()
rng = np.random.RandomState(2)
indices = np.arange(len(digits.data))
rng.shuffle(indices)

X = digits.data[indices[:340]]
y = digits.target[indices[:340]]
images = digits.images[indices[:340]]

n_total_samples = len(y)
n_labeled_points = 40

indices = np.arange(n_total_samples)

unlabeled_set = indices[n_labeled_points:]

# ############################################################################
# Shuffle everything around
y_train = np.copy(y)
y_train[unlabeled_set] = -1

# ############################################################################
# Learn with LabelSpreading
lp_model = LabelSpreading(gamma=.25, max_iter=20)
lp_model.fit(X, y_train)
predicted_labels = lp_model.transduction_[unlabeled_set]
true_labels = y[unlabeled_set]

cm = confusion_matrix(true_labels, predicted_labels, labels=lp_model.classes_)

print("Label Spreading model: %d labeled & %d unlabeled points (%d total)" %
```

(continues on next page)

(continued from previous page)

```
      (n_labeled_points, n_total_samples - n_labeled_points, n_total_samples))

print(classification_report(true_labels, predicted_labels))

print("Confusion matrix")
print(cm)

# #############################################################################
# Calculate uncertainty values for each transduced distribution
pred_entropies = stats.distributions.entropy(lp_model.label_distributions_.T)

# #############################################################################
# Pick the top 10 most uncertain labels
uncertainty_index = np.argsort(pred_entropies)[-10:]

# #############################################################################
# Plot
f = plt.figure(figsize=(7, 5))
for index, image_index in enumerate(uncertainty_index):
    image = images[image_index]

    sub = f.add_subplot(2, 5, index + 1)
    sub.imshow(image, cmap=plt.cm.gray_r)
    plt.xticks([])
    plt.yticks([])
    sub.set_title('predict: %i\ntrue: %i' % (
        lp_model.transduction_[image_index], y[image_index]))

f.suptitle('Learning with small amount of labeled data')
plt.show()
```

Total running time of the script: (0 minutes 0.544 seconds)

Estimated memory usage: 8 MB

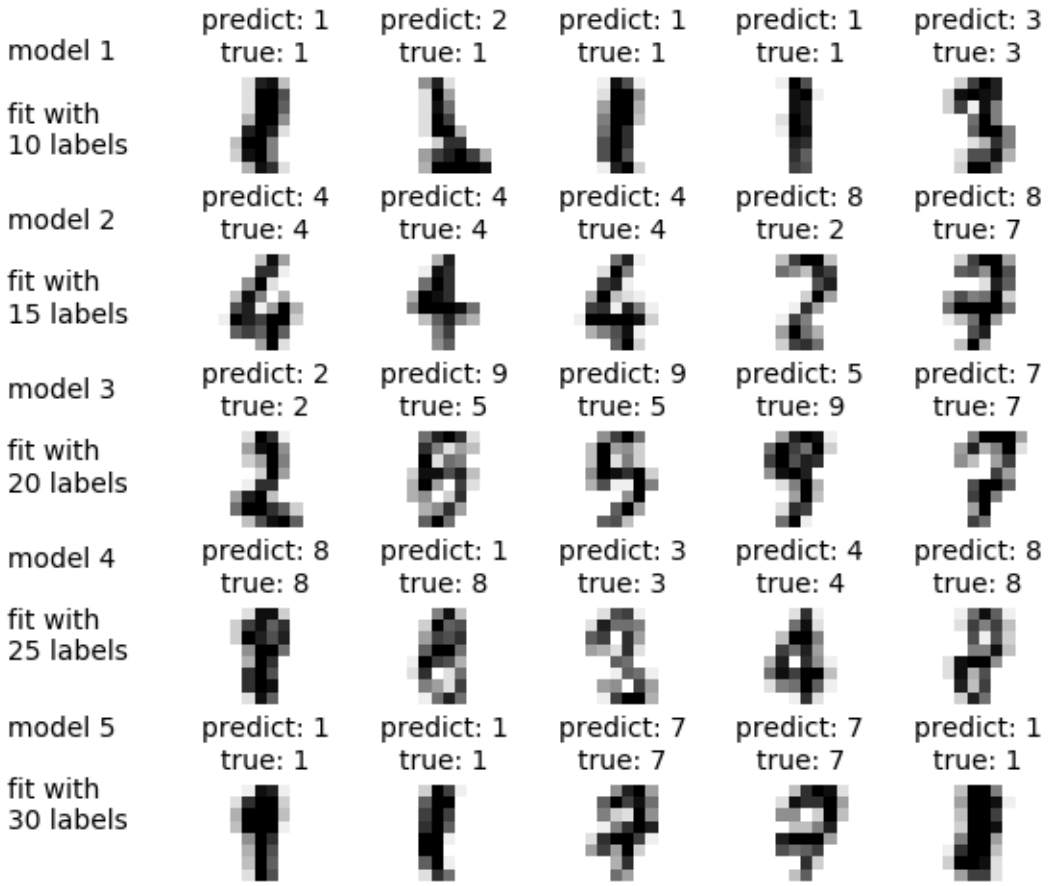
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.27.4 Label Propagation digits active learning

Demonstrates an active learning technique to learn handwritten digits using label propagation.

We start by training a label propagation model with only 10 labeled points, then we select the top five most uncertain points to label. Next, we train with 15 labeled points (original 10 + 5 new ones). We repeat this process four times to have a model trained with 30 labeled examples. Note you can increase this to label more than 30 by changing `max_iterations`. Labeling more than 30 can be useful to get a sense for the speed of convergence of this active learning technique.

A plot will appear showing the top 5 most uncertain digits for each iteration of training. These may or may not contain mistakes, but we will train the next model with their true labels.



Out:

```
Iteration 0 ___________________________________________________________________________
Label Spreading model: 40 labeled & 290 unlabeled (330 total)
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        22
           1       0.78      0.69      0.73        26
           2       0.93      0.93      0.93        29
           3       1.00      0.89      0.94        27
           4       0.92      0.96      0.94        23
           5       0.96      0.70      0.81        33
           6       0.97      0.97      0.97        35
           7       0.94      0.91      0.92        33
           8       0.62      0.89      0.74        28
           9       0.73      0.79      0.76        34

    accuracy                           0.87       290
   macro avg       0.89      0.87      0.87       290
weighted avg       0.88      0.87      0.87       290

Confusion matrix
[[22  0  0  0  0  0  0  0  0  0]
 [ 0 18  2  0  0  0  1  0  5  0]
 [ 0  0 27  0  0  0  0  0  2  0]
 [ 0  0  0 24  0  0  0  0  3  0]
```

(continues on next page)

(continued from previous page)

```
 [ 0  1  0  0 22  0  0  0  0  0]
 [ 0  0  0  0  0 23  0  0  0 10]
 [ 0  1  0  0  0  0 34  0  0  0]
 [ 0  0  0  0  0  0  0 30  3  0]
 [ 0  3  0  0  0  0  0  0 25  0]
 [ 0  0  0  0  2  1  0  2  2 27]]
Iteration 1 ___________________________________________________________________________
Label Spreading model: 45 labeled & 285 unlabeled (330 total)
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        22
           1       0.79      1.00      0.88        22
           2       1.00      0.93      0.96        29
           3       1.00      1.00      1.00        26
           4       0.92      0.96      0.94        23
           5       0.96      0.70      0.81        33
           6       1.00      0.97      0.99        35
           7       0.94      0.91      0.92        33
           8       0.77      0.86      0.81        28
           9       0.73      0.79      0.76        34

    accuracy                           0.90       285
   macro avg       0.91      0.91      0.91       285
weighted avg       0.91      0.90      0.90       285

Confusion matrix
[[22  0  0  0  0  0  0  0  0  0]
 [ 0 22  0  0  0  0  0  0  0  0]
 [ 0  0 27  0  0  0  0  0  2  0]
 [ 0  0  0 26  0  0  0  0  0  0]
 [ 0  1  0  0 22  0  0  0  0  0]
 [ 0  0  0  0  0 23  0  0  0 10]
 [ 0  1  0  0  0  0 34  0  0  0]
 [ 0  0  0  0  0  0  0 30  3  0]
 [ 0  4  0  0  0  0  0  0 24  0]
 [ 0  0  0  0  2  1  0  2  2 27]]
Iteration 2 ___________________________________________________________________________
Label Spreading model: 50 labeled & 280 unlabeled (330 total)
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        22
           1       0.85      1.00      0.92        22
           2       1.00      1.00      1.00        28
           3       1.00      1.00      1.00        26
           4       0.87      1.00      0.93        20
           5       0.96      0.70      0.81        33
           6       1.00      0.97      0.99        35
           7       0.94      1.00      0.97        32
           8       0.92      0.86      0.89        28
           9       0.73      0.79      0.76        34

    accuracy                           0.92       280
   macro avg       0.93      0.93      0.93       280
weighted avg       0.93      0.92      0.92       280

Confusion matrix
[[22  0  0  0  0  0  0  0  0  0]
```

(continues on next page)

(continued from previous page)

```
 [ 0 22  0  0  0  0  0  0  0  0]
 [ 0  0 28  0  0  0  0  0  0  0]
 [ 0  0  0 26  0  0  0  0  0  0]
 [ 0  0  0  0 20  0  0  0  0  0]
 [ 0  0  0  0  0 23  0  0  0 10]
 [ 0  1  0  0  0  0 34  0  0  0]
 [ 0  0  0  0  0  0  0 32  0  0]
 [ 0  3  0  0  1  0  0  0 24  0]
 [ 0  0  0  0  2  1  0  2  2 27]]
Iteration 3 ______________________________________________________________________
Label Spreading model: 55 labeled & 275 unlabeled (330 total)
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        22
           1       0.85      1.00      0.92        22
           2       1.00      1.00      1.00        27
           3       1.00      1.00      1.00        26
           4       0.87      1.00      0.93        20
           5       0.96      0.87      0.92        31
           6       1.00      0.97      0.99        35
           7       1.00      1.00      1.00        31
           8       0.92      0.86      0.89        28
           9       0.88      0.85      0.86        33

    accuracy                           0.95       275
   macro avg       0.95      0.95      0.95       275
weighted avg       0.95      0.95      0.95       275

Confusion matrix
[[22  0  0  0  0  0  0  0  0  0]
 [ 0 22  0  0  0  0  0  0  0  0]
 [ 0  0 27  0  0  0  0  0  0  0]
 [ 0  0  0 26  0  0  0  0  0  0]
 [ 0  0  0  0 20  0  0  0  0  0]
 [ 0  0  0  0  0 27  0  0  0  4]
 [ 0  1  0  0  0  0 34  0  0  0]
 [ 0  0  0  0  0  0  0 31  0  0]
 [ 0  3  0  0  1  0  0  0 24  0]
 [ 0  0  0  0  2  1  0  0  2 28]]
Iteration 4 ______________________________________________________________________
Label Spreading model: 60 labeled & 270 unlabeled (330 total)
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        22
           1       0.96      1.00      0.98        22
           2       1.00      0.96      0.98        27
           3       0.96      1.00      0.98        25
           4       0.86      1.00      0.93        19
           5       0.96      0.87      0.92        31
           6       1.00      0.97      0.99        35
           7       1.00      1.00      1.00        31
           8       0.92      0.96      0.94        25
           9       0.88      0.85      0.86        33

    accuracy                           0.96       270
   macro avg       0.95      0.96      0.96       270
weighted avg       0.96      0.96      0.96       270
```

(continues on next page)

(continued from previous page)

```
Confusion matrix
[[22  0  0  0  0  0  0  0  0  0]
 [ 0 22  0  0  0  0  0  0  0  0]
 [ 0  0 26  1  0  0  0  0  0  0]
 [ 0  0  0 25  0  0  0  0  0  0]
 [ 0  0  0  0 19  0  0  0  0  0]
 [ 0  0  0  0  0 27  0  0  0  4]
 [ 0  1  0  0  0  0 34  0  0  0]
 [ 0  0  0  0  0  0  0 31  0  0]
 [ 0  0  0  0  1  0  0  0 24  0]
 [ 0  0  0  0  2  1  0  0  2 28]]
```

```
print(__doc__)

# Authors: Clay Woolam <clay@woolam.org>
# License: BSD

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

from sklearn import datasets
from sklearn.semi_supervised import LabelSpreading
from sklearn.metrics import classification_report, confusion_matrix

digits = datasets.load_digits()
rng = np.random.RandomState(0)
indices = np.arange(len(digits.data))
rng.shuffle(indices)

X = digits.data[indices[:330]]
y = digits.target[indices[:330]]
images = digits.images[indices[:330]]

n_total_samples = len(y)
n_labeled_points = 40
max_iterations = 5

unlabeled_indices = np.arange(n_total_samples)[n_labeled_points:]
f = plt.figure()

for i in range(max_iterations):
    if len(unlabeled_indices) == 0:
        print("No unlabeled items left to label.")
        break
    y_train = np.copy(y)
    y_train[unlabeled_indices] = -1

    lp_model = LabelSpreading(gamma=0.25, max_iter=20)
    lp_model.fit(X, y_train)
```

(continues on next page)

(continued from previous page)

```
    predicted_labels = lp_model.transduction_[unlabeled_indices]
    true_labels = y[unlabeled_indices]

    cm = confusion_matrix(true_labels, predicted_labels,
                          labels=lp_model.classes_)

    print("Iteration %i %s" % (i, 70 * "_"))
    print("Label Spreading model: %d labeled & %d unlabeled (%d total)"
          % (n_labeled_points, n_total_samples - n_labeled_points,
             n_total_samples))

    print(classification_report(true_labels, predicted_labels))

    print("Confusion matrix")
    print(cm)

    # compute the entropies of transduced label distributions
    pred_entropies = stats.distributions.entropy(
        lp_model.label_distributions_.T)

    # select up to 5 digit examples that the classifier is most uncertain about
    uncertainty_index = np.argsort(pred_entropies)[::-1]
    uncertainty_index = uncertainty_index[
        np.in1d(uncertainty_index, unlabeled_indices)][:5]

    # keep track of indices that we get labels for
    delete_indices = np.array([], dtype=int)

    # for more than 5 iterations, visualize the gain only on the first 5
    if i < 5:
        f.text(.05, (1 - (i + 1) * .183),
               "model %d\n\nfit with\n%d labels" %
               ((i + 1), i * 5 + 10), size=10)
    for index, image_index in enumerate(uncertainty_index):
        image = images[image_index]

        # for more than 5 iterations, visualize the gain only on the first 5
        if i < 5:
            sub = f.add_subplot(5, 5, index + 1 + (5 * i))
            sub.imshow(image, cmap=plt.cm.gray_r, interpolation='none')
            sub.set_title("predict: %i\ntrue: %i" % (
                lp_model.transduction_[image_index], y[image_index]), size=10)
            sub.axis('off')

        # labeling 5 points, remote from labeled set
        delete_index, = np.where(unlabeled_indices == image_index)
        delete_indices = np.concatenate((delete_indices, delete_index))

    unlabeled_indices = np.delete(unlabeled_indices, delete_indices)
    n_labeled_points += len(uncertainty_index)

f.suptitle("Active learning with Label Propagation.\nRows show 5 most "
           "uncertain labels to learn with the next model.", y=1.15)
plt.subplots_adjust(left=0.2, bottom=0.03, right=0.9, top=0.9, wspace=0.2,
                    hspace=0.85)
plt.show()
```

Total running time of the script: (0 minutes 0.756 seconds)

Estimated memory usage: 8 MB

6.28 Support Vector Machines

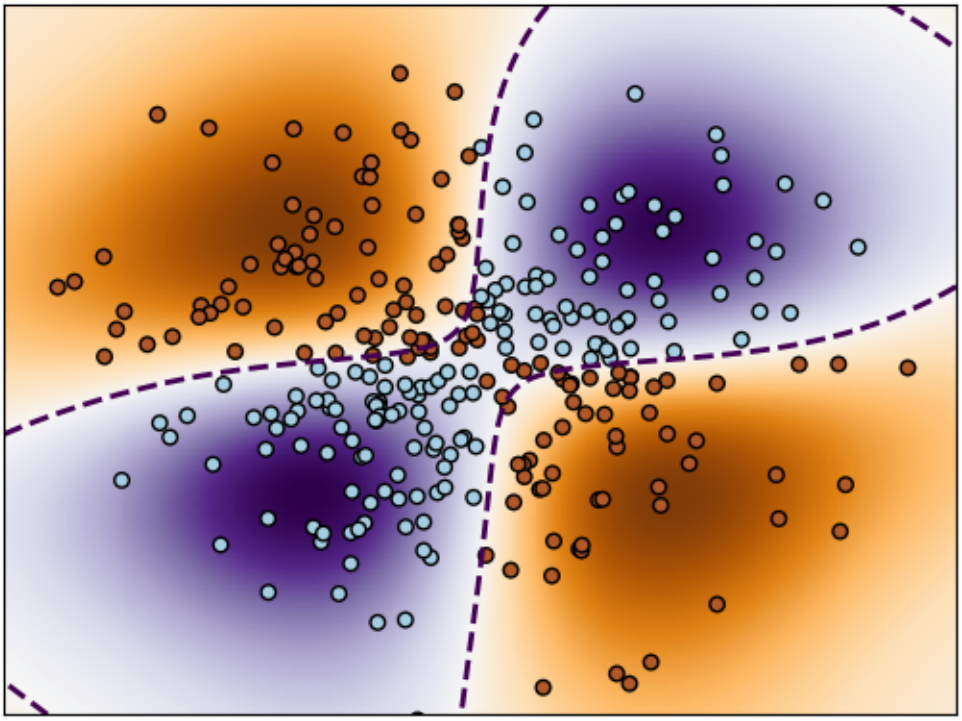
Examples concerning the *sklearn.svm* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.28.1 Non-linear SVM

Perform binary classification using non-linear SVC with RBF kernel. The target to predict is a XOR of the inputs.

The color map illustrates the decision function learned by the SVC.



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
```

(continues on next page)

(continued from previous page)

```
xx, yy = np.meshgrid(np.linspace(-3, 3, 500),
                     np.linspace(-3, 3, 500))
np.random.seed(0)
X = np.random.randn(300, 2)
Y = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)

# fit the model
clf = svm.NuSVC(gamma='auto')
clf.fit(X, Y)

# plot the decision function for each datapoint on the grid
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.imshow(Z, interpolation='nearest',
           extent=(xx.min(), xx.max(), yy.min(), yy.max()), aspect='auto',
           origin='lower', cmap=plt.cm.PuOr_r)
contours = plt.contour(xx, yy, Z, levels=[0], linewidths=2,
                       linestyles='dashed')
plt.scatter(X[:, 0], X[:, 1], s=30, c=Y, cmap=plt.cm.Paired,
            edgecolors='k')
plt.xticks(())
plt.yticks(())
plt.axis([-3, 3, -3, 3])
plt.show()
```

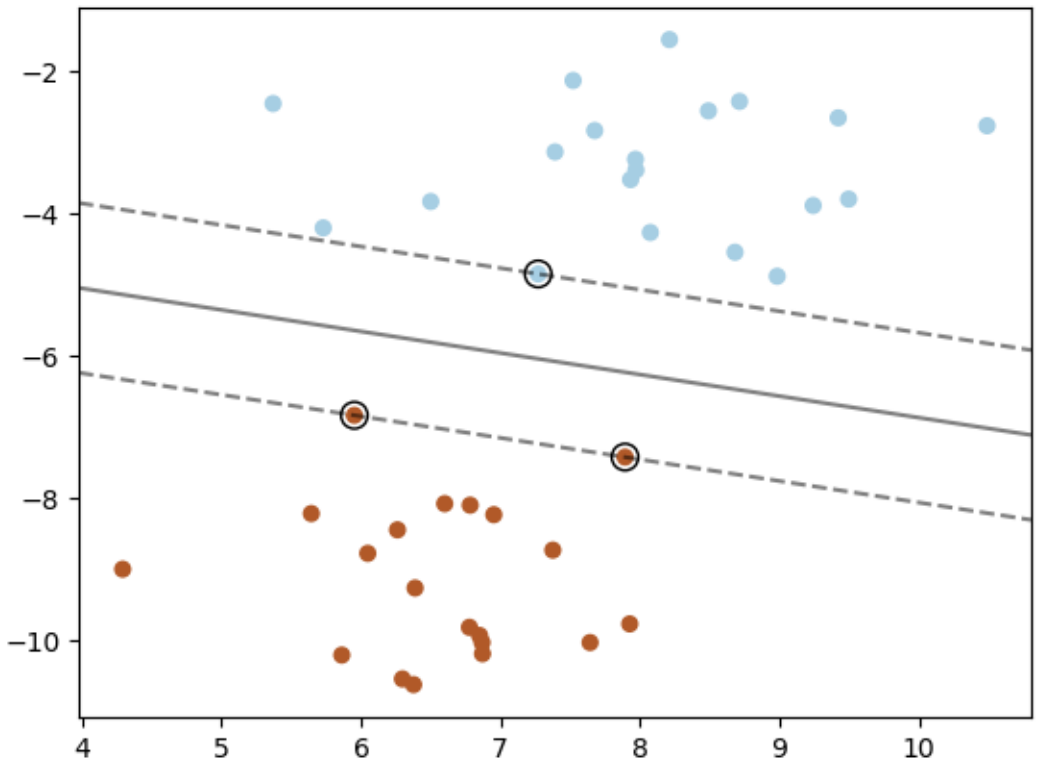
Total running time of the script: (0 minutes 1.300 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.28.2 SVM: Maximum margin separating hyperplane

Plot the maximum margin separating hyperplane within a two-class separable dataset using a Support Vector Machine classifier with linear kernel.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.datasets import make_blobs


# we create 40 separable points
X, y = make_blobs(n_samples=40, centers=2, random_state=6)

# fit the model, don't regularize for illustration purposes
clf = svm.SVC(kernel='linear', C=1000)
clf.fit(X, y)

plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)

# plot the decision function
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# create grid to evaluate model
xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
```

(continues on next page)

(continued from previous page)

```
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = clf.decision_function(xy).reshape(XX.shape)

# plot decision boundary and margins
ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
           linestyles=['--', '-', '--'])
# plot support vectors
ax.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=100,
           linewidth=1, facecolors='none', edgecolors='k')
plt.show()
```

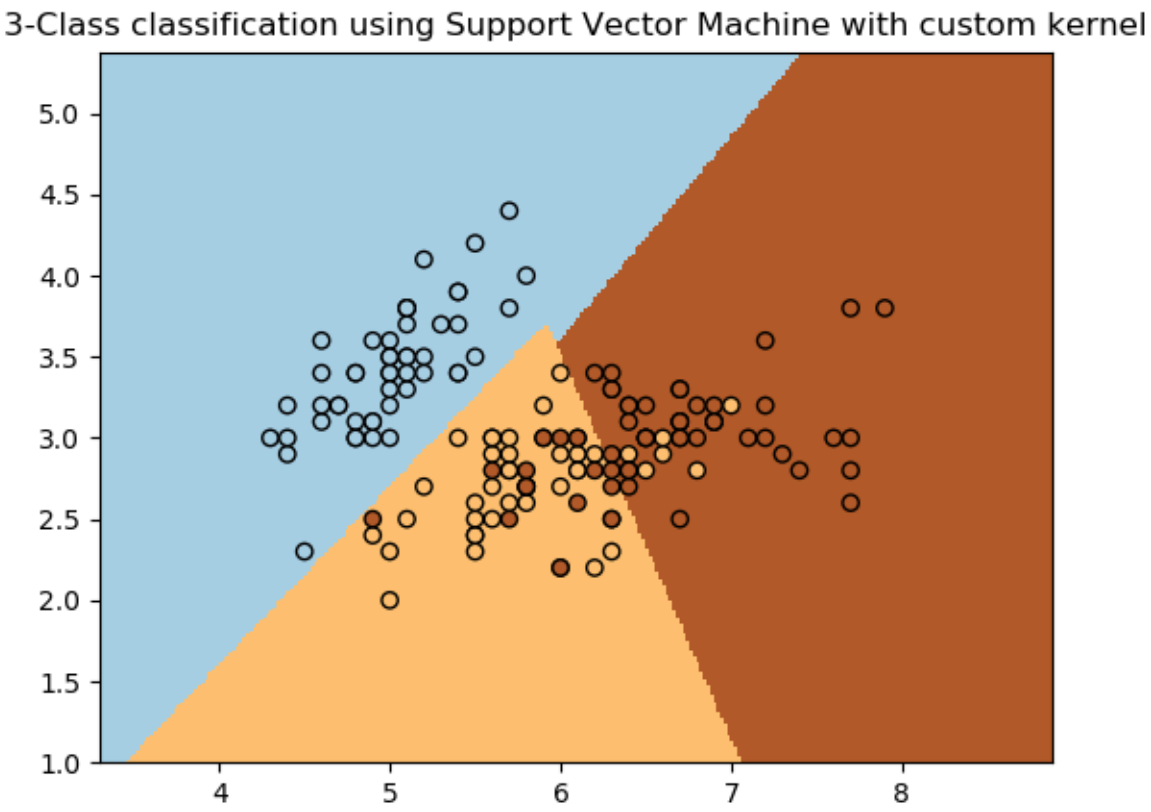
Total running time of the script: (0 minutes 0.549 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.28.3 SVM with custom kernel

Simple usage of Support Vector Machines to classify a sample. It will plot the decision surface and the support vectors.



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2]  # we only take the first two features. We could
                      # avoid this ugly slicing by using a two-dim dataset
Y = iris.target


def my_kernel(X, Y):
    """
    We create a custom kernel:

                 (2  0)
    k(X, Y) = X  (    ) Y.T
                 (0  1)
    """
    M = np.array([[2, 0], [0, 1.0]])
    return np.dot(np.dot(X, M), Y.T)


h = .02  # step size in the mesh

# we create an instance of SVM and fit out data.
clf = svm.SVC(kernel=my_kernel)
clf.fit(X, Y)

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=Y, cmap=plt.cm.Paired, edgecolors='k')
plt.title('3-Class classification using Support Vector Machine with custom'
          ' kernel')
plt.axis('tight')
plt.show()
```

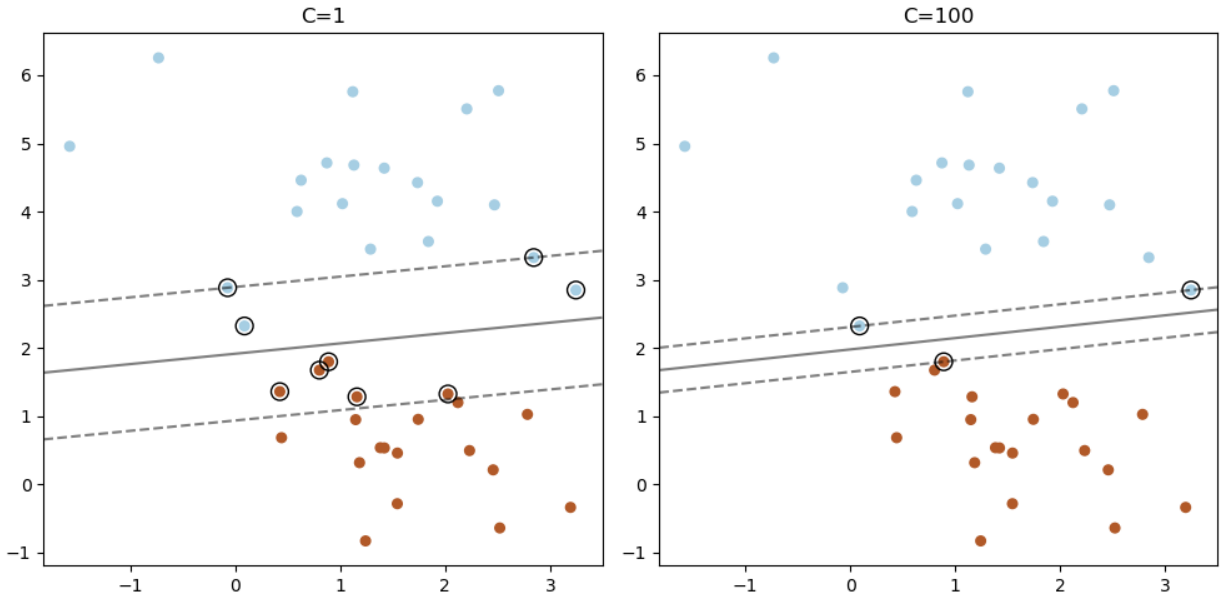
Total running time of the script: (0 minutes 0.679 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.28.4 Plot the support vectors in LinearSVC

Unlike SVC (based on LIBSVM), LinearSVC (based on LIBLINEAR) does not provide the support vectors. This example demonstrates how to obtain the support vectors in LinearSVC.




```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.svm import LinearSVC

X, y = make_blobs(n_samples=40, centers=2, random_state=0)

plt.figure(figsize=(10, 5))
for i, C in enumerate([1, 100]):
    # "hinge" is the standard SVM loss
    clf = LinearSVC(C=C, loss="hinge", random_state=42).fit(X, y)
    # obtain the support vectors through the decision function
    decision_function = clf.decision_function(X)
    # we can also calculate the decision function manually
    # decision_function = np.dot(X, clf.coef_[0]) + clf.intercept_[0]
    support_vector_indices = np.where((2 * y - 1) * decision_function <= 1)[0]
    support_vectors = X[support_vector_indices]

    plt.subplot(1, 2, i + 1)
    plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)
    ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()
    xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 50),
                         np.linspace(ylim[0], ylim[1], 50))
    Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contour(xx, yy, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
                linestyles=['--', '-', '--'])
    plt.scatter(support_vectors[:, 0], support_vectors[:, 1], s=100,
                linewidth=1, facecolors='none', edgecolors='k')
```

(continues on next page)

(continued from previous page)

```
    plt.title("C=" + str(C))
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.634 seconds)

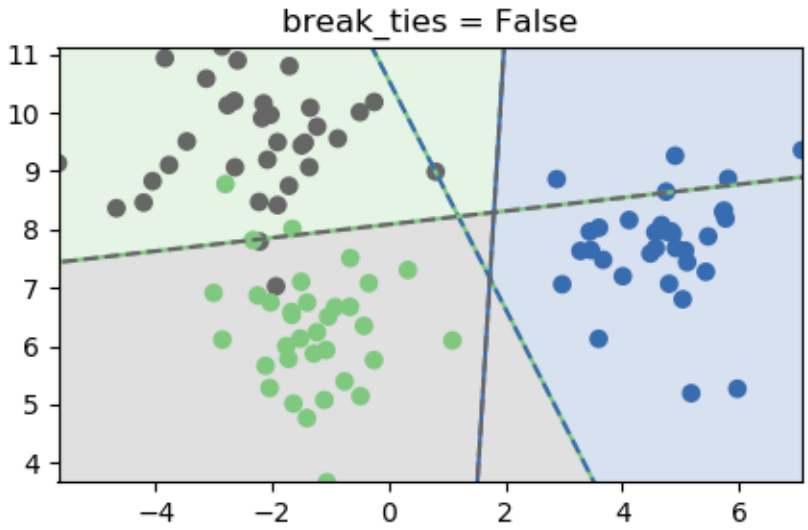
Estimated memory usage: 8 MB

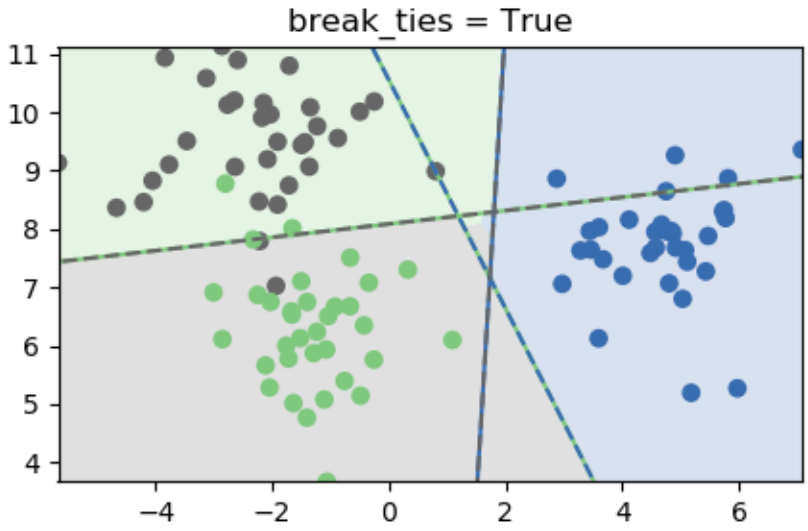
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.28.5 SVM Tie Breaking Example

Tie breaking is costly if `decision_function_shape='ovr'`, and therefore it is not enabled by default. This example illustrates the effect of the `break_ties` parameter for a multiclass classification problem and `decision_function_shape='ovr'`.

The two plots differ only in the area in the middle where the classes are tied. If `break_ties=False`, all input in that area would be classified as one class, whereas if `break_ties=True`, the tie-breaking mechanism will create a non-convex decision boundary in that area.






```
print(__doc__)


# Code source: Andreas Mueller, Adrin Jalali
```

(continues on next page)

(continued from previous page)

```
# License: BSD 3 clause


import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.datasets import make_blobs

X, y = make_blobs(random_state=27)

fig, sub = plt.subplots(2, 1, figsize=(5, 8))
titles = ("break_ties = False",
          "break_ties = True")

for break_ties, title, ax in zip((False, True), titles, sub.flatten()):

    svm = SVC(kernel="linear", C=1, break_ties=break_ties,
              decision_function_shape='ovr').fit(X, y)

    xlim = [X[:, 0].min(), X[:, 0].max()]
    ylim = [X[:, 1].min(), X[:, 1].max()]

    xs = np.linspace(xlim[0], xlim[1], 1000)
    ys = np.linspace(ylim[0], ylim[1], 1000)
    xx, yy = np.meshgrid(xs, ys)

    pred = svm.predict(np.c_[xx.ravel(), yy.ravel()])

    colors = [plt.cm.Accent(i) for i in [0, 4, 7]]

    points = ax.scatter(X[:, 0], X[:, 1], c=y, cmap="Accent")
    classes = [(0, 1), (0, 2), (1, 2)]
    line = np.linspace(X[:, 1].min() - 5, X[:, 1].max() + 5)
    ax.imshow(-pred.reshape(xx.shape), cmap="Accent", alpha=.2,
              extent=(xlim[0], xlim[1], ylim[1], ylim[0]))

    for coef, intercept, col in zip(svm.coef_, svm.intercept_, classes):
        line2 = -(line * coef[1] + intercept) / coef[0]
        ax.plot(line2, line, "-", c=colors[col[0]])
        ax.plot(line2, line, "--", c=colors[col[1]])
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)
    ax.set_title(title)
    ax.set_aspect("equal")

plt.show()
```

Total running time of the script: (0 minutes 1.207 seconds)

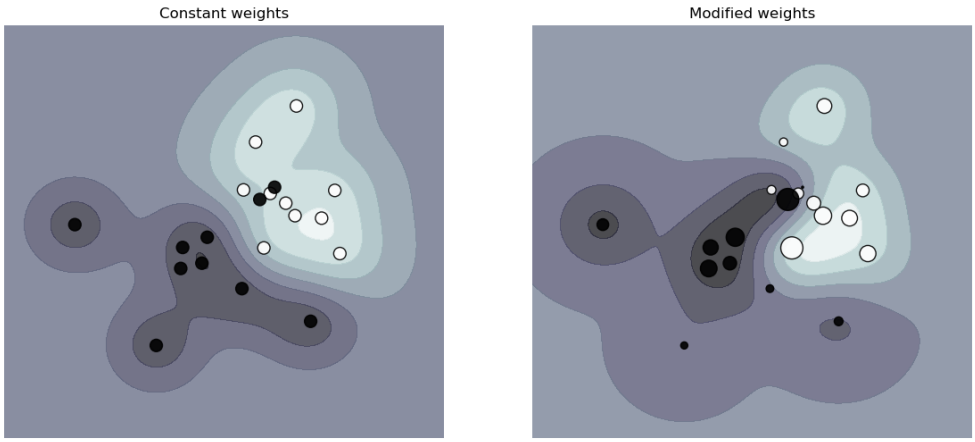
Estimated memory usage: 86 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.28.6 SVM: Weighted samples

Plot decision function of a weighted dataset, where the size of points is proportional to its weight.

The sample weighting rescales the C parameter, which means that the classifier puts more emphasis on getting these points right. The effect might often be subtle. To emphasize the effect here, we particularly weight outliers, making the deformation of the decision boundary very visible.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm


def plot_decision_function(classifier, sample_weight, axis, title):
    # plot the decision function
    xx, yy = np.meshgrid(np.linspace(-4, 5, 500), np.linspace(-4, 5, 500))

    Z = classifier.decision_function(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # plot the line, the points, and the nearest vectors to the plane
    axis.contourf(xx, yy, Z, alpha=0.75, cmap=plt.cm.bone)
    axis.scatter(X[:, 0], X[:, 1], c=y, s=100 * sample_weight, alpha=0.9,
                 cmap=plt.cm.bone, edgecolors='black')

    axis.axis('off')
    axis.set_title(title)


# we create 20 points
np.random.seed(0)
X = np.r_[np.random.randn(10, 2) + [1, 1], np.random.randn(10, 2)]
y = [1] * 10 + [-1] * 10
sample_weight_last_ten = abs(np.random.randn(len(X)))
sample_weight_constant = np.ones(len(X))
# and bigger weights to some outliers
sample_weight_last_ten[15:] *= 5
```

(continues on next page)

(continued from previous page)

```
sample_weight_last_ten[9] *= 15

# for reference, first fit without sample weights

# fit the model
clf_weights = svm.SVC(gamma=1)
clf_weights.fit(X, y, sample_weight=sample_weight_last_ten)

clf_no_weights = svm.SVC(gamma=1)
clf_no_weights.fit(X, y)

fig, axes = plt.subplots(1, 2, figsize=(14, 6))
plot_decision_function(clf_no_weights, sample_weight_constant, axes[0],
                       "Constant weights")
plot_decision_function(clf_weights, sample_weight_last_ten, axes[1],
                       "Modified weights")

plt.show()
```

Total running time of the script: (0 minutes 0.670 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.28.7 SVM: Separating hyperplane for unbalanced classes

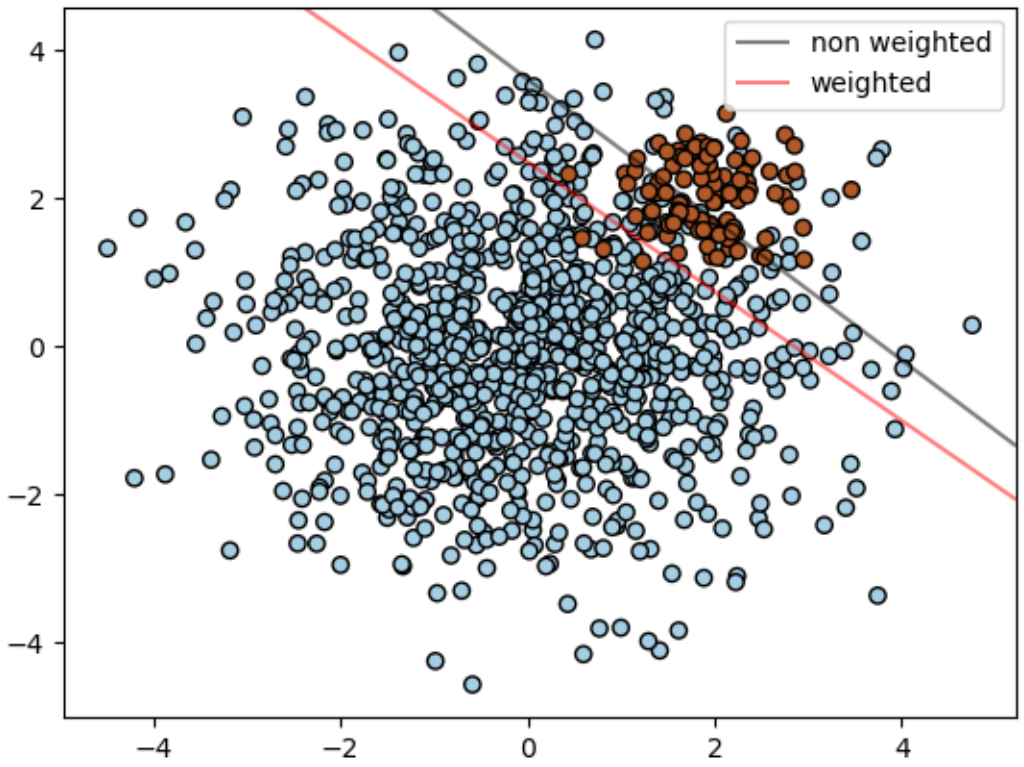
Find the optimal separating hyperplane using an SVC for classes that are unbalanced.

We first find the separating plane with a plain SVC and then plot (dashed) the separating hyperplane with automatically correction for unbalanced classes.

Note: This example will also work by replacing `SVC(kernel="linear")` with `SGDClassifier(loss="hinge")`. Setting the `loss` parameter of the *SGDClassifier* equal to `hinge` will yield behaviour such as that of a SVC with a linear kernel.

For example try instead of the `SVC`:

```
clf = SGDClassifier(n_iter=100, alpha=0.01)
```




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.datasets import make_blobs

# we create two clusters of random points
n_samples_1 = 1000
n_samples_2 = 100
centers = [[0.0, 0.0], [2.0, 2.0]]
clusters_std = [1.5, 0.5]
X, y = make_blobs(n_samples=[n_samples_1, n_samples_2],
                  centers=centers,
                  cluster_std=clusters_std,
                  random_state=0, shuffle=False)

# fit the model and get the separating hyperplane
clf = svm.SVC(kernel='linear', C=1.0)
clf.fit(X, y)

# fit the model and get the separating hyperplane using weighted classes
wclf = svm.SVC(kernel='linear', class_weight={1: 10})
wclf.fit(X, y)
```

(continues on next page)

(continued from previous page)

```
# plot the samples
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired, edgecolors='k')

# plot the decision functions for both classifiers
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# create grid to evaluate model
xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T

# get the separating hyperplane
Z = clf.decision_function(xy).reshape(XX.shape)

# plot decision boundary and margins
a = ax.contour(XX, YY, Z, colors='k', levels=[0], alpha=0.5, linestyles=['-'])

# get the separating hyperplane for weighted classes
Z = wclf.decision_function(xy).reshape(XX.shape)

# plot decision boundary and margins for weighted classes
b = ax.contour(XX, YY, Z, colors='r', levels=[0], alpha=0.5, linestyles=['-'])

plt.legend([a.collections[0], b.collections[0]], ["non weighted", "weighted"],
           loc="upper right")
plt.show()
```

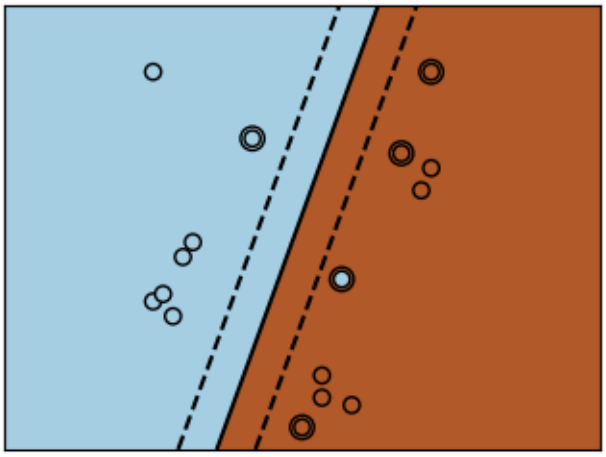
Total running time of the script: (0 minutes 0.634 seconds)

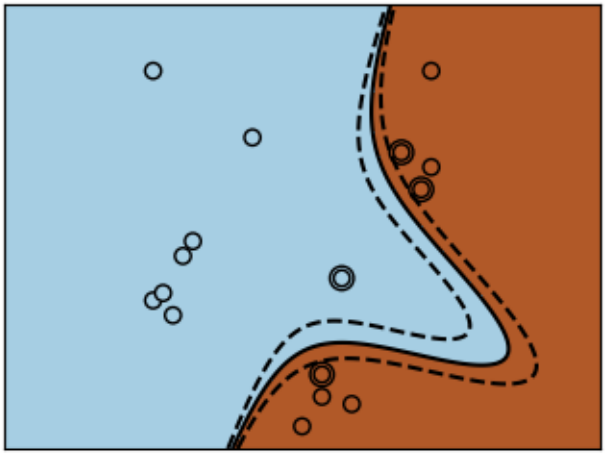
Estimated memory usage: 8 MB

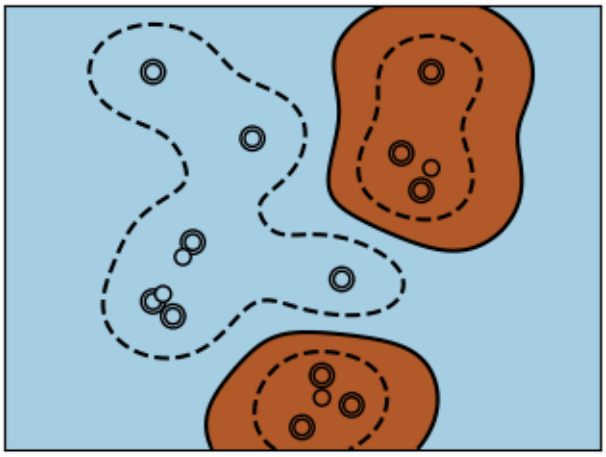
Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.28.8 SVM-Kernels

Three different types of SVM-Kernels are displayed below. The polynomial and RBF are especially useful when the data-points are not linearly separable.







-

```
print(__doc__)


# Code source: Gaël Varoquaux
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm


# Our dataset and targets
X = np.c_[(.4, -.7),
          (-1.5, -1),
          (-1.4, -.9),
          (-1.3, -1.2),
          (-1.1, -.2),
          (-1.2, -.4),
          (-.5, 1.2),
          (-1.5, 2.1),
          (1, 1),
          # --
          (1.3, .8),
          (1.2, .5),
          (.2, -2),
          (.5, -2.4),
          (.2, -2.3),
          (0, -2.7),
          (1.3, 2.1)].T
Y = [0] * 8 + [1] * 8

# figure number
fignum = 1

# fit the model
for kernel in ('linear', 'poly', 'rbf'):
    clf = svm.SVC(kernel=kernel, gamma=2)
```

(continues on next page)

(continued from previous page)

```
    clf.fit(X, Y)

    # plot the line, the points, and the nearest vectors to the plane
    plt.figure(fignum, figsize=(4, 3))
    plt.clf()

    plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=80,
                facecolors='none', zorder=10, edgecolors='k')
    plt.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=plt.cm.Paired,
                edgecolors='k')

    plt.axis('tight')
    x_min = -3
    x_max = 3
    y_min = -3
    y_max = 3

    XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
    Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(XX.shape)
    plt.figure(fignum, figsize=(4, 3))
    plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)
    plt.contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '-', '--'],
                levels=[-.5, 0, .5])

    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)

    plt.xticks(())
    plt.yticks(())
    fignum = fignum + 1
plt.show()
```

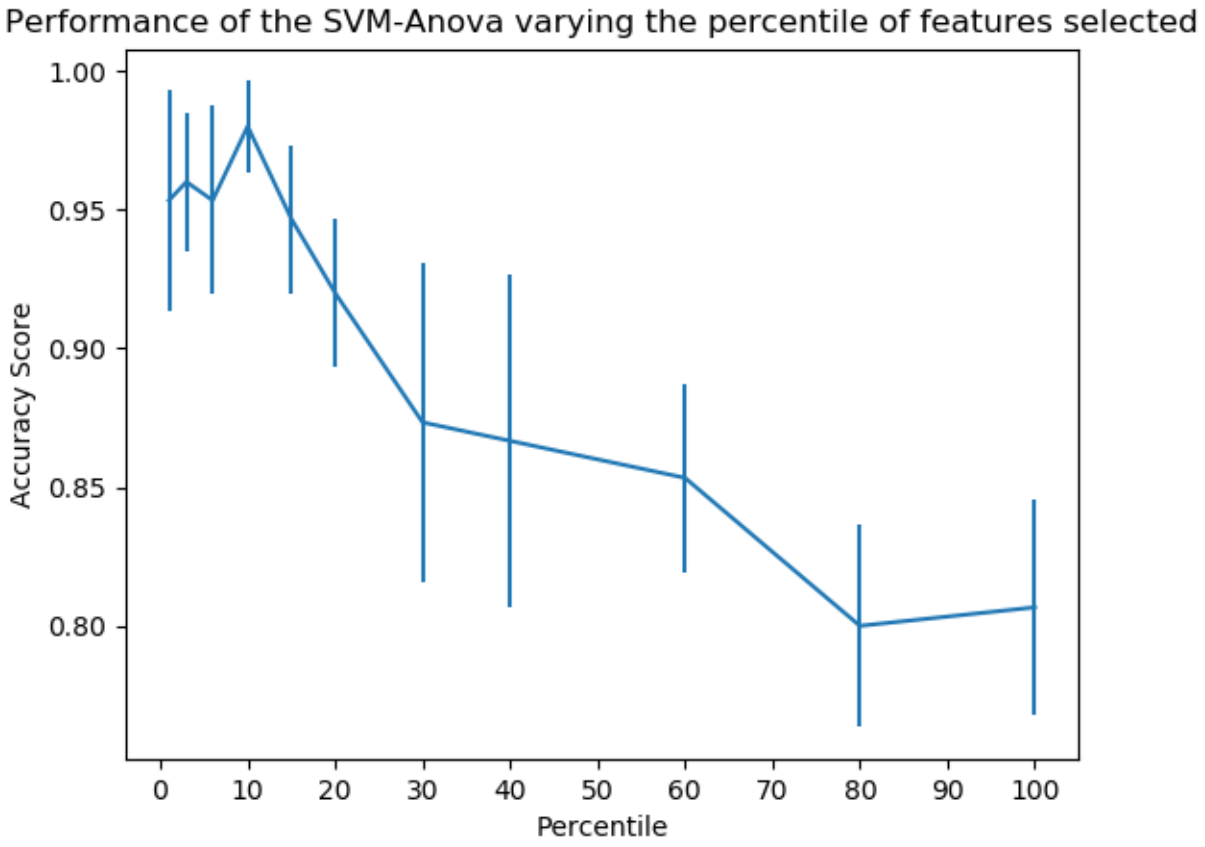
Total running time of the script: (0 minutes 0.679 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.28.9 SVM-Anova: SVM with univariate feature selection

This example shows how to perform univariate feature selection before running a SVC (support vector classifier) to improve the classification scores. We use the iris dataset (4 features) and add 36 non-informative features. We can find that our model achieves best performance when we select around 10% of features.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectPercentile, chi2
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC


# #############################################################################
# Import some data to play with
X, y = load_iris(return_X_y=True)
# Add non-informative features
np.random.seed(0)
X = np.hstack((X, 2 * np.random.random((X.shape[0], 36))))

# #############################################################################
# Create a feature-selection transform, a scaler and an instance of SVM that we
# combine together to have an full-blown estimator
clf = Pipeline([('anova', SelectPercentile(chi2)),
                ('scaler', StandardScaler()),
                ('svc', SVC(gamma="auto"))])
```

(continues on next page)

(continued from previous page)

```
# ##############################################################################
# Plot the cross-validation score as a function of percentile of features
score_means = list()
score_stds = list()
percentiles = (1, 3, 6, 10, 15, 20, 30, 40, 60, 80, 100)

for percentile in percentiles:
    clf.set_params(anova__percentile=percentile)
    this_scores = cross_val_score(clf, X, y)
    score_means.append(this_scores.mean())
    score_stds.append(this_scores.std())

plt.errorbar(percentiles, score_means, np.array(score_stds))
plt.title(
    'Performance of the SVM-Anova varying the percentile of features selected')
plt.xticks(np.linspace(0, 100, 11, endpoint=True))
plt.xlabel('Percentile')
plt.ylabel('Accuracy Score')
plt.axis('tight')
plt.show()
```

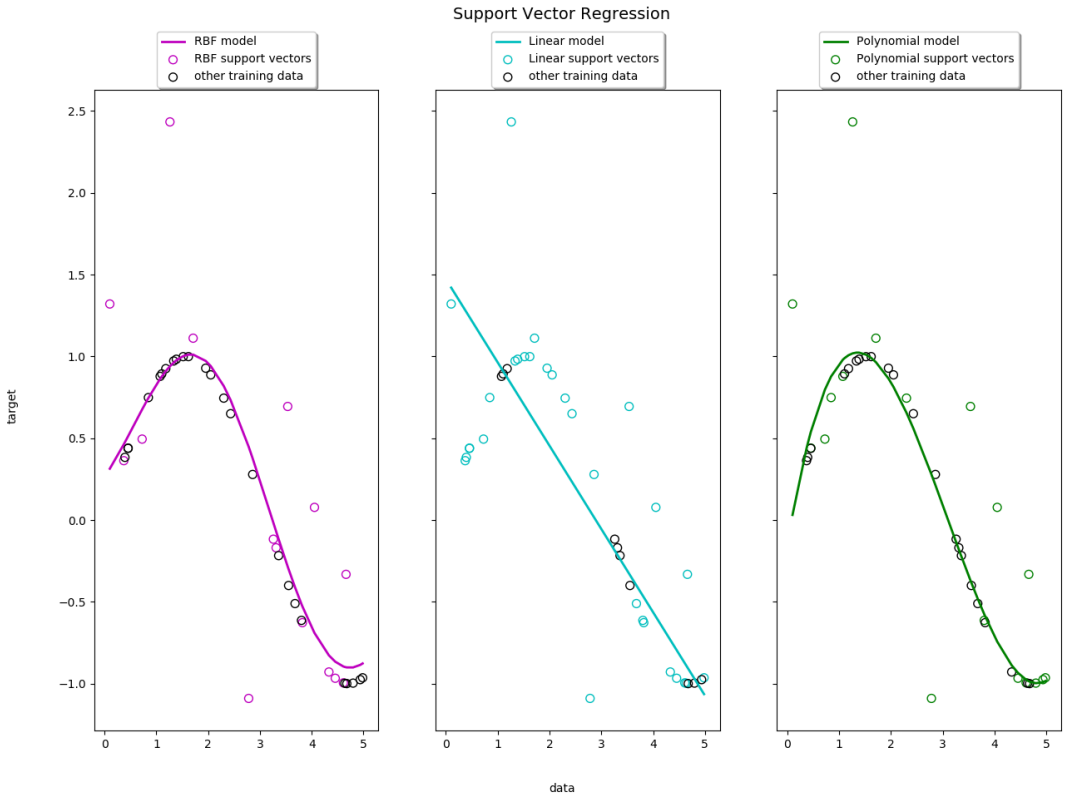
Total running time of the script: (0 minutes 0.564 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.28.10 Support Vector Regression (SVR) using linear and non-linear kernels

Toy example of 1D regression using linear, polynomial and RBF kernels.




```
print(__doc__)

import numpy as np
from sklearn.svm import SVR
import matplotlib.pyplot as plt

# ##############################################################################
# Generate sample data
X = np.sort(5 * np.random.rand(40, 1), axis=0)
y = np.sin(X).ravel()

# ##############################################################################
# Add noise to targets
y[::5] += 3 * (0.5 - np.random.rand(8))

# ##############################################################################
# Fit regression model
svr_rbf = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=.1)
svr_lin = SVR(kernel='linear', C=100, gamma='auto')
svr_poly = SVR(kernel='poly', C=100, gamma='auto', degree=3, epsilon=.1,
               coef0=1)

# ##############################################################################
# Look at the results
lw = 2

svrs = [svr_rbf, svr_lin, svr_poly]
kernel_label = ['RBF', 'Linear', 'Polynomial']
```

(continues on next page)

(continued from previous page)

```
model_color = ['m', 'c', 'g']

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(15, 10), sharey=True)
for ix, svr in enumerate(svrs):
    axes[ix].plot(X, svr.fit(X, y).predict(X), color=model_color[ix], lw=lw,
                  label='{} model'.format(kernel_label[ix]))
    axes[ix].scatter(X[svr.support_], y[svr.support_], facecolor="none",
                     edgecolor=model_color[ix], s=50,
                     label='{} support vectors'.format(kernel_label[ix]))
    axes[ix].scatter(X[np.setdiff1d(np.arange(len(X)), svr.support_)],
                     y[np.setdiff1d(np.arange(len(X)), svr.support_)],
                     facecolor="none", edgecolor="k", s=50,
                     label='other training data')
    axes[ix].legend(loc='upper center', bbox_to_anchor=(0.5, 1.1),
                    ncol=1, fancybox=True, shadow=True)

fig.text(0.5, 0.04, 'data', ha='center', va='center')
fig.text(0.06, 0.5, 'target', ha='center', va='center', rotation='vertical')
fig.suptitle("Support Vector Regression", fontsize=14)
plt.show()
```

Total running time of the script: (0 minutes 3.817 seconds)

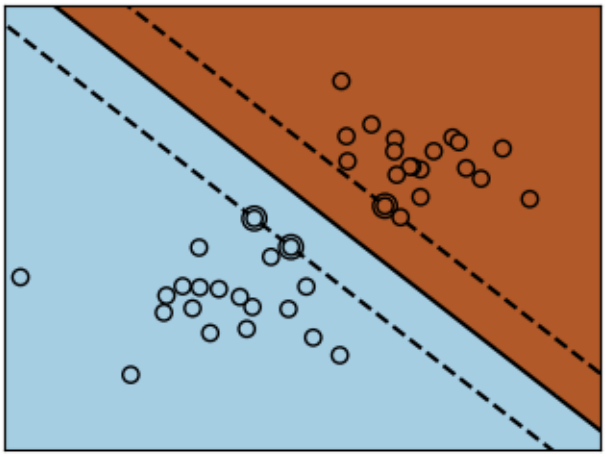
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

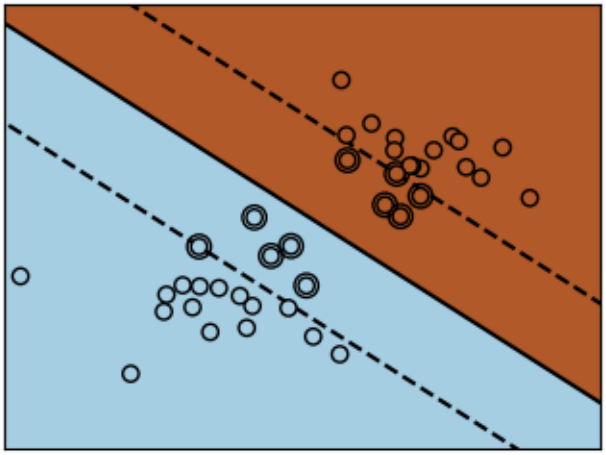
6.28.11 SVM Margins Example

The plots below illustrate the effect the parameter `C` has on the separation line. A large value of `C` basically tells our model that we do not have that much faith in our data's distribution, and will only consider points close to line of separation.

A small value of `C` includes more/all the observations, allowing the margins to be calculated using all the data in the area.



-



-

```
print(__doc__)


# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

# we create 40 separable points
np.random.seed(0)
X = np.r_[np.random.randn(20, 2) - [2, 2], np.random.randn(20, 2) + [2, 2]]
Y = [0] * 20 + [1] * 20

# figure number
```

(continues on next page)

(continued from previous page)

```
fignum = 1

# fit the model
for name, penalty in (('unreg', 1), ('reg', 0.05)):

    clf = svm.SVC(kernel='linear', C=penalty)
    clf.fit(X, Y)

    # get the separating hyperplane
    w = clf.coef_[0]
    a = -w[0] / w[1]
    xx = np.linspace(-5, 5)
    yy = a * xx - (clf.intercept_[0]) / w[1]

    # plot the parallels to the separating hyperplane that pass through the
    # support vectors (margin away from hyperplane in direction
    # perpendicular to hyperplane). This is sqrt(1+a^2) away vertically in
    # 2-d.
    margin = 1 / np.sqrt(np.sum(clf.coef_ ** 2))
    yy_down = yy - np.sqrt(1 + a ** 2) * margin
    yy_up = yy + np.sqrt(1 + a ** 2) * margin

    # plot the line, the points, and the nearest vectors to the plane
    plt.figure(fignum, figsize=(4, 3))
    plt.clf()
    plt.plot(xx, yy, 'k-')
    plt.plot(xx, yy_down, 'k--')
    plt.plot(xx, yy_up, 'k--')

    plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=80,
                facecolors='none', zorder=10, edgecolors='k')
    plt.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=plt.cm.Paired,
                edgecolors='k')

    plt.axis('tight')
    x_min = -4.8
    x_max = 4.2
    y_min = -6
    y_max = 6

    XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
    Z = clf.predict(np.c_[XX.ravel(), YY.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(XX.shape)
    plt.figure(fignum, figsize=(4, 3))
    plt.pcolormesh(XX, YY, Z, cmap=plt.cm.Paired)

    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)

    plt.xticks(())
    plt.yticks(())
    fignum = fignum + 1

plt.show()
```

Total running time of the script: (0 minutes 0.588 seconds)

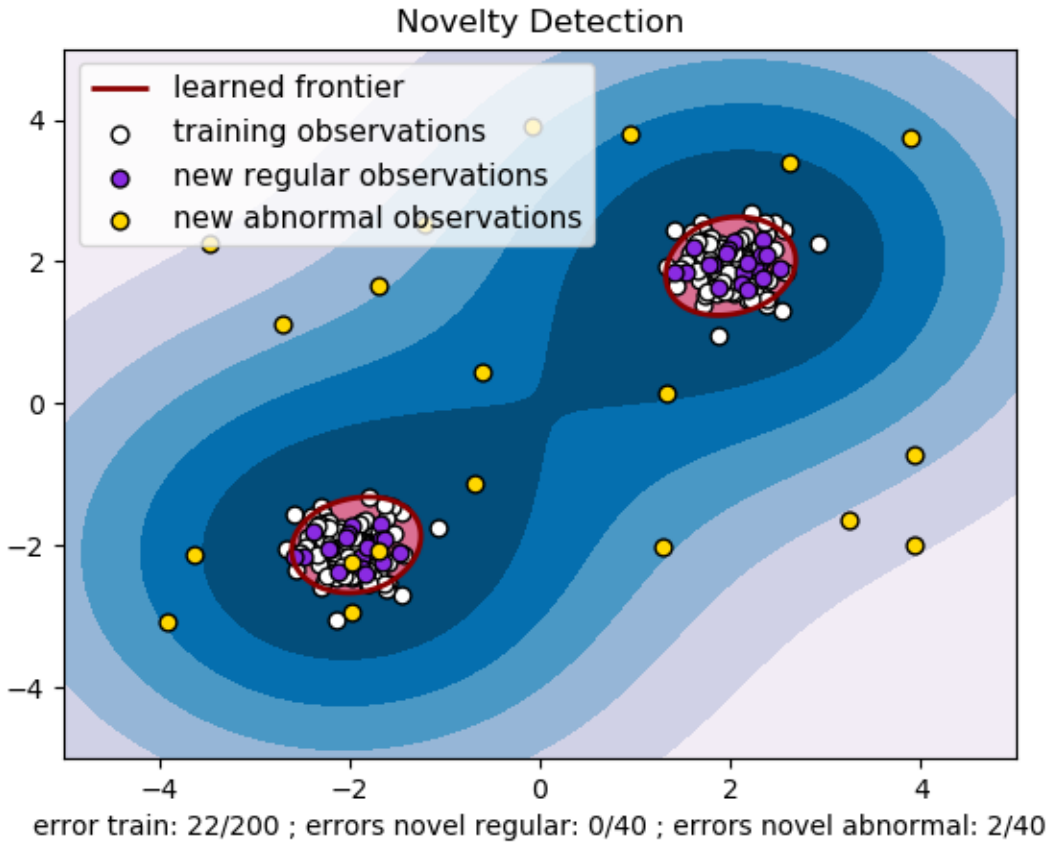
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.28.12 One-class SVM with non-linear kernel (RBF)

An example using a one-class SVM for novelty detection.

One-class SVM is an unsupervised algorithm that learns a decision function for novelty detection: classifying new data as similar or different to the training set.




```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.font_manager
from sklearn import svm

xx, yy = np.meshgrid(np.linspace(-5, 5, 500), np.linspace(-5, 5, 500))
# Generate train data
X = 0.3 * np.random.randn(100, 2)
```

(continues on next page)

(continued from previous page)

```
X_train = np.r_[X + 2, X - 2]
# Generate some regular novel observations
X = 0.3 * np.random.randn(20, 2)
X_test = np.r_[X + 2, X - 2]
# Generate some abnormal novel observations
X_outliers = np.random.uniform(low=-4, high=4, size=(20, 2))

# fit the model
clf = svm.OneClassSVM(nu=0.1, kernel="rbf", gamma=0.1)
clf.fit(X_train)
y_pred_train = clf.predict(X_train)
y_pred_test = clf.predict(X_test)
y_pred_outliers = clf.predict(X_outliers)
n_error_train = y_pred_train[y_pred_train == -1].size
n_error_test = y_pred_test[y_pred_test == -1].size
n_error_outliers = y_pred_outliers[y_pred_outliers == 1].size

# plot the line, the points, and the nearest vectors to the plane
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.title("Novelty Detection")
plt.contourf(xx, yy, Z, levels=np.linspace(Z.min(), 0, 7), cmap=plt.cm.PuBu)
a = plt.contour(xx, yy, Z, levels=[0], linewidths=2, colors='darkred')
plt.contourf(xx, yy, Z, levels=[0, Z.max()], colors='palevioletred')

s = 40
b1 = plt.scatter(X_train[:, 0], X_train[:, 1], c='white', s=s, edgecolors='k')
b2 = plt.scatter(X_test[:, 0], X_test[:, 1], c='blueviolet', s=s,
                 edgecolors='k')
c = plt.scatter(X_outliers[:, 0], X_outliers[:, 1], c='gold', s=s,
                edgecolors='k')
plt.axis('tight')
plt.xlim((-5, 5))
plt.ylim((-5, 5))
plt.legend([a.collections[0], b1, b2, c],
           ["learned frontier", "training observations",
            "new regular observations", "new abnormal observations"],
           loc="upper left",
           prop=matplotlib.font_manager.FontProperties(size=11))
plt.xlabel(
    "error train: %d/200 ; errors novel regular: %d/40 ; "
    "errors novel abnormal: %d/40"
    % (n_error_train, n_error_test, n_error_outliers))
plt.show()
```

Total running time of the script: (0 minutes 0.545 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.28.13 Plot different SVM classifiers in the iris dataset

Comparison of different linear SVM classifiers on a 2D projection of the iris dataset. We only consider the first 2 features of this dataset:

- Sepal length
- Sepal width

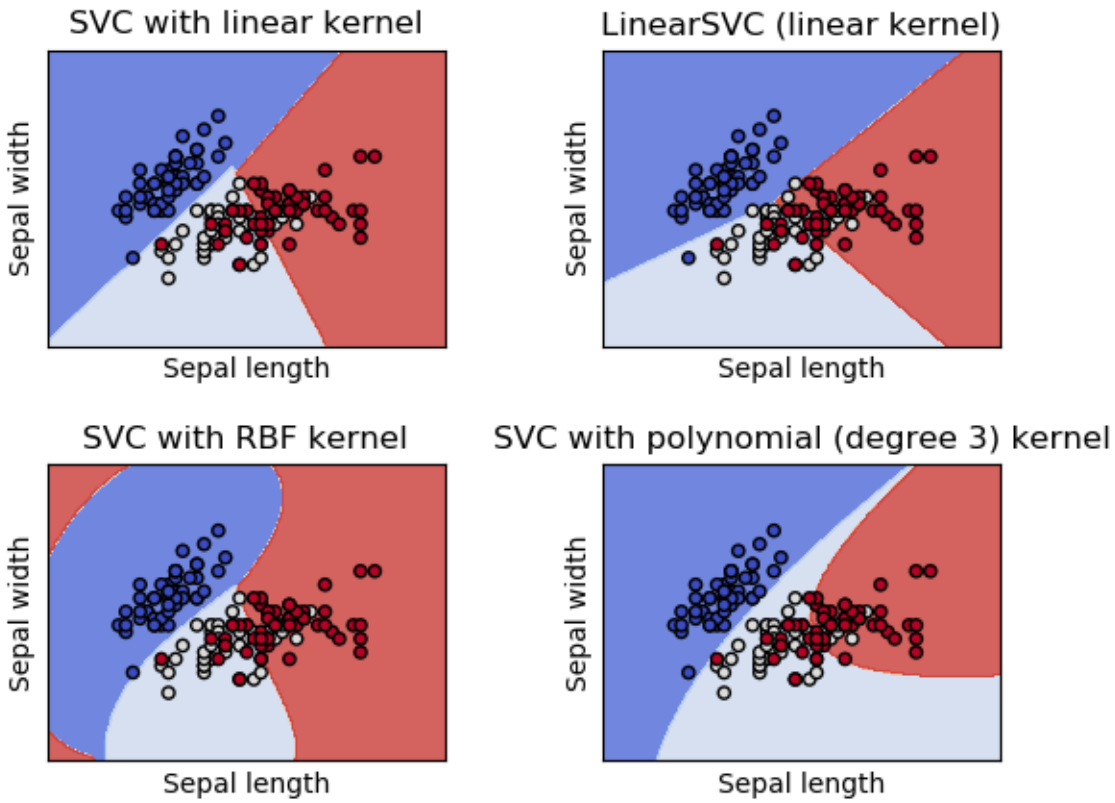
This example shows how to plot the decision surface for four SVM classifiers with different kernels.

The linear models `LinearSVC()` and `SVC(kernel='linear')` yield slightly different decision boundaries. This can be a consequence of the following differences:

- `LinearSVC` minimizes the squared hinge loss while `SVC` minimizes the regular hinge loss.
- `LinearSVC` uses the One-vs-All (also known as One-vs-Rest) multiclass reduction while `SVC` uses the One-vs-One multiclass reduction.

Both linear models have linear decision boundaries (intersecting hyperplanes) while the non-linear kernel models (polynomial or Gaussian RBF) have more flexible non-linear decision boundaries with shapes that depend on the kind of kernel and its parameters.

Note: while plotting the decision function of classifiers for toy 2D datasets can help get an intuitive understanding of their respective expressive power, be aware that those intuitions don't always generalize to more realistic high-dimensional problems.



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets


def make_meshgrid(x, y, h=.02):
    """Create a mesh of points to plot in

    Parameters
    ----------
    x: data to base x-axis meshgrid on
    y: data to base y-axis meshgrid on
    h: stepsize for meshgrid, optional

    Returns
    -------
    xx, yy : ndarray
    """
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    return xx, yy


def plot_contours(ax, clf, xx, yy, **params):
    """Plot the decision boundaries for a classifier.

    Parameters
    ----------
    ax: matplotlib axes object
    clf: a classifier
    xx: meshgrid ndarray
    yy: meshgrid ndarray
    params: dictionary of params to pass to contourf, optional
    """
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = ax.contourf(xx, yy, Z, **params)
    return out


# import some data to play with
iris = datasets.load_iris()
# Take the first two features. We could avoid this by using a two-dim dataset
X = iris.data[:, :2]
y = iris.target

# we create an instance of SVM and fit out data. We do not scale our
# data since we want to plot the support vectors
C = 1.0  # SVM regularization parameter
models = (svm.SVC(kernel='linear', C=C),
          svm.LinearSVC(C=C, max_iter=10000),
          svm.SVC(kernel='rbf', gamma=0.7, C=C),
          svm.SVC(kernel='poly', degree=3, gamma='auto', C=C))
```

(continues on next page)

(continued from previous page)

```
models = (clf.fit(X, y) for clf in models)

# title for the plots
titles = ('SVC with linear kernel',
          'LinearSVC (linear kernel)',
          'SVC with RBF kernel',
          'SVC with polynomial (degree 3) kernel')

# Set-up 2x2 grid for plotting.
fig, sub = plt.subplots(2, 2)
plt.subplots_adjust(wspace=0.4, hspace=0.4)

X0, X1 = X[:, 0], X[:, 1]
xx, yy = make_meshgrid(X0, X1)

for clf, title, ax in zip(models, titles, sub.flatten()):
    plot_contours(ax, clf, xx, yy,
                  cmap=plt.cm.coolwarm, alpha=0.8)
    ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20, edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xlabel('Sepal length')
    ax.set_ylabel('Sepal width')
    ax.set_xticks(())
    ax.set_yticks(())
    ax.set_title(title)

plt.show()
```

Total running time of the script: (0 minutes 0.718 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.28.14 Scaling the regularization parameter for SVCs

The following example illustrates the effect of scaling the regularization parameter when using *Support Vector Machines* for *classification*. For SVC classification, we are interested in a risk minimization for the equation:

$$C \sum_{i=1,n} \mathcal{L}(f(x_i), y_i) + \Omega(w)$$

where

- C is used to set the amount of regularization
- $\mathcal{L}$ is a `loss` function of our samples and our model parameters.
- Ω is a `penalty` function of our model parameters

If we consider the loss function to be the individual error per sample, then the data-fit term, or the sum of the error for each sample, will increase as we add more samples. The penalization term, however, will not increase.

When using, for example, *cross validation*, to set the amount of regularization with `C`, there will be a different amount of samples between the main problem and the smaller problems within the folds of the cross validation.

Since our loss function is dependent on the amount of samples, the latter will influence the selected value of `C`. The question that arises is `How do we optimally adjust C to account for the different amount of training samples?`

The figures below are used to illustrate the effect of scaling our `C` to compensate for the change in the number of samples, in the case of using an `l1` penalty, as well as the `l2` penalty.

l1-penalty case

In the `l1` case, theory says that prediction consistency (i.e. that under given hypothesis, the estimator learned predicts as well as a model knowing the true distribution) is not possible because of the bias of the `l1`. It does say, however, that model consistency, in terms of finding the right set of non-zero parameters as well as their signs, can be achieved by scaling `C1`.

l2-penalty case

The theory says that in order to achieve prediction consistency, the penalty parameter should be kept constant as the number of samples grow.

Simulations

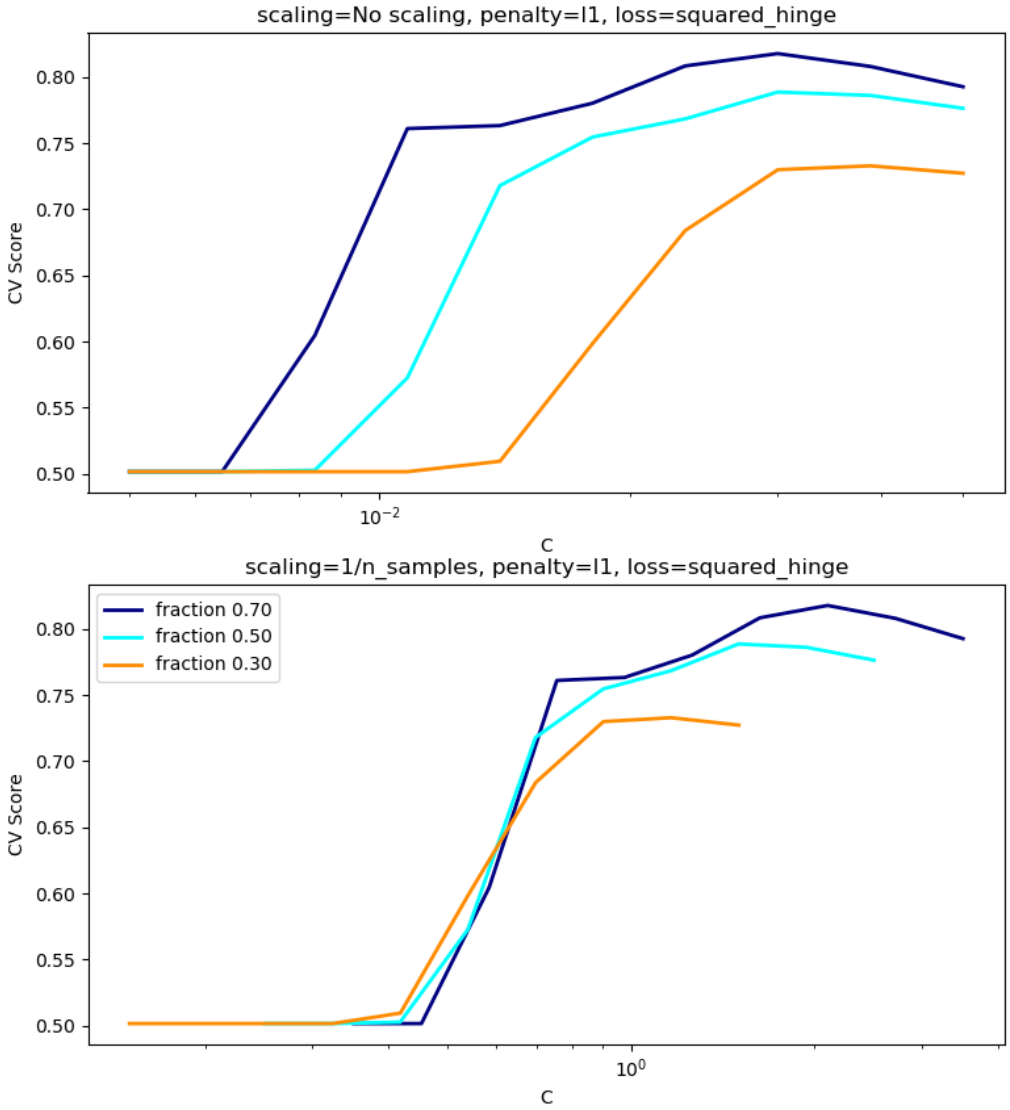
The two figures below plot the values of `C` on the `x-axis` and the corresponding cross-validation scores on the `y-axis`, for several different fractions of a generated data-set.

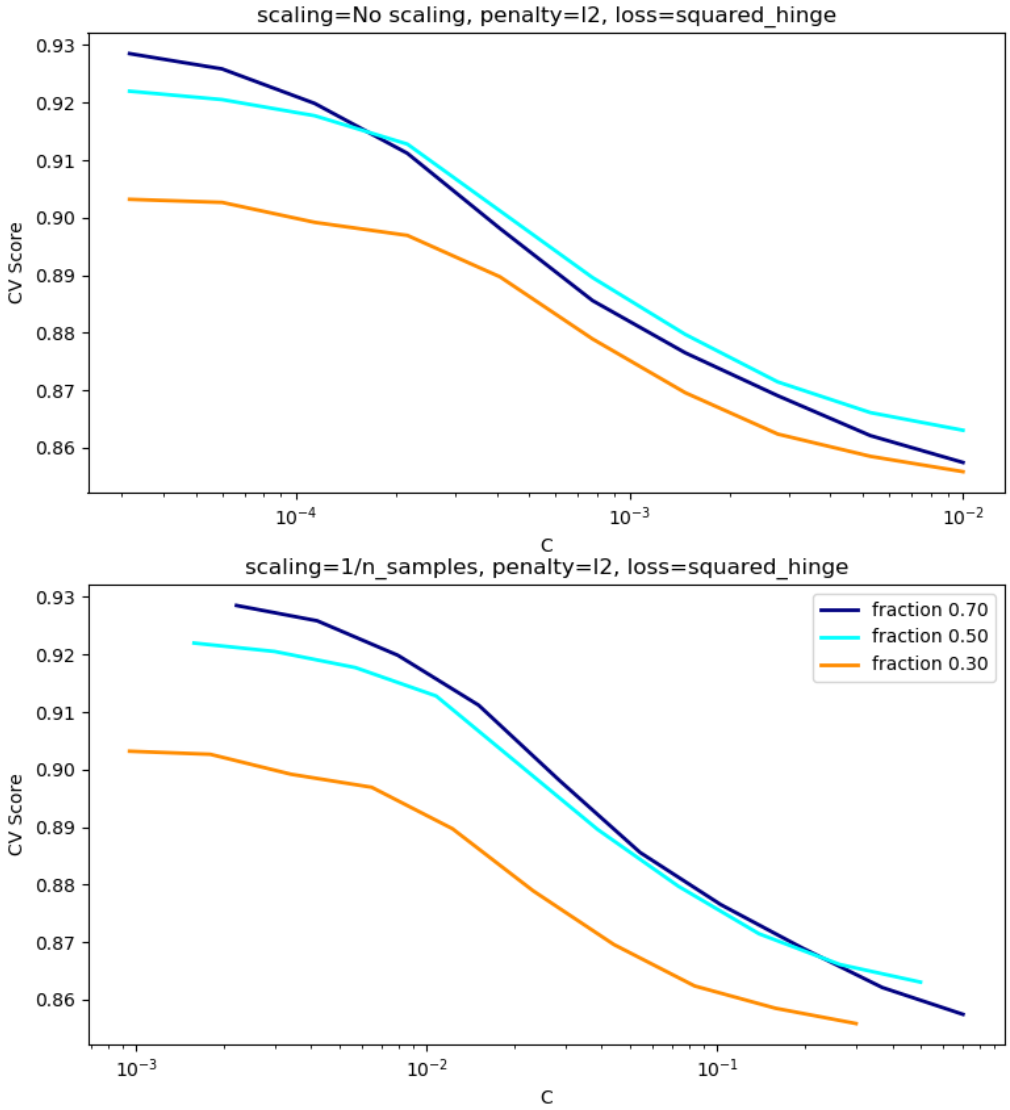
In the `l1` penalty case, the cross-validation-error correlates best with the test-error, when scaling our `C` with the number of samples, `n`, which can be seen in the first figure.

For the `l2` penalty case, the best result comes from the case where `C` is not scaled.

Note:

Two separate datasets are used for the two different plots. The reason behind this is the `l1` case works better on sparse data, while `l2` is better suited to the non-sparse case.


scaling=No scaling, penalty=l1, loss=squared_hinge
CV Score
0.80
0.75
0.70
0.65
0.60
0.55
0.50
10^{-2}
C
scaling=1/n_samples, penalty=l1, loss=squared_hinge
fraction 0.70
fraction 0.50
fraction 0.30
CV Score
0.80
0.75
0.70
0.65
0.60
0.55
0.50
10^{0}
C



-

```
print(__doc__)


# Author: Andreas Mueller <amueller@ais.uni-bonn.de>
#         Jaques Grobler <jaques.grobler@inria.fr>
# License: BSD 3 clause


import numpy as np
import matplotlib.pyplot as plt

from sklearn.svm import LinearSVC
```

(continues on next page)

(continued from previous page)

```
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import GridSearchCV
from sklearn.utils import check_random_state
from sklearn import datasets

rnd = check_random_state(1)

# set up dataset
n_samples = 100
n_features = 300

# l1 data (only 5 informative features)
X_1, y_1 = datasets.make_classification(n_samples=n_samples,
                                        n_features=n_features, n_informative=5,
                                        random_state=1)

# l2 data: non sparse, but less features
y_2 = np.sign(.5 - rnd.rand(n_samples))
X_2 = rnd.randn(n_samples, n_features // 5) + y_2[:, np.newaxis]
X_2 += 5 * rnd.randn(n_samples, n_features // 5)

clf_sets = [(LinearSVC(penalty='l1', loss='squared_hinge', dual=False,
                       tol=1e-3),
             np.logspace(-2.3, -1.3, 10), X_1, y_1),
            (LinearSVC(penalty='l2', loss='squared_hinge', dual=True),
             np.logspace(-4.5, -2, 10), X_2, y_2)]

colors = ['navy', 'cyan', 'darkorange']
lw = 2

for clf, cs, X, y in clf_sets:
    # set up the plot for each regressor
    fig, axes = plt.subplots(nrows=2, sharey=True, figsize=(9, 10))

    for k, train_size in enumerate(np.linspace(0.3, 0.7, 3)[::-1]):
        param_grid = dict(C=cs)
        # To get nice curve, we need a large number of iterations to
        # reduce the variance
        grid = GridSearchCV(clf, refit=False, param_grid=param_grid,
                            cv=ShuffleSplit(train_size=train_size,
                                            test_size=.3,
                                            n_splits=250, random_state=1))
        grid.fit(X, y)
        scores = grid.cv_results_['mean_test_score']

        scales = [(1, 'No scaling'),
                  ((n_samples * train_size), '1/n_samples'),
                  ]

        for ax, (scaler, name) in zip(axes, scales):
            ax.set_xlabel('C')
            ax.set_ylabel('CV Score')
            grid_cs = cs * float(scaler)  # scale the C's
            ax.semilogx(grid_cs, scores, label="fraction %.2f" %
                        train_size, color=colors[k], lw=lw)
            ax.set_title('scaling=%s, penalty=%s, loss=%s' %
                         (name, clf.penalty, clf.loss))
```

(continues on next page)

(continued from previous page)

```
    plt.legend(loc="best")
plt.show()
```

Total running time of the script: (0 minutes 16.905 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.28.15 RBF SVM parameters

This example illustrates the effect of the parameters `gamma` and `C` of the Radial Basis Function (RBF) kernel SVM.

Intuitively, the `gamma` parameter defines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'. The `gamma` parameters can be seen as the inverse of the radius of influence of samples selected by the model as support vectors.

The `C` parameter trades off correct classification of training examples against maximization of the decision function's margin. For larger values of `C`, a smaller margin will be accepted if the decision function is better at classifying all training points correctly. A lower `C` will encourage a larger margin, therefore a simpler decision function, at the cost of training accuracy. In other words''C'' behaves as a regularization parameter in the SVM.

The first plot is a visualization of the decision function for a variety of parameter values on a simplified classification problem involving only 2 input features and 2 possible target classes (binary classification). Note that this kind of plot is not possible to do for problems with more features or target classes.

The second plot is a heatmap of the classifier's cross-validation accuracy as a function of `C` and `gamma`. For this example we explore a relatively large grid for illustration purposes. In practice, a logarithmic grid from 10^{-3} to 10^{3} is usually sufficient. If the best parameters lie on the boundaries of the grid, it can be extended in that direction in a subsequent search.

Note that the heat map plot has a special colorbar with a midpoint value close to the score values of the best performing models so as to make it easy to tell them apart in the blink of an eye.

The behavior of the model is very sensitive to the `gamma` parameter. If `gamma` is too large, the radius of the area of influence of the support vectors only includes the support vector itself and no amount of regularization with `C` will be able to prevent overfitting.

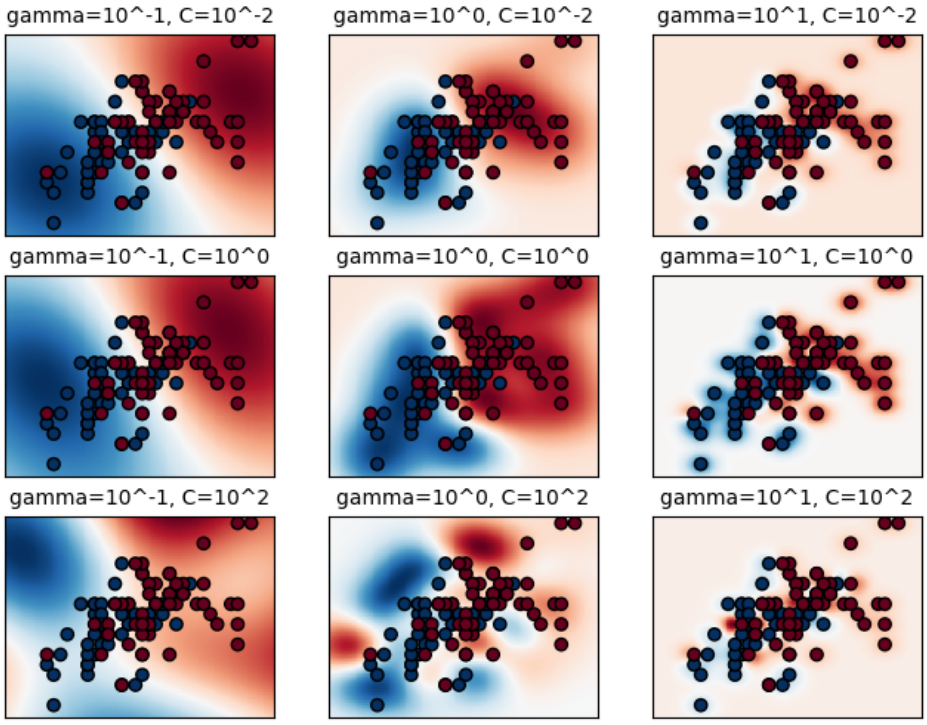
When `gamma` is very small, the model is too constrained and cannot capture the complexity or "shape" of the data. The region of influence of any selected support vector would include the whole training set. The resulting model will behave similarly to a linear model with a set of hyperplanes that separate the centers of high density of any pair of two classes.

For intermediate values, we can see on the second plot that good models can be found on a diagonal of `C` and `gamma`. Smooth models (lower `gamma` values) can be made more complex by increasing the importance of classifying each point correctly (larger `C` values) hence the diagonal of good performing models.

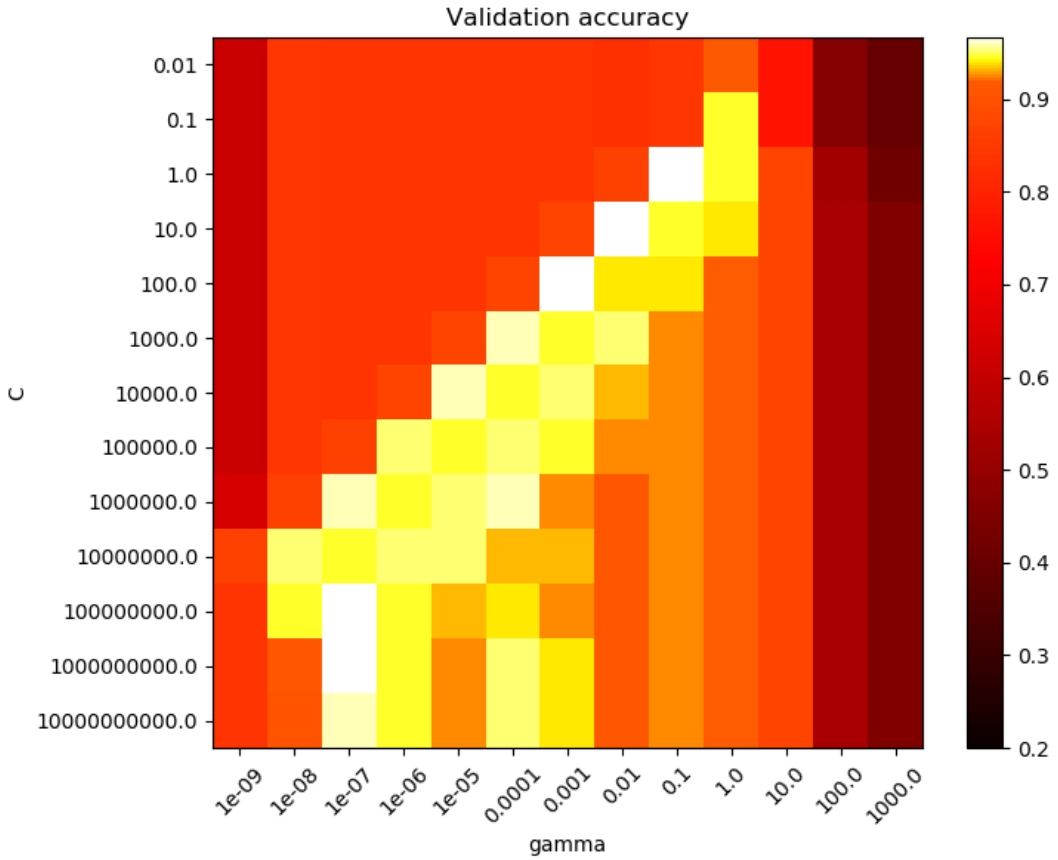
Finally one can also observe that for some intermediate values of `gamma` we get equally performing models when `C` becomes very large: it is not necessary to regularize by enforcing a larger margin. The radius of the RBF kernel alone acts as a good structural regularizer. In practice though it might still be interesting to simplify the decision function with a lower value of `C` so as to favor models that use less memory and that are faster to predict.

We should also note that small differences in scores results from the random splits of the cross-validation procedure. Those spurious variations can be smoothed out by increasing the number of CV iterations `n_splits` at the expense

of compute time. Increasing the value number of `C_range` and `gamma_range` steps will increase the resolution of the hyper-parameter heat map.



-



-

Out:

```
The best parameters are {'C': 1.0, 'gamma': 0.1} with a score of 0.97
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize

from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.model_selection import GridSearchCV


# Utility function to move the midpoint of a colormap to be around
# the values of interest.

class MidpointNormalize(Normalize):

```

(continues on next page)

(continued from previous page)

```
    def __init__(self, vmin=None, vmax=None, midpoint=None, clip=False):
        self.midpoint = midpoint
        Normalize.__init__(self, vmin, vmax, clip)

    def __call__(self, value, clip=None):
        x, y = [self.vmin, self.midpoint, self.vmax], [0, 0.5, 1]
        return np.ma.masked_array(np.interp(value, x, y))

# #############################################################################
# Load and prepare data set
#
# dataset for grid search

iris = load_iris()
X = iris.data
y = iris.target

# Dataset for decision function visualization: we only keep the first two
# features in X and sub-sample the dataset to keep only 2 classes and
# make it a binary classification problem.

X_2d = X[:, :2]
X_2d = X_2d[y > 0]
y_2d = y[y > 0]
y_2d -= 1

# It is usually a good idea to scale the data for SVM training.
# We are cheating a bit in this example in scaling all of the data,
# instead of fitting the transformation on the training set and
# just applying it on the test set.

scaler = StandardScaler()
X = scaler.fit_transform(X)
X_2d = scaler.fit_transform(X_2d)

# #############################################################################
# Train classifiers
#
# For an initial search, a logarithmic grid with basis
# 10 is often helpful. Using a basis of 2, a finer
# tuning can be achieved but at a much higher cost.

C_range = np.logspace(-2, 10, 13)
gamma_range = np.logspace(-9, 3, 13)
param_grid = dict(gamma=gamma_range, C=C_range)
cv = StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
grid = GridSearchCV(SVC(), param_grid=param_grid, cv=cv)
grid.fit(X, y)

print("The best parameters are %s with a score of %0.2f"
      % (grid.best_params_, grid.best_score_))

# Now we need to fit a classifier for all parameters in the 2d version
# (we use a smaller set of parameters here because it takes a while to train)

C_2d_range = [1e-2, 1, 1e2]
gamma_2d_range = [1e-1, 1, 1e1]
```

(continues on next page)

(continued from previous page)

```
classifiers = []
for C in C_2d_range:
    for gamma in gamma_2d_range:
        clf = SVC(C=C, gamma=gamma)
        clf.fit(X_2d, y_2d)
        classifiers.append((C, gamma, clf))

# #############################################################################
# Visualization
#
# draw visualization of parameter effects

plt.figure(figsize=(8, 6))
xx, yy = np.meshgrid(np.linspace(-3, 3, 200), np.linspace(-3, 3, 200))
for (k, (C, gamma, clf)) in enumerate(classifiers):
    # evaluate decision function in a grid
    Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # visualize decision function for these parameters
    plt.subplot(len(C_2d_range), len(gamma_2d_range), k + 1)
    plt.title("gamma=10^%d, C=10^%d" % (np.log10(gamma), np.log10(C)),
              size='medium')

    # visualize parameter's effect on decision function
    plt.pcolormesh(xx, yy, -Z, cmap=plt.cm.RdBu)
    plt.scatter(X_2d[:, 0], X_2d[:, 1], c=y_2d, cmap=plt.cm.RdBu_r,
                edgecolors='k')
    plt.xticks(())
    plt.yticks(())
    plt.axis('tight')

scores = grid.cv_results_['mean_test_score'].reshape(len(C_range),
                                                     len(gamma_range))

# Draw heatmap of the validation accuracy as a function of gamma and C
#
# The score are encoded as colors with the hot colormap which varies from dark
# red to bright yellow. As the most interesting scores are all located in the
# 0.92 to 0.97 range we use a custom normalizer to set the mid-point to 0.92 so
# as to make it easier to visualize the small variations of score values in the
# interesting range while not brutally collapsing all the low score values to
# the same color.

plt.figure(figsize=(8, 6))
plt.subplots_adjust(left=.2, right=0.95, bottom=0.15, top=0.95)
plt.imshow(scores, interpolation='nearest', cmap=plt.cm.hot,
           norm=MidpointNormalize(vmin=0.2, midpoint=0.92))
plt.xlabel('gamma')
plt.ylabel('C')
plt.colorbar()
plt.xticks(np.arange(len(gamma_range)), gamma_range, rotation=45)
plt.yticks(np.arange(len(C_range)), C_range)
plt.title('Validation accuracy')
plt.show()
```

Total running time of the script: (0 minutes 4.277 seconds)

Estimated memory usage: 8 MB

6.29 Tutorial exercises

Exercises for the tutorials

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.29.1 Digits Classification Exercise

A tutorial exercise regarding the use of classification techniques on the Digits dataset.

This exercise is used in the *Classification* part of the *Supervised learning: predicting an output variable from high-dimensional observations* section of the *A tutorial on statistical-learning for scientific data processing*.

Out:

```
KNN score: 0.961111
LogisticRegression score: 0.933333
```

```
print(__doc__)

from sklearn import datasets, neighbors, linear_model

X_digits, y_digits = datasets.load_digits(return_X_y=True)
X_digits = X_digits / X_digits.max()

n_samples = len(X_digits)

X_train = X_digits[:int(.9 * n_samples)]
y_train = y_digits[:int(.9 * n_samples)]
X_test = X_digits[int(.9 * n_samples):]
y_test = y_digits[int(.9 * n_samples):]

knn = neighbors.KNeighborsClassifier()
logistic = linear_model.LogisticRegression(max_iter=1000)

print('KNN score: %f' % knn.fit(X_train, y_train).score(X_test, y_test))
print('LogisticRegression score: %f'
      % logistic.fit(X_train, y_train).score(X_test, y_test))
```

Total running time of the script: (0 minutes 0.607 seconds)

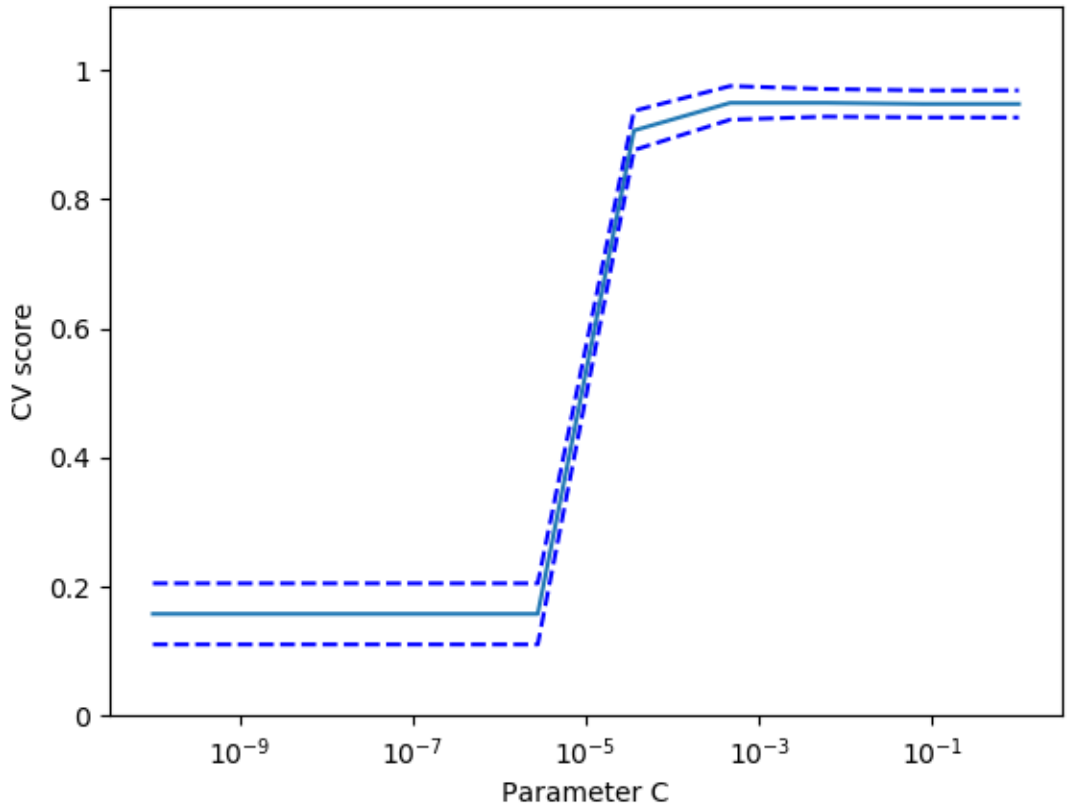
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.29.2 Cross-validation on Digits Dataset Exercise

A tutorial exercise using Cross-validation with an SVM on the Digits dataset.

This exercise is used in the *Cross-validation generators* part of the *Model selection: choosing estimators and their parameters* section of the *A tutorial on statistical-learning for scientific data processing*.




```
print(__doc__)


import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn import datasets, svm

X, y = datasets.load_digits(return_X_y=True)

svc = svm.SVC(kernel='linear')
C_s = np.logspace(-10, 0, 10)

scores = list()
scores_std = list()
for C in C_s:
    svc.C = C
    this_scores = cross_val_score(svc, X, y, n_jobs=1)
    scores.append(np.mean(this_scores))
    scores_std.append(np.std(this_scores))
```

(continues on next page)

(continued from previous page)

```
# Do the plotting
import matplotlib.pyplot as plt
plt.figure()
plt.semilogx(C_s, scores)
plt.semilogx(C_s, np.array(scores) + np.array(scores_std), 'b--')
plt.semilogx(C_s, np.array(scores) - np.array(scores_std), 'b--')
locs, labels = plt.yticks()
plt.yticks(locs, list(map(lambda x: "%g" % x, locs)))
plt.ylabel('CV score')
plt.xlabel('Parameter C')
plt.ylim(0, 1.1)
plt.show()
```

Total running time of the script: (0 minutes 8.988 seconds)

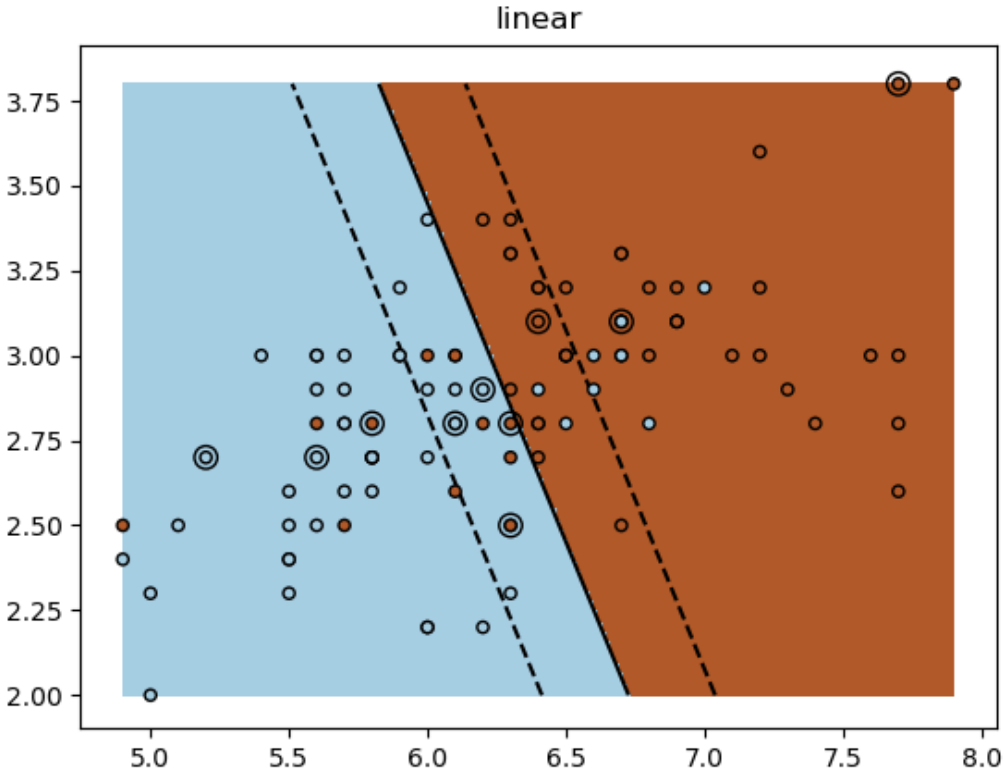
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

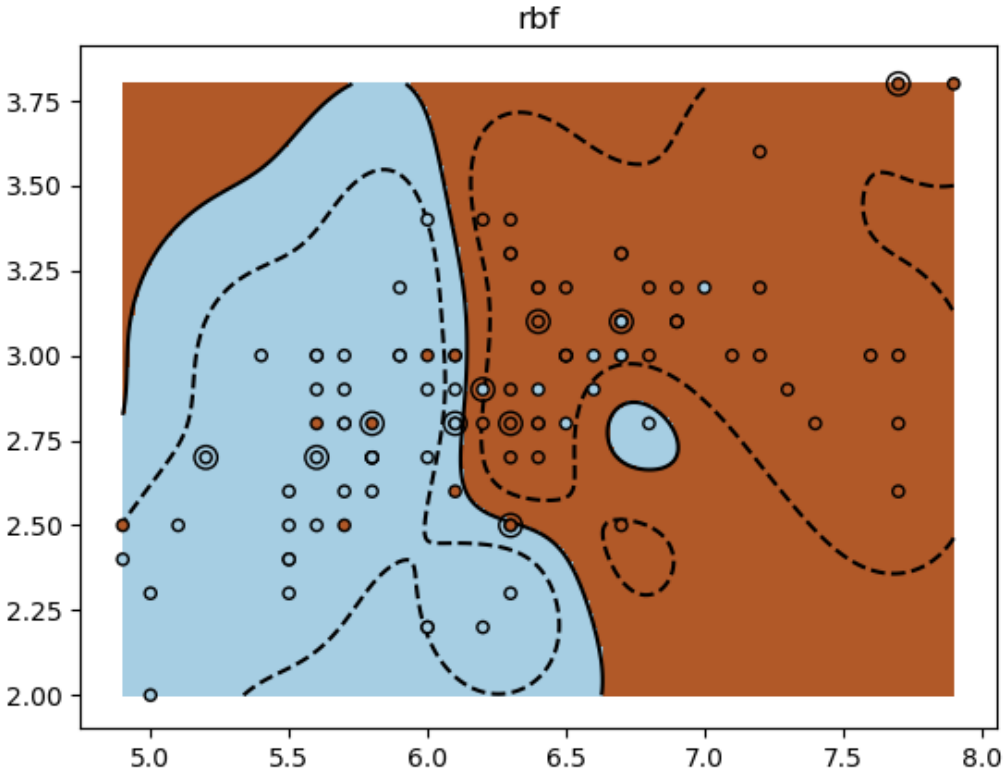
6.29.3 SVM Exercise

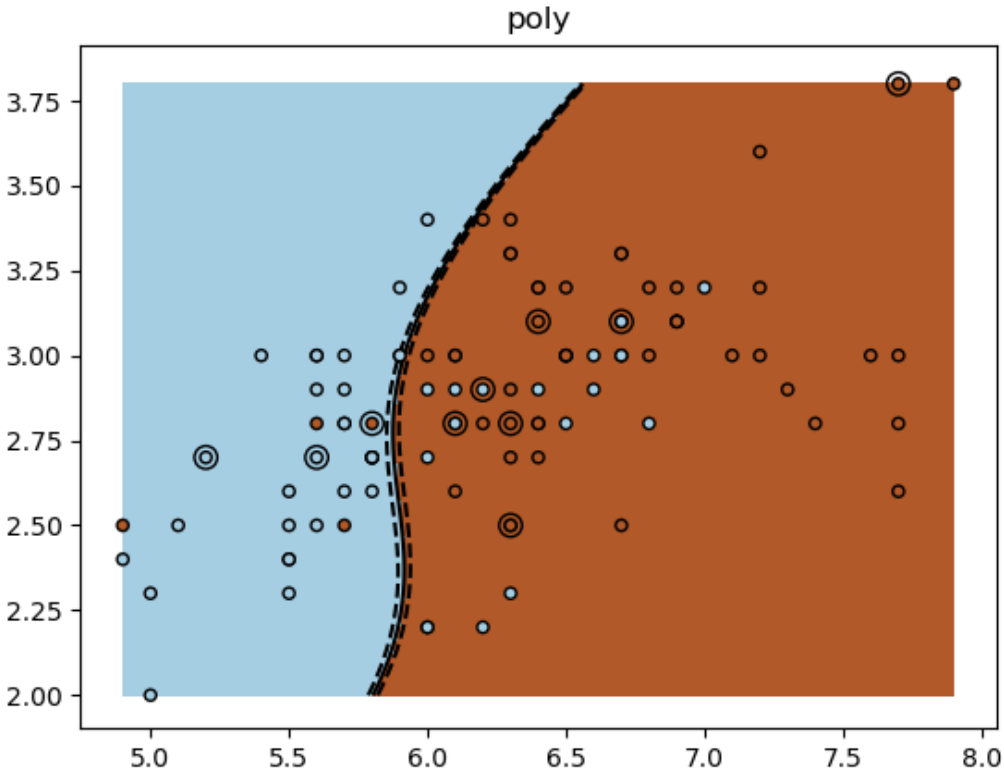
A tutorial exercise for using different SVM kernels.

This exercise is used in the *Using kernels* part of the *Supervised learning: predicting an output variable from high-dimensional observations* section of the *A tutorial on statistical-learning for scientific data processing*.


linear
3.75
3.50
3.25
3.00
2.75
2.50
2.25
2.00
5.0
5.5
6.0
6.5
7.0
7.5
8.0

-


rbf
3.75
3.50
3.25
3.00
2.75
2.50
2.25
2.00
5.0
5.5
6.0
6.5
7.0
7.5
8.0



-

```
print(__doc__)


import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets, svm

iris = datasets.load_iris()
X = iris.data
y = iris.target

X = X[y != 0, :2]
y = y[y != 0]

n_sample = len(X)

np.random.seed(0)
order = np.random.permutation(n_sample)
X = X[order]
y = y[order].astype(np.float)

X_train = X[:int(.9 * n_sample)]
y_train = y[:int(.9 * n_sample)]
X_test = X[int(.9 * n_sample):]
y_test = y[int(.9 * n_sample):]

# fit the model
```

(continues on next page)

(continued from previous page)

```
for kernel in ('linear', 'rbf', 'poly'):
    clf = svm.SVC(kernel=kernel, gamma=10)
    clf.fit(X_train, y_train)

    plt.figure()
    plt.clf()
    plt.scatter(X[:, 0], X[:, 1], c=y, zorder=10, cmap=plt.cm.Paired,
                edgecolor='k', s=20)

    # Circle out the test data
    plt.scatter(X_test[:, 0], X_test[:, 1], s=80, facecolors='none',
                zorder=10, edgecolor='k')

    plt.axis('tight')
    x_min = X[:, 0].min()
    x_max = X[:, 0].max()
    y_min = X[:, 1].min()
    y_max = X[:, 1].max()

    XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
    Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(XX.shape)
    plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)
    plt.contour(XX, YY, Z, colors=['k', 'k', 'k'],
                linestyles=['--', '-', '--'], levels=[-.5, 0, .5])

    plt.title(kernel)
plt.show()
```

Total running time of the script: (0 minutes 5.806 seconds)

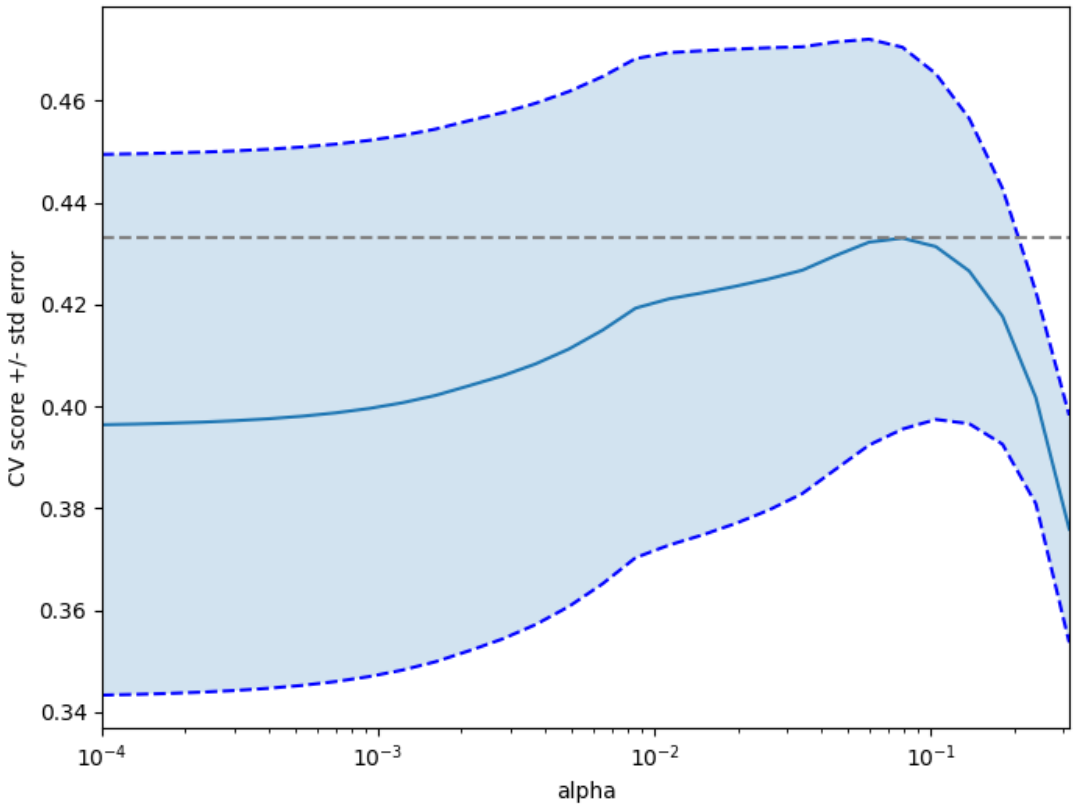
Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.29.4 Cross-validation on diabetes Dataset Exercise

A tutorial exercise which uses cross-validation with linear models.

This exercise is used in the *Cross-validated estimators* part of the *Model selection: choosing estimators and their parameters* section of the *A tutorial on statistical-learning for scientific data processing*.



Out:

```
Answer to the bonus question: how much can you trust the selection of alpha?

Alpha parameters maximising the generalization score on different
subsets of the data:
[fold 0] alpha: 0.05968, score: 0.54209
[fold 1] alpha: 0.04520, score: 0.15523
[fold 2] alpha: 0.07880, score: 0.45193

Answer: Not very much since we obtained different alphas for different
subsets of the data and moreover, the scores for these alphas differ
quite substantially.
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn import datasets
```

(continues on next page)

(continued from previous page)

```
from sklearn.linear_model import LassoCV
from sklearn.linear_model import Lasso
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV

X, y = datasets.load_diabetes(return_X_y=True)
X = X[:150]
y = y[:150]

lasso = Lasso(random_state=0, max_iter=10000)
alphas = np.logspace(-4, -0.5, 30)

tuned_parameters = [{'alpha': alphas}]
n_folds = 5

clf = GridSearchCV(lasso, tuned_parameters, cv=n_folds, refit=False)
clf.fit(X, y)
scores = clf.cv_results_['mean_test_score']
scores_std = clf.cv_results_['std_test_score']
plt.figure().set_size_inches(8, 6)
plt.semilogx(alphas, scores)

# plot error lines showing +/- std. errors of the scores
std_error = scores_std / np.sqrt(n_folds)

plt.semilogx(alphas, scores + std_error, 'b--')
plt.semilogx(alphas, scores - std_error, 'b--')

# alpha=0.2 controls the translucency of the fill color
plt.fill_between(alphas, scores + std_error, scores - std_error, alpha=0.2)

plt.ylabel('CV score +/- std error')
plt.xlabel('alpha')
plt.axhline(np.max(scores), linestyle='--', color='.5')
plt.xlim([alphas[0], alphas[-1]])

# #############################################################################
# Bonus: how much can you trust the selection of alpha?

# To answer this question we use the LassoCV object that sets its alpha
# parameter automatically from the data by internal cross-validation (i.e. it
# performs cross-validation on the training data it receives).
# We use external cross-validation to see how much the automatically obtained
# alphas differ across different cross-validation folds.
lasso_cv = LassoCV(alphas=alphas, random_state=0, max_iter=10000)
k_fold = KFold(3)

print("Answer to the bonus question:",
      "how much can you trust the selection of alpha?")
print()
print("Alpha parameters maximising the generalization score on different")
print("subsets of the data:")
for k, (train, test) in enumerate(k_fold.split(X, y)):
    lasso_cv.fit(X[train], y[train])
    print("[fold {0}] alpha: {1:.5f}, score: {2:.5f}".
          format(k, lasso_cv.alpha_, lasso_cv.score(X[test], y[test])))
print()
```

(continues on next page)

(continued from previous page)

```
print("Answer: Not very much since we obtained different alphas for different")
print("subsets of the data and moreover, the scores for these alphas differ")
print("quite substantially.")

plt.show()
```

Total running time of the script: (0 minutes 0.771 seconds)

Estimated memory usage: 8 MB

6.30 Working with text documents

Examples concerning the *sklearn.feature_extraction.text* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.30.1 FeatureHasher and DictVectorizer Comparison

Compares FeatureHasher and DictVectorizer by using both to vectorize text documents.

The example demonstrates syntax and speed only; it doesn't actually do anything useful with the extracted vectors. See the example scripts {document_classification_20newsgroups,clustering}.py for actual learning on text documents.

A discrepancy between the number of terms reported for DictVectorizer and for FeatureHasher is to be expected due to hash collisions.

Out:

```
Usage: /home/circleci/project/examples/text/plot_hashing_vs_dict_vectorizer.py [n_
↪features_for_hashing]
    The default number of features is 2**18.

Loading 20 newsgroups training data
3803 documents - 6.245MB

DictVectorizer
done in 1.105289s at 5.650MB/s
Found 47928 unique terms

FeatureHasher on frequency dicts
done in 0.777673s at 8.030MB/s
Found 43873 unique terms

FeatureHasher on raw tokens
done in 0.651078s at 9.591MB/s
Found 43873 unique terms
```

```
# Author: Lars Buitinck
# License: BSD 3 clause
from collections import defaultdict
import re
import sys
from time import time

import numpy as np

from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction import DictVectorizer, FeatureHasher


def n_nonzero_columns(X):
    """Returns the number of non-zero columns in a CSR matrix X."""
    return len(np.unique(X.nonzero()[1]))


def tokens(doc):
    """Extract tokens from doc.

    This uses a simple regex to break strings into tokens. For a more
    principled approach, see CountVectorizer or TfidfVectorizer.
    """
    return (tok.lower() for tok in re.findall(r"\w+", doc))


def token_freqs(doc):
    """Extract a dict mapping tokens from doc to their frequencies."""
    freq = defaultdict(int)
    for tok in tokens(doc):
        freq[tok] += 1
    return freq


categories = [
    'alt.atheism',
    'comp.graphics',
    'comp.sys.ibm.pc.hardware',
    'misc.forsale',
    'rec.autos',
    'sci.space',
    'talk.religion.misc',
]
# Uncomment the following line to use a larger set (11k+ documents)
# categories = None

print(__doc__)
print("Usage: %s [n_features_for_hashing]" % sys.argv[0])
print("    The default number of features is 2**18.")
print()

try:
    n_features = int(sys.argv[1])
except IndexError:
    n_features = 2 ** 18
except ValueError:
```

(continues on next page)

(continued from previous page)

```
    print("not a valid number of features: %r" % sys.argv[1])
    sys.exit(1)


print("Loading 20 newsgroups training data")
raw_data, _ = fetch_20newsgroups(subset='train', categories=categories,
                                 return_X_y=True)
data_size_mb = sum(len(s.encode('utf-8')) for s in raw_data) / 1e6
print("%d documents - %0.3fMB" % (len(raw_data), data_size_mb))
print()

print("DictVectorizer")
t0 = time()
vectorizer = DictVectorizer()
vectorizer.fit_transform(token_freqs(d) for d in raw_data)
duration = time() - t0
print("done in %fs at %0.3fMB/s" % (duration, data_size_mb / duration))
print("Found %d unique terms" % len(vectorizer.get_feature_names()))
print()

print("FeatureHasher on frequency dicts")
t0 = time()
hasher = FeatureHasher(n_features=n_features)
X = hasher.transform(token_freqs(d) for d in raw_data)
duration = time() - t0
print("done in %fs at %0.3fMB/s" % (duration, data_size_mb / duration))
print("Found %d unique terms" % n_nonzero_columns(X))
print()

print("FeatureHasher on raw tokens")
t0 = time()
hasher = FeatureHasher(n_features=n_features, input_type="string")
X = hasher.transform(tokens(d) for d in raw_data)
duration = time() - t0
print("done in %fs at %0.3fMB/s" % (duration, data_size_mb / duration))
print("Found %d unique terms" % n_nonzero_columns(X))
```

Total running time of the script: (0 minutes 3.154 seconds)

Estimated memory usage: 42 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.30.2 Clustering text documents using k-means

This is an example showing how the scikit-learn can be used to cluster documents by topics using a bag-of-words approach. This example uses a scipy.sparse matrix to store the features instead of standard numpy arrays.

Two feature extraction methods can be used in this example:

- TfidfVectorizer uses a in-memory vocabulary (a python dict) to map the most frequent words to features indices and hence compute a word occurrence frequency (sparse) matrix. The word frequencies are then reweighted using the Inverse Document Frequency (IDF) vector collected feature-wise over the corpus.
- HashingVectorizer hashes word occurrences to a fixed dimensional space, possibly with collisions. The word

count vectors are then normalized to each have l2-norm equal to one (projected to the euclidean unit-ball) which seems to be important for k-means to work in high dimensional space.

HashingVectorizer does not provide IDF weighting as this is a stateless model (the fit method does nothing). When IDF weighting is needed it can be added by pipelining its output to a TfidfTransformer instance.

Two algorithms are demoed: ordinary k-means and its more scalable cousin minibatch k-means.

Additionally, latent semantic analysis can also be used to reduce dimensionality and discover latent patterns in the data.

It can be noted that k-means (and minibatch k-means) are very sensitive to feature scaling and that in this case the IDF weighting helps improve the quality of the clustering by quite a lot as measured against the "ground truth" provided by the class label assignments of the 20 newsgroups dataset.

This improvement is not visible in the Silhouette Coefficient which is small for both as this measure seem to suffer from the phenomenon called "Concentration of Measure" or "Curse of Dimensionality" for high dimensional datasets such as text data. Other measures such as V-measure and Adjusted Rand Index are information theoretic based evaluation scores: as they are only based on cluster assignments rather than distances, hence not affected by the curse of dimensionality.

Note: as k-means is optimizing a non-convex objective function, it will likely end up in a local optimum. Several runs with independent random init might be necessary to get a good convergence.

Out:

```
Usage: plot_document_clustering.py [options]

Options:
  -h, --help            show this help message and exit
  --lsa=N_COMPONENTS    Preprocess documents with latent semantic analysis.
  --no-minibatch        Use ordinary k-means algorithm (in batch mode).
  --no-idf              Disable Inverse Document Frequency feature weighting.
  --use-hashing         Use a hashing feature vectorizer
  --n-features=N_FEATURES
                        Maximum number of features (dimensions) to extract
                        from text.
  --verbose             Print progress reports inside k-means algorithm.
Loading 20 newsgroups dataset for categories:
['alt.atheism', 'talk.religion.misc', 'comp.graphics', 'sci.space']
3387 documents
4 categories

Extracting features from the training dataset using a sparse vectorizer
done in 0.869272s
n_samples: 3387, n_features: 10000

Clustering sparse data with MiniBatchKMeans(batch_size=1000, init_size=1000, n_
↪clusters=4, n_init=1,
                verbose=False)
done in 0.071s

Homogeneity: 0.219
Completeness: 0.338
V-measure: 0.266
Adjusted Rand-Index: 0.113
Silhouette Coefficient: 0.005

Top terms per cluster:
Cluster 0: cc ibm au buffalo monash com vnet software nicho university
```

(continues on next page)

(continued from previous page)

```
Cluster 1: space nasa henry access digex toronto gov pat alaska shuttle
Cluster 2: com god university article don know graphics people posting like
Cluster 3: sgi keith livesey morality jon solntze wpd caltech objective moral
```

```
# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#         Lars Buitinck
# License: BSD 3 clause
from sklearn.datasets import fetch_20newsgroups
from sklearn.decomposition import TruncatedSVD
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import Normalizer
from sklearn import metrics

from sklearn.cluster import KMeans, MiniBatchKMeans

import logging
from optparse import OptionParser
import sys
from time import time

import numpy as np


# Display progress logs on stdout
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s %(message)s')

# parse commandline arguments
op = OptionParser()
op.add_option("--lsa",
              dest="n_components", type="int",
              help="Preprocess documents with latent semantic analysis.")
op.add_option("--no-minibatch",
              action="store_false", dest="minibatch", default=True,
              help="Use ordinary k-means algorithm (in batch mode).")
op.add_option("--no-idf",
              action="store_false", dest="use_idf", default=True,
              help="Disable Inverse Document Frequency feature weighting.")
op.add_option("--use-hashing",
              action="store_true", default=False,
              help="Use a hashing feature vectorizer")
op.add_option("--n-features", type=int, default=10000,
              help="Maximum number of features (dimensions)"
                   " to extract from text.")
op.add_option("--verbose",
              action="store_true", dest="verbose", default=False,
              help="Print progress reports inside k-means algorithm.")
```

(continues on next page)

(continued from previous page)

```
print(__doc__)
op.print_help()


def is_interactive():
    return not hasattr(sys.modules['__main__'], '__file__')


# work-around for Jupyter notebook and IPython console
argv = [] if is_interactive() else sys.argv[1:]
(opts, args) = op.parse_args(argv)
if len(args) > 0:
    op.error("this script takes no arguments.")
    sys.exit(1)


# #############################################################################
# Load some categories from the training set
categories = [
    'alt.atheism',
    'talk.religion.misc',
    'comp.graphics',
    'sci.space',
]
# Uncomment the following to do the analysis on all the categories
# categories = None

print("Loading 20 newsgroups dataset for categories:")
print(categories)

dataset = fetch_20newsgroups(subset='all', categories=categories,
                             shuffle=True, random_state=42)

print("%d documents" % len(dataset.data))
print("%d categories" % len(dataset.target_names))
print()

labels = dataset.target
true_k = np.unique(labels).shape[0]

print("Extracting features from the training dataset "
      "using a sparse vectorizer")
t0 = time()
if opts.use_hashing:
    if opts.use_idf:
        # Perform an IDF normalization on the output of HashingVectorizer
        hasher = HashingVectorizer(n_features=opts.n_features,
                                   stop_words='english', alternate_sign=False,
                                   norm=None)
        vectorizer = make_pipeline(hasher, TfidfTransformer())
    else:
        vectorizer = HashingVectorizer(n_features=opts.n_features,
                                       stop_words='english',
                                       alternate_sign=False, norm='l2')
else:
    vectorizer = TfidfVectorizer(max_df=0.5, max_features=opts.n_features,
                                 min_df=2, stop_words='english',
```

(continues on next page)

(continued from previous page)

```
                                    use_idf=opts.use_idf)
X = vectorizer.fit_transform(dataset.data)

print("done in %fs" % (time() - t0))
print("n_samples: %d, n_features: %d" % X.shape)
print()

if opts.n_components:
    print("Performing dimensionality reduction using LSA")
    t0 = time()
    # Vectorizer results are normalized, which makes KMeans behave as
    # spherical k-means for better results. Since LSA/SVD results are
    # not normalized, we have to redo the normalization.
    svd = TruncatedSVD(opts.n_components)
    normalizer = Normalizer(copy=False)
    lsa = make_pipeline(svd, normalizer)

    X = lsa.fit_transform(X)

    print("done in %fs" % (time() - t0))

    explained_variance = svd.explained_variance_ratio_.sum()
    print("Explained variance of the SVD step: {}%".format(
        int(explained_variance * 100)))

    print()


# ##############################################################################
# Do the actual clustering

if opts.minibatch:
    km = MiniBatchKMeans(n_clusters=true_k, init='k-means++', n_init=1,
                         init_size=1000, batch_size=1000, verbose=opts.verbose)
else:
    km = KMeans(n_clusters=true_k, init='k-means++', max_iter=100, n_init=1,
                verbose=opts.verbose)

print("Clustering sparse data with %s" % km)
t0 = time()
km.fit(X)
print("done in %0.3fs" % (time() - t0))
print()

print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels, km.labels_))
print("Completeness: %0.3f" % metrics.completeness_score(labels, km.labels_))
print("V-measure: %0.3f" % metrics.v_measure_score(labels, km.labels_))
print("Adjusted Rand-Index: %.3f"
      % metrics.adjusted_rand_score(labels, km.labels_))
print("Silhouette Coefficient: %0.3f"
      % metrics.silhouette_score(X, km.labels_, sample_size=1000))

print()


if not opts.use_hashing:
    print("Top terms per cluster:")
```

(continues on next page)

(continued from previous page)

```
    if opts.n_components:
        original_space_centroids = svd.inverse_transform(km.cluster_centers_)
        order_centroids = original_space_centroids.argsort()[:, ::-1]
    else:
        order_centroids = km.cluster_centers_.argsort()[:, ::-1]

    terms = vectorizer.get_feature_names()
    for i in range(true_k):
        print("Cluster %d:" % i, end='')
        for ind in order_centroids[i, :10]:
            print(' %s' % terms[ind], end='')
        print()
```

Total running time of the script: (0 minutes 1.400 seconds)

Estimated memory usage: 50 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

6.30.3 Classification of text documents using sparse features

This is an example showing how scikit-learn can be used to classify documents by topics using a bag-of-words approach. This example uses a scipy.sparse matrix to store the features and demonstrates various classifiers that can efficiently handle sparse matrices.

The dataset used in this example is the 20 newsgroups dataset. It will be automatically downloaded, then cached.

```
# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#         Olivier Grisel <olivier.grisel@ensta.org>
#         Mathieu Blondel <mathieu@mblondel.org>
#         Lars Buitinck
# License: BSD 3 clause
import logging
import numpy as np
from optparse import OptionParser
import sys
from time import time
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.feature_selection import SelectFromModel
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.linear_model import RidgeClassifier
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC
from sklearn.linear_model import SGDClassifier
from sklearn.linear_model import Perceptron
from sklearn.linear_model import PassiveAggressiveClassifier
from sklearn.naive_bayes import BernoulliNB, ComplementNB, MultinomialNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neighbors import NearestCentroid
```

(continues on next page)

(continued from previous page)

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.utils.extmath import density
from sklearn import metrics


# Display progress logs on stdout
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s %(message)s')

op = OptionParser()
op.add_option("--report",
              action="store_true", dest="print_report",
              help="Print a detailed classification report.")
op.add_option("--chi2_select",
              action="store", type="int", dest="select_chi2",
              help="Select some number of features using a chi-squared test")
op.add_option("--confusion_matrix",
              action="store_true", dest="print_cm",
              help="Print the confusion matrix.")
op.add_option("--top10",
              action="store_true", dest="print_top10",
              help="Print ten most discriminative terms per class"
                   " for every classifier.")
op.add_option("--all_categories",
              action="store_true", dest="all_categories",
              help="Whether to use all categories or not.")
op.add_option("--use_hashing",
              action="store_true",
              help="Use a hashing vectorizer.")
op.add_option("--n_features",
              action="store", type=int, default=2 ** 16,
              help="n_features when using the hashing vectorizer.")
op.add_option("--filtered",
              action="store_true",
              help="Remove newsgroup information that is easily overfit: "
                   "headers, signatures, and quoting.")


def is_interactive():
    return not hasattr(sys.modules['__main__'], '__file__')


# work-around for Jupyter notebook and IPython console
argv = [] if is_interactive() else sys.argv[1:]
(opts, args) = op.parse_args(argv)
if len(args) > 0:
    op.error("this script takes no arguments.")
    sys.exit(1)

print(__doc__)
op.print_help()
print()
```

Out:

```
Usage: plot_document_classification_20newsgroups.py [options]
```

(continues on next page)

(continued from previous page)

```
Options:
  -h, --help            show this help message and exit
  --report              Print a detailed classification report.
  --chi2_select=SELECT_CHI2
                        Select some number of features using a chi-squared
                        test
  --confusion_matrix    Print the confusion matrix.
  --top10               Print ten most discriminative terms per class for
                        every classifier.
  --all_categories      Whether to use all categories or not.
  --use_hashing         Use a hashing vectorizer.
  --n_features=N_FEATURES
                        n_features when using the hashing vectorizer.
  --filtered            Remove newsgroup information that is easily overfit:
                        headers, signatures, and quoting.
```

Load data from the training set

Let's load data from the newsgroups dataset which comprises around 18000 newsgroups posts on 20 topics split in two subsets: one for training (or development) and the other one for testing (or for performance evaluation).

```
if opts.all_categories:
    categories = None
else:
    categories = [
        'alt.atheism',
        'talk.religion.misc',
        'comp.graphics',
        'sci.space',
    ]

if opts.filtered:
    remove = ('headers', 'footers', 'quotes')
else:
    remove = ()

print("Loading 20 newsgroups dataset for categories:")
print(categories if categories else "all")

data_train = fetch_20newsgroups(subset='train', categories=categories,
                                shuffle=True, random_state=42,
                                remove=remove)

data_test = fetch_20newsgroups(subset='test', categories=categories,
                               shuffle=True, random_state=42,
                               remove=remove)
print('data loaded')

# order of labels in `target_names` can be different from `categories`
target_names = data_train.target_names


def size_mb(docs):
    return sum(len(s.encode('utf-8')) for s in docs) / 1e6
```

(continues on next page)

(continued from previous page)

```
data_train_size_mb = size_mb(data_train.data)
data_test_size_mb = size_mb(data_test.data)

print("%d documents - %0.3fMB (training set)" % (
    len(data_train.data), data_train_size_mb))
print("%d documents - %0.3fMB (test set)" % (
    len(data_test.data), data_test_size_mb))
print("%d categories" % len(target_names))
print()

# split a training set and a test set
y_train, y_test = data_train.target, data_test.target

print("Extracting features from the training data using a sparse vectorizer")
t0 = time()
if opts.use_hashing:
    vectorizer = HashingVectorizer(stop_words='english', alternate_sign=False,
                                   n_features=opts.n_features)
    X_train = vectorizer.transform(data_train.data)
else:
    vectorizer = TfidfVectorizer(sublinear_tf=True, max_df=0.5,
                                 stop_words='english')
    X_train = vectorizer.fit_transform(data_train.data)
duration = time() - t0
print("done in %fs at %0.3fMB/s" % (duration, data_train_size_mb / duration))
print("n_samples: %d, n_features: %d" % X_train.shape)
print()

print("Extracting features from the test data using the same vectorizer")
t0 = time()
X_test = vectorizer.transform(data_test.data)
duration = time() - t0
print("done in %fs at %0.3fMB/s" % (duration, data_test_size_mb / duration))
print("n_samples: %d, n_features: %d" % X_test.shape)
print()

# mapping from integer feature name to original token string
if opts.use_hashing:
    feature_names = None
else:
    feature_names = vectorizer.get_feature_names()

if opts.select_chi2:
    print("Extracting %d best features by a chi-squared test" %
          opts.select_chi2)
    t0 = time()
    ch2 = SelectKBest(chi2, k=opts.select_chi2)
    X_train = ch2.fit_transform(X_train, y_train)
    X_test = ch2.transform(X_test)
    if feature_names:
        # keep selected feature names
        feature_names = [feature_names[i] for i
                         in ch2.get_support(indices=True)]
    print("done in %fs" % (time() - t0))
    print()
```

(continues on next page)

(continued from previous page)

```
if feature_names:
    feature_names = np.asarray(feature_names)


def trim(s):
    """Trim string to fit on terminal (assuming 80-column display)"""
    return s if len(s) <= 80 else s[:77] + "..."
```

Out:

```
Loading 20 newsgroups dataset for categories:
['alt.atheism', 'talk.religion.misc', 'comp.graphics', 'sci.space']
data loaded
2034 documents - 3.980MB (training set)
1353 documents - 2.867MB (test set)
4 categories

Extracting features from the training data using a sparse vectorizer
done in 0.494025s at 8.055MB/s
n_samples: 2034, n_features: 33809

Extracting features from the test data using the same vectorizer
done in 0.375047s at 7.646MB/s
n_samples: 1353, n_features: 33809
```

Benchmark classifiers

We train and test the datasets with 15 different classification models and get performance results for each model.

```
def benchmark(clf):
    print('_' * 80)
    print("Training: ")
    print(clf)
    t0 = time()
    clf.fit(X_train, y_train)
    train_time = time() - t0
    print("train time: %0.3fs" % train_time)

    t0 = time()
    pred = clf.predict(X_test)
    test_time = time() - t0
    print("test time:  %0.3fs" % test_time)

    score = metrics.accuracy_score(y_test, pred)
    print("accuracy:   %0.3f" % score)

    if hasattr(clf, 'coef_'):
        print("dimensionality: %d" % clf.coef_.shape[1])
        print("density: %f" % density(clf.coef_))

        if opts.print_top10 and feature_names is not None:
            print("top 10 keywords per class:")
            for i, label in enumerate(target_names):
                top10 = np.argsort(clf.coef_[i])[-10:]
                print(trim("%s: %s" % (label, " ".join(feature_names[top10]))))
```

(continues on next page)

(continued from previous page)

```
        print()

    if opts.print_report:
        print("classification report:")
        print(metrics.classification_report(y_test, pred,
                                            target_names=target_names))

    if opts.print_cm:
        print("confusion matrix:")
        print(metrics.confusion_matrix(y_test, pred))

    print()
    clf_descr = str(clf).split('(')[0]
    return clf_descr, score, train_time, test_time


results = []
for clf, name in (
        (RidgeClassifier(tol=1e-2, solver="sag"), "Ridge Classifier"),
        (Perceptron(max_iter=50), "Perceptron"),
        (PassiveAggressiveClassifier(max_iter=50),
         "Passive-Aggressive"),
        (KNeighborsClassifier(n_neighbors=10), "kNN"),
        (RandomForestClassifier(), "Random forest")):
    print('=' * 80)
    print(name)
    results.append(benchmark(clf))

for penalty in ["l2", "l1"]:
    print('=' * 80)
    print("%s penalty" % penalty.upper())
    # Train Liblinear model
    results.append(benchmark(LinearSVC(penalty=penalty, dual=False,
                                       tol=1e-3)))

    # Train SGD model
    results.append(benchmark(SGDClassifier(alpha=.0001, max_iter=50,
                                           penalty=penalty)))

# Train SGD with Elastic Net penalty
print('=' * 80)
print("Elastic-Net penalty")
results.append(benchmark(SGDClassifier(alpha=.0001, max_iter=50,
                                       penalty="elasticnet")))

# Train NearestCentroid without threshold
print('=' * 80)
print("NearestCentroid (aka Rocchio classifier)")
results.append(benchmark(NearestCentroid()))

# Train sparse Naive Bayes classifiers
print('=' * 80)
print("Naive Bayes")
results.append(benchmark(MultinomialNB(alpha=.01)))
results.append(benchmark(BernoulliNB(alpha=.01)))
results.append(benchmark(ComplementNB(alpha=.1)))
```

(continues on next page)

(continued from previous page)

```
print('=' * 80)
print("LinearSVC with L1-based feature selection")
# The smaller C, the stronger the regularization.
# The more regularization, the more sparsity.
results.append(benchmark(Pipeline([
  ('feature_selection', SelectFromModel(LinearSVC(penalty="l1", dual=False,
                                                  tol=1e-3))),
  ('classification', LinearSVC(penalty="l2"))])))
```

Out:

```
================================================================================
Ridge Classifier
________________________________________________________________________________
Training:
RidgeClassifier(solver='sag', tol=0.01)
/home/circleci/project/sklearn/linear_model/_ridge.py:557: UserWarning: "sag" solver␣
↪requires many iterations to fit an intercept with sparse inputs. Either set the␣
↪solver to "auto" or "sparse_cg", or set a low "tol" and a high "max_iter"␣
↪(especially if inputs are not standardized).
  warnings.warn(
train time: 0.209s
test time:  0.003s
accuracy:   0.897
dimensionality: 33809
density: 1.000000


================================================================================
Perceptron
________________________________________________________________________________
Training:
Perceptron(max_iter=50)
train time: 0.019s
test time:  0.002s
accuracy:   0.888
dimensionality: 33809
density: 0.255302


================================================================================
Passive-Aggressive
________________________________________________________________________________
Training:
PassiveAggressiveClassifier(max_iter=50)
train time: 0.032s
test time:  0.002s
accuracy:   0.902
dimensionality: 33809
density: 0.692841


================================================================================
kNN
________________________________________________________________________________
Training:
KNeighborsClassifier(n_neighbors=10)
```

(continues on next page)

(continued from previous page)

```
train time: 0.003s
test time:  0.215s
accuracy:   0.858

================================================================================
Random forest
________________________________________________________________________________
Training:
RandomForestClassifier()
train time: 1.682s
test time:  0.072s
accuracy:   0.837

================================================================================
L2 penalty
________________________________________________________________________________
Training:
LinearSVC(dual=False, tol=0.001)
train time: 0.075s
test time:  0.001s
accuracy:   0.900
dimensionality: 33809
density: 1.000000



________________________________________________________________________________
Training:
SGDClassifier(max_iter=50)
train time: 0.022s
test time:  0.002s
accuracy:   0.899
dimensionality: 33809
density: 0.569944


================================================================================
L1 penalty
________________________________________________________________________________
Training:
LinearSVC(dual=False, penalty='l1', tol=0.001)
train time: 0.217s
test time:  0.001s
accuracy:   0.873
dimensionality: 33809
density: 0.005553



________________________________________________________________________________
Training:
SGDClassifier(max_iter=50, penalty='l1')
train time: 0.094s
test time:  0.002s
accuracy:   0.888
dimensionality: 33809
density: 0.022982
```

(continues on next page)

(continued from previous page)

```
================================================================================
Elastic-Net penalty
________________________________________________________________________________
Training:
SGDClassifier(max_iter=50, penalty='elasticnet')
train time: 0.123s
test time:  0.002s
accuracy:   0.902
dimensionality: 33809
density: 0.187502


================================================================================
NearestCentroid (aka Rocchio classifier)
________________________________________________________________________________
Training:
NearestCentroid()
train time: 0.005s
test time:  0.002s
accuracy:   0.855

================================================================================
Naive Bayes
________________________________________________________________________________
Training:
MultinomialNB(alpha=0.01)
train time: 0.008s
test time:  0.001s
accuracy:   0.899
dimensionality: 33809
density: 1.000000



________________________________________________________________________________
Training:
BernoulliNB(alpha=0.01)
train time: 0.009s
test time:  0.007s
accuracy:   0.884
dimensionality: 33809
density: 1.000000



________________________________________________________________________________
Training:
ComplementNB(alpha=0.1)
train time: 0.007s
test time:  0.001s
accuracy:   0.911
dimensionality: 33809
density: 1.000000


================================================================================
LinearSVC with L1-based feature selection
________________________________________________________________________________
Training:
```

(continues on next page)

(continued from previous page)

```
Pipeline(steps=[('feature_selection',
                 SelectFromModel(estimator=LinearSVC(dual=False, penalty='l1',
                                                     tol=0.001))),
                ('classification', LinearSVC())])
train time: 0.213s
test time:  0.003s
accuracy:   0.880
```

Add plots

The bar plot indicates the accuracy, training time (normalized) and test time (normalized) of each classifier.

```
indices = np.arange(len(results))

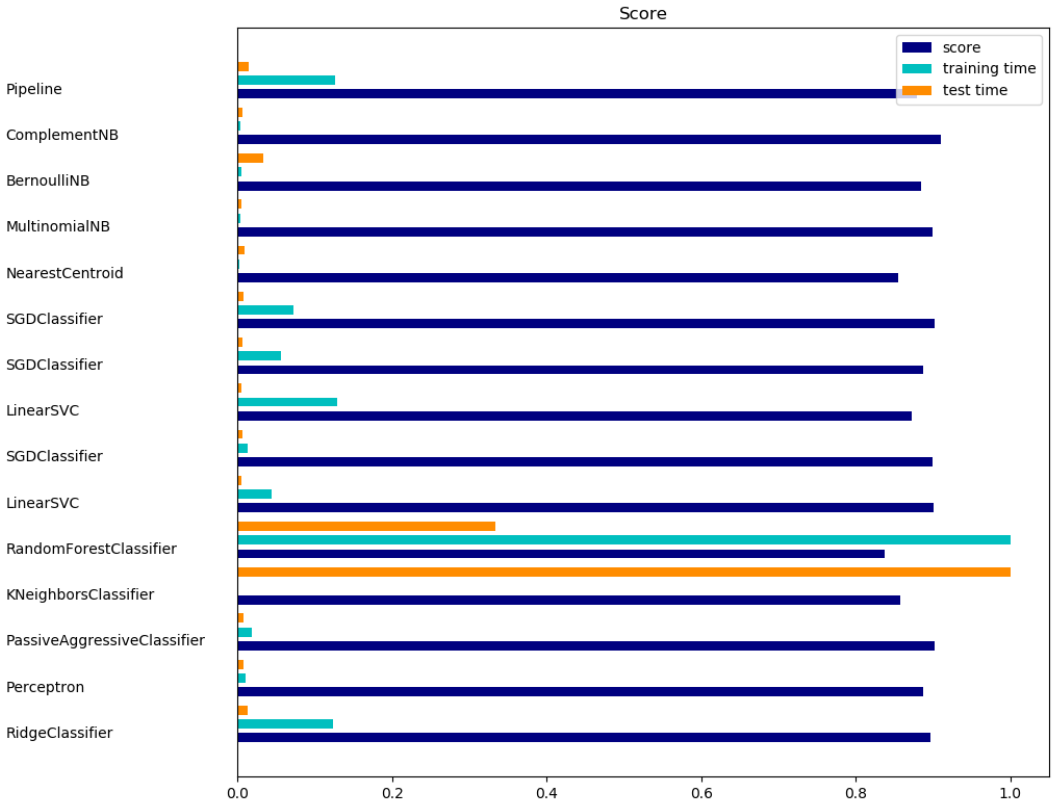
results = [[x[i] for x in results] for i in range(4)]

clf_names, score, training_time, test_time = results
training_time = np.array(training_time) / np.max(training_time)
test_time = np.array(test_time) / np.max(test_time)

plt.figure(figsize=(12, 8))
plt.title("Score")
plt.barh(indices, score, .2, label="score", color='navy')
plt.barh(indices + .3, training_time, .2, label="training time",
         color='c')
plt.barh(indices + .6, test_time, .2, label="test time", color='darkorange')
plt.yticks(())
plt.legend(loc='best')
plt.subplots_adjust(left=.25)
plt.subplots_adjust(top=.95)
plt.subplots_adjust(bottom=.05)

for i, c in zip(indices, clf_names):
    plt.text(-.3, i, c)

plt.show()
```



Total running time of the script: (0 minutes 5.861 seconds)

Estimated memory usage: 46 MB

CHAPTER

SEVEN

API REFERENCE

This is the class and function reference of scikit-learn. Please refer to the *full user guide* for further details, as the class and function raw specifications may not be enough to give full guidelines on their uses. For reference on concepts repeated across the API, see *Glossary of Common Terms and API Elements*.

7.1 `sklearn.base`: Base classes and utility functions

Base classes for all estimators.

Used for VotingClassifier

7.1.1 Base classes

base.BaseEstimator	Base class for all estimators in scikit-learn
base.BiclusterMixin	Mixin class for all bicluster estimators in scikit-learn
base.ClassifierMixin	Mixin class for all classifiers in scikit-learn.
base.ClusterMixin	Mixin class for all cluster estimators in scikit-learn.
base.DensityMixin	Mixin class for all density estimators in scikit-learn.
base.RegressorMixin	Mixin class for all regression estimators in scikit-learn.
base.TransformerMixin	Mixin class for all transformers in scikit-learn.

`sklearn.base.BaseEstimator`

class sklearn.base.**BaseEstimator**

Base class for all estimators in scikit-learn

Notes

All estimators should specify all the parameters that can be set at the class level in their `__init__` as explicit keyword arguments (no `*args` or `**kwargs`).

Methods

get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.base.BaseEstimator`

- *Approximate nearest neighbors in TSNE*

sklearn.base.BiclusterMixin

class sklearn.base.**BiclusterMixin**
Mixin class for all bicluster estimators in scikit-learn

Attributes

biclusters_ Convenient way to get row and column indicators together.

Methods

get_indices(self, i)	Row and column indices of the i'th bicluster.
get_shape(self, i)	Shape of the i'th bicluster.
get_submatrix(self, i, data)	Return the submatrix corresponding to bicluster `i`.

__init__ (*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

property biclusters_
Convenient way to get row and column indicators together.

Returns the `rows_` and `columns_` members.

get_indices (*self*, *i*)

Row and column indices of the i'th bicluster.

Only works if `rows_` and `columns_` attributes exist.

Parameters

i [int] The index of the cluster.

Returns

row_ind [np.array, dtype=np.intp] Indices of rows in the dataset that belong to the bicluster.

col_ind [np.array, dtype=np.intp] Indices of columns in the dataset that belong to the bicluster.

get_shape(*self*, *i*)
Shape of the i'th bicluster.

Parameters

i [int] The index of the cluster.

Returns

shape [(int, int)] Number of rows and columns (resp.) in the bicluster.

get_submatrix(*self*, *i*, *data*)
Return the submatrix corresponding to bicluster `i`.

Parameters

i [int] The index of the cluster.

data [array] The data.

Returns

submatrix [array] The submatrix corresponding to bicluster i.

Notes

Works with sparse matrices. Only works if `rows_` and `columns_` attributes exist.

sklearn.base.ClassifierMixin

class sklearn.base.**ClassifierMixin**
Mixin class for all classifiers in scikit-learn.

Methods

score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.

__init__(*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each

sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

sklearn.base.ClusterMixin

class sklearn.base.**ClusterMixin**

Mixin class for all cluster estimators in scikit-learn.

Methods

fit_predict(self, X[, y])	Perform clustering on X and returns cluster labels.

__init__(*self*, */*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

fit_predict(*self*, *X*, *y=None*)

Perform clustering on X and returns cluster labels.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

y [Ignored] Not used, present for API consistency by convention.

Returns

labels [ndarray, shape (n_samples,)] Cluster labels.

sklearn.base.DensityMixin

class sklearn.base.**DensityMixin**

Mixin class for all density estimators in scikit-learn.

Methods

score(self, X[, y])	Return the score of the model on the data X

__init__(*self*, */*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

score(*self*, *X*, *y=None*)

Return the score of the model on the data X

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

score [float]

sklearn.base.RegressorMixin

class sklearn.base.**RegressorMixin**

Mixin class for all regression estimators in scikit-learn.

Methods

score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.

__init__ (*self*, */*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

sklearn.base.TransformerMixin

class sklearn.base.**TransformerMixin**

Mixin class for all transformers in scikit-learn.

Methods

fit_transform(self, X[, y])	Fit to data, then transform it.

__init__ (*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

Examples using sklearn.base.TransformerMixin

- *Approximate nearest neighbors in TSNE*

7.1.2 Functions

base.clone(estimator[, safe])	Constructs a new estimator with the same parameters.
base.is_classifier(estimator)	Return True if the given estimator is (probably) a classifier.
base.is_regressor(estimator)	Return True if the given estimator is (probably) a regressor.
config_context(**new_config)	Context manager for global scikit-learn configuration
get_config()	Retrieve current values for configuration set by *set_config*
set_config([assume_finite, working_memory, ...])	Set global scikit-learn configuration
show_versions()	Print useful debugging information

sklearn.base.clone

sklearn.base.**clone** (*estimator*, *safe=True*)
Constructs a new estimator with the same parameters.

Clone does a deep copy of the model in an estimator without actually copying attached data. It yields a new estimator with the same parameters that has not been fit on any data.

Parameters

estimator [estimator object, or list, tuple or set of objects] The estimator or group of estimators to be cloned

safe [boolean, optional] If safe is false, clone will fall back to a deep copy on objects that are not estimators.

sklearn.base.is_classifier

sklearn.base.**is_classifier**(*estimator*)
Return True if the given estimator is (probably) a classifier.

Parameters

estimator [object] Estimator object to test.

Returns

out [bool] True if estimator is a classifier and False otherwise.

sklearn.base.is_regressor

sklearn.base.**is_regressor**(*estimator*)
Return True if the given estimator is (probably) a regressor.

Parameters

estimator [object] Estimator object to test.

Returns

out [bool] True if estimator is a regressor and False otherwise.

sklearn.config_context

sklearn.**config_context**(***new_config*)
Context manager for global scikit-learn configuration

Parameters

assume_finite [bool, optional] If True, validation for finiteness will be skipped, saving time, but leading to potential crashes. If False, validation for finiteness will be performed, avoiding error. Global default: False.

working_memory [int, optional] If set, scikit-learn will attempt to limit the size of temporary arrays to this number of MiB (per job when parallelised), often saving both computation time and memory on expensive operations that can be performed in chunks. Global default: 1024.

print_changed_only [bool, optional] If True, only the parameters that were set to non-default values will be printed when printing an estimator. For example, `print(SVC())` while True will only print 'SVC()' while the default behaviour would be to print 'SVC(C=1.0, cache_size=200, ...)' with all the non-changed parameters.

See also:

set_config Set global scikit-learn configuration

get_config Retrieve current values of the global configuration

Notes

All settings, not just those presently modified, will be returned to their previous values when the context manager is exited. This is not thread-safe.

Examples

```
>>> import sklearn
>>> from sklearn.utils.validation import assert_all_finite
>>> with sklearn.config_context(assume_finite=True):
...     assert_all_finite([float('nan')])
>>> with sklearn.config_context(assume_finite=True):
...     with sklearn.config_context(assume_finite=False):
...         assert_all_finite([float('nan')])
Traceback (most recent call last):
...
ValueError: Input contains NaN, ...
```

sklearn.get_config

sklearn.**get_config**()

Retrieve current values for configuration set by *set_config*

Returns

config [dict] Keys are parameter names that can be passed to *set_config*.

See also:

config_context Context manager for global scikit-learn configuration

set_config Set global scikit-learn configuration

sklearn.set_config

sklearn.**set_config**(*assume_finite=None*, *working_memory=None*, *print_changed_only=None*)

Set global scikit-learn configuration

New in version 0.19.

Parameters

assume_finite [bool, optional] If True, validation for finiteness will be skipped, saving time, but leading to potential crashes. If False, validation for finiteness will be performed, avoiding error. Global default: False.

New in version 0.19.

working_memory [int, optional] If set, scikit-learn will attempt to limit the size of temporary arrays to this number of MiB (per job when parallelised), often saving both computation time and memory on expensive operations that can be performed in chunks. Global default: 1024.

New in version 0.20.

print_changed_only [bool, optional] If True, only the parameters that were set to non-default values will be printed when printing an estimator. For example, `print(SVC())` while True will only print 'SVC()' while the default behaviour would be to print 'SVC(C=1.0, cache_size=200, ...)' with all the non-changed parameters.

New in version 0.21.

See also:

config_context Context manager for global scikit-learn configuration

get_config Retrieve current values of the global configuration

Examples using sklearn.set_config

- *Compact estimator representations*

sklearn.show_versions

sklearn.**show_versions**()
Print useful debugging information

7.2 sklearn.calibration: Probability Calibration

Calibration of predicted probabilities.

User guide: See the *Probability calibration* section for further details.

calibration.CalibratedClassifierCV([...])	Probability calibration with isotonic regression or sigmoid.

7.2.1 sklearn.calibration.CalibratedClassifierCV

class sklearn.calibration.**CalibratedClassifierCV**(*base_estimator=None*, *method='sigmoid'*, *cv=None*)

Probability calibration with isotonic regression or sigmoid.

See glossary entry for *cross-validation estimator*.

With this class, the base_estimator is fit on the train set of the cross-validation generator and the test set is used for calibration. The probabilities for each of the folds are then averaged for prediction. In case that cv="prefit" is passed to __init__, it is assumed that base_estimator has been fitted already and all data is used for calibration. Note that data for fitting the classifier and for calibrating it must be disjoint.

Read more in the *User Guide*.

Parameters

base_estimator [instance BaseEstimator] The classifier whose output decision function needs to be calibrated to offer more accurate predict_proba outputs. If cv=prefit, the classifier must have been fit already on data.

method ['sigmoid' or 'isotonic'] The method to use for calibration. Can be 'sigmoid' which corresponds to Platt's method or 'isotonic' which is a non-parametric approach. It is not advised to use isotonic calibration with too few calibration samples (<<1000) since it tends to overfit. Use sigmoids (Platt's calibration) in this case.

cv [integer, cross-validation generator, iterable or "prefit", optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross-validation,
- integer, to specify the number of folds.

- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if `y` is binary or multiclass, *sklearn.model_selection.StratifiedKFold* is used. If `y` is neither binary nor multiclass, *sklearn.model_selection.KFold* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

If "prefit" is passed, it is assumed that base_estimator has been fitted already and all data is used for calibration.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

Attributes

classes_ [array, shape (n_classes)] The class labels.

calibrated_classifiers_ [list (len() equal to cv or 1 if cv == "prefit")] The list of calibrated classifiers, one for each crossvalidation fold, which has been fitted on all but the validation fold and calibrated on the validation fold.

References

[R57cf438d7060-1], [R57cf438d7060-2], [R57cf438d7060-3], [R57cf438d7060-4]

Methods

fit(self, X, y[, sample_weight])	Fit the calibrated model
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the target of new samples.
predict_proba(self, X)	Posterior probabilities of classification
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *base_estimator=None*, *method='sigmoid'*, *cv=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit the calibrated model

Parameters

X [array-like, shape (n_samples, n_features)] Training data.

y [array-like, shape (n_samples,)] Target values.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted.

Returns

self [object] Returns an instance of self.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)

Predict the target of new samples. Can be different from the prediction of the uncalibrated classifier.

Parameters

X [array-like, shape (n_samples, n_features)] The samples.

Returns

C [array, shape (n_samples,)] The predicted class.

predict_proba(*self*, *X*)

Posterior probabilities of classification

This function returns posterior probabilities of classification according to each class on an array of test vectors X.

Parameters

X [array-like, shape (n_samples, n_features)] The samples.

Returns

C [array, shape (n_samples, n_classes)] The predicted probas.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.calibration.CalibratedClassifierCV`

- *Probability Calibration curves*
- *Probability calibration of classifiers*
- *Probability Calibration for 3-class classification*

`calibration.calibration_curve`(y_true, y_prob)	Compute true and predicted probabilities for a calibration curve.

7.2.2 `sklearn.calibration`.calibration_curve

`sklearn.calibration.`**`calibration_curve`**(*y_true*, *y_prob*, *normalize=False*, *n_bins=5*, *strategy='uniform'*)

Compute true and predicted probabilities for a calibration curve.

The method assumes the inputs come from a binary classifier.

Calibration curves may also be referred to as reliability diagrams.

Read more in the *User Guide*.

Parameters

y_true [array, shape (n_samples,)] True targets.

y_prob [array, shape (n_samples,)] Probabilities of the positive class.

normalize [bool, optional, default=False] Whether y_prob needs to be normalized into the bin [0, 1], i.e. is not a proper probability. If True, the smallest value in y_prob is mapped onto 0 and the largest one onto 1.

n_bins [int] Number of bins. A bigger number requires more data. Bins with no data points (i.e. without corresponding values in y_prob) will not be returned, thus there may be fewer than n_bins in the return value.

strategy [{'uniform', 'quantile'}, (default='uniform')] Strategy used to define the widths of the bins.

uniform All bins have identical widths.

quantile All bins have the same number of points.

Returns

prob_true [array, shape (n_bins,) or smaller] The true probability in each bin (fraction of positives).

prob_pred [array, shape (n_bins,) or smaller] The mean predicted probability in each bin.

References

Alexandru Niculescu-Mizil and Rich Caruana (2005) Predicting Good Probabilities With Supervised Learning, in Proceedings of the 22nd International Conference on Machine Learning (ICML). See section 4 (Qualitative Analysis of Predictions).

Examples using `sklearn.calibration.calibration_curve`

- *Comparison of Calibration of Classifiers*
- *Probability Calibration curves*

7.3 `sklearn.cluster`: Clustering

The `sklearn.cluster` module gathers popular unsupervised clustering algorithms.

User guide: See the *Clustering* and *Biclustering* sections for further details.

7.3.1 Classes

`cluster.AffinityPropagation`([damping, ...])	Perform Affinity Propagation Clustering of data.
`cluster.AgglomerativeClustering`([...])	Agglomerative Clustering
`cluster.Birch`([threshold, branching_factor, ...])	Implements the Birch clustering algorithm.
`cluster.DBSCAN`([eps, min_samples, metric, ...])	Perform DBSCAN clustering from vector array or distance matrix.
`cluster.FeatureAgglomeration`([n_clusters, ...])	Agglomerate features.
`cluster.KMeans`([n_clusters, init, n_init, ...])	K-Means clustering.
`cluster.MiniBatchKMeans`([n_clusters, init, ...])	Mini-Batch K-Means clustering.
`cluster.MeanShift`([bandwidth, seeds, ...])	Mean shift clustering using a flat kernel.
`cluster.OPTICS`([min_samples, max_eps, ...])	Estimate clustering structure from vector array.
`cluster.SpectralClustering`([n_clusters, ...])	Apply clustering to a projection of the normalized Laplacian.
`cluster.SpectralBiclustering`([n_clusters, ...])	Spectral biclustering (Kluger, 2003).
`cluster.SpectralCoclustering`([n_clusters, ...])	Spectral Co-Clustering algorithm (Dhillon, 2001).

`sklearn.cluster.AffinityPropagation`

class sklearn.cluster.**AffinityPropagation**(*damping=0.5*, *max_iter=200*, *convergence_iter=15*, *copy=True*, *preference=None*, *affinity='euclidean'*, *verbose=False*)

Perform Affinity Propagation Clustering of data.

Read more in the *User Guide*.

Parameters

damping [float, default=0.5] Damping factor (between 0.5 and 1) is the extent to which the current value is maintained relative to incoming values (weighted 1 - damping). This in order to avoid numerical oscillations when updating these values (messages).

max_iter [int, default=200] Maximum number of iterations.

convergence_iter [int, default=15] Number of iterations with no change in the number of estimated clusters that stops the convergence.

copy [bool, default=True] Make a copy of input data.

preference [array-like of shape (n_samples,) or float, default=None] Preferences for each point - points with larger values of preferences are more likely to be chosen as exemplars. The number of exemplars, ie of clusters, is influenced by the input preferences value. If the preferences are not passed as arguments, they will be set to the median of the input similarities.

affinity [{'euclidean', 'precomputed'}, default='euclidean'] Which affinity to use. At the moment 'precomputed' and `euclidean` are supported. 'euclidean' uses the negative squared euclidean distance between points.

verbose [bool, default=False] Whether to be verbose.

Attributes

cluster_centers_indices_ [ndarray of shape (n_clusters,)] Indices of cluster centers

cluster_centers_ [ndarray of shape (n_clusters, n_features)] Cluster centers (if affinity != `precomputed`).

labels_ [ndarray of shape (n_samples,)] Labels of each point

affinity_matrix_ [ndarray of shape (n_samples, n_samples)] Stores the affinity matrix used in `fit`.

n_iter_ [int] Number of iterations taken to converge.

Notes

For an example, see *examples/cluster/plot_affinity_propagation.py*.

The algorithmic complexity of affinity propagation is quadratic in the number of points.

When `fit` does not converge, `cluster_centers_` becomes an empty array and all training samples will be labelled as `-1`. In addition, `predict` will then label every sample as `-1`.

When all training samples have equal similarities and equal preferences, the assignment of cluster centers and labels depends on the preference. If the preference is smaller than the similarities, `fit` will result in a single cluster center and label `0` for every sample. Otherwise, every training sample becomes its own cluster center and is assigned a unique label.

References

Brendan J. Frey and Delbert Dueck, "Clustering by Passing Messages Between Data Points", Science Feb. 2007

Examples

```
>>> from sklearn.cluster import AffinityPropagation
>>> import numpy as np
>>> X = np.array([[1, 2], [1, 4], [1, 0],
...               [4, 2], [4, 4], [4, 0]])
>>> clustering = AffinityPropagation().fit(X)
>>> clustering
AffinityPropagation()
>>> clustering.labels_
array([0, 0, 0, 1, 1, 1])
>>> clustering.predict([[0, 0], [4, 4]])
array([0, 1])
```

(continues on next page)

(continued from previous page)

```
>>> clustering.cluster_centers_
array([[1, 2],
       [4, 2]])
```

Methods

fit(self, X[, y])	Fit the clustering from features, or affinity matrix.
fit_predict(self, X[, y])	Fit the clustering from features or affinity matrix, and return cluster labels.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the closest cluster each sample in X belongs to.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *damping=0.5*, *max_iter=200*, *convergence_iter=15*, *copy=True*, *preference=None*, *affinity='euclidean'*, *verbose=False*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*)
Fit the clustering from features, or affinity matrix.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features), or array-like, shape (n_samples, n_samples)] Training instances to cluster, or similarities / affinities between instances if `affinity='precomputed'`. If a sparse feature matrix is provided, it will be converted into a sparse `csr_matrix`.

y [Ignored] Not used, present here for API consistency by convention.

Returns

self

fit_predict (*self*, *X*, *y=None*)
Fit the clustering from features or affinity matrix, and return cluster labels.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features), or array-like, shape (n_samples, n_samples)] Training instances to cluster, or similarities / affinities between instances if `affinity='precomputed'`. If a sparse feature matrix is provided, it will be converted into a sparse `csr_matrix`.

y [Ignored] Not used, present here for API consistency by convention.

Returns

labels [ndarray, shape (n_samples,)] Cluster labels.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)

Predict the closest cluster each sample in X belongs to.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] New data to predict. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

labels [ndarray, shape (n_samples,)] Cluster labels.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.cluster.AffinityPropagation`

- *Demo of affinity propagation clustering algorithm*
- *Comparing different clustering algorithms on toy datasets*

sklearn.cluster.AgglomerativeClustering

class sklearn.cluster.**AgglomerativeClustering**(*n_clusters=2*, *affinity='euclidean'*, *memory=None*, *connectivity=None*, *compute_full_tree='auto'*, *linkage='ward'*, *distance_threshold=None*)

Agglomerative Clustering

Recursively merges the pair of clusters that minimally increases a given linkage distance.

Read more in the *User Guide*.

Parameters

n_clusters [int or None, default=2] The number of clusters to find. It must be `None` if `distance_threshold` is not `None`.

affinity [str or callable, default='euclidean'] Metric used to compute the linkage. Can be "euclidean", "l1", "l2", "manhattan", "cosine", or "precomputed". If linkage is "ward", only "euclidean" is accepted. If "precomputed", a distance matrix (instead of a similarity matrix) is needed as input for the fit method.

memory [str or object with the joblib.Memory interface, default=None] Used to cache the output of the computation of the tree. By default, no caching is done. If a string is given, it is the path to the caching directory.

connectivity [array-like or callable, default=None] Connectivity matrix. Defines for each sample the neighboring samples following a given structure of the data. This can be a connectivity matrix itself or a callable that transforms the data into a connectivity matrix, such as derived from kneighbors_graph. Default is None, i.e, the hierarchical clustering algorithm is unstructured.

compute_full_tree ['auto' or bool, default='auto'] Stop early the construction of the tree at n_clusters. This is useful to decrease computation time if the number of clusters is not small compared to the number of samples. This option is useful only when specifying a connectivity matrix. Note also that when varying the number of clusters and using caching, it may be advantageous to compute the full tree. It must be `True` if `distance_threshold` is not `None`. By default `compute_full_tree` is "auto", which is equivalent to `True` when `distance_threshold` is not `None` or that `n_clusters` is inferior to the maximum between 100 or `0.02 * n_samples`. Otherwise, "auto" is equivalent to `False`.

linkage [{"ward", "complete", "average", "single"}, default="ward"] Which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of cluster that minimize this criterion.

- ward minimizes the variance of the clusters being merged.
- average uses the average of the distances of each observation of the two sets.
- complete or maximum linkage uses the maximum distances between all observations of the two sets.
- single uses the minimum of the distances between all observations of the two sets.

distance_threshold [float, default=None] The linkage distance threshold above which, clusters will not be merged. If not `None`, `n_clusters` must be `None` and `compute_full_tree` must be `True`.

New in version 0.21.

Attributes

n_clusters_ [int] The number of clusters found by the algorithm. If `distance_threshold=None`, it will be equal to the given `n_clusters`.

labels_ [ndarray of shape (n_samples)] cluster labels for each point

n_leaves_ [int] Number of leaves in the hierarchical tree.

n_connected_components_ [int] The estimated number of connected components in the graph.

children_ [array-like of shape (n_samples-1, 2)] The children of each non-leaf node. Values less than `n_samples` correspond to leaves of the tree which are the original samples. A node `i` greater than or equal to `n_samples` is a non-leaf node and has children `children_[i - n_samples]`. Alternatively at the i-th iteration, children[i][0] and children[i][1] are merged to form node `n_samples + i`

Examples

```
>>> from sklearn.cluster import AgglomerativeClustering
>>> import numpy as np
>>> X = np.array([[1, 2], [1, 4], [1, 0],
...               [4, 2], [4, 4], [4, 0]])
>>> clustering = AgglomerativeClustering().fit(X)
>>> clustering
AgglomerativeClustering()
```

(continues on next page)

(continued from previous page)

```
>>> clustering.labels_
array([1, 1, 1, 0, 0, 0])
```

Methods

fit(self, X[, y])	Fit the hierarchical clustering from features, or distance matrix.
fit_predict(self, X[, y])	Fit the hierarchical clustering from features or distance matrix, and return cluster labels.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_clusters=2*, *affinity='euclidean'*, *memory=None*, *connectivity=None*, *compute_full_tree='auto'*, *linkage='ward'*, *distance_threshold=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the hierarchical clustering from features, or distance matrix.

Parameters

X [array-like, shape (n_samples, n_features) or (n_samples, n_samples)] Training instances to cluster, or distances between instances if `affinity='precomputed'`.

y [Ignored] Not used, present here for API consistency by convention.

Returns

self

fit_predict(*self*, *X*, *y=None*)
Fit the hierarchical clustering from features or distance matrix, and return cluster labels.

Parameters

X [array-like, shape (n_samples, n_features) or (n_samples, n_samples)] Training instances to cluster, or distances between instances if `affinity='precomputed'`.

y [Ignored] Not used, present here for API consistency by convention.

Returns

labels [ndarray, shape (n_samples,)] Cluster labels.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.cluster.AgglomerativeClustering`

- *Plot Hierarchical Clustering Dendrogram*
- *Agglomerative clustering with and without structure*
- *Various Agglomerative Clustering on a 2D embedding of digits*
- *A demo of structured Ward hierarchical clustering on an image of coins*
- *Hierarchical clustering: structured vs unstructured ward*
- *Agglomerative clustering with different metrics*
- *Inductive Clustering*
- *Comparing different hierarchical linkage methods on toy datasets*
- *Comparing different clustering algorithms on toy datasets*

`sklearn.cluster`.Birch

class sklearn.cluster.**Birch**(*threshold=0.5*, *branching_factor=50*, *n_clusters=3*, *compute_labels=True*, *copy=True*)

Implements the Birch clustering algorithm.

It is a memory-efficient, online-learning algorithm provided as an alternative to `MiniBatchKMeans`. It constructs a tree data structure with the cluster centroids being read off the leaf. These can be either the final cluster centroids or can be provided as input to another clustering algorithm such as `AgglomerativeClustering`.

Read more in the *User Guide*.

New in version 0.16.

Parameters

threshold [float, default=0.5] The radius of the subcluster obtained by merging a new sample and the closest subcluster should be lesser than the threshold. Otherwise a new subcluster is started. Setting this value to be very low promotes splitting and vice-versa.

branching_factor [int, default=50] Maximum number of CF subclusters in each node. If a new samples enters such that the number of subclusters exceed the branching_factor then that node is split into two nodes with the subclusters redistributed in each. The parent subcluster of that node is removed and two new subclusters are added as parents of the 2 split nodes.

n_clusters [int, instance of sklearn.cluster model, default=3] Number of clusters after the final clustering step, which treats the subclusters from the leaves as new samples.

- `None` : the final clustering step is not performed and the subclusters are returned as they are.

- *sklearn.cluster* Estimator : If a model is provided, the model is fit treating the subclusters as new samples and the initial data is mapped to the label of the closest subcluster.
- int : the model fit is *AgglomerativeClustering* with n_clusters set to be equal to the int.

compute_labels [bool, default=True] Whether or not to compute labels for each fit.

copy [bool, default=True] Whether or not to make a copy of the given data. If set to False, the initial data will be overwritten.

Attributes

root_ [_CFNode] Root of the CFTree.

dummy_leaf_ [_CFNode] Start pointer to all the leaves.

subcluster_centers_ [ndarray,] Centroids of all subclusters read directly from the leaves.

subcluster_labels_ [ndarray,] Labels assigned to the centroids of the subclusters after they are clustered globally.

labels_ [ndarray, shape (n_samples,)] Array of labels assigned to the input data. if partial_fit is used instead of fit, they are assigned to the last batch of data.

See also:

MiniBatchKMeans Alternative implementation that does incremental updates of the centers' positions using mini-batches.

Notes

The tree data structure consists of nodes with each node consisting of a number of subclusters. The maximum number of subclusters in a node is determined by the branching factor. Each subcluster maintains a linear sum, squared sum and the number of samples in that subcluster. In addition, each subcluster can also have a node as its child, if the subcluster is not a member of a leaf node.

For a new point entering the root, it is merged with the subcluster closest to it and the linear sum, squared sum and the number of samples of that subcluster are updated. This is done recursively till the properties of the leaf node are updated.

References

- Tian Zhang, Raghu Ramakrishnan, Maron Livny BIRCH: An efficient data clustering method for large databases. https://www.cs.sfu.ca/CourseCentral/459/han/papers/zhang96.pdf
- Roberto Perdisci JBirch - Java implementation of BIRCH clustering algorithm https://code.google.com/archive/p/jbirch

Examples

```
>>> from sklearn.cluster import Birch
>>> X = [[0, 1], [0.3, 1], [-0.3, 1], [0, -1], [0.3, -1], [-0.3, -1]]
>>> brc = Birch(n_clusters=None)
>>> brc.fit(X)
Birch(n_clusters=None)
```

(continues on next page)

(continued from previous page)

```
>>> brc.predict(X)
array([0, 0, 0, 1, 1, 1])
```

Methods

fit(self, X[, y])	Build a CF Tree for the input data.
fit_predict(self, X[, y])	Perform clustering on X and returns cluster labels.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self[, X, y])	Online learning.
predict(self, X)	Predict data using the `centroids_` of subclusters.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform X into subcluster centroids dimension.

__init__(*self*, *threshold=0.5*, *branching_factor=50*, *n_clusters=3*, *compute_labels=True*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Build a CF Tree for the input data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Input data.

y [Ignored] Not used, present here for API consistency by convention.

Returns

self Fitted estimator.

fit_predict(*self*, *X*, *y=None*)
Perform clustering on X and returns cluster labels.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

y [Ignored] Not used, present for API consistency by convention.

Returns

labels [ndarray, shape (n_samples,)] Cluster labels.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit(*self*, *X=None*, *y=None*)
Online learning. Prevents rebuilding of CFTree from scratch.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features), None] Input data. If X is not provided, only the global clustering step is done.

y [Ignored] Not used, present here for API consistency by convention.

Returns

self Fitted estimator.

predict(*self*, *X*)
Predict data using the `centroids_` of subclusters.

Avoid computation of the row norms of X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Input data.

Returns

labels [ndarray, shape(n_samples)] Labelled data.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Transform X into subcluster centroids dimension.

Each dimension represents the distance from the sample point to each cluster centroid.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Input data.

Returns

X_trans [{array-like, sparse matrix}, shape (n_samples, n_clusters)] Transformed data.

Examples using `sklearn.cluster.Birch`

- *Compare BIRCH and MiniBatchKMeans*
- *Comparing different clustering algorithms on toy datasets*

`sklearn.cluster.DBSCAN`

class sklearn.cluster.**DBSCAN**(*eps=0.5*, *min_samples=5*, *metric='euclidean'*, *metric_params=None*, *algorithm='auto'*, *leaf_size=30*, *p=None*, *n_jobs=None*)

Perform DBSCAN clustering from vector array or distance matrix.

DBSCAN - Density-Based Spatial Clustering of Applications with Noise. Finds core samples of high density and expands clusters from them. Good for data which contains clusters of similar density.

Read more in the *User Guide*.

Parameters

eps [float, default=0.5] The maximum distance between two samples for one to be considered as in the neighborhood of the other. This is not a maximum bound on the distances of points within a cluster. This is the most important DBSCAN parameter to choose appropriately for your data set and distance function.

min_samples [int, default=5] The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself.

metric [string, or callable, default='euclidean'] The metric to use when calculating distance between instances in a feature array. If metric is a string or callable, it must be one of the options allowed by `sklearn.metrics.pairwise_distances` for its metric parameter. If metric is "precomputed", X is assumed to be a distance matrix and must be square. X may be a *Glossary*, in which case only "nonzero" elements may be considered neighbors for DBSCAN.

New in version 0.17: metric *precomputed* to accept precomputed sparse matrix.

metric_params [dict, default=None] Additional keyword arguments for the metric function.

New in version 0.19.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'] The algorithm to be used by the NearestNeighbors module to compute pointwise distances and find nearest neighbors. See NearestNeighbors module documentation for details.

leaf_size [int, default=30] Leaf size passed to BallTree or cKDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p [float, default=None] The power of the Minkowski metric to be used to calculate distance between points.

n_jobs [int or None, default=None] The number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

core_sample_indices_ [array, shape = [n_core_samples]] Indices of core samples.

components_ [array, shape = [n_core_samples, n_features]] Copy of each core sample found by training.

labels_ [array, shape = [n_samples]] Cluster labels for each point in the dataset given to fit(). Noisy samples are given the label -1.

See also:

OPTICS A similar clustering at multiple values of eps. Our implementation is optimized for memory usage.

Notes

For an example, see *examples/cluster/plot_dbscan.py*.

This implementation bulk-computes all neighborhood queries, which increases the memory complexity to O(n.d) where d is the average number of neighbors, while original DBSCAN had memory complexity O(n). It may attract a higher memory complexity when querying these nearest neighborhoods, depending on the `algorithm`.

One way to avoid the query complexity is to pre-compute sparse neighborhoods in chunks using *NearestNeighbors.radius_neighbors_graph* with `mode='distance'`, then using `metric='precomputed'` here.

Another way to reduce memory and computation time is to remove (near-)duplicate points and use `sample_weight` instead.

`cluster.OPTICS` provides a similar clustering with lower memory usage.

References

Ester, M., H. P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press, pp. 226-231. 1996

Schubert, E., Sander, J., Ester, M., Kriegel, H. P., & Xu, X. (2017). DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. ACM Transactions on Database Systems (TODS), 42(3), 19.

Examples

```
>>> from sklearn.cluster import DBSCAN
>>> import numpy as np
>>> X = np.array([[1, 2], [2, 2], [2, 3],
...               [8, 7], [8, 8], [25, 80]])
>>> clustering = DBSCAN(eps=3, min_samples=2).fit(X)
>>> clustering.labels_
array([ 0,  0,  0,  1,  1, -1])
>>> clustering
DBSCAN(eps=3, min_samples=2)
```

Methods

fit(self, X[, y, sample_weight])	Perform DBSCAN clustering from features, or distance matrix.
fit_predict(self, X[, y, sample_weight])	Perform DBSCAN clustering from features or distance matrix, and return cluster labels.

Continued on next page

Table 17 – continued from previous page

get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *eps=0.5*, *min_samples=5*, *metric='euclidean'*, *metric_params=None*, *algorithm='auto'*, *leaf_size=30*, *p=None*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*, *sample_weight=None*)
Perform DBSCAN clustering from features, or distance matrix.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features), or (n_samples, n_samples)] Training instances to cluster, or distances between instances if `metric='precomputed'`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

sample_weight [array, shape (n_samples,), optional] Weight of each sample, such that a sample with a weight of at least `min_samples` is by itself a core sample; a sample with a negative weight may inhibit its eps-neighbor from being core. Note that weights are absolute, and default to 1.

y [Ignored] Not used, present here for API consistency by convention.

Returns

self

fit_predict(*self*, *X*, *y=None*, *sample_weight=None*)
Perform DBSCAN clustering from features or distance matrix, and return cluster labels.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features), or (n_samples, n_samples)] Training instances to cluster, or distances between instances if `metric='precomputed'`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

sample_weight [array, shape (n_samples,), optional] Weight of each sample, such that a sample with a weight of at least `min_samples` is by itself a core sample; a sample with a negative weight may inhibit its eps-neighbor from being core. Note that weights are absolute, and default to 1.

y [Ignored] Not used, present here for API consistency by convention.

Returns

labels [ndarray, shape (n_samples,)] Cluster labels. Noisy samples are given the label -1.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.cluster.DBSCAN`

- *Demo of DBSCAN clustering algorithm*
- *Demo of OPTICS clustering algorithm*
- *Comparing different clustering algorithms on toy datasets*

`sklearn.cluster`.FeatureAgglomeration

class sklearn.cluster.**FeatureAgglomeration**(*n_clusters=2*, *affinity='euclidean'*, *memory=None*, *connectivity=None*, *compute_full_tree='auto'*, *linkage='ward'*, *pooling_func=<function mean>*, *distance_threshold=None*)
Agglomerate features.

Similar to AgglomerativeClustering, but recursively merges features instead of samples.

Read more in the *User Guide*.

Parameters

n_clusters [int, default=2] The number of clusters to find. It must be `None` if `distance_threshold` is not `None`.

affinity [str or callable, default='euclidean'] Metric used to compute the linkage. Can be "euclidean", "l1", "l2", "manhattan", "cosine", or 'precomputed'. If linkage is "ward", only "euclidean" is accepted.

memory [str or object with the joblib.Memory interface, default=None] Used to cache the output of the computation of the tree. By default, no caching is done. If a string is given, it is the path to the caching directory.

connectivity [array-like or callable, default=None] Connectivity matrix. Defines for each feature the neighboring features following a given structure of the data. This can be a connectivity matrix itself or a callable that transforms the data into a connectivity matrix, such as derived from kneighbors_graph. Default is None, i.e, the hierarchical clustering algorithm is unstructured.

compute_full_tree ['auto' or bool, optional, default='auto'] Stop early the construction of the tree at n_clusters. This is useful to decrease computation time if the number of clusters is not small compared to the number of features. This option is useful only when specifying a connectivity matrix. Note also that when varying the number of clusters and using caching, it may be advantageous to compute the full tree. It must be

`True` if `distance_threshold` is not `None`. By default `compute_full_tree` is "auto", which is equivalent to `True` when `distance_threshold` is not `None` or that `n_clusters` is inferior to the maximum between 100 or `0.02 * n_samples`. Otherwise, "auto" is equivalent to `False`.

linkage [{'ward', 'complete', 'average', 'single'}, default='ward'] Which linkage criterion to use. The linkage criterion determines which distance to use between sets of features. The algorithm will merge the pairs of cluster that minimize this criterion.

- ward minimizes the variance of the clusters being merged.
- average uses the average of the distances of each feature of the two sets.
- complete or maximum linkage uses the maximum distances between all features of the two sets.
- single uses the minimum of the distances between all observations of the two sets.

pooling_func [callable, default=np.mean] This combines the values of agglomerated features into a single value, and should accept an array of shape [M, N] and the keyword argument `axis=1`, and reduce it to an array of size [M].

distance_threshold [float, default=None] The linkage distance threshold above which, clusters will not be merged. If not `None`, `n_clusters` must be `None` and `compute_full_tree` must be `True`.

New in version 0.21.

Attributes

n_clusters_ [int] The number of clusters found by the algorithm. If `distance_threshold=None`, it will be equal to the given `n_clusters`.

labels_ [array-like of (n_features,)] cluster labels for each feature.

n_leaves_ [int] Number of leaves in the hierarchical tree.

n_connected_components_ [int] The estimated number of connected components in the graph.

children_ [array-like of shape (n_nodes-1, 2)] The children of each non-leaf node. Values less than `n_features` correspond to leaves of the tree which are the original samples. A node `i` greater than or equal to `n_features` is a non-leaf node and has children `children_[i - n_features]`. Alternatively at the i-th iteration, children[i][0] and children[i][1] are merged to form node `n_features + i`

distances_ [array-like of shape (n_nodes-1,)] Distances between nodes in the corresponding place in `children_`. Only computed if distance_threshold is not None.

Examples

```
>>> import numpy as np
>>> from sklearn import datasets, cluster
>>> digits = datasets.load_digits()
>>> images = digits.images
>>> X = np.reshape(images, (len(images), -1))
>>> agglo = cluster.FeatureAgglomeration(n_clusters=32)
>>> agglo.fit(X)
FeatureAgglomeration(n_clusters=32)
>>> X_reduced = agglo.transform(X)
>>> X_reduced.shape
(1797, 32)
```

Methods

fit(self, X[, y])	Fit the hierarchical clustering on the data
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, Xred)	Inverse the transformation.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform a new matrix using the built clustering

__init__(*self*, *n_clusters=2*, *affinity='euclidean'*, *memory=None*, *connectivity=None*, *compute_full_tree='auto'*, *linkage='ward'*, *pooling_func=<function mean at 0x7fdf82331c10>*, *distance_threshold=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*, ***params*)
Fit the hierarchical clustering on the data

Parameters

X [array-like of shape (n_samples, n_features)] The data

y [Ignored]

Returns

self

property fit_predict
Fit the hierarchical clustering from features or distance matrix, and return cluster labels.

Parameters

X [array-like, shape (n_samples, n_features) or (n_samples, n_samples)] Training instances to cluster, or distances between instances if `affinity='precomputed'`.

y [Ignored] Not used, present here for API consistency by convention.

Returns

labels [ndarray, shape (n_samples,)] Cluster labels.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *Xred*)
Inverse the transformation. Return a vector of size nb_features with the values of Xred assigned to each group of features

Parameters

Xred [array-like of shape (n_samples, n_clusters) or (n_clusters,)] The values to be assigned to each cluster of samples

Returns

X [array, shape=[n_samples, n_features] or [n_features]] A vector of size n_samples with the values of Xred assigned to each of the cluster of samples.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Transform a new matrix using the built clustering

Parameters

X [array-like of shape (n_samples, n_features) or (n_samples,)] A M by N array of M observations in N dimensions or a length M array of M one-dimensional observations.

Returns

Y [array, shape = [n_samples, n_clusters] or [n_clusters]] The pooled values for each feature cluster.

Examples using `sklearn.cluster.FeatureAgglomeration`

- *Feature agglomeration*
- *Feature agglomeration vs. univariate selection*

sklearn.cluster.KMeans

class sklearn.cluster.**KMeans**(*n_clusters=8*, *init='k-means++'*, *n_init=10*, *max_iter=300*, *tol=0.0001*, *precompute_distances='auto'*, *verbose=0*, *random_state=None*, *copy_x=True*, *n_jobs=None*, *algorithm='auto'*)
K-Means clustering.

Read more in the *User Guide*.

Parameters

n_clusters [int, default=8] The number of clusters to form as well as the number of centroids to generate.

init [{'k-means++', 'random'} or ndarray of shape (n_clusters, n_features), default='k-means++'] Method for initialization, defaults to 'k-means++':

'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence. See section Notes in k_init for more details.

'random': choose k observations (rows) at random from data for the initial centroids.

If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.

n_init [int, default=10] Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

max_iter [int, default=300] Maximum number of iterations of the k-means algorithm for a single run.

tol [float, default=1e-4] Relative tolerance with regards to inertia to declare convergence.

precompute_distances ['auto' or bool, default='auto'] Precompute distances (faster but takes more memory).

'auto' : do not precompute distances if n_samples * n_clusters > 12 million. This corresponds to about 100MB overhead per job using double precision.

True : always precompute distances.

False : never precompute distances.

verbose [int, default=0] Verbosity mode.

random_state [int, RandomState instance, default=None] Determines random number generation for centroid initialization. Use an int to make the randomness deterministic. See *Glossary*.

copy_x [bool, default=True] When pre-computing distances it is more numerically accurate to center the data first. If copy_x is True (default), then the original data is not modified, ensuring X is C-contiguous. If False, the original data is modified, and put back before the function returns, but small numerical differences may be introduced by subtracting and then adding the data mean, in this case it will also not ensure that data is C-contiguous which may cause a significant slowdown.

n_jobs [int, default=None] The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

algorithm [{"auto", "full", "elkan"}, default="auto"] K-means algorithm to use. The classical EM-style algorithm is "full". The "elkan" variation is more efficient by using the triangle inequality, but currently doesn't support sparse data. "auto" chooses "elkan" for dense data and "full" for sparse data.

Attributes

cluster_centers_ [ndarray of shape (n_clusters, n_features)] Coordinates of cluster centers. If the algorithm stops before fully converging (see `tol` and `max_iter`), these will not be consistent with `labels_`.

labels_ [ndarray of shape (n_samples,)] Labels of each point

inertia_ [float] Sum of squared distances of samples to their closest cluster center.

n_iter_ [int] Number of iterations run.

See also:

MiniBatchKMeans Alternative online implementation that does incremental updates of the centers positions using mini-batches. For large scale learning (say n_samples > 10k) MiniBatchKMeans is probably much faster than the default batch implementation.

Notes

The k-means problem is solved using either Lloyd's or Elkan's algorithm.

The average complexity is given by O(k n T), were n is the number of samples and T is the number of iteration.

The worst case complexity is given by O(n^(k+2/p)) with n = n_samples, p = n_features. (D. Arthur and S. Vassilvitskii, 'How slow is the k-means method?' SoCG2006)

In practice, the k-means algorithm is very fast (one of the fastest clustering algorithms available), but it falls in local minima. That's why it can be useful to restart it several times.

If the algorithm stops before fully converging (because of `tol` or `max_iter`), `labels_` and `cluster_centers_` will not be consistent, i.e. the `cluster_centers_` will not be the means of the points in each cluster. Also, the estimator will reassign `labels_` after the last iteration to make `labels_` consistent with `predict` on the training set.

Examples

```
>>> from sklearn.cluster import KMeans
>>> import numpy as np
>>> X = np.array([[1, 2], [1, 4], [1, 0],
...               [10, 2], [10, 4], [10, 0]])
>>> kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
>>> kmeans.labels_
array([1, 1, 1, 0, 0, 0], dtype=int32)
>>> kmeans.predict([[0, 0], [12, 3]])
array([1, 0], dtype=int32)
>>> kmeans.cluster_centers_
array([[10.,  2.],
       [ 1.,  2.]])
```

Methods

fit(self, X[, y, sample_weight])	Compute k-means clustering.
fit_predict(self, X[, y, sample_weight])	Compute cluster centers and predict cluster index for each sample.

Continued on next page

Table 19 – continued from previous page

fit_transform(self, X[, y, sample_weight])	Compute clustering and transform X to cluster-distance space.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, sample_weight])	Predict the closest cluster each sample in X belongs to.
score(self, X[, y, sample_weight])	Opposite of the value of X on the K-means objective.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform X to a cluster-distance space.

__init__(*self*, *n_clusters=8*, *init='k-means++'*, *n_init=10*, *max_iter=300*, *tol=0.0001*, *precompute_distances='auto'*, *verbose=0*, *random_state=None*, *copy_x=True*, *n_jobs=None*, *algorithm='auto'*)

Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*, *sample_weight=None*)

Compute k-means clustering.

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] Training instances to cluster. It must be noted that the data will be converted to C ordering, which will cause a memory copy if the given data is not C-contiguous.

y [Ignored] Not used, present here for API consistency by convention.

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None).

Returns

self Fitted estimator.

fit_predict(*self*, *X*, *y=None*, *sample_weight=None*)

Compute cluster centers and predict cluster index for each sample.

Convenience method; equivalent to calling fit(X) followed by predict(X).

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] New data to transform.

y [Ignored] Not used, present here for API consistency by convention.

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None).

Returns

labels [array, shape [n_samples,]] Index of the cluster each sample belongs to.

fit_transform(*self*, *X*, *y=None*, *sample_weight=None*)

Compute clustering and transform X to cluster-distance space.

Equivalent to fit(X).transform(X), but more efficiently implemented.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] New data to transform.

y [Ignored] Not used, present here for API consistency by convention.

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None).

Returns

X_new [array, shape [n_samples, k]] X transformed in the new space.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*, *sample_weight=None*)
Predict the closest cluster each sample in X belongs to.

In the vector quantization literature, `cluster_centers_` is called the code book and each value returned by `predict` is the index of the closest code in the code book.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] New data to predict.

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None).

Returns

labels [array, shape [n_samples,]] Index of the cluster each sample belongs to.

score(*self*, *X*, *y=None*, *sample_weight=None*)
Opposite of the value of X on the K-means objective.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] New data.

y [Ignored] Not used, present here for API consistency by convention.

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None).

Returns

score [float] Opposite of the value of X on the K-means objective.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Transform X to a cluster-distance space.

In the new space, each dimension is the distance to the cluster centers. Note that even if X is sparse, the array returned by `transform` will typically be dense.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] New data to transform.

Returns

X_new [array, shape [n_samples, k]] X transformed in the new space.

Examples using `sklearn.cluster.KMeans`

- *Demonstration of k-means assumptions*
- *Vector Quantization Example*
- *K-means Clustering*
- *Color Quantization using K-Means*
- *Empirical evaluation of the impact of k-means initialization*
- *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*
- *A demo of K-Means clustering on the handwritten digits data*
- *Selecting the number of clusters with silhouette analysis on KMeans clustering*
- *Clustering text documents using k-means*

`sklearn.cluster.MiniBatchKMeans`

class sklearn.cluster.**MiniBatchKMeans**(*n_clusters=8*, *init='k-means++'*, *max_iter=100*, *batch_size=100*, *verbose=0*, *compute_labels=True*, *random_state=None*, *tol=0.0*, *max_no_improvement=10*, *init_size=None*, *n_init=3*, *reassignment_ratio=0.01*)

Mini-Batch K-Means clustering.

Read more in the *User Guide*.

Parameters

n_clusters [int, default=8] The number of clusters to form as well as the number of centroids to generate.

init [{'k-means++', 'random'} or ndarray of shape (n_clusters, n_features), default='k-means++'] Method for initialization

'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence. See section Notes in k_init for more details.

'random': choose k observations (rows) at random from data for the initial centroids.

If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.

max_iter [int, default=100] Maximum number of iterations over the complete dataset before stopping independently of any early stopping criterion heuristics.

batch_size [int, default=100] Size of the mini batches.

verbose [int, default=0] Verbosity mode.

compute_labels [bool, default=True] Compute label assignment and inertia for the complete dataset once the minibatch optimization has converged in fit.

random_state [int, RandomState instance, default=None] Determines random number generation for centroid initialization and random reassignment. Use an int to make the randomness deterministic. See *Glossary*.

tol [float, default=0.0] Control early stopping based on the relative center changes as measured by a smoothed, variance-normalized of the mean center squared position changes. This early stopping heuristics is closer to the one used for the batch variant of the algorithms but induces a slight computational and memory overhead over the inertia heuristic.

To disable convergence detection based on normalized center change, set tol to 0.0 (default).

max_no_improvement [int, default=10] Control early stopping based on the consecutive number of mini batches that does not yield an improvement on the smoothed inertia.

To disable convergence detection based on inertia, set max_no_improvement to None.

init_size [int, default=None] Number of samples to randomly sample for speeding up the initialization (sometimes at the expense of accuracy): the only algorithm is initialized by running a batch KMeans on a random subset of the data. This needs to be larger than n_clusters.

If `None`, `init_size= 3 * batch_size`.

n_init [int, default=3] Number of random initializations that are tried. In contrast to KMeans, the algorithm is only run once, using the best of the `n_init` initializations as measured by inertia.

reassignment_ratio [float, default=0.01] Control the fraction of the maximum number of counts for a center to be reassigned. A higher value means that low count centers are more easily reassigned, which means that the model will take longer to converge, but should converge in a better clustering.

Attributes

cluster_centers_ [ndarray of shape (n_clusters, n_features)] Coordinates of cluster centers

labels_ [int] Labels of each point (if compute_labels is set to True).

inertia_ [float] The value of the inertia criterion associated with the chosen partition (if compute_labels is set to True). The inertia is defined as the sum of square distances of samples to their nearest neighbor.

See also:

KMeans The classic implementation of the clustering method based on the Lloyd's algorithm. It consumes the whole set of input data at each iteration.

Notes

See https://www.eecs.tufts.edu/~dsculley/papers/fastkmeans.pdf

Examples

```
>>> from sklearn.cluster import MiniBatchKMeans
>>> import numpy as np
>>> X = np.array([[1, 2], [1, 4], [1, 0],
...               [4, 2], [4, 0], [4, 4],
...               [4, 5], [0, 1], [2, 2],
...               [3, 2], [5, 5], [1, -1]])
>>> # manually fit on batches
>>> kmeans = MiniBatchKMeans(n_clusters=2,
...                          random_state=0,
...                          batch_size=6)
>>> kmeans = kmeans.partial_fit(X[0:6,:])
>>> kmeans = kmeans.partial_fit(X[6:12,:])
>>> kmeans.cluster_centers_
array([[2. , 1. ],
       [3.5, 4.5]])
>>> kmeans.predict([[0, 0], [4, 4]])
array([0, 1], dtype=int32)
>>> # fit on the whole data
>>> kmeans = MiniBatchKMeans(n_clusters=2,
...                          random_state=0,
...                          batch_size=6,
...                          max_iter=10).fit(X)
>>> kmeans.cluster_centers_
array([[3.95918367, 2.40816327],
       [1.12195122, 1.3902439 ]])
>>> kmeans.predict([[0, 0], [4, 4]])
array([1, 0], dtype=int32)
```

Methods

fit(self, X[, y, sample_weight])	Compute the centroids on X by chunking it into mini-batches.
fit_predict(self, X[, y, sample_weight])	Compute cluster centers and predict cluster index for each sample.
fit_transform(self, X[, y, sample_weight])	Compute clustering and transform X to cluster-distance space.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X[, y, sample_weight])	Update k means estimate on a single mini-batch X.
predict(self, X[, sample_weight])	Predict the closest cluster each sample in X belongs to.
score(self, X[, y, sample_weight])	Opposite of the value of X on the K-means objective.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform X to a cluster-distance space.

__init__(*self*, *n_clusters=8*, *init='k-means++'*, *max_iter=100*, *batch_size=100*, *verbose=0*, *compute_labels=True*, *random_state=None*, *tol=0.0*, *max_no_improvement=10*, *init_size=None*, *n_init=3*, *reassignment_ratio=0.01*)

Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*, *sample_weight=None*)

Compute the centroids on X by chunking it into mini-batches.

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] Training instances to cluster.

It must be noted that the data will be converted to C ordering, which will cause a memory copy if the given data is not C-contiguous.

y [Ignored] Not used, present here for API consistency by convention.

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None).

Returns

self

fit_predict(*self*, *X*, *y=None*, *sample_weight=None*)
Compute cluster centers and predict cluster index for each sample.

Convenience method; equivalent to calling fit(X) followed by predict(X).

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] New data to transform.

y [Ignored] Not used, present here for API consistency by convention.

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None).

Returns

labels [array, shape [n_samples,]] Index of the cluster each sample belongs to.

fit_transform(*self*, *X*, *y=None*, *sample_weight=None*)
Compute clustering and transform X to cluster-distance space.

Equivalent to fit(X).transform(X), but more efficiently implemented.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] New data to transform.

y [Ignored] Not used, present here for API consistency by convention.

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None).

Returns

X_new [array, shape [n_samples, k]] X transformed in the new space.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit(*self*, *X*, *y=None*, *sample_weight=None*)
Update k means estimate on a single mini-batch X.

Parameters

X [array-like of shape (n_samples, n_features)] Coordinates of the data points to cluster. It must be noted that X will be copied if it is not C-contiguous.

y [Ignored] Not used, present here for API consistency by convention.

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None).

Returns

self

predict(*self*, *X*, *sample_weight=None*)

Predict the closest cluster each sample in X belongs to.

In the vector quantization literature, `cluster_centers_` is called the code book and each value returned by `predict` is the index of the closest code in the code book.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] New data to predict.

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None).

Returns

labels [array, shape [n_samples,]] Index of the cluster each sample belongs to.

score(*self*, *X*, *y=None*, *sample_weight=None*)

Opposite of the value of X on the K-means objective.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] New data.

y [Ignored] Not used, present here for API consistency by convention.

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None).

Returns

score [float] Opposite of the value of X on the K-means objective.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)

Transform X to a cluster-distance space.

In the new space, each dimension is the distance to the cluster centers. Note that even if X is sparse, the array returned by `transform` will typically be dense.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] New data to transform.

Returns

X_new [array, shape [n_samples, k]] X transformed in the new space.

Examples using `sklearn.cluster.MiniBatchKMeans`

- *Biclustering documents with the Spectral Co-clustering algorithm*
- *Online learning of a dictionary of parts of faces*
- *Compare BIRCH and MiniBatchKMeans*
- *Empirical evaluation of the impact of k-means initialization*
- *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*
- *Comparing different clustering algorithms on toy datasets*
- *Faces dataset decompositions*
- *Clustering text documents using k-means*

`sklearn.cluster`.MeanShift

class sklearn.cluster.**MeanShift**(*bandwidth=None*, *seeds=None*, *bin_seeding=False*, *min_bin_freq=1*, *cluster_all=True*, *n_jobs=None*, *max_iter=300*)

Mean shift clustering using a flat kernel.

Mean shift clustering aims to discover "blobs" in a smooth density of samples. It is a centroid-based algorithm, which works by updating candidates for centroids to be the mean of the points within a given region. These candidates are then filtered in a post-processing stage to eliminate near-duplicates to form the final set of centroids.

Seeding is performed using a binning technique for scalability.

Read more in the *User Guide*.

Parameters

bandwidth [float, optional] Bandwidth used in the RBF kernel.

If not given, the bandwidth is estimated using sklearn.cluster.estimate_bandwidth; see the documentation for that function for hints on scalability (see also the Notes, below).

seeds [array, shape=[n_samples, n_features], optional] Seeds used to initialize kernels. If not set, the seeds are calculated by clustering.get_bin_seeds with bandwidth as the grid size and default values for other parameters.

bin_seeding [boolean, optional] If true, initial kernel locations are not locations of all points, but rather the location of the discretized version of points, where points are binned onto a grid whose coarseness corresponds to the bandwidth. Setting this option to True will speed up the algorithm because fewer seeds will be initialized. default value: False Ignored if seeds argument is not None.

min_bin_freq [int, optional] To speed up the algorithm, accept only those bins with at least min_bin_freq points as seeds. If not defined, set to 1.

cluster_all [boolean, default True] If true, then all points are clustered, even those orphans that are not within any kernel. Orphans are assigned to the nearest kernel. If false, then orphans are given cluster label -1.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

max_iter [int, default=300] Maximum number of iterations, per seed point before the clustering operation terminates (for that seed point), if has not converged yet.

New in version 0.22.

Attributes

cluster_centers_ [array, [n_clusters, n_features]] Coordinates of cluster centers.

labels_ : Labels of each point.

n_iter_ [int] Maximum number of iterations performed on each seed.

New in version 0.22.

Notes

Scalability:

Because this implementation uses a flat kernel and a Ball Tree to look up members of each kernel, the complexity will tend towards O(T*n*log(n)) in lower dimensions, with n the number of samples and T the number of points. In higher dimensions the complexity will tend towards O(T*n^2).

Scalability can be boosted by using fewer seeds, for example by using a higher value of min_bin_freq in the get_bin_seeds function.

Note that the estimate_bandwidth function is much less scalable than the mean shift algorithm and will be the bottleneck if it is used.

References

Dorin Comaniciu and Peter Meer, "Mean Shift: A robust approach toward feature space analysis". IEEE Transactions on Pattern Analysis and Machine Intelligence. 2002. pp. 603-619.

Examples

```
>>> from sklearn.cluster import MeanShift
>>> import numpy as np
>>> X = np.array([[1, 1], [2, 1], [1, 0],
...               [4, 7], [3, 5], [3, 6]])
>>> clustering = MeanShift(bandwidth=2).fit(X)
>>> clustering.labels_
array([1, 1, 1, 0, 0, 0])
>>> clustering.predict([[0, 0], [5, 5]])
array([1, 0])
>>> clustering
MeanShift(bandwidth=2)
```

Methods

fit(self, X[, y])	Perform clustering.
fit_predict(self, X[, y])	Perform clustering on X and returns cluster labels.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the closest cluster each sample in X belongs to.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *bandwidth=None*, *seeds=None*, *bin_seeding=False*, *min_bin_freq=1*, *cluster_all=True*, *n_jobs=None*, *max_iter=300*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*)
Perform clustering.

Parameters

X [array-like of shape (n_samples, n_features)] Samples to cluster.

y [Ignored]

fit_predict (*self*, *X*, *y=None*)
Perform clustering on X and returns cluster labels.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

y [Ignored] Not used, present for API consistency by convention.

Returns

labels [ndarray, shape (n_samples,)] Cluster labels.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict the closest cluster each sample in X belongs to.

Parameters

X [{array-like, sparse matrix}, shape=[n_samples, n_features]] New data to predict.

Returns

labels [array, shape [n_samples,]] Index of the cluster each sample belongs to.

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.cluster.MeanShift`

- *A demo of the mean-shift clustering algorithm*
- *Comparing different clustering algorithms on toy datasets*

`sklearn.cluster.OPTICS`

class sklearn.cluster.**OPTICS**(*min_samples=5*, *max_eps=inf*, *metric='minkowski'*, *p=2*, *metric_params=None*, *cluster_method='xi'*, *eps=None*, *xi=0.05*, *predecessor_correction=True*, *min_cluster_size=None*, *algorithm='auto'*, *leaf_size=30*, *n_jobs=None*)

Estimate clustering structure from vector array.

OPTICS (Ordering Points To Identify the Clustering Structure), closely related to DBSCAN, finds core sample of high density and expands clusters from them [R2c55e37003fe-1]. Unlike DBSCAN, keeps cluster hierarchy for a variable neighborhood radius. Better suited for usage on large datasets than the current sklearn implementation of DBSCAN.

Clusters are then extracted using a DBSCAN-like method (cluster_method = 'dbscan') or an automatic technique proposed in [R2c55e37003fe-1] (cluster_method = 'xi').

This implementation deviates from the original OPTICS by first performing k-nearest-neighborhood searches on all points to identify core sizes, then computing only the distances to unprocessed points when constructing the cluster order. Note that we do not employ a heap to manage the expansion candidates, so the time complexity will be O(n^2).

Read more in the *User Guide*.

Parameters

min_samples [int > 1 or float between 0 and 1 (default=5)] The number of samples in a neighborhood for a point to be considered as a core point. Also, up and down steep regions can't have more then `min_samples` consecutive non-steep points. Expressed as an absolute number or a fraction of the number of samples (rounded to be at least 2).

max_eps [float, optional (default=np.inf)] The maximum distance between two samples for one to be considered as in the neighborhood of the other. Default value of `np.inf` will identify clusters across all scales; reducing `max_eps` will result in shorter run times.

metric [str or callable, optional (default='minkowski')] Metric to use for distance computation. Any metric from scikit-learn or scipy.spatial.distance can be used.

If metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays as input and return one value indicating the distance between them. This works for Scipy's metrics, but is less efficient than passing the metric name as a string. If metric is "precomputed", X is assumed to be a distance matrix and must be square.

Valid values for metric are:

- from scikit-learn: ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']

- from scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule']

See the documentation for scipy.spatial.distance for details on these metrics.

p [int, optional (default=2)] Parameter for the Minkowski metric from *sklearn.metrics.pairwise_distances*. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params [dict, optional (default=None)] Additional keyword arguments for the metric function.

cluster_method [str, optional (default='xi')] The extraction method used to extract clusters using the calculated reachability and ordering. Possible values are "xi" and "dbscan".

eps [float, optional (default=None)] The maximum distance between two samples for one to be considered as in the neighborhood of the other. By default it assumes the same value as `max_eps`. Used only when `cluster_method='dbscan'`.

xi [float, between 0 and 1, optional (default=0.05)] Determines the minimum steepness on the reachability plot that constitutes a cluster boundary. For example, an upwards point in the reachability plot is defined by the ratio from one point to its successor being at most 1-xi. Used only when `cluster_method='xi'`.

predecessor_correction [bool, optional (default=True)] Correct clusters according to the predecessors calculated by OPTICS [R2c55e37003fe-2]. This parameter has minimal effect on most datasets. Used only when `cluster_method='xi'`.

min_cluster_size [int > 1 or float between 0 and 1 (default=None)] Minimum number of samples in an OPTICS cluster, expressed as an absolute number or a fraction of the number of samples (rounded to be at least 2). If `None`, the value of `min_samples` is used instead. Used only when `cluster_method='xi'`.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to *fit* method. (default)

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, optional (default=30)] Leaf size passed to `BallTree` or `KDTree`. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

labels_ [array, shape (n_samples,)] Cluster labels for each point in the dataset given to fit(). Noisy samples and points which are not included in a leaf cluster of `cluster_hierarchy_` are labeled as -1.

reachability_ [array, shape (n_samples,)] Reachability distances per sample, indexed by object order. Use `clust.reachability_[clust.ordering_]` to access in cluster order.

ordering_ [array, shape (n_samples,)] The cluster ordered list of sample indices.

core_distances_ [array, shape (n_samples,)] Distance at which each sample becomes a core point, indexed by object order. Points which will never be core have a distance of inf. Use `clust.core_distances_[clust.ordering_]` to access in cluster order.

predecessor_ [array, shape (n_samples,)] Point that a sample was reached from, indexed by object order. Seed points have a predecessor of -1.

cluster_hierarchy_ [array, shape (n_clusters, 2)] The list of clusters in the form of `[start, end]` in each row, with all indices inclusive. The clusters are ordered according to `(end, -start)` (ascending) so that larger clusters encompassing smaller clusters come after those smaller ones. Since `labels_` does not reflect the hierarchy, usually `len(cluster_hierarchy_) > np.unique(optics.labels_)`. Please also note that these indices are of the `ordering_`, i.e. `X[ordering_][start:end + 1]` form a cluster. Only available when `cluster_method='xi'`.

See also:

DBSCAN A similar clustering for a specified neighborhood radius (eps). Our implementation is optimized for runtime.

References

[R2c55e37003fe-1], [R2c55e37003fe-2]

Examples

```
>>> from sklearn.cluster import OPTICS
>>> import numpy as np
>>> X = np.array([[1, 2], [2, 5], [3, 6],
...               [8, 7], [8, 8], [7, 3]])
>>> clustering = OPTICS(min_samples=2).fit(X)
>>> clustering.labels_
array([0, 0, 0, 1, 1, 1])
```

Methods

fit(self, X[, y])	Perform OPTICS clustering.
fit_predict(self, X[, y])	Perform clustering on X and returns cluster labels.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *min_samples=5*, *max_eps=inf*, *metric='minkowski'*, *p=2*, *metric_params=None*, *cluster_method='xi'*, *eps=None*, *xi=0.05*, *predecessor_correction=True*, *min_cluster_size=None*, *algorithm='auto'*, *leaf_size=30*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Perform OPTICS clustering.

Extracts an ordered list of points and reachability distances, and performs initial clustering using `max_eps` distance specified at OPTICS object instantiation.

Parameters

X [array, shape (n_samples, n_features), or (n_samples, n_samples) if metric='precomputed'] A feature array, or array of distances between samples if metric='precomputed'.

y [ignored] Ignored.

Returns

self [instance of OPTICS] The instance.

fit_predict (*self*, *X*, *y=None*)

Perform clustering on X and returns cluster labels.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

y [Ignored] Not used, present for API consistency by convention.

Returns

labels [ndarray, shape (n_samples,)] Cluster labels.

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.cluster.OPTICS`

- *Demo of OPTICS clustering algorithm*
- *Comparing different clustering algorithms on toy datasets*

sklearn.cluster.SpectralClustering

class sklearn.cluster.**SpectralClustering**(*n_clusters=8*, *eigen_solver=None*, *n_components=None*, *random_state=None*, *n_init=10*, *gamma=1.0*, *affinity='rbf'*, *n_neighbors=10*, *eigen_tol=0.0*, *assign_labels='kmeans'*, *degree=3*, *coef0=1*, *kernel_params=None*, *n_jobs=None*)

Apply clustering to a projection of the normalized Laplacian.

In practice Spectral Clustering is very useful when the structure of the individual clusters is highly non-convex or more generally when a measure of the center and spread of the cluster is not a suitable description of the complete cluster. For instance when clusters are nested circles on the 2D plane.

If affinity is the adjacency matrix of a graph, this method can be used to find normalized graph cuts.

When calling `fit`, an affinity matrix is constructed using either kernel function such the Gaussian (aka RBF) kernel of the euclidean distanced `d(X, X)`:

```
np.exp(-gamma * d(X,X) ** 2)
```

or a k-nearest neighbors connectivity matrix.

Alternatively, using `precomputed`, a user-provided affinity matrix can be used.

Read more in the *User Guide*.

Parameters

n_clusters [integer, optional] The dimension of the projection subspace.

eigen_solver [{None, 'arpack', 'lobpcg', or 'amg'}] The eigenvalue decomposition strategy to use. AMG requires pyamg to be installed. It can be faster on very large, sparse problems, but may also lead to instabilities.

n_components [integer, optional, default=n_clusters] Number of eigen vectors to use for the spectral embedding

random_state [int, RandomState instance or None (default)] A pseudo random number generator used for the initialization of the lobpcg eigen vectors decomposition when `eigen_solver='amg'` and by the K-Means initialization. Use an int to make the randomness deterministic. See *Glossary*.

n_init [int, optional, default: 10] Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

gamma [float, default=1.0] Kernel coefficient for rbf, poly, sigmoid, laplacian and chi2 kernels. Ignored for `affinity='nearest_neighbors'`.

affinity [string or callable, default 'rbf']

How to construct the affinity matrix.

- 'nearest_neighbors' : construct the affinity matrix by computing a graph of nearest neighbors.
- 'rbf' : construct the affinity matrix using a radial basis function (RBF) kernel.
- 'precomputed' : interpret `X` as a precomputed affinity matrix.

- 'precomputed_nearest_neighbors' : interpret `X` as a sparse graph of precomputed nearest neighbors, and constructs the affinity matrix by selecting the `n_neighbors` nearest neighbors.
- one of the kernels supported by `pairwise_kernels`.

Only kernels that produce similarity scores (non-negative values that increase with similarity) should be used. This property is not checked by the clustering algorithm.

n_neighbors [integer] Number of neighbors to use when constructing the affinity matrix using the nearest neighbors method. Ignored for `affinity='rbf'`.

eigen_tol [float, optional, default: 0.0] Stopping criterion for eigendecomposition of the Laplacian matrix when `eigen_solver='arpack'`.

assign_labels [{'kmeans', 'discretize'}, default: 'kmeans'] The strategy to use to assign labels in the embedding space. There are two ways to assign labels after the laplacian embedding. k-means can be applied and is a popular choice. But it can also be sensitive to initialization. Discretization is another approach which is less sensitive to random initialization.

degree [float, default=3] Degree of the polynomial kernel. Ignored by other kernels.

coef0 [float, default=1] Zero coefficient for polynomial and sigmoid kernels. Ignored by other kernels.

kernel_params [dictionary of string to any, optional] Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

affinity_matrix_ [array-like, shape (n_samples, n_samples)] Affinity matrix used for clustering. Available only if after calling `fit`.

labels_ [array, shape (n_samples,)] Labels of each point

Notes

If you have an affinity matrix, such as a distance matrix, for which 0 means identical elements, and high values means very dissimilar elements, it can be transformed in a similarity matrix that is well suited for the algorithm by applying the Gaussian (RBF, heat) kernel:

```
np.exp(- dist_matrix ** 2 / (2. * delta ** 2))
```

Where `delta` is a free parameter representing the width of the Gaussian kernel.

Another alternative is to take a symmetric version of the k nearest neighbors connectivity matrix of the points.

If the pyamg package is installed, it is used: this greatly speeds up computation.

References

- Normalized cuts and image segmentation, 2000 Jianbo Shi, Jitendra Malik http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.160.2324
- A Tutorial on Spectral Clustering, 2007 Ulrike von Luxburg http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.165.9323

- Multiclass spectral clustering, 2003 Stella X. Yu, Jianbo Shi https://www1.icsi.berkeley.edu/~stellayu/publication/doc/2003kwayICCV.pdf

Examples

```
>>> from sklearn.cluster import SpectralClustering
>>> import numpy as np
>>> X = np.array([[1, 1], [2, 1], [1, 0],
...               [4, 7], [3, 5], [3, 6]])
>>> clustering = SpectralClustering(n_clusters=2,
...         assign_labels="discretize",
...         random_state=0).fit(X)
>>> clustering.labels_
array([1, 1, 1, 0, 0, 0])
>>> clustering
SpectralClustering(assign_labels='discretize', n_clusters=2,
    random_state=0)
```

Methods

fit(self, X[, y])	Perform spectral clustering from features, or affinity matrix.
fit_predict(self, X[, y])	Perform spectral clustering from features, or affinity matrix, and return cluster labels.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_clusters=8*, *eigen_solver=None*, *n_components=None*, *random_state=None*, *n_init=10*, *gamma=1.0*, *affinity='rbf'*, *n_neighbors=10*, *eigen_tol=0.0*, *assign_labels='kmeans'*, *degree=3*, *coef0=1*, *kernel_params=None*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Perform spectral clustering from features, or affinity matrix.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features), or array-like, shape (n_samples, n_samples)] Training instances to cluster, or similarities / affinities between instances if `affinity='precomputed'`. If a sparse matrix is provided in a format other than `csr_matrix`, `csc_matrix`, or `coo_matrix`, it will be converted into a sparse `csr_matrix`.

y [Ignored] Not used, present here for API consistency by convention.

Returns

self

fit_predict(*self*, *X*, *y=None*)
Perform spectral clustering from features, or affinity matrix, and return cluster labels.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features), or array-like, shape (n_samples, n_samples)] Training instances to cluster, or similarities / affinities between

instances if `affinity='precomputed'`. If a sparse matrix is provided in a format other than `csr_matrix`, `csc_matrix`, or `coo_matrix`, it will be converted into a sparse `csr_matrix`.

y [Ignored] Not used, present here for API consistency by convention.

Returns

labels [ndarray, shape (n_samples,)] Cluster labels.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.cluster.SpectralClustering`

- *Comparing different clustering algorithms on toy datasets*

`sklearn.cluster`.SpectralBiclustering

class sklearn.cluster.**SpectralBiclustering**(*n_clusters=3*, *method='bistochastic'*, *n_components=6*, *n_best=3*, *svd_method='randomized'*, *n_svd_vecs=None*, *mini_batch=False*, *init='k-means++'*, *n_init=10*, *n_jobs=None*, *random_state=None*)
Spectral biclustering (Kluger, 2003).

Partitions rows and columns under the assumption that the data has an underlying checkerboard structure. For instance, if there are two row partitions and three column partitions, each row will belong to three biclusters, and each column will belong to two biclusters. The outer product of the corresponding row and column label vectors gives this checkerboard structure.

Read more in the *User Guide*.

Parameters

n_clusters [int or tuple (n_row_clusters, n_column_clusters), default=3] The number of row and column clusters in the checkerboard structure.

method [{'bistochastic', 'scale', 'log'}, default='bistochastic'] Method of normalizing and converting singular vectors into biclusters. May be one of 'scale', 'bistochastic', or 'log'. The authors recommend using 'log'. If the data is sparse, however, log normalization will not work, which is why the default is 'bistochastic'.

> **Warning:** if `method='log'`, the data must be sparse.

n_components [int, default=6] Number of singular vectors to check.

n_best [int, default=3] Number of best singular vectors to which to project the data for clustering.

svd_method [{'randomized', 'arpack'}, default='randomized'] Selects the algorithm for finding singular vectors. May be 'randomized' or 'arpack'. If 'randomized', uses *randomized_svd*, which may be faster for large matrices. If 'arpack', uses `scipy.sparse.linalg.svds`, which is more accurate, but possibly slower in some cases.

n_svd_vecs [int, default=None] Number of vectors to use in calculating the SVD. Corresponds to `ncv` when `svd_method=arpack` and `n_oversamples` when `svd_method` is 'randomized'.

mini_batch [bool, default=False] Whether to use mini-batch k-means, which is faster but may get different results.

init [{'k-means++', 'random'} or ndarray of (n_clusters, n_features), default='k-means++'] Method for initialization of k-means algorithm; defaults to 'k-means++'.

n_init [int, default=10] Number of random initializations that are tried with the k-means algorithm.

If mini-batch k-means is used, the best initialization is chosen and the algorithm runs once. Otherwise, the algorithm is run for each initialization and the best solution chosen.

n_jobs [int, default=None] The number of jobs to use for the computation. This works by breaking down the pairwise matrix into n_jobs even slices and computing them in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance, default=None] Used for randomizing the singular value decomposition and the k-means initialization. Use an int to make the randomness deterministic. See *Glossary*.

Attributes

rows_ [array-like of shape (n_row_clusters, n_rows)] Results of the clustering. `rows[i, r]` is True if cluster `i` contains row `r`. Available only after calling `fit`.

columns_ [array-like of shape (n_column_clusters, n_columns)] Results of the clustering, like `rows`.

row_labels_ [array-like of shape (n_rows,)] Row partition labels.

column_labels_ [array-like of shape (n_cols,)] Column partition labels.

References

- Kluger, Yuval, et. al., 2003. Spectral biclustering of microarray data: coclustering genes and conditions.

Examples

```
>>> from sklearn.cluster import SpectralBiclustering
>>> import numpy as np
>>> X = np.array([[1, 1], [2, 1], [1, 0],
...               [4, 7], [3, 5], [3, 6]])
>>> clustering = SpectralBiclustering(n_clusters=2, random_state=0).fit(X)
>>> clustering.row_labels_
array([1, 1, 1, 0, 0, 0], dtype=int32)
>>> clustering.column_labels_
array([0, 1], dtype=int32)
>>> clustering
SpectralBiclustering(n_clusters=2, random_state=0)
```

Methods

fit(self, X[, y])	Creates a biclustering for X.
get_indices(self, i)	Row and column indices of the i'th bicluster.
get_params(self[, deep])	Get parameters for this estimator.
get_shape(self, i)	Shape of the i'th bicluster.
get_submatrix(self, i, data)	Return the submatrix corresponding to bicluster `i`.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *n_clusters=3*, *method='bistochastic'*, *n_components=6*, *n_best=3*, *svd_method='randomized'*, *n_svd_vecs=None*, *mini_batch=False*, *init='k-means++'*, *n_init=10*, *n_jobs=None*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

property biclusters_
Convenient way to get row and column indicators together.

Returns the `rows_` and `columns_` members.

fit (*self*, *X*, *y=None*)
Creates a biclustering for X.

Parameters

X [array-like, shape (n_samples, n_features)]

y [Ignored]

get_indices (*self*, *i*)
Row and column indices of the i'th bicluster.

Only works if `rows_` and `columns_` attributes exist.

Parameters

i [int] The index of the cluster.

Returns

row_ind [np.array, dtype=np.intp] Indices of rows in the dataset that belong to the bicluster.

col_ind [np.array, dtype=np.intp] Indices of columns in the dataset that belong to the bicluster.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_shape(*self*, *i*)
Shape of the i'th bicluster.

Parameters

i [int] The index of the cluster.

Returns

shape [(int, int)] Number of rows and columns (resp.) in the bicluster.

get_submatrix(*self*, *i*, *data*)
Return the submatrix corresponding to bicluster `i`.

Parameters

i [int] The index of the cluster.

data [array] The data.

Returns

submatrix [array] The submatrix corresponding to bicluster i.

Notes

Works with sparse matrices. Only works if `rows_` and `columns_` attributes exist.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.cluster.SpectralBiclustering`

- *A demo of the Spectral Biclustering algorithm*

sklearn.cluster.SpectralCoclustering

class sklearn.cluster.**SpectralCoclustering**(*n_clusters=3*, *svd_method='randomized'*, *n_svd_vecs=None*, *mini_batch=False*, *init='k-means++'*, *n_init=10*, *n_jobs=None*, *random_state=None*)

Spectral Co-Clustering algorithm (Dhillon, 2001).

Clusters rows and columns of an array `X` to solve the relaxed normalized cut of the bipartite graph created from `X` as follows: the edge between row vertex `i` and column vertex `j` has weight `X[i, j]`.

The resulting bicluster structure is block-diagonal, since each row and each column belongs to exactly one bicluster.

Supports sparse matrices, as long as they are nonnegative.

Read more in the *User Guide*.

Parameters

n_clusters [int, default=3] The number of biclusters to find.

svd_method [{'randomized', 'arpack'}, default='randomized'] Selects the algorithm for finding singular vectors. May be 'randomized' or 'arpack'. If 'randomized', use *sklearn.utils.extmath.randomized_svd*, which may be faster for large matrices. If 'arpack', use `scipy.sparse.linalg.svds`, which is more accurate, but possibly slower in some cases.

n_svd_vecs [int, default=None] Number of vectors to use in calculating the SVD. Corresponds to `ncv` when `svd_method=arpack` and `n_oversamples` when `svd_method` is 'randomized'.

mini_batch [bool, default=False] Whether to use mini-batch k-means, which is faster but may get different results.

init [{'k-means++', 'random', or ndarray of shape (n_clusters, n_features), default='k-means++'] Method for initialization of k-means algorithm; defaults to 'k-means++'.

n_init [int, default=10] Number of random initializations that are tried with the k-means algorithm.

If mini-batch k-means is used, the best initialization is chosen and the algorithm runs once. Otherwise, the algorithm is run for each initialization and the best solution chosen.

n_jobs [int, default=None] The number of jobs to use for the computation. This works by breaking down the pairwise matrix into n_jobs even slices and computing them in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance, default=None] Used for randomizing the singular value decomposition and the k-means initialization. Use an int to make the randomness deterministic. See *Glossary*.

Attributes

rows_ [array-like of shape (n_row_clusters, n_rows)] Results of the clustering. `rows[i, r]` is True if cluster `i` contains row `r`. Available only after calling `fit`.

columns_ [array-like of shape (n_column_clusters, n_columns)] Results of the clustering, like `rows`.

row_labels_ [array-like of shape (n_rows,)] The bicluster label of each row.

column_labels_ [array-like of shape (n_cols,)] The bicluster label of each column.

References

- Dhillon, Inderjit S, 2001. Co-clustering documents and words using bipartite spectral graph partitioning.

Examples

```
>>> from sklearn.cluster import SpectralCoclustering
>>> import numpy as np
>>> X = np.array([[1, 1], [2, 1], [1, 0],
...               [4, 7], [3, 5], [3, 6]])
>>> clustering = SpectralCoclustering(n_clusters=2, random_state=0).fit(X)
>>> clustering.row_labels_ #doctest: +SKIP
array([0, 1, 1, 0, 0, 0], dtype=int32)
>>> clustering.column_labels_ #doctest: +SKIP
array([0, 0], dtype=int32)
>>> clustering
SpectralCoclustering(n_clusters=2, random_state=0)
```

Methods

fit(self, X[, y])	Creates a biclustering for X.
get_indices(self, i)	Row and column indices of the i'th bicluster.
get_params(self[, deep])	Get parameters for this estimator.
get_shape(self, i)	Shape of the i'th bicluster.
get_submatrix(self, i, data)	Return the submatrix corresponding to bicluster `i`.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *n_clusters=3*, *svd_method='randomized'*, *n_svd_vecs=None*, *mini_batch=False*, *init='k-means++'*, *n_init=10*, *n_jobs=None*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

property biclusters_
Convenient way to get row and column indicators together.

Returns the `rows_` and `columns_` members.

fit (*self*, *X*, *y=None*)
Creates a biclustering for X.

Parameters

X [array-like, shape (n_samples, n_features)]

y [Ignored]

get_indices (*self*, *i*)
Row and column indices of the i'th bicluster.

Only works if `rows_` and `columns_` attributes exist.

Parameters

i [int] The index of the cluster.

Returns

row_ind [np.array, dtype=np.intp] Indices of rows in the dataset that belong to the bicluster.

col_ind [np.array, dtype=np.intp] Indices of columns in the dataset that belong to the bicluster.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_shape(*self*, *i*)
Shape of the i'th bicluster.

Parameters

i [int] The index of the cluster.

Returns

shape [(int, int)] Number of rows and columns (resp.) in the bicluster.

get_submatrix(*self*, *i*, *data*)
Return the submatrix corresponding to bicluster `i`.

Parameters

i [int] The index of the cluster.

data [array] The data.

Returns

submatrix [array] The submatrix corresponding to bicluster i.

Notes

Works with sparse matrices. Only works if `rows_` and `columns_` attributes exist.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.cluster.SpectralCoclustering`

- *A demo of the Spectral Co-Clustering algorithm*
- *Biclustering documents with the Spectral Co-clustering algorithm*

7.3.2 Functions

`cluster.affinity_propagation`(S[, ...])	Perform Affinity Propagation Clustering of data
`cluster.cluster_optics_dbscan`(reachability, ...)	Performs DBSCAN extraction for an arbitrary epsilon.
`cluster.cluster_optics_xi`(reachability, ...)	Automatically extract clusters according to the Xi-steep method.
`cluster.compute_optics_graph`(X, min_samples, ...)	Computes the OPTICS reachability graph.
`cluster.dbscan`(X[, eps, min_samples, ...])	Perform DBSCAN clustering from vector array or distance matrix.
`cluster.estimate_bandwidth`(X[, quantile, ...])	Estimate the bandwidth to use with the mean-shift algorithm.
`cluster.k_means`(X, n_clusters[, ...])	K-means clustering algorithm.
`cluster.mean_shift`(X[, bandwidth, seeds, ...])	Perform mean shift clustering of data using a flat kernel.
`cluster.spectral_clustering`(affinity[, ...])	Apply clustering to a projection of the normalized Laplacian.
`cluster.ward_tree`(X[, connectivity, ...])	Ward clustering based on a Feature matrix.

sklearn.cluster.affinity_propagation

sklearn.cluster.**affinity_propagation**(*S*, *preference=None*, *convergence_iter=15*, *max_iter=200*, *damping=0.5*, *copy=True*, *verbose=False*, *return_n_iter=False*)

Perform Affinity Propagation Clustering of data

Read more in the *User Guide*.

Parameters

S [array-like, shape (n_samples, n_samples)] Matrix of similarities between points

preference [array-like, shape (n_samples,) or float, optional] Preferences for each point - points with larger values of preferences are more likely to be chosen as exemplars. The number of exemplars, i.e. of clusters, is influenced by the input preferences value. If the preferences are not passed as arguments, they will be set to the median of the input similarities (resulting in a moderate number of clusters). For a smaller amount of clusters, this can be set to the minimum value of the similarities.

convergence_iter [int, optional, default: 15] Number of iterations with no change in the number of estimated clusters that stops the convergence.

max_iter [int, optional, default: 200] Maximum number of iterations

damping [float, optional, default: 0.5] Damping factor between 0.5 and 1.

copy [boolean, optional, default: True] If copy is False, the affinity matrix is modified inplace by the algorithm, for memory efficiency

verbose [boolean, optional, default: False] The verbosity level

return_n_iter [bool, default False] Whether or not to return the number of iterations.

Returns

cluster_centers_indices [array, shape (n_clusters,)] index of clusters centers

labels [array, shape (n_samples,)] cluster labels for each point

n_iter [int] number of iterations run. Returned only if `return_n_iter` is set to True.

Notes

For an example, see *examples/cluster/plot_affinity_propagation.py*.

When the algorithm does not converge, it returns an empty array as `cluster_center_indices` and `-1` as label for each training sample.

When all training samples have equal similarities and equal preferences, the assignment of cluster centers and labels depends on the preference. If the preference is smaller than the similarities, a single cluster center and label `0` for every sample will be returned. Otherwise, every training sample becomes its own cluster center and is assigned a unique label.

References

Brendan J. Frey and Delbert Dueck, "Clustering by Passing Messages Between Data Points", Science Feb. 2007

Examples using `sklearn.cluster.affinity_propagation`

- *Visualizing the stock market structure*

`sklearn.cluster`.cluster_optics_dbscan

`sklearn.cluster.`**`cluster_optics_dbscan`**(*reachability*, *core_distances*, *ordering*, *eps*)

Performs DBSCAN extraction for an arbitrary epsilon.

Extracting the clusters runs in linear time. Note that this results in `labels_` which are close to a *DBSCAN* with similar settings and `eps`, only if `eps` is close to `max_eps`.

Parameters

reachability [array, shape (n_samples,)] Reachability distances calculated by OPTICS (`reachability_`)

core_distances [array, shape (n_samples,)] Distances at which points become core (`core_distances_`)

ordering [array, shape (n_samples,)] OPTICS ordered point indices (`ordering_`)

eps [float] DBSCAN `eps` parameter. Must be set to < `max_eps`. Results will be close to DBSCAN algorithm if `eps` and `max_eps` are close to one another.

Returns

labels_ [array, shape (n_samples,)] The estimated labels.

Examples using `sklearn.cluster.cluster_optics_dbscan`

- *Demo of OPTICS clustering algorithm*

sklearn.cluster.cluster_optics_xi

sklearn.cluster.**cluster_optics_xi**(*reachability*, *predecessor*, *ordering*, *min_samples*, *min_cluster_size=None*, *xi=0.05*, *predecessor_correction=True*)

Automatically extract clusters according to the Xi-steep method.

Parameters

reachability [array, shape (n_samples,)] Reachability distances calculated by OPTICS (`reachability_`)

predecessor [array, shape (n_samples,)] Predecessors calculated by OPTICS.

ordering [array, shape (n_samples,)] OPTICS ordered point indices (`ordering_`)

min_samples [int > 1 or float between 0 and 1] The same as the min_samples given to OPTICS. Up and down steep regions can't have more then `min_samples` consecutive non-steep points. Expressed as an absolute number or a fraction of the number of samples (rounded to be at least 2).

min_cluster_size [int > 1 or float between 0 and 1 (default=None)] Minimum number of samples in an OPTICS cluster, expressed as an absolute number or a fraction of the number of samples (rounded to be at least 2). If `None`, the value of `min_samples` is used instead.

xi [float, between 0 and 1, optional (default=0.05)] Determines the minimum steepness on the reachability plot that constitutes a cluster boundary. For example, an upwards point in the reachability plot is defined by the ratio from one point to its successor being at most 1-xi.

predecessor_correction [bool, optional (default=True)] Correct clusters based on the calculated predecessors.

Returns

labels [array, shape (n_samples)] The labels assigned to samples. Points which are not included in any cluster are labeled as -1.

clusters [array, shape (n_clusters, 2)] The list of clusters in the form of `[start, end]` in each row, with all indices inclusive. The clusters are ordered according to `(end, -start)` (ascending) so that larger clusters encompassing smaller clusters come after such nested smaller clusters. Since `labels` does not reflect the hierarchy, usually `len(clusters) > np.unique(labels)`.

sklearn.cluster.compute_optics_graph

sklearn.cluster.**compute_optics_graph**(*X*, *min_samples*, *max_eps*, *metric*, *p*, *metric_params*, *algorithm*, *leaf_size*, *n_jobs*)

Computes the OPTICS reachability graph.

Read more in the *User Guide*.

Parameters

X [array, shape (n_samples, n_features), or (n_samples, n_samples) if metric='precomputed'.] A feature array, or array of distances between samples if metric='precomputed'

min_samples [int > 1 or float between 0 and 1] The number of samples in a neighborhood for a point to be considered as a core point. Expressed as an absolute number or a fraction of the number of samples (rounded to be at least 2).

max_eps [float, optional (default=np.inf)] The maximum distance between two samples for one to be considered as in the neighborhood of the other. Default value of `np.inf` will identify clusters across all scales; reducing `max_eps` will result in shorter run times.

metric [string or callable, optional (default='minkowski')] Metric to use for distance computation. Any metric from scikit-learn or scipy.spatial.distance can be used.

If metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays as input and return one value indicating the distance between them. This works for Scipy's metrics, but is less efficient than passing the metric name as a string. If metric is "precomputed", X is assumed to be a distance matrix and must be square.

Valid values for metric are:

- from scikit-learn: ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']
- from scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule']

See the documentation for scipy.spatial.distance for details on these metrics.

p [integer, optional (default=2)] Parameter for the Minkowski metric from *sklearn.metrics.pairwise_distances*. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params [dict, optional (default=None)] Additional keyword arguments for the metric function.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit` method. (default)

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, optional (default=30)] Leaf size passed to `BallTree` or `KDTree`. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Returns

ordering_ [array, shape (n_samples,)] The cluster ordered list of sample indices.

core_distances_ [array, shape (n_samples,)] Distance at which each sample becomes a core point, indexed by object order. Points which will never be core have a distance of inf. Use `clust.core_distances_[clust.ordering_]` to access in cluster order.

reachability_ [array, shape (n_samples,)] Reachability distances per sample, indexed by object order. Use `clust.reachability_[clust.ordering_]` to access in cluster order.

predecessor_ [array, shape (n_samples,)] Point that a sample was reached from, indexed by object order. Seed points have a predecessor of -1.

References

[1]

sklearn.cluster.dbscan

sklearn.cluster.**dbscan**(*X*, *eps=0.5*, *min_samples=5*, *metric='minkowski'*, *metric_params=None*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *sample_weight=None*, *n_jobs=None*)

Perform DBSCAN clustering from vector array or distance matrix.

Read more in the *User Guide*.

Parameters

X [array or sparse (CSR) matrix of shape (n_samples, n_features), or array of shape (n_samples, n_samples)] A feature array, or array of distances between samples if `metric='precomputed'`.

eps [float, optional] The maximum distance between two samples for one to be considered as in the neighborhood of the other. This is not a maximum bound on the distances of points within a cluster. This is the most important DBSCAN parameter to choose appropriately for your data set and distance function.

min_samples [int, optional] The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself.

metric [string, or callable] The metric to use when calculating distance between instances in a feature array. If metric is a string or callable, it must be one of the options allowed by *sklearn.metrics.pairwise_distances* for its metric parameter. If metric is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a *Glossary*, in which case only "nonzero" elements may be considered neighbors.

metric_params [dict, optional] Additional keyword arguments for the metric function.

New in version 0.19.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] The algorithm to be used by the NearestNeighbors module to compute pointwise distances and find nearest neighbors. See NearestNeighbors module documentation for details.

leaf_size [int, optional (default = 30)] Leaf size passed to BallTree or cKDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p [float, optional] The power of the Minkowski metric to be used to calculate distance between points.

sample_weight [array, shape (n_samples,), optional] Weight of each sample, such that a sample with a weight of at least `min_samples` is by itself a core sample; a sample with negative weight may inhibit its eps-neighbor from being core. Note that weights are absolute, and default to 1.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Returns

core_samples [array [n_core_samples]] Indices of core samples.

labels [array [n_samples]] Cluster labels for each point. Noisy samples are given the label -1.

See also:

DBSCAN An estimator interface for this clustering algorithm.

OPTICS A similar estimator interface clustering at multiple values of eps. Our implementation is optimized for memory usage.

Notes

For an example, see *examples/cluster/plot_dbscan.py*.

This implementation bulk-computes all neighborhood queries, which increases the memory complexity to O(n.d) where d is the average number of neighbors, while original DBSCAN had memory complexity O(n). It may attract a higher memory complexity when querying these nearest neighborhoods, depending on the `algorithm`.

One way to avoid the query complexity is to pre-compute sparse neighborhoods in chunks using *NearestNeighbors.radius_neighbors_graph* with `mode='distance'`, then using `metric='precomputed'` here.

Another way to reduce memory and computation time is to remove (near-)duplicate points and use `sample_weight` instead.

`cluster.optics` provides a similar clustering with lower memory usage.

References

Ester, M., H. P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press, pp. 226-231. 1996

Schubert, E., Sander, J., Ester, M., Kriegel, H. P., & Xu, X. (2017). DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. ACM Transactions on Database Systems (TODS), 42(3), 19.

sklearn.cluster.estimate_bandwidth

sklearn.cluster.**estimate_bandwidth**(*X*, *quantile=0.3*, *n_samples=None*, *random_state=0*, *n_jobs=None*)

Estimate the bandwidth to use with the mean-shift algorithm.

That this function takes time at least quadratic in n_samples. For large datasets, it's wise to set that parameter to a small value.

Parameters

X [array-like of shape (n_samples, n_features)] Input points.

quantile [float, default 0.3] should be between [0, 1] 0.5 means that the median of all pairwise distances is used.

n_samples [int, optional] The number of samples to use. If not given, all samples are used.

random_state [int, RandomState instance or None (default)] The generator used to randomly select the samples from input points for bandwidth estimation. Use an int to make the randomness deterministic. See *Glossary*.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Returns

bandwidth [float] The bandwidth parameter.

Examples using `sklearn.cluster.estimate_bandwidth`

- *A demo of the mean-shift clustering algorithm*
- *Comparing different clustering algorithms on toy datasets*

`sklearn.cluster.k_means`

`sklearn.cluster.k_means(X, n_clusters, sample_weight=None, init='k-means++', precompute_distances='auto', n_init=10, max_iter=300, verbose=False, tol=0.0001, random_state=None, copy_x=True, n_jobs=None, algorithm='auto', return_n_iter=False)`

K-means clustering algorithm.

Read more in the *User Guide*.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] The observations to cluster. It must be noted that the data will be converted to C ordering, which will cause a memory copy if the given data is not C-contiguous.

n_clusters [int] The number of clusters to form as well as the number of centroids to generate.

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None)

init [{'k-means++', 'random', or ndarray, or a callable}, optional] Method for initialization, default to 'k-means++':

'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence. See section Notes in k_init for more details.

'random': choose k observations (rows) at random from data for the initial centroids.

If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.

If a callable is passed, it should take arguments X, k and and a random state and return an initialization.

precompute_distances [{'auto', True, False}] Precompute distances (faster but takes more memory).

'auto' : do not precompute distances if n_samples * n_clusters > 12 million. This corresponds to about 100MB overhead per job using double precision.

True : always precompute distances

False : never precompute distances

n_init [int, optional, default: 10] Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

max_iter [int, optional, default 300] Maximum number of iterations of the k-means algorithm to run.

verbose [boolean, optional] Verbosity mode.

tol [float, optional] The relative increment in the results before declaring convergence.

random_state [int, RandomState instance or None (default)] Determines random number generation for centroid initialization. Use an int to make the randomness deterministic. See *Glossary*.

copy_x [bool, optional] When pre-computing distances it is more numerically accurate to center the data first. If copy_x is True (default), then the original data is not modified, ensuring X is C-contiguous. If False, the original data is modified, and put back before the function returns, but small numerical differences may be introduced by subtracting and then adding the data mean, in this case it will also not ensure that data is C-contiguous which may cause a significant slowdown.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

algorithm ["auto", "full" or "elkan", default="auto"] K-means algorithm to use. The classical EM-style algorithm is "full". The "elkan" variation is more efficient by using the triangle inequality, but currently doesn't support sparse data. "auto" chooses "elkan" for dense data and "full" for sparse data.

return_n_iter [bool, optional] Whether or not to return the number of iterations.

Returns

centroid [float ndarray with shape (k, n_features)] Centroids found at the last iteration of k-means.

label [integer ndarray with shape (n_samples,)] label[i] is the code or index of the centroid the i'th observation is closest to.

inertia [float] The final value of the inertia criterion (sum of squared distances to the closest centroid for all observations in the training set).

best_n_iter [int] Number of iterations corresponding to the best results. Returned only if `return_n_iter` is set to True.

`sklearn.cluster`.mean_shift

`sklearn.cluster.`**`mean_shift`**(*X*, *bandwidth=None*, *seeds=None*, *bin_seeding=False*, *min_bin_freq=1*, *cluster_all=True*, *max_iter=300*, *n_jobs=None*)

Perform mean shift clustering of data using a flat kernel.

Read more in the *User Guide*.

Parameters

X [array-like of shape (n_samples, n_features)] Input data.

bandwidth [float, optional] Kernel bandwidth.

If bandwidth is not given, it is determined using a heuristic based on the median of all pairwise distances. This will take quadratic time in the number of samples. The sklearn.cluster.estimate_bandwidth function can be used to do this more efficiently.

seeds [array-like of shape (n_seeds, n_features) or None] Point used as initial kernel locations. If None and bin_seeding=False, each data point is used as a seed. If None and bin_seeding=True, see bin_seeding.

bin_seeding [boolean, default=False] If true, initial kernel locations are not locations of all points, but rather the location of the discretized version of points, where points are binned onto a grid whose coarseness corresponds to the bandwidth. Setting this option to True will speed up the algorithm because fewer seeds will be initialized. Ignored if seeds argument is not None.

min_bin_freq [int, default=1] To speed up the algorithm, accept only those bins with at least min_bin_freq points as seeds.

cluster_all [boolean, default True] If true, then all points are clustered, even those orphans that are not within any kernel. Orphans are assigned to the nearest kernel. If false, then orphans are given cluster label -1.

max_iter [int, default 300] Maximum number of iterations, per seed point before the clustering operation terminates (for that seed point), if has not converged yet.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

New in version 0.17: Parallel Execution using *n_jobs*.

Returns

cluster_centers [array, shape=[n_clusters, n_features]] Coordinates of cluster centers.

labels [array, shape=[n_samples]] Cluster labels for each point.

Notes

For an example, see *examples/cluster/plot_mean_shift.py*.

sklearn.cluster.spectral_clustering

sklearn.cluster.**spectral_clustering**(*affinity*, *n_clusters=8*, *n_components=None*, *eigen_solver=None*, *random_state=None*, *n_init=10*, *eigen_tol=0.0*, *assign_labels='kmeans'*)

Apply clustering to a projection of the normalized Laplacian.

In practice Spectral Clustering is very useful when the structure of the individual clusters is highly non-convex or more generally when a measure of the center and spread of the cluster is not a suitable description of the complete cluster. For instance, when clusters are nested circles on the 2D plane.

If affinity is the adjacency matrix of a graph, this method can be used to find normalized graph cuts.

Read more in the *User Guide*.

Parameters

affinity [array-like or sparse matrix, shape: (n_samples, n_samples)] The affinity matrix describing the relationship of the samples to embed. **Must be symmetric**.

Possible examples:

- adjacency matrix of a graph,
- heat kernel of the pairwise distance matrix of the samples,
- symmetric k-nearest neighbours connectivity matrix of the samples.

n_clusters [integer, optional] Number of clusters to extract.

n_components [integer, optional, default is n_clusters] Number of eigen vectors to use for the spectral embedding

eigen_solver [{None, 'arpack', 'lobpcg', or 'amg'}] The eigenvalue decomposition strategy to use. AMG requires pyamg to be installed. It can be faster on very large, sparse problems, but may also lead to instabilities

random_state [int, RandomState instance or None (default)] A pseudo random number generator used for the initialization of the lobpcg eigen vectors decomposition when eigen_solver == 'amg' and by the K-Means initialization. Use an int to make the randomness deterministic. See *Glossary*.

n_init [int, optional, default: 10] Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

eigen_tol [float, optional, default: 0.0] Stopping criterion for eigendecomposition of the Laplacian matrix when using arpack eigen_solver.

assign_labels [{'kmeans', 'discretize'}, default: 'kmeans'] The strategy to use to assign labels in the embedding space. There are two ways to assign labels after the laplacian embedding. k-means can be applied and is a popular choice. But it can also be sensitive to initialization. Discretization is another approach which is less sensitive to random initialization. See the 'Multiclass spectral clustering' paper referenced below for more details on the discretization approach.

Returns

labels [array of integers, shape: n_samples] The labels of the clusters.

Notes

The graph should contain only one connect component, elsewhere the results make little sense.

This algorithm solves the normalized cut for k=2: it is a normalized spectral clustering.

References

- Normalized cuts and image segmentation, 2000 Jianbo Shi, Jitendra Malik http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.160.2324
- A Tutorial on Spectral Clustering, 2007 Ulrike von Luxburg http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.165.9323
- Multiclass spectral clustering, 2003 Stella X. Yu, Jianbo Shi https://www1.icsi.berkeley.edu/~stellayu/publication/doc/2003kwayICCV.pdf

Examples using `sklearn.cluster.spectral_clustering`

- *Segmenting the picture of greek coins in regions*
- *Spectral clustering for image segmentation*

sklearn.cluster.ward_tree

sklearn.cluster.**ward_tree**(*X*, *connectivity=None*, *n_clusters=None*, *return_distance=False*)

Ward clustering based on a Feature matrix.

Recursively merges the pair of clusters that minimally increases within-cluster variance.

The inertia matrix uses a Heapq-based representation.

This is the structured version, that takes into account some topological structure between samples.

Read more in the *User Guide*.

Parameters

X [array, shape (n_samples, n_features)] feature matrix representing n_samples samples to be clustered

connectivity [sparse matrix (optional).] connectivity matrix. Defines for each sample the neighboring samples following a given structure of the data. The matrix is assumed to be symmetric and only the upper triangular half is used. Default is None, i.e, the Ward algorithm is unstructured.

n_clusters [int (optional)] Stop early the construction of the tree at n_clusters. This is useful to decrease computation time if the number of clusters is not small compared to the number of samples. In this case, the complete tree is not computed, thus the 'children' output is of limited use, and the 'parents' output should rather be used. This option is valid only when specifying a connectivity matrix.

return_distance [bool (optional)] If True, return the distance between the clusters.

Returns

children [2D array, shape (n_nodes-1, 2)] The children of each non-leaf node. Values less than `n_samples` correspond to leaves of the tree which are the original samples. A node `i` greater than or equal to `n_samples` is a non-leaf node and has children `children_[i - n_samples]`. Alternatively at the i-th iteration, children[i][0] and children[i][1] are merged to form node `n_samples + i`

n_connected_components [int] The number of connected components in the graph.

n_leaves [int] The number of leaves in the tree

parents [1D array, shape (n_nodes,) or None] The parent of each node. Only returned when a connectivity matrix is specified, elsewhere 'None' is returned.

distances [1D array, shape (n_nodes-1,)] Only returned if return_distance is set to True (for compatibility). The distances between the centers of the nodes. `distances[i]` corresponds to a weighted euclidean distance between the nodes `children[i, 1]` and `children[i, 2]`. If the nodes refer to leaves of the tree, then `distances[i]` is their unweighted euclidean distance. Distances are updated in the following way (from scipy.hierarchy.linkage):

The new entry $d(u, v)$ is computed as follows,

$$d(u,v) = \sqrt{\frac{|v|+|s|}{T}d(v,s)^2 + \frac{|v|+|t|}{T}d(v,t)^2 - \frac{|v|}{T}d(s,t)^2}$$

where u is the newly joined cluster consisting of clusters s and t, v is an unused cluster in the forest, $T = |v| + |s| + |t|$, and $|*|$ is the cardinality of its argument. This is also known as the incremental algorithm.

7.4 `sklearn.compose`: Composite Estimators

Meta-estimators for building composite models with transformers

In addition to its current contents, this module will eventually be home to refurbished versions of Pipeline and FeatureUnion.

User guide: See the *Pipelines and composite estimators* section for further details.

`compose.ColumnTransformer`(transformers[, ...])	Applies transformers to columns of an array or pandas DataFrame.
`compose.TransformedTargetRegressor`([...])	Meta-estimator to regress on a transformed target.

7.4.1 `sklearn.compose`.ColumnTransformer

class sklearn.compose.**ColumnTransformer**(*transformers*, *remainder='drop'*, *sparse_threshold=0.3*, *n_jobs=None*, *transformer_weights=None*, *verbose=False*)

Applies transformers to columns of an array or pandas DataFrame.

This estimator allows different columns or column subsets of the input to be transformed separately and the features generated by each transformer will be concatenated to form a single feature space. This is useful for heterogeneous or columnar data, to combine several feature extraction mechanisms or transformations into a single transformer.

Read more in the *User Guide*.

New in version 0.20.

Parameters

transformers [list of tuples] List of (name, transformer, column(s)) tuples specifying the transformer objects to be applied to subsets of the data.

name [string] Like in Pipeline and FeatureUnion, this allows the transformer and its parameters to be set using `set_params` and searched in grid search.

transformer [estimator or {'passthrough', 'drop'}] Estimator must support *fit* and *transform*. Special-cased strings 'drop' and 'passthrough' are accepted as well, to indicate to drop the columns or to pass them through untransformed, respectively.

column(s) [string or int, array-like of string or int, slice, boolean mask array or callable] Indexes the data on its second axis. Integers are interpreted as positional columns, while strings can reference DataFrame columns by name. A scalar string or int should be used where `transformer` expects X to be a 1d array-like (vector), otherwise a 2d array will be passed to the transformer. A callable is passed the input data `X` and can

return any of the above. To select multiple columns by name or dtype, you can use *make_column_transformer*.

remainder [{'drop', 'passthrough'} or estimator, default 'drop'] By default, only the specified columns in `transformers` are transformed and combined in the output, and the non-specified columns are dropped. (default of `'drop'`). By specifying `remainder='passthrough'`, all remaining columns that were not specified in `transformers` will be automatically passed through. This subset of columns is concatenated with the output of the transformers. By setting `remainder` to be an estimator, the remaining non-specified columns will use the `remainder` estimator. The estimator must support *fit* and *transform*. Note that using this feature requires that the DataFrame columns input at *fit* and *transform* have identical order.

sparse_threshold [float, default = 0.3] If the output of the different transformers contains sparse matrices, these will be stacked as a sparse matrix if the overall density is lower than this value. Use `sparse_threshold=0` to always return dense. When the transformed output consists of all dense data, the stacked result will be dense, and this keyword will be ignored.

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

transformer_weights [dict, optional] Multiplicative weights for features per transformer. The output of the transformer is multiplied by these weights. Keys are transformer names, values the weights.

verbose [boolean, optional(default=False)] If True, the time elapsed while fitting each transformer will be printed as it is completed.

Attributes

transformers_ [list] The collection of fitted transformers as tuples of (name, fitted_transformer, column). `fitted_transformer` can be an estimator, 'drop', or 'passthrough'. In case there were no columns selected, this will be the unfitted transformer. If there are remaining columns, the final element is a tuple of the form: ('remainder', transformer, remaining_columns) corresponding to the `remainder` parameter. If there are remaining columns, then `len(transformers_)==len(transformers)+1`, otherwise `len(transformers_)==len(transformers)`.

named_transformers_ [Bunch object, a dictionary with attribute access] Access the fitted transformer by name.

sparse_output_ [boolean] Boolean flag indicating wether the output of `transform` is a sparse matrix or a dense numpy array, which depends on the output of the individual transformers and the `sparse_threshold` keyword.

See also:

sklearn.compose.make_column_transformer convenience function for combining the outputs of multiple transformer objects applied to column subsets of the original feature space.

sklearn.compose.make_column_selector convenience function for selecting columns based on datatype or the columns name with a regex pattern.

Notes

The order of the columns in the transformed feature matrix follows the order of how the columns are specified in the `transformers` list. Columns of the original feature matrix that are not specified are dropped from the

resulting transformed feature matrix, unless specified in the `passthrough` keyword. Those columns specified with `passthrough` are added at the right to the output of the transformers.

Examples

```
>>> import numpy as np
>>> from sklearn.compose import ColumnTransformer
>>> from sklearn.preprocessing import Normalizer
>>> ct = ColumnTransformer(
...     [("norm1", Normalizer(norm='l1'), [0, 1]),
...      ("norm2", Normalizer(norm='l1'), slice(2, 4))])
>>> X = np.array([[0., 1., 2., 2.],
...               [1., 1., 0., 1.]])
>>> # Normalizer scales each row of X to unit norm. A separate scaling
>>> # is applied for the two first and two last elements of each
>>> # row independently.
>>> ct.fit_transform(X)
array([[0. , 1. , 0.5, 0.5],
       [0.5, 0.5, 0. , 1. ]])
```

Methods

fit(self, X[, y])	Fit all transformers using X.
fit_transform(self, X[, y])	Fit all transformers, transform the data and concatenate results.
get_feature_names(self)	Get feature names from all transformers.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **kwargs)	Set the parameters of this estimator.
transform(self, X)	Transform X separately by each transformer, concatenate results.

__init__(*self*, *transformers*, *remainder='drop'*, *sparse_threshold=0.3*, *n_jobs=None*, *transformer_weights=None*, *verbose=False*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit all transformers using X.

Parameters

X [array-like or DataFrame of shape [n_samples, n_features]] Input data, of which specified subsets are used to fit the transformers.

y [array-like, shape (n_samples, . . .), optional] Targets for supervised learning.

Returns

self [ColumnTransformer] This estimator

fit_transform(*self*, *X*, *y=None*)
Fit all transformers, transform the data and concatenate results.

Parameters

X [array-like or DataFrame of shape [n_samples, n_features]] Input data, of which specified subsets are used to fit the transformers.

y [array-like, shape (n_samples, ...), optional] Targets for supervised learning.

Returns

X_t [array-like or sparse matrix, shape (n_samples, sum_n_components)] hstack of results of transformers. sum_n_components is the sum of n_components (output dimension) over transformers. If any result is a sparse matrix, everything will be converted to sparse matrices.

get_feature_names(*self*)
Get feature names from all transformers.

Returns

feature_names [list of strings] Names of the features produced by transform.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property named_transformers_
Access the fitted transformer by name.

Read-only attribute to access any transformer by given name. Keys are transformer names and values are the fitted transformer objects.

set_params(*self*, ***kwargs*)
Set the parameters of this estimator.

Valid parameter keys can be listed with `get_params()`.

Returns

self

transform(*self*, *X*)
Transform X separately by each transformer, concatenate results.

Parameters

X [array-like or DataFrame of shape [n_samples, n_features]] The data to be transformed by subset.

Returns

X_t [array-like or sparse matrix, shape (n_samples, sum_n_components)] hstack of results of transformers. sum_n_components is the sum of n_components (output dimension) over transformers. If any result is a sparse matrix, everything will be converted to sparse matrices.

Examples using `sklearn.compose.ColumnTransformer`

- *Permutation Importance vs Random Forest Feature Importance (MDI)*
- *Column Transformer with Mixed Types*
- *Column Transformer with Heterogeneous Data Sources*

7.4.2 sklearn.compose.TransformedTargetRegressor

class sklearn.compose.**TransformedTargetRegressor**(*regressor=None*, *transformer=None*, *func=None*, *inverse_func=None*, *check_inverse=True*)

Meta-estimator to regress on a transformed target.

Useful for applying a non-linear transformation to the target `y` in regression problems. This transformation can be given as a Transformer such as the QuantileTransformer or as a function and its inverse such as `log` and `exp`.

The computation during `fit` is:

```
regressor.fit(X, func(y))
```

or:

```
regressor.fit(X, transformer.transform(y))
```

The computation during `predict` is:

```
inverse_func(regressor.predict(X))
```

or:

```
transformer.inverse_transform(regressor.predict(X))
```

Read more in the *User Guide*.

Parameters

regressor [object, default=LinearRegression()] Regressor object such as derived from `RegressorMixin`. This regressor will automatically be cloned each time prior to fitting.

transformer [object, default=None] Estimator object such as derived from `TransformerMixin`. Cannot be set at the same time as `func` and `inverse_func`. If `transformer` is `None` as well as `func` and `inverse_func`, the transformer will be an identity transformer. Note that the transformer will be cloned during fitting. Also, the transformer is restricting `y` to be a numpy array.

func [function, optional] Function to apply to `y` before passing to `fit`. Cannot be set at the same time as `transformer`. The function needs to return a 2-dimensional array. If `func` is `None`, the function used will be the identity function.

inverse_func [function, optional] Function to apply to the prediction of the regressor. Cannot be set at the same time as `transformer` as well. The function needs to return a 2-dimensional array. The inverse function is used to return predictions to the same space of the original training labels.

check_inverse [bool, default=True] Whether to check that `transform` followed by `inverse_transform` or `func` followed by `inverse_func` leads to the original targets.

Attributes

regressor_ [object] Fitted regressor.

transformer_ [object] Transformer used in `fit` and `predict`.

Notes

Internally, the target `y` is always converted into a 2-dimensional array to be used by scikit-learn transformers. At the time of prediction, the output will be reshaped to a have the same number of dimensions as `y`.

See *examples/compose/plot_transformed_target.py*.

Examples

```
>>> import numpy as np
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.compose import TransformedTargetRegressor
>>> tt = TransformedTargetRegressor(regressor=LinearRegression(),
...                                 func=np.log, inverse_func=np.exp)
>>> X = np.arange(4).reshape(-1, 1)
>>> y = np.exp(2 * X).ravel()
>>> tt.fit(X, y)
TransformedTargetRegressor(...)
>>> tt.score(X, y)
1.0
>>> tt.regressor_.coef_
array([2.])
```

Methods

fit(self, X, y, **fit_params)	Fit the model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the base regressor, applying inverse.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *regressor=None*, *transformer=None*, *func=None*, *inverse_func=None*, *check_inverse=True*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, ***fit_params*)
Fit the model according to the given training data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape (n_samples,)] Target values.

****fit_params** [dict of string -> object] Parameters passed to the `fit` method of the underlying regressor.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)

Predict using the base regressor, applying inverse.

The regressor is used to predict and the `inverse_func` or `inverse_transform` is applied before returning the prediction.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Samples.

Returns

y_hat [array, shape = (n_samples,)] Predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.compose.TransformedTargetRegressor`

- *Effect of transforming the targets in regression model*

compose.make_column_transformer(...)	Construct a ColumnTransformer from the given transformers.
compose.make_column_selector([pattern, ...])	Create a callable to select columns to be used with `ColumnTransformer`.

7.4.3 `sklearn.compose`.make_column_transformer

`sklearn.compose.`**`make_column_transformer`**(**transformers*, ***kwargs*)

Construct a ColumnTransformer from the given transformers.

This is a shorthand for the ColumnTransformer constructor; it does not require, and does not permit, naming the transformers. Instead, they will be given names automatically based on their types. It also does not allow weighting with `transformer_weights`.

Read more in the *User Guide*.

Parameters

***transformers** [tuples] Tuples of the form (transformer, column(s)) specifying the transformer objects to be applied to subsets of the data.

transformer [estimator or {'passthrough', 'drop'}] Estimator must support *fit* and *transform*. Special-cased strings 'drop' and 'passthrough' are accepted as well, to indicate to drop the columns or to pass them through untransformed, respectively.

column(s) [string or int, array-like of string or int, slice, boolean mask array or callable] Indexes the data on its second axis. Integers are interpreted as positional columns, while strings can reference DataFrame columns by name. A scalar string or int should be used where `transformer` expects X to be a 1d array-like (vector), otherwise a 2d array will be passed to the transformer. A callable is passed the input data `X` and can return any of the above.

remainder [{'drop', 'passthrough'} or estimator, default 'drop'] By default, only the specified columns in `transformers` are transformed and combined in the output, and the non-specified columns are dropped. (default of `'drop'`). By specifying `remainder='passthrough'`, all remaining columns that were not specified in `transformers` will be automatically passed through. This subset of columns is concatenated with the output of the transformers. By setting `remainder` to be an estimator, the remaining non-specified columns will use the `remainder` estimator. The estimator must support *fit* and *transform*.

sparse_threshold [float, default = 0.3] If the transformed output consists of a mix of sparse and dense data, it will be stacked as a sparse matrix if the density is lower than this value. Use `sparse_threshold=0` to always return dense. When the transformed output consists of all sparse or all dense data, the stacked result will be sparse or dense, respectively, and this keyword will be ignored.

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [boolean, optional(default=False)] If True, the time elapsed while fitting each transformer will be printed as it is completed.

Returns

ct [ColumnTransformer]

See also:

sklearn.compose.ColumnTransformer Class that allows combining the outputs of multiple transformer objects used on column subsets of the data into a single feature space.

Examples

```
>>> from sklearn.preprocessing import StandardScaler, OneHotEncoder
>>> from sklearn.compose import make_column_transformer
>>> make_column_transformer(
...     (StandardScaler(), ['numerical_column']),
...     (OneHotEncoder(), ['categorical_column']))
ColumnTransformer(transformers=[('standardscaler', StandardScaler(...),
                                 ['numerical_column']),
                                ('onehotencoder', OneHotEncoder(...),
                                 ['categorical_column'])])
```

7.4.4 sklearn.compose.make_column_selector

sklearn.compose.**make_column_selector**(*pattern=None*, *dtype_include=None*, *dtype_exclude=None*)

Create a callable to select columns to be used with *ColumnTransformer*.

make_column_selector can select columns based on datatype or the columns name with a regex. When using multiple selection criteria, **all** criteria must match for a column to be selected.

Parameters

pattern [str, default=None] Name of columns containing this regex pattern will be included. If None, column selection will not be selected based on pattern.

dtype_include [column dtype or list of column dtypes, default=None] A selection of dtypes to include. For more details, see pandas.DataFrame.select_dtypes.

dtype_exclude [column dtype or list of column dtypes, default=None] A selection of dtypes to exclude. For more details, see pandas.DataFrame.select_dtypes.

Returns

selector [callable] Callable for column selection to be used by a *ColumnTransformer*.

See also:

sklearn.compose.ColumnTransformer Class that allows combining the outputs of multiple transformer objects used on column subsets of the data into a single feature space.

Examples

```
>>> from sklearn.preprocessing import StandardScaler, OneHotEncoder
>>> from sklearn.compose import make_column_transformer
>>> from sklearn.compose import make_column_selector
>>> import pandas as pd  # doctest: +SKIP
>>> X = pd.DataFrame({'city': ['London', 'London', 'Paris', 'Sallisaw'],
...                   'rating': [5, 3, 4, 5]})  # doctest: +SKIP
>>> ct = make_column_transformer(
...       (StandardScaler(),
...        make_column_selector(dtype_include=np.number)),  # rating
...       (OneHotEncoder(),
...        make_column_selector(dtype_include=object)))  # city
>>> ct.fit_transform(X)  # doctest: +SKIP
array([[ 0.90453403,  1.        ,  0.        ,  0.        ],
       [-1.50755672,  1.        ,  0.        ,  0.        ],
       [-0.30151134,  0.        ,  1.        ,  0.        ],
       [ 0.90453403,  0.        ,  0.        ,  1.        ]])
```

7.5 sklearn.covariance: Covariance Estimators

The *sklearn.covariance* module includes methods and algorithms to robustly estimate the covariance of features given a set of points. The precision matrix defined as the inverse of the covariance is also estimated. Covariance estimation is closely related to the theory of Gaussian Graphical Models.

User guide: See the *Covariance estimation* section for further details.

covariance.EmpiricalCovariance([...])	Maximum likelihood covariance estimator
covariance.EllipticEnvelope([...])	An object for detecting outliers in a Gaussian distributed dataset.
covariance.GraphicalLasso([alpha, mode, ...])	Sparse inverse covariance estimation with an l1-penalized estimator.
covariance.GraphicalLassoCV([alphas, ...])	Sparse inverse covariance w/ cross-validated choice of the l1 penalty.
covariance.LedoitWolf([store_precision, ...])	LedoitWolf Estimator
covariance.MinCovDet([store_precision, ...])	Minimum Covariance Determinant (MCD): robust estimator of covariance.
covariance.OAS([store_precision, ...])	Oracle Approximating Shrinkage Estimator
covariance.ShrunkCovariance([...])	Covariance estimator with shrinkage

7.5.1 sklearn.covariance.EmpiricalCovariance

class sklearn.covariance.**EmpiricalCovariance**(*store_precision=True*, *assume_centered=False*)

Maximum likelihood covariance estimator

Read more in the *User Guide*.

Parameters

store_precision [bool] Specifies if the estimated precision is stored.

assume_centered [bool] If True, data are not centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False (default), data are centered before computation.

Attributes

location_ [array-like, shape (n_features,)] Estimated location, i.e. the estimated mean.

covariance_ [2D ndarray, shape (n_features, n_features)] Estimated covariance matrix

precision_ [2D ndarray, shape (n_features, n_features)] Estimated pseudo-inverse matrix. (stored only if store_precision is True)

Examples

```
>>> import numpy as np
>>> from sklearn.covariance import EmpiricalCovariance
>>> from sklearn.datasets import make_gaussian_quantiles
>>> real_cov = np.array([[.8, .3],
...                      [.3, .4]])
>>> rng = np.random.RandomState(0)
>>> X = rng.multivariate_normal(mean=[0, 0],
...                             cov=real_cov,
...                             size=500)
>>> cov = EmpiricalCovariance().fit(X)
>>> cov.covariance_
array([[0.7569..., 0.2818...],
       [0.2818..., 0.3928...]])
>>> cov.location_
array([0.0622..., 0.0193...])
```

Methods

error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits the Maximum Likelihood Estimator covariance model according to the given training data and parameters.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *store_precision=True*, *assume_centered=False*)
Initialize self. See help(type(self)) for accurate signature.

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)
Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like of shape (n_features, n_features)] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius'

(default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

`fit`(*self*, *X*, *y=None*)

Fits the Maximum Likelihood Estimator covariance model according to the given training data and parameters.

Parameters

X [array-like of shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y not used, present for API consistence purpose.

Returns

self [object]

`get_params`(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

`get_precision`(*self*)

Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

`mahalanobis`(*self*, *X*)

Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like of shape (n_samples, n_features)] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

`score`(*self*, *X_test*, *y=None*)

Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like of shape (n_samples, n_features)] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.covariance.EmpiricalCovariance`

- *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*
- *Robust covariance estimation and Mahalanobis distances relevance*
- *Robust vs Empirical covariance estimate*

7.5.2 sklearn.covariance.EllipticEnvelope

class sklearn.covariance.**EllipticEnvelope**(*store_precision=True*, *assume_centered=False*, *support_fraction=None*, *contamination=0.1*, *random_state=None*)
An object for detecting outliers in a Gaussian distributed dataset.

Read more in the *User Guide*.

Parameters

store_precision [boolean, optional (default=True)] Specify if the estimated precision is stored.

assume_centered [boolean, optional (default=False)] If True, the support of robust location and covariance estimates is computed, and a covariance estimate is recomputed from it, without centering the data. Useful to work with data whose mean is significantly equal to zero but is not exactly zero. If False, the robust location and covariance are directly computed with the FastMCD algorithm without additional treatment.

support_fraction [float in (0., 1.), optional (default=None)] The proportion of points to be included in the support of the raw MCD estimate. If None, the minimum value of support_fraction will be used within the algorithm: `[n_sample + n_features + 1] / 2`.

contamination [float in (0., 0.5), optional (default=0.1)] The amount of contamination of the data set, i.e. the proportion of outliers in the data set.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Attributes

location_ [array-like, shape (n_features,)] Estimated robust location

covariance_ [array-like, shape (n_features, n_features)] Estimated robust covariance matrix

precision_ [array-like, shape (n_features, n_features)] Estimated pseudo inverse matrix. (stored only if store_precision is True)

support_ [array-like, shape (n_samples,)] A mask of the observations that have been used to compute the robust estimates of location and shape.

offset_ [float] Offset used to define the decision function from the raw scores. We have the relation: `decision_function = score_samples - offset_`. The offset depends on the contamination parameter and is defined in such a way we obtain the expected number of outliers (samples with decision function < 0) in training.

See also:

EmpiricalCovariance, ***MinCovDet***

Notes

Outlier detection from covariance estimation may break or not perform well in high-dimensional settings. In particular, one will always take care to work with `n_samples > n_features ** 2`.

References

[R68ae096da0e4-1]

Examples

```
>>> import numpy as np
>>> from sklearn.covariance import EllipticEnvelope
>>> true_cov = np.array([[.8, .3],
...                      [.3, .4]])
>>> X = np.random.RandomState(0).multivariate_normal(mean=[0, 0],
...                                                  cov=true_cov,
...                                                  size=500)
>>> cov = EllipticEnvelope(random_state=0).fit(X)
>>> # predict returns 1 for an inlier and -1 for an outlier
>>> cov.predict([[0, 0],
...              [3, 3]])
array([ 1, -1])
>>> cov.covariance_
array([[0.7411..., 0.2535...],
       [0.2535..., 0.3053...]])
>>> cov.location_
array([0.0813... , 0.0427...])
```

Methods

correct_covariance(self, data)	Apply a correction to raw Minimum Covariance Determinant estimates.
decision_function(self, X)	Compute the decision function of the given observations.
error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fit the EllipticEnvelope model.
fit_predict(self, X[, y])	Perform fit on X and returns labels for X.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
predict(self, X)	Predict the labels (1 inlier, -1 outlier) of X according to the fitted model.
reweight_covariance(self, data)	Re-weight raw Minimum Covariance Determinant estimates.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
score_samples(self, X)	Compute the negative Mahalanobis distances.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *store_precision=True*, *assume_centered=False*, *support_fraction=None*, *contamination=0.1*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

correct_covariance(*self*, *data*)
Apply a correction to raw Minimum Covariance Determinant estimates.

Correction using the empirical correction factor suggested by Rousseeuw and Van Driessen in [RVD].

Parameters

data [array-like, shape (n_samples, n_features)] The data matrix, with p features and n samples. The data set must be the one which was used to compute the raw estimates.

Returns

covariance_corrected [array-like, shape (n_features, n_features)] Corrected robust covariance estimate.

References

[RVD]

decision_function(*self*, *X*)
Compute the decision function of the given observations.

Parameters

X [array-like, shape (n_samples, n_features)]

Returns

decision [array-like, shape (n_samples,)] Decision function of the samples. It is equal to the shifted Mahalanobis distances. The threshold for being an outlier is 0, which ensures a compatibility with other outlier detection algorithms.

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)
Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like of shape (n_features, n_features)] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

fit(*self*, *X*, *y=None*)
Fit the EllipticEnvelope model.

Parameters

X [numpy array or sparse matrix, shape (n_samples, n_features).] Training data

y [Ignored] not used, present for API consistency by convention.

fit_predict(*self*, *X*, *y=None*)
Perform fit on X and returns labels for X.

Returns -1 for outliers and 1 for inliers.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

y [Ignored] Not used, present for API consistency by convention.

Returns

y [ndarray, shape (n_samples,)] 1 for inliers, -1 for outliers.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_precision(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

mahalanobis (*self*, *X*)

Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like of shape (n_samples, n_features)] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

predict (*self*, *X*)

Predict the labels (1 inlier, -1 outlier) of X according to the fitted model.

Parameters

X [array-like, shape (n_samples, n_features)]

Returns

is_inlier [array, shape (n_samples,)] Returns -1 for anomalies/outliers and +1 for inliers.

reweight_covariance (*self*, *data*)

Re-weight raw Minimum Covariance Determinant estimates.

Re-weight observations using Rousseeuw's method (equivalent to deleting outlying observations from the data set before computing location and covariance estimates) described in [RVDriessen].

Parameters

data [array-like, shape (n_samples, n_features)] The data matrix, with p features and n samples. The data set must be the one which was used to compute the raw estimates.

Returns

location_reweighted [array-like, shape (n_features,)] Re-weighted robust location estimate.

covariance_reweighted [array-like, shape (n_features, n_features)] Re-weighted robust covariance estimate.

support_reweighted [array-like, type boolean, shape (n_samples,)] A mask of the observations that have been used to compute the re-weighted robust location and covariance estimates.

References

[RVDriessen]

score (*self*, *X*, *y*, *sample_weight=None*)

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like, shape (n_samples, n_features)] Test samples.

y [array-like, shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like, shape (n_samples,), optional] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

score_samples(*self*, *X*)

Compute the negative Mahalanobis distances.

Parameters

X [array-like, shape (n_samples, n_features)]

Returns

negative_mahal_distances [array-like, shape (n_samples,)] Opposite of the Mahalanobis distances.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.covariance.EllipticEnvelope`

- *Comparing anomaly detection algorithms for outlier detection on toy datasets*
- *Outlier detection on a real data set*

7.5.3 sklearn.covariance.GraphicalLasso

class sklearn.covariance.**GraphicalLasso**(*alpha=0.01*, *mode='cd'*, *tol=0.0001*, *enet_tol=0.0001*, *max_iter=100*, *verbose=False*, *assume_centered=False*)

Sparse inverse covariance estimation with an l1-penalized estimator.

Read more in the *User Guide*.

Parameters

alpha [positive float, default 0.01] The regularization parameter: the higher alpha, the more regularization, the sparser the inverse covariance.

mode [{'cd', 'lars'}, default 'cd'] The Lasso solver to use: coordinate descent or LARS. Use LARS for very sparse underlying graphs, where p > n. Elsewhere prefer cd which is more numerically stable.

tol [positive float, default 1e-4] The tolerance to declare convergence: if the dual gap goes below this value, iterations are stopped.

enet_tol [positive float, optional] The tolerance for the elastic net solver used to calculate the descent direction. This parameter controls the accuracy of the search direction for a given column update, not of the overall parameter estimate. Only used for mode='cd'.

max_iter [integer, default 100] The maximum number of iterations.

verbose [boolean, default False] If verbose is True, the objective function and dual gap are plotted at each iteration.

assume_centered [boolean, default False] If True, data are not centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False, data are centered before computation.

Attributes

location_ [array-like, shape (n_features,)] Estimated location, i.e. the estimated mean.

covariance_ [array-like, shape (n_features, n_features)] Estimated covariance matrix

precision_ [array-like, shape (n_features, n_features)] Estimated pseudo inverse matrix.

n_iter_ [int] Number of iterations run.

See also:

graphical_lasso, *GraphicalLassoCV*

Examples

```
>>> import numpy as np
>>> from sklearn.covariance import GraphicalLasso
>>> true_cov = np.array([[0.8, 0.0, 0.2, 0.0],
...                      [0.0, 0.4, 0.0, 0.0],
...                      [0.2, 0.0, 0.3, 0.1],
...                      [0.0, 0.0, 0.1, 0.7]])
>>> np.random.seed(0)
>>> X = np.random.multivariate_normal(mean=[0, 0, 0, 0],
...                                   cov=true_cov,
...                                   size=200)
>>> cov = GraphicalLasso().fit(X)
>>> np.around(cov.covariance_, decimals=3)
array([[0.816, 0.049, 0.218, 0.019],
       [0.049, 0.364, 0.017, 0.034],
       [0.218, 0.017, 0.322, 0.093],
       [0.019, 0.034, 0.093, 0.69 ]])
>>> np.around(cov.location_, decimals=3)
array([0.073, 0.04 , 0.038, 0.143])
```

Methods

error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits the GraphicalLasso model to X.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.

Continued on next page

Table 34 – continued from previous page

mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=0.01*, *mode='cd'*, *tol=0.0001*, *enet_tol=0.0001*, *max_iter=100*, *verbose=False*, *assume_centered=False*)
Initialize self. See help(type(self)) for accurate signature.

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)
Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like of shape (n_features, n_features)] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

fit(*self*, *X*, *y=None*)
Fits the GraphicalLasso model to X.

Parameters

X [ndarray, shape (n_samples, n_features)] Data from which to compute the covariance estimate

y [(ignored)]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_precision(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

mahalanobis(*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like of shape (n_samples, n_features)] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

score(*self*, *X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like of shape (n_samples, n_features)] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

7.5.4 sklearn.covariance.GraphicalLassoCV

class sklearn.covariance.**GraphicalLassoCV**(*alphas=4*, *n_refinements=4*, *cv=None*, *tol=0.0001*, *enet_tol=0.0001*, *max_iter=100*, *mode='cd'*, *n_jobs=None*, *verbose=False*, *assume_centered=False*)
Sparse inverse covariance w/ cross-validated choice of the l1 penalty.

See glossary entry for *cross-validation estimator*.

Read more in the *User Guide*.

Parameters

alphas [integer, or list positive float, optional] If an integer is given, it fixes the number of points on the grids of alpha to be used. If a list is given, it gives the grid to be used. See the notes in the class docstring for more details.

n_refinements [strictly positive integer] The number of times the grid is refined. Not used if explicit values of alphas are passed.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None changed from 3-fold to 5-fold.

tol [positive float, optional] The tolerance to declare convergence: if the dual gap goes below this value, iterations are stopped.

enet_tol [positive float, optional] The tolerance for the elastic net solver used to calculate the descent direction. This parameter controls the accuracy of the search direction for a given column update, not of the overall parameter estimate. Only used for mode='cd'.

max_iter [integer, optional] Maximum number of iterations.

mode [{'cd', 'lars'}] The Lasso solver to use: coordinate descent or LARS. Use LARS for very sparse underlying graphs, where number of features is greater than number of samples. Elsewhere prefer cd which is more numerically stable.

n_jobs [int or None, optional (default=None)] number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [boolean, optional] If verbose is True, the objective function and duality gap are printed at each iteration.

assume_centered [boolean] If True, data are not centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False, data are centered before computation.

Attributes

location_ [array-like, shape (n_features,)] Estimated location, i.e. the estimated mean.

covariance_ [numpy.ndarray, shape (n_features, n_features)] Estimated covariance matrix.

precision_ [numpy.ndarray, shape (n_features, n_features)] Estimated precision matrix (inverse covariance).

alpha_ [float] Penalization parameter selected.

cv_alphas_ [list of float] All penalization parameters explored.

grid_scores_ [2D numpy.ndarray (n_alphas, n_folds)] Log-likelihood score on left-out data across folds.

n_iter_ [int] Number of iterations run for the optimal alpha.

See also:

graphical_lasso*, *GraphicalLasso

Notes

The search for the optimal penalization parameter (alpha) is done on an iteratively refined grid: first the cross-validated scores on a grid are computed, then a new refined grid is centered around the maximum, and so on.

One of the challenges which is faced here is that the solvers can fail to converge to a well-conditioned estimate. The corresponding values of alpha then come out as missing values, but the optimum may be close to these missing values.

Examples

```
>>> import numpy as np
>>> from sklearn.covariance import GraphicalLassoCV
>>> true_cov = np.array([[0.8, 0.0, 0.2, 0.0],
...                      [0.0, 0.4, 0.0, 0.0],
...                      [0.2, 0.0, 0.3, 0.1],
...                      [0.0, 0.0, 0.1, 0.7]])
>>> np.random.seed(0)
>>> X = np.random.multivariate_normal(mean=[0, 0, 0, 0],
...                                   cov=true_cov,
...                                   size=200)
>>> cov = GraphicalLassoCV().fit(X)
>>> np.around(cov.covariance_, decimals=3)
array([[0.816, 0.051, 0.22 , 0.017],
       [0.051, 0.364, 0.018, 0.036],
       [0.22 , 0.018, 0.322, 0.094],
       [0.017, 0.036, 0.094, 0.69 ]])
>>> np.around(cov.location_, decimals=3)
array([0.073, 0.04 , 0.038, 0.143])
```

Methods

error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits the GraphicalLasso covariance model to X.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alphas=4*, *n_refinements=4*, *cv=None*, *tol=0.0001*, *enet_tol=0.0001*, *max_iter=100*, *mode='cd'*, *n_jobs=None*, *verbose=False*, *assume_centered=False*)
Initialize self. See help(type(self)) for accurate signature.

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)
Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like of shape (n_features, n_features)] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

`fit`(*self*, *X*, *y=None*)
Fits the GraphicalLasso covariance model to X.

Parameters

X [ndarray, shape (n_samples, n_features)] Data from which to compute the covariance estimate

y [(ignored)]

`get_params`(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

`get_precision`(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

`mahalanobis`(*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like of shape (n_samples, n_features)] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

`score`(*self*, *X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like of shape (n_samples, n_features)] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.covariance.GraphicalLassoCV`

- *Sparse inverse covariance estimation*
- *Visualizing the stock market structure*

7.5.5 `sklearn.covariance`.LedoitWolf

class sklearn.covariance.**LedoitWolf**(*store_precision=True*, *assume_centered=False*, *block_size=1000*)
LedoitWolf Estimator

Ledoit-Wolf is a particular form of shrinkage, where the shrinkage coefficient is computed using O. Ledoit and M. Wolf's formula as described in "A Well-Conditioned Estimator for Large-Dimensional Covariance Matrices", Ledoit and Wolf, Journal of Multivariate Analysis, Volume 88, Issue 2, February 2004, pages 365-411.

Read more in the *User Guide*.

Parameters

store_precision [bool, default=True] Specify if the estimated precision is stored.

assume_centered [bool, default=False] If True, data will not be centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False (default), data will be centered before computation.

block_size [int, default=1000] Size of the blocks into which the covariance matrix will be split during its Ledoit-Wolf estimation. This is purely a memory optimization and does not affect results.

Attributes

location_ [array-like, shape (n_features,)] Estimated location, i.e. the estimated mean.

covariance_ [array-like, shape (n_features, n_features)] Estimated covariance matrix

precision_ [array-like, shape (n_features, n_features)] Estimated pseudo inverse matrix. (stored only if store_precision is True)

shrinkage_ [float, 0 <= shrinkage <= 1] Coefficient in the convex combination used for the computation of the shrunk estimate.

Notes

The regularised covariance is:

(1 - shrinkage) * cov + shrinkage * mu * np.identity(n_features)

where mu = trace(cov) / n_features and shrinkage is given by the Ledoit and Wolf formula (see References)

References

"A Well-Conditioned Estimator for Large-Dimensional Covariance Matrices", Ledoit and Wolf, Journal of Multivariate Analysis, Volume 88, Issue 2, February 2004, pages 365-411.

Examples

```
>>> import numpy as np
>>> from sklearn.covariance import LedoitWolf
>>> real_cov = np.array([[.4, .2],
...                      [.2, .8]])
>>> np.random.seed(0)
>>> X = np.random.multivariate_normal(mean=[0, 0],
...                                   cov=real_cov,
...                                   size=50)
>>> cov = LedoitWolf().fit(X)
>>> cov.covariance_
array([[0.4406..., 0.1616...],
       [0.1616..., 0.8022...]])
>>> cov.location_
array([ 0.0595... , -0.0075...])
```

Methods

error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits the Ledoit-Wolf shrunk covariance model according to the given training data and parameters.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *store_precision=True*, *assume_centered=False*, *block_size=1000*)
Initialize self. See help(type(self)) for accurate signature.

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)
Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like of shape (n_features, n_features)] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

fit(*self*, *X*, *y=None*)
Fits the Ledoit-Wolf shrunk covariance model according to the given training data and parameters.

Parameters

X [array-like of shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y not used, present for API consistence purpose.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_precision(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

mahalanobis(*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like of shape (n_samples, n_features)] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

score (*self*, *X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like of shape (n_samples, n_features)] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.covariance.LedoitWolf`

- *Ledoit-Wolf vs OAS estimation*
- *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*
- *Model selection with Probabilistic PCA and Factor Analysis (FA)*

7.5.6 `sklearn.covariance`.MinCovDet

class sklearn.covariance.**MinCovDet** (*store_precision=True*, *assume_centered=False*, *support_fraction=None*, *random_state=None*)
Minimum Covariance Determinant (MCD): robust estimator of covariance.

The Minimum Covariance Determinant covariance estimator is to be applied on Gaussian-distributed data, but could still be relevant on data drawn from a unimodal, symmetric distribution. It is not meant to be used with multi-modal data (the algorithm used to fit a MinCovDet object is likely to fail in such a case). One should consider projection pursuit methods to deal with multi-modal datasets.

Read more in the *User Guide*.

Parameters

store_precision [bool] Specify if the estimated precision is stored.

assume_centered [bool] If True, the support of the robust location and the covariance estimates is computed, and a covariance estimate is recomputed from it, without centering the data. Useful to work with data whose mean is significantly equal to zero but is not exactly zero. If False, the robust location and covariance are directly computed with the FastMCD algorithm without additional treatment.

support_fraction [float, 0 < support_fraction < 1] The proportion of points to be included in the support of the raw MCD estimate. Default is None, which implies that the minimum value of support_fraction will be used within the algorithm: [n_sample + n_features + 1] / 2

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Attributes

raw_location_ [array-like, shape (n_features,)] The raw robust estimated location before correction and re-weighting.

raw_covariance_ [array-like, shape (n_features, n_features)] The raw robust estimated covariance before correction and re-weighting.

raw_support_ [array-like, shape (n_samples,)] A mask of the observations that have been used to compute the raw robust estimates of location and shape, before correction and re-weighting.

location_ [array-like, shape (n_features,)] Estimated robust location

covariance_ [array-like, shape (n_features, n_features)] Estimated robust covariance matrix

precision_ [array-like, shape (n_features, n_features)] Estimated pseudo inverse matrix. (stored only if store_precision is True)

support_ [array-like, shape (n_samples,)] A mask of the observations that have been used to compute the robust estimates of location and shape.

dist_ [array-like, shape (n_samples,)] Mahalanobis distances of the training set (on which *fit* is called) observations.

References

[R9f63e655f7bd-Rouseeuw1984], [R9f63e655f7bd-Rousseeuw], [R9f63e655f7bd-ButlerDavies]

Examples

```
>>> import numpy as np
>>> from sklearn.covariance import MinCovDet
>>> from sklearn.datasets import make_gaussian_quantiles
>>> real_cov = np.array([[.8, .3],
...                      [.3, .4]])
>>> rng = np.random.RandomState(0)
>>> X = rng.multivariate_normal(mean=[0, 0],
...                                   cov=real_cov,
...                                   size=500)
```

(continues on next page)

(continued from previous page)

```
>>> cov = MinCovDet(random_state=0).fit(X)
>>> cov.covariance_
array([[0.7411..., 0.2535...],
       [0.2535..., 0.3053...]])
>>> cov.location_
array([0.0813... , 0.0427...])
```

Methods

correct_covariance(self, data)	Apply a correction to raw Minimum Covariance Determinant estimates.
error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits a Minimum Covariance Determinant with the FastMCD algorithm.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
reweight_covariance(self, data)	Re-weight raw Minimum Covariance Determinant estimates.
score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *store_precision=True*, *assume_centered=False*, *support_fraction=None*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

correct_covariance(*self*, *data*)
Apply a correction to raw Minimum Covariance Determinant estimates.

Correction using the empirical correction factor suggested by Rousseeuw and Van Driessen in [RVD].

Parameters

data [array-like, shape (n_samples, n_features)] The data matrix, with p features and n samples. The data set must be the one which was used to compute the raw estimates.

Returns

covariance_corrected [array-like, shape (n_features, n_features)] Corrected robust covariance estimate.

References

[RVD]

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)
Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like of shape (n_features, n_features)] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

`fit`(*self*, *X*, *y=None*)
Fits a Minimum Covariance Determinant with the FastMCD algorithm.

Parameters

X [array-like of shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y not used, present for API consistence purpose.

Returns

self [object]

`get_params`(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

`get_precision`(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

`mahalanobis`(*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like of shape (n_samples, n_features)] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

`reweight_covariance`(*self*, *data*)
Re-weight raw Minimum Covariance Determinant estimates.

Re-weight observations using Rousseeuw's method (equivalent to deleting outlying observations from the data set before computing location and covariance estimates) described in [RVDriessen].

Parameters

data [array-like, shape (n_samples, n_features)] The data matrix, with p features and n samples. The data set must be the one which was used to compute the raw estimates.

Returns

location_reweighted [array-like, shape (n_features,)] Re-weighted robust location estimate.

covariance_reweighted [array-like, shape (n_features, n_features)] Re-weighted robust covariance estimate.

support_reweighted [array-like, type boolean, shape (n_samples,)] A mask of the observations that have been used to compute the re-weighted robust location and covariance estimates.

References

[RVDriessen]

score(*self*, *X_test*, *y=None*)

Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like of shape (n_samples, n_features)] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.covariance.MinCovDet`

- *Robust covariance estimation and Mahalanobis distances relevance*
- *Robust vs Empirical covariance estimate*

7.5.7 sklearn.covariance.OAS

class sklearn.covariance.**OAS**(*store_precision=True*, *assume_centered=False*)
Oracle Approximating Shrinkage Estimator

Read more in the *User Guide*.

OAS is a particular form of shrinkage described in "Shrinkage Algorithms for MMSE Covariance Estimation" Chen et al., IEEE Trans. on Sign. Proc., Volume 58, Issue 10, October 2010.

The formula used here does not correspond to the one given in the article. In the original article, formula (23) states that 2/p is multiplied by Trace(cov*cov) in both the numerator and denominator, but this operation is omitted because for a large p, the value of 2/p is so small that it doesn't affect the value of the estimator.

Parameters

store_precision [bool, default=True] Specify if the estimated precision is stored.

assume_centered [bool, default=False] If True, data will not be centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False (default), data will be centered before computation.

Attributes

covariance_ [array-like, shape (n_features, n_features)] Estimated covariance matrix.

precision_ [array-like, shape (n_features, n_features)] Estimated pseudo inverse matrix. (stored only if store_precision is True)

shrinkage_ [float, 0 <= shrinkage <= 1] coefficient in the convex combination used for the computation of the shrunk estimate.

Notes

The regularised covariance is:

(1 - shrinkage) * cov + shrinkage * mu * np.identity(n_features)

where mu = trace(cov) / n_features and shrinkage is given by the OAS formula (see References)

References

"Shrinkage Algorithms for MMSE Covariance Estimation" Chen et al., IEEE Trans. on Sign. Proc., Volume 58, Issue 10, October 2010.

Methods

error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits the Oracle Approximating Shrinkage covariance model according to the given training data and parameters.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.

Continued on next page

Table 38 – continued from previous page

score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with self.covariance_ as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *store_precision=True*, *assume_centered=False*)
Initialize self. See help(type(self)) for accurate signature.

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)
Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like of shape (n_features, n_features)] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error (comp_cov - self.covariance_).

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

self and comp_cov covariance estimators.

fit(*self*, *X*, *y=None*)
Fits the Oracle Approximating Shrinkage covariance model according to the given training data and parameters.

Parameters

X [array-like of shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y not used, present for API consistence purpose.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_precision(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

mahalanobis(*self*, *X*)

Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like of shape (n_samples, n_features)] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

score(*self*, *X_test*, *y=None*)

Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like of shape (n_samples, n_features)] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.covariance.OAS`

- *Ledoit-Wolf vs OAS estimation*
- *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*

7.5.8 `sklearn.covariance`.ShrunkCovariance

class sklearn.covariance.**ShrunkCovariance**(*store_precision=True*, *assume_centered=False*, *shrinkage=0.1*)

Covariance estimator with shrinkage

Read more in the *User Guide*.

Parameters

store_precision [boolean, default True] Specify if the estimated precision is stored

assume_centered [boolean, default False] If True, data will not be centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False, data will be centered before computation.

shrinkage [float, 0 <= shrinkage <= 1, default 0.1] Coefficient in the convex combination used for the computation of the shrunk estimate.

Attributes

location_ [array-like, shape (n_features,)] Estimated location, i.e. the estimated mean.

covariance_ [array-like, shape (n_features, n_features)] Estimated covariance matrix

precision_ [array-like, shape (n_features, n_features)] Estimated pseudo inverse matrix. (stored only if store_precision is True)

Notes

The regularized covariance is given by:

(1 - shrinkage) * cov + shrinkage * mu * np.identity(n_features)

where mu = trace(cov) / n_features

Examples

```
>>> import numpy as np
>>> from sklearn.covariance import ShrunkCovariance
>>> from sklearn.datasets import make_gaussian_quantiles
>>> real_cov = np.array([[.8, .3],
...                      [.3, .4]])
>>> rng = np.random.RandomState(0)
>>> X = rng.multivariate_normal(mean=[0, 0],
...                                   cov=real_cov,
...                                   size=500)
>>> cov = ShrunkCovariance().fit(X)
>>> cov.covariance_
array([[0.7387..., 0.2536...],
       [0.2536..., 0.4110...]])
>>> cov.location_
array([0.0622..., 0.0193...])
```

Methods

error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits the shrunk covariance model according to the given training data and parameters.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.

Continued on next page

Table 39 – continued from previous page

score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *store_precision=True*, *assume_centered=False*, *shrinkage=0.1*)
Initialize self. See help(type(self)) for accurate signature.

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)
Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like of shape (n_features, n_features)] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

fit(*self*, *X*, *y=None*)
Fits the shrunk covariance model according to the given training data and parameters.

Parameters

X [array-like of shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y not used, present for API consistence purpose.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_precision(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

mahalanobis(*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like of shape (n_samples, n_features)] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

score(*self*, *X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like of shape (n_samples, n_features)] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.covariance.ShrunkCovariance`

- *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*
- *Model selection with Probabilistic PCA and Factor Analysis (FA)*

`covariance.empirical_covariance`(X[, ...])	Computes the Maximum likelihood covariance estimator
`covariance.graphical_lasso`(emp_cov, alpha[, ...])	l1-penalized covariance estimator
`covariance.ledoit_wolf`(X[, assume_centered, ...])	Estimates the shrunk Ledoit-Wolf covariance matrix.
`covariance.oas`(X[, assume_centered])	Estimate covariance with the Oracle Approximating Shrinkage algorithm.

Continued on next page

Table 40 – continued from previous page

covariance.shrunk_covariance(emp_cov[, ...])	Calculates a covariance matrix shrunk on the diagonal

7.5.9 sklearn.covariance.empirical_covariance

sklearn.covariance.**empirical_covariance**(*X*, *assume_centered=False*)

Computes the Maximum likelihood covariance estimator

Parameters

X [ndarray, shape (n_samples, n_features)] Data from which to compute the covariance estimate

assume_centered [boolean] If True, data will not be centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False, data will be centered before computation.

Returns

covariance [2D ndarray, shape (n_features, n_features)] Empirical covariance (Maximum Likelihood Estimator).

Examples using sklearn.covariance.empirical_covariance

- *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*

7.5.10 sklearn.covariance.graphical_lasso

sklearn.covariance.**graphical_lasso**(*emp_cov*, *alpha*, *cov_init=None*, *mode='cd'*, *tol=0.0001*, *enet_tol=0.0001*, *max_iter=100*, *verbose=False*, *return_costs=False*, *eps=2.220446049250313e-16*, *return_n_iter=False*)

l1-penalized covariance estimator

Read more in the *User Guide*.

Parameters

emp_cov [2D ndarray, shape (n_features, n_features)] Empirical covariance from which to compute the covariance estimate.

alpha [positive float] The regularization parameter: the higher alpha, the more regularization, the sparser the inverse covariance.

cov_init [2D array (n_features, n_features), optional] The initial guess for the covariance.

mode [{'cd', 'lars'}] The Lasso solver to use: coordinate descent or LARS. Use LARS for very sparse underlying graphs, where p > n. Elsewhere prefer cd which is more numerically stable.

tol [positive float, optional] The tolerance to declare convergence: if the dual gap goes below this value, iterations are stopped.

enet_tol [positive float, optional] The tolerance for the elastic net solver used to calculate the descent direction. This parameter controls the accuracy of the search direction for a given column update, not of the overall parameter estimate. Only used for mode='cd'.

max_iter [integer, optional] The maximum number of iterations.

verbose [boolean, optional] If verbose is True, the objective function and dual gap are printed at each iteration.

return_costs [boolean, optional] If return_costs is True, the objective function and dual gap at each iteration are returned.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.

return_n_iter [bool, optional] Whether or not to return the number of iterations.

Returns

covariance [2D ndarray, shape (n_features, n_features)] The estimated covariance matrix.

precision [2D ndarray, shape (n_features, n_features)] The estimated (sparse) precision matrix.

costs [list of (objective, dual_gap) pairs] The list of values of the objective function and the dual gap at each iteration. Returned only if return_costs is True.

n_iter [int] Number of iterations. Returned only if `return_n_iter` is set to True.

See also:

GraphicalLasso*, *GraphicalLassoCV

Notes

The algorithm employed to solve this problem is the GLasso algorithm, from the Friedman 2008 Biostatistics paper. It is the same algorithm as in the R `glasso` package.

One possible difference with the `glasso` R package is that the diagonal coefficients are not penalized.

7.5.11 sklearn.covariance.ledoit_wolf

sklearn.covariance.**ledoit_wolf**(*X*, *assume_centered=False*, *block_size=1000*)

Estimates the shrunk Ledoit-Wolf covariance matrix.

Read more in the *User Guide*.

Parameters

X [array-like, shape (n_samples, n_features)] Data from which to compute the covariance estimate

assume_centered [boolean, default=False] If True, data will not be centered before computation. Useful to work with data whose mean is significantly equal to zero but is not exactly zero. If False, data will be centered before computation.

block_size [int, default=1000] Size of the blocks into which the covariance matrix will be split. This is purely a memory optimization and does not affect results.

Returns

shrunk_cov [array-like, shape (n_features, n_features)] Shrunk covariance.

shrinkage [float] Coefficient in the convex combination used for the computation of the shrunk estimate.

Notes

The regularized (shrunk) covariance is:

(1 - shrinkage) * cov + shrinkage * mu * np.identity(n_features)

where mu = trace(cov) / n_features

Examples using `sklearn.covariance.ledoit_wolf`

- *Sparse inverse covariance estimation*

7.5.12 sklearn.covariance.oas

`sklearn.covariance.oas`(*X*, *assume_centered=False*)

Estimate covariance with the Oracle Approximating Shrinkage algorithm.

Parameters

X [array-like, shape (n_samples, n_features)] Data from which to compute the covariance estimate.

assume_centered [boolean] If True, data will not be centered before computation. Useful to work with data whose mean is significantly equal to zero but is not exactly zero. If False, data will be centered before computation.

Returns

shrunk_cov [array-like, shape (n_features, n_features)] Shrunk covariance.

shrinkage [float] Coefficient in the convex combination used for the computation of the shrunk estimate.

Notes

The regularised (shrunk) covariance is:

(1 - shrinkage) * cov + shrinkage * mu * np.identity(n_features)

where mu = trace(cov) / n_features

The formula we used to implement the OAS is slightly modified compared to the one given in the article. See *OAS* for more details.

7.5.13 sklearn.covariance.shrunk_covariance

`sklearn.covariance.shrunk_covariance`(*emp_cov*, *shrinkage=0.1*)

Calculates a covariance matrix shrunk on the diagonal

Read more in the *User Guide*.

Parameters

emp_cov [array-like, shape (n_features, n_features)] Covariance matrix to be shrunk

shrinkage [float, 0 <= shrinkage <= 1] Coefficient in the convex combination used for the computation of the shrunk estimate.

Returns

shrunk_cov [array-like] Shrunk covariance.

Notes

The regularized (shrunk) covariance is given by:

(1 - shrinkage) * cov + shrinkage * mu * np.identity(n_features)

where mu = trace(cov) / n_features

7.6 sklearn.cross_decomposition: Cross decomposition

User guide: See the *Cross decomposition* section for further details.

cross_decomposition.CCA([n_components, ...])	CCA Canonical Correlation Analysis.
cross_decomposition.PLSCanonical([...])	PLSCanonical implements the 2 blocks canonical PLS of the original Wold algorithm [Tenenhaus 1998] p.204, referred as PLS-C2A in [Wegelin 2000].
cross_decomposition.PLSRegression([...])	PLS regression
cross_decomposition.PLSSVD([n_components, ...])	Partial Least Square SVD

7.6.1 sklearn.cross_decomposition.CCA

class sklearn.cross_decomposition.**CCA**(*n_components=2*, *scale=True*, *max_iter=500*, *tol=1e-06*, *copy=True*)

CCA Canonical Correlation Analysis.

CCA inherits from PLS with mode="B" and deflation_mode="canonical".

Read more in the *User Guide*.

Parameters

n_components [int, (default 2).] number of components to keep.

scale [boolean, (default True)] whether to scale the data?

max_iter [an integer, (default 500)] the maximum number of iterations of the NIPALS inner loop

tol [non-negative real, default 1e-06.] the tolerance used in the iterative algorithm

copy [boolean] Whether the deflation be done on a copy. Let the default value to True unless you don't care about side effects

Attributes

x_weights_ [array, [p, n_components]] X block weights vectors.

y_weights_ [array, [q, n_components]] Y block weights vectors.

x_loadings_ [array, [p, n_components]] X block loadings vectors.

y_loadings_ [array, [q, n_components]] Y block loadings vectors.

x_scores_ [array, [n_samples, n_components]] X scores.

y_scores_ [array, [n_samples, n_components]] Y scores.

x_rotations_ [array, [p, n_components]] X block to latents rotations.

y_rotations_ [array, [q, n_components]] Y block to latents rotations.

n_iter_ [array-like] Number of iterations of the NIPALS inner loop for each component.

See also:

PLSCanonical

PLSSVD

Notes

For each component k, find the weights u, v that maximizes max corr(Xk u, Yk v), such that `|u| = |v| = 1`

Note that it maximizes only the correlations between the scores.

The residual matrix of X (Xk+1) block is obtained by the deflation on the current X score: x_score.

The residual matrix of Y (Yk+1) block is obtained by deflation on the current Y score.

References

Jacob A. Wegelin. A survey of Partial Least Squares (PLS) methods, with emphasis on the two-block case. Technical Report 371, Department of Statistics, University of Washington, Seattle, 2000.

In french but still a reference: Tenenhaus, M. (1998). La regression PLS: theorie et pratique. Paris: Editions Technic.

Examples

```
>>> from sklearn.cross_decomposition import CCA
>>> X = [[0., 0., 1.], [1.,0.,0.], [2.,2.,2.], [3.,5.,4.]]
>>> Y = [[0.1, -0.2], [0.9, 1.1], [6.2, 5.9], [11.9, 12.3]]
>>> cca = CCA(n_components=1)
>>> cca.fit(X, Y)
CCA(n_components=1)
>>> X_c, Y_c = cca.transform(X, Y)
```

Methods

fit(self, X, Y)	Fit model to data.
fit_transform(self, X[, y])	Learn and apply the dimension reduction on the train data.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Transform data back to its original space.
predict(self, X[, copy])	Apply the dimension reduction learned on the train data.

Continued on next page

Table 42 – continued from previous page

score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, Y, copy])	Apply the dimension reduction learned on the train data.

__init__ (*self*, *n_components=2*, *scale=True*, *max_iter=500*, *tol=1e-06*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *Y*)
Fit model to data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like of shape (n_samples, n_targets)] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

fit_transform (*self*, *X*, *y=None*)
Learn and apply the dimension reduction on the train data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

y [array-like of shape (n_samples, n_targets)] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

Returns

x_scores if Y is not given, (x_scores, y_scores) otherwise.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform (*self*, *X*)
Transform data back to its original space.

Parameters

X [array-like of shape (n_samples, n_components)] New data, where n_samples is the number of samples and n_components is the number of pls components.

Returns

x_reconstructed [array-like of shape (n_samples, n_features)]

Notes

This transformation will only be exact if n_components=n_features

predict (*self*, *X*, *copy=True*)
Apply the dimension reduction learned on the train data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

copy [boolean, default True] Whether to copy X and Y, or perform in-place normalization.

Notes

This call requires the estimation of a p x q matrix, which may be an issue in high dimensional space.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*, *Y=None*, *copy=True*)

Apply the dimension reduction learned on the train data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like of shape (n_samples, n_targets)] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

copy [boolean, default True] Whether to copy X and Y, or perform in-place normalization.

Returns

x_scores if Y is not given, (x_scores, y_scores) otherwise.

Examples using `sklearn.cross_decomposition.CCA`

- *Multilabel classification*
- *Compare cross decomposition methods*

7.6.2 `sklearn.cross_decomposition`.PLSCanonical

class sklearn.cross_decomposition.**PLSCanonical**(*n_components=2*, *scale=True*, *algorithm='nipals'*, *max_iter=500*, *tol=1e-06*, *copy=True*)

PLSCanonical implements the 2 blocks canonical PLS of the original Wold algorithm [Tenenhaus 1998] p.204, referred as PLS-C2A in [Wegelin 2000].

This class inherits from PLS with mode="A" and deflation_mode="canonical", norm_y_weights=True and algorithm="nipals", but svd should provide similar results up to numerical errors.

Read more in the *User Guide*.

New in version 0.8.

Parameters

n_components [int, (default 2).] Number of components to keep

scale [boolean, (default True)] Option to scale data

algorithm [string, "nipals" or "svd"] The algorithm used to estimate the weights. It will be called n_components times, i.e. once for each iteration of the outer loop.

max_iter [an integer, (default 500)] the maximum number of iterations of the NIPALS inner loop (used only if algorithm="nipals")

tol [non-negative real, default 1e-06] the tolerance used in the iterative algorithm

copy [boolean, default True] Whether the deflation should be done on a copy. Let the default value to True unless you don't care about side effect

Attributes

x_weights_ [array, shape = [p, n_components]] X block weights vectors.

y_weights_ [array, shape = [q, n_components]] Y block weights vectors.

x_loadings_ [array, shape = [p, n_components]] X block loadings vectors.

y_loadings_ [array, shape = [q, n_components]] Y block loadings vectors.

x_scores_ [array, shape = [n_samples, n_components]] X scores.

y_scores_ [array, shape = [n_samples, n_components]] Y scores.

x_rotations_ [array, shape = [p, n_components]] X block to latents rotations.

y_rotations_ [array, shape = [q, n_components]] Y block to latents rotations.

n_iter_ [array-like] Number of iterations of the NIPALS inner loop for each component. Not useful if the algorithm provided is "svd".

See also:

CCA

PLSSVD

Notes

Matrices:

```
T: x_scores_
U: y_scores_
W: x_weights_
C: y_weights_
P: x_loadings_
Q: y_loadings__
```

Are computed such that:

```
X = T P.T + Err and Y = U Q.T + Err
T[:, k] = Xk W[:, k] for k in range(n_components)
U[:, k] = Yk C[:, k] for k in range(n_components)
x_rotations_ = W (P.T W)^(-1)
y_rotations_ = C (Q.T C)^(-1)
```

where Xk and Yk are residual matrices at iteration k.

Slides explaining PLS

For each component k, find weights u, v that optimize:

```
max corr(Xk u, Yk v) * std(Xk u) std(Yk u), such that ``|u| = |v| = 1``
```

Note that it maximizes both the correlations between the scores and the intra-block variances.

The residual matrix of X (Xk+1) block is obtained by the deflation on the current X score: x_score.

The residual matrix of Y (Yk+1) block is obtained by deflation on the current Y score. This performs a canonical symmetric version of the PLS regression. But slightly different than the CCA. This is mostly used for modeling.

This implementation provides the same results that the "plspm" package provided in the R language (R-project), using the function plsca(X, Y). Results are equal or collinear with the function `pls(..., mode = "canonical")` of the "mixOmics" package. The difference relies in the fact that mixOmics implementation does not exactly implement the Wold algorithm since it does not normalize y_weights to one.

References

Jacob A. Wegelin. A survey of Partial Least Squares (PLS) methods, with emphasis on the two-block case. Technical Report 371, Department of Statistics, University of Washington, Seattle, 2000.

Tenenhaus, M. (1998). La regression PLS: theorie et pratique. Paris: Editions Technic.

Examples

```
>>> from sklearn.cross_decomposition import PLSCanonical
>>> X = [[0., 0., 1.], [1.,0.,0.], [2.,2.,2.], [2.,5.,4.]]
>>> Y = [[0.1, -0.2], [0.9, 1.1], [6.2, 5.9], [11.9, 12.3]]
>>> plsca = PLSCanonical(n_components=2)
>>> plsca.fit(X, Y)
PLSCanonical()
>>> X_c, Y_c = plsca.transform(X, Y)
```

Methods

fit(self, X, Y)	Fit model to data.
fit_transform(self, X[, y])	Learn and apply the dimension reduction on the train data.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Transform data back to its original space.
predict(self, X[, copy])	Apply the dimension reduction learned on the train data.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, Y, copy])	Apply the dimension reduction learned on the train data.

__init__(*self*, *n_components=2*, *scale=True*, *algorithm='nipals'*, *max_iter=500*, *tol=1e-06*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *Y*)
Fit model to data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like of shape (n_samples, n_targets)] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

fit_transform(*self*, *X*, *y=None*)
Learn and apply the dimension reduction on the train data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

y [array-like of shape (n_samples, n_targets)] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

Returns

x_scores if Y is not given, (x_scores, y_scores) otherwise.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*)
Transform data back to its original space.

Parameters

X [array-like of shape (n_samples, n_components)] New data, where n_samples is the number of samples and n_components is the number of pls components.

Returns

x_reconstructed [array-like of shape (n_samples, n_features)]

Notes

This transformation will only be exact if n_components=n_features

predict(*self*, *X*, *copy=True*)
Apply the dimension reduction learned on the train data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

copy [boolean, default True] Whether to copy X and Y, or perform in-place normalization.

Notes

This call requires the estimation of a p x q matrix, which may be an issue in high dimensional space.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*, *Y=None*, *copy=True*)

Apply the dimension reduction learned on the train data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like of shape (n_samples, n_targets)] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

copy [boolean, default True] Whether to copy X and Y, or perform in-place normalization.

Returns

x_scores if Y is not given, (x_scores, y_scores) otherwise.

Examples using `sklearn.cross_decomposition.PLSCanonical`

- *Compare cross decomposition methods*

7.6.3 `sklearn.cross_decomposition`.PLSRegression

class sklearn.cross_decomposition.**PLSRegression**(*n_components=2*, *scale=True*, *max_iter=500*, *tol=1e-06*, *copy=True*)

PLS regression

PLSRegression implements the PLS 2 blocks regression known as PLS2 or PLS1 in case of one dimensional response. This class inherits from _PLS with mode="A", deflation_mode="regression", norm_y_weights=False and algorithm="nipals".

Read more in the *User Guide*.

New in version 0.8.

Parameters

n_components [int, (default 2)] Number of components to keep.

scale [boolean, (default True)] whether to scale the data

max_iter [an integer, (default 500)] the maximum number of iterations of the NIPALS inner loop (used only if algorithm="nipals")

tol [non-negative real] Tolerance used in the iterative algorithm default 1e-06.

copy [boolean, default True] Whether the deflation should be done on a copy. Let the default value to True unless you don't care about side effect

Attributes

x_weights_ [array, [p, n_components]] X block weights vectors.

y_weights_ [array, [q, n_components]] Y block weights vectors.

x_loadings_ [array, [p, n_components]] X block loadings vectors.

y_loadings_ [array, [q, n_components]] Y block loadings vectors.

x_scores_ [array, [n_samples, n_components]] X scores.

y_scores_ [array, [n_samples, n_components]] Y scores.

x_rotations_ [array, [p, n_components]] X block to latents rotations.

y_rotations_ [array, [q, n_components]] Y block to latents rotations.

coef_ [array, [p, q]] The coefficients of the linear model: `Y = X coef_ + Err`

n_iter_ [array-like] Number of iterations of the NIPALS inner loop for each component.

Notes

Matrices:

```
T: x_scores_
U: y_scores_
W: x_weights_
C: y_weights_
P: x_loadings_
Q: y_loadings_
```

Are computed such that:

```
X = T P.T + Err and Y = U Q.T + Err
T[:, k] = Xk W[:, k] for k in range(n_components)
U[:, k] = Yk C[:, k] for k in range(n_components)
x_rotations_ = W (P.T W)^(-1)
y_rotations_ = C (Q.T C)^(-1)
```

where Xk and Yk are residual matrices at iteration k.

Slides explaining PLS

For each component k, find weights u, v that optimizes: `max corr(Xk u, Yk v) * std(Xk u) std(Yk u)`, such that `|u| = 1`

Note that it maximizes both the correlations between the scores and the intra-block variances.

The residual matrix of X (Xk+1) block is obtained by the deflation on the current X score: x_score.

The residual matrix of Y (Yk+1) block is obtained by deflation on the current X score. This performs the PLS regression known as PLS2. This mode is prediction oriented.

This implementation provides the same results that 3 PLS packages provided in the R language (R-project):

- "mixOmics" with function pls(X, Y, mode = "regression")
- "plspm " with function plsreg2(X, Y)
- "pls" with function oscorespls.fit(X, Y)

References

Jacob A. Wegelin. A survey of Partial Least Squares (PLS) methods, with emphasis on the two-block case. Technical Report 371, Department of Statistics, University of Washington, Seattle, 2000.

In french but still a reference: Tenenhaus, M. (1998). La regression PLS: theorie et pratique. Paris: Editions Technic.

Examples

```
>>> from sklearn.cross_decomposition import PLSRegression
>>> X = [[0., 0., 1.], [1.,0.,0.], [2.,2.,2.], [2.,5.,4.]]
>>> Y = [[0.1, -0.2], [0.9, 1.1], [6.2, 5.9], [11.9, 12.3]]
>>> pls2 = PLSRegression(n_components=2)
>>> pls2.fit(X, Y)
PLSRegression()
>>> Y_pred = pls2.predict(X)
```

Methods

fit(self, X, Y)	Fit model to data.
fit_transform(self, X[, y])	Learn and apply the dimension reduction on the train data.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Transform data back to its original space.
predict(self, X[, copy])	Apply the dimension reduction learned on the train data.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, Y, copy])	Apply the dimension reduction learned on the train data.

__init__(*self*, *n_components=2*, *scale=True*, *max_iter=500*, *tol=1e-06*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *Y*)
Fit model to data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like of shape (n_samples, n_targets)] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

fit_transform(*self*, *X*, *y=None*)
Learn and apply the dimension reduction on the train data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

y [array-like of shape (n_samples, n_targets)] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

Returns

x_scores if Y is not given, (x_scores, y_scores) otherwise.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*)
Transform data back to its original space.

Parameters

X [array-like of shape (n_samples, n_components)] New data, where n_samples is the number of samples and n_components is the number of pls components.

Returns

x_reconstructed [array-like of shape (n_samples, n_features)]

Notes

This transformation will only be exact if n_components=n_features

predict(*self*, *X*, *copy=True*)
Apply the dimension reduction learned on the train data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

copy [boolean, default True] Whether to copy X and Y, or perform in-place normalization.

Notes

This call requires the estimation of a p x q matrix, which may be an issue in high dimensional space.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*, *Y=None*, *copy=True*)

Apply the dimension reduction learned on the train data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like of shape (n_samples, n_targets)] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

copy [boolean, default True] Whether to copy X and Y, or perform in-place normalization.

Returns

x_scores if Y is not given, (x_scores, y_scores) otherwise.

Examples using `sklearn.cross_decomposition.PLSRegression`

- *Compare cross decomposition methods*

7.6.4 `sklearn.cross_decomposition.PLSSVD`

class `sklearn.cross_decomposition.`**PLSSVD**(*n_components=2*, *scale=True*, *copy=True*)

Partial Least Square SVD

Simply perform a svd on the crosscovariance matrix: X'Y There are no iterative deflation here.

Read more in the *User Guide*.

New in version 0.8.

Parameters

n_components [int, default 2] Number of components to keep.

scale [boolean, default True] Whether to scale X and Y.

copy [boolean, default True] Whether to copy X and Y, or perform in-place computations.

Attributes

x_weights_ [array, [p, n_components]] X block weights vectors.

y_weights_ [array, [q, n_components]] Y block weights vectors.

x_scores_ [array, [n_samples, n_components]] X scores.

y_scores_ [array, [n_samples, n_components]] Y scores.

See also:

PLSCanonical

CCA

Examples

```
>>> import numpy as np
>>> from sklearn.cross_decomposition import PLSSVD
>>> X = np.array([[0., 0., 1.],
...      [1.,0.,0.],
...      [2.,2.,2.],
...      [2.,5.,4.]])
>>> Y = np.array([[0.1, -0.2],
...      [0.9, 1.1],
...      [6.2, 5.9],
...      [11.9, 12.3]])
>>> plsca = PLSSVD(n_components=2)
>>> plsca.fit(X, Y)
PLSSVD()
```

(continues on next page)

(continued from previous page)

```
>>> X_c, Y_c = plsca.transform(X, Y)
>>> X_c.shape, Y_c.shape
((4, 2), (4, 2))
```

Methods

fit(self, X, Y)	Fit model to data.
fit_transform(self, X[, y])	Learn and apply the dimension reduction on the train data.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, Y])	Apply the dimension reduction learned on the train data.

__init__ (*self*, *n_components=2*, *scale=True*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *Y*)
Fit model to data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like of shape (n_samples, n_targets)] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

fit_transform (*self*, *X*, *y=None*)
Learn and apply the dimension reduction on the train data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

y [array-like of shape (n_samples, n_targets)] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

Returns

x_scores if Y is not given, (x_scores, y_scores) otherwise.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*, *Y=None*)

Apply the dimension reduction learned on the train data.

Parameters

X [array-like of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like of shape (n_samples, n_targets)] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

7.7 `sklearn.datasets`: Datasets

The *sklearn.datasets* module includes utilities to load datasets, including methods to load and fetch popular reference datasets. It also features some artificial data generators.

User guide: See the *Dataset loading utilities* section for further details.

7.7.1 Loaders

datasets.clear_data_home([data_home])	Delete all the content of the data home cache.
datasets.dump_svmlight_file(X, y, f[, ...])	Dump the dataset in svmlight / libsvm file format.
datasets.fetch_20newsgroups([data_home, ...])	Load the filenames and data from the 20 newsgroups dataset (classification).
datasets.fetch_20newsgroups_vectorized([...])	Load the 20 newsgroups dataset and vectorize it into token counts (classification).
datasets.fetch_california_housing([...])	Load the California housing dataset (regression).
datasets.fetch_covtype([data_home, ...])	Load the covertype dataset (classification).
datasets.fetch_kddcup99([subset, data_home, ...])	Load the kddcup99 dataset (classification).
datasets.fetch_lfw_pairs([subset, ...])	Load the Labeled Faces in the Wild (LFW) pairs dataset (classification).
datasets.fetch_lfw_people([data_home, ...])	Load the Labeled Faces in the Wild (LFW) people dataset (classification).
datasets.fetch_olivetti_faces([data_home, ...])	Load the Olivetti faces data-set from AT&T (classification).
datasets.fetch_openml([name, version, ...])	Fetch dataset from openml by name or dataset id.
datasets.fetch_rcv1([data_home, subset, ...])	Load the RCV1 multilabel dataset (classification).
datasets.fetch_species_distributions([...])	Loader for species distribution dataset from Phillips et.
datasets.get_data_home([data_home])	Return the path of the scikit-learn data dir.
datasets.load_boston([return_X_y])	Load and return the boston house-prices dataset (regression).

Continued on next page

Table 46 – continued from previous page

datasets.load_breast_cancer([return_X_y])	Load and return the breast cancer wisconsin dataset (classification).
datasets.load_diabetes([return_X_y])	Load and return the diabetes dataset (regression).
datasets.load_digits([n_class, return_X_y])	Load and return the digits dataset (classification).
datasets.load_files(container_path[, ...])	Load text files with categories as subfolder names.
datasets.load_iris([return_X_y])	Load and return the iris dataset (classification).
datasets.load_linnerud([return_X_y])	Load and return the linnerud dataset (multivariate regression).
datasets.load_sample_image(image_name)	Load the numpy array of a single sample image
datasets.load_sample_images()	Load sample images for image manipulation.
datasets.load_svmlight_file(f[, n_features, ...])	Load datasets in the svmlight / libsvm format into sparse CSR matrix
datasets.load_svmlight_files(files[, ...])	Load dataset from multiple files in SVMlight format
datasets.load_wine([return_X_y])	Load and return the wine dataset (classification).

sklearn.datasets.clear_data_home

sklearn.datasets.**clear_data_home**(*data_home=None*)

Delete all the content of the data home cache.

Parameters

data_home [str | None] The path to scikit-learn data dir.

sklearn.datasets.dump_svmlight_file

sklearn.datasets.**dump_svmlight_file**(*X*, *y*, *f*, *zero_based=True*, *comment=None*, *query_id=None*, *multilabel=False*)

Dump the dataset in svmlight / libsvm file format.

This format is a text-based format, with one sample per line. It does not store zero valued features hence is suitable for sparse dataset.

The first element of each line can be used to store a target variable to predict.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [{array-like, sparse matrix}, shape = [n_samples (, n_labels)]] Target values. Class labels must be an integer or float, or array-like objects of integer or float for multilabel classifications.

f [string or file-like in binary mode] If string, specifies the path that will contain the data. If file-like, data will be written to f. f should be opened in binary mode.

zero_based [boolean, optional] Whether column indices should be written zero-based (True) or one-based (False).

comment [string, optional] Comment to insert at the top of the file. This should be either a Unicode string, which will be encoded as UTF-8, or an ASCII byte string. If a comment is given, then it will be preceded by one that identifies the file as having been dumped by scikit-learn. Note that not all tools grok comments in SVMlight files.

query_id [array-like of shape (n_samples,)] Array containing pairwise preference constraints (qid in svmlight format).

multilabel [boolean, optional] Samples may have several labels each (see https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multilabel.html)

New in version 0.17: parameter *multilabel* to support multilabel datasets.

Examples using `sklearn.datasets.dump_svmlight_file`

- *Libsvm GUI*

`sklearn.datasets`.fetch_20newsgroups

`sklearn.datasets.`**`fetch_20newsgroups`**(*data_home=None*, *subset='train'*, *categories=None*, *shuffle=True*, *random_state=42*, *remove=()*, *download_if_missing=True*, *return_X_y=False*)

Load the filenames and data from the 20 newsgroups dataset (classification).

Download it if necessary.

Classes	20
Samples total	18846
Dimensionality	1
Features	text

Read more in the *User Guide*.

Parameters

data_home [optional, default: None] Specify a download and cache folder for the datasets. If None, all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

subset ['train' or 'test', 'all', optional] Select the dataset to load: 'train' for the training set, 'test' for the test set, 'all' for both, with shuffled ordering.

categories [None or collection of string or unicode] If None (default), load all the categories. If not None, list of category names to load (other categories ignored).

shuffle [bool, optional] Whether or not to shuffle the data: might be important for models that make the assumption that the samples are independent and identically distributed (i.i.d.), such as stochastic gradient descent.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset shuffling. Pass an int for reproducible output across multiple function calls. See *Glossary*.

remove [tuple] May contain any subset of ('headers', 'footers', 'quotes'). Each of these are kinds of text that will be detected and removed from the newsgroup posts, preventing classifiers from overfitting on metadata.

'headers' removes newsgroup headers, 'footers' removes blocks at the ends of posts that look like signatures, and 'quotes' removes lines that appear to be quoting another post.

'headers' follows an exact standard; the other filters are not always correct.

download_if_missing [optional, True by default] If False, raise an IOError if the data is not locally available instead of trying to download the data from the source site.

return_X_y [bool, default=False.] If True, returns `(data.data, data.target)` instead of a Bunch object.

New in version 0.22.

Returns

bunch [Bunch object with the following attribute:]

- data: list, length [n_samples]
- target: array, shape [n_samples]
- filenames: list, length [n_samples]
- DESCR: a description of the dataset.
- target_names: a list of categories of the returned data, length [n_classes]. This depends on the `categories` parameter.

(data, target) [tuple if `return_X_y=True`] New in version 0.22.

Examples using `sklearn.datasets.fetch_20newsgroups`

- *Sample pipeline for text feature extraction and evaluation*

`sklearn.datasets`.fetch_20newsgroups_vectorized

`sklearn.datasets.`**`fetch_20newsgroups_vectorized`**(*subset='train'*, *remove=()*, *data_home=None*, *download_if_missing=True*, *return_X_y=False*, *normalize=True*)

Load the 20 newsgroups dataset and vectorize it into token counts (classification).

Download it if necessary.

This is a convenience function; the transformation is done using the default settings for *sklearn.feature_extraction.text.CountVectorizer*. For more advanced usage (stopword filtering, n-gram extraction, etc.), combine fetch_20newsgroups with a custom *sklearn.feature_extraction.text.CountVectorizer*, *sklearn.feature_extraction.text.HashingVectorizer*, *sklearn.feature_extraction.text.TfidfTransformer* or *sklearn.feature_extraction.text.TfidfVectorizer*.

The resulting counts are normalized using *sklearn.preprocessing.normalize* unless normalize is set to False.

Classes	20
Samples total	18846
Dimensionality	130107
Features	real

Read more in the *User Guide*.

Parameters

subset ['train' or 'test', 'all', optional] Select the dataset to load: 'train' for the training set, 'test' for the test set, 'all' for both, with shuffled ordering.

remove [tuple] May contain any subset of ('headers', 'footers', 'quotes'). Each of these are kinds of text that will be detected and removed from the newsgroup posts, preventing classifiers from overfitting on metadata.

'headers' removes newsgroup headers, 'footers' removes blocks at the ends of posts that look like signatures, and 'quotes' removes lines that appear to be quoting another post.

data_home [optional, default: None] Specify an download and cache folder for the datasets. If None, all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

download_if_missing [optional, True by default] If False, raise an IOError if the data is not locally available instead of trying to download the data from the source site.

return_X_y [bool, default=False] If True, returns `(data.data, data.target)` instead of a Bunch object.

New in version 0.20.

normalize [bool, default=True] If True, normalizes each document's feature vector to unit norm using *sklearn.preprocessing.normalize*.

New in version 0.22.

Returns

bunch [Bunch object with the following attribute:]

- bunch.data: sparse matrix, shape [n_samples, n_features]
- bunch.target: array, shape [n_samples]
- bunch.target_names: a list of categories of the returned data, length [n_classes].
- bunch.DESCR: a description of the dataset.

(data, target) [tuple if `return_X_y` is True] New in version 0.20.

Examples using `sklearn.datasets.fetch_20newsgroups_vectorized`

- *The Johnson-Lindenstrauss bound for embedding with random projections*
- *Model Complexity Influence*
- *Multiclass sparse logistic regression on 20newgroups*

`sklearn.datasets`.fetch_california_housing

`sklearn.datasets.`**`fetch_california_housing`**(*data_home=None*, *download_if_missing=True*, *return_X_y=False*)

Load the California housing dataset (regression).

Samples total	20640
Dimensionality	8
Features	real
Target	real 0.15 - 5.

Read more in the *User Guide*.

Parameters

data_home [optional, default: None] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

download_if_missing [optional, default=True] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

return_X_y [boolean, default=False.] If True, returns `(data.data, data.target)` instead of a Bunch object.

New in version 0.20.

Returns

dataset [dict-like object with the following attributes:]

dataset.data [ndarray, shape [20640, 8]] Each row corresponding to the 8 feature values in order.

dataset.target [numpy array of shape (20640,)] Each value corresponds to the average house value in units of 100,000.

dataset.feature_names [array of length 8] Array of ordered feature names used in the dataset.

dataset.DESCR [string] Description of the California housing dataset.

(data, target) [tuple if `return_X_y` is True] New in version 0.20.

Notes

This dataset consists of 20,640 samples and 9 features.

Examples using `sklearn.datasets.fetch_california_housing`

- *Partial Dependence Plots*
- *Imputing missing values with variants of IterativeImputer*
- *Compare the effect of different scalers on data with outliers*

sklearn.datasets.fetch_covtype

sklearn.datasets.**fetch_covtype**(*data_home=None*, *download_if_missing=True*, *random_state=None*, *shuffle=False*, *return_X_y=False*)

Load the covertype dataset (classification).

Download it if necessary.

Classes	7
Samples total	581012
Dimensionality	54
Features	int

Read more in the *User Guide*.

Parameters

data_home [string, optional] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

download_if_missing [boolean, default=True] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset shuffling. Pass an int for reproducible output across multiple function calls. See *Glossary*.

shuffle [bool, default=False] Whether to shuffle dataset.

return_X_y [boolean, default=False.] If True, returns `(data.data, data.target)` instead of a Bunch object.

New in version 0.20.

Returns

dataset [dict-like object with the following attributes:]

dataset.data [numpy array of shape (581012, 54)] Each row corresponds to the 54 features in the dataset.

dataset.target [numpy array of shape (581012,)] Each value corresponds to one of the 7 forest covertypes with values ranging between 1 to 7.

dataset.DESCR [string] Description of the forest covertype dataset.

(data, target) [tuple if `return_X_y` is True] New in version 0.20.

sklearn.datasets.fetch_kddcup99

sklearn.datasets.**fetch_kddcup99**(*subset=None*, *data_home=None*, *shuffle=False*, *random_state=None*, *percent10=True*, *download_if_missing=True*, *return_X_y=False*)

Load the kddcup99 dataset (classification).

Download it if necessary.

Classes	23
Samples total	4898431
Dimensionality	41
Features	discrete (int) or continuous (float)

Read more in the *User Guide*.

New in version 0.18.

Parameters

subset [None, 'SA', 'SF', 'http', 'smtp'] To return the corresponding classical subsets of kddcup 99. If None, return the entire kddcup 99 dataset.

data_home [string, optional] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders. .. versionadded:: 0.19

shuffle [bool, default=False] Whether to shuffle dataset.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset shuffling and for selection of abnormal samples if `subset='SA'`. Pass an int for reproducible output across multiple function calls. See *Glossary*.

percent10 [bool, default=True] Whether to load only 10 percent of the data.

download_if_missing [bool, default=True] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and `target` object.

New in version 0.20.

Returns

data [Bunch]

Dictionary-like object, the interesting attributes are:

- 'data', the data to learn.
- 'target', the regression target for each sample.
- 'DESCR', a description of the dataset.

(data, target) [tuple if `return_X_y` is True] New in version 0.20.

sklearn.datasets.fetch_lfw_pairs

sklearn.datasets.**fetch_lfw_pairs**(*subset='train'*, *data_home=None*, *funneled=True*, *resize=0.5*, *color=False*, *slice_=(slice(70*, *195*, *None)*, *slice(78*, *172*, *None))*, *download_if_missing=True*)

Load the Labeled Faces in the Wild (LFW) pairs dataset (classification).

Download it if necessary.

Classes	5749
Samples total	13233
Dimensionality	5828
Features	real, between 0 and 255

In the official README.txt this task is described as the "Restricted" task. As I am not sure as to implement the "Unrestricted" variant correctly, I left it as unsupported for now.

The original images are 250 x 250 pixels, but the default slice and resize arguments reduce them to 62 x 47.

Read more in the *User Guide*.

Parameters

subset [optional, default: 'train'] Select the dataset to load: 'train' for the development training set, 'test' for the development test set, and '10_folds' for the official evaluation set that is meant to be used with a 10-folds cross validation.

data_home [optional, default: None] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

funneled [boolean, optional, default: True] Download and use the funneled variant of the dataset.

resize [float, optional, default 0.5] Ratio used to resize the each face picture.

color [boolean, optional, default False] Keep the 3 RGB channels instead of averaging them to a single gray level channel. If color is True the shape of the data has one more dimension than the shape with color = False.

slice_ [optional] Provide a custom 2D slice (height, width) to extract the 'interesting' part of the jpeg files and avoid use statistical correlation from the background

download_if_missing [optional, True by default] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

Returns

The data is returned as a Bunch object with the following attributes:

data [numpy array of shape (2200, 5828). Shape depends on `subset`.] Each row corresponds to 2 ravel'd face images of original size 62 x 47 pixels. Changing the `slice_`, `resize` or `subset` parameters will change the shape of the output.

pairs [numpy array of shape (2200, 2, 62, 47). Shape depends on `subset`] Each row has 2 face images corresponding to same or different person from the dataset containing 5749 people. Changing the `slice_`, `resize` or `subset` parameters will change the shape of the output.

target [numpy array of shape (2200,). Shape depends on `subset`.] Labels associated to each pair of images. The two label values being different persons or the same person.

DESCR [string] Description of the Labeled Faces in the Wild (LFW) dataset.

sklearn.datasets.fetch_lfw_people

sklearn.datasets.**fetch_lfw_people**(*data_home=None*, *funneled=True*, *resize=0.5*, *min_faces_per_person=0*, *color=False*, *slice_=(slice(70, 195, None), slice(78, 172, None))*, *download_if_missing=True*, *return_X_y=False*)

Load the Labeled Faces in the Wild (LFW) people dataset (classification).

Download it if necessary.

Classes	5749
Samples total	13233
Dimensionality	5828
Features	real, between 0 and 255

Read more in the *User Guide*.

Parameters

data_home [optional, default: None] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

funneled [boolean, optional, default: True] Download and use the funneled variant of the dataset.

resize [float, optional, default 0.5] Ratio used to resize the each face picture.

min_faces_per_person [int, optional, default None] The extracted dataset will only retain pictures of people that have at least `min_faces_per_person` different pictures.

color [boolean, optional, default False] Keep the 3 RGB channels instead of averaging them to a single gray level channel. If color is True the shape of the data has one more dimension than the shape with color = False.

slice_ [optional] Provide a custom 2D slice (height, width) to extract the 'interesting' part of the jpeg files and avoid use statistical correlation from the background

download_if_missing [optional, True by default] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

return_X_y [boolean, default=False.] If True, returns `(dataset.data, dataset.target)` instead of a Bunch object. See below for more information about the `dataset.data` and `dataset.target` object.

New in version 0.20.

Returns

dataset [dict-like object with the following attributes:]

dataset.data [numpy array of shape (13233, 2914)] Each row corresponds to a ravelled face image of original size 62 x 47 pixels. Changing the `slice_` or resize parameters will change the shape of the output.

dataset.images [numpy array of shape (13233, 62, 47)] Each row is a face image corresponding to one of the 5749 people in the dataset. Changing the `slice_` or resize parameters will change the shape of the output.

dataset.target [numpy array of shape (13233,)] Labels associated to each face image. Those labels range from 0-5748 and correspond to the person IDs.

dataset.DESCR [string] Description of the Labeled Faces in the Wild (LFW) dataset.

(data, target) [tuple if `return_X_y` is True] New in version 0.20.

Examples using `sklearn.datasets.fetch_lfw_people`

- *Faces recognition example using eigenfaces and SVMs*

`sklearn.datasets`.fetch_olivetti_faces

`sklearn.datasets.`**`fetch_olivetti_faces`**(*data_home=None*, *shuffle=False*, *random_state=0*, *download_if_missing=True*, *return_X_y=False*)

Load the Olivetti faces data-set from AT&T (classification).

Download it if necessary.

Classes	40
Samples total	400
Dimensionality	4096
Features	real, between 0 and 1

Read more in the *User Guide*.

Parameters

data_home [optional, default: None] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

shuffle [boolean, optional] If True the order of the dataset is shuffled to avoid having images of the same person grouped.

random_state [int, RandomState instance or None (default=0)] Determines random number generation for dataset shuffling. Pass an int for reproducible output across multiple function calls. See *Glossary*.

download_if_missing [optional, True by default] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a `Bunch` object. See below for more information about the `data` and `target` object.

New in version 0.22.

Returns

bunch [Bunch object with the following attributes:]

- data: ndarray, shape (400, 4096). Each row corresponds to a ravelled face image of original size 64 x 64 pixels.
- images : ndarray, shape (400, 64, 64). Each row is a face image corresponding to one of the 40 subjects of the dataset.
- target : ndarray, shape (400,). Labels associated to each face image. Those labels are ranging from 0-39 and correspond to the Subject IDs.
- DESCR : string. Description of the modified Olivetti Faces Dataset.

(data, target) [tuple if `return_X_y=True`] New in version 0.22.

Examples using `sklearn.datasets.fetch_olivetti_faces`

- *Face completion with a multi-output estimators*
- *Online learning of a dictionary of parts of faces*
- *Faces dataset decompositions*
- *Pixel importances with a parallel forest of trees*

`sklearn.datasets.fetch_openml`

`sklearn.datasets.`**`fetch_openml`**(*name=None*, *version='active'*, *data_id=None*, *data_home=None*, *target_column='default-target'*, *cache=True*, *return_X_y=False*, *as_frame=False*)

Fetch dataset from openml by name or dataset id.

Datasets are uniquely identified by either an integer ID or by a combination of name and version (i.e. there might be multiple versions of the 'iris' dataset). Please give either name or data_id (not both). In case a name is given, a version can also be provided.

Read more in the *User Guide*.

Note: EXPERIMENTAL

The API is experimental (particularly the return value structure), and might have small backward-incompatible changes in future releases.

Parameters

name [str or None] String identifier of the dataset. Note that OpenML can have multiple datasets with the same name.

version [integer or 'active', default='active'] Version of the dataset. Can only be provided if also `name` is given. If 'active' the oldest version that's still active is used. Since there may be more than one active version of a dataset, and those versions may fundamentally be different from one another, setting an exact version is highly recommended.

data_id [int or None] OpenML ID of the dataset. The most specific way of retrieving a dataset. If data_id is not given, name (and potential version) are used to obtain a dataset.

data_home [string or None, default None] Specify another download and cache folder for the data sets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

target_column [string, list or None, default 'default-target'] Specify the column name in the data to use as target. If 'default-target', the standard target column a stored on the server is used. If `None`, all columns are returned as data and the target is `None`. If list (of strings), all columns with these names are returned as multi-target (Note: not all scikit-learn classifiers can handle all types of multi-output combinations)

cache [boolean, default=True] Whether to cache downloaded datasets using joblib.

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and `target` objects.

as_frame [boolean, default=False] If True, the data is a pandas DataFrame including columns with appropriate dtypes (numeric, string or categorical). The target is a pandas DataFrame or Series depending on the number of target_columns. The Bunch will contain a `frame` attribute with the target and the data. If `return_X_y` is True, then `(data, target)` will be pandas DataFrames or Series as describe above.

Returns

data [Bunch] Dictionary-like object, with attributes:

data [np.array, scipy.sparse.csr_matrix of floats, or pandas DataFrame] The feature matrix. Categorical features are encoded as ordinals.

target [np.array, pandas Series or DataFrame] The regression target or classification labels, if applicable. Dtype is float if numeric, and object if categorical. If `as_frame` is True, `target` is a pandas object.

DESCR [str] The full description of the dataset

feature_names [list] The names of the dataset columns

target_names: list The names of the target columns

New in version 0.22.

categories [dict or None] Maps each categorical feature name to a list of values, such that the value encoded as i is ith in the list. If `as_frame` is True, this is None.

details [dict] More metadata from OpenML

frame [pandas DataFrame] Only present when `as_frame=True`. DataFrame with `data` and `target`.

(data, target) [tuple if `return_X_y` is True]

Note: EXPERIMENTAL

This interface is **experimental** and subsequent releases may change attributes without notice (although there should only be minor changes to `data` and `target`).

Missing values in the 'data' are represented as NaN's. Missing values in 'target' are represented as NaN's (numerical target) or None (categorical target)

Examples using `sklearn.datasets.fetch_openml`

- *Approximate nearest neighbors in TSNE*

sklearn.datasets.fetch_rcv1

sklearn.datasets.**fetch_rcv1**(*data_home=None*, *subset='all'*, *download_if_missing=True*, *random_state=None*, *shuffle=False*, *return_X_y=False*)

Load the RCV1 multilabel dataset (classification).

Download it if necessary.

Version: RCV1-v2, vectors, full sets, topics multilabels.

Classes	103
Samples total	804414
Dimensionality	47236
Features	real, between 0 and 1

Read more in the *User Guide*.

New in version 0.17.

Parameters

data_home [string, optional] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

subset [string, 'train', 'test', or 'all', default='all'] Select the dataset to load: 'train' for the training set (23149 samples), 'test' for the test set (781265 samples), 'all' for both, with the training samples first if shuffle is False. This follows the official LYRL2004 chronological split.

download_if_missing [boolean, default=True] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset shuffling. Pass an int for reproducible output across multiple function calls. See *Glossary*.

shuffle [bool, default=False] Whether to shuffle dataset.

return_X_y [boolean, default=False.] If True, returns `(dataset.data, dataset.target)` instead of a Bunch object. See below for more information about the `dataset.data` and `dataset.target` object.

New in version 0.20.

Returns

dataset [dict-like object with the following attributes:]

dataset.data [scipy csr array, dtype np.float64, shape (804414, 47236)] The array has 0.16% of non zero values.

dataset.target [scipy csr array, dtype np.uint8, shape (804414, 103)] Each sample has a value of 1 in its categories, and 0 in others. The array has 3.15% of non zero values.

dataset.sample_id [numpy array, dtype np.uint32, shape (804414,)] Identification number of each sample, as ordered in dataset.data.

dataset.target_names [numpy array, dtype object, length (103)] Names of each target (RCV1 topics), as ordered in dataset.target.

dataset.DESCR [string] Description of the RCV1 dataset.

(data, target) [tuple if `return_X_y` is True] New in version 0.20.

sklearn.datasets.fetch_species_distributions

sklearn.datasets.**fetch_species_distributions**(*data_home=None*, *download_if_missing=True*)

Loader for species distribution dataset from Phillips et. al. (2006)

Read more in the *User Guide*.

Parameters

data_home [optional, default: None] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

download_if_missing [optional, True by default] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

Returns

The data is returned as a Bunch object with the following attributes:

coverages [array, shape = [14, 1592, 1212]] These represent the 14 features measured at each point of the map grid. The latitude/longitude values for the grid are discussed below. Missing data is represented by the value -9999.

train [record array, shape = (1624,)] The training points for the data. Each point has three fields:

- train['species'] is the species name
- train['dd long'] is the longitude, in degrees
- train['dd lat'] is the latitude, in degrees

test [record array, shape = (620,)] The test points for the data. Same format as the training data.

Nx, Ny [integers] The number of longitudes (x) and latitudes (y) in the grid

x_left_lower_corner, y_left_lower_corner [floats] The (x,y) position of the lower-left corner, in degrees

grid_size [float] The spacing between points of the grid, in degrees

Notes

This dataset represents the geographic distribution of species. The dataset is provided by Phillips et. al. (2006).

The two species are:

- "Bradypus variegatus" , the Brown-throated Sloth.
- "Microryzomys minutus" , also known as the Forest Small Rice Rat, a rodent that lives in Peru, Colombia, Ecuador, Peru, and Venezuela.
- For an example of using this dataset with scikit-learn, see *examples/applications/plot_species_distribution_modeling.py*.

References

- "Maximum entropy modeling of species geographic distributions" S. J. Phillips, R. P. Anderson, R. E. Schapire - Ecological Modelling, 190:231-259, 2006.

Examples using `sklearn.datasets.fetch_species_distributions`

- *Species distribution modeling*
- *Kernel Density Estimate of Species Distributions*

sklearn.datasets.get_data_home

`sklearn.datasets.get_data_home(data_home=None)`
Return the path of the scikit-learn data dir.

This folder is used by some large dataset loaders to avoid downloading the data several times.

By default the data dir is set to a folder named 'scikit_learn_data' in the user home folder.

Alternatively, it can be set by the 'SCIKIT_LEARN_DATA' environment variable or programmatically by giving an explicit folder path. The '~' symbol is expanded to the user home folder.

If the folder does not already exist, it is automatically created.

Parameters

data_home [str | None] The path to scikit-learn data dir.

Examples using `sklearn.datasets.get_data_home`

- *Out-of-core classification of text documents*

sklearn.datasets.load_boston

`sklearn.datasets.load_boston(return_X_y=False)`
Load and return the boston house-prices dataset (regression).

Samples total	506
Dimensionality	13
Features	real, positive
Targets	real 5. - 50.

Read more in the *User Guide*.

Parameters

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and `target` object.

New in version 0.18.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the regression targets, 'DESCR', the full description of the dataset, and 'filename', the physical location of boston csv dataset (added in version `0.20`).

(data, target) [tuple if `return_X_y` is True] New in version 0.18.

Notes

Changed in version 0.20: Fixed a wrong data point at [445, 0].

Examples

```
>>> from sklearn.datasets import load_boston
>>> X, y = load_boston(return_X_y=True)
>>> print(X.shape)
(506, 13)
```

Examples using `sklearn.datasets.load_boston`

- *Advanced Plotting With Partial Dependence*
- *Plot individual and voting regression predictions*
- *Gradient Boosting regression*
- *Combine predictors using stacking*
- *Outlier detection on a real data set*
- *Model Complexity Influence*
- *Feature selection using SelectFromModel and LassoCV*
- *Imputing missing values before building an estimator*
- *Plotting Cross-Validated Predictions*
- *Effect of transforming the targets in regression model*

`sklearn.datasets`.load_breast_cancer

`sklearn.datasets.`**`load_breast_cancer`**(*return_X_y=False*)

Load and return the breast cancer wisconsin dataset (classification).

The breast cancer dataset is a classic and very easy binary classification dataset.

Classes	2
Samples per class	212(M),357(B)
Samples total	569
Dimensionality	30
Features	real, positive

Read more in the *User Guide*.

Parameters

return_X_y [boolean, default=False] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and `target` object.

New in version 0.18.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, and 'DESCR', the full description of the dataset, 'filename', the physical location of breast cancer csv dataset (added in version `0.20`).

(data, target) [tuple if `return_X_y` is True] New in version 0.18.

The copy of UCI ML Breast Cancer Wisconsin (Diagnostic) dataset is

downloaded from:

https://goo.gl/U2Uwz2

Examples

Let's say you are interested in the samples 10, 50, and 85, and want to know their class name.

```
>>> from sklearn.datasets import load_breast_cancer
>>> data = load_breast_cancer()
>>> data.target[[10, 50, 85]]
array([0, 1, 0])
>>> list(data.target_names)
['malignant', 'benign']
```

Examples using `sklearn.datasets.load_breast_cancer`

- *Post pruning decision trees with cost complexity pruning*
- *Permutation Importance with Multicollinear or Correlated Features*

`sklearn.datasets.load_diabetes`

`sklearn.datasets.load_diabetes`(*return_X_y=False*)

Load and return the diabetes dataset (regression).

Samples total	442
Dimensionality	10
Features	real, -.2 < x < .2
Targets	integer 25 - 346

Read more in the *User Guide*.

Parameters

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and `target` object.

New in version 0.18.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the regression target for each sample, 'data_filename', the physical location of diabetes data csv dataset, and 'target_filename', the physical location of diabetes targets csv datataset (added in version `0.20`).

(data, target) [tuple if `return_X_y` is True] New in version 0.18.

Examples using `sklearn.datasets.load_diabetes`

- *Lasso path using LARS*
- *Linear Regression Example*
- *Sparsity Example: Fitting only features 1 and 2*
- *Lasso and Elastic Net*
- *Lasso model selection: Cross-Validation / AIC / BIC*
- *Imputing missing values before building an estimator*
- *Cross-validation on diabetes Dataset Exercise*

`sklearn.datasets`.load_digits

`sklearn.datasets.`**`load_digits`**(*n_class=10*, *return_X_y=False*)

Load and return the digits dataset (classification).

Each datapoint is a 8x8 image of a digit.

Classes	10
Samples per class	~180
Samples total	1797
Dimensionality	64
Features	integers 0-16

Read more in the *User Guide*.

Parameters

n_class [integer, between 0 and 10, optional (default=10)] The number of classes to return.

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and `target` object.

New in version 0.18.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'images', the images corresponding to each sample, 'target', the classification labels for each sample, 'target_names', the meaning of the labels, and 'DESCR', the full description of the dataset.

(data, target) [tuple if `return_X_y` is True] New in version 0.18.

This is a copy of the test set of the UCI ML hand-written digits datasets

https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

Examples

To load the data and visualize the images:

```
>>> from sklearn.datasets import load_digits
>>> digits = load_digits()
>>> print(digits.data.shape)
(1797, 64)
>>> import matplotlib.pyplot as plt
>>> plt.gray()
>>> plt.matshow(digits.images[0])
>>> plt.show()
```

Examples using `sklearn.datasets.load_digits`

- *The Johnson-Lindenstrauss bound for embedding with random projections*
- *Explicit feature map approximation for RBF kernels*
- *Recognizing hand-written digits*
- *Feature agglomeration*
- *Various Agglomerative Clustering on a 2D embedding of digits*
- *A demo of K-Means clustering on the handwritten digits data*
- *The Digit Dataset*
- *Early stopping of Gradient Boosting*
- *Recursive feature elimination*
- *Comparing various online solvers*
- *L1 Penalty and Sparsity in Logistic Regression*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*
- *Plotting Validation Curves*
- *Parameter estimation using grid search with cross-validation*
- *Comparing randomized search and grid search for hyperparameter estimation*
- *Balance model complexity and cross-validated score*
- *Plotting Learning Curves*
- *Kernel Density Estimation*
- *Caching nearest neighbors*
- *Dimensionality Reduction with Neighborhood Components Analysis*
- *Restricted Boltzmann Machine features for digit classification*
- *Compare Stochastic learning strategies for MLPClassifier*
- *Pipelining: chaining a PCA and a logistic regression*
- *Selecting dimensionality reduction with Pipeline and GridSearchCV*
- *Label Propagation digits: Demonstrating performance*
- *Label Propagation digits active learning*

- *Digits Classification Exercise*
- *Cross-validation on Digits Dataset Exercise*

sklearn.datasets.load_files

sklearn.datasets.**load_files**(*container_path*, *description=None*, *categories=None*, *load_content=True*, *shuffle=True*, *encoding=None*, *decode_error='strict'*, *random_state=0*)

Load text files with categories as subfolder names.

Individual samples are assumed to be files stored a two levels folder structure such as the following:

container_folder/

category_1_folder/ file_1.txt file_2.txt ... file_42.txt

category_2_folder/ file_43.txt file_44.txt ...

The folder names are used as supervised signal label names. The individual file names are not important.

This function does not try to extract features into a numpy array or scipy sparse matrix. In addition, if load_content is false it does not try to load the files in memory.

To use text files in a scikit-learn classification or clustering algorithm, you will need to use the :mod‘~sklearn.feature_extraction.text‘ module to build a feature extraction transformer that suits your problem.

If you set load_content=True, you should also specify the encoding of the text using the ‘encoding’ parameter. For many modern text files, ‘utf-8’ will be the correct encoding. If you leave encoding equal to None, then the content will be made of bytes instead of Unicode, and you will not be able to use most functions in `text`.

Similar feature extractors should be built for other kind of unstructured data input such as images, audio, video, ...

Read more in the *User Guide*.

Parameters

container_path [string or unicode] Path to the main folder holding one subfolder per category

description [string or unicode, optional (default=None)] A paragraph describing the characteristic of the dataset: its source, reference, etc.

categories [A collection of strings or None, optional (default=None)] If None (default), load all the categories. If not None, list of category names to load (other categories ignored).

load_content [boolean, optional (default=True)] Whether to load or not the content of the different files. If true a ‘data’ attribute containing the text information is present in the data structure returned. If not, a filenames attribute gives the path to the files.

shuffle [bool, optional (default=True)] Whether or not to shuffle the data: might be important for models that make the assumption that the samples are independent and identically distributed (i.i.d.), such as stochastic gradient descent.

encoding [string or None (default is None)] If None, do not try to decode the content of the files (e.g. for images or other non-text content). If not None, encoding to use to decode text files to Unicode if load_content is True.

decode_error [{‘strict’, ‘ignore’, ‘replace’}, optional] Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given `encoding`. Passed as keyword argument ‘errors’ to bytes.decode.

random_state [int, RandomState instance or None (default=0)] Determines random number generation for dataset shuffling. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: either data, the raw text data to learn, or 'filenames', the files holding it, 'target', the classification labels (integer index), 'target_names', the meaning of the labels, and 'DESCR', the full description of the dataset.

sklearn.datasets.load_iris

sklearn.datasets.**load_iris**(*return_X_y=False*)

Load and return the iris dataset (classification).

The iris dataset is a classic and very easy multi-class classification dataset.

Classes	3
Samples per class	50
Samples total	150
Dimensionality	4
Features	real, positive

Read more in the *User Guide*.

Parameters

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and `target` object.

New in version 0.18.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, 'DESCR', the full description of the dataset, 'filename', the physical location of iris csv dataset (added in version `0.20`).

(data, target) [tuple if `return_X_y` is True] New in version 0.18.

Notes

Changed in version 0.20: Fixed two wrong data points according to Fisher's paper. The new version is the same as in R, but not as in the UCI Machine Learning Repository.

Examples

Let's say you are interested in the samples 10, 25, and 50, and want to know their class name.

```
>>> from sklearn.datasets import load_iris
>>> data = load_iris()
>>> data.target[[10, 25, 50]]
array([0, 0, 1])
>>> list(data.target_names)
['setosa', 'versicolor', 'virginica']
```

Examples using `sklearn.datasets.load_iris`

- *Plot classification probability*
- *Plot Hierarchical Clustering Dendrogram*
- *K-means Clustering*
- *The Iris Dataset*
- *Plot the decision surface of a decision tree on the iris dataset*
- *Understanding the decision tree structure*
- *PCA example with Iris Data-set*
- *Incremental PCA*
- *Comparison of LDA and PCA 2D projection of Iris dataset*
- *Plot the decision boundaries of a VotingClassifier*
- *Early stopping of Gradient Boosting*
- *Plot the decision surfaces of ensembles of trees on the iris dataset*
- *Test with permutations the significance of a classification score*
- *Univariate Feature Selection*
- *GMM covariances*
- *Gaussian process classification (GPC) on iris dataset*
- *Regularization path of L1- Logistic Regression*
- *Logistic Regression 3-class Classifier*
- *Plot multi-class SGD on the iris dataset*
- *Confusion matrix*
- *Receiver Operating Characteristic (ROC) with cross validation*
- *Nested versus non-nested cross-validation*
- *Receiver Operating Characteristic (ROC)*
- *Precision-Recall*
- *Nearest Neighbors Classification*
- *Nearest Centroid Classification*
- *Comparing Nearest Neighbors with and without Neighborhood Components Analysis*
- *Compare Stochastic learning strategies for MLPClassifier*
- *Concatenating multiple feature extraction methods*
- *Release Highlights for scikit-learn 0.22*
- *Decision boundary of label propagation versus SVM on the Iris dataset*
- *SVM with custom kernel*
- *SVM-Anova: SVM with univariate feature selection*
- *Plot different SVM classifiers in the iris dataset*
- *RBF SVM parameters*

- *SVM Exercise*

sklearn.datasets.load_linnerud

sklearn.datasets.**load_linnerud**(*return_X_y=False*)
Load and return the linnerud dataset (multivariate regression).

Samples total	20
Dimensionality	3 (for both data and target)
Features	integer
Targets	integer

Read more in the *User Guide*.

Parameters

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and `target` object.

New in version 0.18.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data' and 'target', the two multivariate datasets, with 'data' corresponding to the exercise and 'target' corresponding to the physiological measurements, as well as 'feature_names' and 'target_names'. In addition, you will also have access to 'data_filename', the physical location of linnerud data csv dataset, and 'target_filename', the physical location of linnerud targets csv datataset (added in version `0.20`).

(data, target) [tuple if `return_X_y` is True] New in version 0.18.

sklearn.datasets.load_sample_image

sklearn.datasets.**load_sample_image**(*image_name*)
Load the numpy array of a single sample image

Read more in the *User Guide*.

Parameters

image_name [{`china.jpg`, `flower.jpg`}] The name of the sample image loaded

Returns

img [3D array] The image as a numpy array: height x width x color

Examples

```
>>> from sklearn.datasets import load_sample_image
>>> china = load_sample_image('china.jpg')   # doctest: +SKIP
>>> china.dtype                              # doctest: +SKIP
dtype('uint8')
>>> china.shape                              # doctest: +SKIP
(427, 640, 3)
>>> flower = load_sample_image('flower.jpg') # doctest: +SKIP
```

(continues on next page)

(continued from previous page)

```
>>> flower.dtype                                  # doctest: +SKIP
dtype('uint8')
>>> flower.shape                                  # doctest: +SKIP
(427, 640, 3)
```

Examples using `sklearn.datasets.load_sample_image`

- *Color Quantization using K-Means*

`sklearn.datasets.load_sample_images`

`sklearn.datasets.load_sample_images()`

Load sample images for image manipulation.

Loads both, `china` and `flower`.

Read more in the *User Guide*.

Returns

data [Bunch] Dictionary-like object with the following attributes : 'images', the two sample images, 'filenames', the file names for the images, and 'DESCR' the full description of the dataset.

Examples

To load the data and visualize the images:

```
>>> from sklearn.datasets import load_sample_images
>>> dataset = load_sample_images()     #doctest: +SKIP
>>> len(dataset.images)                #doctest: +SKIP
2
>>> first_img_data = dataset.images[0] #doctest: +SKIP
>>> first_img_data.shape               #doctest: +SKIP
(427, 640, 3)
>>> first_img_data.dtype               #doctest: +SKIP
dtype('uint8')
```

`sklearn.datasets.load_svmlight_file`

`sklearn.datasets.load_svmlight_file(f, n_features=None, dtype=<class 'numpy.float64'>, multilabel=False, zero_based='auto', query_id=False, offset=0, length=-1)`

Load datasets in the svmlight / libsvm format into sparse CSR matrix

This format is a text-based format, with one sample per line. It does not store zero valued features hence is suitable for sparse dataset.

The first element of each line can be used to store a target variable to predict.

This format is used as the default format for both svmlight and the libsvm command line programs.

Parsing a text based source can be expensive. When working on repeatedly on the same dataset, it is recommended to wrap this loader with joblib.Memory.cache to store a memmapped backup of the CSR results of the first call and benefit from the near instantaneous loading of memmapped structures for the subsequent calls.

In case the file contains a pairwise preference constraint (known as "qid" in the svmlight format) these are ignored unless the query_id parameter is set to True. These pairwise preference constraints can be used to constraint the combination of samples when using pairwise loss functions (as is the case in some learning to rank problems) so that only pairs with the same query_id value are considered.

This implementation is written in Cython and is reasonably fast. However, a faster API-compatible loader is also available at:

https://github.com/mblondel/svmlight-loader

Parameters

f [{str, file-like, int}] (Path to) a file to load. If a path ends in ".gz" or ".bz2", it will be uncompressed on the fly. If an integer is passed, it is assumed to be a file descriptor. A file-like or file descriptor will not be closed by this function. A file-like object must be opened in binary mode.

n_features [int or None] The number of features to use. If None, it will be inferred. This argument is useful to load several files that are subsets of a bigger sliced dataset: each subset might not have examples of every feature, hence the inferred shape might vary from one slice to another. n_features is only required if `offset` or `length` are passed a non-default value.

dtype [numpy data type, default np.float64] Data type of dataset to be loaded. This will be the data type of the output numpy arrays `X` and `y`.

multilabel [boolean, optional, default False] Samples may have several labels each (see https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multilabel.html)

zero_based [boolean or "auto", optional, default "auto"] Whether column indices in f are zero-based (True) or one-based (False). If column indices are one-based, they are transformed to zero-based to match Python/NumPy conventions. If set to "auto", a heuristic check is applied to determine this from the file contents. Both kinds of files occur "in the wild", but they are unfortunately not self-identifying. Using "auto" or True should always be safe when no `offset` or `length` is passed. If `offset` or `length` are passed, the "auto" mode falls back to `zero_based=True` to avoid having the heuristic check yield inconsistent results on different segments of the file.

query_id [boolean, default False] If True, will return the query_id array for each file.

offset [integer, optional, default 0] Ignore the offset first bytes by seeking forward, then discarding the following bytes up until the next new line character.

length [integer, optional, default -1] If strictly positive, stop reading any new line of data once the position in the file has reached the (offset + length) bytes threshold.

Returns

X [scipy.sparse matrix of shape (n_samples, n_features)]

y [ndarray of shape (n_samples,), or, in the multilabel a list of] tuples of length n_samples.

query_id [array of shape (n_samples,)] query_id for each sample. Only returned when query_id is set to True.

See also:

load_svmlight_files similar function for loading multiple files in this format, enforcing the same number of features/columns on all of them.

Examples

To use joblib.Memory to cache the svmlight file:

```
from joblib import Memory
from .datasets import load_svmlight_file
mem = Memory("./mycache")

@mem.cache
def get_data():
    data = load_svmlight_file("mysvmlightfile")
    return data[0], data[1]

X, y = get_data()
```

sklearn.datasets.load_svmlight_files

sklearn.datasets.**load_svmlight_files**(*files*, *n_features=None*, *dtype=<class 'numpy.float64'>*, *multilabel=False*, *zero_based='auto'*, *query_id=False*, *offset=0*, *length=-1*)

Load dataset from multiple files in SVMlight format

This function is equivalent to mapping load_svmlight_file over a list of files, except that the results are concatenated into a single, flat list and the samples vectors are constrained to all have the same number of features.

In case the file contains a pairwise preference constraint (known as "qid" in the svmlight format) these are ignored unless the query_id parameter is set to True. These pairwise preference constraints can be used to constraint the combination of samples when using pairwise loss functions (as is the case in some learning to rank problems) so that only pairs with the same query_id value are considered.

Parameters

files [iterable over {str, file-like, int}] (Paths of) files to load. If a path ends in ".gz" or ".bz2", it will be uncompressed on the fly. If an integer is passed, it is assumed to be a file descriptor. File-likes and file descriptors will not be closed by this function. File-like objects must be opened in binary mode.

n_features [int or None] The number of features to use. If None, it will be inferred from the maximum column index occurring in any of the files.

This can be set to a higher value than the actual number of features in any of the input files, but setting it to a lower value will cause an exception to be raised.

dtype [numpy data type, default np.float64] Data type of dataset to be loaded. This will be the data type of the output numpy arrays `X` and `y`.

multilabel [boolean, optional] Samples may have several labels each (see https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multilabel.html)

zero_based [boolean or "auto", optional] Whether column indices in f are zero-based (True) or one-based (False). If column indices are one-based, they are transformed to zero-based to match Python/NumPy conventions. If set to "auto", a heuristic check is applied to determine this from the file contents. Both kinds of files occur "in the wild", but they are unfortunately not self-identifying. Using "auto" or True should always be safe when no offset or length

is passed. If offset or length are passed, the "auto" mode falls back to zero_based=True to avoid having the heuristic check yield inconsistent results on different segments of the file.

query_id [boolean, defaults to False] If True, will return the query_id array for each file.

offset [integer, optional, default 0] Ignore the offset first bytes by seeking forward, then discarding the following bytes up until the next new line character.

length [integer, optional, default -1] If strictly positive, stop reading any new line of data once the position in the file has reached the (offset + length) bytes threshold.

Returns

[X1, y1, . . . , Xn, yn]

where each (Xi, yi) pair is the result from load_svmlight_file(files[i]).

If query_id is set to True, this will return instead [X1, y1, q1,

. . . , Xn, yn, qn] where (Xi, yi, qi) is the result from

load_svmlight_file(files[i])

See also:

load_svmlight_file

Notes

When fitting a model to a matrix X_train and evaluating it against a matrix X_test, it is essential that X_train and X_test have the same number of features (X_train.shape[1] == X_test.shape[1]). This may not be the case if you load the files individually with load_svmlight_file.

sklearn.datasets.load_wine

sklearn.datasets.**load_wine**(*return_X_y=False*)

Load and return the wine dataset (classification).

New in version 0.18.

The wine dataset is a classic and very easy multi-class classification dataset.

Classes	3
Samples per class	[59,71,48]
Samples total	178
Dimensionality	13
Features	real, positive

Read more in the *User Guide*.

Parameters

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and `target` object.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, and 'DESCR', the full description of the dataset.

(data, target) [tuple if `return_X_y` is True]

The copy of UCI ML Wine Data Set dataset is downloaded and modified to fit standard format from:

https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data

Examples

Let's say you are interested in the samples 10, 80, and 140, and want to know their class name.

```
>>> from sklearn.datasets import load_wine
>>> data = load_wine()
>>> data.target[[10, 80, 140]]
array([0, 1, 2])
>>> list(data.target_names)
['class_0', 'class_1', 'class_2']
```

Examples using `sklearn.datasets.load_wine`

- *ROC Curve with Visualization API*
- *Importance of Feature Scaling*

7.7.2 Samples generator

datasets.make_biclusters(shape, n_clusters)	Generate an array with constant block diagonal structure for biclustering.
datasets.make_blobs([n_samples, n_features, ...])	Generate isotropic Gaussian blobs for clustering.
datasets.make_checkerboard(shape, n_clusters)	Generate an array with block checkerboard structure for biclustering.
datasets.make_circles([n_samples, shuffle, ...])	Make a large circle containing a smaller circle in 2d.
datasets.make_classification([n_samples, ...])	Generate a random n-class classification problem.
datasets.make_friedman1([n_samples, ...])	Generate the "Friedman #1" regression problem
datasets.make_friedman2([n_samples, noise, ...])	Generate the "Friedman #2" regression problem
datasets.make_friedman3([n_samples, noise, ...])	Generate the "Friedman #3" regression problem
datasets.make_gaussian_quantiles([mean, ...])	Generate isotropic Gaussian and label samples by quantile
datasets.make_hastie_10_2([n_samples, ...])	Generates data for binary classification used in Hastie et al.
datasets.make_low_rank_matrix([n_samples, ...])	Generate a mostly low rank matrix with bell-shaped singular values
datasets.make_moons([n_samples, shuffle, ...])	Make two interleaving half circles
datasets.make_multilabel_classification([...])	Generate a random multilabel classification problem.
datasets.make_regression([n_samples, ...])	Generate a random regression problem.

Continued on next page

Table 47 – continued from previous page

datasets.make_s_curve([n_samples, noise, ...])	Generate an S curve dataset.
datasets.make_sparse_coded_signal(n_samples, ...)	Generate a signal as a sparse combination of dictionary elements.
datasets.make_sparse_spd_matrix([dim, ...])	Generate a sparse symmetric definite positive matrix.
datasets.make_sparse_uncorrelated([...])	Generate a random regression problem with sparse uncorrelated design
datasets.make_spd_matrix(n_dim[, random_state])	Generate a random symmetric, positive-definite matrix.
datasets.make_swiss_roll([n_samples, noise, ...])	Generate a swiss roll dataset.

sklearn.datasets.make_biclusters

sklearn.datasets.**make_biclusters**(*shape*, *n_clusters*, *noise=0.0*, *minval=10*, *maxval=100*, *shuffle=True*, *random_state=None*)

Generate an array with constant block diagonal structure for biclustering.

Read more in the *User Guide*.

Parameters

shape [iterable (n_rows, n_cols)] The shape of the result.

n_clusters [integer] The number of biclusters.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise.

minval [int, optional (default=10)] Minimum value of a bicluster.

maxval [int, optional (default=100)] Maximum value of a bicluster.

shuffle [boolean, optional (default=True)] Shuffle the samples.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape `shape`] The generated array.

rows [array of shape (n_clusters, X.shape[0],)] The indicators for cluster membership of each row.

cols [array of shape (n_clusters, X.shape[1],)] The indicators for cluster membership of each column.

See also:

make_checkerboard

References

[1]

Examples using `sklearn.datasets.make_biclusters`

- *A demo of the Spectral Co-Clustering algorithm*

`sklearn.datasets.make_blobs`

`sklearn.datasets.`**`make_blobs`**(*n_samples=100*, *n_features=2*, *centers=None*, *cluster_std=1.0*, *center_box=(-10.0, 10.0)*, *shuffle=True*, *random_state=None*)

Generate isotropic Gaussian blobs for clustering.

Read more in the *User Guide*.

Parameters

n_samples [int or array-like, optional (default=100)] If int, it is the total number of points equally divided among clusters. If array-like, each element of the sequence indicates the number of samples per cluster.

n_features [int, optional (default=2)] The number of features for each sample.

centers [int or array of shape [n_centers, n_features], optional] (default=None) The number of centers to generate, or the fixed center locations. If n_samples is an int and centers is None, 3 centers are generated. If n_samples is array-like, centers must be either None or an array of length equal to the length of n_samples.

cluster_std [float or sequence of floats, optional (default=1.0)] The standard deviation of the clusters.

center_box [pair of floats (min, max), optional (default=(-10.0, 10.0))] The bounding box for each cluster center when centers are generated at random.

shuffle [boolean, optional (default=True)] Shuffle the samples.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, n_features]] The generated samples.

y [array of shape [n_samples]] The integer labels for cluster membership of each sample.

See also:

make_classification a more intricate variant

Examples

```
>>> from sklearn.datasets import make_blobs
>>> X, y = make_blobs(n_samples=10, centers=3, n_features=2,
...                   random_state=0)
>>> print(X.shape)
(10, 2)
>>> y
array([0, 0, 1, 0, 2, 2, 2, 1, 1, 0])
>>> X, y = make_blobs(n_samples=[3, 3, 4], centers=None, n_features=2,
...                   random_state=0)
>>> print(X.shape)
```

(continues on next page)

(continued from previous page)

```
(10, 2)
>>> y
array([0, 1, 2, 0, 2, 2, 2, 1, 1, 0])
```

Examples using `sklearn.datasets.make_blobs`

- *Comparing anomaly detection algorithms for outlier detection on toy datasets*
- *Probability calibration of classifiers*
- *Probability Calibration for 3-class classification*
- *Normal and Shrinkage Linear Discriminant Analysis for classification*
- *A demo of the mean-shift clustering algorithm*
- *Demonstration of k-means assumptions*
- *Demo of affinity propagation clustering algorithm*
- *Demo of DBSCAN clustering algorithm*
- *Inductive Clustering*
- *Compare BIRCH and MiniBatchKMeans*
- *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*
- *Comparing different hierarchical linkage methods on toy datasets*
- *Selecting the number of clusters with silhouette analysis on KMeans clustering*
- *Comparing different clustering algorithms on toy datasets*
- *Plot randomly generated classification dataset*
- *SGD: Maximum margin separating hyperplane*
- *Plot multinomial and One-vs-Rest Logistic Regression*
- *Demonstrating the different strategies of KBinsDiscretizer*
- *SVM: Maximum margin separating hyperplane*
- *Plot the support vectors in LinearSVC*
- *SVM Tie Breaking Example*
- *SVM: Separating hyperplane for unbalanced classes*

`sklearn.datasets`.make_checkerboard

`sklearn.datasets.`**`make_checkerboard`**(*shape*, *n_clusters*, *noise=0.0*, *minval=10*, *maxval=100*, *shuffle=True*, *random_state=None*)

Generate an array with block checkerboard structure for biclustering.

Read more in the *User Guide*.

Parameters

shape [iterable (n_rows, n_cols)] The shape of the result.

n_clusters [integer or iterable (n_row_clusters, n_column_clusters)] The number of row and column clusters.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise.

minval [int, optional (default=10)] Minimum value of a bicluster.

maxval [int, optional (default=100)] Maximum value of a bicluster.

shuffle [boolean, optional (default=True)] Shuffle the samples.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape `shape`] The generated array.

rows [array of shape (n_clusters, X.shape[0],)] The indicators for cluster membership of each row.

cols [array of shape (n_clusters, X.shape[1],)] The indicators for cluster membership of each column.

See also:

make_biclusters

References

[1]

Examples using `sklearn.datasets.make_checkerboard`

- *A demo of the Spectral Biclustering algorithm*

`sklearn.datasets`.make_circles

`sklearn.datasets.`**`make_circles`**(*n_samples=100*, *shuffle=True*, *noise=None*, *random_state=None*, *factor=0.8*)

Make a large circle containing a smaller circle in 2d.

A simple toy dataset to visualize clustering and classification algorithms.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The total number of points generated. If odd, the inner circle will have one point more than the outer circle.

shuffle [bool, optional (default=True)] Whether to shuffle the samples.

noise [double or None (default=None)] Standard deviation of Gaussian noise added to the data.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset shuffling and noise. Pass an int for reproducible output across multiple function calls. See *Glossary*.

factor [0 < double < 1 (default=.8)] Scale factor between inner and outer circle.

Returns

X [array of shape [n_samples, 2]] The generated samples.

y [array of shape [n_samples]] The integer labels (0 or 1) for class membership of each sample.

Examples using `sklearn.datasets.make_circles`

- *Classifier comparison*
- *Comparing different hierarchical linkage methods on toy datasets*
- *Comparing different clustering algorithms on toy datasets*
- *Kernel PCA*
- *Hashing feature transformation using Totally Random Trees*
- *t-SNE: The effect of various perplexity values on the shape*
- *Varying regularization in Multi-layer Perceptron*
- *Compare Stochastic learning strategies for MLPClassifier*
- *Feature discretization*
- *Label Propagation learning a complex structure*

`sklearn.datasets`.make_classification

sklearn.datasets.**make_classification**(*n_samples=100*, *n_features=20*, *n_informative=2*, *n_redundant=2*, *n_repeated=0*, *n_classes=2*, *n_clusters_per_class=2*, *weights=None*, *flip_y=0.01*, *class_sep=1.0*, *hypercube=True*, *shift=0.0*, *scale=1.0*, *shuffle=True*, *random_state=None*)

Generate a random n-class classification problem.

This initially creates clusters of points normally distributed (std=1) about vertices of an `n_informative`-dimensional hypercube with sides of length `2*class_sep` and assigns an equal number of clusters to each class. It introduces interdependence between these features and adds various types of further noise to the data.

Without shuffling, `X` horizontally stacks features in the following order: the primary `n_informative` features, followed by `n_redundant` linear combinations of the informative features, followed by `n_repeated` duplicates, drawn randomly with replacement from the informative and redundant features. The remaining features are filled with random noise. Thus, without shuffling, all useful features are contained in the columns `X[:, :n_informative + n_redundant + n_repeated]`.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

n_features [int, optional (default=20)] The total number of features. These comprise `n_informative` informative features, `n_redundant` redundant features, `n_repeated` duplicated features and `n_features-n_informative-n_redundant-n_repeated` useless features drawn at random.

n_informative [int, optional (default=2)] The number of informative features. Each class is composed of a number of gaussian clusters each located around the vertices of a hypercube in a subspace of dimension `n_informative`. For each cluster, informative features are drawn independently from N(0, 1) and then randomly linearly combined within each cluster in order to add covariance. The clusters are then placed on the vertices of the hypercube.

n_redundant [int, optional (default=2)] The number of redundant features. These features are generated as random linear combinations of the informative features.

n_repeated [int, optional (default=0)] The number of duplicated features, drawn randomly from the informative and the redundant features.

n_classes [int, optional (default=2)] The number of classes (or labels) of the classification problem.

n_clusters_per_class [int, optional (default=2)] The number of clusters per class.

weights [array-like of shape (n_classes,) or (n_classes - 1,), (default=None)] The proportions of samples assigned to each class. If None, then classes are balanced. Note that if `len(weights) == n_classes - 1`, then the last class weight is automatically inferred. More than `n_samples` samples may be returned if the sum of `weights` exceeds 1.

flip_y [float, optional (default=0.01)] The fraction of samples whose class is assigned randomly. Larger values introduce noise in the labels and make the classification task harder.

class_sep [float, optional (default=1.0)] The factor multiplying the hypercube size. Larger values spread out the clusters/classes and make the classification task easier.

hypercube [boolean, optional (default=True)] If True, the clusters are put on the vertices of a hypercube. If False, the clusters are put on the vertices of a random polytope.

shift [float, array of shape [n_features] or None, optional (default=0.0)] Shift features by the specified value. If None, then features are shifted by a random value drawn in [-class_sep, class_sep].

scale [float, array of shape [n_features] or None, optional (default=1.0)] Multiply features by the specified value. If None, then features are scaled by a random value drawn in [1, 100]. Note that scaling happens after shifting.

shuffle [boolean, optional (default=True)] Shuffle the samples and the features.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, n_features]] The generated samples.

y [array of shape [n_samples]] The integer labels for class membership of each sample.

See also:

make_blobs simplified variant

make_multilabel_classification unrelated generator for multilabel tasks

Notes

The algorithm is adapted from Guyon [1] and was designed to generate the "Madelon" dataset.

References

[1]

Examples using `sklearn.datasets.make_classification`

- *Comparison of Calibration of Classifiers*
- *Probability Calibration curves*
- *Classifier comparison*
- *Plot randomly generated classification dataset*
- *Feature importances with forests of trees*
- *OOB Errors for Random Forests*
- *Feature transformations with ensembles of trees*
- *Pipeline Anova SVM*
- *Recursive feature elimination with cross-validation*
- *Neighborhood Components Analysis Illustration*
- *Varying regularization in Multi-layer Perceptron*
- *Feature discretization*
- *Release Highlights for scikit-learn 0.22*
- *Scaling the regularization parameter for SVCs*

`sklearn.datasets.make_friedman1`

`sklearn.datasets.`**`make_friedman1`**(*n_samples=100*, *n_features=10*, *noise=0.0*, *random_state=None*)

Generate the "Friedman #1" regression problem

This dataset is described in Friedman [1] and Breiman [2].

Inputs `X` are independent features uniformly distributed on the interval [0, 1]. The output `y` is created according to the formula:

```
y(X) = 10 * sin(pi * X[:, 0] * X[:, 1]) + 20 * (X[:, 2] - 0.5) ** 2 + 10 * X[:,␣
↪3] + 5 * X[:, 4] + noise * N(0, 1).
```

Out of the `n_features` features, only 5 are actually used to compute `y`. The remaining features are independent of `y`.

The number of features has to be >= 5.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

n_features [int, optional (default=10)] The number of features. Should be at least 5.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise applied to the output.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset noise. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, n_features]] The input samples.

y [array of shape [n_samples]] The output values.

References

[1], [2]

sklearn.datasets.make_friedman2

sklearn.datasets.**make_friedman2**(*n_samples=100*, *noise=0.0*, *random_state=None*)

Generate the "Friedman #2" regression problem

This dataset is described in Friedman [1] and Breiman [2].

Inputs `X` are 4 independent features uniformly distributed on the intervals:

```
0 <= X[:, 0] <= 100,
40 * pi <= X[:, 1] <= 560 * pi,
0 <= X[:, 2] <= 1,
1 <= X[:, 3] <= 11.
```

The output `y` is created according to the formula:

```
y(X) = (X[:, 0] ** 2 + (X[:, 1] * X[:, 2]  - 1 / (X[:, 1] * X[:, 3])) ** 2) ** 0.
↪5 + noise * N(0, 1).
```

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise applied to the output.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset noise. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, 4]] The input samples.

y [array of shape [n_samples]] The output values.

References

[1], [2]

sklearn.datasets.make_friedman3

sklearn.datasets.**make_friedman3**(*n_samples=100*, *noise=0.0*, *random_state=None*)

Generate the "Friedman #3" regression problem

This dataset is described in Friedman [1] and Breiman [2].

Inputs `X` are 4 independent features uniformly distributed on the intervals:

```
0 <= X[:, 0] <= 100,
40 * pi <= X[:, 1] <= 560 * pi,
0 <= X[:, 2] <= 1,
1 <= X[:, 3] <= 11.
```

The output `y` is created according to the formula:

```
y(X) = arctan((X[:, 1] * X[:, 2] - 1 / (X[:, 1] * X[:, 3])) / X[:, 0]) + noise *␣
↪N(0, 1).
```

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise applied to the output.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset noise. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, 4]] The input samples.

y [array of shape [n_samples]] The output values.

References

[1], [2]

sklearn.datasets.make_gaussian_quantiles

sklearn.datasets.**make_gaussian_quantiles**(*mean=None*, *cov=1.0*, *n_samples=100*, *n_features=2*, *n_classes=3*, *shuffle=True*, *random_state=None*)

Generate isotropic Gaussian and label samples by quantile

This classification dataset is constructed by taking a multi-dimensional standard normal distribution and defining classes separated by nested concentric multi-dimensional spheres such that roughly equal numbers of samples are in each class (quantiles of the χ^2 distribution).

Read more in the *User Guide*.

Parameters

mean [array of shape [n_features], optional (default=None)] The mean of the multi-dimensional normal distribution. If None then use the origin (0, 0, ...).

cov [float, optional (default=1.)] The covariance matrix will be this value times the unit matrix. This dataset only produces symmetric normal distributions.

n_samples [int, optional (default=100)] The total number of points equally divided among classes.

n_features [int, optional (default=2)] The number of features for each sample.

n_classes [int, optional (default=3)] The number of classes

shuffle [boolean, optional (default=True)] Shuffle the samples.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, n_features]] The generated samples.

y [array of shape [n_samples]] The integer labels for quantile membership of each sample.

Notes

The dataset is from Zhu et al [1].

References

[1]

Examples using `sklearn.datasets.make_gaussian_quantiles`

- *Plot randomly generated classification dataset*
- *Two-class AdaBoost*
- *Multi-class AdaBoosted Decision Trees*

sklearn.datasets.make_hastie_10_2

sklearn.datasets.**make_hastie_10_2**(*n_samples=12000*, *random_state=None*)

Generates data for binary classification used in Hastie et al. 2009, Example 10.2.

The ten features are standard independent Gaussian and the target `y` is defined by:

```
y[i] = 1 if np.sum(X[i] ** 2) > 9.34 else -1
```

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=12000)] The number of samples.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, 10]] The input samples.

y [array of shape [n_samples]] The output values.

See also:

make_gaussian_quantiles a generalization of this dataset approach

References

[1]

Examples using `sklearn.datasets.make_hastie_10_2`

- *Gradient Boosting regularization*
- *Discrete versus Real AdaBoost*
- *Early stopping of Gradient Boosting*
- *Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV*

`sklearn.datasets`.make_low_rank_matrix

`sklearn.datasets.`**`make_low_rank_matrix`**(*n_samples=100*, *n_features=100*, *effective_rank=10*, *tail_strength=0.5*, *random_state=None*)

Generate a mostly low rank matrix with bell-shaped singular values

Most of the variance can be explained by a bell-shaped curve of width effective_rank: the low rank part of the singular values profile is:

```
(1 - tail_strength) * exp(-1.0 * (i / effective_rank) ** 2)
```

The remaining singular values' tail is fat, decreasing as:

```
tail_strength * exp(-0.1 * i / effective_rank).
```

The low rank part of the profile can be considered the structured signal part of the data while the tail can be considered the noisy part of the data that cannot be summarized by a low number of linear components (singular vectors).

This kind of singular profiles is often seen in practice, for instance:

- gray level pictures of faces
- TF-IDF vectors of text documents crawled from the web

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

n_features [int, optional (default=100)] The number of features.

effective_rank [int, optional (default=10)] The approximate number of singular vectors required to explain most of the data by linear combinations.

tail_strength [float between 0.0 and 1.0, optional (default=0.5)] The relative importance of the fat noisy tail of the singular values profile.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, n_features]] The matrix.

sklearn.datasets.make_moons

sklearn.datasets.**make_moons**(*n_samples=100*, *shuffle=True*, *noise=None*, *random_state=None*)

Make two interleaving half circles

A simple toy dataset to visualize clustering and classification algorithms. Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The total number of points generated.

shuffle [bool, optional (default=True)] Whether to shuffle the samples.

noise [double or None (default=None)] Standard deviation of Gaussian noise added to the data.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset shuffling and noise. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, 2]] The generated samples.

y [array of shape [n_samples]] The integer labels (0 or 1) for class membership of each sample.

Examples using sklearn.datasets.make_moons

- *Comparing anomaly detection algorithms for outlier detection on toy datasets*
- *Classifier comparison*
- *Comparing different hierarchical linkage methods on toy datasets*
- *Comparing different clustering algorithms on toy datasets*
- *Varying regularization in Multi-layer Perceptron*
- *Compare Stochastic learning strategies for MLPClassifier*
- *Feature discretization*

sklearn.datasets.make_multilabel_classification

sklearn.datasets.**make_multilabel_classification**(*n_samples=100*, *n_features=20*, *n_classes=5*, *n_labels=2*, *length=50*, *allow_unlabeled=True*, *sparse=False*, *return_indicator='dense'*, *return_distributions=False*, *random_state=None*)

Generate a random multilabel classification problem.

For each sample, the generative process is:

- pick the number of labels: n ~ Poisson(n_labels)
- n times, choose a class c: c ~ Multinomial(theta)
- pick the document length: k ~ Poisson(length)
- k times, choose a word: w ~ Multinomial(theta_c)

In the above process, rejection sampling is used to make sure that n is never zero or more than `n_classes`, and that the document length is never zero. Likewise, we reject classes which have already been chosen.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

n_features [int, optional (default=20)] The total number of features.

n_classes [int, optional (default=5)] The number of classes of the classification problem.

n_labels [int, optional (default=2)] The average number of labels per instance. More precisely, the number of labels per sample is drawn from a Poisson distribution with `n_labels` as its expected value, but samples are bounded (using rejection sampling) by `n_classes`, and must be nonzero if `allow_unlabeled` is False.

length [int, optional (default=50)] The sum of the features (number of words if documents) is drawn from a Poisson distribution with this expected value.

allow_unlabeled [bool, optional (default=True)] If `True`, some instances might not belong to any class.

sparse [bool, optional (default=False)] If `True`, return a sparse feature matrix

New in version 0.17: parameter to allow *sparse* output.

return_indicator [‘dense’ (default) | ‘sparse’ | False] If `dense` return `Y` in the dense binary indicator format. If `'sparse'` return `Y` in the sparse binary indicator format. `False` returns a list of lists of labels.

return_distributions [bool, optional (default=False)] If `True`, return the prior class probability and conditional probabilities of features given classes, from which the data was drawn.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, n_features]] The generated samples.

Y [array or sparse CSR matrix of shape [n_samples, n_classes]] The label sets.

p_c [array, shape [n_classes]] The probability of each class being drawn. Only returned if `return_distributions=True`.

p_w_c [array, shape [n_features, n_classes]] The probability of each feature being drawn given each class. Only returned if `return_distributions=True`.

Examples using `sklearn.datasets.make_multilabel_classification`

- *Multilabel classification*
- *Plot randomly generated multilabel dataset*

sklearn.datasets.make_regression

sklearn.datasets.**make_regression**(*n_samples=100*, *n_features=100*, *n_informative=10*, *n_targets=1*, *bias=0.0*, *effective_rank=None*, *tail_strength=0.5*, *noise=0.0*, *shuffle=True*, *coef=False*, *random_state=None*)

Generate a random regression problem.

The input set can either be well conditioned (by default) or have a low rank-fat tail singular profile. See *make_low_rank_matrix* for more details.

The output is generated by applying a (potentially biased) random linear regression model with `n_informative` nonzero regressors to the previously generated input and some gaussian centered noise with some adjustable scale.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

n_features [int, optional (default=100)] The number of features.

n_informative [int, optional (default=10)] The number of informative features, i.e., the number of features used to build the linear model used to generate the output.

n_targets [int, optional (default=1)] The number of regression targets, i.e., the dimension of the y output vector associated with a sample. By default, the output is a scalar.

bias [float, optional (default=0.0)] The bias term in the underlying linear model.

effective_rank [int or None, optional (default=None)]

if not None: The approximate number of singular vectors required to explain most of the input data by linear combinations. Using this kind of singular spectrum in the input allows the generator to reproduce the correlations often observed in practice.

if None: The input set is well conditioned, centered and gaussian with unit variance.

tail_strength [float between 0.0 and 1.0, optional (default=0.5)] The relative importance of the fat noisy tail of the singular values profile if `effective_rank` is not None.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise applied to the output.

shuffle [boolean, optional (default=True)] Shuffle the samples and the features.

coef [boolean, optional (default=False)] If True, the coefficients of the underlying linear model are returned.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, n_features]] The input samples.

y [array of shape [n_samples] or [n_samples, n_targets]] The output values.

coef [array of shape [n_features] or [n_features, n_targets], optional] The coefficient of the underlying linear model. It is returned only if coef is True.

Examples using sklearn.datasets.make_regression

- *Prediction Latency*
- *Plot Ridge coefficients as a function of the L2 regularization*
- *Robust linear model estimation using RANSAC*
- *HuberRegressor vs Ridge on dataset with strong outliers*
- *Lasso on dense and sparse data*
- *Effect of transforming the targets in regression model*

sklearn.datasets.make_s_curve

sklearn.datasets.**make_s_curve**(*n_samples=100*, *noise=0.0*, *random_state=None*)

Generate an S curve dataset.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of sample points on the S curve.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, 3]] The points.

t [array of shape [n_samples]] The univariate position of the sample according to the main dimension of the points in the manifold.

Examples using sklearn.datasets.make_s_curve

- *Comparison of Manifold Learning methods*
- *t-SNE: The effect of various perplexity values on the shape*

sklearn.datasets.make_sparse_coded_signal

sklearn.datasets.**make_sparse_coded_signal**(*n_samples*, *n_components*, *n_features*, *n_nonzero_coefs*, *random_state=None*)

Generate a signal as a sparse combination of dictionary elements.

Returns a matrix Y = DX, such as D is (n_features, n_components), X is (n_components, n_samples) and each column of X has exactly n_nonzero_coefs non-zero elements.

Read more in the *User Guide*.

Parameters

n_samples [int] number of samples to generate

n_components [int,] number of components in the dictionary

n_features [int] number of features of the dataset to generate

n_nonzero_coefs [int] number of active (non-zero) coefficients in each sample

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

data [array of shape [n_features, n_samples]] The encoded signal (Y).

dictionary [array of shape [n_features, n_components]] The dictionary with normalized components (D).

code [array of shape [n_components, n_samples]] The sparse code such that each column of this matrix has exactly n_nonzero_coefs non-zero items (X).

Examples using `sklearn.datasets.make_sparse_coded_signal`

- *Orthogonal Matching Pursuit*

`sklearn.datasets.make_sparse_spd_matrix`

`sklearn.datasets.`**`make_sparse_spd_matrix`**(*dim=1*, *alpha=0.95*, *norm_diag=False*, *smallest_coef=0.1*, *largest_coef=0.9*, *random_state=None*)

Generate a sparse symmetric definite positive matrix.

Read more in the *User Guide*.

Parameters

dim [integer, optional (default=1)] The size of the random matrix to generate.

alpha [float between 0 and 1, optional (default=0.95)] The probability that a coefficient is zero (see notes). Larger values enforce more sparsity.

norm_diag [boolean, optional (default=False)] Whether to normalize the output matrix to make the leading diagonal elements all 1

smallest_coef [float between 0 and 1, optional (default=0.1)] The value of the smallest coefficient.

largest_coef [float between 0 and 1, optional (default=0.9)] The value of the largest coefficient.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

prec [sparse matrix of shape (dim, dim)] The generated matrix.

See also:

make_spd_matrix

Notes

The sparsity is actually imposed on the cholesky factor of the matrix. Thus alpha does not translate directly into the filling fraction of the matrix itself.

Examples using `sklearn.datasets.make_sparse_spd_matrix`

- *Sparse inverse covariance estimation*

`sklearn.datasets`.make_sparse_uncorrelated

`sklearn.datasets.`**`make_sparse_uncorrelated`**(*n_samples=100*, *n_features=10*, *random_state=None*)

Generate a random regression problem with sparse uncorrelated design

This dataset is described in Celeux et al [1]. as:

```
X ~ N(0, 1)
y(X) = X[:, 0] + 2 * X[:, 1] - 2 * X[:, 2] - 1.5 * X[:, 3]
```

Only the first 4 features are informative. The remaining features are useless.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

n_features [int, optional (default=10)] The number of features.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, n_features]] The input samples.

y [array of shape [n_samples]] The output values.

References

[1]

`sklearn.datasets`.make_spd_matrix

`sklearn.datasets.`**`make_spd_matrix`**(*n_dim*, *random_state=None*)

Generate a random symmetric, positive-definite matrix.

Read more in the *User Guide*.

Parameters

n_dim [int] The matrix dimension.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_dim, n_dim]] The random symmetric, positive-definite matrix.

See also:

make_sparse_spd_matrix

sklearn.datasets.make_swiss_roll

sklearn.datasets.**make_swiss_roll**(*n_samples=100*, *noise=0.0*, *random_state=None*)

Generate a swiss roll dataset.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of sample points on the S curve.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, 3]] The points.

t [array of shape [n_samples]] The univariate position of the sample according to the main dimension of the points in the manifold.

Notes

The algorithm is from Marsland [1].

References

[1]

Examples using sklearn.datasets.make_swiss_roll

- *Hierarchical clustering: structured vs unstructured ward*
- *Swiss Roll reduction with LLE*

7.8 sklearn.decomposition: Matrix Decomposition

The *sklearn.decomposition* module includes matrix decomposition algorithms, including among others PCA, NMF or ICA. Most of the algorithms of this module can be regarded as dimensionality reduction techniques.

User guide: See the *Decomposing signals in components (matrix factorization problems)* section for further details.

decomposition.DictionaryLearning([...])	Dictionary learning
decomposition.FactorAnalysis([n_components, ...])	Factor Analysis (FA)
decomposition.FastICA([n_components, ...])	FastICA: a fast algorithm for Independent Component Analysis.
decomposition.IncrementalPCA([n_components, ...])	Incremental principal components analysis (IPCA).
decomposition.KernelPCA([n_components, ...])	Kernel Principal component analysis (KPCA)
decomposition.LatentDirichletAllocation([...])	Latent Dirichlet Allocation with online variational Bayes algorithm
decomposition.MiniBatchDictionaryLearning([...])	Mini-batch dictionary learning
decomposition.MiniBatchSparsePCA([...])	Mini-batch Sparse Principal Components Analysis
decomposition.NMF([n_components, init, ...])	Non-Negative Matrix Factorization (NMF)
decomposition.PCA([n_components, copy, ...])	Principal component analysis (PCA).
decomposition.SparsePCA([n_components, ...])	Sparse Principal Components Analysis (SparsePCA)
decomposition.SparseCoder(dictionary[, ...])	Sparse coding
decomposition.TruncatedSVD([n_components, ...])	Dimensionality reduction using truncated SVD (aka LSA).

7.8.1 sklearn.decomposition.DictionaryLearning

class sklearn.decomposition.**DictionaryLearning**(*n_components=None*, *alpha=1*, *max_iter=1000*, *tol=1e-08*, *fit_algorithm='lars'*, *transform_algorithm='omp'*, *transform_n_nonzero_coefs=None*, *transform_alpha=None*, *n_jobs=None*, *code_init=None*, *dict_init=None*, *verbose=False*, *split_sign=False*, *random_state=None*, *positive_code=False*, *positive_dict=False*, *transform_max_iter=1000*)

Dictionary learning

Finds a dictionary (a set of atoms) that can best be used to represent data using a sparse code.

Solves the optimization problem:

```
(U^*,V^*) = argmin 0.5 || Y - U V ||_2^2 + alpha * || U ||_1
            (U,V)
            with || V_k ||_2 = 1 for all  0 <= k < n_components
```

Read more in the *User Guide*.

Parameters

n_components [int, default=n_features] number of dictionary elements to extract

alpha [float, default=1.0] sparsity controlling parameter

max_iter [int, default=1000] maximum number of iterations to perform

tol [float, default=1e-8] tolerance for numerical error

fit_algorithm [{'lars', 'cd'}, default='lars'] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

New in version 0.17: *cd* coordinate descent method to improve speed.

transform_algorithm [{'lasso_lars', 'lasso_cd', 'lars', 'omp', 'threshold'}, default='omp'] Algorithm used to transform the data lars: uses the least angle regression method (linear_model.lars_path) lasso_lars: uses Lars to compute the Lasso solution lasso_cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). lasso_lars will be faster if the estimated components are sparse. omp: uses orthogonal matching pursuit to estimate the sparse solution threshold: squashes to zero all coefficients less than alpha from the projection `dictionary * X'`

New in version 0.17: *lasso_cd* coordinate descent method to improve speed.

transform_n_nonzero_coefs [int, default=0.1*n_features] Number of nonzero coefficients to target in each column of the solution. This is only used by `algorithm='lars'` and `algorithm='omp'` and is overridden by `alpha` in the `omp` case.

transform_alpha [float, default=1.0] If `algorithm='lasso_lars'` or `algorithm='lasso_cd'`, `alpha` is the penalty applied to the L1 norm. If `algorithm='threshold'`, `alpha` is the absolute value of the threshold below which coefficients will be squashed to zero. If `algorithm='omp'`, `alpha` is the tolerance parameter: the value of the reconstruction error targeted. In this case, it overrides `n_nonzero_coefs`.

n_jobs [int or None, default=None] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

code_init [array of shape (n_samples, n_components), default=None] initial value for the code, for warm restart

dict_init [array of shape (n_components, n_features), default=None] initial values for the dictionary, for warm restart

verbose [bool, default=False] To control the verbosity of the procedure.

split_sign [bool, default=False] Whether to split the sparse feature vector into the concatenation of its negative part and its positive part. This can improve the performance of downstream classifiers.

random_state [int, RandomState instance or None, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

positive_code [bool, default=False] Whether to enforce positivity when finding the code.

New in version 0.20.

positive_dict [bool, default=False] Whether to enforce positivity when finding the dictionary

New in version 0.20.

transform_max_iter [int, default=1000] Maximum number of iterations to perform if `algorithm='lasso_cd'` or `lasso_lars`.

New in version 0.22.

Attributes

components_ [array, [n_components, n_features]] dictionary atoms extracted from the data

error_ [array] vector of errors at each iteration

n_iter_ [int] Number of iterations run.

See also:

SparseCoder

MiniBatchDictionaryLearning

SparsePCA

MiniBatchSparsePCA

Notes

References:

J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009: Online dictionary learning for sparse coding (https://www.di.ens.fr/sierra/pdfs/icml09.pdf)

Methods

fit(self, X[, y])	Fit the model from data in X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Encode the data as a sparse combination of the dictionary atoms.

__init__(*self*, *n_components=None*, *alpha=1*, *max_iter=1000*, *tol=1e-08*, *fit_algorithm='lars'*, *transform_algorithm='omp'*, *transform_n_nonzero_coefs=None*, *transform_alpha=None*, *n_jobs=None*, *code_init=None*, *dict_init=None*, *verbose=False*, *split_sign=False*, *random_state=None*, *positive_code=False*, *positive_dict=False*, *transform_max_iter=1000*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the model from data in X.

Parameters

X [array-like, shape (n_samples, n_features)] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [Ignored]

Returns

self [object] Returns the object itself

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Encode the data as a sparse combination of the dictionary atoms.

Coding method is determined by the object parameter `transform_algorithm`.

Parameters

X [array of shape (n_samples, n_features)] Test data to be transformed, must have the same number of features as the data used to train the model.

Returns

X_new [array, shape (n_samples, n_components)] Transformed data

7.8.2 sklearn.decomposition.FactorAnalysis

class sklearn.decomposition.**FactorAnalysis**(*n_components=None*, *tol=0.01*, *copy=True*, *max_iter=1000*, *noise_variance_init=None*, *svd_method='randomized'*, *iterated_power=3*, *random_state=0*)
Factor Analysis (FA)

A simple linear generative model with Gaussian latent variables.

The observations are assumed to be caused by a linear transformation of lower dimensional latent factors and added Gaussian noise. Without loss of generality the factors are distributed according to a Gaussian with zero mean and unit covariance. The noise is also zero mean and has an arbitrary diagonal covariance matrix.

If we would restrict the model further, by assuming that the Gaussian noise is even isotropic (all diagonal entries are the same) we would obtain `PPCA`.

FactorAnalysis performs a maximum likelihood estimate of the so-called `loading` matrix, the transformation of the latent variables to the observed ones, using SVD based approach.

Read more in the *User Guide*.

New in version 0.13.

Parameters

n_components [int | None] Dimensionality of latent space, the number of components of `X` that are obtained after `transform`. If None, n_components is set to the number of features.

tol [float] Stopping tolerance for log-likelihood increase.

copy [bool] Whether to make a copy of X. If `False`, the input X gets overwritten during fitting.

max_iter [int] Maximum number of iterations.

noise_variance_init [None | array, shape=(n_features,)] The initial guess of the noise variance for each feature. If None, it defaults to np.ones(n_features)

svd_method [{'lapack', 'randomized'}] Which SVD method to use. If 'lapack' use standard SVD from scipy.linalg, if 'randomized' use fast `randomized_svd` function. Defaults to 'randomized'. For most applications 'randomized' will be sufficiently precise while providing significant speed gains. Accuracy can also be improved by setting higher values for `iterated_power`. If this is not sufficient, for maximum precision you should choose 'lapack'.

iterated_power [int, optional] Number of iterations for the power method. 3 by default. Only used if `svd_method` equals 'randomized'

random_state [int, RandomState instance or None, optional (default=0)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Only used when `svd_method` equals 'randomized'.

Attributes

components_ [array, [n_components, n_features]] Components with maximum variance.

loglike_ [list, [n_iterations]] The log likelihood at each iteration.

noise_variance_ [array, shape=(n_features,)] The estimated noise variance for each feature.

n_iter_ [int] Number of iterations run.

mean_ [array, shape (n_features,)] Per-feature empirical mean, estimated from the training set.

See also:

PCA Principal component analysis is also a latent linear variable model which however assumes equal noise variance for each feature. This extra assumption makes probabilistic PCA faster as it can be computed in closed form.

FastICA Independent component analysis, a latent variable model with non-Gaussian latent variables.

References

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.decomposition import FactorAnalysis
>>> X, _ = load_digits(return_X_y=True)
>>> transformer = FactorAnalysis(n_components=7, random_state=0)
>>> X_transformed = transformer.fit_transform(X)
>>> X_transformed.shape
(1797, 7)
```

Methods

fit(self, X[, y])	Fit the FactorAnalysis model to X using SVD based approach
fit_transform(self, X[, y])	Fit to data, then transform it.
get_covariance(self)	Compute data covariance with the FactorAnalysis model.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Compute data precision matrix with the FactorAnalysis model.
score(self, X[, y])	Compute the average log-likelihood of the samples
score_samples(self, X)	Compute the log-likelihood of each sample
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply dimensionality reduction to X using the model.

__init__(*self*, *n_components=None*, *tol=0.01*, *copy=True*, *max_iter=1000*, *noise_variance_init=None*, *svd_method='randomized'*, *iterated_power=3*, *random_state=0*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the FactorAnalysis model to X using SVD based approach

Parameters

X [array-like, shape (n_samples, n_features)] Training data.

y [Ignored]

Returns

self

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_covariance(*self*)
Compute data covariance with the FactorAnalysis model.

```
cov = components_.T * components_ + diag(noise_variance)
```

Returns

cov [array, shape (n_features, n_features)] Estimated covariance of data.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_precision(*self*)
Compute data precision matrix with the FactorAnalysis model.

Returns

precision [array, shape (n_features, n_features)] Estimated precision of data.

score(*self*, *X*, *y=None*)
Compute the average log-likelihood of the samples

Parameters

X [array, shape (n_samples, n_features)] The data

y [Ignored]

Returns

ll [float] Average log-likelihood of the samples under the current model

score_samples(*self*, *X*)
Compute the log-likelihood of each sample

Parameters

X [array, shape (n_samples, n_features)] The data

Returns

ll [array, shape (n_samples,)] Log-likelihood of each sample under the current model

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Apply dimensionality reduction to X using the model.

Compute the expected mean of the latent variables. See Barber, 21.2.33 (or Bishop, 12.66).

Parameters

X [array-like, shape (n_samples, n_features)] Training data.

Returns

X_new [array-like, shape (n_samples, n_components)] The latent variables of X.

Examples using `sklearn.decomposition.FactorAnalysis`

- *Model selection with Probabilistic PCA and Factor Analysis (FA)*
- *Faces dataset decompositions*

7.8.3 `sklearn.decomposition`.FastICA

class sklearn.decomposition.**FastICA**(*n_components=None*, *algorithm='parallel'*, *whiten=True*, *fun='logcosh'*, *fun_args=None*, *max_iter=200*, *tol=0.0001*, *w_init=None*, *random_state=None*)
FastICA: a fast algorithm for Independent Component Analysis.

Read more in the *User Guide*.

Parameters

n_components [int, optional] Number of components to use. If none is passed, all are used.

algorithm [{'parallel', 'deflation'}] Apply parallel or deflational algorithm for FastICA.

whiten [boolean, optional] If whiten is false, the data is already considered to be whitened, and no whitening is performed.

fun [string or function, optional. Default: 'logcosh'] The functional form of the G function used in the approximation to neg-entropy. Could be either 'logcosh', 'exp', or 'cube'. You can also provide your own function. It should return a tuple containing the value of the function, and of its derivative, in the point. Example:

def my_g(x): return x ** 3, (3 * x ** 2).mean(axis=-1)

fun_args [dictionary, optional] Arguments to send to the functional form. If empty and if fun='logcosh', fun_args will take value {'alpha' : 1.0}.

max_iter [int, optional] Maximum number of iterations during fit.

tol [float, optional] Tolerance on update at each iteration.

w_init [None of an (n_components, n_components) ndarray] The mixing matrix to be used to initialize the algorithm.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Attributes

components_ [2D array, shape (n_components, n_features)] The linear operator to apply to the data to get the independent sources. This is equal to the unmixing matrix when `whiten` is False, and equal to `np.dot(unmixing_matrix, self.whitening_)` when `whiten` is True.

mixing_ [array, shape (n_features, n_components)] The pseudo-inverse of `components_`. It is the linear operator that maps independent sources to the data.

mean_ [array, shape(n_features)] The mean over features. Only set if `self.whiten` is True.

n_iter_ [int] If the algorithm is "deflation", n_iter is the maximum number of iterations run across all components. Else they are just the number of iterations taken to converge.

whitening_ [array, shape (n_components, n_features)] Only set if whiten is 'True'. This is the pre-whitening matrix that projects data onto the first `n_components` principal components.

Notes

Implementation based on *A. Hyvarinen and E. Oja, Independent Component Analysis: Algorithms and Applications, Neural Networks, 13(4-5), 2000, pp. 411-430*

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.decomposition import FastICA
>>> X, _ = load_digits(return_X_y=True)
>>> transformer = FastICA(n_components=7,
...         random_state=0)
>>> X_transformed = transformer.fit_transform(X)
>>> X_transformed.shape
(1797, 7)
```

Methods

fit(self, X[, y])	Fit the model to X.
fit_transform(self, X[, y])	Fit the model and recover the sources from X.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X[, copy])	Transform the sources back to the mixed data (apply mixing matrix).
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, copy])	Recover the sources from X (apply the unmixing matrix).

__init__(*self*, *n_components=None*, *algorithm='parallel'*, *whiten=True*, *fun='logcosh'*, *fun_args=None*, *max_iter=200*, *tol=0.0001*, *w_init=None*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the model to X.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number

of samples and n_features is the number of features.

y [Ignored]

Returns

self

fit_transform(*self*, *X*, *y=None*)
Fit the model and recover the sources from X.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [Ignored]

Returns

X_new [array-like, shape (n_samples, n_components)]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*, *copy=True*)
Transform the sources back to the mixed data (apply mixing matrix).

Parameters

X [array-like, shape (n_samples, n_components)] Sources, where n_samples is the number of samples and n_components is the number of components.

copy [bool (optional)] If False, data passed to fit are overwritten. Defaults to True.

Returns

X_new [array-like, shape (n_samples, n_features)]

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*, *copy=True*)
Recover the sources from X (apply the unmixing matrix).

Parameters

X [array-like, shape (n_samples, n_features)] Data to transform, where n_samples is the number of samples and n_features is the number of features.

copy [bool (optional)] If False, data passed to fit are overwritten. Defaults to True.

Returns

X_new [array-like, shape (n_samples, n_components)]

Examples using `sklearn.decomposition.FastICA`

- *Blind source separation using FastICA*
- *FastICA on 2D point clouds*
- *Faces dataset decompositions*

7.8.4 `sklearn.decomposition`.IncrementalPCA

class sklearn.decomposition.**IncrementalPCA**(*n_components=None*, *whiten=False*, *copy=True*, *batch_size=None*)

Incremental principal components analysis (IPCA).

Linear dimensionality reduction using Singular Value Decomposition of the data, keeping only the most significant singular vectors to project the data to a lower dimensional space. The input data is centered but not scaled for each feature before applying the SVD.

Depending on the size of the input data, this algorithm can be much more memory efficient than a PCA, and allows sparse input.

This algorithm has constant memory complexity, on the order of `batch_size * n_features`, enabling use of np.memmap files without loading the entire file into memory. For sparse matrices, the input is converted to dense in batches (in order to be able to subtract the mean) which avoids storing the entire dense matrix at any one time.

The computational overhead of each SVD is `O(batch_size * n_features ** 2)`, but only 2 * batch_size samples remain in memory at a time. There will be `n_samples / batch_size` SVD computations to get the principal components, versus 1 large SVD of complexity `O(n_samples * n_features ** 2)` for PCA.

Read more in the *User Guide*.

New in version 0.16.

Parameters

n_components [int or None, (default=None)] Number of components to keep. If `n_components `` is ``None`, then `n_components` is set to `min(n_samples, n_features)`.

whiten [bool, optional] When True (False by default) the `components_` vectors are divided by `n_samples` times `components_` to ensure uncorrelated outputs with unit component-wise variances.

Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometimes improve the predictive accuracy of the downstream estimators by making data respect some hard-wired assumptions.

copy [bool, (default=True)] If False, X will be overwritten. `copy=False` can be used to save memory but is unsafe for general use.

batch_size [int or None, (default=None)] The number of samples to use for each batch. Only used when calling `fit`. If `batch_size` is `None`, then `batch_size` is inferred from the data and set to `5 * n_features`, to provide a balance between approximation accuracy and memory consumption.

Attributes

components_ [array, shape (n_components, n_features)] Components with maximum variance.

explained_variance_ [array, shape (n_components,)] Variance explained by each of the selected components.

explained_variance_ratio_ [array, shape (n_components,)] Percentage of variance explained by each of the selected components. If all components are stored, the sum of explained variances is equal to 1.0.

singular_values_ [array, shape (n_components,)] The singular values corresponding to each of the selected components. The singular values are equal to the 2-norms of the `n_components` variables in the lower-dimensional space.

mean_ [array, shape (n_features,)] Per-feature empirical mean, aggregate over calls to `partial_fit`.

var_ [array, shape (n_features,)] Per-feature empirical variance, aggregate over calls to `partial_fit`.

noise_variance_ [float] The estimated noise covariance following the Probabilistic PCA model from Tipping and Bishop 1999. See "Pattern Recognition and Machine Learning" by C. Bishop, 12.2.1 p. 574 or http://www.miketipping.com/papers/met-mppca.pdf.

n_components_ [int] The estimated number of components. Relevant when `n_components=None`.

n_samples_seen_ [int] The number of samples processed by the estimator. Will be reset on new calls to fit, but increments across `partial_fit` calls.

See also:

PCA

KernelPCA

SparsePCA

TruncatedSVD

Notes

Implements the incremental PCA model from: *D. Ross, J. Lim, R. Lin, M. Yang, Incremental Learning for Robust Visual Tracking, International Journal of Computer Vision, Volume 77, Issue 1-3, pp. 125-141, May 2008.* See https://www.cs.toronto.edu/~dross/ivt/RossLimLinYang_ijcv.pdf

This model is an extension of the Sequential Karhunen-Loeve Transform from: *A. Levy and M. Lindenbaum, Sequential Karhunen-Loeve Basis Extraction and its Application to Images, IEEE Transactions on Image Processing, Volume 9, Number 8, pp. 1371-1374, August 2000.* See https://www.cs.technion.ac.il/~mic/doc/skl-ip.pdf

We have specifically abstained from an optimization used by authors of both papers, a QR decomposition used in specific situations to reduce the algorithmic complexity of the SVD. The source for this technique is *Matrix Computations, Third Edition, G. Holub and C. Van Loan, Chapter 5, section 5.4.4, pp 252-253.*. This technique has been omitted because it is advantageous only when decomposing a matrix with `n_samples` (rows) >=

5/3 * n_features (columns), and hurts the readability of the implemented algorithm. This would be a good opportunity for future optimization, if it is deemed necessary.

References

D. Ross, J. Lim, R. Lin, M. Yang. Incremental Learning for Robust Visual Tracking, International Journal of Computer Vision, Volume 77, Issue 1-3, pp. 125-141, May 2008.

G. Golub and C. Van Loan. Matrix Computations, Third Edition, Chapter 5, Section 5.4.4, pp. 252-253.

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.decomposition import IncrementalPCA
>>> from scipy import sparse
>>> X, _ = load_digits(return_X_y=True)
>>> transformer = IncrementalPCA(n_components=7, batch_size=200)
>>> # either partially fit on smaller batches of data
>>> transformer.partial_fit(X[:100, :])
IncrementalPCA(batch_size=200, n_components=7)
>>> # or let the fit function itself divide the data into batches
>>> X_sparse = sparse.csr_matrix(X)
>>> X_transformed = transformer.fit_transform(X_sparse)
>>> X_transformed.shape
(1797, 7)
```

Methods

fit(self, X[, y])	Fit the model with X, using minibatches of size batch_size.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_covariance(self)	Compute data covariance with the generative model.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Compute data precision matrix with the generative model.
inverse_transform(self, X)	Transform data back to its original space.
partial_fit(self, X[, y, check_input])	Incremental fit with X.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply dimensionality reduction to X.

__init__(*self*, *n_components=None*, *whiten=False*, *copy=True*, *batch_size=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the model with X, using minibatches of size batch_size.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [Ignored]

Returns

self [object] Returns the instance itself.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_covariance(*self*)
Compute data covariance with the generative model.

`cov = components_.T * S**2 * components_ + sigma2 * eye(n_features)`
where S**2 contains the explained variances, and sigma2 contains the noise variances.

Returns

cov [array, shape=(n_features, n_features)] Estimated covariance of data.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_precision(*self*)
Compute data precision matrix with the generative model.

Equals the inverse of the covariance but computed with the matrix inversion lemma for efficiency.

Returns

precision [array, shape=(n_features, n_features)] Estimated precision of data.

inverse_transform(*self*, *X*)
Transform data back to its original space.

In other words, return an input X_original whose transform would be X.

Parameters

X [array-like, shape (n_samples, n_components)] New data, where n_samples is the number of samples and n_components is the number of components.

Returns

X_original array-like, shape (n_samples, n_features)

Notes

If whitening is enabled, inverse_transform will compute the exact inverse operation, which includes reversing whitening.

partial_fit(*self*, *X*, *y=None*, *check_input=True*)
Incremental fit with X. All of X is processed as a single batch.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

check_input [bool] Run check_array on X.

y [Ignored]

Returns

self [object] Returns the instance itself.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Apply dimensionality reduction to X.

X is projected on the first principal components previously extracted from a training set, using minibatches of size batch_size if X is sparse.

Parameters

X [array-like, shape (n_samples, n_features)] New data, where n_samples is the number of samples and n_features is the number of features.

Returns

X_new [array-like, shape (n_samples, n_components)]

Examples

```
>>> import numpy as np
>>> from sklearn.decomposition import IncrementalPCA
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2],
...               [1, 1], [2, 1], [3, 2]])
>>> ipca = IncrementalPCA(n_components=2, batch_size=3)
>>> ipca.fit(X)
IncrementalPCA(batch_size=3, n_components=2)
>>> ipca.transform(X) # doctest: +SKIP
```

Examples using `sklearn.decomposition.IncrementalPCA`

- *Incremental PCA*

7.8.5 `sklearn.decomposition`.KernelPCA

class sklearn.decomposition.**KernelPCA**(*n_components=None*, *kernel='linear'*, *gamma=None*, *degree=3*, *coef0=1*, *kernel_params=None*, *alpha=1.0*, *fit_inverse_transform=False*, *eigen_solver='auto'*, *tol=0*, *max_iter=None*, *remove_zero_eig=False*, *random_state=None*, *copy_X=True*, *n_jobs=None*)

Kernel Principal component analysis (KPCA)

Non-linear dimensionality reduction through the use of kernels (see *Pairwise metrics, Affinities and Kernels*).

Read more in the *User Guide*.

Parameters

n_components [int, default=None] Number of components. If None, all non-zero components are kept.

kernel ["linear" | "poly" | "rbf" | "sigmoid" | "cosine" | "precomputed"] Kernel. Default="linear".

gamma [float, default=1/n_features] Kernel coefficient for rbf, poly and sigmoid kernels. Ignored by other kernels.

degree [int, default=3] Degree for poly kernels. Ignored by other kernels.

coef0 [float, default=1] Independent term in poly and sigmoid kernels. Ignored by other kernels.

kernel_params [mapping of string to any, default=None] Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.

alpha [int, default=1.0] Hyperparameter of the ridge regression that learns the inverse transform (when fit_inverse_transform=True).

fit_inverse_transform [bool, default=False] Learn the inverse transform for non-precomputed kernels. (i.e. learn to find the pre-image of a point)

eigen_solver [string ['auto'|'dense'|'arpack'], default='auto'] Select eigensolver to use. If n_components is much less than the number of training samples, arpack may be more efficient than the dense eigensolver.

tol [float, default=0] Convergence tolerance for arpack. If 0, optimal value will be chosen by arpack.

max_iter [int, default=None] Maximum number of iterations for arpack. If None, optimal value will be chosen by arpack.

remove_zero_eig [boolean, default=False] If True, then all components with zero eigenvalues are removed, so that the number of components in the output may be < n_components (and sometimes even zero due to numerical instability). When n_components is None, this parameter is ignored and components with zero eigenvalues are removed regardless.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `eigen_solver` == 'arpack'.

New in version 0.18.

copy_X [boolean, default=True] If True, input X is copied and stored by the model in the `X_fit_` attribute. If no further changes will be done to X, setting `copy_X=False` saves memory by storing a reference.

New in version 0.18.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

New in version 0.18.

Attributes

lambdas_ [array, (n_components,)] Eigenvalues of the centered kernel matrix in decreasing order. If `n_components` and `remove_zero_eig` are not set, then all values are stored.

alphas_ [array, (n_samples, n_components)] Eigenvectors of the centered kernel matrix. If `n_components` and `remove_zero_eig` are not set, then all components are stored.

dual_coef_ [array, (n_samples, n_features)] Inverse transform matrix. Only available when `fit_inverse_transform` is True.

X_transformed_fit_ [array, (n_samples, n_components)] Projection of the fitted data on the kernel principal components. Only available when `fit_inverse_transform` is True.

X_fit_ [(n_samples, n_features)] The data used to fit the model. If `copy_X=False`, then `X_fit_` is a reference. This attribute is used for the calls to transform.

References

Kernel PCA was introduced in: Bernhard Schoelkopf, Alexander J. Smola, and Klaus-Robert Mueller. 1999. Kernel principal component analysis. In Advances in kernel methods, MIT Press, Cambridge, MA, USA 327-352.

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.decomposition import KernelPCA
>>> X, _ = load_digits(return_X_y=True)
>>> transformer = KernelPCA(n_components=7, kernel='linear')
>>> X_transformed = transformer.fit_transform(X)
>>> X_transformed.shape
(1797, 7)
```

Methods

fit(self, X[, y])	Fit the model from data in X.
fit_transform(self, X[, y])	Fit the model from data in X and transform X.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Transform X back to original space.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform X.

__init__(*self*, *n_components=None*, *kernel='linear'*, *gamma=None*, *degree=3*, *coef0=1*, *kernel_params=None*, *alpha=1.0*, *fit_inverse_transform=False*, *eigen_solver='auto'*, *tol=0*, *max_iter=None*, *remove_zero_eig=False*, *random_state=None*, *copy_X=True*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the model from data in X.

Parameters

X [array-like, shape (n_samples, n_features)] Training vector, where n_samples in the number of samples and n_features is the number of features.

Returns

self [object] Returns the instance itself.

fit_transform(*self*, *X*, *y=None*, ***params*)
Fit the model from data in X and transform X.

Parameters

X [array-like, shape (n_samples, n_features)] Training vector, where n_samples in the number of samples and n_features is the number of features.

Returns

X_new [array-like, shape (n_samples, n_components)]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*)
Transform X back to original space.

Parameters

X [array-like, shape (n_samples, n_components)]

Returns

X_new [array-like, shape (n_samples, n_features)]

References

"Learning to Find Pre-Images", G BakIr et al, 2004.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Transform X.

Parameters

X [array-like, shape (n_samples, n_features)]

Returns

X_new [array-like, shape (n_samples, n_components)]

Examples using `sklearn.decomposition.KernelPCA`

- *Kernel PCA*

7.8.6 `sklearn.decomposition`.LatentDirichletAllocation

class sklearn.decomposition.**LatentDirichletAllocation**(*n_components=10*, *doc_topic_prior=None*, *topic_word_prior=None*, *learning_method='batch'*, *learning_decay=0.7*, *learning_offset=10.0*, *max_iter=10*, *batch_size=128*, *evaluate_every=-1*, *total_samples=1000000.0*, *perp_tol=0.1*, *mean_change_tol=0.001*, *max_doc_update_iter=100*, *n_jobs=None*, *verbose=0*, *random_state=None*)

Latent Dirichlet Allocation with online variational Bayes algorithm

New in version 0.17.

Read more in the *User Guide*.

Parameters

n_components [int, optional (default=10)] Number of topics.

doc_topic_prior [float, optional (default=None)] Prior of document topic distribution `theta`. If the value is None, defaults to `1 / n_components`. In [Re25e5648fc37-1], this is called `alpha`.

topic_word_prior [float, optional (default=None)] Prior of topic word distribution `beta`. If the value is None, defaults to `1 / n_components`. In [Re25e5648fc37-1], this is called `eta`.

learning_method ['batch' | 'online', default='batch'] Method used to update `_component`. Only used in *`fit`* method. In general, if the data size is large, the online update will be much faster than the batch update.

Valid options:

```
'batch': Batch variational Bayes method. Use all training data in
    each EM update.
    Old `components_` will be overwritten in each iteration.
'online': Online variational Bayes method. In each EM update, use
    mini-batch of training data to update the ``components_``
    variable incrementally. The learning rate is controlled by the
    ``learning_decay`` and the ``learning_offset`` parameters.
```

Changed in version 0.20: The default learning method is now `"batch"`.

learning_decay [float, optional (default=0.7)] It is a parameter that control learning rate in the online learning method. The value should be set between (0.5, 1.0] to guarantee asymptotic convergence. When the value is 0.0 and batch_size is `n_samples`, the update method is same as batch learning. In the literature, this is called kappa.

learning_offset [float, optional (default=10.)] A (positive) parameter that downweights early iterations in online learning. It should be greater than 1.0. In the literature, this is called tau_0.

max_iter [integer, optional (default=10)] The maximum number of iterations.

batch_size [int, optional (default=128)] Number of documents to use in each EM iteration. Only used in online learning.

evaluate_every [int, optional (default=0)] How often to evaluate perplexity. Only used in `fit` method. set it to 0 or negative number to not evaluate perplexity in training at all. Evaluating perplexity can help you check convergence in training process, but it will also increase total training time. Evaluating perplexity in every iteration might increase training time up to two-fold.

total_samples [int, optional (default=1e6)] Total number of documents. Only used in the *partial_fit* method.

perp_tol [float, optional (default=1e-1)] Perplexity tolerance in batch learning. Only used when `evaluate_every` is greater than 0.

mean_change_tol [float, optional (default=1e-3)] Stopping tolerance for updating document topic distribution in E-step.

max_doc_update_iter [int (default=100)] Max number of iterations for updating document topic distribution in the E-step.

n_jobs [int or None, optional (default=None)] The number of jobs to use in the E-step. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [int, optional (default=0)] Verbosity level.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Attributes

components_ [array, [n_components, n_features]] Variational parameters for topic word distribution. Since the complete conditional for topic word distribution is a Dirichlet, `components_[i, j]` can be viewed as pseudocount that represents the number of times word `j` was assigned to topic `i`. It can also be viewed as distribution over the words

for each topic after normalization: `model.components_ / model.components_.sum(axis=1)[:, np.newaxis]`.

n_batch_iter_ [int] Number of iterations of the EM step.

n_iter_ [int] Number of passes over the dataset.

bound_ [float] Final perplexity score on training set.

doc_topic_prior_ [float] Prior of document topic distribution `theta`. If the value is None, it is `1 / n_components`.

topic_word_prior_ [float] Prior of topic word distribution `beta`. If the value is None, it is `1 / n_components`.

References

[2] "Stochastic Variational Inference", Matthew D. Hoffman, David M. Blei, Chong Wang, John Paisley, 2013

[3] Matthew D. Hoffman's onlineldavb code. Link: https://github.com/blei-lab/onlineldavb

[Re25e5648fc37-1]

Examples

```
>>> from sklearn.decomposition import LatentDirichletAllocation
>>> from sklearn.datasets import make_multilabel_classification
>>> # This produces a feature matrix of token counts, similar to what
>>> # CountVectorizer would produce on text.
>>> X, _ = make_multilabel_classification(random_state=0)
>>> lda = LatentDirichletAllocation(n_components=5,
...     random_state=0)
>>> lda.fit(X)
LatentDirichletAllocation(...)
>>> # get topics for some given samples:
>>> lda.transform(X[-2:])
array([[0.00360392, 0.25499205, 0.0036211 , 0.64236448, 0.09541846],
       [0.15297572, 0.00362644, 0.44412786, 0.39568399, 0.003586  ]])
```

Methods

fit(self, X[, y])	Learn model for the data X with variational Bayes method.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X[, y])	Online VB with Mini-Batch update.
perplexity(self, X[, sub_sampling])	Calculate approximate perplexity for data X.
score(self, X[, y])	Calculate approximate log-likelihood as score.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform data X according to the fitted model.

__init__(*self*, *n_components=10*, *doc_topic_prior=None*, *topic_word_prior=None*, *learning_method='batch'*, *learning_decay=0.7*, *learning_offset=10.0*, *max_iter=10*, *batch_size=128*, *evaluate_every=-1*, *total_samples=1000000.0*, *perp_tol=0.1*, *mean_change_tol=0.001*, *max_doc_update_iter=100*, *n_jobs=None*, *verbose=0*, *random_state=None*)

Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)

Learn model for the data X with variational Bayes method.

When `learning_method` is 'online', use mini-batch update. Otherwise, use batch update.

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] Document word matrix.

y [Ignored]

Returns

self

fit_transform(*self*, *X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit(*self*, *X*, *y=None*)

Online VB with Mini-Batch update.

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] Document word matrix.

y [Ignored]

Returns

self

perplexity(*self*, *X*, *sub_sampling=False*)

Calculate approximate perplexity for data X.

Perplexity is defined as exp(-1. * log-likelihood per word)

Changed in version 0.19: *doc_topic_distr* argument has been deprecated and is ignored because user no longer has access to unnormalized distribution

Parameters

X [array-like or sparse matrix, [n_samples, n_features]] Document word matrix.

sub_sampling [bool] Do sub-sampling or not.

Returns

score [float] Perplexity score.

score(*self*, *X*, *y=None*)
Calculate approximate log-likelihood as score.

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] Document word matrix.

y [Ignored]

Returns

score [float] Use approximate bound as score.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Transform data X according to the fitted model.

Changed in version 0.18: *doc_topic_distr* is now normalized

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] Document word matrix.

Returns

doc_topic_distr [shape=(n_samples, n_components)] Document topic distribution for X.

Examples using `sklearn.decomposition.LatentDirichletAllocation`

- *Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation*

7.8.7 sklearn.decomposition.MiniBatchDictionaryLearning

class sklearn.decomposition.**MiniBatchDictionaryLearning**(*n_components=None*, *alpha=1*, *n_iter=1000*, *fit_algorithm='lars'*, *n_jobs=None*, *batch_size=3*, *shuffle=True*, *dict_init=None*, *transform_algorithm='omp'*, *transform_n_nonzero_coefs=None*, *transform_alpha=None*, *verbose=False*, *split_sign=False*, *random_state=None*, *positive_code=False*, *positive_dict=False*, *transform_max_iter=1000*)

Mini-batch dictionary learning

Finds a dictionary (a set of atoms) that can best be used to represent data using a sparse code.

Solves the optimization problem:

```
(U^*,V^*) = argmin 0.5 || Y - U V ||_2^2 + alpha * || U ||_1
             (U,V)
             with || V_k ||_2 = 1 for all  0 <= k < n_components
```

Read more in the *User Guide*.

Parameters

n_components [int,] number of dictionary elements to extract

alpha [float,] sparsity controlling parameter

n_iter [int,] total number of iterations to perform

fit_algorithm [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

n_jobs [int or None, optional (default=None)] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

batch_size [int,] number of samples in each mini-batch

shuffle [bool,] whether to shuffle the samples before forming batches

dict_init [array of shape (n_components, n_features),] initial value of the dictionary for warm restart scenarios

transform_algorithm [{'lasso_lars', 'lasso_cd', 'lars', 'omp', 'threshold'}] Algorithm used to transform the data. lars: uses the least angle regression method (linear_model.lars_path) lasso_lars: uses Lars to compute the Lasso solution lasso_cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). lasso_lars will be faster if the estimated components are sparse. omp: uses orthogonal matching pursuit to estimate

the sparse solution threshold: squashes to zero all coefficients less than alpha from the projection dictionary * X'

transform_n_nonzero_coefs [int, `0.1 * n_features` by default] Number of nonzero coefficients to target in each column of the solution. This is only used by `algorithm='lars'` and `algorithm='omp'` and is overridden by `alpha` in the `omp` case.

transform_alpha [float, 1. by default] If `algorithm='lasso_lars'` or `algorithm='lasso_cd'`, `alpha` is the penalty applied to the L1 norm. If `algorithm='threshold'`, `alpha` is the absolute value of the threshold below which coefficients will be squashed to zero. If `algorithm='omp'`, `alpha` is the tolerance parameter: the value of the reconstruction error targeted. In this case, it overrides `n_nonzero_coefs`.

verbose [bool, optional (default: False)] To control the verbosity of the procedure.

split_sign [bool, False by default] Whether to split the sparse feature vector into the concatenation of its negative part and its positive part. This can improve the performance of downstream classifiers.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

positive_code [bool] Whether to enforce positivity when finding the code.

New in version 0.20.

positive_dict [bool] Whether to enforce positivity when finding the dictionary.

New in version 0.20.

transform_max_iter [int, optional (default=1000)] Maximum number of iterations to perform if `algorithm='lasso_cd'` or `lasso_lars`.

New in version 0.22.

Attributes

components_ [array, [n_components, n_features]] components extracted from the data

inner_stats_ [tuple of (A, B) ndarrays] Internal sufficient statistics that are kept by the algorithm. Keeping them is useful in online settings, to avoid losing the history of the evolution, but they shouldn't have any use for the end user. A (n_components, n_components) is the dictionary covariance matrix. B (n_features, n_components) is the data approximation matrix

n_iter_ [int] Number of iterations run.

iter_offset_ [int] The number of iteration on data batches that has been performed before.

random_state_ [RandomState] RandomState instance that is generated either from a seed, the random number generattor or by `np.random`.

See also:

SparseCoder

DictionaryLearning

SparsePCA

MiniBatchSparsePCA

Notes

References:

J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009: Online dictionary learning for sparse coding (https://www.di.ens.fr/sierra/pdfs/icml09.pdf)

Methods

fit(self, X[, y])	Fit the model from data in X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X[, y, iter_offset])	Updates the model using the data in X as a mini-batch.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Encode the data as a sparse combination of the dictionary atoms.

__init__(*self*, *n_components=None*, *alpha=1*, *n_iter=1000*, *fit_algorithm='lars'*, *n_jobs=None*, *batch_size=3*, *shuffle=True*, *dict_init=None*, *transform_algorithm='omp'*, *transform_n_nonzero_coefs=None*, *transform_alpha=None*, *verbose=False*, *split_sign=False*, *random_state=None*, *positive_code=False*, *positive_dict=False*, *transform_max_iter=1000*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the model from data in X.

Parameters

X [array-like, shape (n_samples, n_features)] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [Ignored]

Returns

self [object] Returns the instance itself.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit(*self*, *X*, *y=None*, *iter_offset=None*)
Updates the model using the data in X as a mini-batch.

Parameters

X [array-like, shape (n_samples, n_features)] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [Ignored]

iter_offset [integer, optional] The number of iteration on data batches that has been performed before this call to partial_fit. This is optional: if no number is passed, the memory of the object is used.

Returns

self [object] Returns the instance itself.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Encode the data as a sparse combination of the dictionary atoms.

Coding method is determined by the object parameter `transform_algorithm`.

Parameters

X [array of shape (n_samples, n_features)] Test data to be transformed, must have the same number of features as the data used to train the model.

Returns

X_new [array, shape (n_samples, n_components)] Transformed data

Examples using `sklearn.decomposition.MiniBatchDictionaryLearning`

- *Image denoising using dictionary learning*
- *Faces dataset decompositions*

7.8.8 sklearn.decomposition.MiniBatchSparsePCA

class sklearn.decomposition.**MiniBatchSparsePCA**(*n_components=None*, *alpha=1*, *ridge_alpha=0.01*, *n_iter=100*, *callback=None*, *batch_size=3*, *verbose=False*, *shuffle=True*, *n_jobs=None*, *method='lars'*, *random_state=None*, *normalize_components='deprecated'*)

Mini-batch Sparse Principal Components Analysis

Finds the set of sparse components that can optimally reconstruct the data. The amount of sparseness is controllable by the coefficient of the L1 penalty, given by the parameter alpha.

Read more in the *User Guide*.

Parameters

n_components [int,] number of sparse atoms to extract

alpha [int,] Sparsity controlling parameter. Higher values lead to sparser components.

ridge_alpha [float,] Amount of ridge shrinkage to apply in order to improve conditioning when calling the transform method.

n_iter [int,] number of iterations to perform for each mini batch

callback [callable or None, optional (default: None)] callable that gets invoked every five iterations

batch_size [int,] the number of features to take in each mini batch

verbose [int] Controls the verbosity; the higher, the more messages. Defaults to 0.

shuffle [boolean,] whether to shuffle the data before splitting it in batches

n_jobs [int or None, optional (default=None)] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

method [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

normalize_components ['deprecated'] This parameter does not have any effect. The components are always normalized.

New in version 0.20.

Deprecated since version 0.22: `normalize_components` is deprecated in 0.22 and will be removed in 0.24.

Attributes

components_ [array, [n_components, n_features]] Sparse components extracted from the data.

n_iter_ [int] Number of iterations run.

mean_ [array, shape (n_features,)] Per-feature empirical mean, estimated from the training set. Equal to `X.mean(axis=0)`.

See also:

PCA

SparsePCA

DictionaryLearning

Examples

```
>>> import numpy as np
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.decomposition import MiniBatchSparsePCA
>>> X, _ = make_friedman1(n_samples=200, n_features=30, random_state=0)
>>> transformer = MiniBatchSparsePCA(n_components=5, batch_size=50,
...                                  random_state=0)
>>> transformer.fit(X)
MiniBatchSparsePCA(...)
>>> X_transformed = transformer.transform(X)
>>> X_transformed.shape
(200, 5)
>>> # most values in the components_ are zero (sparsity)
>>> np.mean(transformer.components_ == 0)
0.94
```

Methods

fit(self, X[, y])	Fit the model from data in X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Least Squares projection of the data onto the sparse components.

__init__(*self*, *n_components=None*, *alpha=1*, *ridge_alpha=0.01*, *n_iter=100*, *callback=None*, *batch_size=3*, *verbose=False*, *shuffle=True*, *n_jobs=None*, *method='lars'*, *random_state=None*, *normalize_components='deprecated'*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the model from data in X.

Parameters

X [array-like, shape (n_samples, n_features)] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [Ignored]

Returns

self [object] Returns the instance itself.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Least Squares projection of the data onto the sparse components.

To avoid instability issues in case the system is under-determined, regularization can be applied (Ridge regression) via the `ridge_alpha` parameter.

Note that Sparse PCA components orthogonality is not enforced as in PCA hence one cannot use a simple linear projection.

Parameters

X [array of shape (n_samples, n_features)] Test data to be transformed, must have the same number of features as the data used to train the model.

Returns

X_new array, shape (n_samples, n_components) Transformed data.

Examples using `sklearn.decomposition.MiniBatchSparsePCA`

- *Faces dataset decompositions*

7.8.9 sklearn.decomposition.NMF

class sklearn.decomposition.**NMF**(*n_components=None*, *init=None*, *solver='cd'*, *beta_loss='frobenius'*, *tol=0.0001*, *max_iter=200*, *random_state=None*, *alpha=0.0*, *l1_ratio=0.0*, *verbose=0*, *shuffle=False*)

Non-Negative Matrix Factorization (NMF)

Find two non-negative matrices (W, H) whose product approximates the non- negative matrix X. This factorization can be used for example for dimensionality reduction, source separation or topic extraction.

The objective function is:

```
0.5 * ||X - WH||_Fro^2
+ alpha * l1_ratio * ||vec(W)||_1
+ alpha * l1_ratio * ||vec(H)||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
+ 0.5 * alpha * (1 - l1_ratio) * ||H||_Fro^2
```

Where:

```
||A||_Fro^2 = \sum_{i,j} A_{ij}^2 (Frobenius norm)
||vec(A)||_1 = \sum_{i,j} abs(A_{ij}) (Elementwise L1 norm)
```

For multiplicative-update ('mu') solver, the Frobenius norm (0.5 * ||X - WH||_Fro^2) can be changed into another beta-divergence loss, by changing the beta_loss parameter.

The objective function is minimized with an alternating minimization of W and H.

Read more in the *User Guide*.

Parameters

n_components [int or None] Number of components, if n_components is not set all features are kept.

init [None | 'random' | 'nndsvd' | 'nndsvda' | 'nndsvdar' | 'custom'] Method used to initialize the procedure. Default: None. Valid options:

- **None: 'nndsvd' if n_components <= min(n_samples, n_features),** otherwise random.
- **'random': non-negative random matrices, scaled with:** sqrt(X.mean() / n_components)
- **'nndsvd': Nonnegative Double Singular Value Decomposition (NNDSVD)** initialization (better for sparseness)
- **'nndsvda': NNDSVD with zeros filled with the average of X** (better when sparsity is not desired)
- **'nndsvdar': NNDSVD with zeros filled with small random values** (generally faster, less accurate alternative to NNDSVDa for when sparsity is not desired)
- 'custom': use custom matrices W and H

solver ['cd' | 'mu'] Numerical solver to use: 'cd' is a Coordinate Descent solver. 'mu' is a Multiplicative Update solver.

New in version 0.17: Coordinate Descent solver.

New in version 0.19: Multiplicative Update solver.

beta_loss [float or string, default 'frobenius'] String must be in {'frobenius', 'kullback-leibler', 'itakura-saito'}. Beta divergence to be minimized, measuring the distance between X and the dot product WH. Note that values different from 'frobenius' (or 2) and 'kullback-leibler' (or 1) lead to significantly slower fits. Note that for beta_loss <= 0 (or 'itakura-saito'), the input matrix X cannot contain zeros. Used only in 'mu' solver.

New in version 0.19.

tol [float, default: 1e-4] Tolerance of the stopping condition.

max_iter [integer, default: 200] Maximum number of iterations before timing out.

random_state [int, RandomState instance or None, optional, default: None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

alpha [double, default: 0.] Constant that multiplies the regularization terms. Set it to zero to have no regularization.

New in version 0.17: *alpha* used in the Coordinate Descent solver.

l1_ratio [double, default: 0.] The regularization mixing parameter, with 0 <= l1_ratio <= 1. For l1_ratio = 0 the penalty is an elementwise L2 penalty (aka Frobenius Norm). For l1_ratio = 1 it is an elementwise L1 penalty. For 0 < l1_ratio < 1, the penalty is a combination of L1 and L2.

New in version 0.17: Regularization parameter *l1_ratio* used in the Coordinate Descent solver.

verbose [bool, default=False] Whether to be verbose.

shuffle [boolean, default: False] If true, randomize the order of coordinates in the CD solver.

New in version 0.17: *shuffle* parameter used in the Coordinate Descent solver.

Attributes

components_ [array, [n_components, n_features]] Factorization matrix, sometimes called 'dictionary'.

n_components_ [integer] The number of components. It is same as the `n_components` parameter if it was given. Otherwise, it will be same as the number of features.

reconstruction_err_ [number] Frobenius norm of the matrix difference, or beta-divergence, between the training data `X` and the reconstructed data `WH` from the fitted model.

n_iter_ [int] Actual number of iterations.

References

Cichocki, Andrzej, and P. H. A. N. Anh-Huy. "Fast local algorithms for large scale nonnegative matrix and tensor factorizations." IEICE transactions on fundamentals of electronics, communications and computer sciences 92.3: 708-721, 2009.

Fevotte, C., & Idier, J. (2011). Algorithms for nonnegative matrix factorization with the beta-divergence. Neural Computation, 23(9).

Examples

```
>>> import numpy as np
>>> X = np.array([[1, 1], [2, 1], [3, 1.2], [4, 1], [5, 0.8], [6, 1]])
>>> from sklearn.decomposition import NMF
>>> model = NMF(n_components=2, init='random', random_state=0)
>>> W = model.fit_transform(X)
>>> H = model.components_
```

Methods

fit(self, X[, y])	Learn a NMF model for the data X.
fit_transform(self, X[, y, W, H])	Learn a NMF model for the data X and returns the transformed data.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, W)	Transform data back to its original space.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform the data X according to the fitted NMF model

__init__(*self*, *n_components=None*, *init=None*, *solver='cd'*, *beta_loss='frobenius'*, *tol=0.0001*, *max_iter=200*, *random_state=None*, *alpha=0.0*, *l1_ratio=0.0*, *verbose=0*, *shuffle=False*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*, ***params*)
Learn a NMF model for the data X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Data matrix to be decomposed

y [Ignored]

Returns

self

fit_transform(*self*, *X*, *y=None*, *W=None*, *H=None*)
Learn a NMF model for the data X and returns the transformed data.

This is more efficient than calling fit followed by transform.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Data matrix to be decomposed

y [Ignored]

W [array-like, shape (n_samples, n_components)] If init='custom', it is used as initial guess for the solution.

H [array-like, shape (n_components, n_features)] If init='custom', it is used as initial guess for the solution.

Returns

W [array, shape (n_samples, n_components)] Transformed data.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *W*)
Transform data back to its original space.

Parameters

W [{array-like, sparse matrix}, shape (n_samples, n_components)] Transformed data matrix

Returns

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Data matrix of original shape

New in version 0.18: ..

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Transform the data X according to the fitted NMF model

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Data matrix to be transformed by the model

Returns

W [array, shape (n_samples, n_components)] Transformed data

Examples using `sklearn.decomposition.NMF`

- *Beta-divergence loss functions*
- *Faces dataset decompositions*
- *Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation*
- *Selecting dimensionality reduction with Pipeline and GridSearchCV*

7.8.10 sklearn.decomposition.PCA

class sklearn.decomposition.**PCA**(*n_components=None*, *copy=True*, *whiten=False*, *svd_solver='auto'*, *tol=0.0*, *iterated_power='auto'*, *random_state=None*)

Principal component analysis (PCA).

Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space. The input data is centered but not scaled for each feature before applying the SVD.

It uses the LAPACK implementation of the full SVD or a randomized truncated SVD by the method of Halko et al. 2009, depending on the shape of the input data and the number of components to extract.

It can also use the scipy.sparse.linalg ARPACK implementation of the truncated SVD.

Notice that this class does not support sparse input. See *TruncatedSVD* for an alternative with sparse data.

Read more in the *User Guide*.

Parameters

n_components [int, float, None or str] Number of components to keep. if n_components is not set all components are kept:

```
n_components == min(n_samples, n_features)
```

If `n_components == 'mle'` and `svd_solver == 'full'`, Minka's MLE is used to guess the dimension. Use of `n_components == 'mle'` will interpret `svd_solver == 'auto'` as `svd_solver == 'full'`.

If `0 < n_components < 1` and `svd_solver == 'full'`, select the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by n_components.

If `svd_solver == 'arpack'`, the number of components must be strictly less than the minimum of n_features and n_samples.

Hence, the None case results in:

```
n_components == min(n_samples, n_features) - 1
```

copy [bool, default=True] If False, data passed to fit are overwritten and running fit(X).transform(X) will not yield the expected results, use fit_transform(X) instead.

whiten [bool, optional (default False)] When True (False by default) the `components_` vectors are multiplied by the square root of n_samples and then divided by the singular values to ensure uncorrelated outputs with unit component-wise variances.

Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making their data respect some hard-wired assumptions.

svd_solver [str {'auto', 'full', 'arpack', 'randomized'}]

If auto : The solver is selected by a default policy based on `X.shape` and `n_components`: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient 'randomized' method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.

If full : run exact full SVD calling the standard LAPACK solver via `scipy.linalg.svd` and select the components by postprocessing

If arpack : run SVD truncated to n_components calling ARPACK solver via `scipy.sparse.linalg.svds`. It requires strictly 0 < n_components < min(X.shape)

If randomized : run randomized SVD by the method of Halko et al.

New in version 0.18.0.

tol [float >= 0, optional (default .0)] Tolerance for singular values computed by svd_solver == 'arpack'.

New in version 0.18.0.

iterated_power [int >= 0, or 'auto', (default 'auto')] Number of iterations for the power method computed by svd_solver == 'randomized'.

New in version 0.18.0.

random_state [int, RandomState instance or None, optional (default None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `svd_solver` == 'arpack' or 'randomized'.

New in version 0.18.0.

Attributes

components_ [array, shape (n_components, n_features)] Principal axes in feature space, representing the directions of maximum variance in the data. The components are sorted by `explained_variance_`.

explained_variance_ [array, shape (n_components,)] The amount of variance explained by each of the selected components.

Equal to n_components largest eigenvalues of the covariance matrix of X.

New in version 0.18.

explained_variance_ratio_ [array, shape (n_components,)] Percentage of variance explained by each of the selected components.

If `n_components` is not set then all components are stored and the sum of the ratios is equal to 1.0.

singular_values_ [array, shape (n_components,)] The singular values corresponding to each of the selected components. The singular values are equal to the 2-norms of the `n_components` variables in the lower-dimensional space.

New in version 0.19.

mean_ [array, shape (n_features,)] Per-feature empirical mean, estimated from the training set.

Equal to `X.mean(axis=0)`.

n_components_ [int] The estimated number of components. When n_components is set to 'mle' or a number between 0 and 1 (with svd_solver == 'full') this number is estimated from input data. Otherwise it equals the parameter n_components, or the lesser value of n_features and n_samples if n_components is None.

n_features_ [int] Number of features in the training data.

n_samples_ [int] Number of samples in the training data.

noise_variance_ [float] The estimated noise covariance following the Probabilistic PCA model from Tipping and Bishop 1999. See "Pattern Recognition and Machine Learning" by C.

Bishop, 12.2.1 p. 574 or http://www.miketipping.com/papers/met-mppca.pdf. It is required to compute the estimated data covariance and score samples.

Equal to the average of (min(n_features, n_samples) - n_components) smallest eigenvalues of the covariance matrix of X.

See also:

KernelPCA Kernel Principal Component Analysis.

SparsePCA Sparse Principal Component Analysis.

TruncatedSVD Dimensionality reduction using truncated SVD.

IncrementalPCA Incremental Principal Component Analysis.

References

For n_components == 'mle', this class uses the method of *Minka, T. P. "Automatic choice of dimensionality for PCA". In NIPS, pp. 598-604*

Implements the probabilistic PCA model from: Tipping, M. E., and Bishop, C. M. (1999). "Probabilistic principal component analysis". Journal of the Royal Statistical Society: Series B (Statistical Methodology), 61(3), 611-622. via the score and score_samples methods. See http://www.miketipping.com/papers/met-mppca.pdf

For svd_solver == 'arpack', refer to `scipy.sparse.linalg.svds`.

For svd_solver == 'randomized', see: *Halko, N., Martinsson, P. G., and Tropp, J. A. (2011). "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions". SIAM review, 53(2), 217-288.* and also *Martinsson, P. G., Rokhlin, V., and Tygert, M. (2011). "A randomized algorithm for the decomposition of matrices". Applied and Computational Harmonic Analysis, 30(1), 47-68.*

Examples

```
>>> import numpy as np
>>> from sklearn.decomposition import PCA
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> pca = PCA(n_components=2)
>>> pca.fit(X)
PCA(n_components=2)
>>> print(pca.explained_variance_ratio_)
[0.9924... 0.0075...]
>>> print(pca.singular_values_)
[6.30061... 0.54980...]
```

```
>>> pca = PCA(n_components=2, svd_solver='full')
>>> pca.fit(X)
PCA(n_components=2, svd_solver='full')
>>> print(pca.explained_variance_ratio_)
[0.9924... 0.00755...]
>>> print(pca.singular_values_)
[6.30061... 0.54980...]
```

```
>>> pca = PCA(n_components=1, svd_solver='arpack')
>>> pca.fit(X)
```

(continues on next page)

(continued from previous page)

```
PCA(n_components=1, svd_solver='arpack')
>>> print(pca.explained_variance_ratio_)
[0.99244...]
>>> print(pca.singular_values_)
[6.30061...]
```

Methods

fit(self, X[, y])	Fit the model with X.
fit_transform(self, X[, y])	Fit the model with X and apply the dimensionality reduction on X.
get_covariance(self)	Compute data covariance with the generative model.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Compute data precision matrix with the generative model.
inverse_transform(self, X)	Transform data back to its original space.
score(self, X[, y])	Return the average log-likelihood of all samples.
score_samples(self, X)	Return the log-likelihood of each sample.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply dimensionality reduction to X.

__init__(*self*, *n_components=None*, *copy=True*, *whiten=False*, *svd_solver='auto'*, *tol=0.0*, *iterated_power='auto'*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the model with X.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [None] Ignored variable.

Returns

self [object] Returns the instance itself.

fit_transform(*self*, *X*, *y=None*)
Fit the model with X and apply the dimensionality reduction on X.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [None] Ignored variable.

Returns

X_new [array-like, shape (n_samples, n_components)] Transformed values.

Notes

This method returns a Fortran-ordered array. To convert it to a C-ordered array, use 'np.ascontiguousarray'.

get_covariance(*self*)
Compute data covariance with the generative model.

`cov = components_.T * S**2 * components_ + sigma2 * eye(n_features)`
where S**2 contains the explained variances, and sigma2 contains the noise variances.

Returns

cov [array, shape=(n_features, n_features)] Estimated covariance of data.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_precision(*self*)
Compute data precision matrix with the generative model.

Equals the inverse of the covariance but computed with the matrix inversion lemma for efficiency.

Returns

precision [array, shape=(n_features, n_features)] Estimated precision of data.

inverse_transform(*self*, *X*)
Transform data back to its original space.

In other words, return an input X_original whose transform would be X.

Parameters

X [array-like, shape (n_samples, n_components)] New data, where n_samples is the number of samples and n_components is the number of components.

Returns

X_original array-like, shape (n_samples, n_features)

Notes

If whitening is enabled, inverse_transform will compute the exact inverse operation, which includes reversing whitening.

score(*self*, *X*, *y=None*)
Return the average log-likelihood of all samples.

See. "Pattern Recognition and Machine Learning" by C. Bishop, 12.2.1 p. 574 or http://www.miketipping.com/papers/met-mppca.pdf

Parameters

X [array, shape(n_samples, n_features)] The data.

y [None] Ignored variable.

Returns

ll [float] Average log-likelihood of the samples under the current model.

score_samples(*self*, *X*)
Return the log-likelihood of each sample.

See. "Pattern Recognition and Machine Learning" by C. Bishop, 12.2.1 p. 574 or http://www.miketipping.com/papers/met-mppca.pdf

Parameters

X [array, shape(n_samples, n_features)] The data.

Returns

ll [array, shape (n_samples,)] Log-likelihood of each sample under the current model.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Apply dimensionality reduction to X.

X is projected on the first principal components previously extracted from a training set.

Parameters

X [array-like, shape (n_samples, n_features)] New data, where n_samples is the number of samples and n_features is the number of features.

Returns

X_new [array-like, shape (n_samples, n_components)]

Examples

```
>>> import numpy as np
>>> from sklearn.decomposition import IncrementalPCA
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> ipca = IncrementalPCA(n_components=2, batch_size=3)
>>> ipca.fit(X)
IncrementalPCA(batch_size=3, n_components=2)
>>> ipca.transform(X) # doctest: +SKIP
```

Examples using `sklearn.decomposition.PCA`

- *Multilabel classification*
- *Explicit feature map approximation for RBF kernels*
- *A demo of K-Means clustering on the handwritten digits data*
- *The Iris Dataset*

- *PCA example with Iris Data-set*
- *Incremental PCA*
- *Comparison of LDA and PCA 2D projection of Iris dataset*
- *Blind source separation using FastICA*
- *Principal components analysis (PCA)*
- *FastICA on 2D point clouds*
- *Kernel PCA*
- *Model selection with Probabilistic PCA and Factor Analysis (FA)*
- *Faces dataset decompositions*
- *Faces recognition example using eigenfaces and SVMs*
- *Multi-dimensional scaling*
- *Balance model complexity and cross-validated score*
- *Kernel Density Estimation*
- *Dimensionality Reduction with Neighborhood Components Analysis*
- *Concatenating multiple feature extraction methods*
- *Pipelining: chaining a PCA and a logistic regression*
- *Selecting dimensionality reduction with Pipeline and GridSearchCV*
- *Using FunctionTransformer to select columns*
- *Importance of Feature Scaling*

7.8.11 sklearn.decomposition.SparsePCA

class sklearn.decomposition.**SparsePCA**(*n_components=None*, *alpha=1*, *ridge_alpha=0.01*, *max_iter=1000*, *tol=1e-08*, *method='lars'*, *n_jobs=None*, *U_init=None*, *V_init=None*, *verbose=False*, *random_state=None*, *normalize_components='deprecated'*)

Sparse Principal Components Analysis (SparsePCA)

Finds the set of sparse components that can optimally reconstruct the data. The amount of sparseness is controllable by the coefficient of the L1 penalty, given by the parameter alpha.

Read more in the *User Guide*.

Parameters

n_components [int,] Number of sparse atoms to extract.

alpha [float,] Sparsity controlling parameter. Higher values lead to sparser components.

ridge_alpha [float,] Amount of ridge shrinkage to apply in order to improve conditioning when calling the transform method.

max_iter [int,] Maximum number of iterations to perform.

tol [float,] Tolerance for the stopping condition.

method [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

n_jobs [int or None, optional (default=None)] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

U_init [array of shape (n_samples, n_components),] Initial values for the loadings for warm restart scenarios.

V_init [array of shape (n_components, n_features),] Initial values for the components for warm restart scenarios.

verbose [int] Controls the verbosity; the higher, the more messages. Defaults to 0.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

normalize_components ['deprecated'] This parameter does not have any effect. The components are always normalized.

New in version 0.20.

Deprecated since version 0.22: `normalize_components` is deprecated in 0.22 and will be removed in 0.24.

Attributes

components_ [array, [n_components, n_features]] Sparse components extracted from the data.

error_ [array] Vector of errors at each iteration.

n_iter_ [int] Number of iterations run.

mean_ [array, shape (n_features,)] Per-feature empirical mean, estimated from the training set. Equal to `X.mean(axis=0)`.

See also:

PCA

MiniBatchSparsePCA

DictionaryLearning

Examples

```
>>> import numpy as np
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.decomposition import SparsePCA
>>> X, _ = make_friedman1(n_samples=200, n_features=30, random_state=0)
>>> transformer = SparsePCA(n_components=5, random_state=0)
>>> transformer.fit(X)
SparsePCA(...)
>>> X_transformed = transformer.transform(X)
>>> X_transformed.shape
(200, 5)
>>> # most values in the components_ are zero (sparsity)
```

(continues on next page)

(continued from previous page)

```
>>> np.mean(transformer.components_ == 0)
0.9666...
```

Methods

fit(self, X[, y])	Fit the model from data in X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Least Squares projection of the data onto the sparse components.

__init__(*self*, *n_components=None*, *alpha=1*, *ridge_alpha=0.01*, *max_iter=1000*, *tol=1e-08*, *method='lars'*, *n_jobs=None*, *U_init=None*, *V_init=None*, *verbose=False*, *random_state=None*, *normalize_components='deprecated'*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the model from data in X.

Parameters

X [array-like, shape (n_samples, n_features)] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [Ignored]

Returns

self [object] Returns the instance itself.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Least Squares projection of the data onto the sparse components.

To avoid instability issues in case the system is under-determined, regularization can be applied (Ridge regression) via the `ridge_alpha` parameter.

Note that Sparse PCA components orthogonality is not enforced as in PCA hence one cannot use a simple linear projection.

Parameters

X [array of shape (n_samples, n_features)] Test data to be transformed, must have the same number of features as the data used to train the model.

Returns

X_new array, shape (n_samples, n_components) Transformed data.

7.8.12 sklearn.decomposition.SparseCoder

class sklearn.decomposition.**SparseCoder**(*dictionary*, *transform_algorithm='omp'*, *transform_n_nonzero_coefs=None*, *transform_alpha=None*, *split_sign=False*, *n_jobs=None*, *positive_code=False*, *transform_max_iter=1000*)
Sparse coding

Finds a sparse representation of data against a fixed, precomputed dictionary.

Each row of the result is the solution to a sparse coding problem. The goal is to find a sparse array `code` such that:

```
X ~= code * dictionary
```

Read more in the *User Guide*.

Parameters

dictionary [array, [n_components, n_features]] The dictionary atoms used for sparse coding. Lines are assumed to be normalized to unit norm.

transform_algorithm [{'lasso_lars', 'lasso_cd', 'lars', 'omp', 'threshold'}, default='omp'] Algorithm used to transform the data: lars: uses the least angle regression method (linear_model.lars_path) lasso_lars: uses Lars to compute the Lasso solution lasso_cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). lasso_lars

will be faster if the estimated components are sparse. omp: uses orthogonal matching pursuit to estimate the sparse solution threshold: squashes to zero all coefficients less than alpha from the projection `dictionary * X'`

transform_n_nonzero_coefs [int, default=0.1*n_features] Number of nonzero coefficients to target in each column of the solution. This is only used by `algorithm='lars'` and `algorithm='omp'` and is overridden by `alpha` in the `omp` case.

transform_alpha [float, default=1.] If `algorithm='lasso_lars'` or `algorithm='lasso_cd'`, `alpha` is the penalty applied to the L1 norm. If `algorithm='threshold'`, `alpha` is the absolute value of the threshold below which coefficients will be squashed to zero. If `algorithm='omp'`, `alpha` is the tolerance parameter: the value of the reconstruction error targeted. In this case, it overrides `n_nonzero_coefs`.

split_sign [bool, default=False] Whether to split the sparse feature vector into the concatenation of its negative part and its positive part. This can improve the performance of downstream classifiers.

n_jobs [int or None, default=None] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

positive_code [bool, default=False] Whether to enforce positivity when finding the code.

New in version 0.20.

transform_max_iter [int, default=1000] Maximum number of iterations to perform if `algorithm='lasso_cd'` or `lasso_lars`.

New in version 0.22.

Attributes

components_ [array, [n_components, n_features]] The unchanged dictionary atoms

See also:

DictionaryLearning

MiniBatchDictionaryLearning

SparsePCA

MiniBatchSparsePCA

sparse_encode

Methods

fit(self, X[, y])	Do nothing and return the estimator unchanged
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Encode the data as a sparse combination of the dictionary atoms.

__init__(*self*, *dictionary*, *transform_algorithm='omp'*, *transform_n_nonzero_coefs=None*, *transform_alpha=None*, *split_sign=False*, *n_jobs=None*, *positive_code=False*, *transform_max_iter=1000*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Do nothing and return the estimator unchanged

This method is just there to implement the usual API and hence work in pipelines.

Parameters

X [Ignored]

y [Ignored]

Returns

self [object] Returns the object itself

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Encode the data as a sparse combination of the dictionary atoms.

Coding method is determined by the object parameter `transform_algorithm`.

Parameters

X [array of shape (n_samples, n_features)] Test data to be transformed, must have the same number of features as the data used to train the model.

Returns

X_new [array, shape (n_samples, n_components)] Transformed data

Examples using `sklearn.decomposition.SparseCoder`

- *Sparse coding with a precomputed dictionary*

7.8.13 `sklearn.decomposition.TruncatedSVD`

class sklearn.decomposition.**TruncatedSVD**(*n_components=2*, *algorithm='randomized'*, *n_iter=5*, *random_state=None*, *tol=0.0*)

Dimensionality reduction using truncated SVD (aka LSA).

This transformer performs linear dimensionality reduction by means of truncated singular value decomposition (SVD). Contrary to PCA, this estimator does not center the data before computing the singular value decomposition. This means it can work with scipy.sparse matrices efficiently.

In particular, truncated SVD works on term count/tf-idf matrices as returned by the vectorizers in sklearn.feature_extraction.text. In that context, it is known as latent semantic analysis (LSA).

This estimator supports two algorithms: a fast randomized SVD solver, and a "naive" algorithm that uses ARPACK as an eigensolver on (X * X.T) or (X.T * X), whichever is more efficient.

Read more in the *User Guide*.

Parameters

n_components [int, default = 2] Desired dimensionality of output data. Must be strictly less than the number of features. The default value is useful for visualisation. For LSA, a value of 100 is recommended.

algorithm [string, default = "randomized"] SVD solver to use. Either "arpack" for the ARPACK wrapper in SciPy (scipy.sparse.linalg.svds), or "randomized" for the randomized algorithm due to Halko (2009).

n_iter [int, optional (default 5)] Number of iterations for randomized SVD solver. Not used by ARPACK. The default is larger than the default in `~sklearn.utils.extmath.randomized_svd` to handle sparse matrices that may have large slowly decaying spectrum.

random_state [int, RandomState instance or None, optional, default = None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

tol [float, optional] Tolerance for ARPACK. 0 means machine precision. Ignored by randomized SVD solver.

Attributes

components_ [array, shape (n_components, n_features)]

explained_variance_ [array, shape (n_components,)] The variance of the training samples transformed by a projection to each component.

explained_variance_ratio_ [array, shape (n_components,)] Percentage of variance explained by each of the selected components.

singular_values_ [array, shape (n_components,)] The singular values corresponding to each of the selected components. The singular values are equal to the 2-norms of the `n_components` variables in the lower-dimensional space.

See also:

PCA

Notes

SVD suffers from a problem called "sign indeterminacy", which means the sign of the `components_` and the output from transform depend on the algorithm and random state. To work around this, fit instances of this class to data once, then keep the instance around to do transformations.

References

Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions Halko, et al., 2009 (arXiv:909) https://arxiv.org/pdf/0909.4061.pdf

Examples

```
>>> from sklearn.decomposition import TruncatedSVD
>>> from scipy.sparse import random as sparse_random
>>> from sklearn.random_projection import sparse_random_matrix
>>> X = sparse_random(100, 100, density=0.01, format='csr',
...                   random_state=42)
>>> svd = TruncatedSVD(n_components=5, n_iter=7, random_state=42)
>>> svd.fit(X)
TruncatedSVD(n_components=5, n_iter=7, random_state=42)
>>> print(svd.explained_variance_ratio_)
[0.0646... 0.0633... 0.0639... 0.0535... 0.0406...]
>>> print(svd.explained_variance_ratio_.sum())
0.286...
>>> print(svd.singular_values_)
[1.553... 1.512...  1.510... 1.370... 1.199...]
```

Methods

fit(self, X[, y])	Fit LSI model on training data X.
fit_transform(self, X[, y])	Fit LSI model to X and perform dimensionality reduction on X.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Transform X back to its original space.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Perform dimensionality reduction on X.

__init__(*self*, *n_components=2*, *algorithm='randomized'*, *n_iter=5*, *random_state=None*, *tol=0.0*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit LSI model on training data X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data.

y [Ignored]

Returns

self [object] Returns the transformer object.

fit_transform(*self*, *X*, *y=None*)
Fit LSI model to X and perform dimensionality reduction on X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data.

y [Ignored]

Returns

X_new [array, shape (n_samples, n_components)] Reduced version of X. This will always be a dense array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*)
Transform X back to its original space.

Returns an array X_original whose transform would be X.

Parameters

X [array-like, shape (n_samples, n_components)] New data.

Returns

X_original [array, shape (n_samples, n_features)] Note that this is always a dense array.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Perform dimensionality reduction on X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] New data.

Returns

X_new [array, shape (n_samples, n_components)] Reduced version of X. This will always be a dense array.

Examples using `sklearn.decomposition.TruncatedSVD`

- *Hashing feature transformation using Totally Random Trees*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*
- *Column Transformer with Heterogeneous Data Sources*
- *Clustering text documents using k-means*

decomposition.dict_learning(X, n_components, ...)	Solves a dictionary learning matrix factorization problem.
decomposition.dict_learning_online(X[, ...])	Solves a dictionary learning matrix factorization problem online.
decomposition.fastica(X[, n_components, ...])	Perform Fast Independent Component Analysis.
decomposition.non_negative_factorization(X)	Compute Non-negative Matrix Factorization (NMF)
decomposition.sparse_encode(X, dictionary[, ...])	Sparse coding

7.8.14 `sklearn.decomposition`.dict_learning

sklearn.decomposition.**dict_learning**(*X*, *n_components*, *alpha*, *max_iter=100*, *tol=1e-08*, *method='lars'*, *n_jobs=None*, *dict_init=None*, *code_init=None*, *callback=None*, *verbose=False*, *random_state=None*, *return_n_iter=False*, *positive_dict=False*, *positive_code=False*, *method_max_iter=1000*)
Solves a dictionary learning matrix factorization problem.

Finds the best dictionary and the corresponding sparse code for approximating the data matrix X by solving:

```
(U^*, V^*) = argmin 0.5 || X - U V ||_2^2 + alpha * || U ||_1
             (U,V)
            with || V_k ||_2 = 1 for all  0 <= k < n_components
```

where V is the dictionary and U is the sparse code.

Read more in the *User Guide*.

Parameters

X [array of shape (n_samples, n_features)] Data matrix.

n_components [int,] Number of dictionary atoms to extract.

alpha [int,] Sparsity controlling parameter.

max_iter [int,] Maximum number of iterations to perform.

tol [float,] Tolerance for the stopping condition.

method [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

n_jobs [int or None, optional (default=None)] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

dict_init [array of shape (n_components, n_features),] Initial value for the dictionary for warm restart scenarios.

code_init [array of shape (n_samples, n_components),] Initial value for the sparse code for warm restart scenarios.

callback [callable or None, optional (default: None)] Callable that gets invoked every five iterations

verbose [bool, optional (default: False)] To control the verbosity of the procedure.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

return_n_iter [bool] Whether or not to return the number of iterations.

positive_dict [bool] Whether to enforce positivity when finding the dictionary.

New in version 0.20.

positive_code [bool] Whether to enforce positivity when finding the code.

New in version 0.20.

method_max_iter [int, optional (default=1000)] Maximum number of iterations to perform.

New in version 0.22.

Returns

code [array of shape (n_samples, n_components)] The sparse code factor in the matrix factorization.

dictionary [array of shape (n_components, n_features),] The dictionary factor in the matrix factorization.

errors [array] Vector of errors at each iteration.

n_iter [int] Number of iterations run. Returned only if `return_n_iter` is set to True.

See also:

dict_learning_online

DictionaryLearning

MiniBatchDictionaryLearning

SparsePCA

MiniBatchSparsePCA

7.8.15 sklearn.decomposition.dict_learning_online

sklearn.decomposition.**dict_learning_online**(*X*, *n_components=2*, *alpha=1*, *n_iter=100*, *return_code=True*, *dict_init=None*, *callback=None*, *batch_size=3*, *verbose=False*, *shuffle=True*, *n_jobs=None*, *method='lars'*, *iter_offset=0*, *random_state=None*, *return_inner_stats=False*, *inner_stats=None*, *return_n_iter=False*, *positive_dict=False*, *positive_code=False*, *method_max_iter=1000*)

Solves a dictionary learning matrix factorization problem online.

Finds the best dictionary and the corresponding sparse code for approximating the data matrix X by solving:

```
(U^*, V^*) = argmin 0.5 || X - U V ||_2^2 + alpha * || U ||_1
             (U,V)
             with || V_k ||_2 = 1 for all  0 <= k < n_components
```

where V is the dictionary and U is the sparse code. This is accomplished by repeatedly iterating over mini-batches by slicing the input data.

Read more in the *User Guide*.

Parameters

X [array of shape (n_samples, n_features)] Data matrix.

n_components [int,] Number of dictionary atoms to extract.

alpha [float,] Sparsity controlling parameter.

n_iter [int,] Number of mini-batch iterations to perform.

return_code [boolean,] Whether to also return the code U or just the dictionary V.

dict_init [array of shape (n_components, n_features),] Initial value for the dictionary for warm restart scenarios.

callback [callable or None, optional (default: None)] callable that gets invoked every five iterations

batch_size [int,] The number of samples to take in each batch.

verbose [bool, optional (default: False)] To control the verbosity of the procedure.

shuffle [boolean,] Whether to shuffle the data before splitting it in batches.

n_jobs [int or None, optional (default=None)] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

method [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

iter_offset [int, default 0] Number of previous iterations completed on the dictionary used for initialization.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

return_inner_stats [boolean, optional] Return the inner statistics A (dictionary covariance) and B (data approximation). Useful to restart the algorithm in an online setting. If return_inner_stats is True, return_code is ignored

inner_stats [tuple of (A, B) ndarrays] Inner sufficient statistics that are kept by the algorithm. Passing them at initialization is useful in online settings, to avoid losing the history of the evolution. A (n_components, n_components) is the dictionary covariance matrix. B (n_features, n_components) is the data approximation matrix

return_n_iter [bool] Whether or not to return the number of iterations.

positive_dict [bool] Whether to enforce positivity when finding the dictionary.

New in version 0.20.

positive_code [bool] Whether to enforce positivity when finding the code.

New in version 0.20.

method_max_iter [int, optional (default=1000)] Maximum number of iterations to perform when solving the lasso problem.

New in version 0.22.

Returns

code [array of shape (n_samples, n_components),] the sparse code (only returned if `return_code=True`)

dictionary [array of shape (n_components, n_features),] the solutions to the dictionary learning problem

n_iter [int] Number of iterations run. Returned only if `return_n_iter` is set to `True`.

See also:

dict_learning

DictionaryLearning

MiniBatchDictionaryLearning

SparsePCA

MiniBatchSparsePCA

7.8.16 sklearn.decomposition.fastica

sklearn.decomposition.**fastica**(*X*, *n_components=None*, *algorithm='parallel'*, *whiten=True*, *fun='logcosh'*, *fun_args=None*, *max_iter=200*, *tol=0.0001*, *w_init=None*, *random_state=None*, *return_X_mean=False*, *compute_sources=True*, *return_n_iter=False*)

Perform Fast Independent Component Analysis.

Read more in the *User Guide*.

Parameters

X [array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

n_components [int, optional] Number of components to extract. If None no dimension reduction is performed.

algorithm [{'parallel', 'deflation'}, optional] Apply a parallel or deflational FASTICA algorithm.

whiten [boolean, optional] If True perform an initial whitening of the data. If False, the data is assumed to have already been preprocessed: it should be centered, normed and white. Otherwise you will get incorrect results. In this case the parameter n_components will be ignored.

fun [string or function, optional. Default: 'logcosh'] The functional form of the G function used in the approximation to neg-entropy. Could be either 'logcosh', 'exp', or 'cube'. You can also provide your own function. It should return a tuple containing the value of the function, and of its derivative, in the point. The derivative should be averaged along its last dimension. Example:

def my_g(x): return x ** 3, np.mean(3 * x ** 2, axis=-1)

fun_args [dictionary, optional] Arguments to send to the functional form. If empty or None and if fun='logcosh', fun_args will take value {'alpha' : 1.0}

max_iter [int, optional] Maximum number of iterations to perform.

tol [float, optional] A positive scalar giving the tolerance at which the un-mixing matrix is considered to have converged.

w_init [(n_components, n_components) array, optional] Initial un-mixing array of dimension (n.comp,n.comp). If None (default) then an array of normal r.v.'s is used.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

return_X_mean [bool, optional] If True, X_mean is returned too.

compute_sources [bool, optional] If False, sources are not computed, but only the rotation matrix. This can save memory when working with big data. Defaults to True.

return_n_iter [bool, optional] Whether or not to return the number of iterations.

Returns

K [array, shape (n_components, n_features) | None.] If whiten is 'True', K is the pre-whitening matrix that projects data onto the first n_components principal components. If whiten is 'False', K is 'None'.

W [array, shape (n_components, n_components)] The square matrix that unmixes the data after whitening. The mixing matrix is the pseudo-inverse of matrix `W K` if K is not None, else it is the inverse of W.

S [array, shape (n_samples, n_components) | None] Estimated source matrix

X_mean [array, shape (n_features,)] The mean over features. Returned only if return_X_mean is True.

n_iter [int] If the algorithm is "deflation", n_iter is the maximum number of iterations run across all components. Else they are just the number of iterations taken to converge. This is returned only when return_n_iter is set to `True`.

Notes

The data matrix X is considered to be a linear combination of non-Gaussian (independent) components i.e. X = AS where columns of S contain the independent components and A is a linear mixing matrix. In short ICA

attempts to `un-mix' the data by estimating an un-mixing matrix W where ``S = W K X.`` While FastICA was proposed to estimate as many sources as features, it is possible to estimate less by setting n_components < n_features. It this case K is not a square matrix and the estimated A is the pseudo-inverse of `W K`.

This implementation was originally made for data of shape [n_features, n_samples]. Now the input is transposed before the algorithm is applied. This makes it slightly faster for Fortran-ordered input.

Implemented using FastICA: *A. Hyvarinen and E. Oja, Independent Component Analysis: Algorithms and Applications, Neural Networks, 13(4-5), 2000, pp. 411-430*

7.8.17 sklearn.decomposition.non_negative_factorization

sklearn.decomposition.**non_negative_factorization**(*X*, *W=None*, *H=None*, *n_components=None*, *init='warn'*, *update_H=True*, *solver='cd'*, *beta_loss='frobenius'*, *tol=0.0001*, *max_iter=200*, *alpha=0.0*, *l1_ratio=0.0*, *regularization=None*, *random_state=None*, *verbose=0*, *shuffle=False*)

Compute Non-negative Matrix Factorization (NMF)

Find two non-negative matrices (W, H) whose product approximates the non- negative matrix X. This factorization can be used for example for dimensionality reduction, source separation or topic extraction.

The objective function is:

```
0.5 * ||X - WH||_Fro^2
+ alpha * l1_ratio * ||vec(W)||_1
+ alpha * l1_ratio * ||vec(H)||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
+ 0.5 * alpha * (1 - l1_ratio) * ||H||_Fro^2
```

Where:

```
||A||_Fro^2 = \sum_{i,j} A_{ij}^2 (Frobenius norm)
||vec(A)||_1 = \sum_{i,j} abs(A_{ij}) (Elementwise L1 norm)
```

For multiplicative-update ('mu') solver, the Frobenius norm (0.5 * ||X - WH||_Fro^2) can be changed into another beta-divergence loss, by changing the beta_loss parameter.

The objective function is minimized with an alternating minimization of W and H. If H is given and update_H=False, it solves for W only.

Parameters

X [array-like, shape (n_samples, n_features)] Constant matrix.

W [array-like, shape (n_samples, n_components)] If init='custom', it is used as initial guess for the solution.

H [array-like, shape (n_components, n_features)] If init='custom', it is used as initial guess for the solution. If update_H=False, it is used as a constant, to solve for W only.

n_components [integer] Number of components, if n_components is not set all features are kept.

init [None | 'random' | 'nndsvd' | 'nndsvda' | 'nndsvdar' | 'custom'] Method used to initialize the procedure. Default: 'random'.

The default value will change from 'random' to None in version 0.23 to make it consistent with decomposition.NMF.

Valid options:

- None: 'nndsvd' if n_components < n_features, otherwise 'random'.
- **'random': non-negative random matrices, scaled with:** sqrt(X.mean() / n_components)
- **'nndsvd': Nonnegative Double Singular Value Decomposition (NNDSVD)** initialization (better for sparseness)
- **'nndsvda': NNDSVD with zeros filled with the average of X** (better when sparsity is not desired)
- **'nndsvdar': NNDSVD with zeros filled with small random values** (generally faster, less accurate alternative to NNDSVDa for when sparsity is not desired)
- 'custom': use custom matrices W and H

update_H [boolean, default: True] Set to True, both W and H will be estimated from initial guesses. Set to False, only W will be estimated.

solver ['cd' | 'mu'] Numerical solver to use:

- **'cd' is a Coordinate Descent solver that uses Fast Hierarchical** Alternating Least Squares (Fast HALS).
- 'mu' is a Multiplicative Update solver.

New in version 0.17: Coordinate Descent solver.

New in version 0.19: Multiplicative Update solver.

beta_loss [float or string, default 'frobenius'] String must be in {'frobenius', 'kullback-leibler', 'itakura-saito'}. Beta divergence to be minimized, measuring the distance between X and the dot product WH. Note that values different from 'frobenius' (or 2) and 'kullback-leibler' (or 1) lead to significantly slower fits. Note that for beta_loss <= 0 (or 'itakura-saito'), the input matrix X cannot contain zeros. Used only in 'mu' solver.

New in version 0.19.

tol [float, default: 1e-4] Tolerance of the stopping condition.

max_iter [integer, default: 200] Maximum number of iterations before timing out.

alpha [double, default: 0.] Constant that multiplies the regularization terms.

l1_ratio [double, default: 0.] The regularization mixing parameter, with 0 <= l1_ratio <= 1. For l1_ratio = 0 the penalty is an elementwise L2 penalty (aka Frobenius Norm). For l1_ratio = 1 it is an elementwise L1 penalty. For 0 < l1_ratio < 1, the penalty is a combination of L1 and L2.

regularization ['both' | 'components' | 'transformation' | None] Select whether the regularization affects the components (H), the transformation (W), both or none of them.

random_state [int, RandomState instance or None, optional, default: None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

verbose [integer, default: 0] The verbosity level.

shuffle [boolean, default: False] If true, randomize the order of coordinates in the CD solver.

Returns

W [array-like, shape (n_samples, n_components)] Solution to the non-negative least squares problem.

H [array-like, shape (n_components, n_features)] Solution to the non-negative least squares problem.

n_iter [int] Actual number of iterations.

References

Cichocki, Andrzej, and P. H. A. N. Anh-Huy. "Fast local algorithms for large scale nonnegative matrix and tensor factorizations." IEICE transactions on fundamentals of electronics, communications and computer sciences 92.3: 708-721, 2009.

Fevotte, C., & Idier, J. (2011). Algorithms for nonnegative matrix factorization with the beta-divergence. Neural Computation, 23(9).

Examples

```
>>> import numpy as np
>>> X = np.array([[1,1], [2, 1], [3, 1.2], [4, 1], [5, 0.8], [6, 1]])
>>> from sklearn.decomposition import non_negative_factorization
>>> W, H, n_iter = non_negative_factorization(X, n_components=2,
... init='random', random_state=0)
```

7.8.18 sklearn.decomposition.sparse_encode

sklearn.decomposition.**sparse_encode**(*X*, *dictionary*, *gram=None*, *cov=None*, *algorithm='lasso_lars'*, *n_nonzero_coefs=None*, *alpha=None*, *copy_cov=True*, *init=None*, *max_iter=1000*, *n_jobs=None*, *check_input=True*, *verbose=0*, *positive=False*)

Sparse coding

Each row of the result is the solution to a sparse coding problem. The goal is to find a sparse array `code` such that:

```
X ~= code * dictionary
```

Read more in the *User Guide*.

Parameters

X [array of shape (n_samples, n_features)] Data matrix

dictionary [array of shape (n_components, n_features)] The dictionary matrix against which to solve the sparse coding of the data. Some of the algorithms assume normalized rows for meaningful output.

gram [array, shape=(n_components, n_components)] Precomputed Gram matrix, dictionary * dictionary'

cov [array, shape=(n_components, n_samples)] Precomputed covariance, dictionary' * X

algorithm [{'lasso_lars', 'lasso_cd', 'lars', 'omp', 'threshold'}] lars: uses the least angle regression method (linear_model.lars_path) lasso_lars: uses Lars to compute the Lasso solution lasso_cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). lasso_lars will be faster if the estimated components are sparse. omp: uses orthogonal matching pursuit to estimate the sparse solution threshold: squashes to zero all coefficients less than alpha from the projection dictionary * X'

n_nonzero_coefs [int, 0.1 * n_features by default] Number of nonzero coefficients to target in each column of the solution. This is only used by `algorithm='lars'` and `algorithm='omp'` and is overridden by `alpha` in the `omp` case.

alpha [float, 1. by default] If `algorithm='lasso_lars'` or `algorithm='lasso_cd'`, `alpha` is the penalty applied to the L1 norm. If `algorithm='threshold'`, `alpha` is the absolute value of the threshold below which coefficients will be squashed to zero. If `algorithm='omp'`, `alpha` is the tolerance parameter: the value of the reconstruction error targeted. In this case, it overrides `n_nonzero_coefs`.

copy_cov [boolean, optional] Whether to copy the precomputed covariance matrix; if False, it may be overwritten.

init [array of shape (n_samples, n_components)] Initialization value of the sparse codes. Only used if `algorithm='lasso_cd'`.

max_iter [int, 1000 by default] Maximum number of iterations to perform if `algorithm='lasso_cd'` or `lasso_lars`.

n_jobs [int or None, optional (default=None)] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

check_input [boolean, optional] If False, the input arrays X and dictionary will not be checked.

verbose [int, optional] Controls the verbosity; the higher, the more messages. Defaults to 0.

positive [boolean, optional] Whether to enforce positivity when finding the encoding.

New in version 0.20.

Returns

code [array of shape (n_samples, n_components)] The sparse codes

See also:

sklearn.linear_model.lars_path

sklearn.linear_model.orthogonal_mp

sklearn.linear_model.Lasso

SparseCoder

7.9 `sklearn.discriminant_analysis`: Discriminant Analysis

Linear Discriminant Analysis and Quadratic Discriminant Analysis

User guide: See the *Linear and Quadratic Discriminant Analysis* section for further details.

discriminant_analysis.LinearDiscriminantAnalysis([...])	Linear Discriminant Analysis
discriminant_analysis.QuadraticDiscriminantAnalysis([...])	Quadratic Discriminant Analysis

7.9.1 sklearn.discriminant_analysis.LinearDiscriminantAnalysis

class sklearn.discriminant_analysis.**LinearDiscriminantAnalysis**(*solver='svd'*, *shrinkage=None*, *priors=None*, *n_components=None*, *store_covariance=False*, *tol=0.0001*)

Linear Discriminant Analysis

A classifier with a linear decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule.

The model fits a Gaussian density to each class, assuming that all classes share the same covariance matrix.

The fitted model can also be used to reduce the dimensionality of the input by projecting it to the most discriminative directions.

New in version 0.17: *LinearDiscriminantAnalysis*.

Read more in the *User Guide*.

Parameters

solver [string, optional]

Solver to use, possible values:

- 'svd': Singular value decomposition (default). Does not compute the covariance matrix, therefore this solver is recommended for data with a large number of features.
- 'lsqr': Least squares solution, can be combined with shrinkage.
- 'eigen': Eigenvalue decomposition, can be combined with shrinkage.

shrinkage [string or float, optional]

Shrinkage parameter, possible values:

- None: no shrinkage (default).
- 'auto': automatic shrinkage using the Ledoit-Wolf lemma.
- float between 0 and 1: fixed shrinkage parameter.

Note that shrinkage works only with 'lsqr' and 'eigen' solvers.

priors [array, optional, shape (n_classes,)] Class priors.

n_components [int, optional (default=None)] Number of components (<= min(n_classes - 1, n_features)) for dimensionality reduction. If None, will be set to min(n_classes - 1, n_features).

store_covariance [bool, optional] Additionally compute class covariance matrix (default False), used only in 'svd' solver.

New in version 0.17.

tol [float, optional, (default 1.0e-4)] Threshold used for rank estimation in SVD solver.

New in version 0.17.

Attributes

coef_ [array, shape (n_features,) or (n_classes, n_features)] Weight vector(s).

intercept_ [array, shape (n_classes,)] Intercept term.

covariance_ [array-like, shape (n_features, n_features)] Covariance matrix (shared by all classes).

explained_variance_ratio_ [array, shape (n_components,)] Percentage of variance explained by each of the selected components. If `n_components` is not set then all components are stored and the sum of explained variances is equal to 1.0. Only available when eigen or svd solver is used.

means_ [array-like, shape (n_classes, n_features)] Class means.

priors_ [array-like, shape (n_classes,)] Class priors (sum to 1).

scalings_ [array-like, shape (rank, n_classes - 1)] Scaling of the features in the space spanned by the class centroids.

xbar_ [array-like, shape (n_features,)] Overall mean.

classes_ [array-like, shape (n_classes,)] Unique class labels.

See also:

sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis Quadratic Discriminant Analysis

Notes

The default solver is 'svd'. It can perform both classification and transform, and it does not rely on the calculation of the covariance matrix. This can be an advantage in situations where the number of features is large. However, the 'svd' solver cannot be used with shrinkage.

The 'lsqr' solver is an efficient algorithm that only works for classification. It supports shrinkage.

The 'eigen' solver is based on the optimization of the between class scatter to within class scatter ratio. It can be used for both classification and transform, and it supports shrinkage. However, the 'eigen' solver needs to compute the covariance matrix, so it might not be suitable for situations with a high number of features.

Examples

```
>>> import numpy as np
>>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> y = np.array([1, 1, 1, 2, 2, 2])
>>> clf = LinearDiscriminantAnalysis()
>>> clf.fit(X, y)
LinearDiscriminantAnalysis()
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
fit(self, X, y)	Fit LinearDiscriminantAnalysis model according to the given
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class labels for samples in X.
predict_log_proba(self, X)	Estimate log probability.
predict_proba(self, X)	Estimate probability.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Project data to maximize class separation.

__init__(*self*, *solver='svd'*, *shrinkage=None*, *priors=None*, *n_components=None*, *store_covariance=False*, *tol=0.0001*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

fit(*self*, *X*, *y*)

Fit LinearDiscriminantAnalysis model according to the given training data and parameters.

Changed in version 0.19: *store_covariance* has been moved to main constructor.

Changed in version 0.19: *tol* has been moved to main constructor.

Parameters

X [array-like, shape (n_samples, n_features)] Training data.

y [array, shape (n_samples,)] Target values.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

predict_log_proba(*self*, *X*)
Estimate log probability.

Parameters

X [array-like, shape (n_samples, n_features)] Input data.

Returns

C [array, shape (n_samples, n_classes)] Estimated log probabilities.

predict_proba(*self*, *X*)
Estimate probability.

Parameters

X [array-like, shape (n_samples, n_features)] Input data.

Returns

C [array, shape (n_samples, n_classes)] Estimated probabilities.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)

Project data to maximize class separation.

Parameters

X [array-like, shape (n_samples, n_features)] Input data.

Returns

X_new [array, shape (n_samples, n_components)] Transformed data.

Examples using `sklearn.discriminant_analysis.LinearDiscriminantAnalysis`

- *Normal and Shrinkage Linear Discriminant Analysis for classification*
- *Linear and Quadratic Discriminant Analysis with covariance ellipsoid*
- *Comparison of LDA and PCA 2D projection of Iris dataset*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*
- *Dimensionality Reduction with Neighborhood Components Analysis*

7.9.2 `sklearn.discriminant_analysis`.QuadraticDiscriminantAnalysis

class sklearn.discriminant_analysis.**QuadraticDiscriminantAnalysis**(*priors=None*, *reg_param=0.0*, *store_covariance=False*, *tol=0.0001*)

Quadratic Discriminant Analysis

A classifier with a quadratic decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule.

The model fits a Gaussian density to each class.

New in version 0.17: *QuadraticDiscriminantAnalysis*

Read more in the *User Guide*.

Parameters

priors [array, optional, shape = [n_classes]] Priors on classes

reg_param [float, optional] Regularizes the covariance estimate as `(1-reg_param)*Sigma + reg_param*np.eye(n_features)`

store_covariance [boolean] If True the covariance matrices are computed and stored in the `self.covariance_` attribute.

New in version 0.17.

tol [float, optional, default 1.0e-4] Threshold used for rank estimation.

New in version 0.17.

Attributes

covariance_ [list of array-like of shape (n_features, n_features)] Covariance matrices of each class.

means_ [array-like of shape (n_classes, n_features)] Class means.

priors_ [array-like of shape (n_classes)] Class priors (sum to 1).

rotations_ [list of arrays] For each class k an array of shape [n_features, n_k], with `n_k = min(n_features, number of elements in class k)` It is the rotation of the Gaussian distribution, i.e. its principal axis.

scalings_ [list of arrays] For each class k an array of shape [n_k]. It contains the scaling of the Gaussian distributions along its principal axes, i.e. the variance in the rotated coordinate system.

classes_ [array-like, shape (n_classes,)] Unique class labels.

See also:

sklearn.discriminant_analysis.LinearDiscriminantAnalysis Linear Discriminant Analysis

Examples

```
>>> from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> y = np.array([1, 1, 1, 2, 2, 2])
>>> clf = QuadraticDiscriminantAnalysis()
>>> clf.fit(X, y)
QuadraticDiscriminantAnalysis()
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

decision_function(self, X)	Apply decision function to an array of samples.
fit(self, X, y)	Fit the model according to the given training data and parameters.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Perform classification on an array of test vectors X.
predict_log_proba(self, X)	Return posterior probabilities of classification.
predict_proba(self, X)	Return posterior probabilities of classification.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *priors=None*, *reg_param=0.0*, *store_covariance=False*, *tol=0.0001*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Apply decision function to an array of samples.

Parameters

X [array-like of shape (n_samples, n_features)] Array of samples (test vectors).

Returns

C [ndarray of shape (n_samples,) or (n_samples, n_classes)] Decision function values related to each class, per sample. In the two-class case, the shape is [n_samples,], giving the log likelihood ratio of the positive class.

fit(*self*, *X*, *y*)
Fit the model according to the given training data and parameters.

Changed in version 0.19: `store_covariances` has been moved to main constructor as `store_covariance`

Changed in version 0.19: `tol` has been moved to main constructor.

Parameters

X [array-like of shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y [array, shape = [n_samples]] Target values (integers)

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Perform classification on an array of test vectors X.

The predicted class C for each sample in X is returned.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [ndarray of shape (n_samples,)]

predict_log_proba(*self*, *X*)
Return posterior probabilities of classification.

Parameters

X [array-like of shape (n_samples, n_features)] Array of samples/test vectors.

Returns

C [ndarray of shape (n_samples, n_classes)] Posterior log-probabilities of classification per class.

predict_proba(*self*, *X*)
Return posterior probabilities of classification.

Parameters

X [array-like of shape (n_samples, n_features)] Array of samples/test vectors.

Returns

C [ndarray of shape (n_samples, n_classes)] Posterior probabilities of classification per class.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis`

- *Classifier comparison*
- *Linear and Quadratic Discriminant Analysis with covariance ellipsoid*

7.10 `sklearn.dummy`: Dummy estimators

User guide: See the *Metrics and scoring: quantifying the quality of predictions* section for further details.

dummy.DummyClassifier([strategy, ...])	DummyClassifier is a classifier that makes predictions using simple rules.
dummy.DummyRegressor([strategy, constant, ...])	DummyRegressor is a regressor that makes predictions using simple rules.

7.10.1 sklearn.dummy.DummyClassifier

class sklearn.dummy.**DummyClassifier**(*strategy='warn'*, *random_state=None*, *constant=None*)

DummyClassifier is a classifier that makes predictions using simple rules.

This classifier is useful as a simple baseline to compare with other (real) classifiers. Do not use it for real problems.

Read more in the *User Guide*.

New in version 0.13.

Parameters

strategy [str, default="stratified"] Strategy to use to generate predictions.

- "stratified": generates predictions by respecting the training set's class distribution.
- "most_frequent": always predicts the most frequent label in the training set.
- "prior": always predicts the class that maximizes the class prior (like "most_frequent") and `predict_proba` returns the class prior.
- "uniform": generates predictions uniformly at random.
- "constant": always predicts a constant label that is provided by the user. This is useful for metrics that evaluate a non-majority class

  Changed in version 0.22: The default value of `strategy` will change to "prior" in version 0.24. Starting from version 0.22, a warning will be raised if `strategy` is not explicitly set.

  New in version 0.17: Dummy Classifier now supports prior fitting strategy using parameter *prior*.

random_state [int, RandomState instance or None, optional, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

constant [int or str or array-like of shape (n_outputs,)] The explicit constant as predicted by the "constant" strategy. This parameter is useful only for the "constant" strategy.

Attributes

classes_ [array or list of array of shape (n_classes,)] Class labels for each output.

n_classes_ [array or list of array of shape (n_classes,)] Number of label for each output.

class_prior_ [array or list of array of shape (n_classes,)] Probability of each class for each output.

n_outputs_ [int,] Number of outputs.

sparse_output_ [bool,] True if the array returned from predict is to be in sparse CSC format. Is automatically set to True if the input y is passed in sparse format.

Examples

```
>>> import numpy as np
>>> from sklearn.dummy import DummyClassifier
>>> X = np.array([-1, 1, 1, 1])
```

(continues on next page)

(continued from previous page)

```
>>> y = np.array([0, 1, 1, 1])
>>> dummy_clf = DummyClassifier(strategy="most_frequent")
>>> dummy_clf.fit(X, y)
DummyClassifier(strategy='most_frequent')
>>> dummy_clf.predict(X)
array([1, 1, 1, 1])
>>> dummy_clf.score(X, y)
0.75
```

Methods

fit(self, X, y[, sample_weight])	Fit the random classifier.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Perform classification on test vectors X.
predict_log_proba(self, X)	Return log probability estimates for the test vectors X.
predict_proba(self, X)	Return probability estimates for the test vectors X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *strategy='warn'*, *random_state=None*, *constant=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit the random classifier.

Parameters

X [{array-like, object with finite length or shape}] Training data, requires length = n_samples

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] Target values.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Perform classification on test vectors X.

Parameters

X [{array-like, object with finite length or shape}] Training data, requires length = n_samples

Returns

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] Predicted target values for X.

predict_log_proba(*self*, *X*)
Return log probability estimates for the test vectors X.

Parameters

X [{array-like, object with finite length or shape}] Training data, requires length = n_samples

Returns

P [array-like or list of array-like of shape (n_samples, n_classes)] Returns the log probability of the sample for each class in the model, where classes are ordered arithmetically for each output.

predict_proba(*self*, *X*)
Return probability estimates for the test vectors X.

Parameters

X [{array-like, object with finite length or shape}] Training data, requires length = n_samples

Returns

P [array-like or list of array-lke of shape (n_samples, n_classes)] Returns the probability of the sample for each class in the model, where classes are ordered arithmetically, for each output.

score(*self*, *X*, *y*, *sample_weight=None*)
Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [{array-like, None}] Test samples with shape = (n_samples, n_features) or None. Passing None as test samples gives the same result as passing real test samples, since DummyClassifier operates independently of the sampled observations.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

7.10.2 sklearn.dummy.DummyRegressor

class sklearn.dummy.**DummyRegressor**(*strategy='mean'*, *constant=None*, *quantile=None*)

DummyRegressor is a regressor that makes predictions using simple rules.

This regressor is useful as a simple baseline to compare with other (real) regressors. Do not use it for real problems.

Read more in the *User Guide*.

New in version 0.13.

Parameters

strategy [str] Strategy to use to generate predictions.

- "mean": always predicts the mean of the training set
- "median": always predicts the median of the training set
- "quantile": always predicts a specified quantile of the training set, provided with the quantile parameter.
- "constant": always predicts a constant value that is provided by the user.

constant [int or float or array-like of shape (n_outputs,)] The explicit constant as predicted by the "constant" strategy. This parameter is useful only for the "constant" strategy.

quantile [float in [0.0, 1.0]] The quantile to predict using the "quantile" strategy. A quantile of 0.5 corresponds to the median, while 0.0 to the minimum and 1.0 to the maximum.

Attributes

constant_ [array, shape (1, n_outputs)] Mean or median or quantile of the training targets or constant value given by the user.

n_outputs_ [int,] Number of outputs.

Examples

```
>>> import numpy as np
>>> from sklearn.dummy import DummyRegressor
>>> X = np.array([1.0, 2.0, 3.0, 4.0])
>>> y = np.array([2.0, 3.0, 5.0, 10.0])
>>> dummy_regr = DummyRegressor(strategy="mean")
>>> dummy_regr.fit(X, y)
DummyRegressor()
>>> dummy_regr.predict(X)
array([5., 5., 5., 5.])
>>> dummy_regr.score(X, y)
0.0
```

Methods

fit(self, X, y[, sample_weight])	Fit the random regressor.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, return_std])	Perform classification on test vectors X.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *strategy='mean'*, *constant=None*, *quantile=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit the random regressor.

Parameters

X [{array-like, object with finite length or shape}] Training data, requires length = n_samples

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] Target values.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*, *return_std=False*)
Perform classification on test vectors X.

Parameters

X [{array-like, object with finite length or shape}] Training data, requires length = n_samples

return_std [boolean, optional] Whether to return the standard deviation of posterior prediction. All zeros in this case.

Returns

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] Predicted target values for X.

y_std [array-like of shape (n_samples,) or (n_samples, n_outputs)] Standard deviation of predictive distribution of query points.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [{array-like, None}] Test samples with shape = (n_samples, n_features) or None. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted], where n_samples_fitted is the number of samples used in the fitting for the estimator. Passing None as test samples gives the same result as passing real test samples, since DummyRegressor operates independently of the sampled observations.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

7.11 `sklearn.ensemble`: Ensemble Methods

The *sklearn.ensemble* module includes ensemble-based methods for classification, regression and anomaly detection.

User guide: See the *Ensemble methods* section for further details.

ensemble.AdaBoostClassifier([...])	An AdaBoost classifier.
ensemble.AdaBoostRegressor([base_estimator, ...])	An AdaBoost regressor.
ensemble.BaggingClassifier([base_estimator, ...])	A Bagging classifier.
ensemble.BaggingRegressor([base_estimator, ...])	A Bagging regressor.
ensemble.ExtraTreesClassifier([...])	An extra-trees classifier.
ensemble.ExtraTreesRegressor([n_estimators, ...])	An extra-trees regressor.
ensemble.GradientBoostingClassifier([loss, ...])	Gradient Boosting for classification.
ensemble.GradientBoostingRegressor([loss, ...])	Gradient Boosting for regression.
ensemble.IsolationForest([n_estimators, ...])	Isolation Forest Algorithm.
ensemble.RandomForestClassifier([...])	A random forest classifier.

Continued on next page

Table 70 – continued from previous page

ensemble.RandomForestRegressor([...])	A random forest regressor.
ensemble.RandomTreesEmbedding([...])	An ensemble of totally random trees.
ensemble.StackingClassifier(estimators[, ...])	Stack of estimators with a final classifier.
ensemble.StackingRegressor(estimators[, ...])	Stack of estimators with a final regressor.
ensemble.VotingClassifier(estimators[, ...])	Soft Voting/Majority Rule classifier for unfitted estimators.
ensemble.VotingRegressor(estimators[, ...])	Prediction voting regressor for unfitted estimators.
ensemble.HistGradientBoostingRegressor([...])	Histogram-based Gradient Boosting Regression Tree.
ensemble.HistGradientBoostingClassifier([...])	Histogram-based Gradient Boosting Classification Tree.

7.11.1 sklearn.ensemble.AdaBoostClassifier

class sklearn.ensemble.**AdaBoostClassifier**(*base_estimator=None*, *n_estimators=50*, *learning_rate=1.0*, *algorithm='SAMME.R'*, *random_state=None*)

An AdaBoost classifier.

An AdaBoost [1] classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

This class implements the algorithm known as AdaBoost-SAMME [2].

Read more in the *User Guide*.

New in version 0.14.

Parameters

base_estimator [object, optional (default=None)] The base estimator from which the boosted ensemble is built. Support for sample weighting is required, as well as proper `classes_` and `n_classes_` attributes. If `None`, then the base estimator is `DecisionTreeClassifier(max_depth=1)`.

n_estimators [int, optional (default=50)] The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early.

learning_rate [float, optional (default=1.)] Learning rate shrinks the contribution of each classifier by `learning_rate`. There is a trade-off between `learning_rate` and `n_estimators`.

algorithm [{'SAMME', 'SAMME.R'}, optional (default='SAMME.R')] If 'SAMME.R' then use the SAMME.R real boosting algorithm. `base_estimator` must support calculation of class probabilities. If 'SAMME' then use the SAMME discrete boosting algorithm. The SAMME.R algorithm typically converges faster than SAMME, achieving a lower test error with fewer boosting iterations.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Attributes

base_estimator_ [estimator] The base estimator from which the ensemble is grown.

estimators_ [list of classifiers] The collection of fitted sub-estimators.

classes_ [array of shape (n_classes,)] The classes labels.

n_classes_ [int] The number of classes.

estimator_weights_ [array of floats] Weights for each estimator in the boosted ensemble.

estimator_errors_ [array of floats] Classification error for each estimator in the boosted ensemble.

feature_importances_ [ndarray of shape (n_features,)] Return the feature importances (the higher, the more important the feature).

See also:

AdaBoostRegressor An AdaBoost regressor that begins by fitting a regressor on the original dataset and then fits additional copies of the regressor on the same dataset but where the weights of instances are adjusted according to the error of the current prediction.

GradientBoostingClassifier GB builds an additive model in a forward stage-wise fashion. Regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced.

sklearn.tree.DecisionTreeClassifier A non-parametric supervised learning method used for classification. Creates a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

References

[R33e4ec8c4ad5-1], [R33e4ec8c4ad5-2]

Examples

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=1000, n_features=4,
...                            n_informative=2, n_redundant=0,
...                            random_state=0, shuffle=False)
>>> clf = AdaBoostClassifier(n_estimators=100, random_state=0)
>>> clf.fit(X, y)
AdaBoostClassifier(n_estimators=100, random_state=0)
>>> clf.feature_importances_
array([0.28..., 0.42..., 0.14..., 0.16...])
>>> clf.predict([[0, 0, 0, 0]])
array([1])
>>> clf.score(X, y)
0.983...
```

Methods

decision_function(self, X)	Compute the decision function of X.
fit(self, X, y[, sample_weight])	Build a boosted classifier from the training set (X, y).
get_params(self[, deep])	Get parameters for this estimator.

Continued on next page

Table 71 – continued from previous page

predict(self, X)	Predict classes for X.
predict_log_proba(self, X)	Predict class log-probabilities for X.
predict_proba(self, X)	Predict class probabilities for X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.
staged_decision_function(self, X)	Compute decision function of X for each boosting iteration.
staged_predict(self, X)	Return staged predictions for X.
staged_predict_proba(self, X)	Predict class probabilities for X.
staged_score(self, X, y[, sample_weight])	Return staged scores for X, y.

__init__(*self*, *base_estimator=None*, *n_estimators=50*, *learning_rate=1.0*, *algorithm='SAMME.R'*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Compute the decision function of X.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Returns

score [array, shape = [n_samples, k]] The decision function of the input samples. The order of outputs is the same of that of the *classes_* attribute. Binary classification is a special cases with `k == 1`, otherwise `k==n_classes`. For binary classification, values closer to -1 or 1 mean more like the first or second class in `classes_`, respectively.

property feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [ndarray of shape (n_features,)] The feature importances.

fit(*self*, *X*, *y*, *sample_weight=None*)
Build a boosted classifier from the training set (X, y).

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

y [array-like of shape (n_samples,)] The target values (class labels).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, the sample weights are initialized to `1 / n_samples`.

Returns

self [object] Fitted estimator.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)

Predict classes for X.

The predicted class of an input sample is computed as the weighted mean prediction of the classifiers in the ensemble.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Returns

y [ndarray of shape (n_samples,)] The predicted classes.

predict_log_proba (*self*, *X*)

Predict class log-probabilities for X.

The predicted class log-probabilities of an input sample is computed as the weighted mean predicted class log-probabilities of the classifiers in the ensemble.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Returns

p [array of shape (n_samples, n_classes)] The class probabilities of the input samples. The order of outputs is the same of that of the *classes_* attribute.

predict_proba (*self*, *X*)

Predict class probabilities for X.

The predicted class probabilities of an input sample is computed as the weighted mean predicted class probabilities of the classifiers in the ensemble.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Returns

p [array of shape (n_samples, n_classes)] The class probabilities of the input samples. The order of outputs is the same of that of the *classes_* attribute.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

staged_decision_function(*self*, *X*)
Compute decision function of `X` for each boosting iteration.

This method allows monitoring (i.e. determine error on testing set) after each boosting iteration.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Yields

score [generator of array, shape = [n_samples, k]] The decision function of the input samples. The order of outputs is the same of that of the *classes_* attribute. Binary classification is a special cases with `k == 1`, otherwise `k==n_classes`. For binary classification, values closer to -1 or 1 mean more like the first or second class in `classes_`, respectively.

staged_predict(*self*, *X*)
Return staged predictions for X.

The predicted class of an input sample is computed as the weighted mean prediction of the classifiers in the ensemble.

This generator method yields the ensemble prediction after each iteration of boosting and therefore allows monitoring, such as to determine the prediction on a test set after each boost.

Parameters

X [array-like of shape (n_samples, n_features)] The input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Yields

y [generator of array, shape = [n_samples]] The predicted classes.

staged_predict_proba(*self*, *X*)
Predict class probabilities for X.

The predicted class probabilities of an input sample is computed as the weighted mean predicted class probabilities of the classifiers in the ensemble.

This generator method yields the ensemble predicted class probabilities after each iteration of boosting and therefore allows monitoring, such as to determine the predicted class probabilities on a test set after each boost.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Yields

p [generator of array, shape = [n_samples]] The class probabilities of the input samples. The order of outputs is the same of that of the *classes_* attribute.

staged_score(*self*, *X*, *y*, *sample_weight=None*)

Return staged scores for X, y.

This generator method yields the ensemble score after each iteration of boosting and therefore allows monitoring, such as to determine the score on a test set after each boost.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

y [array-like of shape (n_samples,)] Labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Yields

z [float]

Examples using `sklearn.ensemble.AdaBoostClassifier`

- *Classifier comparison*
- *Two-class AdaBoost*
- *Multi-class AdaBoosted Decision Trees*
- *Discrete versus Real AdaBoost*
- *Plot the decision surfaces of ensembles of trees on the iris dataset*

7.11.2 sklearn.ensemble.AdaBoostRegressor

class sklearn.ensemble.**AdaBoostRegressor**(*base_estimator=None*, *n_estimators=50*, *learning_rate=1.0*, *loss='linear'*, *random_state=None*)

An AdaBoost regressor.

An AdaBoost [1] regressor is a meta-estimator that begins by fitting a regressor on the original dataset and then fits additional copies of the regressor on the same dataset but where the weights of instances are adjusted according to the error of the current prediction. As such, subsequent regressors focus more on difficult cases.

This class implements the algorithm known as AdaBoost.R2 [2].

Read more in the *User Guide*.

New in version 0.14.

Parameters

base_estimator [object, optional (default=None)] The base estimator from which the boosted ensemble is built. If `None`, then the base estimator is `DecisionTreeRegressor(max_depth=3)`.

n_estimators [integer, optional (default=50)] The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early.

learning_rate [float, optional (default=1.)] Learning rate shrinks the contribution of each regressor by `learning_rate`. There is a trade-off between `learning_rate` and `n_estimators`.

loss [{'linear', 'square', 'exponential'}, optional (default='linear')] The loss function to use when updating the weights after each boosting iteration.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Attributes

base_estimator_ [estimator] The base estimator from which the ensemble is grown.

estimators_ [list of classifiers] The collection of fitted sub-estimators.

estimator_weights_ [array of floats] Weights for each estimator in the boosted ensemble.

estimator_errors_ [array of floats] Regression error for each estimator in the boosted ensemble.

feature_importances_ [ndarray of shape (n_features,)] Return the feature importances (the higher, the more important the feature).

See also:

AdaBoostClassifier, ***GradientBoostingRegressor***

sklearn.tree.DecisionTreeRegressor

References

[R0c261b7dee9d-1], [R0c261b7dee9d-2]

Examples

```
>>> from sklearn.ensemble import AdaBoostRegressor
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(n_features=4, n_informative=2,
...                        random_state=0, shuffle=False)
>>> regr = AdaBoostRegressor(random_state=0, n_estimators=100)
>>> regr.fit(X, y)
AdaBoostRegressor(n_estimators=100, random_state=0)
>>> regr.feature_importances_
array([0.2788..., 0.7109..., 0.0065..., 0.0036...])
>>> regr.predict([[0, 0, 0, 0]])
array([4.7972...])
```

(continues on next page)

(continued from previous page)

```
>>> regr.score(X, y)
0.9771...
```

Methods

fit(self, X, y[, sample_weight])	Build a boosted regressor from the training set (X, y).
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict regression value for X.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.
staged_predict(self, X)	Return staged predictions for X.
staged_score(self, X, y[, sample_weight])	Return staged scores for X, y.

__init__(*self*, *base_estimator=None*, *n_estimators=50*, *learning_rate=1.0*, *loss='linear'*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

property feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [ndarray of shape (n_features,)] The feature importances.

fit(*self*, *X*, *y*, *sample_weight=None*)
Build a boosted regressor from the training set (X, y).

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

y [array-like of shape (n_samples,)] The target values (real numbers).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, the sample weights are initialized to 1 / n_samples.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict regression value for X.

The predicted regression value of an input sample is computed as the weighted median prediction of the classifiers in the ensemble.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Returns

y [ndarray of shape (n_samples,)] The predicted regression values.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

staged_predict(*self*, *X*)
Return staged predictions for X.

The predicted regression value of an input sample is computed as the weighted median prediction of the classifiers in the ensemble.

This generator method yields the ensemble prediction after each iteration of boosting and therefore allows monitoring, such as to determine the prediction on a test set after each boost.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples.

Yields

y [generator of array, shape = [n_samples]] The predicted regression values.

staged_score(*self*, *X*, *y*, *sample_weight=None*)
Return staged scores for X, y.

This generator method yields the ensemble score after each iteration of boosting and therefore allows monitoring, such as to determine the score on a test set after each boost.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

y [array-like of shape (n_samples,)] Labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Yields

z [float]

Examples using sklearn.ensemble.AdaBoostRegressor

- *Decision Tree Regression with AdaBoost*

7.11.3 sklearn.ensemble.BaggingClassifier

class sklearn.ensemble.**BaggingClassifier**(*base_estimator=None*, *n_estimators=10*, *max_samples=1.0*, *max_features=1.0*, *bootstrap=True*, *bootstrap_features=False*, *oob_score=False*, *warm_start=False*, *n_jobs=None*, *random_state=None*, *verbose=0*)

A Bagging classifier.

A Bagging classifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

This algorithm encompasses several works from the literature. When random subsets of the dataset are drawn as random subsets of the samples, then this algorithm is known as Pasting [Rb1846455d0e5-1]. If samples are drawn with replacement, then the method is known as Bagging [Rb1846455d0e5-2]. When random subsets of the dataset are drawn as random subsets of the features, then the method is known as Random Subspaces

[Rb1846455d0e5-3]. Finally, when base estimators are built on subsets of both samples and features, then the method is known as Random Patches [Rb1846455d0e5-4].

Read more in the *User Guide*.

New in version 0.15.

Parameters

base_estimator [object or None, optional (default=None)] The base estimator to fit on random subsets of the dataset. If None, then the base estimator is a decision tree.

n_estimators [int, optional (default=10)] The number of base estimators in the ensemble.

max_samples [int or float, optional (default=1.0)] The number of samples to draw from X to train each base estimator.

- If int, then draw `max_samples` samples.
- If float, then draw `max_samples * X.shape[0]` samples.

max_features [int or float, optional (default=1.0)] The number of features to draw from X to train each base estimator.

- If int, then draw `max_features` features.
- If float, then draw `max_features * X.shape[1]` features.

bootstrap [boolean, optional (default=True)] Whether samples are drawn with replacement. If False, sampling without replacement is performed.

bootstrap_features [boolean, optional (default=False)] Whether features are drawn with replacement.

oob_score [bool, optional (default=False)] Whether to use out-of-bag samples to estimate the generalization error.

warm_start [bool, optional (default=False)] When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new ensemble. See *the Glossary*.

New in version 0.17: *warm_start* constructor parameter.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for both *fit* and *predict*. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

verbose [int, optional (default=0)] Controls the verbosity when fitting and predicting.

Attributes

base_estimator_ [estimator] The base estimator from which the ensemble is grown.

n_features_ [int] The number of features when *fit* is performed.

estimators_ [list of estimators] The collection of fitted base estimators.

estimators_samples_ [list of arrays] The subset of drawn samples for each base estimator.

estimators_features_ [list of arrays] The subset of drawn features for each base estimator.

classes_ [array of shape (n_classes,)] The classes labels.

n_classes_ [int or list] The number of classes.

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate. This attribute exists only when `oob_score` is True.

oob_decision_function_ [array of shape (n_samples, n_classes)] Decision function computed with out-of-bag estimate on the training set. If n_estimators is small it might be possible that a data point was never left out during the bootstrap. In this case, `oob_decision_function_` might contain NaN. This attribute exists only when `oob_score` is True.

References

[Rb1846455d0e5-1], [Rb1846455d0e5-2], [Rb1846455d0e5-3], [Rb1846455d0e5-4]

Examples

```
>>> from sklearn.svm import SVC
>>> from sklearn.ensemble import BaggingClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=100, n_features=4,
...                            n_informative=2, n_redundant=0,
...                            random_state=0, shuffle=False)
>>> clf = BaggingClassifier(base_estimator=SVC(),
...                         n_estimators=10, random_state=0).fit(X, y)
>>> clf.predict([[0, 0, 0, 0]])
array([1])
```

Methods

decision_function(self, X)	Average of the decision functions of the base classifiers.
fit(self, X, y[, sample_weight])	Build a Bagging ensemble of estimators from the training
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class for X.
predict_log_proba(self, X)	Predict class log-probabilities for X.
predict_proba(self, X)	Predict class probabilities for X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *base_estimator=None*, *n_estimators=10*, *max_samples=1.0*, *max_features=1.0*, *bootstrap=True*, *bootstrap_features=False*, *oob_score=False*, *warm_start=False*, *n_jobs=None*, *random_state=None*, *verbose=0*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Average of the decision functions of the base classifiers.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrices are accepted only if they are supported by the base estimator.

Returns

score [array, shape = [n_samples, k]] The decision function of the input samples. The columns correspond to the classes in sorted order, as they appear in the attribute `classes_`. Regression and binary classification are special cases with `k == 1`, otherwise `k==n_classes`.

property estimators_samples_

The subset of drawn samples for each base estimator.

Returns a dynamically generated list of indices identifying the samples used for fitting each member of the ensemble, i.e., the in-bag samples.

Note: the list is re-created at each call to the property in order to reduce the object memory footprint by not storing the sampling data. Thus fetching the property may be slower than expected.

fit(*self*, *X*, *y*, *sample_weight=None*)

Build a Bagging ensemble of estimators from the training set (X, y).

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrices are accepted only if they are supported by the base estimator.

y [array-like of shape (n_samples,)] The target values (class labels in classification, real numbers in regression).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Note that this is supported only if the base estimator supports sample weighting.

Returns

self [object]

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)

Predict class for X.

The predicted class of an input sample is computed as the class with the highest mean predicted probability. If base estimators do not implement a `predict_proba` method, then it resorts to voting.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrices are accepted only if they are supported by the base estimator.

Returns

y [ndarray of shape (n_samples,)] The predicted classes.

predict_log_proba(*self*, *X*)
Predict class log-probabilities for X.

The predicted class log-probabilities of an input sample is computed as the log of the mean predicted class probabilities of the base estimators in the ensemble.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrices are accepted only if they are supported by the base estimator.

Returns

p [array of shape (n_samples, n_classes)] The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

predict_proba(*self*, *X*)
Predict class probabilities for X.

The predicted class probabilities of an input sample is computed as the mean predicted class probabilities of the base estimators in the ensemble. If base estimators do not implement a `predict_proba` method, then it resorts to voting and the predicted class probabilities of an input sample represents the proportion of estimators predicting each class.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrices are accepted only if they are supported by the base estimator.

Returns

p [array of shape (n_samples, n_classes)] The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

7.11.4 sklearn.ensemble.BaggingRegressor

class sklearn.ensemble.**BaggingRegressor**(*base_estimator=None*, *n_estimators=10*, *max_samples=1.0*, *max_features=1.0*, *bootstrap=True*, *bootstrap_features=False*, *oob_score=False*, *warm_start=False*, *n_jobs=None*, *random_state=None*, *verbose=0*)

A Bagging regressor.

A Bagging regressor is an ensemble meta-estimator that fits base regressors each on random subsets of the original dataset and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

This algorithm encompasses several works from the literature. When random subsets of the dataset are drawn as random subsets of the samples, then this algorithm is known as Pasting [R4d113ba76fc0-1]. If samples are drawn with replacement, then the method is known as Bagging [R4d113ba76fc0-2]. When random subsets of the dataset are drawn as random subsets of the features, then the method is known as Random Subspaces [R4d113ba76fc0-3]. Finally, when base estimators are built on subsets of both samples and features, then the method is known as Random Patches [R4d113ba76fc0-4].

Read more in the *User Guide*.

New in version 0.15.

Parameters

base_estimator [object or None, optional (default=None)] The base estimator to fit on random subsets of the dataset. If None, then the base estimator is a decision tree.

n_estimators [int, optional (default=10)] The number of base estimators in the ensemble.

max_samples [int or float, optional (default=1.0)] The number of samples to draw from X to train each base estimator.

- If int, then draw `max_samples` samples.
- If float, then draw `max_samples * X.shape[0]` samples.

max_features [int or float, optional (default=1.0)] The number of features to draw from X to train each base estimator.

- If int, then draw `max_features` features.
- If float, then draw `max_features * X.shape[1]` features.

bootstrap [boolean, optional (default=True)] Whether samples are drawn with replacement. If False, sampling without replacement is performed.

bootstrap_features [boolean, optional (default=False)] Whether features are drawn with replacement.

oob_score [bool] Whether to use out-of-bag samples to estimate the generalization error.

warm_start [bool, optional (default=False)] When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new ensemble. See *the Glossary*.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for both *fit* and *predict*. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

verbose [int, optional (default=0)] Controls the verbosity when fitting and predicting.

Attributes

base_estimator_ [estimator] The base estimator from which the ensemble is grown.

n_features_ [int] The number of features when *fit* is performed.

estimators_ [list of estimators] The collection of fitted sub-estimators.

estimators_samples_ [list of arrays] The subset of drawn samples for each base estimator.

estimators_features_ [list of arrays] The subset of drawn features for each base estimator.

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate. This attribute exists only when `oob_score` is True.

oob_prediction_ [ndarray of shape (n_samples,)] Prediction computed with out-of-bag estimate on the training set. If n_estimators is small it might be possible that a data point was never left out during the bootstrap. In this case, `oob_prediction_` might contain NaN. This attribute exists only when `oob_score` is True.

References

[R4d113ba76fc0-1], [R4d113ba76fc0-2], [R4d113ba76fc0-3], [R4d113ba76fc0-4]

Examples

```
>>> from sklearn.svm import SVR
>>> from sklearn.ensemble import BaggingRegressor
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(n_samples=100, n_features=4,
...                        n_informative=2, n_targets=1,
...                        random_state=0, shuffle=False)
>>> regr = BaggingRegressor(base_estimator=SVR(),
...                         n_estimators=10, random_state=0).fit(X, y)
>>> regr.predict([[0, 0, 0, 0]])
array([-2.8720...])
```

Methods

fit(self, X, y[, sample_weight])	Build a Bagging ensemble of estimators from the training
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict regression target for X.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *base_estimator=None*, *n_estimators=10*, *max_samples=1.0*, *max_features=1.0*, *bootstrap=True*, *bootstrap_features=False*, *oob_score=False*, *warm_start=False*, *n_jobs=None*, *random_state=None*, *verbose=0*)
Initialize self. See help(type(self)) for accurate signature.

property estimators_samples_
The subset of drawn samples for each base estimator.

Returns a dynamically generated list of indices identifying the samples used for fitting each member of the ensemble, i.e., the in-bag samples.

Note: the list is re-created at each call to the property in order to reduce the object memory footprint by not storing the sampling data. Thus fetching the property may be slower than expected.

fit (*self*, *X*, *y*, *sample_weight=None*)

Build a Bagging ensemble of estimators from the training set (X, y).

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrices are accepted only if they are supported by the base estimator.

y [array-like of shape (n_samples,)] The target values (class labels in classification, real numbers in regression).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Note that this is supported only if the base estimator supports sample weighting.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict regression target for X.

The predicted regression target of an input sample is computed as the mean predicted regression targets of the estimators in the ensemble.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Sparse matrices are accepted only if they are supported by the base estimator.

Returns

y [ndarray of shape (n_samples,)] The predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.ensemble.BaggingRegressor`

- *Single estimator versus bagging: bias-variance decomposition*

7.11.5 `sklearn.ensemble`.IsolationForest

class sklearn.ensemble.**IsolationForest**(*n_estimators=100*, *max_samples='auto'*, *contamination='auto'*, *max_features=1.0*, *bootstrap=False*, *n_jobs=None*, *behaviour='deprecated'*, *random_state=None*, *verbose=0*, *warm_start=False*)

Isolation Forest Algorithm.

Return the anomaly score of each sample using the IsolationForest algorithm

The IsolationForest 'isolates' observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature.

Since recursive partitioning can be represented by a tree structure, the number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node.

This path length, averaged over a forest of such random trees, is a measure of normality and our decision function.

Random partitioning produces noticeably shorter paths for anomalies. Hence, when a forest of random trees collectively produce shorter path lengths for particular samples, they are highly likely to be anomalies.

Read more in the *User Guide*.

New in version 0.18.

Parameters

n_estimators [int, optional (default=100)] The number of base estimators in the ensemble.

max_samples [int or float, optional (default="auto")]

The number of samples to draw from X to train each base estimator.

- If int, then draw `max_samples` samples.
- If float, then draw `max_samples * X.shape[0]` samples.
- If "auto", then `max_samples=min(256, n_samples)`.

If max_samples is larger than the number of samples provided, all samples will be used for all trees (no sampling).

contamination ['auto' or float, optional (default='auto')] The amount of contamination of the data set, i.e. the proportion of outliers in the data set. Used when fitting to define the threshold on the scores of the samples.

- If 'auto', the threshold is determined as in the original paper.
- If float, the contamination should be in the range [0, 0.5].

Changed in version 0.22: The default value of `contamination` changed from 0.1 to `'auto'`.

max_features [int or float, optional (default=1.0)] The number of features to draw from X to train each base estimator.

- If int, then draw `max_features` features.
- If float, then draw `max_features * X.shape[1]` features.

bootstrap [bool, optional (default=False)] If True, individual trees are fit on random subsets of the training data sampled with replacement. If False, sampling without replacement is performed.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for both *fit* and *predict*. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

behaviour [str, default='deprecated'] This parameter has not effect, is deprecated, and will be removed.

New in version 0.20: `behaviour` is added in 0.20 for back-compatibility purpose.

Deprecated since version 0.20: `behaviour='old'` is deprecated in 0.20 and will not be possible in 0.22.

Deprecated since version 0.22: `behaviour` parameter is deprecated in 0.22 and removed in 0.24.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

verbose [int, optional (default=0)] Controls the verbosity of the tree building process.

warm_start [bool, optional (default=False)] When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See *the Glossary*.

New in version 0.21.

Attributes

estimators_ [list of DecisionTreeClassifier] The collection of fitted sub-estimators.

estimators_samples_ [list of arrays] The subset of drawn samples for each base estimator.

max_samples_ [integer] The actual number of samples

offset_ [float] Offset used to define the decision function from the raw scores. We have the relation: `decision_function = score_samples - offset_`. `offset_` is defined as follows. When the contamination parameter is set to "auto", the offset is equal to -0.5 as the scores of inliers are close to 0 and the scores of outliers are close to -1. When a contamination parameter different than "auto" is provided, the offset is defined in such a way we obtain the expected number of outliers (samples with decision function < 0) in training.

See also:

sklearn.covariance.EllipticEnvelope An object for detecting outliers in a Gaussian distributed dataset.

sklearn.svm.OneClassSVM Unsupervised Outlier Detection. Estimate the support of a high-dimensional distribution. The implementation is based on libsvm.

sklearn.neighbors.LocalOutlierFactor Unsupervised Outlier Detection using Local Outlier Factor (LOF).

Notes

The implementation is based on an ensemble of ExtraTreeRegressor. The maximum depth of each tree is set to `ceil(log_2(n))` where n is the number of samples used to build the tree (see (Liu et al., 2008) for more details).

References

[Rd7ae0a2ae688-1], [Rd7ae0a2ae688-2]

Examples

```
>>> from sklearn.ensemble import IsolationForest
>>> X = [[-1.1], [0.3], [0.5], [100]]
>>> clf = IsolationForest(random_state=0).fit(X)
```

(continues on next page)

(continued from previous page)

```
>>> clf.predict([[0.1], [0], [90]])
array([ 1,  1, -1])
```

Methods

decision_function(self, X)	Average anomaly score of X of the base classifiers.
fit(self, X[, y, sample_weight])	Fit estimator.
fit_predict(self, X[, y])	Perform fit on X and returns labels for X.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict if a particular sample is an outlier or not.
score_samples(self, X)	Opposite of the anomaly score defined in the original paper.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_estimators=100*, *max_samples='auto'*, *contamination='auto'*, *max_features=1.0*, *bootstrap=False*, *n_jobs=None*, *behaviour='deprecated'*, *random_state=None*, *verbose=0*, *warm_start=False*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Average anomaly score of X of the base classifiers.

The anomaly score of an input sample is computed as the mean anomaly score of the trees in the forest.

The measure of normality of an observation given a tree is the depth of the leaf containing this observation, which is equivalent to the number of splittings required to isolate this point. In case of several observations n_left in the leaf, the average path length of a n_left samples isolation tree is added.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

scores [array, shape (n_samples,)] The anomaly score of the input samples. The lower, the more abnormal. Negative scores represent outliers, positive scores represent inliers.

property estimators_samples_
The subset of drawn samples for each base estimator.

Returns a dynamically generated list of indices identifying the samples used for fitting each member of the ensemble, i.e., the in-bag samples.

Note: the list is re-created at each call to the property in order to reduce the object memory footprint by not storing the sampling data. Thus fetching the property may be slower than expected.

fit(*self*, *X*, *y=None*, *sample_weight=None*)
Fit estimator.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] The input samples. Use `dtype=np.float32` for maximum efficiency. Sparse matrices are also supported, use sparse `csc_matrix` for maximum efficiency.

y [Ignored] Not used, present for API consistency by convention.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted.

Returns

self [object] Fitted estimator.

fit_predict(*self*, *X*, *y=None*)
Perform fit on X and returns labels for X.

Returns -1 for outliers and 1 for inliers.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

y [Ignored] Not used, present for API consistency by convention.

Returns

y [ndarray, shape (n_samples,)] 1 for inliers, -1 for outliers.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict if a particular sample is an outlier or not.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

is_inlier [array, shape (n_samples,)] For each observation, tells whether or not (+1 or -1) it should be considered as an inlier according to the fitted model.

score_samples(*self*, *X*)
Opposite of the anomaly score defined in the original paper.

The anomaly score of an input sample is computed as the mean anomaly score of the trees in the forest.

The measure of normality of an observation given a tree is the depth of the leaf containing this observation, which is equivalent to the number of splittings required to isolate this point. In case of several observations n_left in the leaf, the average path length of a n_left samples isolation tree is added.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] The input samples.

Returns

scores [array, shape (n_samples,)] The anomaly score of the input samples. The lower, the more abnormal.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.ensemble.IsolationForest`

- *Comparing anomaly detection algorithms for outlier detection on toy datasets*
- *IsolationForest example*

7.11.6 `sklearn.ensemble`.RandomTreesEmbedding

class sklearn.ensemble.**RandomTreesEmbedding**(*n_estimators=100*, *max_depth=5*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *sparse_output=True*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*)

An ensemble of totally random trees.

An unsupervised transformation of a dataset to a high-dimensional sparse representation. A datapoint is coded according to which leaf of each tree it is sorted into. Using a one-hot encoding of the leaves, this leads to a binary coding with as many ones as there are trees in the forest.

The dimensionality of the resulting representation is `n_out <= n_estimators * max_leaf_nodes`. If `max_leaf_nodes == None`, the number of leaf nodes is at most `n_estimators * 2 ** max_depth`.

Read more in the *User Guide*.

Parameters

n_estimators [integer, optional (default=10)] Number of trees in the forest.

Changed in version 0.22: The default value of `n_estimators` changed from 10 to 100 in 0.22.

max_depth [integer, optional (default=5)] The maximum depth of each tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.

- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` is the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` is the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_leaf_nodes [int or None, optional (default=None)] Grow trees with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

sparse_output [bool, optional (default=True)] Whether or not to return a sparse CSR matrix, as default behavior, or to return a dense array compatible with dense pipeline operators.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel. *fit*, *transform*, *decision_path* and *apply* are all parallelized over the trees. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional (default=None)] Controls the generation of the random `y` used to fit the trees and the draw of the splits for each feature at the trees' nodes. See *Glossary* for details.

verbose [int, optional (default=0)] Controls the verbosity when fitting and predicting.

warm_start [bool, optional (default=False)] When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See *the Glossary*.

Attributes

estimators_ [list of DecisionTreeClassifier] The collection of fitted sub-estimators.

References

[R6e47e53bacbd-1], [R6e47e53bacbd-2]

Methods

apply(self, X)	Apply trees in the forest to X, return leaf indices.
decision_path(self, X)	Return the decision path in the forest.
fit(self, X[, y, sample_weight])	Fit estimator.
fit_transform(self, X[, y, sample_weight])	Fit estimator and transform dataset.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform dataset.

__init__(*self*, *n_estimators=100*, *max_depth=5*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *sparse_output=True*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*)
Initialize self. See help(type(self)) for accurate signature.

apply(*self*, *X*)
Apply trees in the forest to X, return leaf indices.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

X_leaves [array_like, shape = [n_samples, n_estimators]] For each datapoint x in X and for each tree in the forest, return the index of the leaf x ends up in.

decision_path(*self*, *X*)
Return the decision path in the forest.

New in version 0.18.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

indicator [sparse csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

n_nodes_ptr [array of size (n_estimators + 1,)] The columns from indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]] gives the indicator value for the i-th estimator.

property feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [array, shape = [n_features]] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

fit (*self*, *X*, *y=None*, *sample_weight=None*)
Fit estimator.

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] The input samples. Use `dtype=np.float32` for maximum efficiency. Sparse matrices are also supported, use sparse `csc_matrix` for maximum efficiency.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

self [object]

fit_transform (*self*, *X*, *y=None*, *sample_weight=None*)
Fit estimator and transform dataset.

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] Input data used to build forests. Use `dtype=np.float32` for maximum efficiency.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

X_transformed [sparse matrix, shape=(n_samples, n_out)] Transformed dataset.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Transform dataset.

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] Input data to be transformed. Use `dtype=np.float32` for maximum efficiency. Sparse matrices are also supported, use sparse `csr_matrix` for maximum efficiency.

Returns

X_transformed [sparse matrix, shape=(n_samples, n_out)] Transformed dataset.

Examples using `sklearn.ensemble.RandomTreesEmbedding`

- *Hashing feature transformation using Totally Random Trees*
- *Feature transformations with ensembles of trees*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

7.11.7 sklearn.ensemble.StackingClassifier

class sklearn.ensemble.**StackingClassifier**(*estimators*, *final_estimator=None*, *cv=None*, *stack_method='auto'*, *n_jobs=None*, *passthrough=False*, *verbose=0*)
Stack of estimators with a final classifier.

Stacked generalization consists in stacking the output of individual estimator and use a classifier to compute the final prediction. Stacking allows to use the strength of each individual estimator by using their output as input of a final estimator.

Note that `estimators_` are fitted on the full `X` while `final_estimator_` is trained using cross-validated predictions of the base estimators using `cross_val_predict`.

New in version 0.22.

Read more in the *User Guide*.

Parameters

estimators [list of (str, estimator)] Base estimators which will be stacked together. Each element of the list is defined as a tuple of string (i.e. name) and an estimator instance. An estimator can be set to 'drop' using `set_params`.

final_estimator [estimator, default=None] A classifier which will be used to combine the base estimators. The default classifier is a `LogisticRegression`.

cv [int, cross-validation generator or an iterable, default=None] Determines the cross-validation splitting strategy used in `cross_val_predict` to train `final_estimator`. Possible inputs for cv are:

- None, to use the default 5-fold cross validation,
- integer, to specify the number of folds in a (Stratified) KFold,
- An object to be used as a cross-validation generator,
- An iterable yielding train, test splits.

For integer/None inputs, if the estimator is a classifier and y is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Note: A larger number of split will provide no benefits if the number of training samples is large enough. Indeed, the training time will increase. `cv` is not used for model evaluation but for prediction.

---

stack_method [{'auto', 'predict_proba', 'decision_function', 'predict'}, default='auto'] Methods called for each base estimator. It can be:

- if 'auto', it will try to invoke, for each estimator, `'predict_proba'`, `'decision_function'` or `'predict'` in that order.
- otherwise, one of `'predict_proba'`, `'decision_function'` or `'predict'`. If the method is not implemented by the estimator, it will raise an error.

n_jobs [int, default=None] The number of jobs to run in parallel all `estimators` `fit`. `None` means 1 unless in a `joblib.parallel_backend` context. -1 means using all processors. See Glossary for more details.

passthrough [bool, default=False] When False, only the predictions of estimators will be used as training data for `final_estimator`. When True, the `final_estimator` is trained on the predictions as well as the original training data.

Attributes

estimators_ [list of estimators] The elements of the estimators parameter, having been fitted on the training data. If an estimator has been set to `'drop'`, it will not appear in `estimators_`.

named_estimators_ [Bunch] Attribute to access any fitted sub-estimators by name.

final_estimator_ [estimator] The classifier which predicts given the output of `estimators_`.

stack_method_ [list of str] The method used by each base estimator.

Notes

When `predict_proba` is used by each estimator (i.e. most of the time for `stack_method='auto'` or specifically for `stack_method='predict_proba'`), The first column predicted by each estimator will be dropped in the case of a binary classification problem. Indeed, both feature will be perfectly collinear.

References

[Rb91ed47a817e-1]

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.svm import LinearSVC
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.ensemble import StackingClassifier
>>> X, y = load_iris(return_X_y=True)
>>> estimators = [
...     ('rf', RandomForestClassifier(n_estimators=10, random_state=42)),
...     ('svr', make_pipeline(StandardScaler(),
...                           LinearSVC(random_state=42)))
... ]
>>> clf = StackingClassifier(
...     estimators=estimators, final_estimator=LogisticRegression()
... )
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, stratify=y, random_state=42
... )
>>> clf.fit(X_train, y_train).score(X_test, y_test)
0.9...
```

Methods

decision_function(self, X)	Predict decision function for samples in X using `final_estimator_.decision_function`.
fit(self, X, y[, sample_weight])	Fit the estimators.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get the parameters of an estimator from the ensemble.
predict(self, X, **predict_params)	Predict target for X.
predict_proba(self, X)	Predict class probabilities for X using `final_estimator_.predict_proba`.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of an estimator from the ensemble.
transform(self, X)	Return class labels or probabilities for X for each estimator.

__init__(*self*, *estimators*, *final_estimator=None*, *cv=None*, *stack_method='auto'*, *n_jobs=None*, *passthrough=False*, *verbose=0*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Predict decision function for samples in X using `final_estimator_.decision_function`.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

Returns

decisions [ndarray of shape (n_samples,), (n_samples, n_classes), or (n_samples, n_classes * (n_classes-1) / 2)] The decision function computed the final estimator.

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit the estimators.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where `n_samples` is the number of samples and `n_features` is the number of features.

y [array-like of shape (n_samples,)] Target values.

sample_weight [array-like of shape (n_samples,) or None] Sample weights. If None, then samples are equally weighted. Note that this is supported only if all underlying estimators support sample weights.

Returns

self [object]

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get the parameters of an estimator from the ensemble.

Parameters

deep [bool] Setting it to True gets the various classifiers and the parameters of the classifiers as well.

predict (*self*, *X*, ***predict_params*)
Predict target for X.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

****predict_params** [dict of str -> obj] Parameters to the `predict` called by the `final_estimator`. Note that this may be used to return uncertainties from some estimators with `return_std` or `return_cov`. Be aware that it will only accounts for uncertainty in the final estimator.

Returns

y_pred [ndarray of shape (n_samples,) or (n_samples, n_output)] Predicted targets.

predict_proba (*self*, *X*)
Predict class probabilities for X using `final_estimator_.predict_proba`.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

Returns

probabilities [ndarray of shape (n_samples, n_classes) or list of ndarray of shape (n_output,)] The class probabilities of the input samples.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)

Set the parameters of an estimator from the ensemble.

Valid parameter keys can be listed with `get_params()`.

Parameters

****params** [keyword arguments] Specific parameters using e.g. `set_params(parameter_name=new_value)`. In addition, to setting the parameters of the stacking estimator, the individual estimator of the stacking estimators can also be set, or can be removed by setting them to 'drop'.

transform(*self*, *X*)

Return class labels or probabilities for X for each estimator.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns

y_preds [ndarray of shape (n_samples, n_estimators) or (n_samples, n_classes * n_estimators)] Prediction outputs for each estimator.

Examples using `sklearn.ensemble.StackingClassifier`

- *Release Highlights for scikit-learn 0.22*

7.11.8 `sklearn.ensemble`.StackingRegressor

class sklearn.ensemble.**StackingRegressor**(*estimators*, *final_estimator=None*, *cv=None*, *n_jobs=None*, *passthrough=False*, *verbose=0*)

Stack of estimators with a final regressor.

Stacked generalization consists in stacking the output of individual estimator and use a regressor to compute the final prediction. Stacking allows to use the strength of each individual estimator by using their output as input of a final estimator.

Note that `estimators_` are fitted on the full `X` while `final_estimator_` is trained using cross-validated predictions of the base estimators using `cross_val_predict`.

New in version 0.22.

Read more in the *User Guide*.

Parameters

estimators [list of (str, estimator)] Base estimators which will be stacked together. Each element of the list is defined as a tuple of string (i.e. name) and an estimator instance. An estimator can be set to 'drop' using `set_params`.

final_estimator [estimator, default=None] A regressor which will be used to combine the base estimators. The default regressor is a `RidgeCV`.

cv [int, cross-validation generator or an iterable, default=None] Determines the cross-validation splitting strategy used in `cross_val_predict` to train `final_estimator`. Possible inputs for cv are:

- None, to use the default 5-fold cross validation,
- integer, to specify the number of folds in a (Stratified) KFold,
- An object to be used as a cross-validation generator,
- An iterable yielding train, test splits.

For integer/None inputs, if the estimator is a classifier and y is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Note: A larger number of split will provide no benefits if the number of training samples is large enough. Indeed, the training time will increase. `cv` is not used for model evaluation but for prediction.

---

n_jobs [int, default=None] The number of jobs to run in parallel for `fit` of all `estimators`. `None` means 1 unless in a `joblib.parallel_backend` context. -1 means using all processors. See Glossary for more details.

passthrough [bool, default=False] When False, only the predictions of estimators will be used as training data for `final_estimator`. When True, the `final_estimator` is trained on the predictions as well as the original training data.

Attributes

estimators_ [list of estimator] The elements of the estimators parameter, having been fitted on the training data. If an estimator has been set to `'drop'`, it will not appear in `estimators_`.

named_estimators_ [Bunch] Attribute to access any fitted sub-estimators by name.

final_estimator_ [estimator] The regressor to stacked the base estimators fitted.

References

[R606df7ffad02-1]

Examples

```
>>> from sklearn.datasets import load_diabetes
>>> from sklearn.linear_model import RidgeCV
>>> from sklearn.svm import LinearSVR
>>> from sklearn.ensemble import RandomForestRegressor
>>> from sklearn.ensemble import StackingRegressor
>>> X, y = load_diabetes(return_X_y=True)
>>> estimators = [
...     ('lr', RidgeCV()),
...     ('svr', LinearSVR(random_state=42))
... ]
>>> reg = StackingRegressor(
...     estimators=estimators,
...     final_estimator=RandomForestRegressor(n_estimators=10,
...                                           random_state=42)
... )
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, random_state=42
... )
>>> reg.fit(X_train, y_train).score(X_test, y_test)
0.3...
```

Methods

fit(self, X, y[, sample_weight])	Fit the estimators.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get the parameters of an estimator from the ensemble.
predict(self, X, **predict_params)	Predict target for X.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of an estimator from the ensemble.
transform(self, X)	Return the predictions for X for each estimator.

__init__(*self*, *estimators*, *final_estimator=None*, *cv=None*, *n_jobs=None*, *passthrough=False*, *verbose=0*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit the estimators.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like of shape (n_samples,)] Target values.

sample_weight [array-like of shape (n_samples,) or None] Sample weights. If None, then samples are equally weighted. Note that this is supported only if all underlying estimators support sample weights.

Returns

self [object]

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get the parameters of an estimator from the ensemble.

Parameters

deep [bool] Setting it to True gets the various classifiers and the parameters of the classifiers as well.

predict(*self*, *X*, ***predict_params*)
Predict target for X.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

****predict_params** [dict of str -> obj] Parameters to the `predict` called by the `final_estimator`. Note that this may be used to return uncertainties from some estimators with `return_std` or `return_cov`. Be aware that it will only accounts for uncertainty in the final estimator.

Returns

y_pred [ndarray of shape (n_samples,) or (n_samples, n_output)] Predicted targets.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of an estimator from the ensemble.

Valid parameter keys can be listed with `get_params()`.

Parameters

****params** [keyword arguments] Specific parameters using e.g. `set_params(parameter_name=new_value)`. In addition, to setting the parameters of the stacking estimator, the individual estimator of the stacking estimators can also be set, or can be removed by setting them to 'drop'.

transform(*self*, *X*)
Return the predictions for X for each estimator.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns

y_preds [ndarray of shape (n_samples, n_estimators)] Prediction outputs for each estimator.

Examples using `sklearn.ensemble.StackingRegressor`

- *Combine predictors using stacking*

7.11.9 `sklearn.ensemble`.VotingClassifier

class sklearn.ensemble.**VotingClassifier**(*estimators*, *voting='hard'*, *weights=None*, *n_jobs=None*, *flatten_transform=True*)
Soft Voting/Majority Rule classifier for unfitted estimators.

New in version 0.17.

Read more in the *User Guide*.

Parameters

estimators [list of (str, estimator) tuples] Invoking the `fit` method on the `VotingClassifier` will fit clones of those original estimators that will be stored in the class attribute `self.estimators_`. An estimator can be set to `'drop'` using `set_params`.

Deprecated since version 0.22: Using `None` to drop an estimator is deprecated in 0.22 and support will be dropped in 0.24. Use the string `'drop'` instead.

voting [str, {'hard', 'soft'} (default='hard')] If 'hard', uses predicted class labels for majority rule voting. Else if 'soft', predicts the class label based on the argmax of the sums of the predicted probabilities, which is recommended for an ensemble of well-calibrated classifiers.

weights [array-like, shape (n_classifiers,), optional (default='None')] Sequence of weights (`float` or `int`) to weight the occurrences of predicted class labels (`hard` voting) or class probabilities before averaging (`soft` voting). Uses uniform weights if `None`.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for `fit`. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

flatten_transform [bool, optional (default=True)] Affects shape of transform output only when voting='soft' If voting='soft' and flatten_transform=True, transform method returns matrix with shape (n_samples, n_classifiers * n_classes). If flatten_transform=False, it returns (n_classifiers, n_samples, n_classes).

Attributes

estimators_ [list of classifiers] The collection of fitted sub-estimators as defined in `estimators` that are not 'drop'.

named_estimators_ [Bunch object, a dictionary with attribute access] Attribute to access any fitted sub-estimators by name.

New in version 0.20.

classes_ [array-like, shape (n_predictions,)] The classes labels.

See also:

VotingRegressor Prediction voting regressor.

Examples

```
>>> import numpy as np
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.ensemble import RandomForestClassifier, VotingClassifier
>>> clf1 = LogisticRegression(multi_class='multinomial', random_state=1)
>>> clf2 = RandomForestClassifier(n_estimators=50, random_state=1)
>>> clf3 = GaussianNB()
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> y = np.array([1, 1, 1, 2, 2, 2])
>>> eclf1 = VotingClassifier(estimators=[
...         ('lr', clf1), ('rf', clf2), ('gnb', clf3)], voting='hard')
>>> eclf1 = eclf1.fit(X, y)
>>> print(eclf1.predict(X))
[1 1 1 2 2 2]
>>> np.array_equal(eclf1.named_estimators_.lr.predict(X),
...                eclf1.named_estimators_['lr'].predict(X))
True
>>> eclf2 = VotingClassifier(estimators=[
...         ('lr', clf1), ('rf', clf2), ('gnb', clf3)],
...         voting='soft')
>>> eclf2 = eclf2.fit(X, y)
```

(continues on next page)

(continued from previous page)

```
>>> print(eclf2.predict(X))
[1 1 1 2 2 2]
>>> eclf3 = VotingClassifier(estimators=[
...         ('lr', clf1), ('rf', clf2), ('gnb', clf3)],
...         voting='soft', weights=[2,1,1],
...         flatten_transform=True)
>>> eclf3 = eclf3.fit(X, y)
>>> print(eclf3.predict(X))
[1 1 1 2 2 2]
>>> print(eclf3.transform(X).shape)
(6, 6)
```

Methods

fit(self, X, y[, sample_weight])	Fit the estimators.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get the parameters of an estimator from the ensemble.
predict(self, X)	Predict class labels for X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of an estimator from the ensemble.
transform(self, X)	Return class labels or probabilities for X for each estimator.

__init__(*self*, *estimators*, *voting='hard'*, *weights=None*, *n_jobs=None*, *flatten_transform=True*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit the estimators.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape (n_samples,)] Target values.

sample_weight [array-like, shape (n_samples,) or None] Sample weights. If None, then samples are equally weighted. Note that this is supported only if all underlying estimators support sample weights.

Returns

self [object]

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get the parameters of an estimator from the ensemble.

Parameters

deep [bool] Setting it to True gets the various classifiers and the parameters of the classifiers as well.

predict (*self*, *X*)
Predict class labels for X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples.

Returns

maj [array-like, shape (n_samples,)] Predicted class labels.

property predict_proba
Compute probabilities of possible outcomes for samples in X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples.

Returns

avg [array-like, shape (n_samples, n_classes)] Weighted average probability for each class per sample.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params (*self*, ***params*)
Set the parameters of an estimator from the ensemble.

Valid parameter keys can be listed with `get_params()`.

Parameters

****params** [keyword arguments] Specific parameters using e.g. `set_params(parameter_name=new_value)`. In addition, to setting the parameters of the stacking estimator, the individual estimator of the stacking estimators can also be set, or can be removed by setting them to 'drop'.

transform(*self*, *X*)
Return class labels or probabilities for X for each estimator.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

Returns

probabilities_or_labels

If voting='soft' and flatten_transform=True: returns array-like of shape (n_classifiers, n_samples * n_classes), being class probabilities calculated by each classifier.

If voting='soft' and `flatten_transform=False: array-like of shape (n_classifiers, n_samples, n_classes)

If voting='hard': array-like of shape (n_samples, n_classifiers), being class labels predicted by each classifier.

Examples using sklearn.ensemble.VotingClassifier

- *Plot the decision boundaries of a VotingClassifier*
- *Plot class probabilities calculated by the VotingClassifier*

7.11.10 sklearn.ensemble.VotingRegressor

class sklearn.ensemble.**VotingRegressor**(*estimators*, *weights=None*, *n_jobs=None*)
Prediction voting regressor for unfitted estimators.

New in version 0.21.

A voting regressor is an ensemble meta-estimator that fits base regressors each on the whole dataset. It, then, averages the individual predictions to form a final prediction.

Read more in the *User Guide*.

Parameters

estimators [list of (str, estimator) tuples] Invoking the `fit` method on the `VotingRegressor` will fit clones of those original estimators that will be stored in the class attribute `self.estimators_`. An estimator can be set to `'drop'` using `set_params`.

Deprecated since version 0.22: Using `None` to drop an estimator is deprecated in 0.22 and support will be dropped in 0.24. Use the string `'drop'` instead.

weights [array-like, shape (n_regressors,), optional (default=‘None‘)] Sequence of weights (`float` or `int`) to weight the occurrences of predicted values before averaging. Uses uniform weights if `None`.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for `fit`. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

estimators_ [list of regressors] The collection of fitted sub-estimators as defined in `estimators` that are not 'drop'.

named_estimators_ [Bunch object, a dictionary with attribute access] Attribute to access any fitted sub-estimators by name.

New in version 0.20.

See also:

VotingClassifier Soft Voting/Majority Rule classifier.

Examples

```
>>> import numpy as np
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.ensemble import RandomForestRegressor
>>> from sklearn.ensemble import VotingRegressor
>>> r1 = LinearRegression()
>>> r2 = RandomForestRegressor(n_estimators=10, random_state=1)
>>> X = np.array([[1, 1], [2, 4], [3, 9], [4, 16], [5, 25], [6, 36]])
>>> y = np.array([2, 6, 12, 20, 30, 42])
>>> er = VotingRegressor([('lr', r1), ('rf', r2)])
>>> print(er.fit(X, y).predict(X))
[ 3.3  5.7 11.8 19.7 28.  40.3]
```

Methods

fit(self, X, y[, sample_weight])	Fit the estimators.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get the parameters of an estimator from the ensemble.
predict(self, X)	Predict regression target for X.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of an estimator from the ensemble.
transform(self, X)	Return predictions for X for each estimator.

__init__ (*self*, *estimators*, *weights=None*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit the estimators.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape (n_samples,)] Target values.

sample_weight [array-like, shape (n_samples,) or None] Sample weights. If None, then samples are equally weighted. Note that this is supported only if all underlying estimators support sample weights.

Returns

self [object] Fitted estimator.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get the parameters of an estimator from the ensemble.

Parameters

deep [bool] Setting it to True gets the various classifiers and the parameters of the classifiers as well.

predict(*self*, *X*)
Predict regression target for X.

The predicted regression target of an input sample is computed as the mean predicted regression targets of the estimators in the ensemble.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples.

Returns

y [array of shape (n_samples,)] The predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of an estimator from the ensemble.

Valid parameter keys can be listed with `get_params()`.

Parameters

****params** [keyword arguments] Specific parameters using e.g. `set_params(parameter_name=new_value)`. In addition, to setting the parameters of the stacking estimator, the individual estimator of the stacking estimators can also be set, or can be removed by setting them to 'drop'.

transform(*self*, *X*)
Return predictions for X for each estimator.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples.

Returns

predictions: array of shape (n_samples, n_classifiers) Values predicted by each regressor.

Examples using `sklearn.ensemble.VotingRegressor`

- *Plot individual and voting regression predictions*

7.11.11 sklearn.ensemble.HistGradientBoostingRegressor

class sklearn.ensemble.**HistGradientBoostingRegressor**(*loss='least_squares'*, *learning_rate=0.1*, *max_iter=100*, *max_leaf_nodes=31*, *max_depth=None*, *min_samples_leaf=20*, *l2_regularization=0.0*, *max_bins=255*, *warm_start=False*, *scoring=None*, *validation_fraction=0.1*, *n_iter_no_change=None*, *tol=1e-07*, *verbose=0*, *random_state=None*)
Histogram-based Gradient Boosting Regression Tree.

This estimator is much faster than *GradientBoostingRegressor* for big datasets (n_samples >= 10 000).

This estimator has native support for missing values (NaNs). During training, the tree grower learns at each split point whether samples with missing values should go to the left or right child, based on the potential gain. When predicting, samples with missing values are assigned to the left or right child consequently. If no missing

values were encountered for a given feature during training, then samples with missing values are mapped to whichever child has the most samples.

This implementation is inspired by LightGBM.

Note: This estimator is still **experimental** for now: the predictions and the API might change without any deprecation cycle. To use it, you need to explicitly import `enable_hist_gradient_boosting`:

```
>>> # explicitly require this experimental feature
>>> from sklearn.experimental import enable_hist_gradient_boosting  # noqa
>>> # now you can import normally from ensemble
>>> from sklearn.ensemble import HistGradientBoostingClassifier
```

Read more in the *User Guide*.

New in version 0.21.

Parameters

loss [{'least_squares', 'least_absolute_deviation'}, optional (default='least_squares')] The loss function to use in the boosting process. Note that the "least squares" loss actually implements an "half least squares loss" to simplify the computation of the gradient.

learning_rate [float, optional (default=0.1)] The learning rate, also known as *shrinkage*. This is used as a multiplicative factor for the leaves values. Use `1` for no shrinkage.

max_iter [int, optional (default=100)] The maximum number of iterations of the boosting process, i.e. the maximum number of trees.

max_leaf_nodes [int or None, optional (default=31)] The maximum number of leaves for each tree. Must be strictly greater than 1. If None, there is no maximum limit.

max_depth [int or None, optional (default=None)] The maximum depth of each tree. The depth of a tree is the number of nodes to go from the root to the deepest leaf. Must be strictly greater than 1. Depth isn't constrained by default.

min_samples_leaf [int, optional (default=20)] The minimum number of samples per leaf. For small datasets with less than a few hundred samples, it is recommended to lower this value since only very shallow trees would be built.

l2_regularization [float, optional (default=0)] The L2 regularization parameter. Use `0` for no regularization (default).

max_bins [int, optional (default=255)] The maximum number of bins to use for non-missing values. Before training, each feature of the input array `X` is binned into integer-valued bins, which allows for a much faster training stage. Features with a small number of unique values may use less than `max_bins` bins. In addition to the `max_bins` bins, one more bin is always reserved for missing values. Must be no larger than 255.

warm_start [bool, optional (default=False)] When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble. For results to be valid, the estimator should be re-trained on the same data only. See *the Glossary*.

scoring [str or callable or None, optional (default=None)] Scoring parameter to use for early stopping. It can be a single string (see *The scoring parameter: defining model evaluation rules*) or a callable (see *Defining your scoring strategy from metric functions*). If None, the estimator's default scorer is used. If `scoring='loss'`, early stopping is checked w.r.t the loss value. Only used if `n_iter_no_change` is not None.

validation_fraction [int or float or None, optional (default=0.1)] Proportion (or absolute size) of training data to set aside as validation data for early stopping. If None, early stopping is done on the training data. Only used if `n_iter_no_change` is not None.

n_iter_no_change [int or None, optional (default=None)] Used to determine when to "early stop". The fitting process is stopped when none of the last `n_iter_no_change` scores are better than the `n_iter_no_change - 1` -th-to-last one, up to some tolerance. If None or 0, no early-stopping is done.

tol [float or None, optional (default=1e-7)] The absolute tolerance to use when comparing scores during early stopping. The higher the tolerance, the more likely we are to early stop: higher tolerance means that it will be harder for subsequent iterations to be considered an improvement upon the reference score.

verbose: int, optional (default=0) The verbosity level. If not zero, print some information about the fitting process.

random_state [int, np.random.RandomStateInstance or None, optional (default=None)] Pseudo-random number generator to control the subsampling in the binning process, and the train/validation data split if early stopping is enabled. See *random_state*.

Attributes

n_iter_ [int] The number of iterations as selected by early stopping (if n_iter_no_change is not None). Otherwise it corresponds to max_iter.

n_trees_per_iteration_ [int] The number of tree that are built at each iteration. For regressors, this is always 1.

train_score_ [ndarray, shape (n_iter_+1,)] The scores at each iteration on the training data. The first entry is the score of the ensemble before the first iteration. Scores are computed according to the `scoring` parameter. If `scoring` is not 'loss', scores are computed on a subset of at most 10 000 samples. Empty if no early stopping.

validation_score_ [ndarray, shape (n_iter_+1,)] The scores at each iteration on the held-out validation data. The first entry is the score of the ensemble before the first iteration. Scores are computed according to the `scoring` parameter. Empty if no early stopping or if `validation_fraction` is None.

Examples

```
>>> # To use this experimental feature, we need to explicitly ask for it:
>>> from sklearn.experimental import enable_hist_gradient_boosting  # noqa
>>> from sklearn.ensemble import HistGradientBoostingRegressor
>>> from sklearn.datasets import load_boston
>>> X, y = load_boston(return_X_y=True)
>>> est = HistGradientBoostingRegressor().fit(X, y)
>>> est.score(X, y)
0.98...
```

Methods

fit(self, X, y)	Fit the gradient boosting model.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict values for X.

Continued on next page

Table 81 – continued from previous page

score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *loss='least_squares'*, *learning_rate=0.1*, *max_iter=100*, *max_leaf_nodes=31*, *max_depth=None*, *min_samples_leaf=20*, *l2_regularization=0.0*, *max_bins=255*, *warm_start=False*, *scoring=None*, *validation_fraction=0.1*, *n_iter_no_change=None*, *tol=1e-07*, *verbose=0*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Fit the gradient boosting model.

Parameters

X [array-like of shape (n_samples, n_features)] The input samples.

y [array-like of shape (n_samples,)] Target values.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict values for X.

Parameters

X [array-like, shape (n_samples, n_features)] The input samples.

Returns

y [ndarray, shape (n_samples,)] The predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.ensemble.HistGradientBoostingRegressor`

- *Combine predictors using stacking*
- *Partial Dependence Plots*

7.11.12 `sklearn.ensemble`.HistGradientBoostingClassifier

class sklearn.ensemble.**HistGradientBoostingClassifier**(*loss='auto'*, *learning_rate=0.1*, *max_iter=100*, *max_leaf_nodes=31*, *max_depth=None*, *min_samples_leaf=20*, *l2_regularization=0.0*, *max_bins=255*, *warm_start=False*, *scoring=None*, *validation_fraction=0.1*, *n_iter_no_change=None*, *tol=1e-07*, *verbose=0*, *random_state=None*)

Histogram-based Gradient Boosting Classification Tree.

This estimator is much faster than *GradientBoostingClassifier* for big datasets (n_samples >= 10 000).

This estimator has native support for missing values (NaNs). During training, the tree grower learns at each split point whether samples with missing values should go to the left or right child, based on the potential gain. When predicting, samples with missing values are assigned to the left or right child consequently. If no missing

values were encountered for a given feature during training, then samples with missing values are mapped to whichever child has the most samples.

This implementation is inspired by LightGBM.

Note: This estimator is still **experimental** for now: the predictions and the API might change without any deprecation cycle. To use it, you need to explicitly import `enable_hist_gradient_boosting`:

```
>>> # explicitly require this experimental feature
>>> from sklearn.experimental import enable_hist_gradient_boosting  # noqa
>>> # now you can import normally from ensemble
>>> from sklearn.ensemble import HistGradientBoostingClassifier
```

Read more in the *User Guide*.

New in version 0.21.

Parameters

loss [{'auto', 'binary_crossentropy', 'categorical_crossentropy'}, optional (default='auto')] The loss function to use in the boosting process. 'binary_crossentropy' (also known as logistic loss) is used for binary classification and generalizes to 'categorical_crossentropy' for multiclass classification. 'auto' will automatically choose either loss depending on the nature of the problem.

learning_rate [float, optional (default=0.1)] The learning rate, also known as *shrinkage*. This is used as a multiplicative factor for the leaves values. Use `1` for no shrinkage.

max_iter [int, optional (default=100)] The maximum number of iterations of the boosting process, i.e. the maximum number of trees for binary classification. For multiclass classification, `n_classes` trees per iteration are built.

max_leaf_nodes [int or None, optional (default=31)] The maximum number of leaves for each tree. Must be strictly greater than 1. If None, there is no maximum limit.

max_depth [int or None, optional (default=None)] The maximum depth of each tree. The depth of a tree is the number of nodes to go from the root to the deepest leaf. Must be strictly greater than 1. Depth isn't constrained by default.

min_samples_leaf [int, optional (default=20)] The minimum number of samples per leaf. For small datasets with less than a few hundred samples, it is recommended to lower this value since only very shallow trees would be built.

l2_regularization [float, optional (default=0)] The L2 regularization parameter. Use 0 for no regularization.

max_bins [int, optional (default=255)] The maximum number of bins to use for non-missing values. Before training, each feature of the input array `X` is binned into integer-valued bins, which allows for a much faster training stage. Features with a small number of unique values may use less than `max_bins` bins. In addition to the `max_bins` bins, one more bin is always reserved for missing values. Must be no larger than 255.

warm_start [bool, optional (default=False)] When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble. For results to be valid, the estimator should be re-trained on the same data only. See *the Glossary*.

scoring [str or callable or None, optional (default=None)] Scoring parameter to use for early stopping. It can be a single string (see *The scoring parameter: defining model evaluation rules*) or a callable (see *Defining your scoring strategy from metric functions*). If None, the

estimator's default scorer is used. If `scoring='loss'`, early stopping is checked w.r.t the loss value. Only used if `n_iter_no_change` is not None.

validation_fraction [int or float or None, optional (default=0.1)] Proportion (or absolute size) of training data to set aside as validation data for early stopping. If None, early stopping is done on the training data.

n_iter_no_change [int or None, optional (default=None)] Used to determine when to "early stop". The fitting process is stopped when none of the last `n_iter_no_change` scores are better than the `n_iter_no_change - 1` -th-to-last one, up to some tolerance. If None or 0, no early-stopping is done.

tol [float or None, optional (default=1e-7)] The absolute tolerance to use when comparing scores. The higher the tolerance, the more likely we are to early stop: higher tolerance means that it will be harder for subsequent iterations to be considered an improvement upon the reference score.

verbose: int, optional (default=0) The verbosity level. If not zero, print some information about the fitting process.

random_state [int, np.random.RandomStateInstance or None, optional (default=None)] Pseudo-random number generator to control the subsampling in the binning process, and the train/validation data split if early stopping is enabled. See *random_state*.

Attributes

n_iter_ [int] The number of estimators as selected by early stopping (if n_iter_no_change is not None). Otherwise it corresponds to max_iter.

n_trees_per_iteration_ [int] The number of tree that are built at each iteration. This is equal to 1 for binary classification, and to `n_classes` for multiclass classification.

train_score_ [ndarray, shape (n_iter_+1,)] The scores at each iteration on the training data. The first entry is the score of the ensemble before the first iteration. Scores are computed according to the `scoring` parameter. If `scoring` is not 'loss', scores are computed on a subset of at most 10 000 samples. Empty if no early stopping.

validation_score_ [ndarray, shape (n_iter_+1,)] The scores at each iteration on the held-out validation data. The first entry is the score of the ensemble before the first iteration. Scores are computed according to the `scoring` parameter. Empty if no early stopping or if `validation_fraction` is None.

Examples

```
>>> # To use this experimental feature, we need to explicitly ask for it:
>>> from sklearn.experimental import enable_hist_gradient_boosting  # noqa
>>> from sklearn.ensemble import HistGradientBoostingClassifier
>>> from sklearn.datasets import load_iris
>>> X, y = load_iris(return_X_y=True)
>>> clf = HistGradientBoostingClassifier().fit(X, y)
>>> clf.score(X, y)
1.0
```

Methods

decision_function(self, X)	Compute the decision function of X.
fit(self, X, y)	Fit the gradient boosting model.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict classes for X.
predict_proba(self, X)	Predict class probabilities for X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *loss='auto'*, *learning_rate=0.1*, *max_iter=100*, *max_leaf_nodes=31*, *max_depth=None*, *min_samples_leaf=20*, *l2_regularization=0.0*, *max_bins=255*, *warm_start=False*, *scoring=None*, *validation_fraction=0.1*, *n_iter_no_change=None*, *tol=1e-07*, *verbose=0*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Compute the decision function of X.

Parameters

X [array-like, shape (n_samples, n_features)] The input samples.

Returns

decision [ndarray, shape (n_samples,) or (n_samples, n_trees_per_iteration)] The raw predicted values (i.e. the sum of the trees leaves) for each sample. n_trees_per_iteration is equal to the number of classes in multiclass classification.

fit(*self*, *X*, *y*)
Fit the gradient boosting model.

Parameters

X [array-like of shape (n_samples, n_features)] The input samples.

y [array-like of shape (n_samples,)] Target values.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict classes for X.

Parameters

X [array-like, shape (n_samples, n_features)] The input samples.

Returns

y [ndarray, shape (n_samples,)] The predicted classes.

predict_proba(*self*, *X*)
Predict class probabilities for X.

Parameters

X [array-like, shape (n_samples, n_features)] The input samples.

Returns

p [ndarray, shape (n_samples, n_classes)] The class probabilities of the input samples.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.ensemble.HistGradientBoostingClassifier`

- *Release Highlights for scikit-learn 0.22*

7.12 `sklearn.exceptions`: Exceptions and warnings

The *`sklearn.exceptions`* module includes all custom warnings and error classes used across scikit-learn.

`exceptions.ChangedBehaviorWarning`	Warning class used to notify the user of any change in the behavior.
`exceptions.ConvergenceWarning`	Custom warning to capture convergence problems
`exceptions.DataConversionWarning`	Warning used to notify implicit data conversions happening in the code.

Continued on next page

Table 84 – continued from previous page

exceptions.DataDimensionalityWarning	Custom warning to notify potential issues with data dimensionality.
exceptions.EfficiencyWarning	Warning used to notify the user of inefficient computation.
exceptions.FitFailedWarning	Warning class used if there is an error while fitting the estimator.
exceptions.NotFittedError	Exception class to raise if estimator is used before fitting.
exceptions.NonBLASDotWarning	Warning used when the dot operation does not use BLAS.
exceptions.UndefinedMetricWarning	Warning used when the metric is invalid

7.12.1 sklearn.exceptions.ChangedBehaviorWarning

class sklearn.exceptions.**ChangedBehaviorWarning**

Warning class used to notify the user of any change in the behavior.

Changed in version 0.18: Moved from sklearn.base.

Attributes

args

Methods

with_traceback()	Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

7.12.2 sklearn.exceptions.ConvergenceWarning

class sklearn.exceptions.**ConvergenceWarning**

Custom warning to capture convergence problems

Attributes

args

Examples

```
>>> import numpy as np
>>> import warnings
>>> from sklearn.cluster import KMeans
>>> from sklearn.exceptions import ConvergenceWarning
>>> warnings.simplefilter("always", ConvergenceWarning)
>>> X = np.asarray([[0, 0],
...                 [0, 1],
...                 [1, 0],
```

(continues on next page)

(continued from previous page)

```
...                 [1, 0]])  # last point is duplicated
>>> with warnings.catch_warnings(record=True) as w:
...     km = KMeans(n_clusters=4).fit(X)
...     print(w[-1].message)
Number of distinct clusters (3) found smaller than n_clusters (4).
Possibly due to duplicate points in X.
```

Changed in version 0.18: Moved from sklearn.utils.

Methods

with_traceback()	Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

Examples using `sklearn.exceptions.ConvergenceWarning`

- *Multiclass sparse logistic regression on 20newgroups*
- *Early stopping of Stochastic Gradient Descent*
- *Compare Stochastic learning strategies for MLPClassifier*
- *Feature discretization*

7.12.3 `sklearn.exceptions`.DataConversionWarning

class sklearn.exceptions.**DataConversionWarning**
Warning used to notify implicit data conversions happening in the code.

This warning occurs when some input data needs to be converted or interpreted in a way that may not match the user's expectations.

For example, this warning may occur when the user

- passes an integer array to a function which expects float input and will convert the input
- requests a non-copying operation, but a copy is required to meet the implementation's data-type expectations;
- passes an input whose shape can be interpreted ambiguously.

Changed in version 0.18: Moved from sklearn.utils.validation.

Attributes

args

Methods

with_traceback()	Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

7.12.4 sklearn.exceptions.DataDimensionalityWarning

class sklearn.exceptions.**DataDimensionalityWarning**
Custom warning to notify potential issues with data dimensionality.

For example, in random projection, this warning is raised when the number of components, which quantifies the dimensionality of the target projection space, is higher than the number of features, which quantifies the dimensionality of the original source space, to imply that the dimensionality of the problem will not be reduced.

Changed in version 0.18: Moved from sklearn.utils.

Attributes

args

Methods

with_traceback()	Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

7.12.5 sklearn.exceptions.EfficiencyWarning

class sklearn.exceptions.**EfficiencyWarning**
Warning used to notify the user of inefficient computation.

This warning notifies the user that the efficiency may not be optimal due to some reason which may be included as a part of the warning message. This may be subclassed into a more specific Warning class.

New in version 0.18.

Attributes

args

Methods

with_traceback()	Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

7.12.6 sklearn.exceptions.FitFailedWarning

class sklearn.exceptions.**FitFailedWarning**

Warning class used if there is an error while fitting the estimator.

This Warning is used in meta estimators GridSearchCV and RandomizedSearchCV and the cross-validation helper function cross_val_score to warn when there is an error while fitting the estimator.

Attributes

args

Examples

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.svm import LinearSVC
>>> from sklearn.exceptions import FitFailedWarning
>>> import warnings
>>> warnings.simplefilter('always', FitFailedWarning)
>>> gs = GridSearchCV(LinearSVC(), {'C': [-1, -2]}, error_score=0, cv=2)
>>> X, y = [[1, 2], [3, 4], [5, 6], [7, 8]], [0, 0, 1, 1]
>>> with warnings.catch_warnings(record=True) as w:
...     try:
...         gs.fit(X, y)  # This will raise a ValueError since C is < 0
...     except ValueError:
...         pass
...     print(repr(w[-1].message))
FitFailedWarning('Estimator fit failed. The score on this train-test
partition for these parameters will be set to 0.000000.
Details:...ValueError: Penalty term must be positive; got (C=-2)...
```

Changed in version 0.18: Moved from sklearn.cross_validation.

Methods

with_traceback()	Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

7.12.7 sklearn.exceptions.NotFittedError

class sklearn.exceptions.**NotFittedError**

Exception class to raise if estimator is used before fitting.

This class inherits from both ValueError and AttributeError to help with exception handling and backward compatibility.

Attributes

args

Examples

```
>>> from sklearn.svm import LinearSVC
>>> from sklearn.exceptions import NotFittedError
>>> try:
...     LinearSVC().predict([[1, 2], [2, 3], [3, 4]])
... except NotFittedError as e:
...     print(repr(e))
NotFittedError("This LinearSVC instance is not fitted yet. Call 'fit' with
appropriate arguments before using this estimator."...)
```

Changed in version 0.18: Moved from sklearn.utils.validation.

Methods

with_traceback()	Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

7.12.8 sklearn.exceptions.NonBLASDotWarning

class sklearn.exceptions.**NonBLASDotWarning**
Warning used when the dot operation does not use BLAS.

This warning is used to notify the user that BLAS was not used for dot operation and hence the efficiency may be affected.

Changed in version 0.18: Moved from sklearn.utils.validation, extends EfficiencyWarning.

Attributes

args

Methods

with_traceback()	Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

7.12.9 sklearn.exceptions.UndefinedMetricWarning

class sklearn.exceptions.**UndefinedMetricWarning**
Warning used when the metric is invalid

Changed in version 0.18: Moved from sklearn.base.

Attributes

args

Methods

with_traceback()	Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

7.13 sklearn.experimental: Experimental

The *sklearn.experimental* module provides importable modules that enable the use of experimental features or estimators.

The features and estimators that are experimental aren't subject to deprecation cycles. Use them at your own risks!

experimental.enable_hist_gradient_boosting	Enables histogram-based gradient boosting estimators.
experimental.enable_iterative_imputer	Enables IterativeImputer

7.13.1 sklearn.experimental.enable_hist_gradient_boosting

Enables histogram-based gradient boosting estimators.

The API and results of these estimators might change without any deprecation cycle.

Importing this file dynamically sets the *sklearn.ensemble.HistGradientBoostingClassifier* and *sklearn.ensemble.HistGradientBoostingRegressor* as attributes of the ensemble module:

```
>>> # explicitly require this experimental feature
>>> from sklearn.experimental import enable_hist_gradient_boosting  # noqa
>>> # now you can import normally from ensemble
>>> from sklearn.ensemble import HistGradientBoostingClassifier
>>> from sklearn.ensemble import HistGradientBoostingRegressor
```

The `# noqa` comment comment can be removed: it just tells linters like flake8 to ignore the import, which appears as unused.

7.13.2 sklearn.experimental.enable_iterative_imputer

Enables IterativeImputer

The API and results of this estimator might change without any deprecation cycle.

Importing this file dynamically sets *sklearn.impute.IterativeImputer* as an attribute of the impute module:

```
>>> # explicitly require this experimental feature
>>> from sklearn.experimental import enable_iterative_imputer  # noqa
>>> # now you can import normally from impute
>>> from sklearn.impute import IterativeImputer
```

7.14 sklearn.feature_extraction: Feature Extraction

The *sklearn.feature_extraction* module deals with feature extraction from raw data. It currently includes methods to extract features from text and images.

User guide: See the *Feature extraction* section for further details.

feature_extraction.DictVectorizer([dtype, ...])	Transforms lists of feature-value mappings to vectors.
feature_extraction.FeatureHasher([...])	Implements feature hashing, aka the hashing trick.

7.14.1 sklearn.feature_extraction.DictVectorizer

class sklearn.feature_extraction.**DictVectorizer**(*dtype=<class 'numpy.float64'>*, *separator='='*, *sparse=True*, *sort=True*)

Transforms lists of feature-value mappings to vectors.

This transformer turns lists of mappings (dict-like objects) of feature names to feature values into Numpy arrays or scipy.sparse matrices for use with scikit-learn estimators.

When feature values are strings, this transformer will do a binary one-hot (aka one-of-K) coding: one boolean-valued feature is constructed for each of the possible string values that the feature can take on. For instance, a feature "f" that can take on the values "ham" and "spam" will become two features in the output, one signifying "f=ham", the other "f=spam".

However, note that this transformer will only do a binary one-hot encoding when feature values are of type string. If categorical features are represented as numeric values such as int, the DictVectorizer can be followed by *sklearn.preprocessing.OneHotEncoder* to complete binary one-hot encoding.

Features that do not occur in a sample (mapping) will have a zero value in the resulting array/matrix.

Read more in the *User Guide*.

Parameters

dtype [callable, optional] The type of feature values. Passed to Numpy array/scipy.sparse matrix constructors as the dtype argument.

separator [string, optional] Separator string used when constructing new features for one-hot coding.

sparse [boolean, optional.] Whether transform should produce scipy.sparse matrices. True by default.

sort [boolean, optional.] Whether `feature_names_` and `vocabulary_` should be sorted when fitting. True by default.

Attributes

vocabulary_ [dict] A dictionary mapping feature names to feature indices.

feature_names_ [list] A list of length n_features containing the feature names (e.g., "f=ham" and "f=spam").

See also:

FeatureHasher performs vectorization using only a hash function.

sklearn.preprocessing.OrdinalEncoder handles nominal/categorical features encoded as columns of arbitrary data types.

Examples

```
>>> from sklearn.feature_extraction import DictVectorizer
>>> v = DictVectorizer(sparse=False)
>>> D = [{'foo': 1, 'bar': 2}, {'foo': 3, 'baz': 1}]
>>> X = v.fit_transform(D)
>>> X
array([[2., 0., 1.],
       [0., 1., 3.]])
>>> v.inverse_transform(X) ==         [{'bar': 2.0, 'foo': 1.0}, {'baz': 1.0, 'foo
↪': 3.0}]
True
>>> v.transform({'foo': 4, 'unseen_feature': 3})
array([[0., 0., 4.]])
```

Methods

fit(self, X[, y])	Learn a list of feature name -> indices mappings.
fit_transform(self, X[, y])	Learn a list of feature name -> indices mappings and transform X.
get_feature_names(self)	Returns a list of feature names, ordered by their indices.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X[, dict_type])	Transform array or sparse matrix X back to feature mappings.
restrict(self, support[, indices])	Restrict the features to those in support using feature selection.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform feature->value dicts to array or sparse matrix.

__init__(*self*, *dtype=<class 'numpy.float64'>*, *separator='='*, *sparse=True*, *sort=True*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Learn a list of feature name -> indices mappings.

Parameters

X [Mapping or iterable over Mappings] Dict(s) or Mapping(s) from feature names (arbitrary Python objects) to feature values (strings or convertible to dtype).

y [(ignored)]

Returns

self

fit_transform(*self*, *X*, *y=None*)
Learn a list of feature name -> indices mappings and transform X.

Like fit(X) followed by transform(X), but does not require materializing X in memory.

Parameters

X [Mapping or iterable over Mappings] Dict(s) or Mapping(s) from feature names (arbitrary Python objects) to feature values (strings or convertible to dtype).

y [(ignored)]

Returns

Xa [{array, sparse matrix}] Feature vectors; always 2-d.

get_feature_names(*self*)
Returns a list of feature names, ordered by their indices.

If one-of-K coding is applied to categorical features, this will include the constructed feature names but not the original ones.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*, *dict_type=<class 'dict'>*)
Transform array or sparse matrix X back to feature mappings.

X must have been produced by this DictVectorizer's transform or fit_transform method; it may only have passed through transformers that preserve the number of features and their order.

In the case of one-hot/one-of-K coding, the constructed feature names and values are returned rather than the original ones.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Sample matrix.

dict_type [callable, optional] Constructor for feature mappings. Must conform to the collections.Mapping API.

Returns

D [list of dict_type objects, length = n_samples] Feature mappings for the samples in X.

restrict(*self*, *support*, *indices=False*)
Restrict the features to those in support using feature selection.

This function modifies the estimator in-place.

Parameters

support [array-like] Boolean mask or list of indices (as returned by the get_support member of feature selectors).

indices [boolean, optional] Whether support is a list of indices.

Returns

self

Examples

```
>>> from sklearn.feature_extraction import DictVectorizer
>>> from sklearn.feature_selection import SelectKBest, chi2
>>> v = DictVectorizer()
>>> D = [{'foo': 1, 'bar': 2}, {'foo': 3, 'baz': 1}]
>>> X = v.fit_transform(D)
>>> support = SelectKBest(chi2, k=2).fit(X, [0, 1])
>>> v.get_feature_names()
['bar', 'baz', 'foo']
>>> v.restrict(support.get_support())
DictVectorizer()
>>> v.get_feature_names()
['bar', 'foo']
```

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Transform feature->value dicts to array or sparse matrix.

Named features not encountered during fit or fit_transform will be silently ignored.

Parameters

X [Mapping or iterable over Mappings, length = n_samples] Dict(s) or Mapping(s) from feature names (arbitrary Python objects) to feature values (strings or convertible to dtype).

Returns

Xa [{array, sparse matrix}] Feature vectors; always 2-d.

Examples using sklearn.feature_extraction.DictVectorizer

- *Column Transformer with Heterogeneous Data Sources*
- *FeatureHasher and DictVectorizer Comparison*

7.14.2 sklearn.feature_extraction.FeatureHasher

class sklearn.feature_extraction.**FeatureHasher**(*n_features=1048576*, *input_type='dict'*, *dtype=<class 'numpy.float64'>*, *alternate_sign=True*)
Implements feature hashing, aka the hashing trick.

This class turns sequences of symbolic feature names (strings) into scipy.sparse matrices, using a hash function to compute the matrix column corresponding to a name. The hash function employed is the signed 32-bit version of Murmurhash3.

Feature names of type byte string are used as-is. Unicode strings are converted to UTF-8 first, but no Unicode normalization is done. Feature values must be (finite) numbers.

This class is a low-memory alternative to DictVectorizer and CountVectorizer, intended for large-scale (online) learning and situations where memory is tight, e.g. when running prediction code on embedded devices.

Read more in the *User Guide*.

New in version 0.13.

Parameters

n_features [integer, optional] The number of features (columns) in the output matrices. Small numbers of features are likely to cause hash collisions, but large numbers will cause larger coefficient dimensions in linear learners.

input_type [string, optional, default "dict"] Either "dict" (the default) to accept dictionaries over (feature_name, value); "pair" to accept pairs of (feature_name, value); or "string" to accept single strings. feature_name should be a string, while value should be a number. In the case of "string", a value of 1 is implied. The feature_name is hashed to find the appropriate column for the feature. The value's sign might be flipped in the output (but see non_negative, below).

dtype [numpy type, optional, default np.float64] The type of feature values. Passed to scipy.sparse matrix constructors as the dtype argument. Do not set this to bool, np.boolean or any unsigned integer type.

alternate_sign [boolean, optional, default True] When True, an alternating sign is added to the features as to approximately conserve the inner product in the hashed space even for small n_features. This approach is similar to sparse random projection.

See also:

DictVectorizer vectorizes string-valued features using a hash table.

sklearn.preprocessing.OneHotEncoder handles nominal/categorical features.

Examples

```
>>> from sklearn.feature_extraction import FeatureHasher
>>> h = FeatureHasher(n_features=10)
>>> D = [{'dog': 1, 'cat':2, 'elephant':4},{'dog': 2, 'run': 5}]
>>> f = h.transform(D)
>>> f.toarray()
array([[ 0.,  0., -4., -1.,  0.,  0.,  0.,  0.,  0.,  2.],
       [ 0.,  0.,  0., -2., -5.,  0.,  0.,  0.,  0.,  0.]])
```

Methods

fit(self[, X, y])	No-op.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, raw_X)	Transform a sequence of instances to a scipy.sparse matrix.

__init__(*self*, *n_features=1048576*, *input_type='dict'*, *dtype=<class 'numpy.float64'>*, *alternate_sign=True*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X=None*, *y=None*)
No-op.

This method doesn't do anything. It exists purely for compatibility with the scikit-learn transformer API.

Parameters

X [array-like]

Returns

self [FeatureHasher]

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *raw_X*)
Transform a sequence of instances to a scipy.sparse matrix.

Parameters

raw_X [iterable over iterable over raw features, length = n_samples] Samples. Each sample must be iterable an (e.g., a list or tuple) containing/generating feature names (and optionally values, see the input_type constructor argument) which will be hashed. raw_X need not support the len function, so it can be the result of a generator; n_samples is determined on the fly.

Returns

X [sparse matrix of shape (n_samples, n_features)] Feature matrix, for use with estimators or further transformers.

Examples using `sklearn.feature_extraction.FeatureHasher`

- *FeatureHasher and DictVectorizer Comparison*

7.14.3 From images

The *sklearn.feature_extraction.image* submodule gathers utilities to extract features from images.

feature_extraction.image.extract_patches_2d(...)	Reshape a 2D image into a collection of patches
feature_extraction.image.grid_to_graph(n_x, n_y)	Graph of the pixel-to-pixel connections
feature_extraction.image.img_to_graph(img[, ...])	Graph of the pixel-to-pixel gradient connections
feature_extraction.image.reconstruct_from_patches_2d(...)	Reconstruct the image from all of its patches.
feature_extraction.image.PatchExtractor([...])	Extracts patches from a collection of images

sklearn.feature_extraction.image.extract_patches_2d

sklearn.feature_extraction.image.**extract_patches_2d**(*image*, *patch_size*, *max_patches=None*, *random_state=None*)

Reshape a 2D image into a collection of patches

The resulting patches are allocated in a dedicated array.

Read more in the *User Guide*.

Parameters

image [array, shape = (image_height, image_width) or] (image_height, image_width, n_channels) The original image data. For color images, the last dimension specifies the channel: a RGB image would have `n_channels=3`.

patch_size [tuple of ints (patch_height, patch_width)] the dimensions of one patch

max_patches [integer or float, optional default is None] The maximum number of patches to extract. If max_patches is a float between 0 and 1, it is taken to be a proportion of the total number of patches.

random_state [int, RandomState instance or None, optional (default=None)] Determines the random number generator used for random sampling when `max_patches` is not None. Use an int to make the randomness deterministic. See *Glossary*.

Returns

patches [array, shape = (n_patches, patch_height, patch_width) or] (n_patches, patch_height, patch_width, n_channels) The collection of patches extracted from the image, where `n_patches` is either `max_patches` or the total number of patches that can be extracted.

Examples

```
>>> from sklearn.datasets import load_sample_image
>>> from sklearn.feature_extraction import image
>>> # Use the array data from the first image in this dataset:
>>> one_image = load_sample_image("china.jpg")
>>> print('Image shape: {}'.format(one_image.shape))
Image shape: (427, 640, 3)
>>> patches = image.extract_patches_2d(one_image, (2, 2))
>>> print('Patches shape: {}'.format(patches.shape))
Patches shape: (272214, 2, 2, 3)
>>> # Here are just two of these patches:
>>> print(patches[1])
[[[174 201 231]
  [174 201 231]]
 [[173 200 230]
  [173 200 230]]]
>>> print(patches[800])
[[[187 214 243]
  [188 215 244]]
 [[187 214 243]
  [188 215 244]]]
```

Examples using `sklearn.feature_extraction.image.extract_patches_2d`

- *Online learning of a dictionary of parts of faces*
- *Image denoising using dictionary learning*

`sklearn.feature_extraction.image.grid_to_graph`

`sklearn.feature_extraction.image.`**`grid_to_graph`**(*n_x*, *n_y*, *n_z=1*, *mask=None*, *return_as=<class 'scipy.sparse.coo.coo_matrix'>*, *dtype=<class 'int'>*)

Graph of the pixel-to-pixel connections

Edges exist if 2 voxels are connected.

Parameters

n_x [int] Dimension in x axis

n_y [int] Dimension in y axis

n_z [int, optional, default 1] Dimension in z axis

mask [ndarray of booleans, optional] An optional mask of the image, to consider only part of the pixels.

return_as [np.ndarray or a sparse matrix class, optional] The class to use to build the returned adjacency matrix.

dtype [dtype, optional, default int] The data of the returned sparse matrix. By default it is int

Notes

For scikit-learn versions 0.14.1 and prior, return_as=np.ndarray was handled by returning a dense np.matrix instance. Going forward, np.ndarray returns an np.ndarray, as expected.

For compatibility, user code relying on this method should wrap its calls in `np.asarray` to avoid type issues.

sklearn.feature_extraction.image.img_to_graph

sklearn.feature_extraction.image.**img_to_graph**(*img*, *mask=None*, *return_as=<class 'scipy.sparse.coo.coo_matrix'>*, *dtype=None*)

Graph of the pixel-to-pixel gradient connections

Edges are weighted with the gradient values.

Read more in the *User Guide*.

Parameters

img [ndarray, 2D or 3D] 2D or 3D image

mask [ndarray of booleans, optional] An optional mask of the image, to consider only part of the pixels.

return_as [np.ndarray or a sparse matrix class, optional] The class to use to build the returned adjacency matrix.

dtype [None or dtype, optional] The data of the returned sparse matrix. By default it is the dtype of img

Notes

For scikit-learn versions 0.14.1 and prior, return_as=np.ndarray was handled by returning a dense np.matrix instance. Going forward, np.ndarray returns an np.ndarray, as expected.

For compatibility, user code relying on this method should wrap its calls in `np.asarray` to avoid type issues.

sklearn.feature_extraction.image.reconstruct_from_patches_2d

sklearn.feature_extraction.image.**reconstruct_from_patches_2d**(*patches*, *image_size*)

Reconstruct the image from all of its patches.

Patches are assumed to overlap and the image is constructed by filling in the patches from left to right, top to bottom, averaging the overlapping regions.

Read more in the *User Guide*.

Parameters

patches [array, shape = (n_patches, patch_height, patch_width) or] (n_patches, patch_height, patch_width, n_channels) The complete set of patches. If the patches contain colour information, channels are indexed along the last dimension: RGB patches would have `n_channels=3`.

image_size [tuple of ints (image_height, image_width) or] (image_height, image_width, n_channels) the size of the image that will be reconstructed

Returns

image [array, shape = image_size] the reconstructed image

Examples using `sklearn.feature_extraction.image.reconstruct_from_patches_2d`

- *Image denoising using dictionary learning*

`sklearn.feature_extraction.image.PatchExtractor`

class sklearn.feature_extraction.image.**PatchExtractor**(*patch_size=None*, *max_patches=None*, *random_state=None*)

Extracts patches from a collection of images

Read more in the *User Guide*.

New in version 0.9.

Parameters

patch_size [tuple of ints (patch_height, patch_width)] the dimensions of one patch

max_patches [integer or float, optional default is None] The maximum number of patches per image to extract. If max_patches is a float in (0, 1), it is taken to mean a proportion of the total number of patches.

random_state [int, RandomState instance or None, optional (default=None)] Determines the random number generator used for random sampling when `max_patches` is not None. Use an int to make the randomness deterministic. See *Glossary*.

Examples

```
>>> from sklearn.datasets import load_sample_images
>>> from sklearn.feature_extraction import image
>>> # Use the array data from the second image in this dataset:
>>> X = load_sample_images().images[1]
>>> print('Image shape: {}'.format(X.shape))
Image shape: (427, 640, 3)
>>> pe = image.PatchExtractor(patch_size=(2, 2))
>>> pe_fit = pe.fit(X)
>>> pe_trans = pe.transform(X)
>>> print('Patches shape: {}'.format(pe_trans.shape))
Patches shape: (545706, 2, 2)
```

Methods

fit(self, X[, y])	Do nothing and return the estimator unchanged
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transforms the image samples in X into a matrix of patch data.

__init__ (*self*, *patch_size=None*, *max_patches=None*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*)
Do nothing and return the estimator unchanged

This method is just there to implement the usual API and hence work in pipelines.

Parameters

X [array-like, shape [n_samples, n_features]] Training data.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform (*self*, *X*)
Transforms the image samples in X into a matrix of patch data.

Parameters

X [array, shape = (n_samples, image_height, image_width) or] (n_samples, image_height, image_width, n_channels) Array of images from which to extract patches. For color images, the last dimension specifies the channel: a RGB image would have `n_channels=3`.

Returns

patches [array, shape = (n_patches, patch_height, patch_width) or] (n_patches, patch_height, patch_width, n_channels) The collection of patches extracted from the images, where `n_patches` is either `n_samples * max_patches` or the total number of patches that can be extracted.

7.14.4 From text

The *sklearn.feature_extraction.text* submodule gathers utilities to build feature vectors from text documents.

feature_extraction.text.CountVectorizer([...])	Convert a collection of text documents to a matrix of token counts
feature_extraction.text.HashingVectorizer([...])	Convert a collection of text documents to a matrix of token occurrences
feature_extraction.text.TfidfTransformer([...])	Transform a count matrix to a normalized tf or tf-idf representation
feature_extraction.text.TfidfVectorizer([...])	Convert a collection of raw documents to a matrix of TF-IDF features.

sklearn.feature_extraction.text.CountVectorizer

class sklearn.feature_extraction.text.**CountVectorizer**(*input='content'*, *encoding='utf-8'*, *decode_error='strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *stop_words=None*, *token_pattern='(?u)\b\w\w+\b'*, *ngram_range=(1, 1)*, *analyzer='word'*, *max_df=1.0*, *min_df=1*, *max_features=None*, *vocabulary=None*, *binary=False*, *dtype=<class 'numpy.int64'>*)

Convert a collection of text documents to a matrix of token counts

This implementation produces a sparse representation of the counts using scipy.sparse.csr_matrix.

If you do not provide an a-priori dictionary and you do not use an analyzer that does some kind of feature selection then the number of features will be equal to the vocabulary size found by analyzing the data.

Read more in the *User Guide*.

Parameters

input [string {'filename', 'file', 'content'}] If 'filename', the sequence passed as an argument to fit is expected to be a list of filenames that need reading to fetch the raw content to analyze.

If 'file', the sequence items must have a 'read' method (file-like object) that is called to fetch the bytes in memory.

Otherwise the input is expected to be a sequence of items that can be of type string or byte.

encoding [string, 'utf-8' by default.] If bytes or files are given to analyze, this encoding is used to decode.

decode_error [{'strict', 'ignore', 'replace'}] Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given `encoding`. By default, it is 'strict', meaning that a UnicodeDecodeError will be raised. Other values are 'ignore' and 'replace'.

strip_accents [{'ascii', 'unicode', None}] Remove accents and perform other character normalization during the preprocessing step. 'ascii' is a fast method that only works on characters that have an direct ASCII mapping. 'unicode' is a slightly slower method that works on any characters. None (default) does nothing.

Both 'ascii' and 'unicode' use NFKD normalization from `unicodedata.normalize`.

lowercase [boolean, True by default] Convert all characters to lowercase before tokenizing.

preprocessor [callable or None (default)] Override the preprocessing (string transformation) stage while preserving the tokenizing and n-grams generation steps. Only applies if `analyzer is not callable`.

tokenizer [callable or None (default)] Override the string tokenization step while preserving the preprocessing and n-grams generation steps. Only applies if `analyzer == 'word'`.

stop_words [string {'english'}, list, or None (default)] If 'english', a built-in stop word list for English is used. There are several known issues with 'english' and you should consider an alternative (see *Using stop words*).

If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens. Only applies if `analyzer == 'word'`.

If None, no stop words will be used. max_df can be set to a value in the range [0.7, 1.0) to automatically detect and filter stop words based on intra corpus document frequency of terms.

token_pattern [string] Regular expression denoting what constitutes a "token", only used if `analyzer == 'word'`. The default regexp select tokens of 2 or more alphanumeric characters (punctuation is completely ignored and always treated as a token separator).

ngram_range [tuple (min_n, max_n), default=(1, 1)] The lower and upper boundary of the range of n-values for different word n-grams or char n-grams to be extracted. All values of n such such that min_n <= n <= max_n will be used. For example an `ngram_range` of `(1, 1)` means only unigrams, `(1, 2)` means unigrams and bigrams, and `(2, 2)` means only bigrams. Only applies if `analyzer is not callable`.

analyzer [string, {'word', 'char', 'char_wb'} or callable] Whether the feature should be made of word n-gram or character n-grams. Option 'char_wb' creates character n-grams only from text inside word boundaries; n-grams at the edges of words are padded with space.

If a callable is passed it is used to extract the sequence of features out of the raw, unprocessed input.

Changed in version 0.21.

Since v0.21, if `input` is `filename` or `file`, the data is first read from the file and then passed to the given callable analyzer.

max_df [float in range [0.0, 1.0] or int, default=1.0] When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold (corpus-specific stop words). If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

min_df [float in range [0.0, 1.0] or int, default=1] When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold. This value is also called cut-off in the literature. If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

max_features [int or None, default=None] If not None, build a vocabulary that only consider the top max_features ordered by term frequency across the corpus.

This parameter is ignored if vocabulary is not None.

vocabulary [Mapping or iterable, optional] Either a Mapping (e.g., a dict) where keys are terms and values are indices in the feature matrix, or an iterable over terms. If not given, a vocabulary is determined from the input documents. Indices in the mapping should not be repeated and should not have any gap between 0 and the largest index.

binary [boolean, default=False] If True, all non zero counts are set to 1. This is useful for discrete probabilistic models that model binary events rather than integer counts.

dtype [type, optional] Type of the matrix returned by fit_transform() or transform().

Attributes

vocabulary_ [dict] A mapping of terms to feature indices.

fixed_vocabulary_: boolean True if a fixed vocabulary of term to indices mapping is provided by the user

stop_words_ [set] Terms that were ignored because they either:

- occurred in too many documents (`max_df`)
- occurred in too few documents (`min_df`)
- were cut off by feature selection (`max_features`).

This is only available if no vocabulary was given.

See also:

HashingVectorizer, ***TfidfVectorizer***

Notes

The `stop_words_` attribute can get large and increase the model size when pickling. This attribute is provided only for introspection and can be safely removed using delattr or set to None before pickling.

Examples

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = [
...     'This is the first document.',
...     'This document is the second document.',
...     'And this is the third one.',
...     'Is this the first document?',
... ]
>>> vectorizer = CountVectorizer()
>>> X = vectorizer.fit_transform(corpus)
>>> print(vectorizer.get_feature_names())
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
>>> print(X.toarray())
[[0 1 1 1 0 0 1 0 1]
 [0 2 0 1 0 1 1 0 1]
 [1 0 0 1 1 0 1 1 1]
 [0 1 1 1 0 0 1 0 1]]
>>> vectorizer2 = CountVectorizer(analyzer='word', ngram_range=(2, 2))
>>> X2 = vectorizer2.fit_transform(corpus)
>>> print(vectorizer2.get_feature_names())
```

(continues on next page)

(continued from previous page)

```
['and this', 'document is', 'first document', 'is the', 'is this',
'second document', 'the first', 'the second', 'the third', 'third one',
 'this document', 'this is', 'this the']
 >>> print(X2.toarray())
 [[0 0 1 1 0 0 1 0 0 0 0 1 0]
 [0 1 0 1 0 1 0 1 0 0 1 0 0]
 [1 0 0 1 0 0 0 0 1 1 0 1 0]
 [0 0 1 0 1 0 1 0 0 0 0 0 1]]
```

Methods

build_analyzer(self)	Return a callable that handles preprocessing, tokenization and n-grams generation.
build_preprocessor(self)	Return a function to preprocess the text before tokenization.
build_tokenizer(self)	Return a function that splits a string into a sequence of tokens.
decode(self, doc)	Decode the input into a string of unicode symbols.
fit(self, raw_documents[, y])	Learn a vocabulary dictionary of all tokens in the raw documents.
fit_transform(self, raw_documents[, y])	Learn the vocabulary dictionary and return term-document matrix.
get_feature_names(self)	Array mapping from feature integer indices to feature name.
get_params(self[, deep])	Get parameters for this estimator.
get_stop_words(self)	Build or fetch the effective stop words list.
inverse_transform(self, X)	Return terms per document with nonzero entries in X.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, raw_documents)	Transform documents to document-term matrix.

__init__(*self*, *input='content'*, *encoding='utf-8'*, *decode_error='strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *stop_words=None*, *token_pattern='(?u)\b\w\w+\b'*, *ngram_range=(1, 1)*, *analyzer='word'*, *max_df=1.0*, *min_df=1*, *max_features=None*, *vocabulary=None*, *binary=False*, *dtype=<class 'numpy.int64'>*)

Initialize self. See help(type(self)) for accurate signature.

build_analyzer(*self*)

Return a callable that handles preprocessing, tokenization and n-grams generation.

Returns

analyzer: callable A function to handle preprocessing, tokenization and n-grams generation.

build_preprocessor(*self*)

Return a function to preprocess the text before tokenization.

Returns

preprocessor: callable A function to preprocess the text before tokenization.

build_tokenizer(*self*)

Return a function that splits a string into a sequence of tokens.

Returns

tokenizer: callable A function to split a string into a sequence of tokens.

decode(*self*, *doc*)
Decode the input into a string of unicode symbols.

The decoding strategy depends on the vectorizer parameters.

Parameters

doc [str] The string to decode.

Returns

doc: str A string of unicode symbols.

fit(*self*, *raw_documents*, *y=None*)
Learn a vocabulary dictionary of all tokens in the raw documents.

Parameters

raw_documents [iterable] An iterable which yields either str, unicode or file objects.

Returns

self

fit_transform(*self*, *raw_documents*, *y=None*)
Learn the vocabulary dictionary and return term-document matrix.

This is equivalent to fit followed by transform, but more efficiently implemented.

Parameters

raw_documents [iterable] An iterable which yields either str, unicode or file objects.

Returns

X [array, [n_samples, n_features]] Document-term matrix.

get_feature_names(*self*)
Array mapping from feature integer indices to feature name.

Returns

feature_names [list] A list of feature names.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_stop_words(*self*)
Build or fetch the effective stop words list.

Returns

stop_words: list or None A list of stop words.

inverse_transform(*self*, *X*)
Return terms per document with nonzero entries in X.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Document-term matrix.

Returns

X_inv [list of arrays, len = n_samples] List of arrays of terms.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *raw_documents*)
Transform documents to document-term matrix.

Extract token counts out of raw text documents using the vocabulary fitted with fit or the one provided to the constructor.

Parameters

raw_documents [iterable] An iterable which yields either str, unicode or file objects.

Returns

X [sparse matrix, [n_samples, n_features]] Document-term matrix.

Examples using `sklearn.feature_extraction.text.CountVectorizer`

- *Sample pipeline for text feature extraction and evaluation*

sklearn.feature_extraction.text.HashingVectorizer

class sklearn.feature_extraction.text.**HashingVectorizer**(*input='content', encoding='utf-8', decode_error='strict', strip_accents=None, lowercase=True, preprocessor=None, tokenizer=None, stop_words=None, token_pattern='(?u)\b\w\w+\b', ngram_range=(1, 1), analyzer='word', n_features=1048576, binary=False, norm='l2', alternate_sign=True, dtype=<class 'numpy.float64'>*)

Convert a collection of text documents to a matrix of token occurrences

It turns a collection of text documents into a scipy.sparse matrix holding token occurrence counts (or binary occurrence information), possibly normalized as token frequencies if norm='l1' or projected on the euclidean unit sphere if norm='l2'.

This text vectorizer implementation uses the hashing trick to find the token string name to feature integer index mapping.

This strategy has several advantages:

- it is very low memory scalable to large datasets as there is no need to store a vocabulary dictionary in memory
- it is fast to pickle and un-pickle as it holds no state besides the constructor parameters
- it can be used in a streaming (partial fit) or parallel pipeline as there is no state computed during fit.

There are also a couple of cons (vs using a CountVectorizer with an in-memory vocabulary):

- there is no way to compute the inverse transform (from feature indices to string feature names) which can be a problem when trying to introspect which features are most important to a model.
- there can be collisions: distinct tokens can be mapped to the same feature index. However in practice this is rarely an issue if n_features is large enough (e.g. 2 ** 18 for text classification problems).
- no IDF weighting as this would render the transformer stateful.

The hash function employed is the signed 32-bit version of Murmurhash3.

Read more in the *User Guide*.

Parameters

input [string {'filename', 'file', 'content'}] If 'filename', the sequence passed as an argument to fit is expected to be a list of filenames that need reading to fetch the raw content to analyze.

If 'file', the sequence items must have a 'read' method (file-like object) that is called to fetch the bytes in memory.

Otherwise the input is expected to be a sequence of items that can be of type string or byte.

encoding [string, default='utf-8'] If bytes or files are given to analyze, this encoding is used to decode.

decode_error [{'strict', 'ignore', 'replace'}] Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given `encoding`. By default, it is 'strict', meaning that a UnicodeDecodeError will be raised. Other values are 'ignore' and 'replace'.

strip_accents [{'ascii', 'unicode', None}] Remove accents and perform other character normalization during the preprocessing step. 'ascii' is a fast method that only works on characters that have an direct ASCII mapping. 'unicode' is a slightly slower method that works on any characters. None (default) does nothing.

Both 'ascii' and 'unicode' use NFKD normalization from `unicodedata.normalize`.

lowercase [boolean, default=True] Convert all characters to lowercase before tokenizing.

preprocessor [callable or None (default)] Override the preprocessing (string transformation) stage while preserving the tokenizing and n-grams generation steps. Only applies if `analyzer is not callable`.

tokenizer [callable or None (default)] Override the string tokenization step while preserving the preprocessing and n-grams generation steps. Only applies if `analyzer == 'word'`.

stop_words [string {'english'}, list, or None (default)] If 'english', a built-in stop word list for English is used. There are several known issues with 'english' and you should consider an alternative (see *Using stop words*).

If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens. Only applies if `analyzer == 'word'`.

token_pattern [string] Regular expression denoting what constitutes a "token", only used if `analyzer == 'word'`. The default regexp selects tokens of 2 or more alphanumeric characters (punctuation is completely ignored and always treated as a token separator).

ngram_range [tuple (min_n, max_n), default=(1, 1)] The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used. For example an `ngram_range` of `(1, 1)` means only unigrams, `(1, 2)` means unigrams and bigrams, and `(2, 2)` means only bigrams. Only applies if `analyzer is not callable`.

analyzer [string, {'word', 'char', 'char_wb'} or callable] Whether the feature should be made of word or character n-grams. Option 'char_wb' creates character n-grams only from text inside word boundaries; n-grams at the edges of words are padded with space.

If a callable is passed it is used to extract the sequence of features out of the raw, unprocessed input.

Changed in version 0.21.

Since v0.21, if `input` is `filename` or `file`, the data is first read from the file and then passed to the given callable analyzer.

n_features [integer, default=(2 ** 20)] The number of features (columns) in the output matrices. Small numbers of features are likely to cause hash collisions, but large numbers will cause larger coefficient dimensions in linear learners.

binary [boolean, default=False.] If True, all non zero counts are set to 1. This is useful for discrete probabilistic models that model binary events rather than integer counts.

norm ['l1', 'l2' or None, optional] Norm used to normalize term vectors. None for no normalization.

alternate_sign [boolean, optional, default True] When True, an alternating sign is added to the features as to approximately conserve the inner product in the hashed space even for small n_features. This approach is similar to sparse random projection.

New in version 0.19.

dtype [type, optional] Type of the matrix returned by fit_transform() or transform().

See also:

CountVectorizer, ***TfidfVectorizer***

Examples

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> corpus = [
...     'This is the first document.',
...     'This document is the second document.',
...     'And this is the third one.',
...     'Is this the first document?',
... ]
>>> vectorizer = HashingVectorizer(n_features=2**4)
>>> X = vectorizer.fit_transform(corpus)
>>> print(X.shape)
(4, 16)
```

Methods

build_analyzer(self)	Return a callable that handles preprocessing, tokenization and n-grams generation.
build_preprocessor(self)	Return a function to preprocess the text before tokenization.
build_tokenizer(self)	Return a function that splits a string into a sequence of tokens.
decode(self, doc)	Decode the input into a string of unicode symbols.
fit(self, X[, y])	Does nothing: this transformer is stateless.
fit_transform(self, X[, y])	Transform a sequence of documents to a document-term matrix.
get_params(self[, deep])	Get parameters for this estimator.
get_stop_words(self)	Build or fetch the effective stop words list.
partial_fit(self, X[, y])	Does nothing: this transformer is stateless.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform a sequence of documents to a document-term matrix.

__init__(*self*, *input='content'*, *encoding='utf-8'*, *decode_error='strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *stop_words=None*, *token_pattern='(?u)\b\w\w+\b'*, *ngram_range=(1, 1)*, *analyzer='word'*, *n_features=1048576*, *binary=False*, *norm='l2'*, *alternate_sign=True*, *dtype=<class 'numpy.float64'>*)

Initialize self. See help(type(self)) for accurate signature.

build_analyzer(*self*)

Return a callable that handles preprocessing, tokenization and n-grams generation.

Returns

analyzer: callable A function to handle preprocessing, tokenization and n-grams generation.

build_preprocessor(*self*)
Return a function to preprocess the text before tokenization.

Returns

preprocessor: callable A function to preprocess the text before tokenization.

build_tokenizer(*self*)
Return a function that splits a string into a sequence of tokens.

Returns

tokenizer: callable A function to split a string into a sequence of tokens.

decode(*self*, *doc*)
Decode the input into a string of unicode symbols.

The decoding strategy depends on the vectorizer parameters.

Parameters

doc [str] The string to decode.

Returns

doc: str A string of unicode symbols.

fit(*self*, *X*, *y=None*)
Does nothing: this transformer is stateless.

Parameters

X [array-like, shape [n_samples, n_features]] Training data.

fit_transform(*self*, *X*, *y=None*)
Transform a sequence of documents to a document-term matrix.

Parameters

X [iterable over raw text documents, length = n_samples] Samples. Each sample must be a text document (either bytes or unicode strings, file name or file object depending on the constructor argument) which will be tokenized and hashed.

y [any] Ignored. This parameter exists only for compatibility with sklearn.pipeline.Pipeline.

Returns

X [sparse matrix of shape (n_samples, n_features)] Document-term matrix.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_stop_words(*self*)
Build or fetch the effective stop words list.

Returns

stop_words: list or None A list of stop words.

partial_fit(*self*, *X*, *y=None*)
Does nothing: this transformer is stateless.

This method is just there to mark the fact that this transformer can work in a streaming setup.

Parameters

X [array-like, shape [n_samples, n_features]] Training data.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Transform a sequence of documents to a document-term matrix.

Parameters

X [iterable over raw text documents, length = n_samples] Samples. Each sample must be a text document (either bytes or unicode strings, file name or file object depending on the constructor argument) which will be tokenized and hashed.

Returns

X [sparse matrix of shape (n_samples, n_features)] Document-term matrix.

Examples using `sklearn.feature_extraction.text.HashingVectorizer`

- *Out-of-core classification of text documents*
- *Clustering text documents using k-means*
- *Classification of text documents using sparse features*

`sklearn.feature_extraction.text.`TfidfTransformer

class sklearn.feature_extraction.text.**TfidfTransformer**(*norm='l2'*, *use_idf=True*, *smooth_idf=True*, *sublinear_tf=False*)
Transform a count matrix to a normalized tf or tf-idf representation

Tf means term-frequency while tf-idf means term-frequency times inverse document-frequency. This is a common term weighting scheme in information retrieval, that has also found good use in document classification.

The goal of using tf-idf instead of the raw frequencies of occurrence of a token in a given document is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than features that occur in a small fraction of the training corpus.

The formula that is used to compute the tf-idf for a term t of a document d in a document set is tf-idf(t, d) = tf(t, d) * idf(t), and the idf is computed as idf(t) = log [n / df(t)] + 1 (if `smooth_idf=False`), where n is the total number of documents in the document set and df(t) is the document frequency of t; the document frequency is the number of documents in the document set that contain the term t. The effect of adding "1" to the idf in the equation above is that terms with zero idf, i.e., terms that occur in all documents in a training set, will not be entirely ignored. (Note that the idf formula above differs from the standard textbook notation that defines the idf as idf(t) = log [n / (df(t) + 1)]).

If `smooth_idf=True` (the default), the constant "1" is added to the numerator and denominator of the idf as if an extra document was seen containing every term in the collection exactly once, which prevents zero divisions: idf(d, t) = log [(1 + n) / (1 + df(d, t))] + 1.

Furthermore, the formulas used to compute tf and idf depend on parameter settings that correspond to the SMART notation used in IR as follows:

Tf is "n" (natural) by default, "l" (logarithmic) when `sublinear_tf=True`. Idf is "t" when use_idf is given, "n" (none) otherwise. Normalization is "c" (cosine) when `norm='l2'`, "n" (none) when `norm=None`.

Read more in the *User Guide*.

Parameters

norm ['l1', 'l2' or None, optional (default='l2')] Each output row will have unit norm, either: * 'l2': Sum of squares of vector elements is 1. The cosine similarity between two vectors is their dot product when l2 norm has been applied. * 'l1': Sum of absolute values of vector elements is 1. See `preprocessing.normalize`

use_idf [boolean (default=True)] Enable inverse-document-frequency reweighting.

smooth_idf [boolean (default=True)] Smooth idf weights by adding one to document frequencies, as if an extra document was seen containing every term in the collection exactly once. Prevents zero divisions.

sublinear_tf [boolean (default=False)] Apply sublinear tf scaling, i.e. replace tf with 1 + log(tf).

Attributes

idf_ [array, shape (n_features)] The inverse document frequency (IDF) vector; only defined if `use_idf` is True.

References

[R1b90ac3ca370-Yates2011], [R1b90ac3ca370-MRS2008]

Examples

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
>>> corpus = ['this is the first document',
...           'this document is the second document',
...           'and this is the third one',
...           'is this the first document']
>>> vocabulary = ['this', 'document', 'first', 'is', 'second', 'the',
...               'and', 'one']
>>> pipe = Pipeline([('count', CountVectorizer(vocabulary=vocabulary)),
```

(continues on next page)

(continued from previous page)

```
...                  ('tfid', TfidfTransformer())]).fit(corpus)
>>> pipe['count'].transform(corpus).toarray()
array([[1, 1, 1, 1, 0, 1, 0, 0],
       [1, 2, 0, 1, 1, 1, 0, 0],
       [1, 0, 0, 1, 0, 1, 1, 1],
       [1, 1, 1, 1, 0, 1, 0, 0]])
>>> pipe['tfid'].idf_
array([1.        , 1.22314355, 1.51082562, 1.        , 1.91629073,
       1.        , 1.91629073, 1.91629073])
>>> pipe.transform(corpus).shape
(4, 8)
```

Methods

fit(self, X[, y])	Learn the idf vector (global term weights)
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, copy])	Transform a count matrix to a tf or tf-idf representation

__init__(*self*, *norm='l2'*, *use_idf=True*, *smooth_idf=True*, *sublinear_tf=False*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Learn the idf vector (global term weights)

Parameters

X [sparse matrix, [n_samples, n_features]] a matrix of term/token counts

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*, *copy=True*)
Transform a count matrix to a tf or tf-idf representation

Parameters

X [sparse matrix, [n_samples, n_features]] a matrix of term/token counts

copy [boolean, default True] Whether to copy X and operate on the copy or perform in-place operations.

Returns

vectors [sparse matrix, [n_samples, n_features]]

Examples using `sklearn.feature_extraction.text.TfidfTransformer`

- *Sample pipeline for text feature extraction and evaluation*

`sklearn.feature_extraction.text.TfidfVectorizer`

class sklearn.feature_extraction.text.**TfidfVectorizer**(*input='content'*, *encoding='utf-8'*, *decode_error='strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *analyzer='word'*, *stop_words=None*, *token_pattern='(?u)\b\w\w+\b'*, *ngram_range=(1, 1)*, *max_df=1.0*, *min_df=1*, *max_features=None*, *vocabulary=None*, *binary=False*, *dtype=<class 'numpy.float64'>*, *norm='l2'*, *use_idf=True*, *smooth_idf=True*, *sublinear_tf=False*)
Convert a collection of raw documents to a matrix of TF-IDF features.

Equivalent to `CountVectorizer` followed by `TfidfTransformer`.

Read more in the *User Guide*.

Parameters

input [str {'filename', 'file', 'content'}] If 'filename', the sequence passed as an argument to fit is expected to be a list of filenames that need reading to fetch the raw content to analyze.

If 'file', the sequence items must have a 'read' method (file-like object) that is called to fetch the bytes in memory.

Otherwise the input is expected to be a sequence of items that can be of type string or byte.

encoding [str, default='utf-8'] If bytes or files are given to analyze, this encoding is used to decode.

decode_error [{'strict', 'ignore', 'replace'} (default='strict')] Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given `encoding`. By default, it is 'strict', meaning that a UnicodeDecodeError will be raised. Other values are 'ignore' and 'replace'.

strip_accents [{'ascii', 'unicode', None} (default=None)] Remove accents and perform other character normalization during the preprocessing step. 'ascii' is a fast method that only works on characters that have an direct ASCII mapping. 'unicode' is a slightly slower method that works on any characters. None (default) does nothing.

Both 'ascii' and 'unicode' use NFKD normalization from `unicodedata.normalize`.

lowercase [bool (default=True)] Convert all characters to lowercase before tokenizing.

preprocessor [callable or None (default=None)] Override the preprocessing (string transformation) stage while preserving the tokenizing and n-grams generation steps. Only applies if `analyzer is not callable`.

tokenizer [callable or None (default=None)] Override the string tokenization step while preserving the preprocessing and n-grams generation steps. Only applies if `analyzer == 'word'`.

analyzer [str, {'word', 'char', 'char_wb'} or callable] Whether the feature should be made of word or character n-grams. Option 'char_wb' creates character n-grams only from text inside word boundaries; n-grams at the edges of words are padded with space.

If a callable is passed it is used to extract the sequence of features out of the raw, unprocessed input.

Changed in version 0.21.

Since v0.21, if `input` is `filename` or `file`, the data is first read from the file and then passed to the given callable analyzer.

stop_words [str {'english'}, list, or None (default=None)] If a string, it is passed to _check_stop_list and the appropriate stop list is returned. 'english' is currently the only supported string value. There are several known issues with 'english' and you should consider an alternative (see *Using stop words*).

If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens. Only applies if `analyzer == 'word'`.

If None, no stop words will be used. max_df can be set to a value in the range [0.7, 1.0) to automatically detect and filter stop words based on intra corpus document frequency of terms.

token_pattern [str] Regular expression denoting what constitutes a "token", only used if `analyzer == 'word'`. The default regexp selects tokens of 2 or more alphanumeric characters (punctuation is completely ignored and always treated as a token separator).

ngram_range [tuple (min_n, max_n), default=(1, 1)] The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used. For example an `ngram_range` of `(1, 1)` means only unigrams, `(1, 2)` means unigrams and bigrams, and `(2, 2)` means only bigrams. Only applies if `analyzer is not callable`.

max_df [float in range [0.0, 1.0] or int (default=1.0)] When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold (corpus-specific stop words). If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

min_df [float in range [0.0, 1.0] or int (default=1)] When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold. This value is also called cut-off in the literature. If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

max_features [int or None (default=None)] If not None, build a vocabulary that only consider the top max_features ordered by term frequency across the corpus.

This parameter is ignored if vocabulary is not None.

vocabulary [Mapping or iterable, optional (default=None)] Either a Mapping (e.g., a dict) where keys are terms and values are indices in the feature matrix, or an iterable over terms. If not given, a vocabulary is determined from the input documents.

binary [bool (default=False)] If True, all non-zero term counts are set to 1. This does not mean outputs will have only 0/1 values, only that the tf term in tf-idf is binary. (Set idf and normalization to False to get 0/1 outputs).

dtype [type, optional (default=float64)] Type of the matrix returned by fit_transform() or transform().

norm ['l1', 'l2' or None, optional (default='l2')] Each output row will have unit norm, either: * 'l2': Sum of squares of vector elements is 1. The cosine similarity between two vectors is their dot product when l2 norm has been applied. * 'l1': Sum of absolute values of vector elements is 1. See `preprocessing.normalize`.

use_idf [bool (default=True)] Enable inverse-document-frequency reweighting.

smooth_idf [bool (default=True)] Smooth idf weights by adding one to document frequencies, as if an extra document was seen containing every term in the collection exactly once. Prevents zero divisions.

sublinear_tf [bool (default=False)] Apply sublinear tf scaling, i.e. replace tf with 1 + log(tf).

Attributes

vocabulary_ [dict] A mapping of terms to feature indices.

fixed_vocabulary_: bool True if a fixed vocabulary of term to indices mapping is provided by the user

idf_ [array, shape (n_features)] The inverse document frequency (IDF) vector; only defined if `use_idf` is True.

stop_words_ [set] Terms that were ignored because they either:

- occurred in too many documents (`max_df`)
- occurred in too few documents (`min_df`)
- were cut off by feature selection (`max_features`).

This is only available if no vocabulary was given.

See also:

CountVectorizer Transforms text into a sparse matrix of n-gram counts.

TfidfTransformer Performs the TF-IDF transformation from a provided matrix of counts.

Notes

The `stop_words_` attribute can get large and increase the model size when pickling. This attribute is provided only for introspection and can be safely removed using delattr or set to None before pickling.

Examples

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> corpus = [
...     'This is the first document.',
...     'This document is the second document.',
...     'And this is the third one.',
...     'Is this the first document?',
... ]
>>> vectorizer = TfidfVectorizer()
>>> X = vectorizer.fit_transform(corpus)
>>> print(vectorizer.get_feature_names())
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
>>> print(X.shape)
(4, 9)
```

Methods

build_analyzer(self)	Return a callable that handles preprocessing, tokenization and n-grams generation.
build_preprocessor(self)	Return a function to preprocess the text before tokenization.
build_tokenizer(self)	Return a function that splits a string into a sequence of tokens.
decode(self, doc)	Decode the input into a string of unicode symbols.
fit(self, raw_documents[, y])	Learn vocabulary and idf from training set.
fit_transform(self, raw_documents[, y])	Learn vocabulary and idf, return term-document matrix.
get_feature_names(self)	Array mapping from feature integer indices to feature name.
get_params(self[, deep])	Get parameters for this estimator.
get_stop_words(self)	Build or fetch the effective stop words list.
inverse_transform(self, X)	Return terms per document with nonzero entries in X.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, raw_documents[, copy])	Transform documents to document-term matrix.

__init__(*self*, *input='content'*, *encoding='utf-8'*, *decode_error='strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *analyzer='word'*, *stop_words=None*, *token_pattern='(?u)\b\w\w+\b'*, *ngram_range=(1, 1)*, *max_df=1.0*, *min_df=1*, *max_features=None*, *vocabulary=None*, *binary=False*, *dtype=<class 'numpy.float64'>*, *norm='l2'*, *use_idf=True*, *smooth_idf=True*, *sublinear_tf=False*)
Initialize self. See help(type(self)) for accurate signature.

build_analyzer(*self*)
Return a callable that handles preprocessing, tokenization and n-grams generation.

Returns

analyzer: callable A function to handle preprocessing, tokenization and n-grams generation.

build_preprocessor(*self*)
Return a function to preprocess the text before tokenization.

Returns

preprocessor: callable A function to preprocess the text before tokenization.

build_tokenizer(*self*)
Return a function that splits a string into a sequence of tokens.

Returns

tokenizer: callable A function to split a string into a sequence of tokens.

decode(*self*, *doc*)
Decode the input into a string of unicode symbols.

The decoding strategy depends on the vectorizer parameters.

Parameters

doc [str] The string to decode.

Returns

doc: str A string of unicode symbols.

fit(*self*, *raw_documents*, *y=None*)
Learn vocabulary and idf from training set.

Parameters

raw_documents [iterable] An iterable which yields either str, unicode or file objects.

y [None] This parameter is not needed to compute tfidf.

Returns

self [object] Fitted vectorizer.

fit_transform(*self*, *raw_documents*, *y=None*)
Learn vocabulary and idf, return term-document matrix.

This is equivalent to fit followed by transform, but more efficiently implemented.

Parameters

raw_documents [iterable] An iterable which yields either str, unicode or file objects.

y [None] This parameter is ignored.

Returns

X [sparse matrix, [n_samples, n_features]] Tf-idf-weighted document-term matrix.

get_feature_names(*self*)
Array mapping from feature integer indices to feature name.

Returns

feature_names [list] A list of feature names.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_stop_words(*self*)
Build or fetch the effective stop words list.

Returns

stop_words: list or None A list of stop words.

inverse_transform(*self*, *X*)
Return terms per document with nonzero entries in X.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Document-term matrix.

Returns

X_inv [list of arrays, len = n_samples] List of arrays of terms.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *raw_documents*, *copy='deprecated'*)
Transform documents to document-term matrix.

Uses the vocabulary and document frequencies (df) learned by fit (or fit_transform).

Parameters

raw_documents [iterable] An iterable which yields either str, unicode or file objects.

copy [bool, default True] Whether to copy X and operate on the copy or perform in-place operations.

Deprecated since version 0.22: The `copy` parameter is unused and was deprecated in version 0.22 and will be removed in 0.24. This parameter will be ignored.

Returns

X [sparse matrix, [n_samples, n_features]] Tf-idf-weighted document-term matrix.

Examples using `sklearn.feature_extraction.text.TfidfVectorizer`

- *Biclustering documents with the Spectral Co-clustering algorithm*
- *Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation*
- *Column Transformer with Heterogeneous Data Sources*
- *Clustering text documents using k-means*
- *Classification of text documents using sparse features*

7.15 `sklearn.feature_selection`: Feature Selection

The *`sklearn.feature_selection`* module implements feature selection algorithms. It currently includes univariate filter selection methods and the recursive feature elimination algorithm.

User guide: See the *Feature selection* section for further details.

`feature_selection.GenericUnivariateSelect`([...])	Univariate feature selector with configurable strategy.
`feature_selection.SelectPercentile`([...])	Select features according to a percentile of the highest scores.
`feature_selection.SelectKBest`([score_func, k])	Select features according to the k highest scores.
`feature_selection.SelectFpr`([score_func, alpha])	Filter: Select the pvalues below alpha based on a FPR test.
`feature_selection.SelectFdr`([score_func, alpha])	Filter: Select the p-values for an estimated false discovery rate
`feature_selection.SelectFromModel`(estimator)	Meta-transformer for selecting features based on importance weights.
`feature_selection.SelectFwe`([score_func, alpha])	Filter: Select the p-values corresponding to Family-wise error rate
`feature_selection.RFE`(estimator[, ...])	Feature ranking with recursive feature elimination.
`feature_selection.RFECV`(estimator[, step, ...])	Feature ranking with recursive feature elimination and cross-validated selection of the best number of features.
`feature_selection.VarianceThreshold`([threshold])	Feature selector that removes all low-variance features.

7.15.1 `sklearn.feature_selection`.GenericUnivariateSelect

class sklearn.feature_selection.**GenericUnivariateSelect**(*score_func=<function f_classif>*, *mode='percentile'*, *param=1e-05*)

Univariate feature selector with configurable strategy.

Read more in the *User Guide*.

Parameters

score_func [callable] Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues). For modes 'percentile' or 'kbest' it can return a single array scores.

mode [{'percentile', 'k_best', 'fpr', 'fdr', 'fwe'}] Feature selection mode.

param [float or int depending on the feature selection mode] Parameter of the corresponding mode.

Attributes

scores_ [array-like of shape (n_features,)] Scores of features.

pvalues_ [array-like of shape (n_features,)] p-values of feature scores, None if `score_func` returned scores only.

See also:

f_classif ANOVA F-value between label/feature for classification tasks.

mutual_info_classif Mutual information for a discrete target.

chi2 Chi-squared stats of non-negative features for classification tasks.

f_regression F-value between label/feature for regression tasks.

mutual_info_regression Mutual information for a continuous target.

SelectPercentile Select features based on percentile of the highest scores.

SelectKBest Select features based on the k highest scores.

SelectFpr Select features based on a false positive rate test.

SelectFdr Select features based on an estimated false discovery rate.

SelectFwe Select features based on family-wise error rate.

Examples

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.feature_selection import GenericUnivariateSelect, chi2
>>> X, y = load_breast_cancer(return_X_y=True)
>>> X.shape
(569, 30)
>>> transformer = GenericUnivariateSelect(chi2, 'k_best', param=20)
>>> X_new = transformer.fit_transform(X, y)
>>> X_new.shape
(569, 20)
```

Methods

fit(self, X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
set_params(self, **params)	Set the parameters of this estimator.

Continued on next page

Table 106 – continued from previous page

transform(self, X)	Reduce X to the selected features.

__init__(*self*, *score_func=<function f_classif at 0x7fdf6f144040>*, *mode='percentile'*, *param=1e-05*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)
Run score function on (X, y) and get the appropriate features.

Parameters

X [array-like of shape (n_samples, n_features)] The training input samples.

y [array-like of shape (n_samples,)] The target values (class labels in classification, real numbers in regression).

Returns

self [object]

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_support(*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

indices [boolean (default False)] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support [array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

X [array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_original_features]] `X` with columns of zeros inserted where features would have been removed by *transform*.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Reduce X to the selected features.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

7.15.2 sklearn.feature_selection.SelectPercentile

class sklearn.feature_selection.**SelectPercentile**(*score_func=<function f_classif>*, *percentile=10*)
Select features according to a percentile of the highest scores.

Read more in the *User Guide*.

Parameters

score_func [callable] Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues) or a single array with scores. Default is f_classif (see below "See also"). The default function only works with classification tasks.

percentile [int, optional, default=10] Percent of features to keep.

Attributes

scores_ [array-like of shape (n_features,)] Scores of features.

pvalues_ [array-like of shape (n_features,)] p-values of feature scores, None if `score_func` returned only scores.

See also:

f_classif ANOVA F-value between label/feature for classification tasks.

mutual_info_classif Mutual information for a discrete target.

chi2 Chi-squared stats of non-negative features for classification tasks.

f_regression F-value between label/feature for regression tasks.

mutual_info_regression Mutual information for a continuous target.

SelectKBest Select features based on the k highest scores.

SelectFpr Select features based on a false positive rate test.

SelectFdr Select features based on an estimated false discovery rate.

SelectFwe Select features based on family-wise error rate.

GenericUnivariateSelect Univariate feature selector with configurable mode.

Notes

Ties between features with equal scores will be broken in an unspecified way.

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.feature_selection import SelectPercentile, chi2
>>> X, y = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> X_new = SelectPercentile(chi2, percentile=10).fit_transform(X, y)
>>> X_new.shape
(1797, 7)
```

Methods

fit(self, X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__ (*self*, *score_func=<function f_classif at 0x7fdf6f144040>*, *percentile=10*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Run score function on (X, y) and get the appropriate features.

Parameters

X [array-like of shape (n_samples, n_features)] The training input samples.

y [array-like of shape (n_samples,)] The target values (class labels in classification, real numbers in regression).

Returns

self [object]

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_support(*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

indices [boolean (default False)] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support [array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

X [array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_original_features]] `X` with columns of zeros inserted where features would have been removed by *`transform`*.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)

Reduce X to the selected features.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

Examples using `sklearn.feature_selection.SelectPercentile`

- *Feature agglomeration vs. univariate selection*
- *SVM-Anova: SVM with univariate feature selection*

7.15.3 `sklearn.feature_selection`.SelectKBest

class sklearn.feature_selection.**SelectKBest**(*score_func=<function f_classif>*, *k=10*)

Select features according to the k highest scores.

Read more in the *User Guide*.

Parameters

score_func [callable] Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues) or a single array with scores. Default is f_classif (see below "See also"). The default function only works with classification tasks.

k [int or "all", optional, default=10] Number of top features to select. The "all" option bypasses selection, for use in a parameter search.

Attributes

scores_ [array-like of shape (n_features,)] Scores of features.

pvalues_ [array-like of shape (n_features,)] p-values of feature scores, None if `score_func` returned only scores.

See also:

f_classif ANOVA F-value between label/feature for classification tasks.

mutual_info_classif Mutual information for a discrete target.

chi2 Chi-squared stats of non-negative features for classification tasks.

f_regression F-value between label/feature for regression tasks.

mutual_info_regression Mutual information for a continuous target.

SelectPercentile Select features based on percentile of the highest scores.

SelectFpr Select features based on a false positive rate test.

SelectFdr Select features based on an estimated false discovery rate.

SelectFwe Select features based on family-wise error rate.

GenericUnivariateSelect Univariate feature selector with configurable mode.

Notes

Ties between features with equal scores will be broken in an unspecified way.

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.feature_selection import SelectKBest, chi2
>>> X, y = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> X_new = SelectKBest(chi2, k=20).fit_transform(X, y)
>>> X_new.shape
(1797, 20)
```

Methods

fit(self, X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__ (*self*, *score_func=<function f_classif at 0x7fdf6f144040>*, *k=10*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Run score function on (X, y) and get the appropriate features.

Parameters

X [array-like of shape (n_samples, n_features)] The training input samples.

y [array-like of shape (n_samples,)] The target values (class labels in classification, real numbers in regression).

Returns

self [object]

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_support(*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

indices [boolean (default False)] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support [array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

X [array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_original_features]] `X` with columns of zeros inserted where features would have been removed by *`transform`*.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Reduce X to the selected features.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

Examples using `sklearn.feature_selection.SelectKBest`

- *Pipeline Anova SVM*
- *Univariate Feature Selection*
- *Concatenating multiple feature extraction methods*
- *Selecting dimensionality reduction with Pipeline and GridSearchCV*
- *Classification of text documents using sparse features*

7.15.4 `sklearn.feature_selection.SelectFpr`

class sklearn.feature_selection.**SelectFpr**(*score_func=<function f_classif>*, *alpha=0.05*)

Filter: Select the pvalues below alpha based on a FPR test.

FPR test stands for False Positive Rate test. It controls the total amount of false detections.

Read more in the *User Guide*.

Parameters

score_func [callable] Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues). Default is f_classif (see below "See also"). The default function only works with classification tasks.

alpha [float, optional] The highest p-value for features to be kept.

Attributes

scores_ [array-like of shape (n_features,)] Scores of features.

pvalues_ [array-like of shape (n_features,)] p-values of feature scores.

See also:

f_classif ANOVA F-value between label/feature for classification tasks.

chi2 Chi-squared stats of non-negative features for classification tasks.

mutual_info_classif

f_regression F-value between label/feature for regression tasks.

mutual_info_regression Mutual information between features and the target.

SelectPercentile Select features based on percentile of the highest scores.

SelectKBest Select features based on the k highest scores.

SelectFdr Select features based on an estimated false discovery rate.

SelectFwe Select features based on family-wise error rate.

GenericUnivariateSelect Univariate feature selector with configurable mode.

Examples

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.feature_selection import SelectFpr, chi2
>>> X, y = load_breast_cancer(return_X_y=True)
>>> X.shape
(569, 30)
>>> X_new = SelectFpr(chi2, alpha=0.01).fit_transform(X, y)
>>> X_new.shape
(569, 16)
```

Methods

fit(self, X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__ (*self*, *score_func=<function f_classif at 0x7fdf6f144040>*, *alpha=0.05*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Run score function on (X, y) and get the appropriate features.

Parameters

X [array-like of shape (n_samples, n_features)] The training input samples.

y [array-like of shape (n_samples,)] The target values (class labels in classification, real numbers in regression).

Returns

self [object]

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_support(*self*, *indices=False*)

Get a mask, or integer index, of the features selected

Parameters

indices [boolean (default False)] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support [array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

inverse_transform(*self*, *X*)

Reverse the transformation operation

Parameters

X [array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_original_features]] `X` with columns of zeros inserted where features would have been removed by *`transform`*.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)

Reduce X to the selected features.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

7.15.5 sklearn.feature_selection.SelectFdr

class sklearn.feature_selection.**SelectFdr**(*score_func=<function f_classif>*, *alpha=0.05*)

Filter: Select the p-values for an estimated false discovery rate

This uses the Benjamini-Hochberg procedure. `alpha` is an upper bound on the expected false discovery rate.

Read more in the *User Guide*.

Parameters

score_func [callable] Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues). Default is f_classif (see below "See also"). The default function only works with classification tasks.

alpha [float, optional] The highest uncorrected p-value for features to keep.

Attributes

scores_ [array-like of shape (n_features,)] Scores of features.

pvalues_ [array-like of shape (n_features,)] p-values of feature scores.

See also:

f_classif ANOVA F-value between label/feature for classification tasks.

mutual_info_classif Mutual information for a discrete target.

chi2 Chi-squared stats of non-negative features for classification tasks.

f_regression F-value between label/feature for regression tasks.

mutual_info_regression Mutual information for a contnuous target.

SelectPercentile Select features based on percentile of the highest scores.

SelectKBest Select features based on the k highest scores.

SelectFpr Select features based on a false positive rate test.

SelectFwe Select features based on family-wise error rate.

GenericUnivariateSelect Univariate feature selector with configurable mode.

References

https://en.wikipedia.org/wiki/False_discovery_rate

Examples

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.feature_selection import SelectFdr, chi2
>>> X, y = load_breast_cancer(return_X_y=True)
>>> X.shape
(569, 30)
>>> X_new = SelectFdr(chi2, alpha=0.01).fit_transform(X, y)
>>> X_new.shape
(569, 16)
```

Methods

fit(self, X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__ (*self*, *score_func=<function f_classif at 0x7fdf6f144040>*, *alpha=0.05*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Run score function on (X, y) and get the appropriate features.

Parameters

X [array-like of shape (n_samples, n_features)] The training input samples.

y [array-like of shape (n_samples,)] The target values (class labels in classification, real numbers in regression).

Returns

self [object]

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_support (*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

indices [boolean (default False)] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support [array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

inverse_transform(*self*, *X*)

Reverse the transformation operation

Parameters

X [array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_original_features]] `X` with columns of zeros inserted where features would have been removed by *transform*.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)

Reduce X to the selected features.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

7.15.6 sklearn.feature_selection.SelectFromModel

class sklearn.feature_selection.**SelectFromModel**(*estimator*, *threshold=None*, *prefit=False*, *norm_order=1*, *max_features=None*)

Meta-transformer for selecting features based on importance weights.

New in version 0.17.

Parameters

estimator [object] The base estimator from which the transformer is built. This can be both a fitted (if `prefit` is set to True) or a non-fitted estimator. The estimator must have either a `feature_importances_` or `coef_` attribute after fitting.

threshold [string, float, optional default None] The threshold value to use for feature selection. Features whose importance is greater or equal are kept while the others are discarded. If "median" (resp. "mean"), then the `threshold` value is the median (resp. the mean) of the feature importances. A scaling factor (e.g., "1.25*mean") may also be used. If None and if the estimator has a parameter penalty set to l1, either explicitly or implicitly (e.g, Lasso), the threshold used is 1e-5. Otherwise, "mean" is used by default.

prefit [bool, default False] Whether a prefit model is expected to be passed into the constructor directly or not. If True, `transform` must be called directly and SelectFromModel cannot be used with `cross_val_score`, `GridSearchCV` and similar utilities that clone the estimator. Otherwise train the model using `fit` and then `transform` to do feature selection.

norm_order [non-zero int, inf, -inf, default 1] Order of the norm used to filter the vectors of coefficients below `threshold` in the case where the `coef_` attribute of the estimator is of dimension 2.

max_features [int or None, optional] The maximum number of features selected scoring above `threshold`. To disable `threshold` and only select based on `max_features`, set `threshold=-np.inf`.

New in version 0.20.

Attributes

estimator_ [an estimator] The base estimator from which the transformer is built. This is stored only when a non-fitted estimator is passed to the `SelectFromModel`, i.e when prefit is False.

threshold_ [float] The threshold value used for feature selection.

Notes

Allows NaN/Inf in the input if the underlying estimator does as well.

Examples

```
>>> from sklearn.feature_selection import SelectFromModel
>>> from sklearn.linear_model import LogisticRegression
>>> X = [[ 0.87, -1.34,  0.31 ],
...      [-2.79, -0.02, -0.85 ],
...      [-1.34, -0.48, -2.55 ],
...      [ 1.92,  1.48,  0.65 ]]
>>> y = [0, 1, 0, 1]
>>> selector = SelectFromModel(estimator=LogisticRegression()).fit(X, y)
>>> selector.estimator_.coef_
array([[-0.3252302 ,  0.83462377,  0.49750423]])
>>> selector.threshold_
0.55245...
>>> selector.get_support()
array([False,  True, False])
>>> selector.transform(X)
array([[-1.34],
       [-0.02],
       [-0.48],
       [ 1.48]])
```

Methods

fit(self, X[, y])	Fit the SelectFromModel meta-transformer.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
partial_fit(self, X[, y])	Fit the SelectFromModel meta-transformer only once.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__ (*self*, *estimator*, *threshold=None*, *prefit=False*, *norm_order=1*, *max_features=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*, ***fit_params*)
Fit the SelectFromModel meta-transformer.

Parameters

X [array-like of shape (n_samples, n_features)] The training input samples.

y [array-like, shape (n_samples,)] The target values (integers that correspond to classes in classification, real numbers in regression).

****fit_params** [Other estimator specific parameters]

Returns

self [object]

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_support (*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

indices [boolean (default False)] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support [array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

inverse_transform(*self*, *X*)

Reverse the transformation operation

Parameters

X [array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_original_features]] `X` with columns of zeros inserted where features would have been removed by *transform*.

partial_fit(*self*, *X*, *y=None*, ***fit_params*)

Fit the SelectFromModel meta-transformer only once.

Parameters

X [array-like of shape (n_samples, n_features)] The training input samples.

y [array-like, shape (n_samples,)] The target values (integers that correspond to classes in classification, real numbers in regression).

****fit_params** [Other estimator specific parameters]

Returns

self [object]

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)

Reduce X to the selected features.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

Examples using `sklearn.feature_selection.SelectFromModel`

- *Feature selection using SelectFromModel and LassoCV*
- *Classification of text documents using sparse features*

7.15.7 `sklearn.feature_selection`.SelectFwe

class sklearn.feature_selection.**SelectFwe**(*score_func=<function f_classif>*, *alpha=0.05*)

Filter: Select the p-values corresponding to Family-wise error rate

Read more in the *User Guide*.

Parameters

score_func [callable] Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues). Default is f_classif (see below "See also"). The default function only works with classification tasks.

alpha [float, optional] The highest uncorrected p-value for features to keep.

Attributes

scores_ [array-like of shape (n_features,)] Scores of features.

pvalues_ [array-like of shape (n_features,)] p-values of feature scores.

See also:

f_classif ANOVA F-value between label/feature for classification tasks.

chi2 Chi-squared stats of non-negative features for classification tasks.

f_regression F-value between label/feature for regression tasks.

SelectPercentile Select features based on percentile of the highest scores.

SelectKBest Select features based on the k highest scores.

SelectFpr Select features based on a false positive rate test.

SelectFdr Select features based on an estimated false discovery rate.

GenericUnivariateSelect Univariate feature selector with configurable mode.

Examples

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.feature_selection import SelectFwe, chi2
>>> X, y = load_breast_cancer(return_X_y=True)
>>> X.shape
(569, 30)
>>> X_new = SelectFwe(chi2, alpha=0.01).fit_transform(X, y)
>>> X_new.shape
(569, 15)
```

Methods

fit(self, X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__ (*self*, *score_func=<function f_classif at 0x7fdf6f144040>*, *alpha=0.05*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Run score function on (X, y) and get the appropriate features.

Parameters

X [array-like of shape (n_samples, n_features)] The training input samples.

y [array-like of shape (n_samples,)] The target values (class labels in classification, real numbers in regression).

Returns

self [object]

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_support (*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

indices [boolean (default False)] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support [array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

X [array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_original_features]] `X` with columns of zeros inserted where features would have been removed by *transform*.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Reduce X to the selected features.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

7.15.8 sklearn.feature_selection.RFE

class sklearn.feature_selection.**RFE**(*estimator*, *n_features_to_select=None*, *step=1*, *verbose=0*)
Feature ranking with recursive feature elimination.

Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model), the goal of recursive feature elimination (RFE) is to select features by recursively considering smaller and smaller sets of features. First, the estimator is trained on the initial set of features and the importance of each feature is obtained either through a `coef_` attribute or through a `feature_importances_` attribute. Then, the least important features are pruned from current set of features. That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.

Read more in the *User Guide*.

Parameters

estimator [object] A supervised learning estimator with a `fit` method that provides information about feature importance either through a `coef_` attribute or through a `feature_importances_` attribute.

n_features_to_select [int or None (default=None)] The number of features to select. If `None`, half of the features are selected.

step [int or float, optional (default=1)] If greater than or equal to 1, then `step` corresponds to the (integer) number of features to remove at each iteration. If within (0.0, 1.0), then `step` corresponds to the percentage (rounded down) of features to remove at each iteration.

verbose [int, (default=0)] Controls verbosity of output.

Attributes

n_features_ [int] The number of selected features.

support_ [array of shape [n_features]] The mask of selected features.

ranking_ [array of shape [n_features]] The feature ranking, such that `ranking_[i]` corresponds to the ranking position of the i-th feature. Selected (i.e., estimated best) features are assigned rank 1.

estimator_ [object] The external estimator fit on the reduced dataset.

See also:

RFECV Recursive feature elimination with built-in cross-validated selection of the best number of features

Notes

Allows NaN/Inf in the input if the underlying estimator does as well.

References

[Re310f679c81e-1]

Examples

The following example shows how to retrieve the 5 most informative features in the Friedman #1 dataset.

```
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.feature_selection import RFE
>>> from sklearn.svm import SVR
>>> X, y = make_friedman1(n_samples=50, n_features=10, random_state=0)
>>> estimator = SVR(kernel="linear")
>>> selector = RFE(estimator, 5, step=1)
>>> selector = selector.fit(X, y)
>>> selector.support_
array([ True,  True,  True,  True,  True, False, False, False, False,
       False])
>>> selector.ranking_
array([1, 1, 1, 1, 1, 6, 4, 3, 2, 5])
```

Methods

decision_function(self, X)	Compute the decision function of X.
fit(self, X, y)	Fit the RFE model and then the underlying estimator on the selected
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
predict(self, X)	Reduce X to the selected features and then predict using the
predict_log_proba(self, X)	Predict class log-probabilities for X.
predict_proba(self, X)	Predict class probabilities for X.
score(self, X, y)	Reduce X to the selected features and then return the score of the
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__(*self*, *estimator*, *n_features_to_select=None*, *step=1*, *verbose=0*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Compute the decision function of X.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

score [array, shape = [n_samples, n_classes] or [n_samples]] The decision function of the input samples. The order of the classes corresponds to that in the attribute *classes_*. Regression and binary classification produce an array of shape [n_samples].

fit(*self*, *X*, *y*)

Fit the RFE model and then the underlying estimator on the selected features.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples.

y [array-like of shape (n_samples,)] The target values.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_support(*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

indices [boolean (default False)] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support [array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

X [array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_original_features]] `X` with columns of zeros inserted where features would have been removed by *transform*.

predict(*self*, *X*)

Reduce X to the selected features and then predict using the underlying estimator.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

Returns

y [array of shape [n_samples]] The predicted target values.

predict_log_proba(*self*, *X*)
Predict class log-probabilities for X.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

Returns

p [array of shape (n_samples, n_classes)] The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

predict_proba(*self*, *X*)
Predict class probabilities for X.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

p [array of shape (n_samples, n_classes)] The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

score(*self*, *X*, *y*)

Reduce X to the selected features and then return the score of the underlying estimator.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

y [array of shape [n_samples]] The target values.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)

Reduce X to the selected features.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

Examples using `sklearn.feature_selection.RFE`

- *Recursive feature elimination*

7.15.9 `sklearn.feature_selection`.RFECV

class sklearn.feature_selection.**RFECV**(*estimator*, *step=1*, *min_features_to_select=1*, *cv=None*, *scoring=None*, *verbose=0*, *n_jobs=None*)

Feature ranking with recursive feature elimination and cross-validated selection of the best number of features.

See glossary entry for *cross-validation estimator*.

Read more in the *User Guide*.

Parameters

estimator [object] A supervised learning estimator with a `fit` method that provides information about feature importance either through a `coef_` attribute or through a `feature_importances_` attribute.

step [int or float, optional (default=1)] If greater than or equal to 1, then `step` corresponds to the (integer) number of features to remove at each iteration. If within (0.0, 1.0), then `step` corresponds to the percentage (rounded down) of features to remove at each iteration. Note that the last iteration may remove fewer than `step` features in order to reach `min_features_to_select`.

min_features_to_select [int, (default=1)] The minimum number of features to be selected. This number of features will always be scored, even if the difference between the original feature count and `min_features_to_select` isn't divisible by `step`.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if `y` is binary or multiclass, *sklearn.model_selection.StratifiedKFold* is used. If the estimator is a classifier or if `y` is neither binary nor multiclass, *sklearn.model_selection.KFold* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value of None changed from 3-fold to 5-fold.

scoring [string, callable or None, optional, (default=None)] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`.

verbose [int, (default=0)] Controls verbosity of output.

n_jobs [int or None, optional (default=None)] Number of cores to run in parallel while fitting across folds. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

n_features_ [int] The number of selected features with cross-validation.

support_ [array of shape [n_features]] The mask of selected features.

ranking_ [array of shape [n_features]] The feature ranking, such that `ranking_[i]` corresponds to the ranking position of the i-th feature. Selected (i.e., estimated best) features are assigned rank 1.

grid_scores_ [array of shape [n_subsets_of_features]] The cross-validation scores such that `grid_scores_[i]` corresponds to the CV score of the i-th subset of features.

estimator_ [object] The external estimator fit on the reduced dataset.

See also:

RFE Recursive feature elimination

Notes

The size of `grid_scores_` is equal to `ceil((n_features - min_features_to_select) / step) + 1`, where step is the number of features removed at each iteration.

Allows NaN/Inf in the input if the underlying estimator does as well.

References

[R6f4d61ceb411-1]

Examples

The following example shows how to retrieve the a-priori not known 5 informative features in the Friedman #1 dataset.

```
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.feature_selection import RFECV
>>> from sklearn.svm import SVR
>>> X, y = make_friedman1(n_samples=50, n_features=10, random_state=0)
>>> estimator = SVR(kernel="linear")
>>> selector = RFECV(estimator, step=1, cv=5)
>>> selector = selector.fit(X, y)
>>> selector.support_
array([ True,  True,  True,  True,  True, False, False, False, False,
       False])
>>> selector.ranking_
array([1, 1, 1, 1, 1, 6, 4, 3, 2, 5])
```

Methods

decision_function(self, X)	Compute the decision function of X.
fit(self, X, y[, groups])	Fit the RFE model and automatically tune the number of selected
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
predict(self, X)	Reduce X to the selected features and then predict using the
predict_log_proba(self, X)	Predict class log-probabilities for X.
predict_proba(self, X)	Predict class probabilities for X.
score(self, X, y)	Reduce X to the selected features and then return the score of the
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__(*self*, *estimator*, *step=1*, *min_features_to_select=1*, *cv=None*, *scoring=None*, *verbose=0*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)

Compute the decision function of X.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

score [array, shape = [n_samples, n_classes] or [n_samples]] The decision function of the input samples. The order of the classes corresponds to that in the attribute *classes_*. Regression and binary classification produce an array of shape [n_samples].

fit (*self*, *X*, *y*, *groups=None*)

Fit the RFE model and automatically tune the number of selected features.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vector, where `n_samples` is the number of samples and `n_features` is the total number of features.

y [array-like of shape (n_samples,)] Target values (integers for classification, real numbers for regression).

groups [array-like of shape (n_samples,) or None] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., *GroupKFold*).

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_support (*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

indices [boolean (default False)] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support [array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

inverse_transform(*self*, *X*)

Reverse the transformation operation

Parameters

X [array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_original_features]] `X` with columns of zeros inserted where features would have been removed by *transform*.

predict(*self*, *X*)

Reduce X to the selected features and then predict using the underlying estimator.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

Returns

y [array of shape [n_samples]] The predicted target values.

predict_log_proba(*self*, *X*)

Predict class log-probabilities for X.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

Returns

p [array of shape (n_samples, n_classes)] The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

predict_proba(*self*, *X*)

Predict class probabilities for X.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

p [array of shape (n_samples, n_classes)] The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

score(*self*, *X*, *y*)

Reduce X to the selected features and then return the score of the underlying estimator.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

y [array of shape [n_samples]] The target values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Reduce X to the selected features.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

Examples using `sklearn.feature_selection.RFECV`

- *Recursive feature elimination with cross-validation*

7.15.10 `sklearn.feature_selection`.VarianceThreshold

class sklearn.feature_selection.**VarianceThreshold**(*threshold=0.0*)
Feature selector that removes all low-variance features.

This feature selection algorithm looks only at the features (X), not the desired outputs (y), and can thus be used for unsupervised learning.

Read more in the *User Guide*.

Parameters

threshold [float, optional] Features with a training-set variance lower than this threshold will be removed. The default is to keep all features with non-zero variance, i.e. remove the features that have the same value in all samples.

Attributes

variances_ [array, shape (n_features,)] Variances of individual features.

Notes

Allows NaN in the input.

Examples

The following dataset has integer features, two of which are the same in every sample. These are removed with the default setting for threshold:

```
>>> X = [[0, 2, 0, 3], [0, 1, 4, 3], [0, 1, 1, 3]]
>>> selector = VarianceThreshold()
>>> selector.fit_transform(X)
array([[2, 0],
       [1, 4],
       [1, 1]])
```

Methods

fit(self, X[, y])	Learn empirical variances from X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__(*self*, *threshold=0.0*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Learn empirical variances from X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Sample vectors from which to compute variances.

y [any] Ignored. This parameter exists only for compatibility with sklearn.pipeline.Pipeline.

Returns

self

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_support(*self*, *indices=False*)

Get a mask, or integer index, of the features selected

Parameters

indices [boolean (default False)] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support [array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

inverse_transform(*self*, *X*)

Reverse the transformation operation

Parameters

X [array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_original_features]] `X` with columns of zeros inserted where features would have been removed by *transform*.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)

Reduce X to the selected features.

Parameters

X [array of shape [n_samples, n_features]] The input samples.

Returns

X_r [array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

feature_selection.chi2(X, y)	Compute chi-squared stats between each non-negative feature and class.

Continued on next page

Table 116 – continued from previous page

feature_selection.f_classif(X, y)	Compute the ANOVA F-value for the provided sample.
feature_selection.f_regression(X, y[, center])	Univariate linear regression tests.
feature_selection.mutual_info_classif(X, y)	Estimate mutual information for a discrete target variable.
feature_selection.mutual_info_regression(X, y)	Estimate mutual information for a continuous target variable.

7.15.11 sklearn.feature_selection.chi2

sklearn.feature_selection.**chi2**(*X*, *y*)

Compute chi-squared stats between each non-negative feature and class.

This score can be used to select the n_features features with the highest values for the test chi-squared statistic from X, which must contain only non-negative features such as booleans or frequencies (e.g., term counts in document classification), relative to the classes.

Recall that the chi-square test measures dependence between stochastic variables, so using this function "weeds out" the features that are the most likely to be independent of class and therefore irrelevant for classification.

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Sample vectors.

y [array-like of shape (n_samples,)] Target vector (class labels).

Returns

chi2 [array, shape = (n_features,)] chi2 statistics of each feature.

pval [array, shape = (n_features,)] p-values of each feature.

See also:

f_classif ANOVA F-value between label/feature for classification tasks.

f_regression F-value between label/feature for regression tasks.

Notes

Complexity of this algorithm is O(n_classes * n_features).

Examples using sklearn.feature_selection.chi2

- *Selecting dimensionality reduction with Pipeline and GridSearchCV*
- *SVM-Anova: SVM with univariate feature selection*
- *Classification of text documents using sparse features*

7.15.12 sklearn.feature_selection.f_classif

sklearn.feature_selection.**f_classif**(*X*, *y*)

Compute the ANOVA F-value for the provided sample.

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix} shape = [n_samples, n_features]] The set of regressors that will be tested sequentially.

y [array of shape(n_samples)] The data matrix.

Returns

F [array, shape = [n_features,]] The set of F values.

pval [array, shape = [n_features,]] The set of p-values.

See also:

chi2 Chi-squared stats of non-negative features for classification tasks.

f_regression F-value between label/feature for regression tasks.

Examples using sklearn.feature_selection.f_classif

- *Univariate Feature Selection*

7.15.13 sklearn.feature_selection.f_regression

sklearn.feature_selection.**f_regression**(*X*, *y*, *center=True*)

Univariate linear regression tests.

Linear model for testing the individual effect of each of many regressors. This is a scoring function to be used in a feature selection procedure, not a free standing feature selection procedure.

This is done in 2 steps:

1. The correlation between each regressor and the target is computed, that is, ((X[:, i] - mean(X[:, i])) * (y - mean_y)) / (std(X[:, i]) * std(y)).
2. It is converted to an F score then to a p-value.

For more on usage see the *User Guide*.

Parameters

X [{array-like, sparse matrix} shape = (n_samples, n_features)] The set of regressors that will be tested sequentially.

y [array of shape(n_samples).] The data matrix

center [True, bool,] If true, X and y will be centered.

Returns

F [array, shape=(n_features,)] F values of features.

pval [array, shape=(n_features,)] p-values of F-scores.

See also:

mutual_info_regression Mutual information for a continuous target.

f_classif ANOVA F-value between label/feature for classification tasks.

chi2 Chi-squared stats of non-negative features for classification tasks.

SelectKBest Select features based on the k highest scores.

SelectFpr Select features based on a false positive rate test.

SelectFdr Select features based on an estimated false discovery rate.

SelectFwe Select features based on family-wise error rate.

SelectPercentile Select features based on percentile of the highest scores.

Examples using `sklearn.feature_selection.f_regression`

- *Feature agglomeration vs. univariate selection*
- *Comparison of F-test and mutual information*
- *Pipeline Anova SVM*

7.15.14 `sklearn.feature_selection`.mutual_info_classif

`sklearn.feature_selection.`**`mutual_info_classif`**(*X*, *y*, *discrete_features='auto'*, *n_neighbors=3*, *copy=True*, *random_state=None*)

Estimate mutual information for a discrete target variable.

Mutual information (MI) [1] between two random variables is a non-negative value, which measures the dependency between the variables. It is equal to zero if and only if two random variables are independent, and higher values mean higher dependency.

The function relies on nonparametric methods based on entropy estimation from k-nearest neighbors distances as described in [2] and [3]. Both methods are based on the idea originally proposed in [4].

It can be used for univariate features selection, read more in the *User Guide*.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Feature matrix.

y [array_like, shape (n_samples,)] Target vector.

discrete_features [{'auto', bool, array_like}, default 'auto'] If bool, then determines whether to consider all features discrete or continuous. If array, then it should be either a boolean mask with shape (n_features,) or array with indices of discrete features. If 'auto', it is assigned to False for dense `X` and to True for sparse `X`.

n_neighbors [int, default 3] Number of neighbors to use for MI estimation for continuous variables, see [2] and [3]. Higher values reduce variance of the estimation, but could introduce a bias.

copy [bool, default True] Whether to make a copy of the given data. If set to False, the initial data will be overwritten.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator for adding small noise to continuous variables in order to remove repeated values. If int, random_state is the seed used by the random number

generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Returns

mi [ndarray, shape (n_features,)] Estimated mutual information between each feature and the target.

Notes

1. The term "discrete features" is used instead of naming them "categorical", because it describes the essence more accurately. For example, pixel intensities of an image are discrete features (but hardly categorical) and you will get better results if mark them as such. Also note, that treating a continuous variable as discrete and vice versa will usually give incorrect results, so be attentive about that.
2. True mutual information can't be negative. If its estimate turns out to be negative, it is replaced by zero.

References

[1], [2], [3], [4]

7.15.15 sklearn.feature_selection.mutual_info_regression

sklearn.feature_selection.**mutual_info_regression**(*X*, *y*, *discrete_features='auto'*, *n_neighbors=3*, *copy=True*, *random_state=None*)

Estimate mutual information for a continuous target variable.

Mutual information (MI) [1] between two random variables is a non-negative value, which measures the dependency between the variables. It is equal to zero if and only if two random variables are independent, and higher values mean higher dependency.

The function relies on nonparametric methods based on entropy estimation from k-nearest neighbors distances as described in [2] and [3]. Both methods are based on the idea originally proposed in [4].

It can be used for univariate features selection, read more in the *User Guide*.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Feature matrix.

y [array_like, shape (n_samples,)] Target vector.

discrete_features [{'auto', bool, array_like}, default 'auto'] If bool, then determines whether to consider all features discrete or continuous. If array, then it should be either a boolean mask with shape (n_features,) or array with indices of discrete features. If 'auto', it is assigned to False for dense `X` and to True for sparse `X`.

n_neighbors [int, default 3] Number of neighbors to use for MI estimation for continuous variables, see [2] and [3]. Higher values reduce variance of the estimation, but could introduce a bias.

copy [bool, default True] Whether to make a copy of the given data. If set to False, the initial data will be overwritten.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator for adding small noise to continuous variables in order to remove repeated values. If int, random_state is the seed used by the random number

generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Returns

mi [ndarray, shape (n_features,)] Estimated mutual information between each feature and the target.

Notes

1. The term "discrete features" is used instead of naming them "categorical", because it describes the essence more accurately. For example, pixel intensities of an image are discrete features (but hardly categorical) and you will get better results if mark them as such. Also note, that treating a continuous variable as discrete and vice versa will usually give incorrect results, so be attentive about that.
2. True mutual information can't be negative. If its estimate turns out to be negative, it is replaced by zero.

References

[1], [2], [3], [4]

Examples using `sklearn.feature_selection.mutual_info_regression`

- *Comparison of F-test and mutual information*

7.16 `sklearn.gaussian_process`: Gaussian Processes

The *sklearn.gaussian_process* module implements Gaussian Process based regression and classification.

User guide: See the *Gaussian Processes* section for further details.

gaussian_process.GaussianProcessClassifier([...])	Gaussian process classification (GPC) based on Laplace approximation.
gaussian_process.GaussianProcessRegressor([...])	Gaussian process regression (GPR).

7.16.1 `sklearn.gaussian_process`.GaussianProcessClassifier

class sklearn.gaussian_process.**GaussianProcessClassifier**(*kernel=None*, *optimizer='fmin_l_bfgs_b'*, *n_restarts_optimizer=0*, *max_iter_predict=100*, *warm_start=False*, *copy_X_train=True*, *random_state=None*, *multi_class='one_vs_rest'*, *n_jobs=None*)

Gaussian process classification (GPC) based on Laplace approximation.

The implementation is based on Algorithm 3.1, 3.2, and 5.1 of Gaussian Processes for Machine Learning (GPML) by Rasmussen and Williams.

Internally, the Laplace approximation is used for approximating the non-Gaussian posterior by a Gaussian.

Currently, the implementation is restricted to using the logistic link function. For multi-class classification, several binary one-versus rest classifiers are fitted. Note that this class thus does not implement a true multi-class Laplace approximation.

Parameters

kernel [kernel object] The kernel specifying the covariance function of the GP. If None is passed, the kernel "1.0 * RBF(1.0)" is used as default. Note that the kernel's hyperparameters are optimized during fitting.

optimizer [string or callable, optional (default: "fmin_l_bfgs_b")] Can either be one of the internally supported optimizers for optimizing the kernel's parameters, specified by a string, or an externally defined optimizer passed as a callable. If a callable is passed, it must have the signature:

```
def optimizer(obj_func, initial_theta, bounds):
    # * 'obj_func' is the objective function to be maximized, which
    #   takes the hyperparameters theta as parameter and an
    #   optional flag eval_gradient, which determines if the
    #   gradient is returned additionally to the function value
    # * 'initial_theta': the initial value for theta, which can be
    #   used by local optimizers
    # * 'bounds': the bounds on the values of theta
    ....
    # Returned are the best found hyperparameters theta and
    # the corresponding value of the target function.
    return theta_opt, func_min
```

Per default, the 'L-BFGS-B' algorithm from scipy.optimize.minimize is used. If None is passed, the kernel's parameters are kept fixed. Available internal optimizers are:

```
'fmin_l_bfgs_b'
```

n_restarts_optimizer [int, optional (default: 0)] The number of restarts of the optimizer for finding the kernel's parameters which maximize the log-marginal likelihood. The first run of the optimizer is performed from the kernel's initial parameters, the remaining ones (if any) from thetas sampled log-uniform randomly from the space of allowed theta-values. If greater than 0, all bounds must be finite. Note that n_restarts_optimizer=0 implies that one run is performed.

max_iter_predict [int, optional (default: 100)] The maximum number of iterations in Newton's method for approximating the posterior during predict. Smaller values will reduce computation time at the cost of worse results.

warm_start [bool, optional (default: False)] If warm-starts are enabled, the solution of the last Newton iteration on the Laplace approximation of the posterior mode is used as initialization for the next call of _posterior_mode(). This can speed up convergence when _posterior_mode is called several times on similar problems as in hyperparameter optimization. See *the Glossary*.

copy_X_train [bool, optional (default: True)] If True, a persistent copy of the training data is stored in the object. Otherwise, just a reference to the training data is stored, which might cause predictions to change if the data is modified externally.

random_state [int, RandomState instance or None, optional (default: None)] The generator used to initialize the centers. If int, random_state is the seed used by the random number

generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

multi_class [string, default] Specifies how multi-class classification problems are handled. Supported are "one_vs_rest" and "one_vs_one". In "one_vs_rest", one binary Gaussian process classifier is fitted for each class, which is trained to separate this class from the rest. In "one_vs_one", one binary Gaussian process classifier is fitted for each pair of classes, which is trained to separate these two classes. The predictions of these binary predictors are combined into multi-class predictions. Note that "one_vs_one" does not support predicting probability estimates.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

kernel_ [kernel object] The kernel used for prediction. In case of binary classification, the structure of the kernel is the same as the one passed as parameter but with optimized hyperparameters. In case of multi-class classification, a CompoundKernel is returned which consists of the different kernels used in the one-versus-rest classifiers.

log_marginal_likelihood_value_ [float] The log-marginal-likelihood of `self.kernel_.theta`

classes_ [array-like of shape (n_classes,)] Unique class labels.

n_classes_ [int] The number of classes in the training data

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.gaussian_process import GaussianProcessClassifier
>>> from sklearn.gaussian_process.kernels import RBF
>>> X, y = load_iris(return_X_y=True)
>>> kernel = 1.0 * RBF(1.0)
>>> gpc = GaussianProcessClassifier(kernel=kernel,
...         random_state=0).fit(X, y)
>>> gpc.score(X, y)
0.9866...
>>> gpc.predict_proba(X[:2,:])
array([[0.83548752, 0.03228706, 0.13222543],
       [0.79064206, 0.06525643, 0.14410151]])
```

New in version 0.18.

Methods

fit(self, X, y)	Fit Gaussian process classification model
get_params(self[, deep])	Get parameters for this estimator.
log_marginal_likelihood(self[, theta, ...])	Returns log-marginal likelihood of theta for training data.
predict(self, X)	Perform classification on an array of test vectors X.
predict_proba(self, X)	Return probability estimates for the test vector X.

Continued on next page

Table 118 – continued from previous page

score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *kernel=None*, *optimizer='fmin_l_bfgs_b'*, *n_restarts_optimizer=0*, *max_iter_predict=100*, *warm_start=False*, *copy_X_train=True*, *random_state=None*, *multi_class='one_vs_rest'*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Fit Gaussian process classification model

Parameters

X [sequence of length n_samples] Feature vectors or other representations of training data. Could either be array-like with shape = (n_samples, n_features) or a list of objects.

y [array-like of shape (n_samples,)] Target values, must be binary

Returns

self [returns an instance of self.]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

log_marginal_likelihood (*self*, *theta=None*, *eval_gradient=False*, *clone_kernel=True*)
Returns log-marginal likelihood of theta for training data.

In the case of multi-class classification, the mean log-marginal likelihood of the one-versus-rest classifiers are returned.

Parameters

theta [array-like of shape (n_kernel_params,) or None] Kernel hyperparameters for which the log-marginal likelihood is evaluated. In the case of multi-class classification, theta may be the hyperparameters of the compound kernel or of an individual kernel. In the latter case, all individual kernel get assigned the same theta values. If None, the precomputed log_marginal_likelihood of `self.kernel_.theta` is returned.

eval_gradient [bool, default: False] If True, the gradient of the log-marginal likelihood with respect to the kernel hyperparameters at position theta is returned additionally. Note that gradient computation is not supported for non-binary classification. If True, theta must not be None.

clone_kernel [bool, default=True] If True, the kernel attribute is copied. If False, the kernel attribute is modified, but may result in a performance improvement.

Returns

log_likelihood [float] Log-marginal likelihood of theta for training data.

log_likelihood_gradient [array, shape = (n_kernel_params,), optional] Gradient of the log-marginal likelihood with respect to the kernel hyperparameters at position theta. Only returned when eval_gradient is True.

predict(*self*, *X*)

Perform classification on an array of test vectors X.

Parameters

X [sequence of length n_samples] Query points where the GP is evaluated for classification. Could either be array-like with shape = (n_samples, n_features) or a list of objects.

Returns

C [ndarray of shape (n_samples,)] Predicted target values for X, values are from `classes_`

predict_proba(*self*, *X*)

Return probability estimates for the test vector X.

Parameters

X [sequence of length n_samples] Query points where the GP is evaluated for classification. Could either be array-like with shape = (n_samples, n_features) or a list of objects.

Returns

C [array-like of shape (n_samples, n_classes)] Returns the probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.gaussian_process.GaussianProcessClassifier`

- *Plot classification probability*
- *Classifier comparison*
- *Illustration of Gaussian process classification (GPC) on the XOR dataset*
- *Gaussian process classification (GPC) on iris dataset*
- *Iso-probability lines for Gaussian Processes classification (GPC)*
- *Probabilistic predictions with Gaussian process classification (GPC)*
- *Gaussian processes on discrete data structures*

7.16.2 `sklearn.gaussian_process.GaussianProcessRegressor`

class sklearn.gaussian_process.**GaussianProcessRegressor**(*kernel=None*, *alpha=1e-10*, *optimizer='fmin_l_bfgs_b'*, *n_restarts_optimizer=0*, *normalize_y=False*, *copy_X_train=True*, *random_state=None*)

Gaussian process regression (GPR).

The implementation is based on Algorithm 2.1 of Gaussian Processes for Machine Learning (GPML) by Rasmussen and Williams.

In addition to standard scikit-learn estimator API, GaussianProcessRegressor:

- allows prediction without prior fitting (based on the GP prior)
- provides an additional method sample_y(X), which evaluates samples drawn from the GPR (prior or posterior) at given inputs
- exposes a method log_marginal_likelihood(theta), which can be used externally for other ways of selecting hyperparameters, e.g., via Markov chain Monte Carlo.

Read more in the *User Guide*.

New in version 0.18.

Parameters

kernel [kernel object] The kernel specifying the covariance function of the GP. If None is passed, the kernel "1.0 * RBF(1.0)" is used as default. Note that the kernel's hyperparameters are optimized during fitting.

alpha [float or array-like, optional (default: 1e-10)] Value added to the diagonal of the kernel matrix during fitting. Larger values correspond to increased noise level in the observations. This can also prevent a potential numerical issue during fitting, by ensuring that the calculated values form a positive definite matrix. If an array is passed, it must have the same number of entries as the data used for fitting and is used as datapoint-dependent noise level. Note that this is equivalent to adding a WhiteKernel with c=alpha. Allowing to specify the noise level directly as a parameter is mainly for convenience and for consistency with Ridge.

optimizer [string or callable, optional (default: "fmin_l_bfgs_b")] Can either be one of the internally supported optimizers for optimizing the kernel's parameters, specified by a string, or an externally defined optimizer passed as a callable. If a callable is passed, it must have the signature:

```
def optimizer(obj_func, initial_theta, bounds):
    # * 'obj_func' is the objective function to be minimized, which
    #   takes the hyperparameters theta as parameter and an
    #   optional flag eval_gradient, which determines if the
    #   gradient is returned additionally to the function value
    # * 'initial_theta': the initial value for theta, which can be
    #   used by local optimizers
    # * 'bounds': the bounds on the values of theta
    ....
    # Returned are the best found hyperparameters theta and
    # the corresponding value of the target function.
    return theta_opt, func_min
```

Per default, the 'L-BGFS-B' algorithm from scipy.optimize.minimize is used. If None is passed, the kernel's parameters are kept fixed. Available internal optimizers are:

```
'fmin_l_bfgs_b'
```

n_restarts_optimizer [int, optional (default: 0)] The number of restarts of the optimizer for finding the kernel's parameters which maximize the log-marginal likelihood. The first run of the optimizer is performed from the kernel's initial parameters, the remaining ones (if any) from thetas sampled log-uniform randomly from the space of allowed theta-values. If greater than 0, all bounds must be finite. Note that n_restarts_optimizer == 0 implies that one run is performed.

normalize_y [boolean, optional (default: False)] Whether the target values y are normalized, i.e., the mean of the observed target values become zero. This parameter should be set to True if the target values' mean is expected to differ considerable from zero. When enabled, the normalization effectively modifies the GP's prior based on the data, which contradicts the likelihood principle; normalization is thus disabled per default.

copy_X_train [bool, optional (default: True)] If True, a persistent copy of the training data is stored in the object. Otherwise, just a reference to the training data is stored, which might cause predictions to change if the data is modified externally.

random_state [int, RandomState instance or None, optional (default: None)] The generator used to initialize the centers. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Attributes

X_train_ [sequence of length n_samples] Feature vectors or other representations of training data (also required for prediction). Could either be array-like with shape = (n_samples, n_features) or a list of objects.

y_train_ [array-like of shape (n_samples,) or (n_samples, n_targets)] Target values in training data (also required for prediction)

kernel_ [kernel object] The kernel used for prediction. The structure of the kernel is the same as the one passed as parameter but with optimized hyperparameters

L_ [array-like of shape (n_samples, n_samples)] Lower-triangular Cholesky decomposition of the kernel in `X_train_`

alpha_ [array-like of shape (n_samples,)] Dual coefficients of training data points in kernel space

log_marginal_likelihood_value_ [float] The log-marginal-likelihood of `self.kernel_.theta`

Examples

```
>>> from sklearn.datasets import make_friedman2
>>> from sklearn.gaussian_process import GaussianProcessRegressor
>>> from sklearn.gaussian_process.kernels import DotProduct, WhiteKernel
>>> X, y = make_friedman2(n_samples=500, noise=0, random_state=0)
>>> kernel = DotProduct() + WhiteKernel()
>>> gpr = GaussianProcessRegressor(kernel=kernel,
...         random_state=0).fit(X, y)
>>> gpr.score(X, y)
0.3680...
>>> gpr.predict(X[:2,:], return_std=True)
(array([653.0..., 592.1...]), array([316.6..., 316.6...]))
```

Methods

fit(self, X, y)	Fit Gaussian process regression model.
get_params(self[, deep])	Get parameters for this estimator.
log_marginal_likelihood(self[, theta, ...])	Returns log-marginal likelihood of theta for training data.
predict(self, X[, return_std, return_cov])	Predict using the Gaussian process regression model
sample_y(self, X[, n_samples, random_state])	Draw samples from Gaussian process and evaluate at X.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *kernel=None*, *alpha=1e-10*, *optimizer='fmin_l_bfgs_b'*, *n_restarts_optimizer=0*, *normalize_y=False*, *copy_X_train=True*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Fit Gaussian process regression model.

Parameters

X [sequence of length n_samples] Feature vectors or other representations of training data. Could either be array-like with shape = (n_samples, n_features) or a list of objects.

y [array-like of shape (n_samples,) or (n_samples, n_targets)] Target values

Returns

self [returns an instance of self.]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

log_marginal_likelihood(*self*, *theta=None*, *eval_gradient=False*, *clone_kernel=True*)
Returns log-marginal likelihood of theta for training data.

Parameters

theta [array-like of shape (n_kernel_params,) or None] Kernel hyperparameters for which the log-marginal likelihood is evaluated. If None, the precomputed log_marginal_likelihood of `self.kernel_.theta` is returned.

eval_gradient [bool, default: False] If True, the gradient of the log-marginal likelihood with respect to the kernel hyperparameters at position theta is returned additionally. If True, theta must not be None.

clone_kernel [bool, default=True] If True, the kernel attribute is copied. If False, the kernel attribute is modified, but may result in a performance improvement.

Returns

log_likelihood [float] Log-marginal likelihood of theta for training data.

log_likelihood_gradient [array, shape = (n_kernel_params,), optional] Gradient of the log-marginal likelihood with respect to the kernel hyperparameters at position theta. Only returned when eval_gradient is True.

predict(*self*, *X*, *return_std=False*, *return_cov=False*)
Predict using the Gaussian process regression model

We can also predict based on an unfitted model by using the GP prior. In addition to the mean of the predictive distribution, also its standard deviation (return_std=True) or covariance (return_cov=True). Note that at most one of the two can be requested.

Parameters

X [sequence of length n_samples] Query points where the GP is evaluated. Could either be array-like with shape = (n_samples, n_features) or a list of objects.

return_std [bool, default: False] If True, the standard-deviation of the predictive distribution at the query points is returned along with the mean.

return_cov [bool, default: False] If True, the covariance of the joint predictive distribution at the query points is returned along with the mean

Returns

y_mean [array, shape = (n_samples, [n_output_dims])] Mean of predictive distribution a query points

y_std [array, shape = (n_samples,), optional] Standard deviation of predictive distribution at query points. Only returned when return_std is True.

y_cov [array, shape = (n_samples, n_samples), optional] Covariance of joint predictive distribution a query points. Only returned when return_cov is True.

sample_y(*self*, *X*, *n_samples=1*, *random_state=0*)
Draw samples from Gaussian process and evaluate at X.

Parameters

X [sequence of length n_samples] Query points where the GP is evaluated. Could either be array-like with shape = (n_samples, n_features) or a list of objects.

n_samples [int, default: 1] The number of samples drawn from the Gaussian process

random_state [int, RandomState instance or None, optional (default=0)] If int, random_state is the seed used by the random number generator; If RandomState instance,

random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Returns

y_samples [array, shape = (n_samples_X, [n_output_dims], n_samples)] Values of n_samples samples drawn from Gaussian process and evaluated at query points.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.gaussian_process.GaussianProcessRegressor`

- *Comparison of kernel ridge and Gaussian process regression*
- *Illustration of prior and posterior Gaussian process for different kernels*
- *Gaussian process regression (GPR) with noise-level estimation*

- *Gaussian Processes regression: basic introductory example*
- *Gaussian process regression (GPR) on Mauna Loa CO2 data.*
- *Gaussian processes on discrete data structures*

Kernels:

gaussian_process.kernels.CompoundKernel(kernels)	Kernel which is composed of a set of other kernels.
gaussian_process.kernels.ConstantKernel([...])	Constant kernel.
gaussian_process.kernels.DotProduct([...])	Dot-Product kernel.
gaussian_process.kernels.ExpSineSquared([...])	Exp-Sine-Squared kernel.
gaussian_process.kernels.Exponentiation(...)	Exponentiate kernel by given exponent.
gaussian_process.kernels.Hyperparameter	A kernel hyperparameter's specification in form of a namedtuple.
gaussian_process.kernels.Kernel	Base class for all kernels.
gaussian_process.kernels.Matern([...])	Matern kernel.
gaussian_process.kernels.PairwiseKernel([...])	Wrapper for kernels in sklearn.metrics.pairwise.
gaussian_process.kernels.Product(k1, k2)	Product-kernel k1 * k2 of two kernels k1 and k2.
gaussian_process.kernels.RBF([length_scale, ...])	Radial-basis function kernel (aka squared-exponential kernel).
gaussian_process.kernels.RationalQuadratic([...])	Rational Quadratic kernel.
gaussian_process.kernels.Sum(k1, k2)	Sum-kernel k1 + k2 of two kernels k1 and k2.
gaussian_process.kernels.WhiteKernel([...])	White kernel.

7.16.3 sklearn.gaussian_process.kernels.CompoundKernel

class sklearn.gaussian_process.kernels.**CompoundKernel**(*kernels*)

Kernel which is composed of a set of other kernels.

New in version 0.18.

Parameters

kernels [list of Kernel objects] The other kernels

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

requires_vector_input Returns whether the kernel is defined on discrete structures.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *kernels*)
Initialize self. See help(type(self)) for accurate signature.

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Note that this compound kernel returns the results of all simple kernel stacked along an additional axis.

Parameters

X [sequence of length n_samples_X] Left argument of the returned kernel k(X, Y) Could either be array-like with shape = (n_samples_X, n_features) or a list of objects.

Y [sequence of length n_samples_Y] Right argument of the returned kernel k(X, Y). If None, k(X, X) is evaluated instead. Y could either be array-like with shape = (n_samples_Y, n_features) or a list of objects.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined.

Returns

K [array, shape (n_samples_X, n_samples_Y, n_kernels)] Kernel k(X, Y)

K_gradient [array, shape (n_samples_X, n_samples_X, n_dims, n_kernels)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

property bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta (*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag (*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [sequence of length n_samples_X] Argument to the kernel. Could either be array-like with shape = (n_samples_X, n_features) or a list of objects.

Returns

K_diag [array, shape (n_samples_X, n_kernels)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

property n_dims
Returns the number of non-fixed hyperparameters of the kernel.

property requires_vector_input
Returns whether the kernel is defined on discrete structures.

set_params(*self*, ***params*)
Set the parameters of this kernel.

The method works on simple kernels as well as on nested kernels. The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

property theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

7.16.4 sklearn.gaussian_process.kernels.ConstantKernel

class sklearn.gaussian_process.kernels.**ConstantKernel**(*constant_value=1.0*, *constant_value_bounds=(1e-05, 100000.0)*)

Constant kernel.

Can be used as part of a product-kernel where it scales the magnitude of the other factor (kernel) or as part of a sum-kernel, where it modifies the mean of the Gaussian process.

k(x_1, x_2) = constant_value for all x_1, x_2

New in version 0.18.

Parameters

constant_value [float, default: 1.0] The constant value which defines the covariance: k(x_1, x_2) = constant_value

constant_value_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on constant_value

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameter_constant_value

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

requires_vector_input Whether the kernel works only on fixed-length feature vectors.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *constant_value=1.0*, *constant_value_bounds=(1e-05*, *100000.0)*)
Initialize self. See help(type(self)) for accurate signature.

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [sequence of length n_samples_X] Left argument of the returned kernel k(X, Y) Could either be array-like with shape = (n_samples_X, n_features) or a list of objects.

Y [sequence of length n_samples_Y] Right argument of the returned kernel k(X, Y). If None, k(X, X) is evaluated instead. Y could either be array-like with shape = (n_samples_Y, n_features) or a list of objects.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

property bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta(*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [sequence of length n_samples_X] Argument to the kernel. Could either be array-like with shape = (n_samples_X, n_features) or a list of objects.

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

property n_dims
Returns the number of non-fixed hyperparameters of the kernel.

property requires_vector_input
Whether the kernel works only on fixed-length feature vectors.

set_params(*self*, ***params*)
Set the parameters of this kernel.

The method works on simple kernels as well as on nested kernels. The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

property theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.ConstantKernel`

- *Illustration of prior and posterior Gaussian process for different kernels*
- *Iso-probability lines for Gaussian Processes classification (GPC)*
- *Gaussian Processes regression: basic introductory example*

7.16.5 `sklearn.gaussian_process.kernels.DotProduct`

class sklearn.gaussian_process.kernels.**DotProduct**(*sigma_0=1.0*, *sigma_0_bounds=(1e-05, 100000.0)*)

Dot-Product kernel.

The DotProduct kernel is non-stationary and can be obtained from linear regression by putting N(0, 1) priors on the coefficients of x_d (d = 1, . . . , D) and a prior of N(0, sigma_0^2) on the bias. The DotProduct kernel is invariant to a rotation of the coordinates about the origin, but not translations. It is parameterized by a parameter sigma_0^2. For sigma_0^2 =0, the kernel is called the homogeneous linear kernel, otherwise it is inhomogeneous. The kernel is given by

k(x_i, x_j) = sigma_0 ^ 2 + x_i cdot x_j

The DotProduct kernel is commonly combined with exponentiation.

New in version 0.18.

Parameters

sigma_0 [float >= 0, default: 1.0] Parameter controlling the inhomogenity of the kernel. If sigma_0=0, the kernel is homogenous.

sigma_0_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on l

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameter_sigma_0

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

requires_vector_input Returns whether the kernel is defined on fixed-length feature vectors or generic objects.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *sigma_0=1.0*, *sigma_0_bounds=(1e-05*, *100000.0)*)
Initialize self. See help(type(self)) for accurate signature.

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

property bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta (*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag (*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params (*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

property n_dims
Returns the number of non-fixed hyperparameters of the kernel.

property requires_vector_input
Returns whether the kernel is defined on fixed-length feature vectors or generic objects. Defaults to True for backward compatibility.

set_params(*self*, ***params*)
Set the parameters of this kernel.

The method works on simple kernels as well as on nested kernels. The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

property theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.DotProduct`

- *Illustration of Gaussian process classification (GPC) on the XOR dataset*
- *Illustration of prior and posterior Gaussian process for different kernels*
- *Iso-probability lines for Gaussian Processes classification (GPC)*

7.16.6 sklearn.gaussian_process.kernels.ExpSineSquared

class sklearn.gaussian_process.kernels.**ExpSineSquared**(*length_scale=1.0*, *periodicity=1.0*, *length_scale_bounds=(1e-05, 100000.0)*, *periodicity_bounds=(1e-05, 100000.0)*)

Exp-Sine-Squared kernel.

The ExpSineSquared kernel allows modeling periodic functions. It is parameterized by a length-scale parameter length_scale>0 and a periodicity parameter periodicity>0. Only the isotropic variant where l is a scalar is supported at the moment. The kernel given by:

k(x_i, x_j) = exp(-2 (sin(pi / periodicity * d(x_i, x_j)) / length_scale) ^ 2)

New in version 0.18.

Parameters

length_scale [float > 0, default: 1.0] The length scale of the kernel.

periodicity [float > 0, default: 1.0] The periodicity of the kernel.

length_scale_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on length_scale

periodicity_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on periodicity

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameter_length_scale

hyperparameter_periodicity

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

requires_vector_input Returns whether the kernel is defined on fixed-length feature vectors or generic objects.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *length_scale=1.0*, *periodicity=1.0*, *length_scale_bounds=(1e-05*, *100000.0)*, *periodicity_bounds=(1e-05*, *100000.0)*)
Initialize self. See help(type(self)) for accurate signature.

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with re-

spect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

property bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta(*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [sequence of length n_samples] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

property n_dims
Returns the number of non-fixed hyperparameters of the kernel.

property requires_vector_input
Returns whether the kernel is defined on fixed-length feature vectors or generic objects. Defaults to True for backward compatibility.

set_params(*self*, ***params*)
Set the parameters of this kernel.

The method works on simple kernels as well as on nested kernels. The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

property theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.ExpSineSquared`

- *Comparison of kernel ridge and Gaussian process regression*
- *Illustration of prior and posterior Gaussian process for different kernels*
- *Gaussian process regression (GPR) on Mauna Loa CO2 data.*

7.16.7 sklearn.gaussian_process.kernels.Exponentiation

class sklearn.gaussian_process.kernels.**Exponentiation**(*kernel*, *exponent*)
Exponentiate kernel by given exponent.

The resulting kernel is defined as k_exp(X, Y) = k(X, Y) ** exponent

New in version 0.18.

Parameters

kernel [Kernel object] The base kernel

exponent [float] The exponent for the base kernel

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameters Returns a list of all hyperparameter.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

requires_vector_input Returns whether the kernel is defined on discrete structures.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.

Continued on next page

Table 125 – continued from previous page

clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *kernel*, *exponent*)
Initialize self. See help(type(self)) for accurate signature.

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [sequence of length n_samples_X] Left argument of the returned kernel k(X, Y) Could either be array-like with shape = (n_samples_X, n_features) or a list of objects.

Y [sequence of length n_samples_Y] Right argument of the returned kernel k(X, Y). If None, k(X, X) is evaluated instead. Y could either be array-like with shape = (n_samples_Y, n_features) or a list of objects.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

property bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta (*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag (*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [sequence of length n_samples_X] Argument to the kernel. Could either be array-like with shape = (n_samples_X, n_features) or a list of objects.

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property hyperparameters
Returns a list of all hyperparameter.

is_stationary(*self*)
Returns whether the kernel is stationary.

property n_dims
Returns the number of non-fixed hyperparameters of the kernel.

property requires_vector_input
Returns whether the kernel is defined on discrete structures.

set_params(*self*, ***params*)
Set the parameters of this kernel.

The method works on simple kernels as well as on nested kernels. The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

property theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

7.16.8 sklearn.gaussian_process.kernels.Hyperparameter

class sklearn.gaussian_process.kernels.**Hyperparameter**
A kernel hyperparameter's specification in form of a namedtuple.

New in version 0.18.

Attributes

name [string] The name of the hyperparameter. Note that a kernel using a hyperparameter with name "x" must have the attributes self.x and self.x_bounds

value_type [string] The type of the hyperparameter. Currently, only "numeric" hyperparameters are supported.

bounds [pair of floats >= 0 or "fixed"] The lower and upper bound on the parameter. If n_elements>1, a pair of 1d array with n_elements each may be given alternatively. If the string "fixed" is passed as bounds, the hyperparameter's value cannot be changed.

n_elements [int, default=1] The number of elements of the hyperparameter value. Defaults to 1, which corresponds to a scalar hyperparameter. n_elements > 1 corresponds to a hyperparameter which is vector-valued, such as, e.g., anisotropic length-scales.

fixed [bool, default: None] Whether the value of this hyperparameter is fixed, i.e., cannot be changed during hyperparameter tuning. If None is passed, the "fixed" is derived based on the given bounds.

Methods

count(self, value, /)	Return number of occurrences of value.
index(self, value[, start, stop])	Return first index of value.

__init__ (*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__call__ (**args*, ***kwargs*)
Call self as a function.

bounds
Alias for field number 2

count (*self*, *value*, */*)
Return number of occurrences of value.

fixed
Alias for field number 4

index (*self*, *value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

n_elements
Alias for field number 3

name
Alias for field number 0

value_type
Alias for field number 1

Examples using `sklearn.gaussian_process.kernels.Hyperparameter`

- *Gaussian processes on discrete data structures*

7.16.9 sklearn.gaussian_process.kernels.Kernel

class sklearn.gaussian_process.kernels.**Kernel**
Base class for all kernels.

New in version 0.18.

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

requires_vector_input Returns whether the kernel is defined on fixed-length feature vectors or generic objects.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Evaluate the kernel.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

abstract __call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Evaluate the kernel.

property bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta (*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

abstract diag (*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [sequence of length n_samples] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params (*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property hyperparameters
Returns a list of all hyperparameter specifications.

abstract is_stationary(*self*)
Returns whether the kernel is stationary.

property n_dims
Returns the number of non-fixed hyperparameters of the kernel.

property requires_vector_input
Returns whether the kernel is defined on fixed-length feature vectors or generic objects. Defaults to True for backward compatibility.

set_params(*self*, ***params*)
Set the parameters of this kernel.

The method works on simple kernels as well as on nested kernels. The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

property theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.Kernel`

- *Gaussian processes on discrete data structures*

7.16.10 `sklearn.gaussian_process.kernels`.Matern

class sklearn.gaussian_process.kernels.**Matern**(*length_scale=1.0*, *length_scale_bounds=(1e-05, 100000.0)*, *nu=1.5*)
Matern kernel.

The class of Matern kernels is a generalization of the RBF and the absolute exponential kernel parameterized by an additional parameter nu. The smaller nu, the less smooth the approximated function is. For nu=inf, the kernel becomes equivalent to the RBF kernel and for nu=0.5 to the absolute exponential kernel. Important intermediate values are nu=1.5 (once differentiable functions) and nu=2.5 (twice differentiable functions).

See Rasmussen and Williams 2006, pp84 for details regarding the different variants of the Matern kernel.

New in version 0.18.

Parameters

length_scale [float or array with shape (n_features,), default: 1.0] The length scale of the kernel. If a float, an isotropic kernel is used. If an array, an anisotropic kernel is used where each dimension of l defines the length-scale of the respective feature dimension.

length_scale_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on length_scale

nu [float, default: 1.5] The parameter nu controlling the smoothness of the learned function. The smaller nu, the less smooth the approximated function is. For nu=inf, the kernel becomes equivalent to the RBF kernel and for nu=0.5 to the absolute exponential kernel. Important intermediate values are nu=1.5 (once differentiable functions) and nu=2.5 (twice differentiable functions). Note that values of nu not in [0.5, 1.5, 2.5, inf] incur a considerably higher computational cost (appr. 10 times higher) since they require to evaluate the modified Bessel function. Furthermore, in contrast to l, nu is kept fixed to its initial value and not optimized.

Attributes

anisotropic

bounds Returns the log-transformed bounds on the theta.

hyperparameter_length_scale

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

requires_vector_input Returns whether the kernel is defined on fixed-length feature vectors or generic objects.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *length_scale=1.0*, *length_scale_bounds=(1e-05*, *100000.0)*, *nu=1.5*)
Initialize self. See help(type(self)) for accurate signature.

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

property bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta(*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [sequence of length n_samples] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

property n_dims
Returns the number of non-fixed hyperparameters of the kernel.

property requires_vector_input
Returns whether the kernel is defined on fixed-length feature vectors or generic objects. Defaults to True for backward compatibility.

set_params(*self*, ***params*)
Set the parameters of this kernel.

The method works on simple kernels as well as on nested kernels. The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

property theta

Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.Matern`

- *Illustration of prior and posterior Gaussian process for different kernels*

7.16.11 `sklearn.gaussian_process.kernels`.PairwiseKernel

class sklearn.gaussian_process.kernels.**PairwiseKernel**(*gamma=1.0*, *gamma_bounds=(1e-05, 100000.0)*, *metric='linear'*, *pairwise_kernels_kwargs=None*)

Wrapper for kernels in sklearn.metrics.pairwise.

A thin wrapper around the functionality of the kernels in sklearn.metrics.pairwise.

Note: Evaluation of eval_gradient is not analytic but numeric and all kernels support only isotropic distances. The parameter gamma is considered to be a hyperparameter and may be optimized. The other kernel parameters are set directly at initialization and are kept fixed.

New in version 0.18.

Parameters

gamma [float >= 0, default: 1.0] Parameter gamma of the pairwise kernel specified by metric

gamma_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on gamma

metric [string, or callable, default: "linear"] The metric to use when calculating kernel between instances in a feature array. If metric is a string, it must be one of the metrics in pairwise.PAIRWISE_KERNEL_FUNCTIONS. If metric is "precomputed", X is assumed to be a kernel matrix. Alternatively, if metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays from X as input and return a value indicating the distance between them.

pairwise_kernels_kwargs [dict, default: None] All entries of this dict (if any) are passed as keyword arguments to the pairwise kernel function.

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameter_gamma

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

requires_vector_input Returns whether the kernel is defined on fixed-length feature vectors or generic objects.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *gamma=1.0*, *gamma_bounds=(1e-05*, *100000.0)*, *metric='linear'*, *pairwise_kernels_kwargs=None*)
Initialize self. See help(type(self)) for accurate signature.

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

property bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta (*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag (*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

property n_dims
Returns the number of non-fixed hyperparameters of the kernel.

property requires_vector_input
Returns whether the kernel is defined on fixed-length feature vectors or generic objects. Defaults to True for backward compatibility.

set_params(*self*, ***params*)
Set the parameters of this kernel.

The method works on simple kernels as well as on nested kernels. The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

property theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

7.16.12 sklearn.gaussian_process.kernels.Product

class sklearn.gaussian_process.kernels.**Product**(*k1*, *k2*)
Product-kernel k1 * k2 of two kernels k1 and k2.

The resulting kernel is defined as k_prod(X, Y) = k1(X, Y) * k2(X, Y)

New in version 0.18.

Parameters

k1 [Kernel object] The first base-kernel of the product-kernel

k2 [Kernel object] The second base-kernel of the product-kernel

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameters Returns a list of all hyperparameter.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

requires_vector_input Returns whether the kernel is stationary.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *k1*, *k2*)
Initialize self. See help(type(self)) for accurate signature.

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [sequence of length n_samples_X] Left argument of the returned kernel k(X, Y) Could either be array-like with shape = (n_samples_X, n_features) or a list of objects.

Y [sequence of length n_samples_Y] Right argument of the returned kernel k(X, Y). If None, k(X, X) is evaluated instead. Y could either be array-like with shape = (n_samples_Y, n_features) or a list of objects.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

property bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta(*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [sequence of length n_samples_X] Argument to the kernel. Could either be array-like with shape = (n_samples_X, n_features) or a list of objects.

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property hyperparameters
Returns a list of all hyperparameter.

is_stationary(*self*)
Returns whether the kernel is stationary.

property n_dims
Returns the number of non-fixed hyperparameters of the kernel.

property requires_vector_input
Returns whether the kernel is stationary.

set_params(*self*, ***params*)
Set the parameters of this kernel.

The method works on simple kernels as well as on nested kernels. The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

property theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

7.16.13 sklearn.gaussian_process.kernels.RBF

class sklearn.gaussian_process.kernels.**RBF**(*length_scale=1.0*, *length_scale_bounds=(1e-05, 100000.0)*)

Radial-basis function kernel (aka squared-exponential kernel).

The RBF kernel is a stationary kernel. It is also known as the "squared exponential" kernel. It is parameterized by a length-scale parameter length_scale>0, which can either be a scalar (isotropic variant of the kernel) or a vector with the same number of dimensions as the inputs X (anisotropic variant of the kernel). The kernel is given by:

k(x_i, x_j) = exp(-1 / 2 d(x_i / length_scale, x_j / length_scale)^2)

This kernel is infinitely differentiable, which implies that GPs with this kernel as covariance function have mean square derivatives of all orders, and are thus very smooth.

New in version 0.18.

Parameters

length_scale [float or array with shape (n_features,), default: 1.0] The length scale of the kernel. If a float, an isotropic kernel is used. If an array, an anisotropic kernel is used where each dimension of l defines the length-scale of the respective feature dimension.

length_scale_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on length_scale

Attributes

anisotropic

bounds Returns the log-transformed bounds on the theta.

hyperparameter_length_scale

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

requires_vector_input Returns whether the kernel is defined on fixed-length feature vectors or generic objects.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__(*self*, *length_scale=1.0*, *length_scale_bounds=(1e-05, 100000.0)*)
Initialize self. See help(type(self)) for accurate signature.

__call__(*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

property bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta(*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [sequence of length n_samples] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

property n_dims
Returns the number of non-fixed hyperparameters of the kernel.

property requires_vector_input
Returns whether the kernel is defined on fixed-length feature vectors or generic objects. Defaults to True for backward compatibility.

set_params(*self*, ***params*)
Set the parameters of this kernel.

The method works on simple kernels as well as on nested kernels. The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

property theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.RBF`

- *Plot classification probability*
- *Classifier comparison*
- *Illustration of Gaussian process classification (GPC) on the XOR dataset*
- *Gaussian process classification (GPC) on iris dataset*
- *Illustration of prior and posterior Gaussian process for different kernels*
- *Probabilistic predictions with Gaussian process classification (GPC)*
- *Gaussian process regression (GPR) with noise-level estimation*
- *Gaussian Processes regression: basic introductory example*
- *Gaussian process regression (GPR) on Mauna Loa CO2 data.*

7.16.14 sklearn.gaussian_process.kernels.RationalQuadratic

class sklearn.gaussian_process.kernels.**RationalQuadratic**(*length_scale=1.0*, *alpha=1.0*, *length_scale_bounds=(1e-05, 100000.0)*, *alpha_bounds=(1e-05, 100000.0)*)

Rational Quadratic kernel.

The RationalQuadratic kernel can be seen as a scale mixture (an infinite sum) of RBF kernels with different characteristic length-scales. It is parameterized by a length-scale parameter length_scale>0 and a scale mixture parameter alpha>0. Only the isotropic variant where length_scale is a scalar is supported at the moment. The kernel given by:

k(x_i, x_j) = (1 + d(x_i, x_j)^2 / (2*alpha * length_scale^2))^-alpha

New in version 0.18.

Parameters

length_scale [float > 0, default: 1.0] The length scale of the kernel.

alpha [float > 0, default: 1.0] Scale mixture parameter

length_scale_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on length_scale

alpha_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on alpha

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameter_alpha

hyperparameter_length_scale

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

requires_vector_input Returns whether the kernel is defined on fixed-length feature vectors or generic objects.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__(*self*, *length_scale=1.0*, *alpha=1.0*, *length_scale_bounds=(1e-05*, *100000.0)*, *alpha_bounds=(1e-05*, *100000.0)*)
Initialize self. See help(type(self)) for accurate signature.

__call__(*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when

eval_gradient is True.

property bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta(*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [sequence of length n_samples] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

property n_dims
Returns the number of non-fixed hyperparameters of the kernel.

property requires_vector_input
Returns whether the kernel is defined on fixed-length feature vectors or generic objects. Defaults to True for backward compatibility.

set_params(*self*, ***params*)
Set the parameters of this kernel.

The method works on simple kernels as well as on nested kernels. The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

property theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.RationalQuadratic`

- *Illustration of prior and posterior Gaussian process for different kernels*
- *Gaussian process regression (GPR) on Mauna Loa CO2 data.*

7.16.15 `sklearn.gaussian_process.kernels.Sum`

class sklearn.gaussian_process.kernels.**Sum**(*k1*, *k2*)
Sum-kernel k1 + k2 of two kernels k1 and k2.

The resulting kernel is defined as k_sum(X, Y) = k1(X, Y) + k2(X, Y)

New in version 0.18.

Parameters

k1 [Kernel object] The first base-kernel of the sum-kernel

k2 [Kernel object] The second base-kernel of the sum-kernel

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameters Returns a list of all hyperparameter.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

requires_vector_input Returns whether the kernel is stationary.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__(*self*, *k1*, *k2*)
Initialize self. See help(type(self)) for accurate signature.

__call__(*self*, *X*, *Y=None*, *eval_gradient=False*)

Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [sequence of length n_samples_X] Left argument of the returned kernel k(X, Y) Could either be array-like with shape = (n_samples_X, n_features) or a list of objects.

Y [sequence of length n_samples_Y] Right argument of the returned kernel k(X, Y). If None, k(X, X) is evaluated instead. Y could either be array-like with shape = (n_samples_Y, n_features) or a list of objects.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

property bounds

Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta(*self*, *theta*)

Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)

Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [sequence of length n_samples_X] Argument to the kernel. Could either be array-like with shape = (n_samples_X, n_features) or a list of objects.

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)

Get parameters of this kernel.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property hyperparameters

Returns a list of all hyperparameter.

is_stationary(*self*)
Returns whether the kernel is stationary.

property n_dims
Returns the number of non-fixed hyperparameters of the kernel.

property requires_vector_input
Returns whether the kernel is stationary.

set_params(*self*, ***params*)
Set the parameters of this kernel.

The method works on simple kernels as well as on nested kernels. The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

property theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

7.16.16 sklearn.gaussian_process.kernels.WhiteKernel

class sklearn.gaussian_process.kernels.**WhiteKernel**(*noise_level=1.0*, *noise_level_bounds=(1e-05*, *100000.0)*)

White kernel.

The main use-case of this kernel is as part of a sum-kernel where it explains the noise of the signal as independently and identically normally-distributed. The parameter noise_level equals the variance of this noise.

k(x_1, x_2) = noise_level if x_1 == x_2 else 0

New in version 0.18.

Parameters

noise_level [float, default: 1.0] Parameter controlling the noise level (variance)

noise_level_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on noise_level

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameter_noise_level

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

requires_vector_input Whether the kernel works only on fixed-length feature vectors.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *noise_level=1.0*, *noise_level_bounds=(1e-05*, *100000.0)*)
Initialize self. See help(type(self)) for accurate signature.

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [sequence of length n_samples_X] Left argument of the returned kernel k(X, Y) Could either be array-like with shape = (n_samples_X, n_features) or a list of objects.

Y [sequence of length n_samples_Y] Right argument of the returned kernel k(X, Y). If None, k(X, X) is evaluated instead. Y could either be array-like with shape = (n_samples_Y, n_features) or a list of objects.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

property bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta (*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag (*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [sequence of length n_samples_X] Argument to the kernel. Could either be array-like with shape = (n_samples_X, n_features) or a list of objects.

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params (*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary (*self*)
Returns whether the kernel is stationary.

property n_dims
Returns the number of non-fixed hyperparameters of the kernel.

property requires_vector_input
Whether the kernel works only on fixed-length feature vectors.

set_params (*self*, ***params*)
Set the parameters of this kernel.

The method works on simple kernels as well as on nested kernels. The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

property theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.WhiteKernel`

- *Comparison of kernel ridge and Gaussian process regression*
- *Gaussian process regression (GPR) with noise-level estimation*
- *Gaussian process regression (GPR) on Mauna Loa CO2 data.*

7.17 `sklearn.impute`: Impute

Transformers for missing value imputation

User guide: See the *Imputation of missing values* section for further details.

impute.SimpleImputer([missing_values, ...])	Imputation transformer for completing missing values.
impute.IterativeImputer([estimator, ...])	Multivariate imputer that estimates each feature from all the others.
impute.MissingIndicator([missing_values, ...])	Binary indicators for missing values.
impute.KNNImputer([missing_values, ...])	Imputation for completing missing values using k-Nearest Neighbors.

7.17.1 sklearn.impute.SimpleImputer

class sklearn.impute.**SimpleImputer**(*missing_values=nan*, *strategy='mean'*, *fill_value=None*, *verbose=0*, *copy=True*, *add_indicator=False*)

Imputation transformer for completing missing values.

Read more in the *User Guide*.

Parameters

missing_values [number, string, np.nan (default) or None] The placeholder for the missing values. All occurrences of `missing_values` will be imputed.

strategy [string, default='mean'] The imputation strategy.

- If "mean", then replace missing values using the mean along each column. Can only be used with numeric data.
- If "median", then replace missing values using the median along each column. Can only be used with numeric data.
- If "most_frequent", then replace missing using the most frequent value along each column. Can be used with strings or numeric data.
- If "constant", then replace missing values with fill_value. Can be used with strings or numeric data.

New in version 0.20: strategy="constant" for fixed value imputation.

fill_value [string or numerical value, default=None] When strategy == "constant", fill_value is used to replace all occurrences of missing_values. If left to the default, fill_value will be 0 when imputing numerical data and "missing_value" for strings or object data types.

verbose [integer, default=0] Controls the verbosity of the imputer.

copy [boolean, default=True] If True, a copy of X will be created. If False, imputation will be done in-place whenever possible. Note that, in the following cases, a new copy will always be made, even if `copy=False`:

- If X is not an array of floating values;
- If X is encoded as a CSR matrix;
- If add_indicator=True.

add_indicator [boolean, default=False] If True, a *MissingIndicator* transform will stack onto output of the imputer's transform. This allows a predictive estimator to account for missingness despite imputation. If a feature has no missing values at fit/train time, the feature won't appear on the missing indicator even if there are missing values at transform/test time.

Attributes

statistics_ [array of shape (n_features,)] The imputation fill value for each feature. Computing statistics can result in `np.nan` values. During *transform*, features corresponding to `np.nan` statistics will be discarded.

indicator_ [*sklearn.impute.MissingIndicator*] Indicator used to add binary indicators for missing values. `None` if add_indicator is False.

See also:

IterativeImputer Multivariate imputation of missing values.

Notes

Columns which only contained missing values at *fit* are discarded upon *transform* if strategy is not "constant".

Examples

```
>>> import numpy as np
>>> from sklearn.impute import SimpleImputer
>>> imp_mean = SimpleImputer(missing_values=np.nan, strategy='mean')
>>> imp_mean.fit([[7, 2, 3], [4, np.nan, 6], [10, 5, 9]])
SimpleImputer()
>>> X = [[np.nan, 2, 3], [4, np.nan, 6], [10, np.nan, 9]]
>>> print(imp_mean.transform(X))
[[ 7.   2.   3. ]
 [ 4.   3.5  6. ]
 [10.   3.5  9. ]]
```

Methods

fit(self, X[, y])	Fit the imputer on X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Impute all missing values in X.

__init__(*self*, *missing_values=nan*, *strategy='mean'*, *fill_value=None*, *verbose=0*, *copy=True*, *add_indicator=False*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the imputer on X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Input data, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns

self [SimpleImputer]

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Impute all missing values in X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data to complete.

Examples using `sklearn.impute.SimpleImputer`

- *Permutation Importance vs Random Forest Feature Importance (MDI)*
- *Imputing missing values with variants of IterativeImputer*
- *Imputing missing values before building an estimator*
- *Column Transformer with Mixed Types*

7.17.2 sklearn.impute.IterativeImputer

class sklearn.impute.**IterativeImputer**(*estimator=None*, *missing_values=nan*, *sample_posterior=False*, *max_iter=10*, *tol=0.001*, *n_nearest_features=None*, *initial_strategy='mean'*, *imputation_order='ascending'*, *skip_complete=False*, *min_value=None*, *max_value=None*, *verbose=0*, *random_state=None*, *add_indicator=False*)

Multivariate imputer that estimates each feature from all the others.

A strategy for imputing missing values by modeling each feature with missing values as a function of other features in a round-robin fashion.

Read more in the *User Guide*.

Note: This estimator is still **experimental** for now: the predictions and the API might change without any deprecation cycle. To use it, you need to explicitly import `enable_iterative_imputer`:

```
>>> # explicitly require this experimental feature
>>> from sklearn.experimental import enable_iterative_imputer  # noqa
>>> # now you can import normally from sklearn.impute
>>> from sklearn.impute import IterativeImputer
```

Parameters

estimator [estimator object, default=BayesianRidge()] The estimator to use at each step of the round-robin imputation. If `sample_posterior` is True, the estimator must support `return_std` in its `predict` method.

missing_values [int, np.nan, default=np.nan] The placeholder for the missing values. All occurrences of `missing_values` will be imputed.

sample_posterior [boolean, default=False] Whether to sample from the (Gaussian) predictive posterior of the fitted estimator for each imputation. Estimator must support `return_std` in its `predict` method if set to `True`. Set to `True` if using `IterativeImputer` for multiple imputations.

max_iter [int, default=10] Maximum number of imputation rounds to perform before returning the imputations computed during the final round. A round is a single imputation of each feature with missing values. The stopping criterion is met once `abs(max(X_t - X_{t-1}))/abs(max(X[known_vals]))` < tol, where `X_t` is `X` at iteration `t. Note that early stopping is only applied if ``sample_posterior=False``.`

tol [float, default=1e-3] Tolerance of the stopping condition.

n_nearest_features [int, default=None] Number of other features to use to estimate the missing values of each feature column. Nearness between features is measured using the absolute correlation coefficient between each feature pair (after initial imputation). To ensure coverage of features throughout the imputation process, the neighbor features are not necessarily nearest, but are drawn with probability proportional to correlation for each imputed target feature. Can provide significant speed-up when the number of features is huge. If `None`, all features will be used.

initial_strategy [str, default='mean'] Which strategy to use to initialize the missing values. Same as the `strategy` parameter in *sklearn.impute.SimpleImputer* Valid values: {"mean", "median", "most_frequent", or "constant"}.

imputation_order [str, default='ascending'] The order in which the features will be imputed. Possible values:

"ascending" From features with fewest missing values to most.

"descending" From features with most missing values to fewest.

"roman" Left to right.

"arabic" Right to left.

"random" A random order for each round.

skip_complete [boolean, default=False] If `True` then features with missing values during `transform` which did not have any missing values during `fit` will be imputed with the initial imputation method only. Set to `True` if you have many features with no missing values at both `fit` and `transform` time to save compute.

min_value [float, default=None] Minimum possible imputed value. Default of `None` will set minimum to negative infinity.

max_value [float, default=None] Maximum possible imputed value. Default of `None` will set maximum to positive infinity.

verbose [int, default=0] Verbosity flag, controls the debug messages that are issued as functions are evaluated. The higher, the more verbose. Can be 0, 1, or 2.

random_state [int, RandomState instance or None, default=None] The seed of the pseudo random number generator to use. Randomizes selection of estimator features if n_nearest_features is not None, the `imputation_order` if `random`, and the sampling from posterior if `sample_posterior` is True. Use an integer for determinism. See *the Glossary*.

add_indicator [boolean, default=False] If True, a *`MissingIndicator`* transform will stack onto output of the imputer's transform. This allows a predictive estimator to account for missingness despite imputation. If a feature has no missing values at fit/train time, the feature won't appear on the missing indicator even if there are missing values at transform/test time.

Attributes

initial_imputer_ [object of type *`sklearn.impute.SimpleImputer`*] Imputer used to initialize the missing values.

imputation_sequence_ [list of tuples] Each tuple has `(feat_idx, neighbor_feat_idx, estimator)`, where `feat_idx` is the current feature to be imputed, `neighbor_feat_idx` is the array of other features used to impute the current feature, and `estimator` is the trained estimator used for the imputation. Length is `self.n_features_with_missing_ * self.n_iter_`.

n_iter_ [int] Number of iteration rounds that occurred. Will be less than `self.max_iter` if early stopping criterion was reached.

n_features_with_missing_ [int] Number of features with missing values.

indicator_ [*`sklearn.impute.MissingIndicator`*] Indicator used to add binary indicators for missing values. `None` if add_indicator is False.

random_state_ [RandomState instance] RandomState instance that is generated either from a seed, the random number generator or by `np.random`.

See also:

SimpleImputer Univariate imputation of missing values.

Notes

To support imputation in inductive mode we store each feature's estimator during the `fit` phase, and predict without refitting (in order) during the `transform` phase.

Features which contain all missing values at `fit` are discarded upon `transform`.

References

[Rcd31b817a31e-1], [Rcd31b817a31e-2]

Examples

```
>>> import numpy as np
>>> from sklearn.experimental import enable_iterative_imputer
>>> from sklearn.impute import IterativeImputer
>>> imp_mean = IterativeImputer(random_state=0)
>>> imp_mean.fit([[7, 2, 3], [4, np.nan, 6], [10, 5, 9]])
IterativeImputer(random_state=0)
>>> X = [[np.nan, 2, 3], [4, np.nan, 6], [10, np.nan, 9]]
>>> imp_mean.transform(X)
array([[ 6.9584...,  2.       ,  3.        ],
       [ 4.       ,  2.6000...,  6.        ],
       [10.       ,  4.9999...,  9.        ]])
```

Methods

fit(self, X[, y])	Fits the imputer on X and return self.
fit_transform(self, X[, y])	Fits the imputer on X and return the transformed X.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Imputes all missing values in X.

__init__(*self*, *estimator=None*, *missing_values=nan*, *sample_posterior=False*, *max_iter=10*, *tol=0.001*, *n_nearest_features=None*, *initial_strategy='mean'*, *imputation_order='ascending'*, *skip_complete=False*, *min_value=None*, *max_value=None*, *verbose=0*, *random_state=None*, *add_indicator=False*)

Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)

Fits the imputer on X and return self.

Parameters

X [array-like, shape (n_samples, n_features)] Input data, where "n_samples" is the number of samples and "n_features" is the number of features.

y [ignored]

Returns

self [object] Returns self.

fit_transform(*self*, *X*, *y=None*)
Fits the imputer on X and return the transformed X.

Parameters

X [array-like, shape (n_samples, n_features)] Input data, where "n_samples" is the number of samples and "n_features" is the number of features.

y [ignored.]

Returns

Xt [array-like, shape (n_samples, n_features)] The imputed input data.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Imputes all missing values in X.

Note that this is stochastic, and that if random_state is not fixed, repeated calls, or permuted input, will yield different results.

Parameters

X [array-like of shape (n_samples, n_features)] The input data to complete.

Returns

Xt [array-like, shape (n_samples, n_features)] The imputed input data.

Examples using `sklearn.impute.IterativeImputer`

- *Imputing missing values with variants of IterativeImputer*
- *Imputing missing values before building an estimator*

7.17.3 sklearn.impute.MissingIndicator

class sklearn.impute.**MissingIndicator**(*missing_values=nan*, *features='missing-only'*, *sparse='auto'*, *error_on_new=True*)

Binary indicators for missing values.

Note that this component typically should not be used in a vanilla `Pipeline` consisting of transformers and a classifier, but rather could be added using a `FeatureUnion` or `ColumnTransformer`.

Read more in the *User Guide*.

Parameters

missing_values [number, string, np.nan (default) or None] The placeholder for the missing values. All occurrences of `missing_values` will be indicated (True in the output array), the other values will be marked as False.

features [str, default=None] Whether the imputer mask should represent all or a subset of features.

- If "missing-only" (default), the imputer mask will only represent features containing missing values during fit time.
- If "all", the imputer mask will represent all features.

sparse [boolean or "auto", default=None] Whether the imputer mask format should be sparse or dense.

- If "auto" (default), the imputer mask will be of same type as input.
- If True, the imputer mask will be a sparse matrix.
- If False, the imputer mask will be a numpy array.

error_on_new [boolean, default=None] If True (default), transform will raise an error when there are features with missing values in transform that have no missing values in fit. This is applicable only when `features="missing-only"`.

Attributes

features_ [ndarray, shape (n_missing_features,) or (n_features,)] The features indices which will be returned when calling `transform`. They are computed during `fit`. For `features='all'`, it is to `range(n_features)`.

Examples

```
>>> import numpy as np
>>> from sklearn.impute import MissingIndicator
>>> X1 = np.array([[np.nan, 1, 3],
...                [4, 0, np.nan],
...                [8, 1, 0]])
>>> X2 = np.array([[5, 1, np.nan],
...                [np.nan, 2, 3],
...                [2, 4, 0]])
>>> indicator = MissingIndicator()
>>> indicator.fit(X1)
MissingIndicator()
>>> X2_tr = indicator.transform(X2)
>>> X2_tr
array([[False,  True],
```

(continues on next page)

(continued from previous page)

```
       [ True, False],
       [False, False]])
```

Methods

fit(self, X[, y])	Fit the transformer on X.
fit_transform(self, X[, y])	Generate missing values indicator for X.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Generate missing values indicator for X.

__init__ (*self*, *missing_values=nan*, *features='missing-only'*, *sparse='auto'*, *error_on_new=True*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*)
Fit the transformer on X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Input data, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns

self [object] Returns self.

fit_transform (*self*, *X*, *y=None*)
Generate missing values indicator for X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data to complete.

Returns

Xt [{ndarray or sparse matrix}, shape (n_samples, n_features) or (n_samples, n_features_with_missing)] The missing indicator for input data. The data type of `Xt` will be boolean.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Generate missing values indicator for X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data to complete.

Returns

Xt [{ndarray or sparse matrix}, shape (n_samples, n_features) or (n_samples, n_features_with_missing)] The missing indicator for input data. The data type of `Xt` will be boolean.

Examples using `sklearn.impute.MissingIndicator`

- *Imputing missing values before building an estimator*

7.17.4 `sklearn.impute`.KNNImputer

class sklearn.impute.**KNNImputer**(*missing_values=nan*, *n_neighbors=5*, *weights='uniform'*, *metric='nan_euclidean'*, *copy=True*, *add_indicator=False*)
Imputation for completing missing values using k-Nearest Neighbors.

Each sample's missing values are imputed using the mean value from `n_neighbors` nearest neighbors found in the training set. Two samples are close if the features that neither is missing are close.

Read more in the *User Guide*.

New in version 0.22.

Parameters

missing_values [number, string, np.nan or None, default='np.nan'] The placeholder for the missing values. All occurrences of `missing_values` will be imputed.

n_neighbors [int, default=5] Number of neighboring samples to use for imputation.

weights [{'uniform', 'distance'} or callable, default='uniform'] Weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- callable : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

metric [{'nan_euclidean'} or callable, default='nan_euclidean'] Distance metric for searching neighbors. Possible values:

- 'nan_euclidean'
- callable : a user-defined function which conforms to the definition of `_pairwise_callable(X, Y, metric, **kwds)`. The function accepts two arrays, X and Y, and a `missing_values` keyword in `kwds` and returns a scalar distance value.

copy [bool, default=True] If True, a copy of X will be created. If False, imputation will be done in-place whenever possible.

add_indicator [bool, default=False] If True, a *MissingIndicator* transform will stack onto the output of the imputer's transform. This allows a predictive estimator to account for missingness despite imputation. If a feature has no missing values at fit/train time, the feature won't appear on the missing indicator even if there are missing values at transform/test time.

Attributes

indicator_ [*sklearn.impute.MissingIndicator*] Indicator used to add binary indicators for missing values. `None` if add_indicator is False.

References

- Olga Troyanskaya, Michael Cantor, Gavin Sherlock, Pat Brown, Trevor Hastie, Robert Tibshirani, David Botstein and Russ B. Altman, Missing value estimation methods for DNA microarrays, BIOINFORMATICS Vol. 17 no. 6, 2001 Pages 520-525.

Examples

```
>>> import numpy as np
>>> from sklearn.impute import KNNImputer
>>> X = [[1, 2, np.nan], [3, 4, 3], [np.nan, 6, 5], [8, 8, 7]]
>>> imputer = KNNImputer(n_neighbors=2)
>>> imputer.fit_transform(X)
array([[1. , 2. , 4. ],
       [3. , 4. , 3. ],
       [5.5, 6. , 5. ],
       [8. , 8. , 7. ]])
```

Methods

fit(self, X[, y])	Fit the imputer on X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Impute all missing values in X.

__init__(*self*, *missing_values=nan*, *n_neighbors=5*, *weights='uniform'*, *metric='nan_euclidean'*, *copy=True*, *add_indicator=False*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the imputer on X.

Parameters

X [array-like shape of (n_samples, n_features)] Input data, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns

self [object]

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Impute all missing values in X.

Parameters

X [array-like of shape (n_samples, n_features)] The input data to complete.

Returns

X [array-like of shape (n_samples, n_output_features)] The imputed dataset. `n_output_features` is the number of features that is not always missing during `fit`.

Examples using `sklearn.impute.KNNImputer`

- *Imputing missing values before building an estimator*
- *Release Highlights for scikit-learn 0.22*

7.18 sklearn.inspection: inspection

The *sklearn.inspection* module includes tools for model inspection.

inspection.partial_dependence(estimator, X, ...)	Partial dependence of `features`.
inspection.permutation_importance(estimator, ...)	Permutation importance for feature evaluation [BRE].

7.18.1 sklearn.inspection.partial_dependence

sklearn.inspection.**partial_dependence**(*estimator*, *X*, *features*, *response_method='auto'*, *percentiles=(0.05, 0.95)*, *grid_resolution=100*, *method='auto'*)

Partial dependence of `features`.

Partial dependence of a feature (or a set of features) corresponds to the average response of an estimator for each possible value of the feature.

Read more in the *User Guide*.

Parameters

estimator [BaseEstimator] A fitted estimator object implementing *predict*, *predict_proba*, or *decision_function*. Multioutput-multiclass classifiers are not supported.

X [{array-like or dataframe} of shape (n_samples, n_features)] `X` is used both to generate a grid of values for the `features`, and to compute the averaged predictions when method is 'brute'.

features [array-like of {int, str}] The feature (e.g. `[0]`) or pair of interacting features (e.g. `[(0, 1)]`) for which the partial dependency should be computed.

response_method ['auto', 'predict_proba' or 'decision_function', optional (default='auto')] Specifies whether to use *predict_proba* or *decision_function* as the target response. For regressors this parameter is ignored and the response is always the output of *predict*. By default, *predict_proba* is tried first and we revert to *decision_function* if it doesn't exist. If `method` is 'recursion', the response is always the output of *decision_function*.

percentiles [tuple of float, optional (default=(0.05, 0.95))] The lower and upper percentile used to create the extreme values for the grid. Must be in [0, 1].

grid_resolution [int, optional (default=100)] The number of equally spaced points on the grid, for each target feature.

method [str, optional (default='auto')] The method used to calculate the averaged predictions:

- 'recursion' is only supported for gradient boosting estimator (namely *GradientBoostingClassifier*, *GradientBoostingRegressor*, *HistGradientBoostingClassifier*, *HistGradientBoostingRegressor*) but is more efficient in terms of speed. With this method, `X` is only used to build the grid and the partial dependences are computed using the training data. This method does not account for the `init` predictor of the boosting process, which may lead to incorrect values (see warning below). With this method, the target response of a classifier is always the decision function, not the predicted probabilities.
- 'brute' is supported for any estimator, but is more computationally intensive.

- 'auto':
  - 'recursion' is used for *GradientBoostingClassifier* and *GradientBoostingRegressor* if `init=None`, and for *HistGradientBoostingClassifier* and *HistGradientBoostingRegressor*.
  - 'brute' is used for all other estimators.

Returns

averaged_predictions [ndarray, shape (n_outputs, len(values[0]), len(values[1]), ...)] The predictions for all the points in the grid, averaged over all samples in X (or over the training data if `method` is 'recursion'). `n_outputs` corresponds to the number of classes in a multi-class setting, or to the number of tasks for multi-output regression. For classical regression and binary classification `n_outputs==1`. `n_values_feature_j` corresponds to the size `values[j]`.

values [seq of 1d ndarrays] The values with which the grid has been created. The generated grid is a cartesian product of the arrays in `values`. `len(values) == len(features)`. The size of each array `values[j]` is either `grid_resolution`, or the number of unique values in `X[:, j]`, whichever is smaller.

Warning: The 'recursion' method only works for gradient boosting estimators, and unlike the 'brute' method, it does not account for the `init` predictor of the boosting process. In practice this will produce the same values as 'brute' up to a constant offset in the target response, provided that `init` is a consant estimator (which is the default). However, as soon as `init` is not a constant estimator, the partial dependence values are incorrect for 'recursion'. This is not relevant for *HistGradientBoostingClassifier* and *HistGradientBoostingRegressor*, which do not have an `init` parameter.

See also:

sklearn.inspection.plot_partial_dependence Plot partial dependence

Examples

```
>>> X = [[0, 0, 2], [1, 0, 0]]
>>> y = [0, 1]
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> gb = GradientBoostingClassifier(random_state=0).fit(X, y)
>>> partial_dependence(gb, features=[0], X=X, percentiles=(0, 1),
...                    grid_resolution=2) # doctest: +SKIP
(array([[-4.52...,  4.52...]]), [array([ 0.,  1.])])
```

Examples using `sklearn.inspection.partial_dependence`

- *Partial Dependence Plots*

7.18.2 `sklearn.inspection`.permutation_importance

`sklearn.inspection.`**`permutation_importance`**(*estimator*, *X*, *y*, *scoring=None*, *n_repeats=5*, *n_jobs=None*, *random_state=None*)

Permutation importance for feature evaluation [BRE].

The *estimator* is required to be a fitted estimator. `X` can be the data set used to train the estimator or a hold-out set. The permutation importance of a feature is calculated as follows. First, a baseline metric, defined by *scoring*, is evaluated on a (potentially different) dataset defined by the `X`. Next, a feature column from the validation set is permuted and the metric is evaluated again. The permutation importance is defined to be the difference between the baseline metric and metric from permutating the feature column.

Read more in the *User Guide*.

Parameters

estimator [object] An estimator that has already been *fitted* and is compatible with *scorer*.

X [ndarray or DataFrame, shape (n_samples, n_features)] Data on which permutation importance will be computed.

y [array-like or None, shape (n_samples,) or (n_samples, n_classes)] Targets for supervised or `None` for unsupervised.

scoring [string, callable or None, default=None] Scorer to use. It can be a single string (see *The scoring parameter: defining model evaluation rules*) or a callable (see *Defining your scoring strategy from metric functions*). If None, the estimator's default scorer is used.

n_repeats [int, default=5] Number of times to permute a feature.

n_jobs [int or None, default=None] The number of jobs to use for the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance, or None, default=None] Pseudo-random number generator to control the permutations of each feature. See *random_state*.

Returns

result [Bunch] Dictionary-like object, with attributes:

importances_mean [ndarray, shape (n_features,)] Mean of feature importance over `n_repeats`.

importances_std [ndarray, shape (n_features,)] Standard deviation over `n_repeats`.

importances [ndarray, shape (n_features, n_repeats)] Raw permutation importance scores.

References

[BRE]

Examples using `sklearn.inspection.permutation_importance`

- *Permutation Importance with Multicollinear or Correlated Features*
- *Permutation Importance vs Random Forest Feature Importance (MDI)*
- *Release Highlights for scikit-learn 0.22*

7.18.3 Plotting

inspection.PartialDependenceDisplay(...)	Partial Dependence Plot (PDP) visualization.

sklearn.inspection.PartialDependenceDisplay

class sklearn.inspection.**PartialDependenceDisplay**(*pd_results*, *features*, *feature_names*, *target_idx*, *pdp_lim*, *deciles*)

Partial Dependence Plot (PDP) visualization.

It is recommended to use *plot_partial_dependence* to create a *PartialDependenceDisplay*. All parameters are stored as attributes.

Read more in *Advanced Plotting With Partial Dependence* and the *User Guide*.

New in version 0.22.

Parameters

pd_results [list of (ndarray, ndarray)] Results of *partial_dependence* for `features`. Each tuple corresponds to a (averaged_predictions, grid).

features [list of (int,) or list of (int, int)] Indices of features for a given plot. A tuple of one integer will plot a partial dependence curve of one feature. A tuple of two integers will plot a two-way partial dependence curve as a contour plot.

feature_names [list of str] Feature names corresponding to the indices in `features`.

target_idx [int]

- In a multiclass setting, specifies the class for which the PDPs should be computed. Note that for binary classification, the positive class (index 1) is always used.
- In a multioutput setting, specifies the task for which the PDPs should be computed.

Ignored in binary classification or classical regression settings.

pdp_lim [dict] Global min and max average predictions, such that all plots will have the same scale and y limits. `pdp_lim[1]` is the global min and max for single partial dependence curves. `pdp_lim[2]` is the global min and max for two-way partial dependence curves.

deciles [dict] Deciles for feature indices in `features`.

Attributes

bounding_ax_ [matplotlib Axes or None] If `ax` is an axes or None, the `bounding_ax_` is the axes where the grid of partial dependence plots are drawn. If `ax` is a list of axes or a numpy array of axes, `bounding_ax_` is None.

axes_ [ndarray of matplotlib Axes] If `ax` is an axes or None, `axes_[i, j]` is the axes on the i-th row and j-th column. If `ax` is a list of axes, `axes_[i]` is the i-th item in `ax`. Elements that are None corresponds to a nonexisting axes in that position.

lines_ [ndarray of matplotlib Artists] If `ax` is an axes or None, `line_[i, j]` is the partial dependence curve on the i-th row and j-th column. If `ax` is a list of axes, `lines_[i]` is the partial dependence curve corresponding to the i-th item in `ax`. Elements that are None corresponds to a nonexisting axes or an axes that does not include a line plot.

contours_ [ndarray of matplotlib Artists] If `ax` is an axes or None, `contours_[i, j]` is the partial dependence plot on the i-th row and j-th column. If `ax` is a list of axes, `contours_[i]` is the partial dependence plot corresponding to the i-th item in `ax`. Elements that are None corresponds to a nonexisting axes or an axes that does not include a contour plot.

figure_ [matplotlib Figure] Figure containing partial dependence plots.

Methods

plot(self[, ax, n_cols, line_kw, contour_kw])	Plot partial dependence plots.

__init__ (*self*, *pd_results*, *features*, *feature_names*, *target_idx*, *pdp_lim*, *deciles*)
Initialize self. See help(type(self)) for accurate signature.

plot (*self*, *ax=None*, *n_cols=3*, *line_kw=None*, *contour_kw=None*)
Plot partial dependence plots.

Parameters

ax [Matplotlib axes or array-like of Matplotlib axes, default=None]

- **If a single axis is passed in, it is treated as a bounding axes** and a grid of partial dependence plots will be drawn within these bounds. The `n_cols` parameter controls the number of columns in the grid.
- **If an array-like of axes are passed in, the partial dependence** plots will be drawn directly into these axes.
- **If `None`, a figure and a bounding axes is created and treated** as the single axes case.

n_cols [int, default=3] The maximum number of columns in the grid plot. Only active when `ax` is a single axes or `None`.

line_kw [dict, default=None] Dict with keywords passed to the `matplotlib.pyplot.plot` call. For one-way partial dependence plots.

contour_kw [dict, default=None] Dict with keywords passed to the `matplotlib.pyplot.contourf` call for two-way partial dependence plots.

Returns

display: ***PartialDependenceDisplay***

inspection.plot_partial_dependence(...[, ...])	Partial dependence plots.

sklearn.inspection.plot_partial_dependence

sklearn.inspection.**plot_partial_dependence**(*estimator*, *X*, *features*, *feature_names=None*, *target=None*, *response_method='auto'*, *n_cols=3*, *grid_resolution=100*, *percentiles=(0.05*, *0.95)*, *method='auto'*, *n_jobs=None*, *verbose=0*, *fig=None*, *line_kw=None*, *contour_kw=None*, *ax=None*)

Partial dependence plots.

The `len(features)` plots are arranged in a grid with `n_cols` columns. Two-way partial dependence plots are plotted as contour plots. The deciles of the feature values will be shown with tick marks on the x-axes for one-way plots, and on both axes for two-way plots.

Note: *plot_partial_dependence* does not support using the same axes with multiple calls. To plot the the partial dependence for multiple estimators, please pass the axes created by the first call to the second call:

```
>>> from sklearn.inspection import plot_partial_dependence
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.linear_model import LinearRegression
>>> X, y = make_friedman1()
>>> est = LinearRegression().fit(X, y)
>>> disp1 = plot_partial_dependence(est, X)
>>> disp2 = plot_partial_dependence(est, X,
...                                 ax=disp1.axes_)
```

Read more in the *User Guide*.

Parameters

estimator [BaseEstimator] A fitted estimator object implementing *predict*, *predict_proba*, or *decision_function*. Multioutput-multiclass classifiers are not supported.

X [{array-like or dataframe} of shape (n_samples, n_features)] The data to use to build the grid of values on which the dependence will be evaluated. This is usually the training data.

features [list of {int, str, pair of int, pair of str}] The target features for which to create the PDPs. If features[i] is an int or a string, a one-way PDP is created; if features[i] is a tuple, a two-way PDP is created. Each tuple must be of size 2. if any entry is a string, then it must be in `feature_names`.

feature_names [array-like of shape (n_features,), dtype=str, default=None] Name of each feature; feature_names[i] holds the name of the feature with index i. By default, the name of the feature corresponds to their numerical index for NumPy array and their column name for pandas dataframe.

target [int, optional (default=None)]

- In a multiclass setting, specifies the class for which the PDPs should be computed. Note that for binary classification, the positive class (index 1) is always used.
- In a multioutput setting, specifies the task for which the PDPs should be computed.

Ignored in binary classification or classical regression settings.

response_method ['auto', 'predict_proba' or 'decision_function', optional (default='auto')] Specifies whether to use *predict_proba* or *decision_function* as the target response. For regressors this parameter is ignored and the response is always the output of *predict*. By default, *predict_proba* is tried first and we revert to *decision_function* if it doesn't exist. If `method` is 'recursion', the response is always the output of *decision_function*.

n_cols [int, optional (default=3)] The maximum number of columns in the grid plot. Only active when `ax` is a single axis or `None`.

grid_resolution [int, optional (default=100)] The number of equally spaced points on the axes of the plots, for each target feature.

percentiles [tuple of float, optional (default=(0.05, 0.95))] The lower and upper percentile used to create the extreme values for the PDP axes. Must be in [0, 1].

method [str, optional (default='auto')] The method to use to calculate the partial dependence predictions:

- 'recursion' is only supported for gradient boosting estimator (namely *GradientBoostingClassifier*, *GradientBoostingRegressor*, *HistGradientBoostingClassifier*, *HistGradientBoostingRegressor*) but is more efficient in terms of speed. With

this method, `X` is optional and is only used to build the grid and the partial dependences are computed using the training data. This method does not account for the `init` predictor of the boosting process, which may lead to incorrect values (see warning below. With this method, the target response of a classifier is always the decision function, not the predicted probabilities.

- 'brute' is supported for any estimator, but is more computationally intensive.
- 'auto': - 'recursion' is used for estimators that supports it. - 'brute' is used for all other estimators.

n_jobs [int, optional (default=None)] The number of CPUs to use to compute the partial dependences. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [int, optional (default=0)] Verbose output during PD computations.

fig [Matplotlib figure object, optional (default=None)] A figure object onto which the plots will be drawn, after the figure has been cleared. By default, a new one is created.

Deprecated since version 0.22: `fig` will be removed in 0.24.

line_kw [dict, optional] Dict with keywords passed to the `matplotlib.pyplot.plot` call. For one-way partial dependence plots.

contour_kw [dict, optional] Dict with keywords passed to the `matplotlib.pyplot.contourf` call. For two-way partial dependence plots.

ax [Matplotlib axes or array-like of Matplotlib axes, default=None]

- **If a single axis is passed in, it is treated as a bounding axes** and a grid of partial dependence plots will be drawn within these bounds. The `n_cols` parameter controls the number of columns in the grid.
- **If an array-like of axes are passed in, the partial dependence** plots will be drawn directly into these axes.
- **If `None`, a figure and a bounding axes is created and treated** as the single axes case.

New in version 0.22.

Returns

display: ***PartialDependenceDisplay***

Warning: The 'recursion' method only works for gradient boosting estimators, and unlike the 'brute' method, it does not account for the `init` predictor of the boosting process. In practice this will produce the same values as 'brute' up to a constant offset in the target response, provided that `init` is a consant estimator (which is the default). However, as soon as `init` is not a constant estimator, the partial dependence values are incorrect for 'recursion'. This is not relevant for *`HistGradientBoostingClassifier`* and *`HistGradientBoostingRegressor`*, which do not have an `init` parameter.

See also:

`sklearn.inspection.partial_dependence` Return raw partial dependence values

Examples

```
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> X, y = make_friedman1()
>>> clf = GradientBoostingRegressor(n_estimators=10).fit(X, y)
>>> plot_partial_dependence(clf, X, [0, (0, 1)]) #doctest: +SKIP
```

Examples using `sklearn.inspection.plot_partial_dependence`

- *Advanced Plotting With Partial Dependence*
- *Partial Dependence Plots*

7.19 `sklearn.isotonic`: Isotonic regression

User guide: See the *Isotonic regression* section for further details.

isotonic.IsotonicRegression([y_min, y_max, ...])	Isotonic regression model.

7.19.1 `sklearn.isotonic`.IsotonicRegression

class sklearn.isotonic.**IsotonicRegression**(*y_min=None*, *y_max=None*, *increasing=True*, *out_of_bounds='nan'*)

Isotonic regression model.

The isotonic regression optimization problem is defined by:

```
min sum w_i (y[i] - y_[i]) ** 2

subject to y_[i] <= y_[j] whenever X[i] <= X[j]
and min(y_) = y_min, max(y_) = y_max
```

where:

- `y[i]` are inputs (real numbers)
- `y_[i]` are fitted
- `X` specifies the order. If `X` is non-decreasing then `y_` is non-decreasing.
- `w[i]` are optional strictly positive weights (default to 1.0)

Read more in the *User Guide*.

New in version 0.13.

Parameters

y_min [optional, default: None] If not None, set the lowest value of the fit to y_min.

y_max [optional, default: None] If not None, set the highest value of the fit to y_max.

increasing [boolean or string, optional, default: True] If boolean, whether or not to fit the isotonic regression with y increasing or decreasing.

The string value "auto" determines whether y should increase or decrease based on the Spearman correlation estimate's sign.

out_of_bounds [string, optional, default: "nan"] The `out_of_bounds` parameter handles how x-values outside of the training domain are handled. When set to "nan", predicted y-values will be NaN. When set to "clip", predicted y-values will be set to the value corresponding to the nearest train interval endpoint. When set to "raise", allow `interp1d` to throw ValueError.

Attributes

X_min_ [float] Minimum value of input array `X_` for left bound.

X_max_ [float] Maximum value of input array `X_` for right bound.

f_ [function] The stepwise interpolating function that covers the input domain `X`.

Notes

Ties are broken using the secondary method from Leeuw, 1977.

References

Isotonic Median Regression: A Linear Programming Approach Nilotpal Chakravarti Mathematics of Operations Research Vol. 14, No. 2 (May, 1989), pp. 303-308

Isotone Optimization in R : Pool-Adjacent-Violators Algorithm (PAVA) and Active Set Methods Leeuw, Hornik, Mair Journal of Statistical Software 2009

Correctness of Kruskal's algorithms for monotone regression with ties Leeuw, Psychometrica, 1977

Examples

```
>>> from sklearn.datasets import make_regression
>>> from sklearn.isotonic import IsotonicRegression
>>> X, y = make_regression(n_samples=10, n_features=1, random_state=41)
>>> iso_reg = IsotonicRegression().fit(X.flatten(), y)
>>> iso_reg.predict([.1, .2])
array([1.8628..., 3.7256...])
```

Methods

fit(self, X, y[, sample_weight])	Fit the model using X, y as training data.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, T)	Predict new data by linear interpolation.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, T)	Transform new data by linear interpolation

__init__(*self*, *y_min=None*, *y_max=None*, *increasing=True*, *out_of_bounds='nan'*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit the model using X, y as training data.

Parameters

X [array-like of shape (n_samples,)] Training data.

y [array-like of shape (n_samples,)] Training target.

sample_weight [array-like of shape (n_samples,), default=None] Weights. If set to None, all weights will be set to 1 (equal weights).

Returns

self [object] Returns an instance of self.

Notes

X is stored for future use, as *transform* needs X to interpolate new input data.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *T*)
Predict new data by linear interpolation.

Parameters

T [array-like of shape (n_samples,)] Data to transform.

Returns

T_ [array, shape=(n_samples,)] Transformed data.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *T*)
Transform new data by linear interpolation

Parameters

T [array-like of shape (n_samples,)] Data to transform.

Returns

T_ [array, shape=(n_samples,)] The transformed data

Examples using `sklearn.isotonic.IsotonicRegression`

- *Isotonic Regression*

isotonic.check_increasing(x, y)	Determine whether y is monotonically correlated with x.
isotonic.isotonic_regression(y[, ...])	Solve the isotonic regression model.

7.19.2 sklearn.isotonic.check_increasing

sklearn.isotonic.**check_increasing**(*x*, *y*)

Determine whether y is monotonically correlated with x.

y is found increasing or decreasing with respect to x based on a Spearman correlation test.

Parameters

x [array-like of shape (n_samples,)] Training data.

y [array-like of shape (n_samples,)] Training target.

Returns

increasing_bool [boolean] Whether the relationship is increasing or decreasing.

Notes

The Spearman correlation coefficient is estimated from the data, and the sign of the resulting estimate is used as the result.

In the event that the 95% confidence interval based on Fisher transform spans zero, a warning is raised.

References

Fisher transformation. Wikipedia. https://en.wikipedia.org/wiki/Fisher_transformation

7.19.3 sklearn.isotonic.isotonic_regression

sklearn.isotonic.**isotonic_regression**(*y*, *sample_weight=None*, *y_min=None*, *y_max=None*, *increasing=True*)

Solve the isotonic regression model:

```
min sum w[i] (y[i] - y_[i]) ** 2

subject to y_min = y_[1] <= y_[2] ... <= y_[n] = y_max
```

where:

- y[i] are inputs (real numbers)
- y_[i] are fitted
- w[i] are optional strictly positive weights (default to 1.0)

Read more in the *User Guide*.

Parameters

y [iterable of floats] The data.

sample_weight [iterable of floats, optional, default: None] Weights on each point of the regression. If None, weight is set to 1 (equal weights).

y_min [optional, default: None] If not None, set the lowest value of the fit to y_min.

y_max [optional, default: None] If not None, set the highest value of the fit to y_max.

increasing [boolean, optional, default: True] Whether to compute `y_` is increasing (if set to True) or decreasing (if set to False)

Returns

y_ [list of floats] Isotonic fit of y.

References

"Active set algorithms for isotonic regression; A unifying framework" by Michael J. Best and Nilotpal Chakravarti, section 3.

7.20 `sklearn.kernel_approximation` Kernel Approximation

The *`sklearn.kernel_approximation`* module implements several approximate kernel feature maps base on Fourier transforms.

User guide: See the *Kernel Approximation* section for further details.

`kernel_approximation.AdditiveChi2Sampler`([...])	Approximate feature map for additive chi2 kernel.
`kernel_approximation.Nystroem`([kernel, ...])	Approximate a kernel map using a subset of the training data.
`kernel_approximation.RBFSampler`([gamma, ...])	Approximates feature map of an RBF kernel by Monte Carlo approximation of its Fourier transform.
`kernel_approximation.SkewedChi2Sampler`([...])	Approximates feature map of the "skewed chi-squared" kernel by Monte Carlo approximation of its Fourier transform.

7.20.1 `sklearn.kernel_approximation`.AdditiveChi2Sampler

class sklearn.kernel_approximation.**AdditiveChi2Sampler**(*sample_steps=2*, *sample_interval=None*)

Approximate feature map for additive chi2 kernel.

Uses sampling the fourier transform of the kernel characteristic at regular intervals.

Since the kernel that is to be approximated is additive, the components of the input vectors can be treated separately. Each entry in the original space is transformed into 2*sample_steps+1 features, where sample_steps is a parameter of the method. Typical values of sample_steps include 1, 2 and 3.

Optimal choices for the sampling interval for certain data ranges can be computed (see the reference). The default values should be reasonable.

Read more in the *User Guide*.

Parameters

sample_steps [int, optional] Gives the number of (complex) sampling points.

sample_interval [float, optional] Sampling interval. Must be specified when sample_steps not in {1,2,3}.

Attributes

sample_interval_ [float] Stored sampling interval. Specified as a parameter if sample_steps not in {1,2,3}.

See also:

SkewedChi2Sampler A Fourier-approximation to a non-additive variant of the chi squared kernel.

sklearn.metrics.pairwise.chi2_kernel The exact chi squared kernel.

sklearn.metrics.pairwise.additive_chi2_kernel The exact additive chi squared kernel.

Notes

This estimator approximates a slightly different version of the additive chi squared kernel then `metric.additive_chi2` computes.

References

See "Efficient additive kernels via explicit feature maps" A. Vedaldi and A. Zisserman, Pattern Analysis and Machine Intelligence, 2011

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.linear_model import SGDClassifier
>>> from sklearn.kernel_approximation import AdditiveChi2Sampler
>>> X, y = load_digits(return_X_y=True)
>>> chi2sampler = AdditiveChi2Sampler(sample_steps=2)
>>> X_transformed = chi2sampler.fit_transform(X, y)
>>> clf = SGDClassifier(max_iter=5, random_state=0, tol=1e-3)
>>> clf.fit(X_transformed, y)
SGDClassifier(max_iter=5, random_state=0)
>>> clf.score(X_transformed, y)
0.9499...
```

Methods

fit(self, X[, y])	Set the parameters
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply approximate feature map to X.

__init__ (*self*, *sample_steps=2*, *sample_interval=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*)
Set the parameters

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples in the number of samples and n_features is the number of features.

Returns

self [object] Returns the transformer.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)

Apply approximate feature map to X.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)]

Returns

X_new [{array, sparse matrix}, shape = (n_samples, n_features * (2*sample_steps + 1))] Whether the return value is an array of sparse matrix depends on the type of the input X.

7.20.2 sklearn.kernel_approximation.Nystroem

class sklearn.kernel_approximation.**Nystroem**(*kernel='rbf'*, *gamma=None*, *coef0=None*, *degree=None*, *kernel_params=None*, *n_components=100*, *random_state=None*)

Approximate a kernel map using a subset of the training data.

Constructs an approximate feature map for an arbitrary kernel using a subset of the data as basis.

Read more in the *User Guide*.

New in version 0.13.

Parameters

kernel [string or callable, default="rbf"] Kernel map to be approximated. A callable should accept two arguments and the keyword arguments passed to this object as kernel_params, and should return a floating point number.

gamma [float, default=None] Gamma parameter for the RBF, laplacian, polynomial, exponential chi2 and sigmoid kernels. Interpretation of the default value is left to the kernel; see the documentation for sklearn.metrics.pairwise. Ignored by other kernels.

coef0 [float, default=None] Zero coefficient for polynomial and sigmoid kernels. Ignored by other kernels.

degree [float, default=None] Degree of the polynomial kernel. Ignored by other kernels.

kernel_params [mapping of string to any, optional] Additional parameters (keyword arguments) for kernel function passed as callable object.

n_components [int] Number of features to construct. How many data points will be used to construct the mapping.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Attributes

components_ [array, shape (n_components, n_features)] Subset of training points used to construct the feature map.

component_indices_ [array, shape (n_components)] Indices of `components_` in the training set.

normalization_ [array, shape (n_components, n_components)] Normalization matrix needed for embedding. Square root of the kernel matrix on `components_`.

See also:

RBFSampler An approximation to the RBF kernel using random Fourier features.

sklearn.metrics.pairwise.kernel_metrics List of built-in kernels.

References

- Williams, C.K.I. and Seeger, M. "Using the Nystroem method to speed up kernel machines", Advances in neural information processing systems 2001

- T. Yang, Y. Li, M. Mahdavi, R. Jin and Z. Zhou "Nystroem Method vs Random Fourier Features: A Theoretical and Empirical Comparison", Advances in Neural Information Processing Systems 2012

Examples

```
>>> from sklearn import datasets, svm
>>> from sklearn.kernel_approximation import Nystroem
>>> X, y = datasets.load_digits(n_class=9, return_X_y=True)
>>> data = X / 16.
>>> clf = svm.LinearSVC()
>>> feature_map_nystroem = Nystroem(gamma=.2,
...                                 random_state=1,
...                                 n_components=300)
>>> data_transformed = feature_map_nystroem.fit_transform(data)
>>> clf.fit(data_transformed, y)
LinearSVC()
>>> clf.score(data_transformed, y)
0.9987...
```

Methods

fit(self, X[, y])	Fit estimator to data.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply feature map to X.

__init__(*self*, *kernel='rbf'*, *gamma=None*, *coef0=None*, *degree=None*, *kernel_params=None*, *n_components=100*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit estimator to data.

Samples a subset of training points, computes kernel on these and computes normalization matrix.

Parameters

X [array-like of shape (n_samples, n_features)] Training data.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Apply feature map to X.

Computes an approximate feature map using the kernel between some training points and X.

Parameters

X [array-like of shape (n_samples, n_features)] Data to transform.

Returns

X_transformed [array, shape=(n_samples, n_components)] Transformed data.

Examples using `sklearn.kernel_approximation.Nystroem`

- *Explicit feature map approximation for RBF kernels*

7.20.3 `sklearn.kernel_approximation`.RBFSampler

class sklearn.kernel_approximation.**RBFSampler**(*gamma=1.0*, *n_components=100*, *random_state=None*)
Approximates feature map of an RBF kernel by Monte Carlo approximation of its Fourier transform.

It implements a variant of Random Kitchen Sinks.[1]

Read more in the *User Guide*.

Parameters

gamma [float] Parameter of RBF kernel: exp(-gamma * x^2)

n_components [int] Number of Monte Carlo samples per original feature. Equals the dimensionality of the computed feature space.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Notes

See "Random Features for Large-Scale Kernel Machines" by A. Rahimi and Benjamin Recht.

[1] "Weighted Sums of Random Kitchen Sinks: Replacing minimization with randomization in learning" by A. Rahimi and Benjamin Recht. (https://people.eecs.berkeley.edu/~brecht/papers/08.rah.rec.nips.pdf)

Examples

```
>>> from sklearn.kernel_approximation import RBFSampler
>>> from sklearn.linear_model import SGDClassifier
>>> X = [[0, 0], [1, 1], [1, 0], [0, 1]]
>>> y = [0, 0, 1, 1]
>>> rbf_feature = RBFSampler(gamma=1, random_state=1)
>>> X_features = rbf_feature.fit_transform(X)
>>> clf = SGDClassifier(max_iter=5, tol=1e-3)
>>> clf.fit(X_features, y)
SGDClassifier(max_iter=5)
>>> clf.score(X_features, y)
1.0
```

Methods

fit(self, X[, y])	Fit the model with X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply the approximate feature map to X.

__init__ (*self*, *gamma=1.0*, *n_components=100*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*)
Fit the model with X.

Samples random projection according to n_features.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data, where n_samples in the number of samples and n_features is the number of features.

Returns

self [object] Returns the transformer.

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Apply the approximate feature map to X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] New data, where n_samples in the number of samples and n_features is the number of features.

Returns

X_new [array-like, shape (n_samples, n_components)]

Examples using `sklearn.kernel_approximation.RBFSampler`

- *Explicit feature map approximation for RBF kernels*

7.20.4 `sklearn.kernel_approximation`.SkewedChi2Sampler

class sklearn.kernel_approximation.**SkewedChi2Sampler**(*skewedness=1.0*, *n_components=100*, *random_state=None*)
Approximates feature map of the "skewed chi-squared" kernel by Monte Carlo approximation of its Fourier transform.

Read more in the *User Guide*.

Parameters

skewedness [float] "skewedness" parameter of the kernel. Needs to be cross-validated.

n_components [int] number of Monte Carlo samples per original feature. Equals the dimensionality of the computed feature space.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

See also:

AdditiveChi2Sampler A different approach for approximating an additive variant of the chi squared kernel.

sklearn.metrics.pairwise.chi2_kernel The exact chi squared kernel.

References

See "Random Fourier Approximations for Skewed Multiplicative Histogram Kernels" by Fuxin Li, Catalin Ionescu and Cristian Sminchisescu.

Examples

```
>>> from sklearn.kernel_approximation import SkewedChi2Sampler
>>> from sklearn.linear_model import SGDClassifier
>>> X = [[0, 0], [1, 1], [1, 0], [0, 1]]
>>> y = [0, 0, 1, 1]
>>> chi2_feature = SkewedChi2Sampler(skewedness=.01,
...                                  n_components=10,
...                                  random_state=0)
>>> X_features = chi2_feature.fit_transform(X, y)
>>> clf = SGDClassifier(max_iter=10, tol=1e-3)
>>> clf.fit(X_features, y)
SGDClassifier(max_iter=10)
>>> clf.score(X_features, y)
1.0
```

Methods

fit(self, X[, y])	Fit the model with X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply the approximate feature map to X.

__init__ (*self*, *skewedness=1.0*, *n_components=100*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*)
Fit the model with X.

Samples random projection according to n_features.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples in the number of samples and n_features is the number of features.

Returns

self [object] Returns the transformer.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Apply the approximate feature map to X.

Parameters

X [array-like, shape (n_samples, n_features)] New data, where n_samples in the number of samples and n_features is the number of features. All values of X must be strictly greater than "-skewedness".

Returns

X_new [array-like, shape (n_samples, n_components)]

7.21 sklearn.kernel_ridge Kernel Ridge Regression

Module *sklearn.kernel_ridge* implements kernel ridge regression.

User guide: See the *Kernel ridge regression* section for further details.

kernel_ridge.KernelRidge([alpha, ...])	Kernel ridge regression.

7.21.1 sklearn.kernel_ridge.KernelRidge

class sklearn.kernel_ridge.**KernelRidge**(*alpha=1*, *kernel='linear'*, *gamma=None*, *degree=3*, *coef0=1*, *kernel_params=None*)

Kernel ridge regression.

Kernel ridge regression (KRR) combines ridge regression (linear least squares with l2-norm regularization) with the kernel trick. It thus learns a linear function in the space induced by the respective kernel and the data. For non-linear kernels, this corresponds to a non-linear function in the original space.

The form of the model learned by KRR is identical to support vector regression (SVR). However, different loss functions are used: KRR uses squared error loss while support vector regression uses epsilon-insensitive loss, both combined with l2 regularization. In contrast to SVR, fitting a KRR model can be done in closed-form and is typically faster for medium-sized datasets. On the other hand, the learned model is non-sparse and thus slower than SVR, which learns a sparse model for epsilon > 0, at prediction-time.

This estimator has built-in support for multi-variate regression (i.e., when y is a 2d-array of shape [n_samples, n_targets]).

Read more in the *User Guide*.

Parameters

alpha [{float, array-like}, shape = [n_targets]] Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to `(2*C)^-1` in other linear models such as LogisticRegression or LinearSVC. If an array is passed, penalties are assumed to be specific to the targets. Hence they must correspond in number.

kernel [string or callable, default="linear"] Kernel mapping used internally. A callable should accept two arguments and the keyword arguments passed to this object as kernel_params, and should return a floating point number. Set to "precomputed" in order to pass a precomputed kernel matrix to the estimator methods instead of samples.

gamma [float, default=None] Gamma parameter for the RBF, laplacian, polynomial, exponential chi2 and sigmoid kernels. Interpretation of the default value is left to the kernel; see the documentation for sklearn.metrics.pairwise. Ignored by other kernels.

degree [float, default=3] Degree of the polynomial kernel. Ignored by other kernels.

coef0 [float, default=1] Zero coefficient for polynomial and sigmoid kernels. Ignored by other kernels.

kernel_params [mapping of string to any, optional] Additional parameters (keyword arguments) for kernel function passed as callable object.

Attributes

dual_coef_ [array, shape = [n_samples] or [n_samples, n_targets]] Representation of weight vector(s) in kernel space

X_fit_ [{array-like, sparse matrix} of shape (n_samples, n_features)] Training data, which is also required for prediction. If kernel == "precomputed" this is instead the precomputed training matrix, shape = [n_samples, n_samples].

See also:

sklearn.linear_model.Ridge Linear ridge regression.

sklearn.svm.SVR Support Vector Regression implemented using libsvm.

References

- Kevin P. Murphy "Machine Learning: A Probabilistic Perspective", The MIT Press chapter 14.4.3, pp. 492-493

Examples

```
>>> from sklearn.kernel_ridge import KernelRidge
>>> import numpy as np
>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
>>> y = rng.randn(n_samples)
>>> X = rng.randn(n_samples, n_features)
>>> clf = KernelRidge(alpha=1.0)
>>> clf.fit(X, y)
KernelRidge(alpha=1.0)
```

Methods

fit(self, X[, y, sample_weight])	Fit Kernel Ridge regression model
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the kernel ridge model
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *alpha=1*, *kernel='linear'*, *gamma=None*, *degree=3*, *coef0=1*, *kernel_params=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*, *sample_weight=None*)
Fit Kernel Ridge regression model

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training data. If kernel == "precomputed" this is instead a precomputed kernel matrix, shape = [n_samples, n_samples].

y [array-like of shape (n_samples,) or (n_samples, n_targets)] Target values

sample_weight [float or array-like of shape [n_samples]] Individual weights for each sample, ignored if None is passed.

Returns

self [returns an instance of self.]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict using the kernel ridge model

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Samples. If kernel == "precomputed" this is instead a precomputed kernel matrix, shape = [n_samples, n_samples_fitted], where n_samples_fitted is the number of samples used in the fitting for this estimator.

Returns

C [ndarray of shape (n_samples,) or (n_samples, n_targets)] Returns predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.kernel_ridge.KernelRidge`

- *Comparison of kernel ridge regression and SVR*
- *Comparison of kernel ridge and Gaussian process regression*

7.22 `sklearn.linear_model`: Linear Models

The *`sklearn.linear_model`* module implements a variety of linear models.

User guide: See the *Linear Models* section for further details.

The following subsections are only rough guidelines: the same estimator can fall into multiple categories, depending on its parameters.

7.22.1 Linear classifiers

`linear_model.LogisticRegression`([penalty, ...])	Logistic Regression (aka logit, MaxEnt) classifier.
`linear_model.LogisticRegressionCV`([Cs, ...])	Logistic Regression CV (aka logit, MaxEnt) classifier.
`linear_model.PassiveAggressiveClassifier`([C, ...])	Passive Aggressive Classifier
`linear_model.Perceptron`([penalty, alpha, ...])	Read more in the *User Guide*.
`linear_model.RidgeClassifier`([alpha, ...])	Classifier using Ridge regression.
`linear_model.RidgeClassifierCV`([alphas, ...])	Ridge classifier with built-in cross-validation.
`linear_model.SGDClassifier`([loss, penalty, ...])	Linear classifiers (SVM, logistic regression, a.o.) with SGD training.

`sklearn.linear_model.LogisticRegression`

class sklearn.linear_model.**LogisticRegression**(*penalty='l2'*, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *random_state=None*, *solver='lbfgs'*, *max_iter=100*, *multi_class='auto'*, *verbose=0*, *warm_start=False*, *n_jobs=None*, *l1_ratio=None*)

Logistic Regression (aka logit, MaxEnt) classifier.

In the multiclass case, the training algorithm uses the one-vs-rest (OvR) scheme if the 'multi_class' option is set to 'ovr', and uses the cross-entropy loss if the 'multi_class' option is set to 'multinomial'. (Currently the 'multinomial' option is supported only by the 'lbfgs', 'sag', 'saga' and 'newton-cg' solvers.)

This class implements regularized logistic regression using the 'liblinear' library, 'newton-cg', 'sag', 'saga' and 'lbfgs' solvers. **Note that regularization is applied by default**. It can handle both dense and sparse input. Use C-ordered arrays or CSR matrices containing 64-bit floats for optimal performance; any other input format will be converted (and copied).

The 'newton-cg', 'sag', and 'lbfgs' solvers support only L2 regularization with primal formulation, or no regularization. The 'liblinear' solver supports both L1 and L2 regularization, with a dual formulation only for the L2 penalty. The Elastic-Net regularization is only supported by the 'saga' solver.

Read more in the *User Guide*.

Parameters

penalty [{'l1', 'l2', 'elasticnet', 'none'}, default='l2'] Used to specify the norm used in the penalization. The 'newton-cg', 'sag' and 'lbfgs' solvers support only l2 penalties. 'elasticnet' is only supported by the 'saga' solver. If 'none' (not supported by the liblinear solver), no regularization is applied.

New in version 0.19: l1 penalty with SAGA solver (allowing 'multinomial' + L1)

dual [bool, default=False] Dual or primal formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer dual=False when n_samples > n_features.

tol [float, default=1e-4] Tolerance for stopping criteria.

C [float, default=1.0] Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.

fit_intercept [bool, default=True] Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.

intercept_scaling [float, default=1] Useful only when the solver 'liblinear' is used and self.fit_intercept is set to True. In this case, x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equal to intercept_scaling is appended to the instance vector. The intercept becomes `intercept_scaling * synthetic_feature_weight`.

Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.

class_weight [dict or 'balanced', default=None] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

New in version 0.17: *class_weight='balanced'*

random_state [int, RandomState instance, default=None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'sag' or 'liblinear'.

solver [{'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}, default='lbfgs'] Algorithm to use in the optimization problem.

- For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones.
- For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs' handle multinomial loss; 'liblinear' is limited to one-versus-rest schemes.
- 'newton-cg', 'lbfgs', 'sag' and 'saga' handle L2 or no penalty
- 'liblinear' and 'saga' also handle L1 penalty
- 'saga' also supports 'elasticnet' penalty
- 'liblinear' does not support setting `penalty='none'`

Note that 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from sklearn.preprocessing.

New in version 0.17: Stochastic Average Gradient descent solver.

New in version 0.19: SAGA solver.

Changed in version 0.22: The default solver changed from 'liblinear' to 'lbfgs' in 0.22.

max_iter [int, default=100] Maximum number of iterations taken for the solvers to converge.

multi_class [{'auto', 'ovr', 'multinomial'}, default='auto'] If the option chosen is 'ovr', then a binary problem is fit for each label. For 'multinomial' the loss minimised is the multinomial loss fit across the entire probability distribution, *even when the data is binary*. 'multinomial' is unavailable when solver='liblinear'. 'auto' selects 'ovr' if the data is binary, or if solver='liblinear', and otherwise selects 'multinomial'.

New in version 0.18: Stochastic Average Gradient descent solver for 'multinomial' case.

Changed in version 0.22: Default changed from 'ovr' to 'auto' in 0.22.

verbose [int, default=0] For the liblinear and lbfgs solvers set verbose to any positive number for verbosity.

warm_start [bool, default=False] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. Useless for liblinear solver. See *the Glossary*.

New in version 0.17: *warm_start* to support *lbfgs*, *newton-cg*, *sag*, *saga* solvers.

n_jobs [int, default=None] Number of CPU cores used when parallelizing over classes if multi_class='ovr'". This parameter is ignored when the `solver` is set to 'liblinear' regardless of whether 'multi_class' is specified or not. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

l1_ratio [float, default=None] The Elastic-Net mixing parameter, with `0 <= l1_ratio <= 1`. Only used if `penalty='elasticnet'`. Setting ``l1_ratio=0`` is equivalent to using `penalty='l2'`, while setting `l1_ratio=1` is equivalent to using `penalty='l1'`. For `0 < l1_ratio <1`, the penalty is a combination of L1 and L2.

Attributes

classes_ [ndarray of shape (n_classes,)] A list of class labels known to the classifier.

coef_ [ndarray of shape (1, n_features) or (n_classes, n_features)] Coefficient of the features in the decision function.

`coef_` is of shape (1, n_features) when the given problem is binary. In particular, when `multi_class='multinomial'`, `coef_` corresponds to outcome 1 (True) and `-coef_` corresponds to outcome 0 (False).

intercept_ [ndarray of shape (1,) or (n_classes,)] Intercept (a.k.a. bias) added to the decision function.

If `fit_intercept` is set to False, the intercept is set to zero. `intercept_` is of shape (1,) when the given problem is binary. In particular, when `multi_class='multinomial'`, `intercept_` corresponds to outcome 1 (True) and `-intercept_` corresponds to outcome 0 (False).

n_iter_ [ndarray of shape (n_classes,) or (1,)] Actual number of iterations for all classes. If binary or multinomial, it returns only 1 element. For liblinear solver, only the maximum number of iteration across all classes is given.

Changed in version 0.20: In SciPy <= 1.0.0 the number of lbfgs iterations may exceed `max_iter`. `n_iter_` will now report at most `max_iter`.

See also:

SGDClassifier Incrementally trained logistic regression (when given the parameter `loss="log"`).

LogisticRegressionCV Logistic regression with built-in cross validation.

Notes

The underlying C implementation uses a random number generator to select features when fitting the model. It is thus not uncommon, to have slightly different results for the same input data. If that happens, try with a smaller tol parameter.

Predict output may not match that of standalone liblinear in certain cases. See *differences from liblinear* in the narrative documentation.

References

L-BFGS-B – Software for Large-scale Bound-constrained Optimization Ciyou Zhu, Richard Byrd, Jorge Nocedal and Jose Luis Morales. http://users.iems.northwestern.edu/~nocedal/lbfgsb.html

LIBLINEAR – A Library for Large Linear Classification https://www.csie.ntu.edu.tw/~cjlin/liblinear/

SAG – Mark Schmidt, Nicolas Le Roux, and Francis Bach Minimizing Finite Sums with the Stochastic Average Gradient https://hal.inria.fr/hal-00860051/document

SAGA – Defazio, A., Bach F. & Lacoste-Julien S. (2014). SAGA: A Fast Incremental Gradient Method With Support for Non-Strongly Convex Composite Objectives https://arxiv.org/abs/1407.0202

Hsiang-Fu Yu, Fang-Lan Huang, Chih-Jen Lin (2011). Dual coordinate descent methods for logistic regression and maximum entropy models. Machine Learning 85(1-2):41-75. https://www.csie.ntu.edu.tw/~cjlin/papers/maxent_dual.pdf

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import LogisticRegression
>>> X, y = load_iris(return_X_y=True)
>>> clf = LogisticRegression(random_state=0).fit(X, y)
```

(continues on next page)

(continued from previous page)

```
>>> clf.predict(X[:2, :])
array([0, 0])
>>> clf.predict_proba(X[:2, :])
array([[9.8...e-01, 1.8...e-02, 1.4...e-08],
       [9.7...e-01, 2.8...e-02, ...e-08]])
>>> clf.score(X, y)
0.97...
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
densify(self)	Convert coefficient matrix to dense array format.
fit(self, X, y[, sample_weight])	Fit the model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class labels for samples in X.
predict_log_proba(self, X)	Predict logarithm of probability estimates.
predict_proba(self, X)	Probability estimates.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.
sparsify(self)	Convert coefficient matrix to sparse format.

__init__(*self*, *penalty='l2'*, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *random_state=None*, *solver='lbfgs'*, *max_iter=100*, *multi_class='auto'*, *verbose=0*, *warm_start=False*, *n_jobs=None*, *l1_ratio=None*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

densify(*self*)
Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns

self Fitted estimator.

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit the model according to the given training data.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y [array-like of shape (n_samples,)] Target vector relative to X.

sample_weight [array-like of shape (n_samples,) default=None] Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight.

New in version 0.17: *sample_weight* support to LogisticRegression.

Returns

self Fitted estimator.

Notes

The SAGA solver supports both float64 and float32 bit arrays.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

predict_log_proba(*self*, *X*)
Predict logarithm of probability estimates.

The returned estimates for all classes are ordered by the label of classes.

Parameters

X [array-like of shape (n_samples, n_features)] Vector to be scored, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns

T [array-like of shape (n_samples, n_classes)] Returns the log-probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

predict_proba(*self*, *X*)
Probability estimates.

The returned estimates for all classes are ordered by the label of classes.

For a multi_class problem, if multi_class is set to be "multinomial" the softmax function is used to find the predicted probability of each class. Else use a one-vs-rest approach, i.e calculate the probability of each class assuming it to be positive using the logistic function. and normalize these values across all the classes.

Parameters

X [array-like of shape (n_samples, n_features)] Vector to be scored, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns

T [array-like of shape (n_samples, n_classes)] Returns the probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

sparsify (*self*)

Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual numpy.ndarray representation.

The `intercept_` member is not converted.

Returns

self Fitted estimator.

Notes

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the partial_fit method (if any) will not work until you call densify.

Examples using `sklearn.linear_model.LogisticRegression`

- *Compact estimator representations*
- *Comparison of Calibration of Classifiers*
- *Probability Calibration curves*
- *Plot classification probability*
- *Plot class probabilities calculated by the VotingClassifier*
- *Feature transformations with ensembles of trees*
- *Logistic function*
- *Regularization path of L1- Logistic Regression*
- *Logistic Regression 3-class Classifier*
- *Comparing various online solvers*
- *MNIST classification using multinomial logistic + L1*
- *Plot multinomial and One-vs-Rest Logistic Regression*
- *L1 Penalty and Sparsity in Logistic Regression*
- *Multiclass sparse logistic regression on 20newgroups*
- *Classifier Chain*
- *Restricted Boltzmann Machine features for digit classification*
- *Pipelining: chaining a PCA and a logistic regression*
- *Column Transformer with Mixed Types*
- *Feature discretization*
- *Release Highlights for scikit-learn 0.22*
- *Digits Classification Exercise*

`sklearn.linear_model`.PassiveAggressiveClassifier

class sklearn.linear_model.**PassiveAggressiveClassifier**(*C=1.0*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *shuffle=True*, *verbose=0*, *loss='hinge'*, *n_jobs=None*, *random_state=None*, *warm_start=False*, *class_weight=None*, *average=False*)

Passive Aggressive Classifier

Read more in the *User Guide*.

Parameters

C [float] Maximum step size (regularization). Defaults to 1.0.

fit_intercept [bool, default=False] Whether the intercept should be estimated or not. If False, the data is assumed to be already centered.

max_iter [int, optional (default=1000)] The maximum number of passes over the training data (aka epochs). It only impacts the behavior in the `fit` method, and not the *partial_fit* method.

New in version 0.19.

tol [float or None, optional (default=1e-3)] The stopping criterion. If it is not None, the iterations will stop when (loss > previous_loss - tol).

New in version 0.19.

early_stopping [bool, default=False] Whether to use early stopping to terminate training when validation. score is not improving. If set to True, it will automatically set aside a stratified fraction of training data as validation and terminate training when validation score is not improving by at least tol for n_iter_no_change consecutive epochs.

New in version 0.20.

validation_fraction [float, default=0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.

New in version 0.20.

n_iter_no_change [int, default=5] Number of iterations with no improvement to wait before early stopping.

New in version 0.20.

shuffle [bool, default=True] Whether or not the training data should be shuffled after each epoch.

verbose [integer, optional] The verbosity level

loss [string, optional] The loss function to be used: hinge: equivalent to PA-I in the reference paper. squared_hinge: equivalent to PA-II in the reference paper.

n_jobs [int or None, optional (default=None)] The number of CPUs to use to do the OVA (One Versus All, for multi-class problems) computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional, default=None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

Repeatedly calling fit or partial_fit when warm_start is True can result in a different solution than when calling fit a single time because of the way the data is shuffled.

class_weight [dict, {class_label: weight} or "balanced" or None, optional] Preset for the class_weight fit parameter.

Weights associated with classes. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

New in version 0.17: parameter *class_weight* to automatically weight samples.

average [bool or int, optional] When set to True, computes the averaged SGD weights and stores the result in the `coef_` attribute. If set to an int greater than 1, averaging will begin once the total number of samples seen reaches average. So average=10 will begin averaging after seeing 10 samples.

New in version 0.19: parameter *average* to use weights averaging in SGD

Attributes

coef_ [array, shape = [1, n_features] if n_classes == 2 else [n_classes, n_features]] Weights assigned to the features.

intercept_ [array, shape = [1] if n_classes == 2 else [n_classes]] Constants in decision function.

n_iter_ [int] The actual number of iterations to reach the stopping criterion. For multiclass fits, it is the maximum over every binary fit.

classes_ [array of shape (n_classes,)] The unique classes labels.

t_ [int] Number of weight updates performed during training. Same as `(n_iter_ * n_samples)`.

See also:

SGDClassifier

Perceptron

References

Online Passive-Aggressive Algorithms <http://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06a.pdf> K. Crammer, O. Dekel, J. Keshat, S. Shalev-Shwartz, Y. Singer - JMLR (2006)

Examples

```
>>> from sklearn.linear_model import PassiveAggressiveClassifier
>>> from sklearn.datasets import make_classification
```

```
>>> X, y = make_classification(n_features=4, random_state=0)
>>> clf = PassiveAggressiveClassifier(max_iter=1000, random_state=0,
... tol=1e-3)
>>> clf.fit(X, y)
PassiveAggressiveClassifier(random_state=0)
>>> print(clf.coef_)
[[0.26642044 0.45070924 0.67251877 0.64185414]]
>>> print(clf.intercept_)
[1.84127814]
>>> print(clf.predict([[0, 0, 0, 0]]))
[1]
```

Methods

decision_function(self, X)	Predict confidence scores for samples.

Continued on next page

Table 156 – continued from previous page

densify(self)	Convert coefficient matrix to dense array format.
fit(self, X, y[, coef_init, intercept_init])	Fit linear model with Passive Aggressive algorithm.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes])	Fit linear model with Passive Aggressive algorithm.
predict(self, X)	Predict class labels for samples in X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **kwargs)	Set and validate the parameters of estimator.
sparsify(self)	Convert coefficient matrix to sparse format.

__init__ (*self*, *C=1.0*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *shuffle=True*, *verbose=0*, *loss='hinge'*, *n_jobs=None*, *random_state=None*, *warm_start=False*, *class_weight=None*, *average=False*)

Initialize self. See help(type(self)) for accurate signature.

decision_function (*self*, *X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

densify (*self*)

Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns

self Fitted estimator.

fit (*self*, *X*, *y*, *coef_init=None*, *intercept_init=None*)

Fit linear model with Passive Aggressive algorithm.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training data

y [numpy array of shape [n_samples]] Target values

coef_init [array, shape = [n_classes,n_features]] The initial coefficients to warm-start the optimization.

intercept_init [array, shape = [n_classes]] The initial intercept to warm-start the optimization.

Returns

self [returns an instance of self.]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit(*self*, *X*, *y*, *classes=None*)
Fit linear model with Passive Aggressive algorithm.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Subset of the training data

y [numpy array of shape [n_samples]] Subset of the target values

classes [array, shape = [n_classes]] Classes across all calls to partial_fit. Can be obtained by via `np.unique(y_all)`, where y_all is the target vector of the entire dataset. This argument is required for the first call to partial_fit and can be omitted in the subsequent calls. Note that y doesn't need to contain all labels in `classes`.

Returns

self [returns an instance of self.]

predict(*self*, *X*)
Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***kwargs*)
Set and validate the parameters of estimator.

Parameters

****kwargs** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

sparsify(*self*)

Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual numpy.ndarray representation.

The `intercept_` member is not converted.

Returns

self Fitted estimator.

Notes

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the partial_fit method (if any) will not work until you call densify.

Examples using `sklearn.linear_model.PassiveAggressiveClassifier`

- *Out-of-core classification of text documents*
- *Comparing various online solvers*
- *Classification of text documents using sparse features*

`sklearn.linear_model`.Perceptron

class sklearn.linear_model.**Perceptron**(*penalty=None*, *alpha=0.0001*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *shuffle=True*, *verbose=0*, *eta0=1.0*, *n_jobs=None*, *random_state=0*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *class_weight=None*, *warm_start=False*)

Read more in the *User Guide*.

Parameters

penalty [{'l2','l1','elasticnet'}, default=None] The penalty (aka regularization term) to be used.

alpha [float, default=0.0001] Constant that multiplies the regularization term if regularization is used.

fit_intercept [bool, default=True] Whether the intercept should be estimated or not. If False, the data is assumed to be already centered.

max_iter [int, default=1000] The maximum number of passes over the training data (aka epochs). It only impacts the behavior in the `fit` method, and not the *partial_fit* method.

New in version 0.19.

tol [float, default=1e-3] The stopping criterion. If it is not None, the iterations will stop when (loss > previous_loss - tol).

New in version 0.19.

shuffle [bool, default=True] Whether or not the training data should be shuffled after each epoch.

verbose [int, default=0] The verbosity level

eta0 [double, default=1] Constant by which the updates are multiplied.

n_jobs [int, default=None] The number of CPUs to use to do the OVA (One Versus All, for multi-class problems) computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance, default=None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

early_stopping [bool, default=False] Whether to use early stopping to terminate training when validation. score is not improving. If set to True, it will automatically set aside a stratified fraction of training data as validation and terminate training when validation score is not improving by at least tol for n_iter_no_change consecutive epochs.

New in version 0.20.

validation_fraction [float, default=0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.

New in version 0.20.

n_iter_no_change [int, default=5] Number of iterations with no improvement to wait before early stopping.

New in version 0.20.

class_weight [dict, {class_label: weight} or "balanced", default=None] Preset for the class_weight fit parameter.

Weights associated with classes. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

warm_start [bool, default=False] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

Attributes

coef_ [ndarray of shape = [1, n_features] if n_classes == 2 else [n_classes, n_features]] Weights assigned to the features.

intercept_ [ndarray of shape = [1] if n_classes == 2 else [n_classes]] Constants in decision function.

n_iter_ [int] The actual number of iterations to reach the stopping criterion. For multiclass fits, it is the maximum over every binary fit.

classes_ [ndarray of shape (n_classes,)] The unique classes labels.

t_ [int] Number of weight updates performed during training. Same as `(n_iter_ * n_samples)`.

See also:

SGDClassifier

Notes

`Perceptron` is a classification algorithm which shares the same underlying implementation with `SGDClassifier`. In fact, `Perceptron()` is equivalent to `SGDClassifier(loss="perceptron", eta0=1, learning_rate="constant", penalty=None)`.

References

https://en.wikipedia.org/wiki/Perceptron and references therein.

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.linear_model import Perceptron
>>> X, y = load_digits(return_X_y=True)
>>> clf = Perceptron(tol=1e-3, random_state=0)
>>> clf.fit(X, y)
Perceptron()
>>> clf.score(X, y)
0.939...
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
densify(self)	Convert coefficient matrix to dense array format.
fit(self, X, y[, coef_init, intercept_init, ...])	Fit linear model with Stochastic Gradient Descent.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Perform one epoch of stochastic gradient descent on given samples.
predict(self, X)	Predict class labels for samples in X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **kwargs)	Set and validate the parameters of estimator.
sparsify(self)	Convert coefficient matrix to sparse format.

__init__(*self*, *penalty=None*, *alpha=0.0001*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *shuffle=True*, *verbose=0*, *eta0=1.0*, *n_jobs=None*, *random_state=0*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *class_weight=None*, *warm_start=False*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

densify(*self*)

Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns

self Fitted estimator.

fit(*self*, *X*, *y*, *coef_init=None*, *intercept_init=None*, *sample_weight=None*)

Fit linear model with Stochastic Gradient Descent.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data.

y [ndarray of shape (n_samples,)] Target values.

coef_init [ndarray of shape (n_classes, n_features), default=None] The initial coefficients to warm-start the optimization.

intercept_init [ndarray of shape (n_classes,), default=None] The initial intercept to warm-start the optimization.

sample_weight [array-like, shape (n_samples,), default=None] Weights applied to individual samples. If not provided, uniform weights are assumed. These weights will be multiplied with class_weight (passed through the constructor) if class_weight is specified.

Returns

self : Returns an instance of self.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit(*self*, *X*, *y*, *classes=None*, *sample_weight=None*)

Perform one epoch of stochastic gradient descent on given samples.

Internally, this method uses `max_iter = 1`. Therefore, it is not guaranteed that a minimum of the cost function is reached after calling it once. Matters such as objective convergence and early stopping should be handled by the user.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Subset of the training data.

y [ndarray of shape (n_samples,)] Subset of the target values.

classes [ndarray of shape (n_classes,), default=None] Classes across all calls to partial_fit. Can be obtained by via `np.unique(y_all)`, where y_all is the target vector of the entire dataset. This argument is required for the first call to partial_fit and can be omitted in the subsequent calls. Note that y doesn't need to contain all labels in `classes`.

sample_weight [array-like, shape (n_samples,), default=None] Weights applied to individual samples. If not provided, uniform weights are assumed.

Returns

self : Returns an instance of self.

predict(*self*, *X*)

Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***kwargs*)

Set and validate the parameters of estimator.

Parameters

****kwargs** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

sparsify(*self*)

Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual numpy.ndarray representation.

The `intercept_` member is not converted.

Returns

self Fitted estimator.

Notes

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the partial_fit method (if any) will not work until you call densify.

Examples using `sklearn.linear_model.Perceptron`

- *Out-of-core classification of text documents*
- *Comparing various online solvers*
- *Classification of text documents using sparse features*

`sklearn.linear_model.`RidgeClassifier

class sklearn.linear_model.**RidgeClassifier**(*alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *class_weight=None*, *solver='auto'*, *random_state=None*)

Classifier using Ridge regression.

This classifier first converts the target values into `{-1, 1}` and then treats the problem as a regression task (multi-output regression in the multiclass case).

Read more in the *User Guide*.

Parameters

alpha [float, default=1.0] Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to `C^-1` in other linear models such as LogisticRegression or LinearSVC.

fit_intercept [bool, default=True] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize [bool, default=False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

copy_X [bool, default=True] If True, X will be copied; else, it may be overwritten.

max_iter [int, default=None] Maximum number of iterations for conjugate gradient solver. The default value is determined by scipy.sparse.linalg.

tol [float, default=1e-3] Precision of the solution.

class_weight [dict or 'balanced', default=None] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

solver [{'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga'}, default='auto'] Solver to use in the computational routines:

- 'auto' chooses the solver automatically based on the type of data.
- 'svd' uses a Singular Value Decomposition of X to compute the Ridge coefficients. More stable for singular matrices than 'cholesky'.
- 'cholesky' uses the standard scipy.linalg.solve function to obtain a closed-form solution.
- 'sparse_cg' uses the conjugate gradient solver as found in scipy.sparse.linalg.cg. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set `tol` and `max_iter`).
- 'lsqr' uses the dedicated regularized least-squares routine scipy.sparse.linalg.lsqr. It is the fastest and uses an iterative procedure.
- 'sag' uses a Stochastic Average Gradient descent, and 'saga' uses its unbiased and more flexible version named SAGA. Both methods use an iterative procedure, and are often faster than other solvers when both n_samples and n_features are large. Note that 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from sklearn.preprocessing.

  New in version 0.17: Stochastic Average Gradient descent solver.

  New in version 0.19: SAGA solver.

random_state [int, RandomState instance, default=None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'sag'.

Attributes

coef_ [ndarray of shape (1, n_features) or (n_classes, n_features)] Coefficient of the features in the decision function.

`coef_` is of shape (1, n_features) when the given problem is binary.

intercept_ [float or ndarray of shape (n_targets,)] Independent term in decision function. Set to 0.0 if `fit_intercept = False`.

n_iter_ [None or ndarray of shape (n_targets,)] Actual number of iterations for each target. Available only for sag and lsqr solvers. Other solvers will return None.

classes_ [ndarray of shape (n_classes,)] The classes labels.

See also:

Ridge Ridge regression.

RidgeClassifierCV Ridge classifier with built-in cross validation.

Notes

For multi-class classification, n_class classifiers are trained in a one-versus-all approach. Concretely, this is implemented by taking advantage of the multi-variate response support in Ridge.

Examples

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.linear_model import RidgeClassifier
>>> X, y = load_breast_cancer(return_X_y=True)
>>> clf = RidgeClassifier().fit(X, y)
>>> clf.score(X, y)
0.9595...
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
fit(self, X, y[, sample_weight])	Fit Ridge classifier model.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class labels for samples in X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *class_weight=None*, *solver='auto'*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit Ridge classifier model.

Parameters

X [{ndarray, sparse matrix} of shape (n_samples, n_features)] Training data.

y [ndarray of shape (n_samples,)] Target values.

sample_weight [float or ndarray of shape (n_samples,), default=None] Individual weights for each sample. If given a float, every sample will have the same weight.

New in version 0.17: *sample_weight* support to Classifier.

Returns

self [object] Instance of the estimator.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)

Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.linear_model.RidgeClassifier`

- *Classification of text documents using sparse features*

sklearn.linear_model.SGDClassifier

class sklearn.linear_model.**SGDClassifier**(*loss='hinge'*, *penalty='l2'*, *alpha=0.0001*, *l1_ratio=0.15*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *shuffle=True*, *verbose=0*, *epsilon=0.1*, *n_jobs=None*, *random_state=None*, *learning_rate='optimal'*, *eta0=0.0*, *power_t=0.5*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *class_weight=None*, *warm_start=False*, *average=False*)

Linear classifiers (SVM, logistic regression, a.o.) with SGD training.

This estimator implements regularized linear models with stochastic gradient descent (SGD) learning: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate). SGD allows minibatch (online/out-of-core) learning, see the partial_fit method. For best results using the default learning rate schedule, the data should have zero mean and unit variance.

This implementation works with data represented as dense or sparse arrays of floating point values for the features. The model it fits can be controlled with the loss parameter; by default, it fits a linear support vector machine (SVM).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

Read more in the *User Guide*.

Parameters

loss [str, default='hinge'] The loss function to be used. Defaults to 'hinge', which gives a linear SVM.

The possible options are 'hinge', 'log', 'modified_huber', 'squared_hinge', 'perceptron', or a regression loss: 'squared_loss', 'huber', 'epsilon_insensitive', or 'squared_epsilon_insensitive'.

The 'log' loss gives logistic regression, a probabilistic classifier. 'modified_huber' is another smooth loss that brings tolerance to outliers as well as probability estimates. 'squared_hinge' is like hinge but is quadratically penalized. 'perceptron' is the linear loss used by the perceptron algorithm. The other losses are designed for regression but can be useful in classification as well; see SGDRegressor for a description.

penalty [{'l2', 'l1', 'elasticnet'}, default='l2'] The penalty (aka regularization term) to be used. Defaults to 'l2' which is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' might bring sparsity to the model (feature selection) not achievable with 'l2'.

alpha [float, default=0.0001] Constant that multiplies the regularization term. Defaults to 0.0001. Also used to compute learning_rate when set to 'optimal'.

l1_ratio [float, default=0.15] The Elastic Net mixing parameter, with 0 <= l1_ratio <= 1. l1_ratio=0 corresponds to L2 penalty, l1_ratio=1 to L1. Defaults to 0.15.

fit_intercept [bool, default=True] Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

max_iter [int, default=1000] The maximum number of passes over the training data (aka epochs). It only impacts the behavior in the `fit` method, and not the *partial_fit* method.

New in version 0.19.

tol [float, default=1e-3] The stopping criterion. If it is not None, the iterations will stop when (loss > best_loss - tol) for `n_iter_no_change` consecutive epochs.

New in version 0.19.

shuffle [bool, default=True] Whether or not the training data should be shuffled after each epoch.

verbose [int, default=0] The verbosity level.

epsilon [float, default=0.1] Epsilon in the epsilon-insensitive loss functions; only if `loss` is 'huber', 'epsilon_insensitive', or 'squared_epsilon_insensitive'. For 'huber', determines the threshold at which it becomes less important to get the prediction exactly right. For epsilon-insensitive, any differences between the current prediction and the correct label are ignored if they are less than this threshold.

n_jobs [int, default=None] The number of CPUs to use to do the OVA (One Versus All, for multi-class problems) computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance, default=None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

learning_rate [str, default='optimal'] The learning rate schedule:

'constant': eta = eta0

'optimal': [default] eta = 1.0 / (alpha * (t + t0)) where t0 is chosen by a heuristic proposed by Leon Bottou.

'invscaling': eta = eta0 / pow(t, power_t)

'adaptive': eta = eta0, as long as the training keeps decreasing. Each time n_iter_no_change consecutive epochs fail to decrease the training loss by tol or fail to increase validation score by tol if early_stopping is True, the current learning rate is divided by 5.

eta0 [double, default=0.0] The initial learning rate for the 'constant', 'invscaling' or 'adaptive' schedules. The default value is 0.0 as eta0 is not used by the default schedule 'optimal'.

power_t [double, default=0.5] The exponent for inverse scaling learning rate [default 0.5].

early_stopping [bool, default=False] Whether to use early stopping to terminate training when validation score is not improving. If set to True, it will automatically set aside a stratified fraction of training data as validation and terminate training when validation score is not improving by at least tol for n_iter_no_change consecutive epochs.

New in version 0.20.

validation_fraction [float, default=0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.

New in version 0.20.

n_iter_no_change [int, default=5] Number of iterations with no improvement to wait before early stopping.

New in version 0.20.

class_weight [dict, {class_label: weight} or "balanced", default=None] Preset for the class_weight fit parameter.

Weights associated with classes. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

warm_start [bool, default=False] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

Repeatedly calling fit or partial_fit when warm_start is True can result in a different solution than when calling fit a single time because of the way the data is shuffled. If a dynamic learning rate is used, the learning rate is adapted depending on the number of samples already seen. Calling `fit` resets this counter, while `partial_fit` will result in increasing the existing counter.

average [bool or int, default=False] When set to True, computes the averaged SGD weights and stores the result in the `coef_` attribute. If set to an int greater than 1, averaging will begin once the total number of samples seen reaches average. So `average=10` will begin averaging after seeing 10 samples.

Attributes

coef_ [ndarray of shape (1, n_features) if n_classes == 2 else (n_classes, n_features)] Weights assigned to the features.

intercept_ [ndarray of shape (1,) if n_classes == 2 else (n_classes,)] Constants in decision function.

n_iter_ [int] The actual number of iterations to reach the stopping criterion. For multiclass fits, it is the maximum over every binary fit.

loss_function_ [concrete `LossFunction`]

classes_ [array of shape (n_classes,)]

t_ [int] Number of weight updates performed during training. Same as `(n_iter_ * n_samples)`.

See also:

sklearn.svm.LinearSVC Linear support vector classification.

LogisticRegression Logistic regression.

Perceptron Inherits from SGDClassifier. `Perceptron()` is equivalent to `SGDClassifier(loss="perceptron", eta0=1, learning_rate="constant", penalty=None)`.

Examples

```
>>> import numpy as np
>>> from sklearn import linear_model
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> Y = np.array([1, 1, 2, 2])
>>> clf = linear_model.SGDClassifier(max_iter=1000, tol=1e-3)
>>> clf.fit(X, Y)
SGDClassifier()
```

```
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
densify(self)	Convert coefficient matrix to dense array format.
fit(self, X, y[, coef_init, intercept_init, ...])	Fit linear model with Stochastic Gradient Descent.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Perform one epoch of stochastic gradient descent on given samples.
predict(self, X)	Predict class labels for samples in X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **kwargs)	Set and validate the parameters of estimator.
sparsify(self)	Convert coefficient matrix to sparse format.

__init__(*self*, *loss='hinge'*, *penalty='l2'*, *alpha=0.0001*, *l1_ratio=0.15*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *shuffle=True*, *verbose=0*, *epsilon=0.1*, *n_jobs=None*, *random_state=None*, *learning_rate='optimal'*, *eta0=0.0*, *power_t=0.5*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *class_weight=None*, *warm_start=False*, *average=False*)

Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

densify(*self*)

Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns

self Fitted estimator.

fit(*self*, *X*, *y*, *coef_init=None*, *intercept_init=None*, *sample_weight=None*)

Fit linear model with Stochastic Gradient Descent.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data.

y [ndarray of shape (n_samples,)] Target values.

coef_init [ndarray of shape (n_classes, n_features), default=None] The initial coefficients to warm-start the optimization.

intercept_init [ndarray of shape (n_classes,), default=None] The initial intercept to warm-start the optimization.

sample_weight [array-like, shape (n_samples,), default=None] Weights applied to individual samples. If not provided, uniform weights are assumed. These weights will be multiplied with class_weight (passed through the constructor) if class_weight is specified.

Returns

self : Returns an instance of self.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit(*self*, *X*, *y*, *classes=None*, *sample_weight=None*)
Perform one epoch of stochastic gradient descent on given samples.

Internally, this method uses `max_iter = 1`. Therefore, it is not guaranteed that a minimum of the cost function is reached after calling it once. Matters such as objective convergence and early stopping should be handled by the user.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Subset of the training data.

y [ndarray of shape (n_samples,)] Subset of the target values.

classes [ndarray of shape (n_classes,), default=None] Classes across all calls to partial_fit. Can be obtained by via `np.unique(y_all)`, where y_all is the target vector of the entire dataset. This argument is required for the first call to partial_fit and can be omitted in the subsequent calls. Note that y doesn't need to contain all labels in `classes`.

sample_weight [array-like, shape (n_samples,), default=None] Weights applied to individual samples. If not provided, uniform weights are assumed.

Returns

self : Returns an instance of self.

predict(*self*, *X*)
Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

property predict_log_proba
Log of probability estimates.

This method is only available for log loss and modified Huber loss.

When loss="modified_huber", probability estimates may be hard zeros and ones, so taking the logarithm is not possible.

See `predict_proba` for details.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Input data for prediction.

Returns

T [array-like, shape (n_samples, n_classes)] Returns the log-probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

property predict_proba

Probability estimates.

This method is only available for log loss and modified Huber loss.

Multiclass probability estimates are derived from binary (one-vs.-rest) estimates by simple normalization, as recommended by Zadrozny and Elkan.

Binary probability estimates for loss="modified_huber" are given by (clip(decision_function(X), -1, 1) + 1) / 2. For other loss functions it is necessary to perform proper probability calibration by wrapping the classifier with *sklearn.calibration.CalibratedClassifierCV* instead.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Input data for prediction.

Returns

ndarray of shape (n_samples, n_classes) Returns the probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

References

Zadrozny and Elkan, "Transforming classifier scores into multiclass probability estimates", SIGKDD'02, http://www.research.ibm.com/people/z/zadrozny/kdd2002-Transf.pdf

The justification for the formula in the loss="modified_huber" case is in the appendix B in: http://jmlr.csail.mit.edu/papers/volume2/zhang02c/zhang02c.pdf

score(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***kwargs*)

Set and validate the parameters of estimator.

Parameters

****kwargs** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

sparsify(*self*)

Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual numpy.ndarray representation.

The `intercept_` member is not converted.

Returns

self Fitted estimator.

Notes

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the partial_fit method (if any) will not work until you call densify.

Examples using `sklearn.linear_model.SGDClassifier`

- *Sample pipeline for text feature extraction and evaluation*

7.22.2 Classical linear regressors

linear_model.LinearRegression([...])	Ordinary least squares Linear Regression.
linear_model.Ridge([alpha, fit_intercept, ...])	Linear least squares with l2 regularization.
linear_model.RidgeCV([alphas, ...])	Ridge regression with built-in cross-validation.
linear_model.SGDRegressor([loss, penalty, ...])	Linear model fitted by minimizing a regularized empirical loss with SGD

sklearn.linear_model.LinearRegression

class sklearn.linear_model.**LinearRegression**(*fit_intercept=True*, *normalize=False*, *copy_X=True*, *n_jobs=None*)

Ordinary least squares Linear Regression.

LinearRegression fits a linear model with coefficients w = (w1, ..., wp) to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation.

Parameters

fit_intercept [bool, optional, default True] Whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [bool, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use

sklearn.preprocessing.StandardScaler before calling `fit` on an estimator with `normalize=False`.

copy_X [bool, optional, default True] If True, X will be copied; else, it may be overwritten.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This will only provide speedup for n_targets > 1 and sufficient large problems. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

coef_ [array of shape (n_features,) or (n_targets, n_features)] Estimated coefficients for the linear regression problem. If multiple targets are passed during the fit (y 2D), this is a 2D array of shape (n_targets, n_features), while if only one target is passed, this is a 1D array of length n_features.

rank_ [int] Rank of matrix `X`. Only available when `X` is dense.

singular_ [array of shape (min(X, y),)] Singular values of `X`. Only available when `X` is dense.

intercept_ [float or array of shape of (n_targets,)] Independent term in the linear model. Set to 0.0 if `fit_intercept = False`.

See also:

sklearn.linear_model.Ridge Ridge regression addresses some of the problems of Ordinary Least Squares by imposing a penalty on the size of the coefficients with l2 regularization.

sklearn.linear_model.Lasso The Lasso is a linear model that estimates sparse coefficients with l1 regularization.

sklearn.linear_model.ElasticNet Elastic-Net is a linear regression model trained with both l1 and l2 -norm regularization of the coefficients.

Notes

From the implementation point of view, this is just plain Ordinary Least Squares (scipy.linalg.lstsq) wrapped as a predictor object.

Examples

```
>>> import numpy as np
>>> from sklearn.linear_model import LinearRegression
>>> X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
>>> # y = 1 * x_0 + 2 * x_1 + 3
>>> y = np.dot(X, np.array([1, 2])) + 3
>>> reg = LinearRegression().fit(X, y)
>>> reg.score(X, y)
1.0
>>> reg.coef_
array([1., 2.])
>>> reg.intercept_
3.0000...
>>> reg.predict(np.array([[3, 5]]))
array([16.])
```

Methods

fit(self, X, y[, sample_weight])	Fit linear model.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit linear model.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training data

y [array-like of shape (n_samples,) or (n_samples, n_targets)] Target values. Will be cast to X's dtype if necessary

sample_weight [array-like of shape (n_samples,), default=None] Individual weights for each sample

New in version 0.17: parameter *sample_weight* support to LinearRegression.

Returns

self [returns an instance of self.]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *`r2_score`*. This will influence the `score` method of all the multioutput regressors (except for *`MultiOutputRegressor`*). To specify the default value manually and avoid the warning, please either call *`r2_score`* directly or make a custom scorer with *`make_scorer`* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.linear_model.LinearRegression`

- *Isotonic Regression*
- *Face completion with a multi-output estimators*
- *Plot individual and voting regression predictions*
- *Ordinary Least Squares and Ridge Regression Variance*
- *Logistic function*
- *Linear Regression Example*
- *Robust linear model estimation using RANSAC*
- *Sparsity Example: Fitting only features 1 and 2*
- *Theil-Sen Regression*
- *Robust linear estimator fitting*
- *Automatic Relevance Determination Regression (ARD)*
- *Bayesian Ridge Regression*
- *Plotting Cross-Validated Predictions*

- *Underfitting vs. Overfitting*
- *Using KBinsDiscretizer to discretize continuous features*

sklearn.linear_model.Ridge

class sklearn.linear_model.**Ridge**(*alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *solver='auto'*, *random_state=None*)

Linear least squares with l2 regularization.

Minimizes the objective function:

```
||y - Xw||^2_2 + alpha * ||w||^2_2
```

This model solves a regression model where the loss function is the linear least squares function and regularization is given by the l2-norm. Also known as Ridge Regression or Tikhonov regularization. This estimator has built-in support for multi-variate regression (i.e., when y is a 2d-array of shape (n_samples, n_targets)).

Read more in the *User Guide*.

Parameters

alpha [{float, ndarray of shape (n_targets,)}, default=1.0] Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to `C^-1` in other linear models such as LogisticRegression or LinearSVC. If an array is passed, penalties are assumed to be specific to the targets. Hence they must correspond in number.

fit_intercept [bool, default=True] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [bool, default=False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

copy_X [bool, default=True] If True, X will be copied; else, it may be overwritten.

max_iter [int, default=None] Maximum number of iterations for conjugate gradient solver. For 'sparse_cg' and 'lsqr' solvers, the default value is determined by scipy.sparse.linalg. For 'sag' solver, the default value is 1000.

tol [float, default=1e-3] Precision of the solution.

solver [{'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga'}, default='auto'] Solver to use in the computational routines:

- 'auto' chooses the solver automatically based on the type of data.
- 'svd' uses a Singular Value Decomposition of X to compute the Ridge coefficients. More stable for singular matrices than 'cholesky'.
- 'cholesky' uses the standard scipy.linalg.solve function to obtain a closed-form solution.
- 'sparse_cg' uses the conjugate gradient solver as found in scipy.sparse.linalg.cg. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set `tol` and `max_iter`).

- 'lsqr' uses the dedicated regularized least-squares routine scipy.sparse.linalg.lsqr. It is the fastest and uses an iterative procedure.
- 'sag' uses a Stochastic Average Gradient descent, and 'saga' uses its improved, unbiased version named SAGA. Both methods also use an iterative procedure, and are often faster than other solvers when both n_samples and n_features are large. Note that 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from sklearn.preprocessing.

All last five solvers support both dense and sparse data. However, only 'sparse_cg' supports sparse input when `fit_intercept` is True.

New in version 0.17: Stochastic Average Gradient descent solver.

New in version 0.19: SAGA solver.

random_state [int, RandomState instance, default=None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'sag'.

New in version 0.17: *random_state* to support Stochastic Average Gradient.

Attributes

coef_ [ndarray of shape (n_features,) or (n_targets, n_features)] Weight vector(s).

intercept_ [float or ndarray of shape (n_targets,)] Independent term in decision function. Set to 0.0 if `fit_intercept = False`.

n_iter_ [None or ndarray of shape (n_targets,)] Actual number of iterations for each target. Available only for sag and lsqr solvers. Other solvers will return None.

New in version 0.17.

See also:

RidgeClassifier Ridge classifier

RidgeCV Ridge regression with built-in cross validation

sklearn.kernel_ridge.KernelRidge Kernel ridge regression combines ridge regression with the kernel trick

Examples

```
>>> from sklearn.linear_model import Ridge
>>> import numpy as np
>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
>>> y = rng.randn(n_samples)
>>> X = rng.randn(n_samples, n_features)
>>> clf = Ridge(alpha=1.0)
>>> clf.fit(X, y)
Ridge()
```

Methods

fit(self, X, y[, sample_weight])	Fit Ridge regression model.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *solver='auto'*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit Ridge regression model.

Parameters

X [{ndarray, sparse matrix} of shape (n_samples, n_features)] Training data

y [ndarray of shape (n_samples,) or (n_samples, n_targets)] Target values

sample_weight [float or ndarray of shape (n_samples,), default=None] Individual weights for each sample. If given a float, every sample will have the same weight.

Returns

self [returns an instance of self.]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.linear_model.Ridge`

- *Compressive sensing: tomography reconstruction with L1 prior (Lasso)*
- *Prediction Latency*
- *Plot Ridge coefficients as a function of the regularization*
- *Ordinary Least Squares and Ridge Regression Variance*
- *Plot Ridge coefficients as a function of the L2 regularization*
- *Polynomial interpolation*
- *HuberRegressor vs Ridge on dataset with strong outliers*

`sklearn.linear_model`.SGDRegressor

class sklearn.linear_model.**SGDRegressor**(*loss='squared_loss'*, *penalty='l2'*, *alpha=0.0001*, *l1_ratio=0.15*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *shuffle=True*, *verbose=0*, *epsilon=0.1*, *random_state=None*, *learning_rate='invscaling'*, *eta0=0.01*, *power_t=0.25*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *warm_start=False*, *average=False*)

Linear model fitted by minimizing a regularized empirical loss with SGD

SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

This implementation works with data represented as dense numpy arrays of floating point values for the features.

Read more in the *User Guide*.

Parameters

loss [str, default='squared_loss'] The loss function to be used. The possible values are 'squared_loss', 'huber', 'epsilon_insensitive', or 'squared_epsilon_insensitive'

The 'squared_loss' refers to the ordinary least squares fit. 'huber' modifies 'squared_loss' to focus less on getting outliers correct by switching from squared to linear loss past a distance of epsilon. 'epsilon_insensitive' ignores errors less than epsilon and is linear past that; this is the loss function used in SVR. 'squared_epsilon_insensitive' is the same but becomes squared loss past a tolerance of epsilon.

penalty [{'l2', 'l1', 'elasticnet'}, default='l2'] The penalty (aka regularization term) to be used. Defaults to 'l2' which is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' might bring sparsity to the model (feature selection) not achievable with 'l2'.

alpha [float, default=0.0001] Constant that multiplies the regularization term. Also used to compute learning_rate when set to 'optimal'.

l1_ratio [float, default=0.15] The Elastic Net mixing parameter, with 0 <= l1_ratio <= 1. l1_ratio=0 corresponds to L2 penalty, l1_ratio=1 to L1.

fit_intercept [bool, default=True] Whether the intercept should be estimated or not. If False, the data is assumed to be already centered.

max_iter [int, default=1000] The maximum number of passes over the training data (aka epochs). It only impacts the behavior in the `fit` method, and not the *`partial_fit`* method.

New in version 0.19.

tol [float, default=1e-3] The stopping criterion. If it is not None, the iterations will stop when (loss > best_loss - tol) for `n_iter_no_change` consecutive epochs.

New in version 0.19.

shuffle [bool, default=True] Whether or not the training data should be shuffled after each epoch.

verbose [int, default=0] The verbosity level.

epsilon [float, default=0.1] Epsilon in the epsilon-insensitive loss functions; only if `loss` is 'huber', 'epsilon_insensitive', or 'squared_epsilon_insensitive'. For 'huber', determines the threshold at which it becomes less important to get the prediction exactly right. For epsilon-insensitive, any differences between the current prediction and the correct label are ignored if they are less than this threshold.

random_state [int, RandomState instance, default=None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

learning_rate [string, default='invscaling'] The learning rate schedule:

'constant': eta = eta0

'optimal': eta = 1.0 / (alpha * (t + t0)) where t0 is chosen by a heuristic proposed by Leon Bottou.

'invscaling': [default] eta = eta0 / pow(t, power_t)

'adaptive': eta = eta0, as long as the training keeps decreasing. Each time n_iter_no_change consecutive epochs fail to decrease the training loss by tol or fail to increase validation score by tol if early_stopping is True, the current learning rate is divided by 5.

eta0 [double, default=0.01] The initial learning rate for the 'constant', 'invscaling' or 'adaptive' schedules. The default value is 0.01.

power_t [double, default=0.25] The exponent for inverse scaling learning rate.

early_stopping [bool, default=False] Whether to use early stopping to terminate training when validation score is not improving. If set to True, it will automatically set aside a fraction of training data as validation and terminate training when validation score is not improving by at least tol for n_iter_no_change consecutive epochs.

New in version 0.20.

validation_fraction [float, default=0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.

New in version 0.20.

n_iter_no_change [int, default=5] Number of iterations with no improvement to wait before early stopping.

New in version 0.20.

warm_start [bool, default=False] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

Repeatedly calling fit or partial_fit when warm_start is True can result in a different solution than when calling fit a single time because of the way the data is shuffled. If a dynamic learning rate is used, the learning rate is adapted depending on the number of samples already seen. Calling `fit` resets this counter, while `partial_fit` will result in increasing the existing counter.

average [bool or int, default=False] When set to True, computes the averaged SGD weights and stores the result in the `coef_` attribute. If set to an int greater than 1, averaging will begin once the total number of samples seen reaches average. So `average=10` will begin averaging after seeing 10 samples.

Attributes

coef_ [ndarray of shape (n_features,)] Weights assigned to the features.

intercept_ [ndarray of shape (1,)] The intercept term.

average_coef_ [ndarray of shape (n_features,)] Averaged weights assigned to the features.

average_intercept_ [ndarray of shape (1,)] The averaged intercept term.

n_iter_ [int] The actual number of iterations to reach the stopping criterion.

t_ [int] Number of weight updates performed during training. Same as `(n_iter_ * n_samples)`.

See also:

Ridge, *ElasticNet*, *Lasso*, *sklearn.svm.SVR*

Examples

```
>>> import numpy as np
>>> from sklearn import linear_model
>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
>>> y = rng.randn(n_samples)
>>> X = rng.randn(n_samples, n_features)
>>> clf = linear_model.SGDRegressor(max_iter=1000, tol=1e-3)
>>> clf.fit(X, y)
SGDRegressor()
```

Methods

densify(self)	Convert coefficient matrix to dense array format.
fit(self, X, y[, coef_init, intercept_init, ...])	Fit linear model with Stochastic Gradient Descent.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, sample_weight])	Perform one epoch of stochastic gradient descent on given samples.
predict(self, X)	Predict using the linear model
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **kwargs)	Set and validate the parameters of estimator.
sparsify(self)	Convert coefficient matrix to sparse format.

__init__(*self*, *loss='squared_loss'*, *penalty='l2'*, *alpha=0.0001*, *l1_ratio=0.15*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *shuffle=True*, *verbose=0*, *epsilon=0.1*, *random_state=None*, *learning_rate='invscaling'*, *eta0=0.01*, *power_t=0.25*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *warm_start=False*, *average=False*)
Initialize self. See help(type(self)) for accurate signature.

densify(*self*)
Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns

self Fitted estimator.

fit(*self*, *X*, *y*, *coef_init=None*, *intercept_init=None*, *sample_weight=None*)
Fit linear model with Stochastic Gradient Descent.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data

y [ndarray of shape (n_samples,)] Target values

coef_init [ndarray of shape (n_features,), default=None] The initial coefficients to warm-start the optimization.

intercept_init [ndarray of shape (1,), default=None] The initial intercept to warm-start the optimization.

sample_weight [array-like, shape (n_samples,), default=None] Weights applied to individual samples (1. for unweighted).

Returns

self [returns an instance of self.]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit(*self*, *X*, *y*, *sample_weight=None*)
Perform one epoch of stochastic gradient descent on given samples.

Internally, this method uses `max_iter = 1`. Therefore, it is not guaranteed that a minimum of the cost function is reached after calling it once. Matters such as objective convergence and early stopping should be handled by the user.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Subset of training data

y [numpy array of shape (n_samples,)] Subset of target values

sample_weight [array-like, shape (n_samples,), default=None] Weights applied to individual samples. If not provided, uniform weights are assumed.

Returns

self [returns an instance of self.]

predict(*self*, *X*)
Predict using the linear model

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)]

Returns

ndarray of shape (n_samples,) Predicted target values per element in X.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples,

n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***kwargs*)
Set and validate the parameters of estimator.

Parameters

****kwargs** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

sparsify(*self*)
Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual numpy.ndarray representation.

The `intercept_` member is not converted.

Returns

self Fitted estimator.

Notes

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the partial_fit method (if any) will not work until you call densify.

Examples using `sklearn.linear_model.SGDRegressor`

- *Prediction Latency*

7.22.3 Regressors with variable selection

The following estimators have built-in variable selection fitting procedures, but any estimator using a L1 or elastic-net penalty also performs variable selection: typically *SGDRegressor* or *SGDClassifier* with an appropriate penalty.

linear_model.ElasticNet([alpha, l1_ratio, ...])	Linear regression with combined L1 and L2 priors as regularizer.
linear_model.ElasticNetCV([l1_ratio, eps, ...])	Elastic Net model with iterative fitting along a regularization path.
linear_model.Lars([fit_intercept, verbose, ...])	Least Angle Regression model a.k.a.
linear_model.LarsCV([fit_intercept, ...])	Cross-validated Least Angle Regression model.
linear_model.Lasso([alpha, fit_intercept, ...])	Linear Model trained with L1 prior as regularizer (aka the Lasso)
linear_model.LassoCV([eps, n_alphas, ...])	Lasso linear model with iterative fitting along a regularization path.
linear_model.LassoLars([alpha, ...])	Lasso model fit with Least Angle Regression a.k.a.
linear_model.LassoLarsCV([fit_intercept, ...])	Cross-validated Lasso, using the LARS algorithm.
linear_model.LassoLarsIC([criterion, ...])	Lasso model fit with Lars using BIC or AIC for model selection
linear_model.OrthogonalMatchingPursuit([n_...])	Orthogonal Matching Pursuit model (OMP)
linear_model.OrthogonalMatchingPursuitCV([...])	Cross-validated Orthogonal Matching Pursuit model (OMP).

sklearn.linear_model.ElasticNet

class sklearn.linear_model.**ElasticNet**(*alpha=1.0*, *l1_ratio=0.5*, *fit_intercept=True*, *normalize=False*, *precompute=False*, *max_iter=1000*, *copy_X=True*, *tol=0.0001*, *warm_start=False*, *positive=False*, *random_state=None*, *selection='cyclic'*)

Linear regression with combined L1 and L2 priors as regularizer.

Minimizes the objective function:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

If you are interested in controlling the L1 and L2 penalty separately, keep in mind that this is equivalent to:

```
a * L1 + b * L2
```

where:

```
alpha = a + b and l1_ratio = a / (a + b)
```

The parameter l1_ratio corresponds to alpha in the glmnet R package while alpha corresponds to the lambda parameter in glmnet. Specifically, l1_ratio = 1 is the lasso penalty. Currently, l1_ratio <= 0.01 is not reliable, unless you supply your own sequence of alpha.

Read more in the *User Guide*.

Parameters

alpha [float, optional] Constant that multiplies the penalty terms. Defaults to 1.0. See the notes for the exact mathematical meaning of this parameter. `alpha = 0` is equivalent to an ordinary least square, solved by the *LinearRegression* object. For numerical reasons, using `alpha = 0` with the `Lasso` object is not advised. Given this, you should use the *LinearRegression* object.

l1_ratio [float] The ElasticNet mixing parameter, with `0 <= l1_ratio <= 1`. For `l1_ratio = 0` the penalty is an L2 penalty. `For l1_ratio = 1` it is an L1 penalty. For `0 < l1_ratio < 1`, the penalty is a combination of L1 and L2.

fit_intercept [bool] Whether the intercept should be estimated or not. If `False`, the data is assumed to be already centered.

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *sklearn.preprocessing.StandardScaler* before calling `fit` on an estimator with `normalize=False`.

precompute [True | False | array-like] Whether to use a precomputed Gram matrix to speed up calculations. The Gram matrix can also be passed as argument. For sparse input this option is always `True` to preserve sparsity.

max_iter [int, optional] The maximum number of iterations

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

warm_start [bool, optional] When set to `True`, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

positive [bool, optional] When set to `True`, forces the coefficients to be positive.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection` == 'random'.

selection [str, default 'cyclic'] If set to 'random', a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to 'random') often leads to significantly faster convergence especially when tol is higher than 1e-4.

Attributes

coef_ [array, shape (n_features,) | (n_targets, n_features)] parameter vector (w in the cost function formula)

sparse_coef_ [scipy.sparse matrix, shape (n_features, 1) | (n_targets, n_features)] sparse representation of the fitted `coef_`

intercept_ [float | array, shape (n_targets,)] independent term in decision function.

n_iter_ [array-like, shape (n_targets,)] number of iterations run by the coordinate descent solver to reach the specified tolerance.

See also:

ElasticNetCV Elastic net model with best model selection by cross-validation.

SGDRegressor implements elastic net regression with incremental training.

SGDClassifier implements logistic regression with elastic net penalty (`SGDClassifier(loss="log", penalty="elasticnet")`).

Notes

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn.linear_model import ElasticNet
>>> from sklearn.datasets import make_regression
```

```
>>> X, y = make_regression(n_features=2, random_state=0)
>>> regr = ElasticNet(random_state=0)
>>> regr.fit(X, y)
ElasticNet(random_state=0)
>>> print(regr.coef_)
[18.83816048 64.55968825]
>>> print(regr.intercept_)
1.451...
>>> print(regr.predict([[0, 0]]))
[1.451...]
```

Methods

fit(self, X, y[, check_input])	Fit model with coordinate descent.
get_params(self[, deep])	Get parameters for this estimator.
path(X, y[, l1_ratio, eps, n_alphas, ...])	Compute elastic net path with coordinate descent.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *l1_ratio=0.5*, *fit_intercept=True*, *normalize=False*, *precompute=False*, *max_iter=1000*, *copy_X=True*, *tol=0.0001*, *warm_start=False*, *positive=False*, *random_state=None*, *selection='cyclic'*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *check_input=True*)
Fit model with coordinate descent.

Parameters

X [ndarray or scipy.sparse matrix, (n_samples, n_features)] Data

y [ndarray, shape (n_samples,) or (n_samples, n_targets)] Target. Will be cast to X's dtype if necessary

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Notes

Coordinate descent is an algorithm that considers each column of data at a time hence it will automatically convert the X input as a Fortran-contiguous numpy array if necessary.

To avoid memory re-allocation it is advised to allocate the initial data in memory directly using that format.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

static path(*X*, *y*, *l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, *check_input=True*, ***params*)
Compute elastic net path with coordinate descent.

The elastic net optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] Target values.

l1_ratio [float, optional] Number between 0 and 1 passed to elastic net (scaling between l1 and l2 penalties). `l1_ratio=1` corresponds to the Lasso.

eps [float] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

n_alphas [int, optional] Number of alphas along the regularization path.

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [bool, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or int] Amount of verbosity.

return_n_iter [bool] Whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

check_input [bool, default True] Skip input validation checks, including the Gram matrix when provided assuming there are handled by the caller when check_input=False.

****params** [kwargs] Keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha. (Is returned when `return_n_iter` is set to True).

See also:

MultiTaskElasticNet

MultiTaskElasticNetCV

ElasticNet

ElasticNetCV

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

predict (*self*, *X*)

Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

property sparse_coef_
sparse representation of the fitted `coef_`

Examples using `sklearn.linear_model.ElasticNet`

- *Lasso and Elastic Net for Sparse Signals*
- *Train error vs Test error*

sklearn.linear_model.Lars

class sklearn.linear_model.**Lars**(*fit_intercept=True*, *verbose=False*, *normalize=True*, *precompute='auto'*, *n_nonzero_coefs=500*, *eps=2.220446049250313e-16*, *copy_X=True*, *fit_path=True*)

Least Angle Regression model a.k.a. LAR

Read more in the *User Guide*.

Parameters

fit_intercept [bool, default=True] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

verbose [bool or int, default=False] Sets the verbosity amount

normalize [bool, default=True] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

precompute [bool, 'auto' or array-like , default='auto'] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

n_nonzero_coefs [int, default=500] Target number of non-zero coefficients. Use `np.inf` for no limit.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the `tol` parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization. By default, `np.finfo(np.float).eps` is used.

copy_X [bool, default=True] If `True`, X will be copied; else, it may be overwritten.

fit_path [bool, default=True] If True the full path is stored in the `coef_path_` attribute. If you compute the solution for a large problem or many targets, setting `fit_path` to `False` will lead to a speedup, especially with a small alpha.

Attributes

alphas_ [array-like of shape (n_alphas + 1,) | list of n_targets such arrays] Maximum of covariances (in absolute value) at each iteration. `n_alphas` is either `n_nonzero_coefs` or `n_features`, whichever is smaller.

active_ [list, length = n_alphas | list of n_targets such lists] Indices of active variables at the end of the path.

coef_path_ [array-like of shape (n_features, n_alphas + 1) | list of n_targets such arrays] The varying values of the coefficients along the path. It is not present if the `fit_path` parameter is `False`.

coef_ [array-like of shape (n_features,) or (n_targets, n_features)] Parameter vector (w in the formulation formula).

intercept_ [float or array-like of shape (n_targets,)] Independent term in decision function.

n_iter_ [array-like or int] The number of iterations taken by lars_path to find the grid of alphas for each target.

See also:

lars_path, *LarsCV*

sklearn.decomposition.sparse_encode

Examples

```
>>> from sklearn import linear_model
>>> reg = linear_model.Lars(n_nonzero_coefs=1)
>>> reg.fit([[-1, 1], [0, 0], [1, 1]], [-1.1111, 0, -1.1111])
Lars(n_nonzero_coefs=1)
>>> print(reg.coef_)
[ 0. -1.11...]
```

Methods

fit(self, X, y[, Xy])	Fit the model using X, y as training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *fit_intercept=True*, *verbose=False*, *normalize=True*, *precompute='auto'*, *n_nonzero_coefs=500*, *eps=2.220446049250313e-16*, *copy_X=True*, *fit_path=True*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *Xy=None*)
Fit the model using X, y as training data.

Parameters

X [array-like of shape (n_samples, n_features)] Training data.

y [array-like of shape (n_samples,) or (n_samples, n_targets)] Target values.

Xy [array-like of shape (n_samples,) or (n_samples, n_targets), default=None] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

Returns

self [object] returns an instance of self.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

sklearn.linear_model.Lasso

class sklearn.linear_model.**Lasso**(*alpha=1.0*, *fit_intercept=True*, *normalize=False*, *precompute=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *positive=False*, *random_state=None*, *selection='cyclic'*)

Linear Model trained with L1 prior as regularizer (aka the Lasso)

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Technically the Lasso model is optimizing the same objective function as the Elastic Net with `l1_ratio=1.0` (no L2 penalty).

Read more in the *User Guide*.

Parameters

alpha [float, optional] Constant that multiplies the L1 term. Defaults to 1.0. `alpha = 0` is equivalent to an ordinary least square, solved by the *`LinearRegression`* object. For numerical reasons, using `alpha = 0` with the `Lasso` object is not advised. Given this, you should use the *`LinearRegression`* object.

fit_intercept [boolean, optional, default True] Whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

precompute [True | False | array-like, default=False] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument. For sparse input this option is always `True` to preserve sparsity.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

positive [bool, optional] When set to `True`, forces the coefficients to be positive.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection` == 'random'.

selection [str, default 'cyclic'] If set to 'random', a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to 'random') often leads to significantly faster convergence especially when tol is higher than 1e-4.

Attributes

coef_ [array, shape (n_features,) | (n_targets, n_features)] parameter vector (w in the cost function formula)

sparse_coef_ [scipy.sparse matrix, shape (n_features, 1) | (n_targets, n_features)] sparse representation of the fitted `coef_`

intercept_ [float | array, shape (n_targets,)] independent term in decision function.

n_iter_ [int | array-like, shape (n_targets,)] number of iterations run by the coordinate descent solver to reach the specified tolerance.

See also:

lars_path

lasso_path

LassoLars

LassoCV

LassoLarsCV

sklearn.decomposition.sparse_encode

Notes

The algorithm used to fit the model is coordinate descent.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.Lasso(alpha=0.1)
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
Lasso(alpha=0.1)
>>> print(clf.coef_)
[0.85 0.  ]
>>> print(clf.intercept_)
0.15...
```

Methods

fit(self, X, y[, check_input])	Fit model with coordinate descent.
get_params(self[, deep])	Get parameters for this estimator.
path(X, y[, l1_ratio, eps, n_alphas, ...])	Compute elastic net path with coordinate descent.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *alpha=1.0*, *fit_intercept=True*, *normalize=False*, *precompute=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *positive=False*, *random_state=None*, *selection='cyclic'*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*, *check_input=True*)
Fit model with coordinate descent.

Parameters

X [ndarray or scipy.sparse matrix, (n_samples, n_features)] Data

y [ndarray, shape (n_samples,) or (n_samples, n_targets)] Target. Will be cast to X's dtype if necessary

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Notes

Coordinate descent is an algorithm that considers each column of data at a time hence it will automatically convert the X input as a Fortran-contiguous numpy array if necessary.

To avoid memory re-allocation it is advised to allocate the initial data in memory directly using that format.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

static path(*X*, *y*, *l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, *check_input=True*, ***params*)
Compute elastic net path with coordinate descent.

The elastic net optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] Target values.

l1_ratio [float, optional] Number between 0 and 1 passed to elastic net (scaling between l1 and l2 penalties). `l1_ratio=1` corresponds to the Lasso.

eps [float] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

n_alphas [int, optional] Number of alphas along the regularization path.

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [bool, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or int] Amount of verbosity.

return_n_iter [bool] Whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

check_input [bool, default True] Skip input validation checks, including the Gram matrix when provided assuming there are handled by the caller when check_input=False.

****params** [kwargs] Keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha. (Is returned when `return_n_iter` is set to True).

See also:

MultiTaskElasticNet

MultiTaskElasticNetCV

ElasticNet

ElasticNetCV

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

predict (*self*, *X*)
Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

property sparse_coef_

sparse representation of the fitted `coef_`

Examples using `sklearn.linear_model.Lasso`

- *Compressive sensing: tomography reconstruction with L1 prior (Lasso)*
- *Lasso on dense and sparse data*
- *Joint feature selection with multi-task Lasso*
- *Lasso and Elastic Net for Sparse Signals*
- *Cross-validation on diabetes Dataset Exercise*

sklearn.linear_model.LassoLars

class sklearn.linear_model.**LassoLars**(*alpha=1.0*, *fit_intercept=True*, *verbose=False*, *normalize=True*, *precompute='auto'*, *max_iter=500*, *eps=2.220446049250313e-16*, *copy_X=True*, *fit_path=True*, *positive=False*)

Lasso model fit with Least Angle Regression a.k.a. Lars

It is a Linear Model trained with an L1 prior as regularizer.

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Read more in the *User Guide*.

Parameters

alpha [float, default=1.0] Constant that multiplies the penalty term. Defaults to 1.0. `alpha = 0` is equivalent to an ordinary least square, solved by *`LinearRegression`*. For numerical reasons, using `alpha = 0` with the LassoLars object is not advised and you should prefer the LinearRegression object.

fit_intercept [bool, default=True] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

verbose [bool or int, default=False] Sets the verbosity amount

normalize [bool, default=True] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

precompute [bool, 'auto' or array-like, default='auto'] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

max_iter [int, default=500] Maximum number of iterations to perform.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the `tol` parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization. By default, `np.finfo(np.float).eps` is used.

copy_X [bool, default=True] If True, X will be copied; else, it may be overwritten.

fit_path [bool, default=True] If `True` the full path is stored in the `coef_path_` attribute. If you compute the solution for a large problem or many targets, setting `fit_path` to `False` will lead to a speedup, especially with a small alpha.

positive [bool, default=False] Restrict coefficients to be >= 0. Be aware that you might want to remove fit_intercept which is set True by default. Under the positive restriction the model coefficients will not converge to the ordinary-least-squares solution for small values of alpha. Only coefficients up to the smallest alpha value (`alphas_[alphas_ > 0.].min()` when fit_path=True) reached by the stepwise Lars-Lasso algorithm are typically in congruence with the solution of the coordinate descent Lasso estimator.

Attributes

alphas_ [array-like of shape (n_alphas + 1,) | list of n_targets such arrays] Maximum of covariances (in absolute value) at each iteration. `n_alphas` is either `max_iter`,

`n_features`, or the number of nodes in the path with correlation greater than `alpha`, whichever is smaller.

active_ [list, length = n_alphas | list of n_targets such lists] Indices of active variables at the end of the path.

coef_path_ [array-like of shape (n_features, n_alphas + 1) or list] If a list is passed it's expected to be one of n_targets such arrays. The varying values of the coefficients along the path. It is not present if the `fit_path` parameter is `False`.

coef_ [array-like of shape (n_features,) or (n_targets, n_features)] Parameter vector (w in the formulation formula).

intercept_ [float or array-like of shape (n_targets,)] Independent term in decision function.

n_iter_ [array-like or int.] The number of iterations taken by lars_path to find the grid of alphas for each target.

See also:

lars_path

lasso_path

Lasso

LassoCV

LassoLarsCV

LassoLarsIC

sklearn.decomposition.sparse_encode

Examples

```
>>> from sklearn import linear_model
>>> reg = linear_model.LassoLars(alpha=0.01)
>>> reg.fit([[-1, 1], [0, 0], [1, 1]], [-1, 0, -1])
LassoLars(alpha=0.01)
>>> print(reg.coef_)
[ 0.         -0.963257...]
```

Methods

fit(self, X, y[, Xy])	Fit the model using X, y as training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *fit_intercept=True*, *verbose=False*, *normalize=True*, *precompute='auto'*, *max_iter=500*, *eps=2.220446049250313e-16*, *copy_X=True*, *fit_path=True*, *positive=False*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *Xy=None*)

Fit the model using X, y as training data.

Parameters

X [array-like of shape (n_samples, n_features)] Training data.

y [array-like of shape (n_samples,) or (n_samples, n_targets)] Target values.

Xy [array-like of shape (n_samples,) or (n_samples, n_targets), default=None] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

Returns

self [object] returns an instance of self.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

sklearn.linear_model.OrthogonalMatchingPursuit

class sklearn.linear_model.**OrthogonalMatchingPursuit**(*n_nonzero_coefs=None*, *tol=None*, *fit_intercept=True*, *normalize=True*, *precompute='auto'*)

Orthogonal Matching Pursuit model (OMP)

Read more in the *User Guide*.

Parameters

n_nonzero_coefs [int, optional] Desired number of non-zero entries in the solution. If None (by default) this value is set to 10% of n_features.

tol [float, optional] Maximum norm of the residual. If not None, overrides n_nonzero_coefs.

fit_intercept [boolean, optional] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [boolean, optional, default True] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *sklearn.preprocessing.StandardScaler* before calling `fit` on an estimator with `normalize=False`.

precompute [{True, False, 'auto'}, default 'auto'] Whether to use a precomputed Gram and Xy matrix to speed up calculations. Improves performance when *n_targets* or *n_samples* is very large. Note that if you already have such matrices, you can pass them directly to the fit method.

Attributes

coef_ [array, shape (n_features,) or (n_targets, n_features)] parameter vector (w in the formula)

intercept_ [float or array, shape (n_targets,)] independent term in decision function.

n_iter_ [int or array-like] Number of active features across every target.

See also:

orthogonal_mp

orthogonal_mp_gram

lars_path

Lars

LassoLars

decomposition.sparse_encode

OrthogonalMatchingPursuitCV

Notes

Orthogonal matching pursuit was introduced in G. Mallat, Z. Zhang, Matching pursuits with time-frequency dictionaries, IEEE Transactions on Signal Processing, Vol. 41, No. 12. (December 1993), pp. 3397-3415. (http://blanche.polytechnique.fr/~mallat/papiers/MallatPursuit93.pdf)

This implementation is based on Rubinstein, R., Zibulevsky, M. and Elad, M., Efficient Implementation of the K-SVD Algorithm using Batch Orthogonal Matching Pursuit Technical Report - CS Technion, April 2008. https://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf

Examples

```
>>> from sklearn.linear_model import OrthogonalMatchingPursuit
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(noise=4, random_state=0)
>>> reg = OrthogonalMatchingPursuit().fit(X, y)
>>> reg.score(X, y)
0.9991...
>>> reg.predict(X[:1,])
array([-78.3854...])
```

Methods

fit(self, X, y)	Fit the model using X, y as training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_nonzero_coefs=None*, *tol=None*, *fit_intercept=True*, *normalize=True*, *precompute='auto'*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)
Fit the model using X, y as training data.

Parameters

X [array-like, shape (n_samples, n_features)] Training data.

y [array-like, shape (n_samples,) or (n_samples, n_targets)] Target values. Will be cast to X's dtype if necessary

Returns

self [object] returns an instance of self.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.linear_model.OrthogonalMatchingPursuit`

- *Orthogonal Matching Pursuit*

7.22.4 Bayesian regressors

linear_model.ARDRegression([n_iter, ...])	Bayesian ARD regression.
linear_model.BayesianRidge([n_iter, ...])	Bayesian ridge regression.

sklearn.linear_model.ARDRegression

class sklearn.linear_model.**ARDRegression**(*n_iter=300*, *tol=0.001*, *alpha_1=1e-06*, *alpha_2=1e-06*, *lambda_1=1e-06*, *lambda_2=1e-06*, *compute_score=False*, *threshold_lambda=10000.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *verbose=False*)

Bayesian ARD regression.

Fit the weights of a regression model, using an ARD prior. The weights of the regression model are assumed to be in Gaussian distributions. Also estimate the parameters lambda (precisions of the distributions of the weights) and alpha (precision of the distribution of the noise). The estimation is done by an iterative procedures (Evidence Maximization)

Read more in the *User Guide*.

Parameters

n_iter [int, default=300] Maximum number of iterations.

tol [float, default=1e-3] Stop the algorithm if w has converged.

alpha_1 [float, default=1e-6] Hyper-parameter : shape parameter for the Gamma distribution prior over the alpha parameter.

alpha_2 [float, default=1e-6] Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the alpha parameter.

lambda_1 [float, default=1e-6] Hyper-parameter : shape parameter for the Gamma distribution prior over the lambda parameter.

lambda_2 [float, default=1e-6] Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the lambda parameter.

compute_score [bool, default=False] If True, compute the objective function at each step of the model.

threshold_lambda [float, default=10 000] threshold for removing (pruning) weights with high precision from the computation.

fit_intercept [bool, default=True] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [bool, default=False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

copy_X [bool, default=True] If True, X will be copied; else, it may be overwritten.

verbose [bool, default=False] Verbose mode when fitting the model.

Attributes

coef_ [array-like of shape (n_features,)] Coefficients of the regression model (mean of distribution)

alpha_ [float] estimated precision of the noise.

lambda_ [array-like of shape (n_features,)] estimated precisions of the weights.

sigma_ [array-like of shape (n_features, n_features)] estimated variance-covariance matrix of the weights

scores_ [float] if computed, value of the objective function (to be maximized)

intercept_ [float] Independent term in decision function. Set to 0.0 if `fit_intercept = False`.

Notes

For an example, see *examples/linear_model/plot_ard.py*.

References

D. J. C. MacKay, Bayesian nonlinear modeling for the prediction competition, ASHRAE Transactions, 1994.

R. Salakhutdinov, Lecture notes on Statistical Machine Learning, http://www.utstat.toronto.edu/~rsalakhu/sta4273/notes/Lecture2.pdf#page=15 Their beta is our `self.alpha_` Their alpha is our `self.lambda_` ARD is a little different than the slide: only dimensions/features for which `self.lambda_ < self.threshold_lambda` are kept and the rest are discarded.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.ARDRegression()
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
ARDRegression()
>>> clf.predict([[1, 1]])
array([1.])
```

Methods

fit(self, X, y)	Fit the ARDRegression model according to the given training data and parameters.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, return_std])	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_iter=300*, *tol=0.001*, *alpha_1=1e-06*, *alpha_2=1e-06*, *lambda_1=1e-06*, *lambda_2=1e-06*, *compute_score=False*, *threshold_lambda=10000.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *verbose=False*)

Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)

Fit the ARDRegression model according to the given training data and parameters.

Iterative procedure to maximize the evidence

Parameters

X [array-like of shape (n_samples, n_features)] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [array-like of shape (n_samples,)] Target values (integers). Will be cast to X's dtype if necessary

Returns

self [returns an instance of self.]

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*, *return_std=False*)

Predict using the linear model.

In addition to the mean of the predictive distribution, also its standard deviation can be returned.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Samples.

return_std [bool, default=False] Whether to return the standard deviation of posterior prediction.

Returns

y_mean [array-like of shape (n_samples,)] Mean of predictive distribution of query points.

y_std [array-like of shape (n_samples,)] Standard deviation of predictive distribution of query points.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.linear_model.ARDRegression`

- *Automatic Relevance Determination Regression (ARD)*

sklearn.linear_model.BayesianRidge

class sklearn.linear_model.**BayesianRidge**(*n_iter=300*, *tol=0.001*, *alpha_1=1e-06*, *alpha_2=1e-06*, *lambda_1=1e-06*, *lambda_2=1e-06*, *alpha_init=None*, *lambda_init=None*, *compute_score=False*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *verbose=False*)

Bayesian ridge regression.

Fit a Bayesian ridge model. See the Notes section for details on this implementation and the optimization of the regularization parameters lambda (precision of the weights) and alpha (precision of the noise).

Read more in the *User Guide*.

Parameters

n_iter [int, default=300] Maximum number of iterations. Should be greater than or equal to 1.

tol [float, default=1e-3] Stop the algorithm if w has converged.

alpha_1 [float, default=1e-6] Hyper-parameter : shape parameter for the Gamma distribution prior over the alpha parameter.

alpha_2 [float, default=1e-6] Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the alpha parameter.

lambda_1 [float, default=1e-6] Hyper-parameter : shape parameter for the Gamma distribution prior over the lambda parameter.

lambda_2 [float, default=1e-6] Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the lambda parameter.

alpha_init [float, default=None] Initial value for alpha (precision of the noise). If not set, alpha_init is 1/Var(y).

New in version 0.22.

lambda_init [float, default=None] Initial value for lambda (precision of the weights). If not set, lambda_init is 1.

New in version 0.22.

compute_score [bool, default=False] If True, compute the log marginal likelihood at each iteration of the optimization.

fit_intercept [bool, default=True] Whether to calculate the intercept for this model. The intercept is not treated as a probabilistic parameter and thus has no associated variance. If set to False, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [bool, default=False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

copy_X [bool, default=True] If True, X will be copied; else, it may be overwritten.

verbose [bool, default=False] Verbose mode when fitting the model.

Attributes

coef_ [array-like of shape (n_features,)] Coefficients of the regression model (mean of distribution)

intercept_ [float] Independent term in decision function. Set to 0.0 if `fit_intercept = False`.

alpha_ [float] Estimated precision of the noise.

lambda_ [float] Estimated precision of the weights.

sigma_ [array-like of shape (n_features, n_features)] Estimated variance-covariance matrix of the weights

scores_ [array-like of shape (n_iter_+1,)] If computed_score is True, value of the log marginal likelihood (to be maximized) at each iteration of the optimization. The array starts with the value of the log marginal likelihood obtained for the initial values of alpha and lambda and ends with the value obtained for the estimated alpha and lambda.

n_iter_ [int] The actual number of iterations to reach the stopping criterion.

Notes

There exist several strategies to perform Bayesian ridge regression. This implementation is based on the algorithm described in Appendix A of (Tipping, 2001) where updates of the regularization parameters are done as suggested in (MacKay, 1992). Note that according to A New View of Automatic Relevance Determination (Wipf and Nagarajan, 2008) these update rules do not guarantee that the marginal likelihood is increasing between two consecutive iterations of the optimization.

References

D. J. C. MacKay, Bayesian Interpolation, Computation and Neural Systems, Vol. 4, No. 3, 1992.

M. E. Tipping, Sparse Bayesian Learning and the Relevance Vector Machine, Journal of Machine Learning Research, Vol. 1, 2001.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.BayesianRidge()
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
BayesianRidge()
>>> clf.predict([[1, 1]])
array([1.])
```

Methods

fit(self, X, y[, sample_weight])	Fit the model
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, return_std])	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_iter=300*, *tol=0.001*, *alpha_1=1e-06*, *alpha_2=1e-06*, *lambda_1=1e-06*, *lambda_2=1e-06*, *alpha_init=None*, *lambda_init=None*, *compute_score=False*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *verbose=False*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit the model

Parameters

X [ndarray of shape (n_samples, n_features)] Training data

y [ndarray of shape (n_samples,)] Target values. Will be cast to X's dtype if necessary

sample_weight [ndarray of shape (n_samples,), default=None] Individual weights for each sample

New in version 0.20: parameter *sample_weight* support to BayesianRidge.

Returns

self [returns an instance of self.]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*, *return_std=False*)
Predict using the linear model.

In addition to the mean of the predictive distribution, also its standard deviation can be returned.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Samples.

return_std [bool, default=False] Whether to return the standard deviation of posterior prediction.

Returns

y_mean [array-like of shape (n_samples,)] Mean of predictive distribution of query points.

y_std [array-like of shape (n_samples,)] Standard deviation of predictive distribution of query points.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples,

n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.linear_model.BayesianRidge`

- *Feature agglomeration vs. univariate selection*
- *Curve Fitting with Bayesian Ridge Regression*
- *Bayesian Ridge Regression*
- *Imputing missing values with variants of IterativeImputer*

7.22.5 Multi-task linear regressors with variable selection

These estimators fit multiple regression problems (or tasks) jointly, while inducing sparse coefficients. While the inferred coefficients may differ between the tasks, they are constrained to agree on the features that are selected (non-zero coefficients).

linear_model.MultiTaskElasticNet([alpha, ...])	Multi-task ElasticNet model trained with L1/L2 mixed-norm as regularizer
linear_model.MultiTaskElasticNetCV([...])	Multi-task L1/L2 ElasticNet with built-in cross-validation.
linear_model.MultiTaskLasso([alpha, ...])	Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.

Continued on next page

Table 173 – continued from previous page

linear_model.MultiTaskLassoCV([eps, ...])	Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.

sklearn.linear_model.MultiTaskElasticNet

class sklearn.linear_model.**MultiTaskElasticNet**(*alpha=1.0*, *l1_ratio=0.5*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *random_state=None*, *selection='cyclic'*)

Multi-task ElasticNet model trained with L1/L2 mixed-norm as regularizer

The optimization objective for MultiTaskElasticNet is:

```
(1 / (2 * n_samples)) * ||Y - XW||_Fro^2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = sum_i sqrt(sum_j w_ij ^ 2)
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

alpha [float, optional] Constant that multiplies the L1/L2 term. Defaults to 1.0

l1_ratio [float] The ElasticNet mixing parameter, with 0 < l1_ratio <= 1. For l1_ratio = 1 the penalty is an L1/L2 penalty. For l1_ratio = 0 it is an L2 penalty. For `0 < l1_ratio < 1`, the penalty is a combination of L1/L2 and L2.

fit_intercept [boolean] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *sklearn.preprocessing.StandardScaler* before calling `fit` on an estimator with `normalize=False`.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

warm_start [bool, optional] When set to `True`, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection` == 'random'.

selection [str, default 'cyclic'] If set to 'random', a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to 'random') often leads to significantly faster convergence especially when tol is higher than 1e-4.

Attributes

intercept_ [array, shape (n_tasks,)] Independent term in decision function.

coef_ [array, shape (n_tasks, n_features)] Parameter vector (W in the cost function formula). If a 1D y is passed in at fit (non multi-task usage), `coef_` is then a 1D array. Note that `coef_` stores the transpose of `W`, `W.T`.

n_iter_ [int] number of iterations run by the coordinate descent solver to reach the specified tolerance.

See also:

MultiTaskElasticNet Multi-task L1/L2 ElasticNet with built-in cross-validation.

ElasticNet

MultiTaskLasso

Notes

The algorithm used to fit the model is coordinate descent.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.MultiTaskElasticNet(alpha=0.1)
>>> clf.fit([[0,0], [1, 1], [2, 2]], [[0, 0], [1, 1], [2, 2]])
MultiTaskElasticNet(alpha=0.1)
>>> print(clf.coef_)
[[0.45663524 0.45612256]
 [0.45663524 0.45612256]]
>>> print(clf.intercept_)
[0.0872422 0.0872422]
```

Methods

fit(self, X, y)	Fit MultiTaskElasticNet model with coordinate descent
get_params(self[, deep])	Get parameters for this estimator.
path(X, y[, l1_ratio, eps, n_alphas, ...])	Compute elastic net path with coordinate descent.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *l1_ratio=0.5*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *random_state=None*, *selection='cyclic'*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)
Fit MultiTaskElasticNet model with coordinate descent

Parameters

X [ndarray, shape (n_samples, n_features)] Data

y [ndarray, shape (n_samples, n_tasks)] Target. Will be cast to X's dtype if necessary

Notes

Coordinate descent is an algorithm that considers each column of data at a time hence it will automatically convert the X input as a Fortran-contiguous numpy array if necessary.

To avoid memory re-allocation it is advised to allocate the initial data in memory directly using that format.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

static path(*X*, *y*, *l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, *check_input=True*, ***params*)
Compute elastic net path with coordinate descent.

The elastic net optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] Target values.

l1_ratio [float, optional] Number between 0 and 1 passed to elastic net (scaling between l1 and l2 penalties). `l1_ratio=1` corresponds to the Lasso.

eps [float] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

n_alphas [int, optional] Number of alphas along the regularization path.

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [bool, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or int] Amount of verbosity.

return_n_iter [bool] Whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

check_input [bool, default True] Skip input validation checks, including the Gram matrix when provided assuming there are handled by the caller when check_input=False.

****params** [kwargs] Keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha. (Is returned when `return_n_iter` is set to True).

See also:

MultiTaskElasticNet

MultiTaskElasticNetCV

ElasticNet

ElasticNetCV

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

predict(*self*, *X*)
Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

property sparse_coef_
sparse representation of the fitted `coef_`

sklearn.linear_model.MultiTaskLasso

class sklearn.linear_model.**MultiTaskLasso**(*alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *random_state=None*, *selection='cyclic'*)

Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||Y - XW||^2_Fro + alpha * ||W||_21
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

alpha [float, optional] Constant that multiplies the L1/L2 term. Defaults to 1.0

fit_intercept [boolean] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

warm_start [bool, optional] When set to `True`, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection` == 'random'.

selection [str, default 'cyclic'] If set to 'random', a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to 'random') often leads to significantly faster convergence especially when tol is higher than 1e-4

Attributes

coef_ [array, shape (n_tasks, n_features)] Parameter vector (W in the cost function formula). Note that `coef_` stores the transpose of `W`, `W.T`.

intercept_ [array, shape (n_tasks,)] independent term in decision function.

n_iter_ [int] number of iterations run by the coordinate descent solver to reach the specified tolerance.

See also:

MultiTaskLasso Multi-task L1/L2 Lasso with built-in cross-validation

Lasso

MultiTaskElasticNet

Notes

The algorithm used to fit the model is coordinate descent.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.MultiTaskLasso(alpha=0.1)
>>> clf.fit([[0,0], [1, 1], [2, 2]], [[0, 0], [1, 1], [2, 2]])
MultiTaskLasso(alpha=0.1)
>>> print(clf.coef_)
[[0.89393398 0.        ]
 [0.89393398 0.        ]]
>>> print(clf.intercept_)
[0.10606602 0.10606602]
```

Methods

fit(self, X, y)	Fit MultiTaskElasticNet model with coordinate descent
get_params(self[, deep])	Get parameters for this estimator.
path(X, y[, l1_ratio, eps, n_alphas, ...])	Compute elastic net path with coordinate descent.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *random_state=None*, *selection='cyclic'*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)
Fit MultiTaskElasticNet model with coordinate descent

Parameters

X [ndarray, shape (n_samples, n_features)] Data

y [ndarray, shape (n_samples, n_tasks)] Target. Will be cast to X's dtype if necessary

Notes

Coordinate descent is an algorithm that considers each column of data at a time hence it will automatically convert the X input as a Fortran-contiguous numpy array if necessary.

To avoid memory re-allocation it is advised to allocate the initial data in memory directly using that format.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

static path(*X*, *y*, *l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, *check_input=True*, ***params*)
Compute elastic net path with coordinate descent.

The elastic net optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] Target values.

l1_ratio [float, optional] Number between 0 and 1 passed to elastic net (scaling between l1 and l2 penalties). `l1_ratio=1` corresponds to the Lasso.

eps [float] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

n_alphas [int, optional] Number of alphas along the regularization path.

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [bool, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or int] Amount of verbosity.

return_n_iter [bool] Whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

check_input [bool, default True] Skip input validation checks, including the Gram matrix when provided assuming there are handled by the caller when check_input=False.

****params** [kwargs] Keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha. (Is returned when `return_n_iter` is set to True).

See also:

MultiTaskElasticNet

MultiTaskElasticNetCV

ElasticNet

ElasticNetCV

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

predict(*self*, *X*)

Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

property sparse_coef_
sparse representation of the fitted `coef_`

Examples using `sklearn.linear_model.MultiTaskLasso`

- *Joint feature selection with multi-task Lasso*

7.22.6 Outlier-robust regressors

Any estimator using the Huber loss would also be robust to outliers, e.g. *SGDRegressor* with `loss='huber'`.

linear_model.HuberRegressor([epsilon, ...])	Linear regression model that is robust to outliers.
linear_model.RANSACRegressor([...])	RANSAC (RANdom SAmple Consensus) algorithm.
linear_model.TheilSenRegressor([...])	Theil-Sen Estimator: robust multivariate regression model.

sklearn.linear_model.HuberRegressor

class sklearn.linear_model.**HuberRegressor**(*epsilon=1.35*, *max_iter=100*, *alpha=0.0001*, *warm_start=False*, *fit_intercept=True*, *tol=1e-05*)

Linear regression model that is robust to outliers.

The Huber Regressor optimizes the squared loss for the samples where `|(y - X'w) / sigma| < epsilon` and the absolute loss for the samples where `|(y - X'w) / sigma| > epsilon`, where w and sigma are parameters to be optimized. The parameter sigma makes sure that if y is scaled up or down by a certain factor, one does not need to rescale epsilon to achieve the same robustness. Note that this does not take into account the fact that the different features of X may be of different scales.

This makes sure that the loss function is not heavily influenced by the outliers while not completely ignoring their effect.

Read more in the *User Guide*

New in version 0.18.

Parameters

epsilon [float, greater than 1.0, default 1.35] The parameter epsilon controls the number of samples that should be classified as outliers. The smaller the epsilon, the more robust it is to outliers.

max_iter [int, default 100] Maximum number of iterations that `scipy.optimize.minimize(method="L-BFGS-B")` should run for.

alpha [float, default 0.0001] Regularization parameter.

warm_start [bool, default False] This is useful if the stored attributes of a previously used model has to be reused. If set to False, then the coefficients will be rewritten for every call to fit. See *the Glossary*.

fit_intercept [bool, default True] Whether or not to fit the intercept. This can be set to False if the data is already centered around the origin.

tol [float, default 1e-5] The iteration will stop when `max{|proj g_i | i = 1, ..., n}` <= `tol` where pg_i is the i-th component of the projected gradient.

Attributes

coef_ [array, shape (n_features,)] Features got by optimizing the Huber loss.

intercept_ [float] Bias.

scale_ [float] The value by which `|y - X'w - c|` is scaled down.

n_iter_ [int] Number of iterations that `scipy.optimize.minimize(method="L-BFGS-B")` has run for.

Changed in version 0.20: In SciPy <= 1.0.0 the number of lbfgs iterations may exceed `max_iter`. `n_iter_` will now report at most `max_iter`.

outliers_ [array, shape (n_samples,)] A boolean mask which is set to True where the samples are identified as outliers.

References

[Re4616ef910fb-1], [Re4616ef910fb-2]

Examples

```
>>> import numpy as np
>>> from sklearn.linear_model import HuberRegressor, LinearRegression
>>> from sklearn.datasets import make_regression
>>> rng = np.random.RandomState(0)
>>> X, y, coef = make_regression(
...     n_samples=200, n_features=2, noise=4.0, coef=True, random_state=0)
>>> X[:4] = rng.uniform(10, 20, (4, 2))
>>> y[:4] = rng.uniform(10, 20, 4)
>>> huber = HuberRegressor().fit(X, y)
>>> huber.score(X, y)
-7.284608623514573
>>> huber.predict(X[:1,])
array([806.7200...])
>>> linear = LinearRegression().fit(X, y)
>>> print("True coefficients:", coef)
True coefficients: [20.4923...  34.1698...]
>>> print("Huber coefficients:", huber.coef_)
Huber coefficients: [17.7906... 31.0106...]
>>> print("Linear Regression coefficients:", linear.coef_)
Linear Regression coefficients: [-1.9221...  7.0226...]
```

Methods

fit(self, X, y[, sample_weight])	Fit the model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *epsilon=1.35*, *max_iter=100*, *alpha=0.0001*, *warm_start=False*, *fit_intercept=True*, *tol=1e-05*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit the model according to the given training data.

Parameters

X [array-like, shape (n_samples, n_features)] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [array-like, shape (n_samples,)] Target vector relative to X.

sample_weight [array-like, shape (n_samples,)] Weight given to each sample.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.linear_model.HuberRegressor`

- *HuberRegressor vs Ridge on dataset with strong outliers*
- *Robust linear estimator fitting*

`sklearn.linear_model.`RANSACRegressor

class sklearn.linear_model.**RANSACRegressor**(*base_estimator=None*, *min_samples=None*, *residual_threshold=None*, *is_data_valid=None*, *is_model_valid=None*, *max_trials=100*, *max_skips=inf*, *stop_n_inliers=inf*, *stop_score=inf*, *stop_probability=0.99*, *loss='absolute_loss'*, *random_state=None*)

RANSAC (RANdom SAmple Consensus) algorithm.

RANSAC is an iterative algorithm for the robust estimation of parameters from a subset of inliers from the complete data set.

Read more in the *User Guide*.

Parameters

base_estimator [object, optional] Base estimator object which implements the following methods:

- `fit(X, y)`: Fit model to given training data and target values.
- `score(X, y)`: Returns the mean accuracy on the given test data, which is used for the stop criterion defined by `stop_score`. Additionally, the score is used to decide which of two equally large consensus sets is chosen as the better one.
- `predict(X)`: Returns predicted values using the linear model, which is used to compute residual error using loss function.

If `base_estimator` is None, then `base_estimator=sklearn.linear_model.LinearRegression()` is used for target values of dtype float.

Note that the current implementation only supports regression estimators.

min_samples [int (>= 1) or float ([0, 1]), optional] Minimum number of samples chosen randomly from original data. Treated as an absolute number of samples for `min_samples >= 1`, treated as a relative number `ceil(min_samples * X.shape[0])` for `min_samples < 1`. This is typically chosen as the minimal number of samples necessary to estimate the given `base_estimator`. By default a `sklearn.linear_model.LinearRegression()` estimator is assumed and `min_samples` is chosen as `X.shape[1] + 1`.

residual_threshold [float, optional] Maximum residual for a data sample to be classified as an inlier. By default the threshold is chosen as the MAD (median absolute deviation) of the target values `y`.

is_data_valid [callable, optional] This function is called with the randomly selected data before the model is fitted to it: `is_data_valid(X, y)`. If its return value is False the current randomly chosen sub-sample is skipped.

is_model_valid [callable, optional] This function is called with the estimated model and the randomly selected data: `is_model_valid(model, X, y)`. If its return value is False the current randomly chosen sub-sample is skipped. Rejecting samples with this function is computationally costlier than with `is_data_valid`. `is_model_valid` should therefore only be used if the estimated model is needed for making the rejection decision.

max_trials [int, optional] Maximum number of iterations for random sample selection.

max_skips [int, optional] Maximum number of iterations that can be skipped due to finding zero inliers or invalid data defined by `is_data_valid` or invalid models defined by `is_model_valid`.

New in version 0.19.

stop_n_inliers [int, optional] Stop iteration if at least this number of inliers are found.

stop_score [float, optional] Stop iteration if score is greater equal than this threshold.

stop_probability [float in range [0, 1], optional] RANSAC iteration stops if at least one outlier-free set of the training data is sampled in RANSAC. This requires to generate at least N samples (iterations):

```
N >= log(1 - probability) / log(1 - e**m)
```

where the probability (confidence) is typically set to high value such as 0.99 (the default) and e is the current fraction of inliers w.r.t. the total number of samples.

loss [string, callable, optional, default "absolute_loss"] String inputs, "absolute_loss" and "squared_loss" are supported which find the absolute loss and squared loss per sample respectively.

If `loss` is a callable, then it should be a function that takes two arrays as inputs, the true and predicted value and returns a 1-D array with the i-th value of the array corresponding to the loss on `X[i]`.

If the loss on a sample is greater than the `residual_threshold`, then this sample is classified as an outlier.

random_state [int, RandomState instance or None, optional, default None] The generator used to initialize the centers. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Attributes

estimator_ [object] Best fitted model (copy of the `base_estimator` object).

n_trials_ [int] Number of random selection trials until one of the stop criteria is met. It is always <= `max_trials`.

inlier_mask_ [bool array of shape [n_samples]] Boolean mask of inliers classified as `True`.

n_skips_no_inliers_ [int] Number of iterations skipped due to finding zero inliers.

New in version 0.19.

n_skips_invalid_data_ [int] Number of iterations skipped due to invalid data defined by `is_data_valid`.

New in version 0.19.

n_skips_invalid_model_ [int] Number of iterations skipped due to an invalid model defined by `is_model_valid`.

New in version 0.19.

References

[R80ce5b25cf9d-1], [R80ce5b25cf9d-2], [R80ce5b25cf9d-3]

Examples

```
>>> from sklearn.linear_model import RANSACRegressor
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(
...     n_samples=200, n_features=2, noise=4.0, random_state=0)
>>> reg = RANSACRegressor(random_state=0).fit(X, y)
>>> reg.score(X, y)
0.9885...
>>> reg.predict(X[:1,])
array([-31.9417...])
```

Methods

fit(self, X, y[, sample_weight])	Fit estimator using RANSAC algorithm.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the estimated model.
score(self, X, y)	Returns the score of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *base_estimator=None*, *min_samples=None*, *residual_threshold=None*, *is_data_valid=None*, *is_model_valid=None*, *max_trials=100*, *max_skips=inf*, *stop_n_inliers=inf*, *stop_score=inf*, *stop_probability=0.99*, *loss='absolute_loss'*, *random_state=None*)

Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *sample_weight=None*)

Fit estimator using RANSAC algorithm.

Parameters

X [array-like or sparse matrix, shape [n_samples, n_features]] Training data.

y [array-like of shape (n_samples,) or (n_samples, n_targets)] Target values.

sample_weight [array-like of shape (n_samples,), default=None] Individual weights for each sample raises error if sample_weight is passed and base_estimator fit method does not support it.

Raises

ValueError If no valid consensus set could be found. This occurs if `is_data_valid` and `is_model_valid` return False for all `max_trials` randomly chosen sub-samples.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict using the estimated model.

This is a wrapper for `estimator_.predict(X)`.

Parameters

X [numpy array of shape [n_samples, n_features]]

Returns

y [array, shape = [n_samples] or [n_samples, n_targets]] Returns predicted values.

score (*self*, *X*, *y*)
Returns the score of the prediction.

This is a wrapper for `estimator_.score(X, y)`.

Parameters

X [numpy array or sparse matrix of shape [n_samples, n_features]] Training data.

y [array, shape = [n_samples] or [n_samples, n_targets]] Target values.

Returns

z [float] Score of the prediction.

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.linear_model.RANSACRegressor`

- *Robust linear model estimation using RANSAC*
- *Theil-Sen Regression*
- *Robust linear estimator fitting*

sklearn.linear_model.TheilSenRegressor

class sklearn.linear_model.**TheilSenRegressor**(*fit_intercept=True*, *copy_X=True*, *max_subpopulation=10000.0*, *n_subsamples=None*, *max_iter=300*, *tol=0.001*, *random_state=None*, *n_jobs=None*, *verbose=False*)

Theil-Sen Estimator: robust multivariate regression model.

The algorithm calculates least square solutions on subsets with size n_subsamples of the samples in X. Any value of n_subsamples between the number of features and samples leads to an estimator with a compromise between robustness and efficiency. Since the number of least square solutions is "n_samples choose n_subsamples", it can be extremely large and can therefore be limited with max_subpopulation. If this limit is reached, the subsets are chosen randomly. In a final step, the spatial median (or L1 median) is calculated of all least square solutions.

Read more in the *User Guide*.

Parameters

fit_intercept [boolean, optional, default True] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

max_subpopulation [int, optional, default 1e4] Instead of computing with a set of cardinality 'n choose k', where n is the number of samples and k is the number of subsamples (at least number of features), consider only a stochastic subpopulation of a given maximal size if 'n choose k' is larger than max_subpopulation. For other than small problem sizes this parameter will determine memory usage and runtime if n_subsamples is not changed.

n_subsamples [int, optional, default None] Number of samples to calculate the parameters. This is at least the number of features (plus 1 if fit_intercept=True) and the number of samples as a maximum. A lower number leads to a higher breakdown point and a low efficiency while a high number leads to a low breakdown point and a high efficiency. If None, take the minimum number of subsamples leading to maximal robustness. If n_subsamples is set to n_samples, Theil-Sen is identical to least squares.

max_iter [int, optional, default 300] Maximum number of iterations for the calculation of spatial median.

tol [float, optional, default 1.e-3] Tolerance when calculating spatial median.

random_state [int, RandomState instance or None, optional, default None] A random number generator instance to define the state of the random permutations generator. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

n_jobs [int or None, optional (default=None)] Number of CPUs to use during the cross validation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [boolean, optional, default False] Verbose mode when fitting the model.

Attributes

coef_ [array, shape = (n_features)] Coefficients of the regression model (median of distribution).

intercept_ [float] Estimated intercept of regression model.

breakdown_ [float] Approximated breakdown point.

n_iter_ [int] Number of iterations needed for the spatial median.

n_subpopulation_ [int] Number of combinations taken into account from 'n choose k', where n is the number of samples and k is the number of subsamples.

References

- Theil-Sen Estimators in a Multiple Linear Regression Model, 2009 Xin Dang, Hanxiang Peng, Xueqin Wang and Heping Zhang http://home.olemiss.edu/~xdang/papers/MTSE.pdf

Examples

```
>>> from sklearn.linear_model import TheilSenRegressor
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(
...     n_samples=200, n_features=2, noise=4.0, random_state=0)
>>> reg = TheilSenRegressor(random_state=0).fit(X, y)
>>> reg.score(X, y)
0.9884...
>>> reg.predict(X[:1,])
array([-31.5871...])
```

Methods

fit(self, X, y)	Fit linear model.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *fit_intercept=True*, *copy_X=True*, *max_subpopulation=10000.0*, *n_subsamples=None*, *max_iter=300*, *tol=0.001*, *random_state=None*, *n_jobs=None*, *verbose=False*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Fit linear model.

Parameters

X [numpy array of shape [n_samples, n_features]] Training data

y [numpy array of shape [n_samples]] Target values

Returns

self [returns an instance of self.]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)

Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.linear_model.TheilSenRegressor`

- *Theil-Sen Regression*
- *Robust linear estimator fitting*

7.22.7 Miscellaneous

linear_model.PassiveAggressiveRegressor([C, ...])	Passive Aggressive Regressor
linear_model.enet_path(X, y[, l1_ratio, ...])	Compute elastic net path with coordinate descent.
linear_model.lars_path(X, y[, Xy, Gram, ...])	Compute Least Angle Regression or Lasso path using LARS algorithm [1]
linear_model.lars_path_gram(Xy, Gram, n_samples)	lars_path in the sufficient stats mode [1]
linear_model.lasso_path(X, y[, eps, ...])	Compute Lasso path with coordinate descent
linear_model.orthogonal_mp(X, y[, ...])	Orthogonal Matching Pursuit (OMP)
linear_model.orthogonal_mp_gram(Gram, Xy[, ...])	Gram Orthogonal Matching Pursuit (OMP)
linear_model.ridge_regression(X, y, alpha[, ...])	Solve the ridge equation by the method of normal equations.

`sklearn.linear_model.`PassiveAggressiveRegressor

`sklearn.linear_model.`**`PassiveAggressiveRegressor`**(*C=1.0*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *shuffle=True*, *verbose=0*, *loss='epsilon_insensitive'*, *epsilon=0.1*, *random_state=None*, *warm_start=False*, *average=False*)

Passive Aggressive Regressor

Read more in the *User Guide*.

Parameters

C [float] Maximum step size (regularization). Defaults to 1.0.

fit_intercept [bool] Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

max_iter [int, optional (default=1000)] The maximum number of passes over the training data (aka epochs). It only impacts the behavior in the `fit` method, and not the `partial_fit` method.

New in version 0.19.

tol [float or None, optional (default=1e-3)] The stopping criterion. If it is not None, the iterations will stop when (loss > previous_loss - tol).

New in version 0.19.

early_stopping [bool, default=False] Whether to use early stopping to terminate training when validation. score is not improving. If set to True, it will automatically set aside a fraction of

training data as validation and terminate training when validation score is not improving by at least tol for n_iter_no_change consecutive epochs.

New in version 0.20.

validation_fraction [float, default=0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.

New in version 0.20.

n_iter_no_change [int, default=5] Number of iterations with no improvement to wait before early stopping.

New in version 0.20.

shuffle [bool, default=True] Whether or not the training data should be shuffled after each epoch.

verbose [integer, optional] The verbosity level

loss [string, optional] The loss function to be used: epsilon_insensitive: equivalent to PA-I in the reference paper. squared_epsilon_insensitive: equivalent to PA-II in the reference paper.

epsilon [float] If the difference between the current prediction and the correct label is below this threshold, the model is not updated.

random_state [int, RandomState instance or None, optional, default=None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

Repeatedly calling fit or partial_fit when warm_start is True can result in a different solution than when calling fit a single time because of the way the data is shuffled.

average [bool or int, optional] When set to True, computes the averaged SGD weights and stores the result in the `coef_` attribute. If set to an int greater than 1, averaging will begin once the total number of samples seen reaches average. So average=10 will begin averaging after seeing 10 samples.

New in version 0.19: parameter *average* to use weights averaging in SGD

Attributes

coef_ [array, shape = [1, n_features] if n_classes == 2 else [n_classes, n_features]] Weights assigned to the features.

intercept_ [array, shape = [1] if n_classes == 2 else [n_classes]] Constants in decision function.

n_iter_ [int] The actual number of iterations to reach the stopping criterion.

t_ [int] Number of weight updates performed during training. Same as `(n_iter_ * n_samples)`.

See also:

SGDRegressor

References

Online Passive-Aggressive Algorithms <http://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06a.pdf> K. Crammer, O. Dekel, J. Keshat, S. Shalev-Shwartz, Y. Singer - JMLR (2006)

Examples

```
>>> from sklearn.linear_model import PassiveAggressiveRegressor
>>> from sklearn.datasets import make_regression
```

```
>>> X, y = make_regression(n_features=4, random_state=0)
>>> regr = PassiveAggressiveRegressor(max_iter=100, random_state=0,
... tol=1e-3)
>>> regr.fit(X, y)
PassiveAggressiveRegressor(max_iter=100, random_state=0)
>>> print(regr.coef_)
[20.48736655 34.18818427 67.59122734 87.94731329]
>>> print(regr.intercept_)
[-0.02306214]
>>> print(regr.predict([[0, 0, 0, 0]]))
[-0.02306214]
```

sklearn.linear_model.enet_path

sklearn.linear_model.**enet_path**(*X*, *y*, *l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, *check_input=True*, ***params*)

Compute elastic net path with coordinate descent.

The elastic net optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] Target values.

l1_ratio [float, optional] Number between 0 and 1 passed to elastic net (scaling between l1 and l2 penalties). `l1_ratio=1` corresponds to the Lasso.

eps [float] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

n_alphas [int, optional] Number of alphas along the regularization path.

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [bool, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or int] Amount of verbosity.

return_n_iter [bool] Whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

check_input [bool, default True] Skip input validation checks, including the Gram matrix when provided assuming there are handled by the caller when check_input=False.

****params** [kwargs] Keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha. (Is returned when `return_n_iter` is set to True).

See also:

MultiTaskElasticNet

MultiTaskElasticNetCV

ElasticNet

ElasticNetCV

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

Examples using `sklearn.linear_model.enet_path`

- *Lasso and Elastic Net*

`sklearn.linear_model.lars_path`

sklearn.linear_model.**lars_path**(*X*, *y*, *Xy=None*, *Gram=None*, *max_iter=500*, *alpha_min=0*, *method='lar'*, *copy_X=True*, *eps=2.220446049250313e-16*, *copy_Gram=True*, *verbose=0*, *return_path=True*, *return_n_iter=False*, *positive=False*)

Compute Least Angle Regression or Lasso path using LARS algorithm [1]

The optimization objective for the case method='lasso' is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

in the case of method='lars', the objective function is only known in the form of an implicit equation (see discussion in [1])

Read more in the *User Guide*.

Parameters

X [None or array-like of shape (n_samples, n_features)] Input data. Note that if X is None then the Gram matrix must be specified, i.e., cannot be None or False.

Deprecated since version 0.21: The use of `X` is `None` in combination with `Gram` is not `None` will be removed in v0.23. Use *`lars_path_gram`* instead.

y [None or array-like of shape (n_samples,)] Input targets.

Xy [array-like of shape (n_samples,) or (n_samples, n_targets), default=None] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

Gram [None, 'auto', array-like of shape (n_features, n_features), default=None] Precomputed Gram matrix (X' * X), if `'auto'`, the Gram matrix is precomputed from the given X, if there are more samples than features.

Deprecated since version 0.21: The use of `X` is `None` in combination with `Gram` is not None will be removed in v0.23. Use *`lars_path_gram`* instead.

max_iter [int, default=500] Maximum number of iterations to perform, set to infinity for no limit.

alpha_min [float, default=0] Minimum correlation along the path. It corresponds to the regularization parameter alpha parameter in the Lasso.

method [{'lar', 'lasso'}, default='lar'] Specifies the returned model. Select `'lar'` for Least Angle Regression, `'lasso'` for the Lasso.

copy_X [bool, default=True] If `False`, `X` is overwritten.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. By default, `np.finfo(np.float).eps` is used.

copy_Gram [bool, default=True] If `False`, `Gram` is overwritten.

verbose [int, default=0] Controls output verbosity.

return_path [bool, default=True] If `return_path==True` returns the entire path, else returns only the last point of the path.

return_n_iter [bool, default=False] Whether to return the number of iterations.

positive [bool, default=False] Restrict coefficients to be >= 0. This option is only allowed with method 'lasso'. Note that the model coefficients will not converge to the ordinary-least-squares solution for small values of alpha. Only coefficients up to the smallest alpha value (`alphas_[alphas_ > 0.].min()` when fit_path=True) reached by the stepwise Lars-Lasso algorithm are typically in congruence with the solution of the coordinate descent lasso_path function.

Returns

alphas [array-like of shape (n_alphas + 1,)] Maximum of covariances (in absolute value) at each iteration. `n_alphas` is either `max_iter`, `n_features` or the number of nodes in the path with `alpha >= alpha_min`, whichever is smaller.

active [array-like of shape (n_alphas,)] Indices of active variables at the end of the path.

coefs [array-like of shape (n_features, n_alphas + 1)] Coefficients along the path

n_iter [int] Number of iterations run. Returned only if return_n_iter is set to True.

See also:

lars_path_gram

lasso_path

lasso_path_gram

LassoLars

Lars

LassoLarsCV

LarsCV

sklearn.decomposition.sparse_encode

References

[1], [2], [3]

Examples using `sklearn.linear_model.lars_path`

- *Lasso path using LARS*

sklearn.linear_model.lars_path_gram

sklearn.linear_model.**lars_path_gram**(*Xy*, *Gram*, *n_samples*, *max_iter=500*, *alpha_min=0*, *method='lar'*, *copy_X=True*, *eps=2.220446049250313e-16*, *copy_Gram=True*, *verbose=0*, *return_path=True*, *return_n_iter=False*, *positive=False*)

lars_path in the sufficient stats mode [1]

The optimization objective for the case method='lasso' is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

in the case of method='lars', the objective function is only known in the form of an implicit equation (see discussion in [1])

Read more in the *User Guide*.

Parameters

Xy [array-like of shape (n_samples,) or (n_samples, n_targets)] Xy = np.dot(X.T, y).

Gram [array-like of shape (n_features, n_features)] Gram = np.dot(X.T * X).

n_samples [int or float] Equivalent size of sample.

max_iter [int, default=500] Maximum number of iterations to perform, set to infinity for no limit.

alpha_min [float, default=0] Minimum correlation along the path. It corresponds to the regularization parameter alpha parameter in the Lasso.

method [{'lar', 'lasso'}, default='lar'] Specifies the returned model. Select `'lar'` for Least Angle Regression, `'lasso'` for the Lasso.

copy_X [bool, default=True] If `False`, `X` is overwritten.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. By default, `np.finfo(np.float).eps` is used.

copy_Gram [bool, default=True] If `False`, `Gram` is overwritten.

verbose [int, default=0] Controls output verbosity.

return_path [bool, default=True] If `return_path==True` returns the entire path, else returns only the last point of the path.

return_n_iter [bool, default=False] Whether to return the number of iterations.

positive [bool, default=False] Restrict coefficients to be >= 0. This option is only allowed with method 'lasso'. Note that the model coefficients will not converge to the ordinary-least-squares solution for small values of alpha. Only coefficients up to the smallest alpha value (`alphas_[alphas_ > 0.].min()` when fit_path=True) reached by the stepwise Lars-Lasso algorithm are typically in congruence with the solution of the coordinate descent lasso_path function.

Returns

alphas [array-like of shape (n_alphas + 1,)] Maximum of covariances (in absolute value) at each iteration. `n_alphas` is either `max_iter`, `n_features` or the number of nodes in the path with `alpha >= alpha_min`, whichever is smaller.

active [array-like of shape (n_alphas,)] Indices of active variables at the end of the path.

coefs [array-like of shape (n_features, n_alphas + 1)] Coefficients along the path

n_iter [int] Number of iterations run. Returned only if return_n_iter is set to True.

See also:

lars_path

lasso_path

lasso_path_gram

LassoLars

Lars

LassoLarsCV

LarsCV

sklearn.decomposition.sparse_encode

References

[1], [2], [3]

sklearn.linear_model.lasso_path

sklearn.linear_model.**lasso_path**(*X*, *y*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, ***params*)

Compute Lasso path with coordinate descent

The Lasso optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^2_Fro + alpha * ||W||_21
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,), or (n_samples, n_outputs)] Target values

eps [float, optional] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`

n_alphas [int, optional] Number of alphas along the regularization path

alphas [ndarray, optional] List of alphas where to compute the models. If `None` alphas are set automatically

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or integer] Amount of verbosity.

return_n_iter [bool] whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

****params** [kwargs] keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha.

See also:

lars_path

Lasso

LassoLars

LassoCV

LassoLarsCV

sklearn.decomposition.sparse_encode

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Note that in certain cases, the Lars solver may be significantly faster to implement this functionality. In particular, linear interpolation can be used to retrieve model coefficients between the values output by lars_path

Examples

Comparing lasso_path and lars_path with interpolation:

```
>>> X = np.array([[1, 2, 3.1], [2.3, 5.4, 4.3]]).T
>>> y = np.array([1, 2, 3.1])
>>> # Use lasso_path to compute a coefficient path
>>> _, coef_path, _ = lasso_path(X, y, alphas=[5., 1., .5])
>>> print(coef_path)
[[0.          0.          0.46874778]
 [0.2159048   0.4425765   0.23689075]]
```

```
>>> # Now use lars_path and 1D linear interpolation to compute the
>>> # same path
>>> from sklearn.linear_model import lars_path
>>> alphas, active, coef_path_lars = lars_path(X, y, method='lasso')
>>> from scipy import interpolate
>>> coef_path_continuous = interpolate.interp1d(alphas[::-1],
...                                             coef_path_lars[:, ::-1])
>>> print(coef_path_continuous([5., 1., .5]))
[[0.         0.         0.46915237]
 [0.2159048  0.4425765  0.23668876]]
```

Examples using `sklearn.linear_model.lasso_path`

- *Lasso and Elastic Net*

`sklearn.linear_model`.orthogonal_mp

`sklearn.linear_model.`**`orthogonal_mp`**(*X*, *y*, *n_nonzero_coefs=None*, *tol=None*, *precompute=False*, *copy_X=True*, *return_path=False*, *return_n_iter=False*)

Orthogonal Matching Pursuit (OMP)

Solves n_targets Orthogonal Matching Pursuit problems. An instance of the problem has the form:

When parametrized by the number of non-zero coefficients using `n_nonzero_coefs`: argmin ||y - Xgamma||^2 subject to ||gamma||_0 <= n_{nonzero coefs}

When parametrized by error using the parameter `tol`: argmin ||gamma||_0 subject to ||y - Xgamma||^2 <= tol

Read more in the *User Guide*.

Parameters

X [array, shape (n_samples, n_features)] Input data. Columns are assumed to have unit norm.

y [array, shape (n_samples,) or (n_samples, n_targets)] Input targets

n_nonzero_coefs [int] Desired number of non-zero entries in the solution. If None (by default) this value is set to 10% of n_features.

tol [float] Maximum norm of the residual. If not None, overrides n_nonzero_coefs.

precompute [{True, False, 'auto'},] Whether to perform precomputations. Improves performance when n_targets or n_samples is very large.

copy_X [bool, optional] Whether the design matrix X must be copied by the algorithm. A false value is only helpful if X is already Fortran-ordered, otherwise a copy is made anyway.

return_path [bool, optional. Default: False] Whether to return every value of the nonzero coefficients along the forward path. Useful for cross-validation.

return_n_iter [bool, optional default False] Whether or not to return the number of iterations.

Returns

coef [array, shape (n_features,) or (n_features, n_targets)] Coefficients of the OMP solution. If `return_path=True`, this contains the whole coefficient path. In this case its shape is (n_features, n_features) or (n_features, n_targets, n_features) and iterating over the last axis yields coefficients in increasing order of active features.

n_iters [array-like or int] Number of active features across every target. Returned only if `return_n_iter` is set to True.

See also:

OrthogonalMatchingPursuit

orthogonal_mp_gram

lars_path

decomposition.sparse_encode

Notes

Orthogonal matching pursuit was introduced in S. Mallat, Z. Zhang, Matching pursuits with time-frequency dictionaries, IEEE Transactions on Signal Processing, Vol. 41, No. 12. (December 1993), pp. 3397-3415. (http://blanche.polytechnique.fr/~mallat/papiers/MallatPursuit93.pdf)

This implementation is based on Rubinstein, R., Zibulevsky, M. and Elad, M., Efficient Implementation of the K-SVD Algorithm using Batch Orthogonal Matching Pursuit Technical Report - CS Technion, April 2008. https://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf

sklearn.linear_model.orthogonal_mp_gram

sklearn.linear_model.**orthogonal_mp_gram**(*Gram*, *Xy*, *n_nonzero_coefs=None*, *tol=None*, *norms_squared=None*, *copy_Gram=True*, *copy_Xy=True*, *return_path=False*, *return_n_iter=False*)

Gram Orthogonal Matching Pursuit (OMP)

Solves n_targets Orthogonal Matching Pursuit problems using only the Gram matrix X.T * X and the product X.T * y.

Read more in the *User Guide*.

Parameters

Gram [array, shape (n_features, n_features)] Gram matrix of the input data: X.T * X

Xy [array, shape (n_features,) or (n_features, n_targets)] Input targets multiplied by X: X.T * y

n_nonzero_coefs [int] Desired number of non-zero entries in the solution. If None (by default) this value is set to 10% of n_features.

tol [float] Maximum norm of the residual. If not None, overrides n_nonzero_coefs.

norms_squared [array-like, shape (n_targets,)] Squared L2 norms of the lines of y. Required if tol is not None.

copy_Gram [bool, optional] Whether the gram matrix must be copied by the algorithm. A false value is only helpful if it is already Fortran-ordered, otherwise a copy is made anyway.

copy_Xy [bool, optional] Whether the covariance vector Xy must be copied by the algorithm. If False, it may be overwritten.

return_path [bool, optional. Default: False] Whether to return every value of the nonzero coefficients along the forward path. Useful for cross-validation.

return_n_iter [bool, optional default False] Whether or not to return the number of iterations.

Returns

coef [array, shape (n_features,) or (n_features, n_targets)] Coefficients of the OMP solution. If `return_path=True`, this contains the whole coefficient path. In this case its shape is (n_features, n_features) or (n_features, n_targets, n_features) and iterating over the last axis yields coefficients in increasing order of active features.

n_iters [array-like or int] Number of active features across every target. Returned only if `return_n_iter` is set to True.

See also:

OrthogonalMatchingPursuit

orthogonal_mp

lars_path

decomposition.sparse_encode

Notes

Orthogonal matching pursuit was introduced in G. Mallat, Z. Zhang, Matching pursuits with time-frequency dictionaries, IEEE Transactions on Signal Processing, Vol. 41, No. 12. (December 1993), pp. 3397-3415. (http://blanche.polytechnique.fr/~mallat/papiers/MallatPursuit93.pdf)

This implementation is based on Rubinstein, R., Zibulevsky, M. and Elad, M., Efficient Implementation of the K-SVD Algorithm using Batch Orthogonal Matching Pursuit Technical Report - CS Technion, April 2008. https://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf

sklearn.linear_model.ridge_regression

sklearn.linear_model.**ridge_regression**(*X*, *y*, *alpha*, *sample_weight=None*, *solver='auto'*, *max_iter=None*, *tol=0.001*, *verbose=0*, *random_state=None*, *return_n_iter=False*, *return_intercept=False*, *check_input=True*)

Solve the ridge equation by the method of normal equations.

Read more in the *User Guide*.

Parameters

X [{ndarray, sparse matrix, LinearOperator} of shape (n_samples, n_features)] Training data

y [ndarray of shape (n_samples,) or (n_samples, n_targets)] Target values

alpha [float or array-like of shape (n_targets,)] Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to `C^-1` in other linear models such as LogisticRegression or LinearSVC. If an array is passed, penalties are assumed to be specific to the targets. Hence they must correspond in number.

sample_weight [float or array-like of shape (n_samples,), default=None] Individual weights for each sample. If given a float, every sample will have the same weight. If sample_weight is not None and solver='auto', the solver will be set to 'cholesky'.

New in version 0.17.

solver [{'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga'}, default='auto'] Solver to use in the computational routines:

- 'auto' chooses the solver automatically based on the type of data.

- 'svd' uses a Singular Value Decomposition of X to compute the Ridge coefficients. More stable for singular matrices than 'cholesky'.
- 'cholesky' uses the standard scipy.linalg.solve function to obtain a closed-form solution via a Cholesky decomposition of dot(X.T, X)
- 'sparse_cg' uses the conjugate gradient solver as found in scipy.sparse.linalg.cg. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set `tol` and `max_iter`).
- 'lsqr' uses the dedicated regularized least-squares routine scipy.sparse.linalg.lsqr. It is the fastest and uses an iterative procedure.
- 'sag' uses a Stochastic Average Gradient descent, and 'saga' uses its improved, unbiased version named SAGA. Both methods also use an iterative procedure, and are often faster than other solvers when both n_samples and n_features are large. Note that 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from sklearn.preprocessing.

All last five solvers support both dense and sparse data. However, only 'sag' and 'sparse_cg' supports sparse input when'fit_intercept' is True.

New in version 0.17: Stochastic Average Gradient descent solver.

New in version 0.19: SAGA solver.

max_iter [int, default=None] Maximum number of iterations for conjugate gradient solver. For the 'sparse_cg' and 'lsqr' solvers, the default value is determined by scipy.sparse.linalg. For 'sag' and saga solver, the default value is 1000.

tol [float, default=1e-3] Precision of the solution.

verbose [int, default=0] Verbosity level. Setting verbose > 0 will display additional information depending on the solver used.

random_state [int, RandomState instance, default=None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'sag'.

return_n_iter [bool, default=False] If True, the method also returns `n_iter`, the actual number of iteration performed by the solver.

New in version 0.17.

return_intercept [bool, default=False] If True and if X is sparse, the method also returns the intercept, and the solver is automatically changed to 'sag'. This is only a temporary fix for fitting the intercept with sparse data. For dense data, use sklearn.linear_model._preprocess_data before your regression.

New in version 0.17.

check_input [bool, default=True] If False, the input arrays X and y will not be checked.

New in version 0.21.

Returns

coef [ndarray of shape (n_features,) or (n_targets, n_features)] Weight vector(s).

n_iter [int, optional] The actual number of iteration performed by the solver. Only returned if `return_n_iter` is True.

intercept [float or ndarray of shape (n_targets,)] The intercept of the model. Only returned if `return_intercept` is True and if X is a scipy sparse array.

Notes

This function won't compute the intercept.

7.23 `sklearn.manifold`: Manifold Learning

The *sklearn.manifold* module implements data embedding techniques.

User guide: See the *Manifold learning* section for further details.

manifold.Isomap([n_neighbors, n_components, ...])	Isomap Embedding
manifold.LocallyLinearEmbedding([...])	Locally Linear Embedding
manifold.MDS([n_components, metric, n_init, ...])	Multidimensional scaling
manifold.SpectralEmbedding([n_components, ...])	Spectral embedding for non-linear dimensionality reduction.
manifold.TSNE([n_components, perplexity, ...])	t-distributed Stochastic Neighbor Embedding.

7.23.1 `sklearn.manifold`.Isomap

class sklearn.manifold.**Isomap**(*n_neighbors=5*, *n_components=2*, *eigen_solver='auto'*, *tol=0*, *max_iter=None*, *path_method='auto'*, *neighbors_algorithm='auto'*, *n_jobs=None*, *metric='minkowski'*, *p=2*, *metric_params=None*)

Isomap Embedding

Non-linear dimensionality reduction through Isometric Mapping

Read more in the *User Guide*.

Parameters

n_neighbors [integer] number of neighbors to consider for each point.

n_components [integer] number of coordinates for the manifold

eigen_solver [['auto'|'arpack'|'dense']] 'auto' : Attempt to choose the most efficient solver for the given problem.

'arpack' : Use Arnoldi decomposition to find the eigenvalues and eigenvectors.

'dense' : Use a direct solver (i.e. LAPACK) for the eigenvalue decomposition.

tol [float] Convergence tolerance passed to arpack or lobpcg. not used if eigen_solver == 'dense'.

max_iter [integer] Maximum number of iterations for the arpack solver. not used if eigen_solver == 'dense'.

path_method [string ['auto'|'FW'|'D']] Method to use in finding shortest path.

'auto' : attempt to choose the best algorithm automatically.

'FW' : Floyd-Warshall algorithm.

'D' : Dijkstra's algorithm.

neighbors_algorithm [string ['auto'|'brute'|'kd_tree'|'ball_tree']] Algorithm to use for nearest neighbors search, passed to neighbors.NearestNeighbors instance.

n_jobs [int or None, default=None] The number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

metric [string, or callable, default="minkowski"] The metric to use when calculating distance between instances in a feature array. If metric is a string or callable, it must be one of the options allowed by *sklearn.metrics.pairwise_distances* for its metric parameter. If metric is "precomputed", X is assumed to be a distance matrix and must be square. X may be a *Glossary*.

New in version 0.22.

p [int, default=2] Parameter for the Minkowski metric from sklearn.metrics.pairwise.pairwise_distances. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

New in version 0.22.

metric_params [dict, default=None] Additional keyword arguments for the metric function.

New in version 0.22.

Attributes

embedding_ [array-like, shape (n_samples, n_components)] Stores the embedding vectors.

kernel_pca_ [object] *KernelPCA* object used to implement the embedding.

nbrs_ [sklearn.neighbors.NearestNeighbors instance] Stores nearest neighbors instance, including BallTree or KDtree if applicable.

dist_matrix_ [array-like, shape (n_samples, n_samples)] Stores the geodesic distance matrix of training data.

References

[R7f4d308f5054-1]

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.manifold import Isomap
>>> X, _ = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> embedding = Isomap(n_components=2)
>>> X_transformed = embedding.fit_transform(X[:100])
>>> X_transformed.shape
(100, 2)
```

Methods

fit(self, X[, y])	Compute the embedding vectors for data X
fit_transform(self, X[, y])	Fit the model from data in X and transform X.
get_params(self[, deep])	Get parameters for this estimator.
reconstruction_error(self)	Compute the reconstruction error for the embedding.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform X.

__init__(*self*, *n_neighbors=5*, *n_components=2*, *eigen_solver='auto'*, *tol=0*, *max_iter=None*, *path_method='auto'*, *neighbors_algorithm='auto'*, *n_jobs=None*, *metric='minkowski'*, *p=2*, *metric_params=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Compute the embedding vectors for data X

Parameters

X [{array-like, sparse graph, BallTree, KDTree, NearestNeighbors}] Sample data, shape = (n_samples, n_features), in the form of a numpy array, sparse graph, precomputed tree, or NearestNeighbors object.

y [Ignored]

Returns

self [returns an instance of self.]

fit_transform(*self*, *X*, *y=None*)
Fit the model from data in X and transform X.

Parameters

X [{array-like, sparse graph, BallTree, KDTree}] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [Ignored]

Returns

X_new [array-like, shape (n_samples, n_components)]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

reconstruction_error(*self*)
Compute the reconstruction error for the embedding.

Returns

reconstruction_error [float]

Notes

The cost function of an isomap embedding is

```
E = frobenius_norm[K(D) - K(D_fit)] / n_samples
```

Where D is the matrix of distances for the input data X, D_fit is the matrix of distances for the output embedding X_fit, and K is the isomap kernel:

```
K(D) = -0.5 * (I - 1/n_samples) * D^2 * (I - 1/n_samples)
```

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Transform X.

This is implemented by linking the points X into the graph of geodesic distances of the training data. First the `n_neighbors` nearest neighbors of X are found in the training data, and from these the shortest geodesic distances from each point in X to each point in the training data are computed in order to construct the kernel. The embedding of X is the projection of this kernel onto the embedding vectors of the training set.

Parameters

X [array-like, shape (n_queries, n_features)] If neighbors_algorithm='precomputed', X is assumed to be a distance matrix or a sparse graph of shape (n_queries, n_samples_fit).

Returns

X_new [array-like, shape (n_queries, n_components)]

Examples using `sklearn.manifold.Isomap`

- *Comparison of Manifold Learning methods*
- *Manifold Learning methods on a severed sphere*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*
- *Release Highlights for scikit-learn 0.22*

7.23.2 sklearn.manifold.LocallyLinearEmbedding

class sklearn.manifold.**LocallyLinearEmbedding**(*n_neighbors=5*, *n_components=2*, *reg=0.001*, *eigen_solver='auto'*, *tol=1e-06*, *max_iter=100*, *method='standard'*, *hessian_tol=0.0001*, *modified_tol=1e-12*, *neighbors_algorithm='auto'*, *random_state=None*, *n_jobs=None*)

Locally Linear Embedding

Read more in the *User Guide*.

Parameters

n_neighbors [integer] number of neighbors to consider for each point.

n_components [integer] number of coordinates for the manifold

reg [float] regularization constant, multiplies the trace of the local covariance matrix of the distances.

eigen_solver [string, {'auto', 'arpack', 'dense'}] auto : algorithm will attempt to choose the best method for input data

arpack [use arnoldi iteration in shift-invert mode.] For this method, M may be a dense matrix, sparse matrix, or general linear operator. Warning: ARPACK can be unstable for some problems. It is best to try several random seeds in order to check results.

dense [use standard dense matrix operations for the eigenvalue] decomposition. For this method, M must be an array or matrix type. This method should be avoided for large problems.

tol [float, optional] Tolerance for 'arpack' method Not used if eigen_solver=='dense'.

max_iter [integer] maximum number of iterations for the arpack solver. Not used if eigen_solver=='dense'.

method [string ('standard', 'hessian', 'modified' or 'ltsa')]

standard [use the standard locally linear embedding algorithm. see] reference [1]

hessian [use the Hessian eigenmap method. This method requires] `n_neighbors > n_components * (1 + (n_components + 1) / 2` see reference [2]

modified [use the modified locally linear embedding algorithm.] see reference [3]

ltsa [use local tangent space alignment algorithm] see reference [4]

hessian_tol [float, optional] Tolerance for Hessian eigenmapping method. Only used if `method == 'hessian'`

modified_tol [float, optional] Tolerance for modified LLE method. Only used if `method == 'modified'`

neighbors_algorithm [string ['auto'|'brute'|'kd_tree'|'ball_tree']] algorithm to use for nearest neighbors search, passed to neighbors.NearestNeighbors instance

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `eigen_solver` == 'arpack'.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

embedding_ [array-like, shape [n_samples, n_components]] Stores the embedding vectors

reconstruction_error_ [float] Reconstruction error associated with `embedding_`

nbrs_ [NearestNeighbors object] Stores nearest neighbors instance, including BallTree or KDtree if applicable.

References

[R62e36dd1b056-1], [R62e36dd1b056-2], [R62e36dd1b056-3], [R62e36dd1b056-4]

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.manifold import LocallyLinearEmbedding
>>> X, _ = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> embedding = LocallyLinearEmbedding(n_components=2)
>>> X_transformed = embedding.fit_transform(X[:100])
>>> X_transformed.shape
(100, 2)
```

Methods

fit(self, X[, y])	Compute the embedding vectors for data X
fit_transform(self, X[, y])	Compute the embedding vectors for data X and transform X.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform new points into embedding space.

__init__(*self*, *n_neighbors=5*, *n_components=2*, *reg=0.001*, *eigen_solver='auto'*, *tol=1e-06*, *max_iter=100*, *method='standard'*, *hessian_tol=0.0001*, *modified_tol=1e-12*, *neighbors_algorithm='auto'*, *random_state=None*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Compute the embedding vectors for data X

Parameters

X [array-like of shape [n_samples, n_features]] training set.

y [Ignored]

Returns

self [returns an instance of self.]

fit_transform(*self*, *X*, *y=None*)
Compute the embedding vectors for data X and transform X.

Parameters

X [array-like of shape [n_samples, n_features]] training set.

y [Ignored]

Returns

X_new [array-like, shape (n_samples, n_components)]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Transform new points into embedding space.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

X_new [array, shape = [n_samples, n_components]]

Notes

Because of scaling performed by this method, it is discouraged to use it together with methods that are not scale-invariant (like SVMs)

Examples using `sklearn.manifold.LocallyLinearEmbedding`

- *Visualizing the stock market structure*
- *Comparison of Manifold Learning methods*
- *Manifold Learning methods on a severed sphere*

- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

7.23.3 sklearn.manifold.MDS

class sklearn.manifold.**MDS**(*n_components=2*, *metric=True*, *n_init=4*, *max_iter=300*, *verbose=0*, *eps=0.001*, *n_jobs=None*, *random_state=None*, *dissimilarity='euclidean'*)

Multidimensional scaling

Read more in the *User Guide*.

Parameters

n_components [int, optional, default: 2] Number of dimensions in which to immerse the dissimilarities.

metric [boolean, optional, default: True] If `True`, perform metric MDS; otherwise, perform nonmetric MDS.

n_init [int, optional, default: 4] Number of times the SMACOF algorithm will be run with different initializations. The final results will be the best output of the runs, determined by the run with the smallest final stress.

max_iter [int, optional, default: 300] Maximum number of iterations of the SMACOF algorithm for a single run.

verbose [int, optional, default: 0] Level of verbosity.

eps [float, optional, default: 1e-3] Relative tolerance with respect to stress at which to declare convergence.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. If multiple initializations are used (`n_init`), each run of the algorithm is computed in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional, default: None] The generator used to initialize the centers. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

dissimilarity ['euclidean' | 'precomputed', optional, default: 'euclidean'] Dissimilarity measure to use:

- **'euclidean':** Pairwise Euclidean distances between points in the dataset.
- **'precomputed':** Pre-computed dissimilarities are passed directly to `fit` and `fit_transform`.

Attributes

embedding_ [array-like, shape (n_samples, n_components)] Stores the position of the dataset in the embedding space.

stress_ [float] The final value of the stress (sum of squared distance of the disparities and the distances for all constrained points).

References

"Modern Multidimensional Scaling - Theory and Applications" Borg, I.; Groenen P. Springer Series in Statistics (1997)

"Nonmetric multidimensional scaling: a numerical method" Kruskal, J. Psychometrika, 29 (1964)

"Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis" Kruskal, J. Psychometrika, 29, (1964)

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.manifold import MDS
>>> X, _ = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> embedding = MDS(n_components=2)
>>> X_transformed = embedding.fit_transform(X[:100])
>>> X_transformed.shape
(100, 2)
```

Methods

fit(self, X[, y, init])	Computes the position of the points in the embedding space
fit_transform(self, X[, y, init])	Fit the data from X, and returns the embedded coordinates
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_components=2*, *metric=True*, *n_init=4*, *max_iter=300*, *verbose=0*, *eps=0.001*, *n_jobs=None*, *random_state=None*, *dissimilarity='euclidean'*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*, *init=None*)
Computes the position of the points in the embedding space

Parameters

X [array, shape (n_samples, n_features) or (n_samples, n_samples)] Input data. If `dissimilarity=='precomputed'`, the input should be the dissimilarity matrix.

y [Ignored]

init [ndarray, shape (n_samples,), optional, default: None] Starting configuration of the embedding to initialize the SMACOF algorithm. By default, the algorithm is initialized with a randomly chosen array.

fit_transform(*self*, *X*, *y=None*, *init=None*)
Fit the data from X, and returns the embedded coordinates

Parameters

X [array, shape (n_samples, n_features) or (n_samples, n_samples)] Input data. If `dissimilarity=='precomputed'`, the input should be the dissimilarity matrix.

y [Ignored]

init [ndarray, shape (n_samples,), optional, default: None] Starting configuration of the embedding to initialize the SMACOF algorithm. By default, the algorithm is initialized with a randomly chosen array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.manifold.MDS`

- *Comparison of Manifold Learning methods*
- *Multi-dimensional scaling*
- *Manifold Learning methods on a severed sphere*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

7.23.4 `sklearn.manifold`.SpectralEmbedding

class sklearn.manifold.**SpectralEmbedding**(*n_components=2*, *affinity='nearest_neighbors'*, *gamma=None*, *random_state=None*, *eigen_solver=None*, *n_neighbors=None*, *n_jobs=None*)
Spectral embedding for non-linear dimensionality reduction.

Forms an affinity matrix given by the specified function and applies spectral decomposition to the corresponding graph laplacian. The resulting transformation is given by the value of the eigenvectors for each data point.

Note : Laplacian Eigenmaps is the actual algorithm implemented here.

Read more in the *User Guide*.

Parameters

n_components [integer, default: 2] The dimension of the projected subspace.

affinity [string or callable, default]

How to construct the affinity matrix.

- 'nearest_neighbors' : construct the affinity matrix by computing a graph of nearest neighbors.
- 'rbf' : construct the affinity matrix by computing a radial basis function (RBF) kernel.
- 'precomputed' : interpret `X` as a precomputed affinity matrix.
- 'precomputed_nearest_neighbors' : interpret `X` as a sparse graph of precomputed nearest neighbors, and constructs the affinity matrix by selecting the `n_neighbors` nearest neighbors.
- callable : use passed in function as affinity the function takes in data matrix (n_samples, n_features) and return affinity matrix (n_samples, n_samples).

gamma [float, optional, default] Kernel coefficient for rbf kernel.

random_state [int, RandomState instance or None, optional, default: None] A pseudo random number generator used for the initialization of the lobpcg eigenvectors. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'amg'.

eigen_solver [{None, 'arpack', 'lobpcg', or 'amg'}] The eigenvalue decomposition strategy to use. AMG requires pyamg to be installed. It can be faster on very large, sparse problems.

n_neighbors [int, default] Number of nearest neighbors for nearest_neighbors graph building.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

embedding_ [array, shape = (n_samples, n_components)] Spectral embedding of the training matrix.

affinity_matrix_ [array, shape = (n_samples, n_samples)] Affinity_matrix constructed from samples or precomputed.

n_neighbors_ [int] Number of nearest neighbors effectively used.

References

- A Tutorial on Spectral Clustering, 2007 Ulrike von Luxburg http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.165.9323
- On Spectral Clustering: Analysis and an algorithm, 2001 Andrew Y. Ng, Michael I. Jordan, Yair Weiss http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.8100
- Normalized cuts and image segmentation, 2000 Jianbo Shi, Jitendra Malik http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.160.2324

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.manifold import SpectralEmbedding
>>> X, _ = load_digits(return_X_y=True)
>>> X.shape
```

(continues on next page)

(continued from previous page)

```
(1797, 64)
>>> embedding = SpectralEmbedding(n_components=2)
>>> X_transformed = embedding.fit_transform(X[:100])
>>> X_transformed.shape
(100, 2)
```

Methods

fit(self, X[, y])	Fit the model from data in X.
fit_transform(self, X[, y])	Fit the model from data in X and transform X.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_components=2*, *affinity='nearest_neighbors'*, *gamma=None*, *random_state=None*, *eigen_solver=None*, *n_neighbors=None*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the model from data in X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

If affinity is "precomputed" X : {array-like, sparse matrix}, shape (n_samples, n_samples), Interpret X as precomputed adjacency graph computed from samples.

Returns

self [object] Returns the instance itself.

fit_transform(*self*, *X*, *y=None*)
Fit the model from data in X and transform X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

If affinity is "precomputed" X : {array-like, sparse matrix}, shape (n_samples, n_samples), Interpret X as precomputed adjacency graph computed from samples.

Returns

X_new [array-like, shape (n_samples, n_components)]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.manifold.SpectralEmbedding`

- *Various Agglomerative Clustering on a 2D embedding of digits*
- *Comparison of Manifold Learning methods*
- *Manifold Learning methods on a severed sphere*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

7.23.5 sklearn.manifold.TSNE

class sklearn.manifold.**TSNE**(*n_components=2*, *perplexity=30.0*, *early_exaggeration=12.0*, *learning_rate=200.0*, *n_iter=1000*, *n_iter_without_progress=300*, *min_grad_norm=1e-07*, *metric='euclidean'*, *init='random'*, *verbose=0*, *random_state=None*, *method='barnes_hut'*, *angle=0.5*, *n_jobs=None*)
t-distributed Stochastic Neighbor Embedding.

t-SNE [1] is a tool to visualize high-dimensional data. It converts similarities between data points to joint probabilities and tries to minimize the Kullback-Leibler divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data. t-SNE has a cost function that is not convex, i.e. with different initializations we can get different results.

It is highly recommended to use another dimensionality reduction method (e.g. PCA for dense data or TruncatedSVD for sparse data) to reduce the number of dimensions to a reasonable amount (e.g. 50) if the number of features is very high. This will suppress some noise and speed up the computation of pairwise distances between samples. For more tips see Laurens van der Maaten's FAQ [2].

Read more in the *User Guide*.

Parameters

n_components [int, optional (default: 2)] Dimension of the embedded space.

perplexity [float, optional (default: 30)] The perplexity is related to the number of nearest neighbors that is used in other manifold learning algorithms. Larger datasets usually require a larger perplexity. Consider selecting a value between 5 and 50. Different values can result in significanlty different results.

early_exaggeration [float, optional (default: 12.0)] Controls how tight natural clusters in the original space are in the embedded space and how much space will be between them. For larger values, the space between natural clusters will be larger in the embedded space. Again, the choice of this parameter is not very critical. If the cost function increases during initial optimization, the early exaggeration factor or the learning rate might be too high.

learning_rate [float, optional (default: 200.0)] The learning rate for t-SNE is usually in the range [10.0, 1000.0]. If the learning rate is too high, the data may look like a 'ball' with any point approximately equidistant from its nearest neighbours. If the learning rate is too low, most points may look compressed in a dense cloud with few outliers. If the cost function gets stuck in a bad local minimum increasing the learning rate may help.

n_iter [int, optional (default: 1000)] Maximum number of iterations for the optimization. Should be at least 250.

n_iter_without_progress [int, optional (default: 300)] Maximum number of iterations without progress before we abort the optimization, used after 250 initial iterations with early exaggeration. Note that progress is only checked every 50 iterations so this value is rounded to the next multiple of 50.

New in version 0.17: parameter *n_iter_without_progress* to control stopping criteria.

min_grad_norm [float, optional (default: 1e-7)] If the gradient norm is below this threshold, the optimization will be stopped.

metric [string or callable, optional] The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by scipy.spatial.distance.pdist for its metric parameter, or a metric listed in pairwise.PAIRWISE_DISTANCE_FUNCTIONS. If metric is "precomputed", X is assumed to be a distance matrix. Alternatively, if metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays from X as input and return a value indicating the distance between them. The default is "euclidean" which is interpreted as squared euclidean distance.

init [string or numpy array, optional (default: "random")] Initialization of embedding. Possible options are 'random', 'pca', and a numpy array of shape (n_samples, n_components). PCA initialization cannot be used with precomputed distances and is usually more globally stable than random initialization.

verbose [int, optional (default: 0)] Verbosity level.

random_state [int, RandomState instance or None, optional (default: None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Note that different initializations might result in different local minima of the cost function.

method [string (default: 'barnes_hut')] By default the gradient calculation algorithm uses Barnes-Hut approximation running in O(NlogN) time. method='exact' will run on the slower, but exact, algorithm in O(N^2) time. The exact algorithm should be used when nearest-neighbor errors need to be better than 3%. However, the exact method cannot scale to millions of examples.

New in version 0.17: Approximate optimization *method* via the Barnes-Hut.

angle [float (default: 0.5)] Only used if method='barnes_hut' This is the trade-off between speed and accuracy for Barnes-Hut T-SNE. 'angle' is the angular size (referred to as theta in [3]) of a distant node as measured from a point. If this size is below 'angle' then it is used as a summary node of all points contained within it. This method is not very sensitive to changes in this parameter in the range of 0.2 - 0.8. Angle less than 0.2 has quickly increasing computation time and angle greater 0.8 has quickly increasing error.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run for neighbors search. This parameter has no impact when `metric="precomputed"` or (`metric="euclidean"` and `method="exact"`). `None` means 1 unless in a

joblib.parallel_backend context. -1 means using all processors. See *Glossary* for more details.

New in version 0.22.

Attributes

embedding_ [array-like, shape (n_samples, n_components)] Stores the embedding vectors.

kl_divergence_ [float] Kullback-Leibler divergence after optimization.

n_iter_ [int] Number of iterations run.

References

[1] van der Maaten, L.J.P.; Hinton, G.E. Visualizing High-Dimensional Data Using t-SNE. Journal of Machine Learning Research 9:2579-2605, 2008.

[2] van der Maaten, L.J.P. t-Distributed Stochastic Neighbor Embedding https://lvdmaaten.github.io/tsne/

[3] L.J.P. van der Maaten. Accelerating t-SNE using Tree-Based Algorithms. Journal of Machine Learning Research 15(Oct):3221-3245, 2014. https://lvdmaaten.github.io/publications/papers/JMLR_2014.pdf

Examples

```
>>> import numpy as np
>>> from sklearn.manifold import TSNE
>>> X = np.array([[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 1]])
>>> X_embedded = TSNE(n_components=2).fit_transform(X)
>>> X_embedded.shape
(4, 2)
```

Methods

fit(self, X[, y])	Fit X into an embedded space.
fit_transform(self, X[, y])	Fit X into an embedded space and return that transformed output.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_components=2*, *perplexity=30.0*, *early_exaggeration=12.0*, *learning_rate=200.0*, *n_iter=1000*, *n_iter_without_progress=300*, *min_grad_norm=1e-07*, *metric='euclidean'*, *init='random'*, *verbose=0*, *random_state=None*, *method='barnes_hut'*, *angle=0.5*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit X into an embedded space.

Parameters

X [array, shape (n_samples, n_features) or (n_samples, n_samples)] If the metric is 'precomputed' X must be a square distance matrix. Otherwise it contains a sample per row. If the method is 'exact', X may be a sparse matrix of type 'csr', 'csc' or 'coo'. If the method is 'barnes_hut' and the metric is 'precomputed', X may be a precomputed sparse graph.

y [Ignored]

fit_transform(*self*, *X*, *y=None*)
Fit X into an embedded space and return that transformed output.

Parameters

X [array, shape (n_samples, n_features) or (n_samples, n_samples)] If the metric is 'precomputed' X must be a square distance matrix. Otherwise it contains a sample per row. If the method is 'exact', X may be a sparse matrix of type 'csr', 'csc' or 'coo'. If the method is 'barnes_hut' and the metric is 'precomputed', X may be a precomputed sparse graph.

y [Ignored]

Returns

X_new [array, shape (n_samples, n_components)] Embedding of the training data in low-dimensional space.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.manifold.TSNE`

- *Approximate nearest neighbors in TSNE*

manifold.locally_linear_embedding(X, ...[, ...])	Perform a Locally Linear Embedding analysis on the data.
manifold.smacof(dissimilarities[, metric, ...])	Computes multidimensional scaling using the SMACOF algorithm.
manifold.spectral_embedding(adjacency[, ...])	Project the sample on the first eigenvectors of the graph Laplacian.
manifold.trustworthiness(X, X_embedded[, ...])	Expresses to what extent the local structure is retained.

7.23.6 sklearn.manifold.locally_linear_embedding

sklearn.manifold.**locally_linear_embedding**(*X*, *n_neighbors*, *n_components*, *reg=0.001*, *eigen_solver='auto'*, *tol=1e-06*, *max_iter=100*, *method='standard'*, *hessian_tol=0.0001*, *modified_tol=1e-12*, *random_state=None*, *n_jobs=None*)

Perform a Locally Linear Embedding analysis on the data.

Read more in the *User Guide*.

Parameters

X [{array-like, NearestNeighbors}] Sample data, shape = (n_samples, n_features), in the form of a numpy array or a NearestNeighbors object.

n_neighbors [integer] number of neighbors to consider for each point.

n_components [integer] number of coordinates for the manifold.

reg [float] regularization constant, multiplies the trace of the local covariance matrix of the distances.

eigen_solver [string, {'auto', 'arpack', 'dense'}] auto : algorithm will attempt to choose the best method for input data

arpack [use arnoldi iteration in shift-invert mode.] For this method, M may be a dense matrix, sparse matrix, or general linear operator. Warning: ARPACK can be unstable for some problems. It is best to try several random seeds in order to check results.

dense [use standard dense matrix operations for the eigenvalue] decomposition. For this method, M must be an array or matrix type. This method should be avoided for large problems.

tol [float, optional] Tolerance for 'arpack' method Not used if eigen_solver=='dense'.

max_iter [integer] maximum number of iterations for the arpack solver.

method [{'standard', 'hessian', 'modified', 'ltsa'}]

standard [use the standard locally linear embedding algorithm.] see reference [1]

hessian [use the Hessian eigenmap method. This method requires] n_neighbors > n_components * (1 + (n_components + 1) / 2. see reference [2]

modified [use the modified locally linear embedding algorithm.] see reference [3]

ltsa [use local tangent space alignment algorithm] see reference [4]

hessian_tol [float, optional] Tolerance for Hessian eigenmapping method. Only used if method == 'hessian'

modified_tol [float, optional] Tolerance for modified LLE method. Only used if method == 'modified'

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'arpack'.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Returns

Y [array-like, shape [n_samples, n_components]] Embedding vectors.

squared_error [float] Reconstruction error for the embedding vectors. Equivalent to `norm(Y - W Y, 'fro')**2`, where W are the reconstruction weights.

References

[1], [2], [3], [4]

Examples using `sklearn.manifold.locally_linear_embedding`

- *Swiss Roll reduction with LLE*

7.23.7 `sklearn.manifold`.smacof

`sklearn.manifold.`**`smacof`**(*dissimilarities*, *metric=True*, *n_components=2*, *init=None*, *n_init=8*, *n_jobs=None*, *max_iter=300*, *verbose=0*, *eps=0.001*, *random_state=None*, *return_n_iter=False*)

Computes multidimensional scaling using the SMACOF algorithm.

The SMACOF (Scaling by MAjorizing a COmplicated Function) algorithm is a multidimensional scaling algorithm which minimizes an objective function (the *stress*) using a majorization technique. Stress majorization, also known as the Guttman Transform, guarantees a monotone convergence of stress, and is more powerful than traditional techniques such as gradient descent.

The SMACOF algorithm for metric MDS can summarized by the following steps:

1. Set an initial start configuration, randomly or not.
2. Compute the stress
3. Compute the Guttman Transform
4. Iterate 2 and 3 until convergence.

The nonmetric algorithm adds a monotonic regression step before computing the stress.

Parameters

dissimilarities [ndarray, shape (n_samples, n_samples)] Pairwise dissimilarities between the points. Must be symmetric.

metric [boolean, optional, default: True] Compute metric or nonmetric SMACOF algorithm.

n_components [int, optional, default: 2] Number of dimensions in which to immerse the dissimilarities. If an `init` array is provided, this option is overridden and the shape of `init` is used to determine the dimensionality of the embedding space.

init [ndarray, shape (n_samples, n_components), optional, default: None] Starting configuration of the embedding to initialize the algorithm. By default, the algorithm is initialized with a randomly chosen array.

n_init [int, optional, default: 8] Number of times the SMACOF algorithm will be run with different initializations. The final results will be the best output of the runs, determined by the run with the smallest final stress. If `init` is provided, this option is overridden and a single run is performed.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. If multiple initializations are used (`n_init`), each run of the algorithm is computed in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

max_iter [int, optional, default: 300] Maximum number of iterations of the SMACOF algorithm for a single run.

verbose [int, optional, default: 0] Level of verbosity.

eps [float, optional, default: 1e-3] Relative tolerance with respect to stress at which to declare convergence.

random_state [int, RandomState instance or None, optional, default: None] The generator used to initialize the centers. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

return_n_iter [bool, optional, default: False] Whether or not to return the number of iterations.

Returns

X [ndarray, shape (n_samples, n_components)] Coordinates of the points in a `n_components`-space.

stress [float] The final value of the stress (sum of squared distance of the disparities and the distances for all constrained points).

n_iter [int] The number of iterations corresponding to the best stress. Returned only if `return_n_iter` is set to `True`.

Notes

"Modern Multidimensional Scaling - Theory and Applications" Borg, I.; Groenen P. Springer Series in Statistics (1997)

"Nonmetric multidimensional scaling: a numerical method" Kruskal, J. Psychometrika, 29 (1964)

"Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis" Kruskal, J. Psychometrika, 29, (1964)

7.23.8 sklearn.manifold.spectral_embedding

sklearn.manifold.**spectral_embedding**(*adjacency*, *n_components=8*, *eigen_solver=None*, *random_state=None*, *eigen_tol=0.0*, *norm_laplacian=True*, *drop_first=True*)

Project the sample on the first eigenvectors of the graph Laplacian.

The adjacency matrix is used to compute a normalized graph Laplacian whose spectrum (especially the eigenvectors associated to the smallest eigenvalues) has an interpretation in terms of minimal number of cuts necessary to split the graph into comparably sized components.

This embedding can also 'work' even if the `adjacency` variable is not strictly the adjacency matrix of a graph but more generally an affinity or similarity matrix between samples (for instance the heat kernel of a euclidean distance matrix or a k-NN matrix).

However care must taken to always make the affinity matrix symmetric so that the eigenvector decomposition works as expected.

Note : Laplacian Eigenmaps is the actual algorithm implemented here.

Read more in the *User Guide*.

Parameters

adjacency [array-like or sparse graph, shape: (n_samples, n_samples)] The adjacency matrix of the graph to embed.

n_components [integer, optional, default 8] The dimension of the projection subspace.

eigen_solver [{None, 'arpack', 'lobpcg', or 'amg'}, default None] The eigenvalue decomposition strategy to use. AMG requires pyamg to be installed. It can be faster on very large, sparse problems, but may also lead to instabilities.

random_state [int, RandomState instance or None, optional, default: None] A pseudo random number generator used for the initialization of the lobpcg eigenvectors decomposition. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'amg'.

eigen_tol [float, optional, default=0.0] Stopping criterion for eigendecomposition of the Laplacian matrix when using arpack eigen_solver.

norm_laplacian [bool, optional, default=True] If True, then compute normalized Laplacian.

drop_first [bool, optional, default=True] Whether to drop the first eigenvector. For spectral embedding, this should be True as the first eigenvector should be constant vector for connected graph, but for spectral clustering, this should be kept as False to retain the first eigenvector.

Returns

embedding [array, shape=(n_samples, n_components)] The reduced samples.

Notes

Spectral Embedding (Laplacian Eigenmaps) is most useful when the graph has one connected component. If there graph has many components, the first few eigenvectors will simply uncover the connected components of the graph.

References

- https://en.wikipedia.org/wiki/LOBPCG
- Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method Andrew V. Knyazev https://doi.org/10.1137%2FS1064827500366124

7.23.9 `sklearn.manifold`.trustworthiness

`sklearn.manifold.`**`trustworthiness`**(*X*, *X_embedded*, *n_neighbors=5*, *metric='euclidean'*)

Expresses to what extent the local structure is retained.

The trustworthiness is within [0, 1]. It is defined as

$$T(k) = 1 - \frac{2}{nk(2n - 3k - 1)} \sum_{i=1}^{n} \sum_{j \in \mathcal{N}_i^k} \max(0, (r(i, j) - k))$$

where for each sample i, $\mathcal{N}_i^k$ are its k nearest neighbors in the output space, and every sample j is its $r(i, j)$-th nearest neighbor in the input space. In other words, any unexpected nearest neighbors in the output space are penalised in proportion to their rank in the input space.

- "Neighborhood Preservation in Nonlinear Projection Methods: An Experimental Study" J. Venna, S. Kaski
- "Learning a Parametric Embedding by Preserving Local Structure" L.J.P. van der Maaten

Parameters

X [array, shape (n_samples, n_features) or (n_samples, n_samples)] If the metric is 'precomputed' X must be a square distance matrix. Otherwise it contains a sample per row.

X_embedded [array, shape (n_samples, n_components)] Embedding of the training data in low-dimensional space.

n_neighbors [int, optional (default: 5)] Number of neighbors k that will be considered.

metric [string, or callable, optional, default 'euclidean'] Which metric to use for computing pairwise distances between samples from the original input space. If metric is 'precomputed', X must be a matrix of pairwise distances or squared distances. Otherwise, see the documentation of argument metric in sklearn.pairwise.pairwise_distances for a list of available metrics.

Returns

trustworthiness [float] Trustworthiness of the low-dimensional embedding.

7.24 sklearn.metrics: Metrics

See the *Metrics and scoring: quantifying the quality of predictions* section and the *Pairwise metrics, Affinities and Kernels* section of the user guide for further details.

The `sklearn.metrics` module includes score functions, performance metrics and pairwise metrics and distance computations.

7.24.1 Model Selection Interface

See the *The scoring parameter: defining model evaluation rules* section of the user guide for further details.

`metrics.check_scoring`(estimator[, scoring, ...])	Determine scorer from user options.
`metrics.get_scorer`(scoring)	Get a scorer from string.
`metrics.make_scorer`(score_func[, ...])	Make a scorer from a performance metric or loss function.

sklearn.metrics.check_scoring

sklearn.metrics.**check_scoring**(*estimator*, *scoring=None*, *allow_none=False*)

Determine scorer from user options.

A TypeError will be thrown if the estimator cannot be scored.

Parameters

estimator [estimator object implementing 'fit'] The object to use to fit the data.

scoring [string, callable or None, optional, default: None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`.

allow_none [boolean, optional, default: False] If no scoring is specified and the estimator has no score function, we can either return None or raise an exception.

Returns

scoring [callable] A scorer callable object / function with signature `scorer(estimator, X, y)`.

sklearn.metrics.get_scorer

sklearn.metrics.**get_scorer**(*scoring*)

Get a scorer from string.

Read more in the *User Guide*.

Parameters

scoring [str | callable] scoring method as string. If callable it is returned as is.

Returns

scorer [callable] The scorer.

sklearn.metrics.make_scorer

sklearn.metrics.**make_scorer**(*score_func*, *greater_is_better=True*, *needs_proba=False*, *needs_threshold=False*, ***kwargs*)

Make a scorer from a performance metric or loss function.

This factory function wraps scoring functions for use in GridSearchCV and cross_val_score. It takes a score function, such as `accuracy_score`, `mean_squared_error`, `adjusted_rand_index` or `average_precision` and returns a callable that scores an estimator's output.

Read more in the *User Guide*.

Parameters

score_func [callable,] Score function (or loss function) with signature `score_func(y, y_pred, **kwargs)`.

greater_is_better [boolean, default=True] Whether score_func is a score function (default), meaning high is good, or a loss function, meaning low is good. In the latter case, the scorer object will sign-flip the outcome of the score_func.

needs_proba [boolean, default=False] Whether score_func requires predict_proba to get probability estimates out of a classifier.

If True, for binary `y_true`, the score function is supposed to accept a 1D `y_pred` (i.e., probability of the positive class, shape `(n_samples,)`).

needs_threshold [boolean, default=False] Whether score_func takes a continuous decision certainty. This only works for binary classification using estimators that have either a decision_function or predict_proba method.

If True, for binary `y_true`, the score function is supposed to accept a 1D `y_pred` (i.e., probability of the positive class or the decision function, shape `(n_samples,)`).

For example `average_precision` or the area under the roc curve can not be computed using discrete predictions alone.

****kwargs** [additional arguments] Additional parameters to be passed to score_func.

Returns

scorer [callable] Callable object that returns a scalar score; greater is better.

Notes

If `needs_proba=False` and `needs_threshold=False`, the score function is supposed to accept the output of *predict*. If `needs_proba=True`, the score function is supposed to accept the output of *predict_proba* (For binary `y_true`, the score function is supposed to accept probability of the positive class). If `needs_threshold=True`, the score function is supposed to accept the output of *decision_function*.

Examples

```
>>> from sklearn.metrics import fbeta_score, make_scorer
>>> ftwo_scorer = make_scorer(fbeta_score, beta=2)
>>> ftwo_scorer
make_scorer(fbeta_score, beta=2)
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.svm import LinearSVC
>>> grid = GridSearchCV(LinearSVC(), param_grid={'C': [1, 10]},
...                     scoring=ftwo_scorer)
```

Examples using `sklearn.metrics.make_scorer`

- *Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV*

7.24.2 Classification metrics

See the *Classification metrics* section of the user guide for further details.

metrics.accuracy_score(y_true, y_pred[, ...])	Accuracy classification score.
metrics.auc(x, y)	Compute Area Under the Curve (AUC) using the trapezoidal rule
metrics.average_precision_score(y_true, y_score)	Compute average precision (AP) from prediction scores
metrics.balanced_accuracy_score(y_true, y_pred)	Compute the balanced accuracy
metrics.brier_score_loss(y_true, y_prob[, ...])	Compute the Brier score.
metrics.classification_report(y_true, y_pred)	Build a text report showing the main classification metrics
metrics.cohen_kappa_score(y1, y2[, labels, ...])	Cohen's kappa: a statistic that measures inter-annotator agreement.
metrics.confusion_matrix(y_true, y_pred[, ...])	Compute confusion matrix to evaluate the accuracy of a classification.

Continued on next page

Table 189 – continued from previous page

metrics.dcg_score(y_true, y_score[, k, ...])	Compute Discounted Cumulative Gain.
metrics.f1_score(y_true, y_pred[, labels, ...])	Compute the F1 score, also known as balanced F-score or F-measure
metrics.fbeta_score(y_true, y_pred, beta[, ...])	Compute the F-beta score
metrics.hamming_loss(y_true, y_pred[, ...])	Compute the average Hamming loss.
metrics.hinge_loss(y_true, pred_decision[, ...])	Average hinge loss (non-regularized)
metrics.jaccard_score(y_true, y_pred[, ...])	Jaccard similarity coefficient score
metrics.log_loss(y_true, y_pred[, eps, ...])	Log loss, aka logistic loss or cross-entropy loss.
metrics.matthews_corrcoef(y_true, y_pred[, ...])	Compute the Matthews correlation coefficient (MCC)
metrics.multilabel_confusion_matrix(y_true, ...)	Compute a confusion matrix for each class or sample
metrics.ndcg_score(y_true, y_score[, k, ...])	Compute Normalized Discounted Cumulative Gain.
metrics.precision_recall_curve(y_true, ...)	Compute precision-recall pairs for different probability thresholds
metrics.precision_recall_fscore_support(...)	Compute precision, recall, F-measure and support for each class
metrics.precision_score(y_true, y_pred[, ...])	Compute the precision
metrics.recall_score(y_true, y_pred[, ...])	Compute the recall
metrics.roc_auc_score(y_true, y_score[, ...])	Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.
metrics.roc_curve(y_true, y_score[, ...])	Compute Receiver operating characteristic (ROC)
metrics.zero_one_loss(y_true, y_pred[, ...])	Zero-one classification loss.

sklearn.metrics.accuracy_score

sklearn.metrics.**accuracy_score**(*y_true*, *y_pred*, *normalize=True*, *sample_weight=None*)

Accuracy classification score.

In multilabel classification, this function computes subset accuracy: the set of labels predicted for a sample must *exactly* match the corresponding set of labels in y_true.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) labels.

y_pred [1d array-like, or label indicator array / sparse matrix] Predicted labels, as returned by a classifier.

normalize [bool, optional (default=True)] If `False`, return the number of correctly classified samples. Otherwise, return the fraction of correctly classified samples.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] If `normalize == True`, return the fraction of correctly classified samples (float), else returns the number of correctly classified samples (int).

The best performance is 1 with `normalize == True` and the number of samples with `normalize == False`.

See also:

jaccard_score, *hamming_loss*, *zero_one_loss*

Notes

In binary and multiclass classification, this function is equal to the `jaccard_score` function.

Examples

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]
>>> accuracy_score(y_true, y_pred)
0.5
>>> accuracy_score(y_true, y_pred, normalize=False)
2
```

In the multilabel case with binary label indicators:

```
>>> import numpy as np
>>> accuracy_score(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5
```

Examples using `sklearn.metrics.accuracy_score`

- *Plot classification probability*
- *Multi-class AdaBoosted Decision Trees*
- *Probabilistic predictions with Gaussian process classification (GPC)*
- *Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV*
- *Importance of Feature Scaling*
- *Classification of text documents using sparse features*

sklearn.metrics.auc

sklearn.metrics.**auc**(*x*, *y*)

Compute Area Under the Curve (AUC) using the trapezoidal rule

This is a general function, given points on a curve. For computing the area under the ROC-curve, see *roc_auc_score*. For an alternative way to summarize a precision-recall curve, see *average_precision_score*.

Parameters

x [array, shape = [n]] x coordinates. These must be either monotonic increasing or monotonic decreasing.

y [array, shape = [n]] y coordinates.

Returns

auc [float]

See also:

roc_auc_score Compute the area under the ROC curve

average_precision_score Compute average precision from prediction scores

precision_recall_curve Compute precision-recall pairs for different probability thresholds

Examples

```
>>> import numpy as np
>>> from sklearn import metrics
>>> y = np.array([1, 1, 2, 2])
>>> pred = np.array([0.1, 0.4, 0.35, 0.8])
>>> fpr, tpr, thresholds = metrics.roc_curve(y, pred, pos_label=2)
>>> metrics.auc(fpr, tpr)
0.75
```

Examples using `sklearn.metrics.auc`

- *Species distribution modeling*
- *Receiver Operating Characteristic (ROC) with cross validation*
- *Receiver Operating Characteristic (ROC)*
- *Precision-Recall*

`sklearn.metrics`.average_precision_score

`sklearn.metrics.`**`average_precision_score`**(*y_true*, *y_score*, *average='macro'*, *pos_label=1*, *sample_weight=None*)

Compute average precision (AP) from prediction scores

AP summarizes a precision-recall curve as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight:

$$\text{AP} = \sum_n (R_n - R_{n-1}) P_n$$

where P_n and R_n are the precision and recall at the nth threshold [1]. This implementation is not interpolated and is different from computing the area under the precision-recall curve with the trapezoidal rule, which uses linear interpolation and can be too optimistic.

Note: this implementation is restricted to the binary classification task or multilabel classification task.

Read more in the *User Guide*.

Parameters

y_true [array, shape = [n_samples] or [n_samples, n_classes]] True binary labels or binary label indicators.

y_score [array, shape = [n_samples] or [n_samples, n_classes]] Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).

average [string, [None, 'micro', 'macro' (default), 'samples', 'weighted']] If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

'micro': Calculate metrics globally by considering each element of the label indicator matrix as a label.

'macro': Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

'weighted': Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label).

'samples': Calculate metrics for each instance, and find their average.

Will be ignored when `y_true` is binary.

pos_label [int or str (default=1)] The label of the positive class. Only applied to binary `y_true`. For multilabel-indicator `y_true`, `pos_label` is fixed to 1.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

average_precision [float]

See also:

roc_auc_score Compute the area under the ROC curve

precision_recall_curve Compute precision-recall pairs for different probability thresholds

Notes

Changed in version 0.19: Instead of linearly interpolating between operating points, precisions are weighted by the change in recall since the last operating point.

References

[1]

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import average_precision_score
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> average_precision_score(y_true, y_scores)
0.83...
```

Examples using `sklearn.metrics.average_precision_score`

- *Precision-Recall*

sklearn.metrics.balanced_accuracy_score

sklearn.metrics.**balanced_accuracy_score**(*y_true*, *y_pred*, *sample_weight=None*, *adjusted=False*)

Compute the balanced accuracy

The balanced accuracy in binary and multiclass classification problems to deal with imbalanced datasets. It is defined as the average of recall obtained on each class.

The best value is 1 and the worst value is 0 when `adjusted=False`.

Read more in the *User Guide*.

Parameters

y_true [1d array-like] Ground truth (correct) target values.

y_pred [1d array-like] Estimated targets as returned by a classifier.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

adjusted [bool, default=False] When true, the result is adjusted for chance, so that random performance would score 0, and perfect performance scores 1.

Returns

balanced_accuracy [float]

See also:

recall_score, ***roc_auc_score***

Notes

Some literature promotes alternative definitions of balanced accuracy. Our definition is equivalent to *accuracy_score* with class-balanced sample weights, and shares desirable properties with the binary case. See the *User Guide*.

References

[1], [2]

Examples

```
>>> from sklearn.metrics import balanced_accuracy_score
>>> y_true = [0, 1, 0, 0, 1, 0]
>>> y_pred = [0, 1, 0, 0, 0, 1]
>>> balanced_accuracy_score(y_true, y_pred)
0.625
```

sklearn.metrics.brier_score_loss

sklearn.metrics.**brier_score_loss**(*y_true*, *y_prob*, *sample_weight=None*, *pos_label=None*)

Compute the Brier score. The smaller the Brier score, the better, hence the naming with "loss". Across all items in a set N predictions, the Brier score measures the mean squared difference between (1) the predicted probability assigned to the possible outcomes for item i, and (2) the actual outcome. Therefore, the lower

the Brier score is for a set of predictions, the better the predictions are calibrated. Note that the Brier score always takes on a value between zero and one, since this is the largest possible difference between a predicted probability (which must be between zero and one) and the actual outcome (which can take on values of only 0 and 1). The Brier loss is composed of refinement loss and calibration loss. The Brier score is appropriate for binary and categorical outcomes that can be structured as true or false, but is inappropriate for ordinal variables which can take on three or more values (this is because the Brier score assumes that all possible outcomes are equivalently "distant" from one another). Which label is considered to be the positive label is controlled via the parameter pos_label, which defaults to 1. Read more in the *User Guide*.

Parameters

y_true [array, shape (n_samples,)] True targets.

y_prob [array, shape (n_samples,)] Probabilities of the positive class.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

pos_label [int or str, default=None] Label of the positive class. Defaults to the greater label unless y_true is all 0 or all -1 in which case pos_label defaults to 1.

Returns

score [float] Brier score

References

[1]

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import brier_score_loss
>>> y_true = np.array([0, 1, 1, 0])
>>> y_true_categorical = np.array(["spam", "ham", "ham", "spam"])
>>> y_prob = np.array([0.1, 0.9, 0.8, 0.3])
>>> brier_score_loss(y_true, y_prob)
0.037...
>>> brier_score_loss(y_true, 1-y_prob, pos_label=0)
0.037...
>>> brier_score_loss(y_true_categorical, y_prob, pos_label="ham")
0.037...
>>> brier_score_loss(y_true, np.array(y_prob) > 0.5)
0.0
```

Examples using `sklearn.metrics.brier_score_loss`

- *Probability Calibration curves*
- *Probability calibration of classifiers*

`sklearn.metrics`.classification_report

`sklearn.metrics.`**`classification_report`**(*y_true*, *y_pred*, *labels=None*, *target_names=None*, *sample_weight=None*, *digits=2*, *output_dict=False*, *zero_division='warn'*)

Build a text report showing the main classification metrics

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) target values.

y_pred [1d array-like, or label indicator array / sparse matrix] Estimated targets as returned by a classifier.

labels [array, shape = [n_labels]] Optional list of label indices to include in the report.

target_names [list of strings] Optional display names matching the labels (same order).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

digits [int] Number of digits for formatting output floating point values. When `output_dict` is `True`, this will be ignored and the returned values will not be rounded.

output_dict [bool (default = False)] If True, return output as dict

zero_division ["warn", 0 or 1, default="warn"] Sets the value to return when there is a zero division. If set to "warn", this acts as 0, but warnings are also raised.

Returns

report [string / dict] Text summary of the precision, recall, F1 score for each class. Dictionary returned if output_dict is True. Dictionary has the following structure:

```
{'label 1': {'precision':0.5,
             'recall':1.0,
             'f1-score':0.67,
             'support':1},
 'label 2': { ... },
  ...
}
```

The reported averages include macro average (averaging the unweighted mean per label), weighted average (averaging the support-weighted mean per label), and sample average (only for multilabel classification). Micro average (averaging the total true positives, false negatives and false positives) is only shown for multi-label or multi-class with a subset of classes, because it corresponds to accuracy otherwise. See also *precision_recall_fscore_support* for more details on averages.

Note that in binary classification, recall of the positive class is also known as "sensitivity"; recall of the negative class is "specificity".

See also:

precision_recall_fscore_support, ***confusion_matrix***

multilabel_confusion_matrix

Examples

```
>>> from sklearn.metrics import classification_report
>>> y_true = [0, 1, 2, 2, 2]
>>> y_pred = [0, 0, 2, 2, 1]
>>> target_names = ['class 0', 'class 1', 'class 2']
>>> print(classification_report(y_true, y_pred, target_names=target_names))
```

(continues on next page)

(continued from previous page)

```
              precision    recall  f1-score   support
<BLANKLINE>
     class 0       0.50      1.00      0.67         1
     class 1       0.00      0.00      0.00         1
     class 2       1.00      0.67      0.80         3
<BLANKLINE>
    accuracy                           0.60         5
   macro avg       0.50      0.56      0.49         5
weighted avg       0.70      0.60      0.61         5
<BLANKLINE>
>>> y_pred = [1, 1, 0]
>>> y_true = [1, 1, 1]
>>> print(classification_report(y_true, y_pred, labels=[1, 2, 3]))
              precision    recall  f1-score   support
<BLANKLINE>
           1       1.00      0.67      0.80         3
           2       0.00      0.00      0.00         0
           3       0.00      0.00      0.00         0
<BLANKLINE>
   micro avg       1.00      0.67      0.80         3
   macro avg       0.33      0.22      0.27         3
weighted avg       1.00      0.67      0.80         3
<BLANKLINE>
```

Examples using `sklearn.metrics.classification_report`

- *Recognizing hand-written digits*
- *Faces recognition example using eigenfaces and SVMs*
- *Pipeline Anova SVM*
- *Parameter estimation using grid search with cross-validation*
- *Restricted Boltzmann Machine features for digit classification*
- *Column Transformer with Heterogeneous Data Sources*
- *Label Propagation digits: Demonstrating performance*
- *Label Propagation digits active learning*
- *Classification of text documents using sparse features*

`sklearn.metrics.cohen_kappa_score`

sklearn.metrics.**cohen_kappa_score**(*y1*, *y2*, *labels=None*, *weights=None*, *sample_weight=None*)

Cohen's kappa: a statistic that measures inter-annotator agreement.

This function computes Cohen's kappa [1], a score that expresses the level of agreement between two annotators on a classification problem. It is defined as

$$\kappa = (p_o - p_e)/(1 - p_e)$$

where p_o is the empirical probability of agreement on the label assigned to any sample (the observed agreement ratio), and p_e is the expected agreement when both annotators assign labels randomly. p_e is estimated using a per-annotator empirical prior over the class labels [2].

Read more in the *User Guide*.

Parameters

y1 [array, shape = [n_samples]] Labels assigned by the first annotator.

y2 [array, shape = [n_samples]] Labels assigned by the second annotator. The kappa statistic is symmetric, so swapping `y1` and `y2` doesn't change the value.

labels [array, shape = [n_classes], optional] List of labels to index the matrix. This may be used to select a subset of labels. If None, all labels that appear at least once in `y1` or `y2` are used.

weights [str, optional] Weighting type to calculate the score. None means no weighted; "linear" means linear weighted; "quadratic" means quadratic weighted.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

kappa [float] The kappa statistic, which is a number between -1 and 1. The maximum value means complete agreement; zero or lower means chance agreement.

References

[1], [2], [3]

sklearn.metrics.confusion_matrix

sklearn.metrics.**confusion_matrix**(*y_true*, *y_pred*, *labels=None*, *sample_weight=None*, *normalize=None*)

Compute confusion matrix to evaluate the accuracy of a classification.

By definition a confusion matrix C is such that $C_{i,j}$ is equal to the number of observations known to be in group i and predicted to be in group j.

Thus in binary classification, the count of true negatives is $C_{0,0}$, false negatives is $C_{1,0}$, true positives is $C_{1,1}$ and false positives is $C_{0,1}$.

Read more in the *User Guide*.

Parameters

y_true [array-like of shape (n_samples,)] Ground truth (correct) target values.

y_pred [array-like of shape (n_samples,)] Estimated targets as returned by a classifier.

labels [array-like of shape (n_classes), default=None] List of labels to index the matrix. This may be used to reorder or select a subset of labels. If `None` is given, those that appear at least once in `y_true` or `y_pred` are used in sorted order.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

normalize [{'true', 'pred', 'all'}, default=None] Normalizes confusion matrix over the true (rows), predicted (columns) conditions or all the population. If None, confusion matrix will not be normalized.

Returns

C [ndarray of shape (n_classes, n_classes)] Confusion matrix.

References

[1]

Examples

```
>>> from sklearn.metrics import confusion_matrix
>>> y_true = [2, 0, 2, 2, 0, 1]
>>> y_pred = [0, 0, 2, 2, 0, 2]
>>> confusion_matrix(y_true, y_pred)
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])
```

```
>>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
>>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
>>> confusion_matrix(y_true, y_pred, labels=["ant", "bird", "cat"])
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])
```

In the binary case, we can extract true positives, etc as follows:

```
>>> tn, fp, fn, tp = confusion_matrix([0, 1, 0, 1], [1, 1, 1, 0]).ravel()
>>> (tn, fp, fn, tp)
(0, 2, 1, 1)
```

Examples using `sklearn.metrics.confusion_matrix`

- *Faces recognition example using eigenfaces and SVMs*
- *Label Propagation digits: Demonstrating performance*
- *Label Propagation digits active learning*
- *Classification of text documents using sparse features*

`sklearn.metrics.dcg_score`

`sklearn.metrics.dcg_score`(*y_true*, *y_score*, *k=None*, *log_base=2*, *sample_weight=None*, *ignore_ties=False*)

Compute Discounted Cumulative Gain.

Sum the true scores ranked in the order induced by the predicted scores, after applying a logarithmic discount.

This ranking metric yields a high value if true labels are ranked high by `y_score`.

Usually the Normalized Discounted Cumulative Gain (NDCG, computed by ndcg_score) is preferred.

Parameters

y_true [ndarray, shape (n_samples, n_labels)] True targets of multilabel classification, or true scores of entities to be ranked.

y_score [ndarray, shape (n_samples, n_labels)] Target scores, can either be probability estimates, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).

k [int, optional (default=None)] Only consider the highest k scores in the ranking. If None, use all outputs.

log_base [float, optional (default=2)] Base of the logarithm used for the discount. A low value means a sharper discount (top results are more important).

sample_weight [ndarray, shape (n_samples,), optional (default=None)] Sample weights. If None, all samples are given the same weight.

ignore_ties [bool, optional (default=False)] Assume that there are no ties in y_score (which is likely to be the case if y_score is continuous) for efficiency gains.

Returns

discounted_cumulative_gain [float] The averaged sample DCG scores.

See also:

ndcg_score The Discounted Cumulative Gain divided by the Ideal Discounted Cumulative Gain (the DCG obtained for a perfect ranking), in order to have a score between 0 and 1.

References

Wikipedia entry for Discounted Cumulative Gain

Jarvelin, K., & Kekalainen, J. (2002). Cumulated gain-based evaluation of IR techniques. ACM Transactions on Information Systems (TOIS), 20(4), 422-446.

Wang, Y., Wang, L., Li, Y., He, D., Chen, W., & Liu, T. Y. (2013, May). A theoretical analysis of NDCG ranking measures. In Proceedings of the 26th Annual Conference on Learning Theory (COLT 2013)

McSherry, F., & Najork, M. (2008, March). Computing information retrieval performance measures efficiently in the presence of tied scores. In European conference on information retrieval (pp. 414-421). Springer, Berlin, Heidelberg.

Examples

```
>>> from sklearn.metrics import dcg_score
>>> # we have groud-truth relevance of some answers to a query:
>>> true_relevance = np.asarray([[10, 0, 0, 1, 5]])
>>> # we predict scores for the answers
>>> scores = np.asarray([[.1, .2, .3, 4, 70]])
>>> dcg_score(true_relevance, scores) # doctest: +ELLIPSIS
9.49...
>>> # we can set k to truncate the sum; only top k answers contribute
>>> dcg_score(true_relevance, scores, k=2) # doctest: +ELLIPSIS
5.63...
>>> # now we have some ties in our prediction
>>> scores = np.asarray([[1, 0, 0, 0, 1]])
>>> # by default ties are averaged, so here we get the average true
>>> # relevance of our top predictions: (10 + 5) / 2 = 7.5
>>> dcg_score(true_relevance, scores, k=1) # doctest: +ELLIPSIS
7.5
>>> # we can choose to ignore ties for faster results, but only
>>> # if we know there aren't ties in our scores, otherwise we get
>>> # wrong results:
>>> dcg_score(true_relevance,
```

(continues on next page)

(continued from previous page)

```
...             scores, k=1, ignore_ties=True) # doctest: +ELLIPSIS
5.0
```

sklearn.metrics.f1_score

sklearn.metrics.**f1_score**(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*, *zero_division='warn'*)

Compute the F1 score, also known as balanced F-score or F-measure

The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal. The formula for the F1 score is:

```
F1 = 2 * (precision * recall) / (precision + recall)
```

In the multi-class and multi-label case, this is the average of the F1 score of each class with weighting depending on the `average` parameter.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) target values.

y_pred [1d array-like, or label indicator array / sparse matrix] Estimated targets as returned by a classifier.

labels [list, optional] The set of labels to include when `average != 'binary'`, and their order if `average is None`. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in `y_true` and `y_pred` are used in sorted order.

Changed in version 0.17: parameter *labels* improved for multiclass problem.

pos_label [str or int, 1 by default] The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that label only.

average [string, [None, 'binary' (default), 'micro', 'macro', 'samples', 'weighted']] This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

'binary': Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

'macro': Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from *accuracy_score*).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

zero_division ["warn", 0 or 1, default="warn"] Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to "warn", this acts as 0, but warnings are also raised.

Returns

f1_score [float or array of float, shape = [n_unique_labels]] F1 score of the positive class in binary classification or weighted average of the F1 scores of each class for the multiclass task.

See also:

fbeta_score, ***precision_recall_fscore_support***, ***jaccard_score***

multilabel_confusion_matrix

Notes

When `true positive + false positive == 0`, precision is undefined; When `true positive + false negative == 0`, recall is undefined. In such cases, by default the metric will be set to 0, as will f-score, and `UndefinedMetricWarning` will be raised. This behavior can be modified with `zero_division`.

References

[1]

Examples

```
>>> from sklearn.metrics import f1_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> f1_score(y_true, y_pred, average='macro')
0.26...
>>> f1_score(y_true, y_pred, average='micro')
0.33...
>>> f1_score(y_true, y_pred, average='weighted')
0.26...
>>> f1_score(y_true, y_pred, average=None)
array([0.8, 0. , 0. ])
>>> y_true = [0, 0, 0, 0, 0, 0]
>>> y_pred = [0, 0, 0, 0, 0, 0]
>>> f1_score(y_true, y_pred, zero_division=1)
1.0...
```

Examples using `sklearn.metrics.f1_score`

- *Probability Calibration curves*
- *Precision-Recall*

sklearn.metrics.fbeta_score

sklearn.metrics.**fbeta_score**(*y_true*, *y_pred*, *beta*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*, *zero_division='warn'*)

Compute the F-beta score

The F-beta score is the weighted harmonic mean of precision and recall, reaching its optimal value at 1 and its worst value at 0.

The `beta` parameter determines the weight of recall in the combined score. `beta < 1` lends more weight to precision, while `beta > 1` favors recall (`beta -> 0` considers only precision, `beta -> +inf` only recall).

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) target values.

y_pred [1d array-like, or label indicator array / sparse matrix] Estimated targets as returned by a classifier.

beta [float] Determines the weight of recall in the combined score.

labels [list, optional] The set of labels to include when `average != 'binary'`, and their order if `average is None`. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in `y_true` and `y_pred` are used in sorted order.

Changed in version 0.17: parameter *labels* improved for multiclass problem.

pos_label [str or int, 1 by default] The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that label only.

average [string, [None, 'binary' (default), 'micro', 'macro', 'samples', 'weighted']] This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

'binary': Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

'macro': Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from *accuracy_score*).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

zero_division ["warn", 0 or 1, default="warn"] Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to "warn", this acts as 0, but warnings are also raised.

Returns

fbeta_score [float (if average is not None) or array of float, shape = [n_unique_labels]] F-beta score of the positive class in binary classification or weighted average of the F-beta score of each class for the multiclass task.

See also:

precision_recall_fscore_support, *multilabel_confusion_matrix*

Notes

When `true positive + false positive == 0` or `true positive + false negative == 0`, f-score returns 0 and raises `UndefinedMetricWarning`. This behavior can be modified with `zero_division`.

References

[1], [2]

Examples

```
>>> from sklearn.metrics import fbeta_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> fbeta_score(y_true, y_pred, average='macro', beta=0.5)
0.23...
>>> fbeta_score(y_true, y_pred, average='micro', beta=0.5)
0.33...
>>> fbeta_score(y_true, y_pred, average='weighted', beta=0.5)
0.23...
>>> fbeta_score(y_true, y_pred, average=None, beta=0.5)
array([0.71..., 0.        , 0.        ])
```

sklearn.metrics.hamming_loss

sklearn.metrics.**hamming_loss**(*y_true*, *y_pred*, *labels=None*, *sample_weight=None*)

Compute the average Hamming loss.

The Hamming loss is the fraction of labels that are incorrectly predicted.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) labels.

y_pred [1d array-like, or label indicator array / sparse matrix] Predicted labels, as returned by a classifier.

labels [array, shape = [n_labels], optional (default='deprecated')] Integer array of labels. If not provided, labels will be inferred from y_true and y_pred.

New in version 0.18.

Deprecated since version 0.21: This parameter `labels` is deprecated in version 0.21 and will be removed in version 0.23. Hamming loss uses `y_true.shape[1]` for the number of labels when y_true is binary label indicators, so it is unnecessary for the user to specify.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

New in version 0.18.

Returns

loss [float or int,] Return the average Hamming loss between element of `y_true` and `y_pred`.

See also:

accuracy_score, ***jaccard_score***, ***zero_one_loss***

Notes

In multiclass classification, the Hamming loss corresponds to the Hamming distance between `y_true` and `y_pred` which is equivalent to the subset `zero_one_loss` function, when `normalize` parameter is set to True.

In multilabel classification, the Hamming loss is different from the subset zero-one loss. The zero-one loss considers the entire set of labels for a given sample incorrect if it does not entirely match the true set of labels. Hamming loss is more forgiving in that it penalizes only the individual labels.

The Hamming loss is upperbounded by the subset zero-one loss, when `normalize` parameter is set to True. It is always between 0 and 1, lower being better.

References

[1], [2]

Examples

```
>>> from sklearn.metrics import hamming_loss
>>> y_pred = [1, 2, 3, 4]
>>> y_true = [2, 2, 3, 4]
>>> hamming_loss(y_true, y_pred)
0.25
```

In the multilabel case with binary label indicators:

```
>>> import numpy as np
>>> hamming_loss(np.array([[0, 1], [1, 1]]), np.zeros((2, 2)))
0.75
```

Examples using `sklearn.metrics.hamming_loss`

- *Model Complexity Influence*

sklearn.metrics.hinge_loss

sklearn.metrics.**hinge_loss**(*y_true*, *pred_decision*, *labels=None*, *sample_weight=None*)

Average hinge loss (non-regularized)

In binary class case, assuming labels in y_true are encoded with +1 and -1, when a prediction mistake is made, `margin = y_true * pred_decision` is always negative (since the signs disagree), implying `1 - margin` is always greater than 1. The cumulated hinge loss is therefore an upper bound of the number of mistakes made by the classifier.

In multiclass case, the function expects that either all the labels are included in y_true or an optional labels argument is provided which contains all the labels. The multilabel margin is calculated according to Crammer-Singer's method. As in the binary case, the cumulated hinge loss is an upper bound of the number of mistakes made by the classifier.

Read more in the *User Guide*.

Parameters

y_true [array, shape = [n_samples]] True target, consisting of integers of two values. The positive label must be greater than the negative label.

pred_decision [array, shape = [n_samples] or [n_samples, n_classes]] Predicted decisions, as output by decision_function (floats).

labels [array, optional, default None] Contains all the labels for the problem. Used in multiclass hinge loss.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

loss [float]

References

[1], [2], [3]

Examples

```
>>> from sklearn import svm
>>> from sklearn.metrics import hinge_loss
>>> X = [[0], [1]]
>>> y = [-1, 1]
>>> est = svm.LinearSVC(random_state=0)
>>> est.fit(X, y)
LinearSVC(random_state=0)
>>> pred_decision = est.decision_function([[-2], [3], [0.5]])
>>> pred_decision
array([-2.18...,  2.36...,  0.09...])
>>> hinge_loss([-1, 1, 1], pred_decision)
0.30...
```

In the multiclass case:

```
>>> import numpy as np
>>> X = np.array([[0], [1], [2], [3]])
>>> Y = np.array([0, 1, 2, 3])
```

(continues on next page)

(continued from previous page)

```
>>> labels = np.array([0, 1, 2, 3])
>>> est = svm.LinearSVC()
>>> est.fit(X, Y)
LinearSVC()
>>> pred_decision = est.decision_function([[-1], [2], [3]])
>>> y_true = [0, 2, 3]
>>> hinge_loss(y_true, pred_decision, labels)
0.56...
```

sklearn.metrics.jaccard_score

sklearn.metrics.**jaccard_score**(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*)

Jaccard similarity coefficient score

The Jaccard index [1], or Jaccard similarity coefficient, defined as the size of the intersection divided by the size of the union of two label sets, is used to compare set of predicted labels for a sample to the corresponding set of labels in `y_true`.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) labels.

y_pred [1d array-like, or label indicator array / sparse matrix] Predicted labels, as returned by a classifier.

labels [list, optional] The set of labels to include when `average != 'binary'`, and their order if `average is None`. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in `y_true` and `y_pred` are used in sorted order.

pos_label [str or int, 1 by default] The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that label only.

average [string, [None, 'binary' (default), 'micro', 'macro', 'samples', 'weighted']] If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

'binary': Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

'macro': Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

'weighted': Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance.

'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float (if average is not None) or array of floats, shape = [n_unique_labels]]

See also:

accuracy_score, **f_score**, *multilabel_confusion_matrix*

Notes

jaccard_score may be a poor metric if there are no positives for some samples or classes. Jaccard is undefined if there are no true or predicted labels, and our implementation will return a score of 0 with a warning.

References

[1]

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import jaccard_score
>>> y_true = np.array([[0, 1, 1],
...                    [1, 1, 0]])
>>> y_pred = np.array([[1, 1, 1],
...                    [1, 0, 0]])
```

In the binary case:

```
>>> jaccard_score(y_true[0], y_pred[0])
0.6666...
```

In the multilabel case:

```
>>> jaccard_score(y_true, y_pred, average='samples')
0.5833...
>>> jaccard_score(y_true, y_pred, average='macro')
0.6666...
>>> jaccard_score(y_true, y_pred, average=None)
array([0.5, 0.5, 1. ])
```

In the multiclass case:

```
>>> y_pred = [0, 2, 1, 2]
>>> y_true = [0, 1, 2, 2]
>>> jaccard_score(y_true, y_pred, average=None)
array([1. , 0. , 0.33...])
```

Examples using `sklearn.metrics.jaccard_score`

- *Classifier Chain*

sklearn.metrics.log_loss

sklearn.metrics.**log_loss**(*y_true*, *y_pred*, *eps=1e-15*, *normalize=True*, *sample_weight=None*, *labels=None*)

Log loss, aka logistic loss or cross-entropy loss.

This is the loss function used in (multinomial) logistic regression and extensions of it such as neural networks, defined as the negative log-likelihood of the true labels given a probabilistic classifier's predictions. The log loss is only defined for two or more labels. For a single sample with true label yt in {0,1} and estimated probability yp that yt = 1, the log loss is

-log P(ytlyp) = -(yt log(yp) + (1 - yt) log(1 - yp))

Read more in the *User Guide*.

Parameters

y_true [array-like or label indicator matrix] Ground truth (correct) labels for n_samples samples.

y_pred [array-like of float, shape = (n_samples, n_classes) or (n_samples,)] Predicted probabilities, as returned by a classifier's predict_proba method. If `y_pred.shape = (n_samples,)` the probabilities provided are assumed to be that of the positive class. The labels in `y_pred` are assumed to be ordered alphabetically, as done by `preprocessing.LabelBinarizer`.

eps [float] Log loss is undefined for p=0 or p=1, so probabilities are clipped to max(eps, min(1 - eps, p)).

normalize [bool, optional (default=True)] If true, return the mean loss per sample. Otherwise, return the sum of the per-sample losses.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

labels [array-like, optional (default=None)] If not provided, labels will be inferred from y_true. If `labels` is `None` and `y_pred` has shape (n_samples,) the labels are assumed to be binary and are inferred from `y_true`. .. versionadded:: 0.18

Returns

loss [float]

Notes

The logarithm used is the natural logarithm (base-e).

References

C.M. Bishop (2006). Pattern Recognition and Machine Learning. Springer, p. 209.

Examples

```
>>> from sklearn.metrics import log_loss
>>> log_loss(["spam", "ham", "ham", "spam"],
...          [[.1, .9], [.9, .1], [.8, .2], [.35, .65]])
0.21616...
```

Examples using `sklearn.metrics.log_loss`

- *Probability Calibration for 3-class classification*
- *Probabilistic predictions with Gaussian process classification (GPC)*

`sklearn.metrics`.matthews_corrcoef

sklearn.metrics.**matthews_corrcoef**(*y_true*, *y_pred*, *sample_weight=None*)

Compute the Matthews correlation coefficient (MCC)

The Matthews correlation coefficient is used in machine learning as a measure of the quality of binary and multiclass classifications. It takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used even if the classes are of very different sizes. The MCC is in essence a correlation coefficient value between -1 and +1. A coefficient of +1 represents a perfect prediction, 0 an average random prediction and -1 an inverse prediction. The statistic is also known as the phi coefficient. [source: Wikipedia]

Binary and multiclass labels are supported. Only in the binary case does this relate to information about true and false positives and negatives. See references below.

Read more in the *User Guide*.

Parameters

y_true [array, shape = [n_samples]] Ground truth (correct) target values.

y_pred [array, shape = [n_samples]] Estimated targets as returned by a classifier.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

mcc [float] The Matthews correlation coefficient (+1 represents a perfect prediction, 0 an average random prediction and -1 and inverse prediction).

References

[1], [2], [3], [4]

Examples

```
>>> from sklearn.metrics import matthews_corrcoef
>>> y_true = [+1, +1, +1, -1]
>>> y_pred = [+1, -1, +1, +1]
>>> matthews_corrcoef(y_true, y_pred)
-0.33...
```

`sklearn.metrics`.multilabel_confusion_matrix

sklearn.metrics.**multilabel_confusion_matrix**(*y_true*, *y_pred*, *sample_weight=None*, *labels=None*, *samplewise=False*)

Compute a confusion matrix for each class or sample

New in version 0.21.

Compute class-wise (default) or sample-wise (samplewise=True) multilabel confusion matrix to evaluate the accuracy of a classification, and output confusion matrices for each class or sample.

In multilabel confusion matrix MCM, the count of true negatives is $MCM_{:,0,0}$, false negatives is $MCM_{:,1,0}$, true positives is $MCM_{:,1,1}$ and false positives is $MCM_{:,0,1}$.

Multiclass data will be treated as if binarized under a one-vs-rest transformation. Returned confusion matrices will be in the order of sorted unique labels in the union of (y_true, y_pred).

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] of shape (n_samples, n_outputs) or (n_samples,) Ground truth (correct) target values.

y_pred [1d array-like, or label indicator array / sparse matrix] of shape (n_samples, n_outputs) or (n_samples,) Estimated targets as returned by a classifier

sample_weight [array-like of shape (n_samples,), default=None] Sample weights

labels [array-like] A list of classes or column indices to select some (or to force inclusion of classes absent from the data)

samplewise [bool, default=False] In the multilabel case, this calculates a confusion matrix per sample

Returns

multi_confusion [array, shape (n_outputs, 2, 2)] A 2x2 confusion matrix corresponding to each output in the input. When calculating class-wise multi_confusion (default), then n_outputs = n_labels; when calculating sample-wise multi_confusion (samplewise=True), n_outputs = n_samples. If `labels` is defined, the results will be returned in the order specified in `labels`, otherwise the results will be returned in sorted order by default.

See also:

confusion_matrix

Notes

The multilabel_confusion_matrix calculates class-wise or sample-wise multilabel confusion matrices, and in multiclass tasks, labels are binarized under a one-vs-rest way; while confusion_matrix calculates one confusion matrix for confusion between every two classes.

Examples

Multilabel-indicator case:

```
>>> import numpy as np
>>> from sklearn.metrics import multilabel_confusion_matrix
>>> y_true = np.array([[1, 0, 1],
...                    [0, 1, 0]])
>>> y_pred = np.array([[1, 0, 0],
...                    [0, 1, 1]])
>>> multilabel_confusion_matrix(y_true, y_pred)
array([[[1, 0],
        [0, 1]],
<BLANKLINE>
```

(continues on next page)

(continued from previous page)

```
       [[1, 0],
        [0, 1]],
<BLANKLINE>
       [[0, 1],
        [1, 0]]])
```

Multiclass case:

```
>>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
>>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
>>> multilabel_confusion_matrix(y_true, y_pred,
...                             labels=["ant", "bird", "cat"])
array([[[3, 1],
        [0, 2]],
<BLANKLINE>
       [[5, 0],
        [1, 0]],
<BLANKLINE>
       [[2, 1],
        [1, 2]]])
```

sklearn.metrics.ndcg_score

sklearn.metrics.**ndcg_score**(*y_true*, *y_score*, *k=None*, *sample_weight=None*, *ignore_ties=False*)

Compute Normalized Discounted Cumulative Gain.

Sum the true scores ranked in the order induced by the predicted scores, after applying a logarithmic discount. Then divide by the best possible score (Ideal DCG, obtained for a perfect ranking) to obtain a score between 0 and 1.

This ranking metric yields a high value if true labels are ranked high by `y_score`.

Parameters

y_true [ndarray, shape (n_samples, n_labels)] True targets of multilabel classification, or true scores of entities to be ranked.

y_score [ndarray, shape (n_samples, n_labels)] Target scores, can either be probability estimates, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).

k [int, optional (default=None)] Only consider the highest k scores in the ranking. If None, use all outputs.

sample_weight [ndarray, shape (n_samples,), optional (default=None)] Sample weights. If None, all samples are given the same weight.

ignore_ties [bool, optional (default=False)] Assume that there are no ties in y_score (which is likely to be the case if y_score is continuous) for efficiency gains.

Returns

normalized_discounted_cumulative_gain [float in [0., 1.]] The averaged NDCG scores for all samples.

See also:

dcg_score Discounted Cumulative Gain (not normalized).

References

Wikipedia entry for Discounted Cumulative Gain

Jarvelin, K., & Kekalainen, J. (2002). Cumulated gain-based evaluation of IR techniques. ACM Transactions on Information Systems (TOIS), 20(4), 422-446.

Wang, Y., Wang, L., Li, Y., He, D., Chen, W., & Liu, T. Y. (2013, May). A theoretical analysis of NDCG ranking measures. In Proceedings of the 26th Annual Conference on Learning Theory (COLT 2013)

McSherry, F., & Najork, M. (2008, March). Computing information retrieval performance measures efficiently in the presence of tied scores. In European conference on information retrieval (pp. 414-421). Springer, Berlin, Heidelberg.

Examples

```
>>> from sklearn.metrics import ndcg_score
>>> # we have groud-truth relevance of some answers to a query:
>>> true_relevance = np.asarray([[10, 0, 0, 1, 5]])
>>> # we predict some scores (relevance) for the answers
>>> scores = np.asarray([[.1, .2, .3, 4, 70]])
>>> ndcg_score(true_relevance, scores) # doctest: +ELLIPSIS
0.69...
>>> scores = np.asarray([[.05, 1.1, 1., .5, .0]])
>>> ndcg_score(true_relevance, scores) # doctest: +ELLIPSIS
0.49...
>>> # we can set k to truncate the sum; only top k answers contribute.
>>> ndcg_score(true_relevance, scores, k=4) # doctest: +ELLIPSIS
0.35...
>>> # the normalization takes k into account so a perfect answer
>>> # would still get 1.0
>>> ndcg_score(true_relevance, true_relevance, k=4) # doctest: +ELLIPSIS
1.0
>>> # now we have some ties in our prediction
>>> scores = np.asarray([[1, 0, 0, 0, 1]])
>>> # by default ties are averaged, so here we get the average (normalized)
>>> # true relevance of our top predictions: (10 / 10 + 5 / 10) / 2 = .75
>>> ndcg_score(true_relevance, scores, k=1) # doctest: +ELLIPSIS
0.75
>>> # we can choose to ignore ties for faster results, but only
>>> # if we know there aren't ties in our scores, otherwise we get
>>> # wrong results:
>>> ndcg_score(true_relevance,
...           scores, k=1, ignore_ties=True) # doctest: +ELLIPSIS
0.5
```

sklearn.metrics.precision_recall_curve

sklearn.metrics.**precision_recall_curve**(*y_true*, *probas_pred*, *pos_label=None*, *sample_weight=None*)

Compute precision-recall pairs for different probability thresholds

Note: this implementation is restricted to the binary classification task.

The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The last precision and recall values are 1. and 0. respectively and do not have a corresponding threshold. This ensures that the graph starts on the y axis.

Read more in the *User Guide*.

Parameters

y_true [array, shape = [n_samples]] True binary labels. If labels are not either {-1, 1} or {0, 1}, then pos_label should be explicitly given.

probas_pred [array, shape = [n_samples]] Estimated probabilities or decision function.

pos_label [int or str, default=None] The label of the positive class. When `pos_label=None`, if y_true is in {-1, 1} or {0, 1}, `pos_label` is set to 1, otherwise an error will be raised.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

precision [array, shape = [n_thresholds + 1]] Precision values such that element i is the precision of predictions with score >= thresholds[i] and the last element is 1.

recall [array, shape = [n_thresholds + 1]] Decreasing recall values such that element i is the recall of predictions with score >= thresholds[i] and the last element is 0.

thresholds [array, shape = [n_thresholds <= len(np.unique(probas_pred))]] Increasing thresholds on the decision function used to compute precision and recall.

See also:

average_precision_score Compute average precision from prediction scores

roc_curve Compute Receiver operating characteristic (ROC) curve

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import precision_recall_curve
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> precision, recall, thresholds = precision_recall_curve(
...     y_true, y_scores)
>>> precision
array([0.66666667, 0.5       , 1.        , 1.        ])
>>> recall
array([1. , 0.5, 0.5, 0. ])
>>> thresholds
array([0.35, 0.4 , 0.8 ])
```

Examples using `sklearn.metrics.precision_recall_curve`

- *Precision-Recall*

sklearn.metrics.precision_recall_fscore_support

sklearn.metrics.**precision_recall_fscore_support**(*y_true*, *y_pred*, *beta=1.0*, *labels=None*, *pos_label=1*, *average=None*, *warn_for=('precision', 'recall', 'f-score')*, *sample_weight=None*, *zero_division='warn'*)

Compute precision, recall, F-measure and support for each class

The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The F-beta score can be interpreted as a weighted harmonic mean of the precision and recall, where an F-beta score reaches its best value at 1 and worst score at 0.

The F-beta score weights recall more than precision by a factor of `beta`. `beta == 1.0` means recall and precision are equally important.

The support is the number of occurrences of each class in `y_true`.

If `pos_label is None` and in binary classification, this function returns the average precision, recall and F-measure if `average` is one of `'micro'`, `'macro'`, `'weighted'` or `'samples'`.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) target values.

y_pred [1d array-like, or label indicator array / sparse matrix] Estimated targets as returned by a classifier.

beta [float, 1.0 by default] The strength of recall versus precision in the F-score.

labels [list, optional] The set of labels to include when `average != 'binary'`, and their order if `average is None`. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in `y_true` and `y_pred` are used in sorted order.

pos_label [str or int, 1 by default] The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that label only.

average [string, [None (default), 'binary', 'micro', 'macro', 'samples', 'weighted']] If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

`'binary'`: Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

`'micro'`: Calculate metrics globally by counting the total true positives, false negatives and false positives.

`'macro'`: Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from *accuracy_score*).

warn_for [tuple or set, for internal use] This determines which warnings will be made in the case that this function is being used to return only one of its metrics.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

zero_division ["warn", 0 or 1, default="warn"]

Sets the value to return when there is a zero division:

- recall: when there are no positive labels
- precision: when there are no positive predictions
- f-score: both

If set to "warn", this acts as 0, but warnings are also raised.

Returns

precision [float (if average is not None) or array of float, shape = [n_unique_labels]]

recall [float (if average is not None) or array of float, , shape = [n_unique_labels]]

fbeta_score [float (if average is not None) or array of float, shape = [n_unique_labels]]

support [None (if average is not None) or array of int, shape = [n_unique_labels]] The number of occurrences of each label in `y_true`.

Notes

When `true positive + false positive == 0`, precision is undefined; When `true positive + false negative == 0`, recall is undefined. In such cases, by default the metric will be set to 0, as will f-score, and `UndefinedMetricWarning` will be raised. This behavior can be modified with `zero_division`.

References

[1], [2], [3]

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import precision_recall_fscore_support
>>> y_true = np.array(['cat', 'dog', 'pig', 'cat', 'dog', 'pig'])
>>> y_pred = np.array(['cat', 'pig', 'dog', 'cat', 'cat', 'dog'])
>>> precision_recall_fscore_support(y_true, y_pred, average='macro')
(0.22..., 0.33..., 0.26..., None)
>>> precision_recall_fscore_support(y_true, y_pred, average='micro')
(0.33..., 0.33..., 0.33..., None)
>>> precision_recall_fscore_support(y_true, y_pred, average='weighted')
(0.22..., 0.33..., 0.26..., None)
```

It is possible to compute per-label precisions, recalls, F1-scores and supports instead of averaging:

```
>>> precision_recall_fscore_support(y_true, y_pred, average=None,
... labels=['pig', 'dog', 'cat'])
(array([0.        , 0.        , 0.66...]),
 array([0., 0., 1.]), array([0. , 0. , 0.8]),
 array([2, 2, 2]))
```

sklearn.metrics.precision_score

sklearn.metrics.**precision_score**(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*, *zero_division='warn'*)

Compute the precision

The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The best value is 1 and the worst value is 0.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) target values.

y_pred [1d array-like, or label indicator array / sparse matrix] Estimated targets as returned by a classifier.

labels [list, optional] The set of labels to include when `average != 'binary'`, and their order if `average is None`. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in `y_true` and `y_pred` are used in sorted order.

Changed in version 0.17: parameter *labels* improved for multiclass problem.

pos_label [str or int, 1 by default] The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that label only.

average [string, [None, 'binary' (default), 'micro', 'macro', 'samples', 'weighted']] This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

'binary': Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

'macro': Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from *accuracy_score*).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

zero_division ["warn", 0 or 1, default="warn"] Sets the value to return when there is a zero division. If set to "warn", this acts as 0, but warnings are also raised.

Returns

precision [float (if average is not None) or array of float, shape = [n_unique_labels]] Precision of the positive class in binary classification or weighted average of the precision of each class for the multiclass task.

See also:

precision_recall_fscore_support, ***multilabel_confusion_matrix***

Notes

When `true positive + false positive == 0`, precision returns 0 and raises `UndefinedMetricWarning`. This behavior can be modified with `zero_division`.

Examples

```
>>> from sklearn.metrics import precision_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> precision_score(y_true, y_pred, average='macro')
0.22...
>>> precision_score(y_true, y_pred, average='micro')
0.33...
>>> precision_score(y_true, y_pred, average='weighted')
0.22...
>>> precision_score(y_true, y_pred, average=None)
array([0.66..., 0.        , 0.        ])
>>> y_pred = [0, 0, 0, 0, 0, 0]
>>> precision_score(y_true, y_pred, average=None)
array([0.33..., 0.        , 0.        ])
>>> precision_score(y_true, y_pred, average=None, zero_division=1)
array([0.33..., 1.        , 1.        ])
```

Examples using `sklearn.metrics.precision_score`

- *Probability Calibration curves*
- *Precision-Recall*

`sklearn.metrics.recall_score`

sklearn.metrics.**recall_score**(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*, *zero_division='warn'*)

Compute the recall

The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The best value is 1 and the worst value is 0.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) target values.

y_pred [1d array-like, or label indicator array / sparse matrix] Estimated targets as returned by a classifier.

labels [list, optional] The set of labels to include when `average != 'binary'`, and their order if `average is None`. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in `y_true` and `y_pred` are used in sorted order.

Changed in version 0.17: parameter *labels* improved for multiclass problem.

pos_label [str or int, 1 by default] The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that label only.

average [string, [None, 'binary' (default), 'micro', 'macro', 'samples', 'weighted']] This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

`'binary'`: Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

`'micro'`: Calculate metrics globally by counting the total true positives, false negatives and false positives.

`'macro'`: Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

`'weighted'`: Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

`'samples'`: Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from *`accuracy_score`*).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

zero_division ["warn", 0 or 1, default="warn"] Sets the value to return when there is a zero division. If set to "warn", this acts as 0, but warnings are also raised.

Returns

recall [float (if average is not None) or array of float, shape = [n_unique_labels]] Recall of the positive class in binary classification or weighted average of the recall of each class for the multiclass task.

See also:

`precision_recall_fscore_support`, ***`balanced_accuracy_score`***

`multilabel_confusion_matrix`

Notes

When `true positive + false negative == 0`, recall returns 0 and raises `UndefinedMetricWarning`. This behavior can be modified with `zero_division`.

Examples

```
>>> from sklearn.metrics import recall_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> recall_score(y_true, y_pred, average='macro')
0.33...
>>> recall_score(y_true, y_pred, average='micro')
0.33...
>>> recall_score(y_true, y_pred, average='weighted')
0.33...
>>> recall_score(y_true, y_pred, average=None)
array([1., 0., 0.])
>>> y_true = [0, 0, 0, 0, 0, 0]
>>> recall_score(y_true, y_pred, average=None)
array([0.5, 0. , 0. ])
>>> recall_score(y_true, y_pred, average=None, zero_division=1)
array([0.5, 1. , 1. ])
```

Examples using `sklearn.metrics.recall_score`

- *Probability Calibration curves*
- *Precision-Recall*

sklearn.metrics.roc_auc_score

sklearn.metrics.**roc_auc_score**(*y_true*, *y_score*, *average='macro'*, *sample_weight=None*, *max_fpr=None*, *multi_class='raise'*, *labels=None*)

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.

Note: this implementation can be used with binary, multiclass and multilabel classification, but some restrictions apply (see Parameters).

Read more in the *User Guide*.

Parameters

y_true [array-like of shape (n_samples,) or (n_samples, n_classes)] True labels or binary label indicators. The binary and multiclass cases expect labels with shape (n_samples,) while the multilabel case expects binary label indicators with shape (n_samples, n_classes).

y_score [array-like of shape (n_samples,) or (n_samples, n_classes)] Target scores. In the binary and multilabel cases, these can be either probability estimates or non-thresholded decision values (as returned by `decision_function` on some classifiers). In the multiclass case, these must be probability estimates which sum to 1. The binary case expects a shape (n_samples,), and the scores must be the scores of the class with the greater label. The multiclass and multilabel cases expect a shape (n_samples, n_classes). In the multiclass case, the order of the class scores must correspond to the order of `labels`, if provided, or else to the numerical or lexicographical order of the labels in `y_true`.

average [{'micro', 'macro', 'samples', 'weighted'} or None, default='macro'] If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data: Note: multiclass ROC AUC currently only handles the 'macro' and 'weighted' averages.

`'micro'`: Calculate metrics globally by considering each element of the label indicator matrix as a label.

`'macro'`: Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

`'weighted'`: Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label).

`'samples'`: Calculate metrics for each instance, and find their average.

Will be ignored when `y_true` is binary.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

max_fpr [float > 0 and <= 1, default=None] If not `None`, the standardized partial AUC [2] over the range [0, max_fpr] is returned. For the multiclass case, `max_fpr`, should be either equal to `None` or `1.0` as AUC ROC partial computation currently is not supported for multiclass.

multi_class [{'raise', 'ovr', 'ovo'}, default='raise'] Multiclass only. Determines the type of configuration to use. The default value raises an error, so either `'ovr'` or `'ovo'` must be passed explicitly.

`'ovr'`: Computes the AUC of each class against the rest [3] [4]. This treats the multiclass case in the same way as the multilabel case. Sensitive to class imbalance even when `average == 'macro'`, because class imbalance affects the composition of each of the 'rest' groupings.

`'ovo'`: Computes the average AUC of all possible pairwise combinations of classes [5]. Insensitive to class imbalance when `average == 'macro'`.

labels [array-like of shape (n_classes,), default=None] Multiclass only. List of labels that index the classes in `y_score`. If `None`, the numerical or lexicographical order of the labels in `y_true` is used.

Returns

auc [float]

See also:

`average_precision_score` Area under the precision-recall curve

`roc_curve` Compute Receiver operating characteristic (ROC) curve

References

[1], [2], [3], [4], [5]

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import roc_auc_score
>>> y_true = np.array([0, 0, 1, 1])
```

(continues on next page)

(continued from previous page)

```
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> roc_auc_score(y_true, y_scores)
0.75
```

Examples using `sklearn.metrics.roc_auc_score`

- *Receiver Operating Characteristic (ROC) with cross validation*
- *Receiver Operating Characteristic (ROC)*
- *Release Highlights for scikit-learn 0.22*

`sklearn.metrics.roc_curve`

`sklearn.metrics.roc_curve`(*y_true*, *y_score*, *pos_label=None*, *sample_weight=None*, *drop_intermediate=True*)

Compute Receiver operating characteristic (ROC)

Note: this implementation is restricted to the binary classification task.

Read more in the *User Guide*.

Parameters

y_true [array, shape = [n_samples]] True binary labels. If labels are not either {-1, 1} or {0, 1}, then pos_label should be explicitly given.

y_score [array, shape = [n_samples]] Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).

pos_label [int or str, default=None] The label of the positive class. When `pos_label=None`, if y_true is in {-1, 1} or {0, 1}, `pos_label` is set to 1, otherwise an error will be raised.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

drop_intermediate [boolean, optional (default=True)] Whether to drop some suboptimal thresholds which would not appear on a plotted ROC curve. This is useful in order to create lighter ROC curves.

New in version 0.17: parameter *drop_intermediate*.

Returns

fpr [array, shape = [>2]] Increasing false positive rates such that element i is the false positive rate of predictions with score >= thresholds[i].

tpr [array, shape = [>2]] Increasing true positive rates such that element i is the true positive rate of predictions with score >= thresholds[i].

thresholds [array, shape = [n_thresholds]] Decreasing thresholds on the decision function used to compute fpr and tpr. `thresholds[0]` represents no instances being predicted and is arbitrarily set to `max(y_score) + 1`.

See also:

roc_auc_score Compute the area under the ROC curve

Notes

Since the thresholds are sorted from low to high values, they are reversed upon returning them to ensure they correspond to both `fpr` and `tpr`, which are sorted in reversed order during their calculation.

References

[1], [2]

Examples

```
>>> import numpy as np
>>> from sklearn import metrics
>>> y = np.array([1, 1, 2, 2])
>>> scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> fpr, tpr, thresholds = metrics.roc_curve(y, scores, pos_label=2)
>>> fpr
array([0. , 0. , 0.5, 0.5, 1. ])
>>> tpr
array([0. , 0.5, 0.5, 1. , 1. ])
>>> thresholds
array([1.8 , 0.8 , 0.4 , 0.35, 0.1 ])
```

Examples using `sklearn.metrics.roc_curve`

- *Feature transformations with ensembles of trees*
- *Species distribution modeling*
- *Receiver Operating Characteristic (ROC)*

`sklearn.metrics.zero_one_loss`

sklearn.metrics.**zero_one_loss**(*y_true*, *y_pred*, *normalize=True*, *sample_weight=None*)

Zero-one classification loss.

If normalize is `True`, return the fraction of misclassifications (float), else it returns the number of misclassifications (int). The best performance is 0.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) labels.

y_pred [1d array-like, or label indicator array / sparse matrix] Predicted labels, as returned by a classifier.

normalize [bool, optional (default=True)] If `False`, return the number of misclassifications. Otherwise, return the fraction of misclassifications.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

loss [float or int,] If `normalize == True`, return the fraction of misclassifications (float), else it returns the number of misclassifications (int).

See also:

accuracy_score, *hamming_loss*, *jaccard_score*

Notes

In multilabel classification, the zero_one_loss function corresponds to the subset zero-one loss: for each sample, the entire set of labels must be correctly predicted, otherwise the loss for that sample is equal to one.

Examples

```
>>> from sklearn.metrics import zero_one_loss
>>> y_pred = [1, 2, 3, 4]
>>> y_true = [2, 2, 3, 4]
>>> zero_one_loss(y_true, y_pred)
0.25
>>> zero_one_loss(y_true, y_pred, normalize=False)
1
```

In the multilabel case with binary label indicators:

```
>>> import numpy as np
>>> zero_one_loss(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5
```

Examples using `sklearn.metrics.zero_one_loss`

- *Discrete versus Real AdaBoost*

7.24.3 Regression metrics

See the *Regression metrics* section of the user guide for further details.

metrics.explained_variance_score(y_true, y_pred)	Explained variance regression score function
metrics.max_error(y_true, y_pred)	max_error metric calculates the maximum residual error.
metrics.mean_absolute_error(y_true, y_pred)	Mean absolute error regression loss
metrics.mean_squared_error(y_true, y_pred[, ...])	Mean squared error regression loss
metrics.mean_squared_log_error(y_true, y_pred)	Mean squared logarithmic error regression loss
metrics.median_absolute_error(y_true, y_pred)	Median absolute error regression loss
metrics.r2_score(y_true, y_pred[, ...])	R^2 (coefficient of determination) regression score function.
metrics.mean_poisson_deviance(y_true, y_pred)	Mean Poisson deviance regression loss.

Continued on next page

Table 190 – continued from previous page

metrics.mean_gamma_deviance(y_true, y_pred)	Mean Gamma deviance regression loss.
metrics.mean_tweedie_deviance(y_true, y_pred)	Mean Tweedie deviance regression loss.

sklearn.metrics.explained_variance_score

sklearn.metrics.**explained_variance_score**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

Explained variance regression score function

Best possible score is 1.0, lower values are worse.

Read more in the *User Guide*.

Parameters

y_true [array-like of shape (n_samples,) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape (n_samples,) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape (n_samples,), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average', 'variance_weighted'] or array-like of shape (n_outputs)] Defines aggregating of multiple output scores. Array-like value defines weights used to average scores.

'raw_values' : Returns a full set of scores in case of multioutput input.

'uniform_average' : Scores of all outputs are averaged with uniform weight.

'variance_weighted' : Scores of all outputs are averaged, weighted by the variances of each individual output.

Returns

score [float or ndarray of floats] The explained variance or ndarray if 'multioutput' is 'raw_values'.

Notes

This is not a symmetric function.

Examples

```
>>> from sklearn.metrics import explained_variance_score
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> explained_variance_score(y_true, y_pred)
0.957...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> explained_variance_score(y_true, y_pred, multioutput='uniform_average')
0.983...
```

sklearn.metrics.max_error

sklearn.metrics.**max_error**(*y_true*, *y_pred*)

max_error metric calculates the maximum residual error.

Read more in the *User Guide*.

Parameters

y_true [array-like of shape (n_samples,)] Ground truth (correct) target values.

y_pred [array-like of shape (n_samples,)] Estimated target values.

Returns

max_error [float] A positive floating point value (the best value is 0.0).

Examples

```
>>> from sklearn.metrics import max_error
>>> y_true = [3, 2, 7, 1]
>>> y_pred = [4, 2, 7, 1]
>>> max_error(y_true, y_pred)
1
```

sklearn.metrics.mean_absolute_error

sklearn.metrics.**mean_absolute_error**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

Mean absolute error regression loss

Read more in the *User Guide*.

Parameters

y_true [array-like of shape (n_samples,) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape (n_samples,) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape (n_samples,), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average']] or array-like of shape (n_outputs) Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

'raw_values' : Returns a full set of errors in case of multioutput input.

'uniform_average' : Errors of all outputs are averaged with uniform weight.

Returns

loss [float or ndarray of floats] If multioutput is 'raw_values', then mean absolute error is returned for each output separately. If multioutput is 'uniform_average' or an ndarray of weights, then the weighted average of all output errors is returned.

MAE output is non-negative floating point. The best value is 0.0.

Examples

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_absolute_error(y_true, y_pred)
0.5
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> mean_absolute_error(y_true, y_pred)
0.75
>>> mean_absolute_error(y_true, y_pred, multioutput='raw_values')
array([0.5, 1. ])
>>> mean_absolute_error(y_true, y_pred, multioutput=[0.3, 0.7])
0.85...
```

sklearn.metrics.mean_squared_error

sklearn.metrics.**mean_squared_error**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*, *squared=True*)

Mean squared error regression loss

Read more in the *User Guide*.

Parameters

y_true [array-like of shape (n_samples,) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape (n_samples,) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape (n_samples,), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average']] or array-like of shape (n_outputs) Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

'raw_values' : Returns a full set of errors in case of multioutput input.

'uniform_average' : Errors of all outputs are averaged with uniform weight.

squared [boolean value, optional (default = True)] If True returns MSE value, if False returns RMSE value.

Returns

loss [float or ndarray of floats] A non-negative floating point value (the best value is 0.0), or an array of floating point values, one for each individual target.

Examples

```
>>> from sklearn.metrics import mean_squared_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_squared_error(y_true, y_pred)
0.375
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
```

(continues on next page)

(continued from previous page)

```
>>> mean_squared_error(y_true, y_pred, squared=False)
0.612...
>>> y_true = [[0.5, 1],[-1, 1],[7, -6]]
>>> y_pred = [[0, 2],[-1, 2],[8, -5]]
>>> mean_squared_error(y_true, y_pred)
0.708...
>>> mean_squared_error(y_true, y_pred, multioutput='raw_values')
array([0.41666667, 1.        ])
>>> mean_squared_error(y_true, y_pred, multioutput=[0.3, 0.7])
0.825...
```

Examples using `sklearn.metrics.mean_squared_error`

- *Gradient Boosting regression*
- *Model Complexity Influence*
- *Plot Ridge coefficients as a function of the L2 regularization*
- *Linear Regression Example*
- *Robust linear estimator fitting*

`sklearn.metrics`.mean_squared_log_error

`sklearn.metrics.`**`mean_squared_log_error`**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

Mean squared logarithmic error regression loss

Read more in the *User Guide*.

Parameters

y_true [array-like of shape (n_samples,) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape (n_samples,) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape (n_samples,), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average'] or array-like of shape (n_outputs)] Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

'raw_values' : Returns a full set of errors when the input is of multioutput format.

'uniform_average' : Errors of all outputs are averaged with uniform weight.

Returns

loss [float or ndarray of floats] A non-negative floating point value (the best value is 0.0), or an array of floating point values, one for each individual target.

Examples

```
>>> from sklearn.metrics import mean_squared_log_error
>>> y_true = [3, 5, 2.5, 7]
>>> y_pred = [2.5, 5, 4, 8]
>>> mean_squared_log_error(y_true, y_pred)
0.039...
>>> y_true = [[0.5, 1], [1, 2], [7, 6]]
>>> y_pred = [[0.5, 2], [1, 2.5], [8, 8]]
>>> mean_squared_log_error(y_true, y_pred)
0.044...
>>> mean_squared_log_error(y_true, y_pred, multioutput='raw_values')
array([0.00462428, 0.08377444])
>>> mean_squared_log_error(y_true, y_pred, multioutput=[0.3, 0.7])
0.060...
```

sklearn.metrics.median_absolute_error

sklearn.metrics.**median_absolute_error**(*y_true*, *y_pred*, *multioutput='uniform_average'*)

Median absolute error regression loss

Median absolute error output is non-negative floating point. The best value is 0.0. Read more in the *User Guide*.

Parameters

y_true [array-like of shape = (n_samples) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples) or (n_samples, n_outputs)] Estimated target values.

multioutput [{'raw_values', 'uniform_average'} or array-like of shape] (n_outputs,) Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

'raw_values' : Returns a full set of errors in case of multioutput input.

'uniform_average' : Errors of all outputs are averaged with uniform weight.

Returns

loss [float or ndarray of floats] If multioutput is 'raw_values', then mean absolute error is returned for each output separately. If multioutput is 'uniform_average' or an ndarray of weights, then the weighted average of all output errors is returned.

Examples

```
>>> from sklearn.metrics import median_absolute_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> median_absolute_error(y_true, y_pred)
0.5
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> median_absolute_error(y_true, y_pred)
0.75
>>> median_absolute_error(y_true, y_pred, multioutput='raw_values')
array([0.5, 1. ])
>>> median_absolute_error(y_true, y_pred, multioutput=[0.3, 0.7])
0.85
```

Examples using `sklearn.metrics.median_absolute_error`

- *Effect of transforming the targets in regression model*

`sklearn.metrics.r2_score`

`sklearn.metrics.r2_score`(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

R^2 (coefficient of determination) regression score function.

Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Read more in the *User Guide*.

Parameters

y_true [array-like of shape (n_samples,) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape (n_samples,) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape (n_samples,), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average', 'variance_weighted'] or None or array-like of shape (n_outputs)] Defines aggregating of multiple output scores. Array-like value defines weights used to average scores. Default is "uniform_average".

'raw_values' : Returns a full set of scores in case of multioutput input.

'uniform_average' : Scores of all outputs are averaged with uniform weight.

'variance_weighted' : Scores of all outputs are averaged, weighted by the variances of each individual output.

Changed in version 0.19: Default value of multioutput is 'uniform_average'.

Returns

z [float or ndarray of floats] The R^2 score or ndarray of scores if 'multioutput' is 'raw_values'.

Notes

This is not a symmetric function.

Unlike most other scores, R^2 score may be negative (it need not actually be the square of a quantity R).

This metric is not well-defined for single samples and will return a NaN value if n_samples is less than two.

References

[1]

Examples

```
>>> from sklearn.metrics import r2_score
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> r2_score(y_true, y_pred)
0.948...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> r2_score(y_true, y_pred,
...          multioutput='variance_weighted')
0.938...
>>> y_true = [1, 2, 3]
>>> y_pred = [1, 2, 3]
>>> r2_score(y_true, y_pred)
1.0
>>> y_true = [1, 2, 3]
>>> y_pred = [2, 2, 2]
>>> r2_score(y_true, y_pred)
0.0
>>> y_true = [1, 2, 3]
>>> y_pred = [3, 2, 1]
>>> r2_score(y_true, y_pred)
-3.0
```

Examples using `sklearn.metrics.r2_score`

- *Linear Regression Example*
- *Lasso and Elastic Net for Sparse Signals*
- *Effect of transforming the targets in regression model*

`sklearn.metrics`.mean_poisson_deviance

`sklearn.metrics.`**`mean_poisson_deviance`**(*y_true*, *y_pred*, *sample_weight=None*)

Mean Poisson deviance regression loss.

Poisson deviance is equivalent to the Tweedie deviance with the power parameter `p=1`.

Read more in the *User Guide*.

Parameters

y_true [array-like of shape (n_samples,)] Ground truth (correct) target values. Requires y_true >= 0.

y_pred [array-like of shape (n_samples,)] Estimated target values. Requires y_pred > 0.

sample_weight [array-like, shape (n_samples,), optional] Sample weights.

Returns

loss [float] A non-negative floating point value (the best value is 0.0).

Examples

```
>>> from sklearn.metrics import mean_poisson_deviance
>>> y_true = [2, 0, 1, 4]
>>> y_pred = [0.5, 0.5, 2., 2.]
>>> mean_poisson_deviance(y_true, y_pred)
1.4260...
```

sklearn.metrics.mean_gamma_deviance

sklearn.metrics.**mean_gamma_deviance**(*y_true*, *y_pred*, *sample_weight=None*)

Mean Gamma deviance regression loss.

Gamma deviance is equivalent to the Tweedie deviance with the power parameter `p=2`. It is invariant to scaling of the target variable, and measures relative errors.

Read more in the *User Guide*.

Parameters

y_true [array-like of shape (n_samples,)] Ground truth (correct) target values. Requires y_true > 0.

y_pred [array-like of shape (n_samples,)] Estimated target values. Requires y_pred > 0.

sample_weight [array-like, shape (n_samples,), optional] Sample weights.

Returns

loss [float] A non-negative floating point value (the best value is 0.0).

Examples

```
>>> from sklearn.metrics import mean_gamma_deviance
>>> y_true = [2, 0.5, 1, 4]
>>> y_pred = [0.5, 0.5, 2., 2.]
>>> mean_gamma_deviance(y_true, y_pred)
1.0568...
```

sklearn.metrics.mean_tweedie_deviance

sklearn.metrics.**mean_tweedie_deviance**(*y_true*, *y_pred*, *sample_weight=None*, *power=0*)

Mean Tweedie deviance regression loss.

Read more in the *User Guide*.

Parameters

y_true [array-like of shape (n_samples,)] Ground truth (correct) target values.

y_pred [array-like of shape (n_samples,)] Estimated target values.

sample_weight [array-like, shape (n_samples,), optional] Sample weights.

power [float, default=0] Tweedie power parameter. Either power <= 0 or power >= 1.

The higher `p` the less weight is given to extreme deviations between true and predicted targets.

- power < 0: Extreme stable distribution. Requires: y_pred > 0.

- power = 0 : Normal distribution, output corresponds to mean_squared_error. y_true and y_pred can be any real numbers.
- power = 1 : Poisson distribution. Requires: y_true >= 0 and y_pred > 0.
- 1 < p < 2 : Compound Poisson distribution. Requires: y_true >= 0 and y_pred > 0.
- power = 2 : Gamma distribution. Requires: y_true > 0 and y_pred > 0.
- power = 3 : Inverse Gaussian distribution. Requires: y_true > 0 and y_pred > 0.
- otherwise : Positive stable distribution. Requires: y_true > 0 and y_pred > 0.

Returns

loss [float] A non-negative floating point value (the best value is 0.0).

Examples

```
>>> from sklearn.metrics import mean_tweedie_deviance
>>> y_true = [2, 0, 1, 4]
>>> y_pred = [0.5, 0.5, 2., 2.]
>>> mean_tweedie_deviance(y_true, y_pred, power=1)
1.4260...
```

7.24.4 Multilabel ranking metrics

See the *Multilabel ranking metrics* section of the user guide for further details.

metrics.coverage_error(y_true, y_score[, ...])	Coverage error measure
metrics.label_ranking_average_precision_score(y_true, ...)	Compute ranking-based average precision
metrics.label_ranking_loss(y_true, y_score)	Compute Ranking loss measure

sklearn.metrics.coverage_error

sklearn.metrics.**coverage_error**(*y_true*, *y_score*, *sample_weight=None*)

Coverage error measure

Compute how far we need to go through the ranked scores to cover all true labels. The best value is equal to the average number of labels in `y_true` per sample.

Ties in `y_scores` are broken by giving maximal rank that would have been assigned to all tied values.

Note: Our implementation's score is 1 greater than the one given in Tsoumakas et al., 2010. This extends it to handle the degenerate case in which an instance has 0 true labels.

Read more in the *User Guide*.

Parameters

y_true [array, shape = [n_samples, n_labels]] True binary labels in binary indicator format.

y_score [array, shape = [n_samples, n_labels]] Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

coverage_error [float]

References

[1]

sklearn.metrics.label_ranking_average_precision_score

sklearn.metrics.**label_ranking_average_precision_score**(*y_true*, *y_score*, *sample_weight=None*)

Compute ranking-based average precision

Label ranking average precision (LRAP) is the average over each ground truth label assigned to each sample, of the ratio of true vs. total labels with lower score.

This metric is used in multilabel ranking problem, where the goal is to give better rank to the labels associated to each sample.

The obtained score is always strictly greater than 0 and the best value is 1.

Read more in the *User Guide*.

Parameters

y_true [array or sparse matrix, shape = [n_samples, n_labels]] True binary labels in binary indicator format.

y_score [array, shape = [n_samples, n_labels]] Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float]

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import label_ranking_average_precision_score
>>> y_true = np.array([[1, 0, 0], [0, 0, 1]])
>>> y_score = np.array([[0.75, 0.5, 1], [1, 0.2, 0.1]])
>>> label_ranking_average_precision_score(y_true, y_score)
0.416...
```

sklearn.metrics.label_ranking_loss

sklearn.metrics.**label_ranking_loss**(*y_true*, *y_score*, *sample_weight=None*)

Compute Ranking loss measure

Compute the average number of label pairs that are incorrectly ordered given y_score weighted by the size of the label set and the number of labels not in the label set.

This is similar to the error set size, but weighted by the number of relevant and irrelevant labels. The best performance is achieved with a ranking loss of zero.

Read more in the *User Guide*.

New in version 0.17: A function *label_ranking_loss*

Parameters

y_true [array or sparse matrix, shape = [n_samples, n_labels]] True binary labels in binary indicator format.

y_score [array, shape = [n_samples, n_labels]] Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

loss [float]

References

[1]

7.24.5 Clustering metrics

See the *Clustering performance evaluation* section of the user guide for further details.

The `sklearn.metrics.cluster` submodule contains evaluation metrics for cluster analysis results. There are two forms of evaluation:

- supervised, which uses a ground truth class values for each sample.
- unsupervised, which does not and measures the 'quality' of the model itself.

`metrics.adjusted_mutual_info_score`(...[, ...])	Adjusted Mutual Information between two clusterings.
`metrics.adjusted_rand_score`(labels_true, ...)	Rand index adjusted for chance.
`metrics.calinski_harabasz_score`(X, labels)	Compute the Calinski and Harabasz score.
`metrics.davies_bouldin_score`(X, labels)	Computes the Davies-Bouldin score.
`metrics.completeness_score`(labels_true, ...)	Completeness metric of a cluster labeling given a ground truth.
`metrics.cluster.contingency_matrix`(...[, ...])	Build a contingency matrix describing the relationship between labels.
`metrics.fowlkes_mallows_score`(labels_true, ...)	Measure the similarity of two clusterings of a set of points.
`metrics.homogeneity_completeness_v_measure`(...)	Compute the homogeneity and completeness and V-Measure scores at once.
`metrics.homogeneity_score`(labels_true, ...)	Homogeneity metric of a cluster labeling given a ground truth.
`metrics.mutual_info_score`(labels_true, ...)	Mutual Information between two clusterings.

Continued on next page

Table 192 – continued from previous page

metrics.normalized_mutual_info_score(...	[Normalized Mutual Information between two clusterings.
metrics.silhouette_score(X, labels[, ...])	Compute the mean Silhouette Coefficient of all samples.
metrics.silhouette_samples(X, labels[, metric])	Compute the Silhouette Coefficient for each sample.
metrics.v_measure_score(labels_true, labels_pred)	V-measure cluster labeling given a ground truth.

sklearn.metrics.adjusted_mutual_info_score

sklearn.metrics.**adjusted_mutual_info_score**(*labels_true*, *labels_pred*, *average_method='arithmetic'*)

Adjusted Mutual Information between two clusterings.

Adjusted Mutual Information (AMI) is an adjustment of the Mutual Information (MI) score to account for chance. It accounts for the fact that the MI is generally higher for two clusterings with a larger number of clusters, regardless of whether there is actually more information shared. For two clusterings U and V, the AMI is given as:

```
AMI(U, V) = [MI(U, V) - E(MI(U, V))] / [avg(H(U), H(V)) - E(MI(U, V))]
```

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is furthermore symmetric: switching `label_true` with `label_pred` will return the same score value. This can be useful to measure the agreement of two independent label assignments strategies on the same dataset when the real ground truth is not known.

Be mindful that this function is an order of magnitude slower than other metrics, such as the Adjusted Rand Index.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] A clustering of the data into disjoint subsets.

labels_pred [int array-like of shape (n_samples,)] A clustering of the data into disjoint subsets.

average_method [string, optional (default: 'arithmetic')] How to compute the normalizer in the denominator. Possible options are 'min', 'geometric', 'arithmetic', and 'max'.

New in version 0.20.

Changed in version 0.22: The default value of `average_method` changed from 'max' to 'arithmetic'.

Returns

ami: float (upperlimited by 1.0) The AMI returns a value of 1 when the two partitions are identical (ie perfectly matched). Random partitions (independent labellings) have an expected AMI around 0 on average hence can be negative.

See also:

adjusted_rand_score Adjusted Rand Index

mutual_info_score Mutual Information (not adjusted for chance)

References

[1], [2]

Examples

Perfect labelings are both homogeneous and complete, hence have score 1.0:

```
>>> from sklearn.metrics.cluster import adjusted_mutual_info_score
>>> adjusted_mutual_info_score([0, 0, 1, 1], [0, 0, 1, 1])
...
1.0
>>> adjusted_mutual_info_score([0, 0, 1, 1], [1, 1, 0, 0])
...
1.0
```

If classes members are completely split across different clusters, the assignment is totally in-complete, hence the AMI is null:

```
>>> adjusted_mutual_info_score([0, 0, 0, 0], [0, 1, 2, 3])
...
0.0
```

Examples using `sklearn.metrics.adjusted_mutual_info_score`

- *Demo of affinity propagation clustering algorithm*
- *Demo of DBSCAN clustering algorithm*
- *Adjustment for chance in clustering performance evaluation*
- *A demo of K-Means clustering on the handwritten digits data*

`sklearn.metrics.adjusted_rand_score`

sklearn.metrics.**adjusted_rand_score**(*labels_true*, *labels_pred*)

Rand index adjusted for chance.

The Rand Index computes a similarity measure between two clusterings by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusterings.

The raw RI score is then "adjusted for chance" into the ARI score using the following scheme:

```
ARI = (RI - Expected_RI) / (max(RI) - Expected_RI)
```

The adjusted Rand index is thus ensured to have a value close to 0.0 for random labeling independently of the number of clusters and samples and exactly 1.0 when the clusterings are identical (up to a permutation).

ARI is a symmetric measure:

```
adjusted_rand_score(a, b) == adjusted_rand_score(b, a)
```

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] Ground truth class labels to be used as a reference

labels_pred [array-like of shape (n_samples,)] Cluster labels to evaluate

Returns

ari [float] Similarity score between -1.0 and 1.0. Random labelings have an ARI close to 0.0. 1.0 stands for perfect match.

See also:

adjusted_mutual_info_score Adjusted Mutual Information

References

[Hubert1985], [wk]

Examples

Perfectly matching labelings have a score of 1 even

```
>>> from sklearn.metrics.cluster import adjusted_rand_score
>>> adjusted_rand_score([0, 0, 1, 1], [0, 0, 1, 1])
1.0
>>> adjusted_rand_score([0, 0, 1, 1], [1, 1, 0, 0])
1.0
```

Labelings that assign all classes members to the same clusters are complete be not always pure, hence penalized:

```
>>> adjusted_rand_score([0, 0, 1, 2], [0, 0, 1, 1])
0.57...
```

ARI is symmetric, so labelings that have pure clusters with members coming from the same classes but unnecessary splits are penalized:

```
>>> adjusted_rand_score([0, 0, 1, 1], [0, 0, 1, 2])
0.57...
```

If classes members are completely split across different clusters, the assignment is totally incomplete, hence the ARI is very low:

```
>>> adjusted_rand_score([0, 0, 0, 0], [0, 1, 2, 3])
0.0
```

Examples using `sklearn.metrics.adjusted_rand_score`

- *Demo of affinity propagation clustering algorithm*
- *Demo of DBSCAN clustering algorithm*
- *Adjustment for chance in clustering performance evaluation*
- *A demo of K-Means clustering on the handwritten digits data*
- *Clustering text documents using k-means*

sklearn.metrics.calinski_harabasz_score

sklearn.metrics.**calinski_harabasz_score**(*X*, *labels*)

Compute the Calinski and Harabasz score.

It is also known as the Variance Ratio Criterion.

The score is defined as ratio between the within-cluster dispersion and the between-cluster dispersion.

Read more in the *User Guide*.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

labels [array-like, shape (n_samples,)] Predicted labels for each sample.

Returns

score [float] The resulting Calinski-Harabasz score.

References

[1]

sklearn.metrics.davies_bouldin_score

sklearn.metrics.**davies_bouldin_score**(*X*, *labels*)

Computes the Davies-Bouldin score.

The score is defined as the average similarity measure of each cluster with its most similar cluster, where similarity is the ratio of within-cluster distances to between-cluster distances. Thus, clusters which are farther apart and less dispersed will result in a better score.

The minimum score is zero, with lower values indicating better clustering.

Read more in the *User Guide*.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

labels [array-like, shape (n_samples,)] Predicted labels for each sample.

Returns

score: float The resulting Davies-Bouldin score.

References

[1]

sklearn.metrics.completeness_score

sklearn.metrics.**completeness_score**(*labels_true*, *labels_pred*)

Completeness metric of a cluster labeling given a ground truth.

A clustering result satisfies completeness if all the data points that are members of a given class are elements of the same cluster.

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is not symmetric: switching `label_true` with `label_pred` will return the *homogeneity_score* which will be different in general.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] ground truth class labels to be used as a reference

labels_pred [array-like of shape (n_samples,)] cluster labels to evaluate

Returns

completeness [float] score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

See also:

homogeneity_score

v_measure_score

References

[1]

Examples

Perfect labelings are complete:

```
>>> from sklearn.metrics.cluster import completeness_score
>>> completeness_score([0, 0, 1, 1], [1, 1, 0, 0])
1.0
```

Non-perfect labelings that assign all classes members to the same clusters are still complete:

```
>>> print(completeness_score([0, 0, 1, 1], [0, 0, 0, 0]))
1.0
>>> print(completeness_score([0, 1, 2, 3], [0, 0, 1, 1]))
0.999...
```

If classes members are split across different clusters, the assignment cannot be complete:

```
>>> print(completeness_score([0, 0, 1, 1], [0, 1, 0, 1]))
0.0
>>> print(completeness_score([0, 0, 0, 0], [0, 1, 2, 3]))
0.0
```

Examples using sklearn.metrics.completeness_score

- *Demo of affinity propagation clustering algorithm*
- *Demo of DBSCAN clustering algorithm*
- *A demo of K-Means clustering on the handwritten digits data*
- *Clustering text documents using k-means*

sklearn.metrics.cluster.contingency_matrix

sklearn.metrics.cluster.**contingency_matrix**(*labels_true*, *labels_pred*, *eps=None*, *sparse=False*)

Build a contingency matrix describing the relationship between labels.

Parameters

labels_true [int array, shape = [n_samples]] Ground truth class labels to be used as a reference

labels_pred [array-like of shape (n_samples,)] Cluster labels to evaluate

eps [None or float, optional.] If a float, that value is added to all values in the contingency matrix. This helps to stop NaN propagation. If `None`, nothing is adjusted.

sparse [boolean, optional.] If True, return a sparse CSR continency matrix. If `eps is not None`, and `sparse is True`, will throw ValueError.

New in version 0.18.

Returns

contingency [{array-like, sparse}, shape=[n_classes_true, n_classes_pred]] Matrix C such that $C_{i,j}$ is the number of samples in true class i and in predicted class j. If `eps is None`, the dtype of this array will be integer. If `eps` is given, the dtype will be float. Will be a `scipy.sparse.csr_matrix` if `sparse=True`.

sklearn.metrics.fowlkes_mallows_score

sklearn.metrics.**fowlkes_mallows_score**(*labels_true*, *labels_pred*, *sparse=False*)

Measure the similarity of two clusterings of a set of points.

The Fowlkes-Mallows index (FMI) is defined as the geometric mean between of the precision and recall:

```
FMI = TP / sqrt((TP + FP) * (TP + FN))
```

Where `TP` is the number of **True Positive** (i.e. the number of pair of points that belongs in the same clusters in both `labels_true` and `labels_pred`), `FP` is the number of **False Positive** (i.e. the number of pair of points that belongs in the same clusters in `labels_true` and not in `labels_pred`) and `FN` is the number of **False Negative** (i.e the number of pair of points that belongs in the same clusters in `labels_pred` and not in `labels_True`).

The score ranges from 0 to 1. A high value indicates a good similarity between two clusters.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = (`n_samples`,)] A clustering of the data into disjoint subsets.

labels_pred [array, shape = (`n_samples`,)] A clustering of the data into disjoint subsets.

sparse [bool] Compute contingency matrix internally with sparse matrix.

Returns

score [float] The resulting Fowlkes-Mallows score.

References

[1], [2]

Examples

Perfect labelings are both homogeneous and complete, hence have score 1.0:

```
>>> from sklearn.metrics.cluster import fowlkes_mallows_score
>>> fowlkes_mallows_score([0, 0, 1, 1], [0, 0, 1, 1])
1.0
>>> fowlkes_mallows_score([0, 0, 1, 1], [1, 1, 0, 0])
1.0
```

If classes members are completely split across different clusters, the assignment is totally random, hence the FMI is null:

```
>>> fowlkes_mallows_score([0, 0, 0, 0], [0, 1, 2, 3])
0.0
```

sklearn.metrics.homogeneity_completeness_v_measure

sklearn.metrics.**homogeneity_completeness_v_measure**(*labels_true*, *labels_pred*, *beta=1.0*)

Compute the homogeneity and completeness and V-Measure scores at once.

Those metrics are based on normalized conditional entropy measures of the clustering labeling to evaluate given the knowledge of a Ground Truth class labels of the same samples.

A clustering result satisfies homogeneity if all of its clusters contain only data points which are members of a single class.

A clustering result satisfies completeness if all the data points that are members of a given class are elements of the same cluster.

Both scores have positive values between 0.0 and 1.0, larger values being desirable.

Those 3 metrics are independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score values in any way.

V-Measure is furthermore symmetric: swapping `labels_true` and `label_pred` will give the same score. This does not hold for homogeneity and completeness. V-Measure is identical to *normalized_mutual_info_score* with the arithmetic averaging method.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] ground truth class labels to be used as a reference

labels_pred [array-like of shape (n_samples,)] cluster labels to evaluate

beta [float] Ratio of weight attributed to `homogeneity` vs `completeness`. If `beta` is greater than 1, `completeness` is weighted more strongly in the calculation. If `beta` is less than 1, `homogeneity` is weighted more strongly.

Returns

homogeneity [float] score between 0.0 and 1.0. 1.0 stands for perfectly homogeneous labeling

completeness [float] score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

v_measure [float] harmonic mean of the first two

See also:

homogeneity_score

completeness_score

v_measure_score

sklearn.metrics.homogeneity_score

sklearn.metrics.**homogeneity_score**(*labels_true*, *labels_pred*)

Homogeneity metric of a cluster labeling given a ground truth.

A clustering result satisfies homogeneity if all of its clusters contain only data points which are members of a single class.

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is not symmetric: switching `label_true` with `label_pred` will return the *completeness_score* which will be different in general.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] ground truth class labels to be used as a reference

labels_pred [array-like of shape (n_samples,)] cluster labels to evaluate

Returns

homogeneity [float] score between 0.0 and 1.0. 1.0 stands for perfectly homogeneous labeling

See also:

completeness_score

v_measure_score

References

[1]

Examples

Perfect labelings are homogeneous:

```
>>> from sklearn.metrics.cluster import homogeneity_score
>>> homogeneity_score([0, 0, 1, 1], [1, 1, 0, 0])
1.0
```

Non-perfect labelings that further split classes into more clusters can be perfectly homogeneous:

```
>>> print("%.6f" % homogeneity_score([0, 0, 1, 1], [0, 0, 1, 2]))
1.000000
>>> print("%.6f" % homogeneity_score([0, 0, 1, 1], [0, 1, 2, 3]))
1.000000
```

Clusters that include samples from different classes do not make for an homogeneous labeling:

```
>>> print("%.6f" % homogeneity_score([0, 0, 1, 1], [0, 1, 0, 1]))
0.0...
>>> print("%.6f" % homogeneity_score([0, 0, 1, 1], [0, 0, 0, 0]))
0.0...
```

Examples using `sklearn.metrics.homogeneity_score`

- *Demo of affinity propagation clustering algorithm*
- *Demo of DBSCAN clustering algorithm*
- *A demo of K-Means clustering on the handwritten digits data*
- *Clustering text documents using k-means*

`sklearn.metrics`.mutual_info_score

`sklearn.metrics.`**`mutual_info_score`**(*labels_true*, *labels_pred*, *contingency=None*)

Mutual Information between two clusterings.

The Mutual Information is a measure of the similarity between two labels of the same data. Where $|U_i|$ is the number of the samples in cluster U_i and $|V_j|$ is the number of the samples in cluster V_j, the Mutual Information between clusterings U and V is given as:

$$MI(U, V) = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} \frac{|U_i \cap V_j|}{N} \log \frac{N|U_i \cap V_j|}{|U_i||V_j|}$$

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is furthermore symmetric: switching `label_true` with `label_pred` will return the same score value. This can be useful to measure the agreement of two independent label assignments strategies on the same dataset when the real ground truth is not known.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] A clustering of the data into disjoint subsets.

labels_pred [int array-like of shape (n_samples,)] A clustering of the data into disjoint subsets.

contingency [{None, array, sparse matrix}, shape = [n_classes_true, n_classes_pred]] A contingency matrix given by the `contingency_matrix` function. If value is `None`, it will be computed, otherwise the given value is used, with `labels_true` and `labels_pred` ignored.

Returns

mi [float] Mutual information, a non-negative value

See also:

adjusted_mutual_info_score Adjusted against chance Mutual Information

normalized_mutual_info_score Normalized Mutual Information

Notes

The logarithm used is the natural logarithm (base-e).

Examples using `sklearn.metrics.mutual_info_score`

- *Adjustment for chance in clustering performance evaluation*

`sklearn.metrics`.normalized_mutual_info_score

`sklearn.metrics.`**`normalized_mutual_info_score`**(*labels_true*, *labels_pred*, *average_method='arithmetic'*)

Normalized Mutual Information between two clusterings.

Normalized Mutual Information (NMI) is a normalization of the Mutual Information (MI) score to scale the results between 0 (no mutual information) and 1 (perfect correlation). In this function, mutual information is normalized by some generalized mean of `H(labels_true)` and `H(labels_pred))`, defined by the `average_method`.

This measure is not adjusted for chance. Therefore *`adjusted_mutual_info_score`* might be preferred.

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is furthermore symmetric: switching `label_true` with `label_pred` will return the same score value. This can be useful to measure the agreement of two independent label assignments strategies on the same dataset when the real ground truth is not known.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] A clustering of the data into disjoint subsets.

labels_pred [int array-like of shape (n_samples,)] A clustering of the data into disjoint subsets.

average_method [string, optional (default: 'arithmetic')] How to compute the normalizer in the denominator. Possible options are 'min', 'geometric', 'arithmetic', and 'max'.

New in version 0.20.

Changed in version 0.22: The default value of `average_method` changed from 'geometric' to 'arithmetic'.

Returns

nmi [float] score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

See also:

v_measure_score V-Measure (NMI with arithmetic mean option.)

adjusted_rand_score Adjusted Rand Index

adjusted_mutual_info_score Adjusted Mutual Information (adjusted against chance)

Examples

Perfect labelings are both homogeneous and complete, hence have score 1.0:

```
>>> from sklearn.metrics.cluster import normalized_mutual_info_score
>>> normalized_mutual_info_score([0, 0, 1, 1], [0, 0, 1, 1])
...
1.0
>>> normalized_mutual_info_score([0, 0, 1, 1], [1, 1, 0, 0])
...
1.0
```

If classes members are completely split across different clusters, the assignment is totally in-complete, hence the NMI is null:

```
>>> normalized_mutual_info_score([0, 0, 0, 0], [0, 1, 2, 3])
...
0.0
```

sklearn.metrics.silhouette_score

sklearn.metrics.**silhouette_score**(*X*, *labels*, *metric='euclidean'*, *sample_size=None*, *random_state=None*, ***kwds*)

Compute the mean Silhouette Coefficient of all samples.

The Silhouette Coefficient is calculated using the mean intra-cluster distance (`a`) and the mean nearest-cluster distance (`b`) for each sample. The Silhouette Coefficient for a sample is `(b - a) / max(a, b)`. To clarify, `b` is the distance between a sample and the nearest cluster that the sample is not a part of. Note that Silhouette Coefficient is only defined if number of labels is 2 <= n_labels <= n_samples - 1.

This function returns the mean Silhouette Coefficient over all samples. To obtain the values for each sample, use *silhouette_samples*.

The best value is 1 and the worst value is -1. Values near 0 indicate overlapping clusters. Negative values generally indicate that a sample has been assigned to the wrong cluster, as a different cluster is more similar.

Read more in the *User Guide*.

Parameters

X [array [n_samples_a, n_samples_a] if metric == "precomputed", or, [n_samples_a, n_features] otherwise] Array of pairwise distances between samples, or a feature array.

labels [array, shape = [n_samples]] Predicted labels for each sample.

metric [string, or callable] The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by `metrics.pairwise.pairwise_distances`. If X is the distance array itself, use `metric="precomputed"`.

sample_size [int or None] The size of the sample to use when computing the Silhouette Coefficient on a random subset of the data. If `sample_size is None`, no sampling is used.

random_state [int, RandomState instance or None, optional (default=None)] The generator used to randomly select a subset of samples. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `sample_size is not None`.

****kwds** [optional keyword parameters] Any further parameters are passed directly to the distance function. If using a scipy.spatial.distance metric, the parameters are still metric dependent. See the scipy docs for usage examples.

Returns

silhouette [float] Mean Silhouette Coefficient for all samples.

References

[1], [2]

Examples using `sklearn.metrics.silhouette_score`

- *Demo of affinity propagation clustering algorithm*
- *Demo of DBSCAN clustering algorithm*
- *A demo of K-Means clustering on the handwritten digits data*
- *Selecting the number of clusters with silhouette analysis on KMeans clustering*
- *Clustering text documents using k-means*

`sklearn.metrics.silhouette_samples`

`sklearn.metrics.`**`silhouette_samples`**(*X*, *labels*, *metric='euclidean'*, ***kwds*)

Compute the Silhouette Coefficient for each sample.

The Silhouette Coefficient is a measure of how well samples are clustered with samples that are similar to themselves. Clustering models with a high Silhouette Coefficient are said to be dense, where samples in the same cluster are similar to each other, and well separated, where samples in different clusters are not very similar to each other.

The Silhouette Coefficient is calculated using the mean intra-cluster distance (`a`) and the mean nearest-cluster distance (`b`) for each sample. The Silhouette Coefficient for a sample is `(b - a) / max(a, b)`. Note that Silhouette Coefficient is only defined if number of labels is 2 <= n_labels <= n_samples - 1.

This function returns the Silhouette Coefficient for each sample.

The best value is 1 and the worst value is -1. Values near 0 indicate overlapping clusters.

Read more in the *User Guide*.

Parameters

X [array [n_samples_a, n_samples_a] if metric == "precomputed", or, [n_samples_a, n_features] otherwise] Array of pairwise distances between samples, or a feature array.

labels [array, shape = [n_samples]] label values for each sample

metric [string, or callable] The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by `sklearn.metrics.pairwise.pairwise_distances`. If X is the distance array itself, use "precomputed" as the metric. Precomputed distance matrices must have 0 along the diagonal.

'kwds'** [optional keyword parameters] Any further parameters are passed directly to the distance function. If using a `scipy.spatial.distance` metric, the parameters are still metric dependent. See the scipy docs for usage examples.

Returns

silhouette [array, shape = [n_samples]] Silhouette Coefficient for each samples.

References

[1], [2]

Examples using `sklearn.metrics.silhouette_samples`

- *Selecting the number of clusters with silhouette analysis on KMeans clustering*

`sklearn.metrics.v_measure_score`

`sklearn.metrics.v_measure_score`(*labels_true*, *labels_pred*, *beta=1.0*)

V-measure cluster labeling given a ground truth.

This score is identical to *normalized_mutual_info_score* with the `'arithmetic'` option for averaging.

The V-measure is the harmonic mean between homogeneity and completeness:

```
v = (1 + beta) * homogeneity * completeness
     / (beta * homogeneity + completeness)
```

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is furthermore symmetric: switching `label_true` with `label_pred` will return the same score value. This can be useful to measure the agreement of two independent label assignments strategies on the same dataset when the real ground truth is not known.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] ground truth class labels to be used as a reference

labels_pred [array-like of shape (n_samples,)] cluster labels to evaluate

beta [float] Ratio of weight attributed to `homogeneity` vs `completeness`. If `beta` is greater than 1, `completeness` is weighted more strongly in the calculation. If `beta` is less than 1, `homogeneity` is weighted more strongly.

Returns

v_measure [float] score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

See also:

homogeneity_score

completeness_score

normalized_mutual_info_score

References

[1]

Examples

Perfect labelings are both homogeneous and complete, hence have score 1.0:

```
>>> from sklearn.metrics.cluster import v_measure_score
>>> v_measure_score([0, 0, 1, 1], [0, 0, 1, 1])
1.0
>>> v_measure_score([0, 0, 1, 1], [1, 1, 0, 0])
1.0
```

Labelings that assign all classes members to the same clusters are complete be not homogeneous, hence penalized:

```
>>> print("%.6f" % v_measure_score([0, 0, 1, 2], [0, 0, 1, 1]))
0.8...
>>> print("%.6f" % v_measure_score([0, 1, 2, 3], [0, 0, 1, 1]))
0.66...
```

Labelings that have pure clusters with members coming from the same classes are homogeneous but unnecessary splits harms completeness and thus penalize V-measure as well:

```
>>> print("%.6f" % v_measure_score([0, 0, 1, 1], [0, 0, 1, 2]))
0.8...
>>> print("%.6f" % v_measure_score([0, 0, 1, 1], [0, 1, 2, 3]))
0.66...
```

If classes members are completely split across different clusters, the assignment is totally incomplete, hence the V-Measure is null:

```
>>> print("%.6f" % v_measure_score([0, 0, 0, 0], [0, 1, 2, 3]))
0.0...
```

Clusters that include samples from totally different classes totally destroy the homogeneity of the labeling, hence:

```
>>> print("%.6f" % v_measure_score([0, 0, 1, 1], [0, 0, 0, 0]))
0.0...
```

Examples using `sklearn.metrics.v_measure_score`

- *Biclustering documents with the Spectral Co-clustering algorithm*
- *Demo of affinity propagation clustering algorithm*
- *Demo of DBSCAN clustering algorithm*

- *Adjustment for chance in clustering performance evaluation*
- *A demo of K-Means clustering on the handwritten digits data*
- *Clustering text documents using k-means*

7.24.6 Biclustering metrics

See the *Biclustering evaluation* section of the user guide for further details.

metrics.consensus_score(a, b[, similarity])	The similarity of two sets of biclusters.

sklearn.metrics.consensus_score

sklearn.metrics.**consensus_score**(*a*, *b*, *similarity='jaccard'*)

The similarity of two sets of biclusters.

Similarity between individual biclusters is computed. Then the best matching between sets is found using the Hungarian algorithm. The final score is the sum of similarities divided by the size of the larger set.

Read more in the *User Guide*.

Parameters

a [(rows, columns)] Tuple of row and column indicators for a set of biclusters.

b [(rows, columns)] Another set of biclusters like `a`.

similarity [string or function, optional, default: "jaccard"] May be the string "jaccard" to use the Jaccard coefficient, or any function that takes four arguments, each of which is a 1d indicator vector: (a_rows, a_columns, b_rows, b_columns).

References

- Hochreiter, Bodenhofer, et. al., 2010. FABIA: factor analysis for bicluster acquisition.

Examples using sklearn.metrics.consensus_score

- *A demo of the Spectral Co-Clustering algorithm*
- *A demo of the Spectral Biclustering algorithm*

7.24.7 Pairwise metrics

See the *Pairwise metrics, Affinities and Kernels* section of the user guide for further details.

metrics.pairwise.additive_chi2_kernel(X[, Y])	Computes the additive chi-squared kernel between observations in X and Y
metrics.pairwise.chi2_kernel(X[, Y, gamma])	Computes the exponential chi-squared kernel X and Y.
metrics.pairwise.cosine_similarity(X[, Y, ...])	Compute cosine similarity between samples in X and Y.

Continued on next page

Table 194 – continued from previous page

metrics.pairwise.cosine_distances(X[, Y])	Compute cosine distance between samples in X and Y.
metrics.pairwise.distance_metrics()	Valid metrics for pairwise_distances.
metrics.pairwise.euclidean_distances(X[, Y, ...])	Considering the rows of X (and Y=X) as vectors, compute the distance matrix between each pair of vectors.
metrics.pairwise.haversine_distances(X[, Y])	Compute the Haversine distance between samples in X and Y
metrics.pairwise.kernel_metrics()	Valid metrics for pairwise_kernels
metrics.pairwise.laplacian_kernel(X[, Y, gamma])	Compute the laplacian kernel between X and Y.
metrics.pairwise.linear_kernel(X[, Y, ...])	Compute the linear kernel between X and Y.
metrics.pairwise.manhattan_distances(X[, Y, ...])	Compute the L1 distances between the vectors in X and Y.
metrics.pairwise.nan_euclidean_distances(X)	Calculate the euclidean distances in the presence of missing values.
metrics.pairwise.pairwise_kernels(X[, Y, ...])	Compute the kernel between arrays X and optional array Y.
metrics.pairwise.polynomial_kernel(X[, Y, ...])	Compute the polynomial kernel between X and Y.
metrics.pairwise.rbf_kernel(X[, Y, gamma])	Compute the rbf (gaussian) kernel between X and Y.
metrics.pairwise.sigmoid_kernel(X[, Y, ...])	Compute the sigmoid kernel between X and Y.
metrics.pairwise.paired_euclidean_distances(X, Y)	Computes the paired euclidean distances between X and Y
metrics.pairwise.paired_manhattan_distances(X, Y)	Compute the L1 distances between the vectors in X and Y.
metrics.pairwise.paired_cosine_distances(X, Y)	Computes the paired cosine distances between X and Y
metrics.pairwise.paired_distances(X, Y[, metric])	Computes the paired distances between X and Y.
metrics.pairwise_distances(X[, Y, metric, ...])	Compute the distance matrix from a vector array X and optional Y.
metrics.pairwise_distances_argmin(X, Y[, ...])	Compute minimum distances between one point and a set of points.
metrics.pairwise_distances_argmin_min(X, Y)	Compute minimum distances between one point and a set of points.
metrics.pairwise_distances_chunked(X[, Y, ...])	Generate a distance matrix chunk by chunk with optional reduction

sklearn.metrics.pairwise.additive_chi2_kernel

sklearn.metrics.pairwise.**additive_chi2_kernel**(*X*, *Y=None*)

Computes the additive chi-squared kernel between observations in X and Y

The chi-squared kernel is computed between each pair of rows in X and Y. X and Y have to be non-negative. This kernel is most commonly applied to histograms.

The chi-squared kernel is given by:

```
k(x, y) = -Sum [(x - y)^2 / (x + y)]
```

It can be interpreted as a weighted difference per entry.

Read more in the *User Guide*.

Parameters

X [array-like of shape (n_samples_X, n_features)]

Y [array of shape (n_samples_Y, n_features)]

Returns

kernel_matrix [array of shape (n_samples_X, n_samples_Y)]

See also:

chi2_kernel The exponentiated version of the kernel, which is usually preferable.

sklearn.kernel_approximation.AdditiveChi2Sampler A Fourier approximation to this kernel.

Notes

As the negative of a distance, this kernel is only conditionally positive definite.

References

- Zhang, J. and Marszalek, M. and Lazebnik, S. and Schmid, C. Local features and kernels for classification of texture and object categories: A comprehensive study International Journal of Computer Vision 2007 https://research.microsoft.com/en-us/um/people/manik/projects/trade-off/papers/ZhangIJCV06.pdf

sklearn.metrics.pairwise.chi2_kernel

sklearn.metrics.pairwise.**chi2_kernel**(*X*, *Y=None*, *gamma=1.0*)

Computes the exponential chi-squared kernel X and Y.

The chi-squared kernel is computed between each pair of rows in X and Y. X and Y have to be non-negative. This kernel is most commonly applied to histograms.

The chi-squared kernel is given by:

```
k(x, y) = exp(-gamma Sum [(x - y)^2 / (x + y)])
```

It can be interpreted as a weighted difference per entry.

Read more in the *User Guide*.

Parameters

X [array-like of shape (n_samples_X, n_features)]

Y [array of shape (n_samples_Y, n_features)]

gamma [float, default=1.] Scaling parameter of the chi2 kernel.

Returns

kernel_matrix [array of shape (n_samples_X, n_samples_Y)]

See also:

additive_chi2_kernel The additive version of this kernel

sklearn.kernel_approximation.AdditiveChi2Sampler A Fourier approximation to the additive version of this kernel.

References

- Zhang, J. and Marszalek, M. and Lazebnik, S. and Schmid, C. Local features and kernels for classification of texture and object categories: A comprehensive study International Journal of Computer Vision 2007 https://research.microsoft.com/en-us/um/people/manik/projects/trade-off/papers/ZhangIJCV06.pdf

sklearn.metrics.pairwise.cosine_similarity

sklearn.metrics.pairwise.**cosine_similarity**(*X*, *Y=None*, *dense_output=True*)

Compute cosine similarity between samples in X and Y.

Cosine similarity, or the cosine kernel, computes similarity as the normalized dot product of X and Y:

K(X, Y) = <X, Y> / (||X||*||Y||)

On L2-normalized data, this function is equivalent to linear_kernel.

Read more in the *User Guide*.

Parameters

X [ndarray or sparse array, shape: (n_samples_X, n_features)] Input data.

Y [ndarray or sparse array, shape: (n_samples_Y, n_features)] Input data. If `None`, the output will be the pairwise similarities between all samples in `X`.

dense_output [boolean (optional), default True] Whether to return dense output even when the input is sparse. If `False`, the output is sparse if both input arrays are sparse.

New in version 0.17: parameter `dense_output` for dense output.

Returns

kernel matrix [array] An array with shape (n_samples_X, n_samples_Y).

sklearn.metrics.pairwise.cosine_distances

sklearn.metrics.pairwise.**cosine_distances**(*X*, *Y=None*)

Compute cosine distance between samples in X and Y.

Cosine distance is defined as 1.0 minus the cosine similarity.

Read more in the *User Guide*.

Parameters

X [array_like, sparse matrix] with shape (n_samples_X, n_features).

Y [array_like, sparse matrix (optional)] with shape (n_samples_Y, n_features).

Returns

distance matrix [array] An array with shape (n_samples_X, n_samples_Y).

See also:

sklearn.metrics.pairwise.cosine_similarity

scipy.spatial.distance.cosine dense matrices only

sklearn.metrics.pairwise.distance_metrics

sklearn.metrics.pairwise.**distance_metrics**()

Valid metrics for pairwise_distances.

This function simply returns the valid pairwise distance metrics. It exists to allow for a description of the mapping for each of the valid strings.

The valid distance metrics, and the function they map to, are:

metric	Function
'cityblock'	metrics.pairwise.manhattan_distances
'cosine'	metrics.pairwise.cosine_distances
'euclidean'	metrics.pairwise.euclidean_distances
'haversine'	metrics.pairwise.haversine_distances
'l1'	metrics.pairwise.manhattan_distances
'l2'	metrics.pairwise.euclidean_distances
'manhattan'	metrics.pairwise.manhattan_distances
'nan_euclidean'	metrics.pairwise.nan_euclidean_distances

Read more in the *User Guide*.

sklearn.metrics.pairwise.euclidean_distances

sklearn.metrics.pairwise.**euclidean_distances**(*X*, *Y=None*, *Y_norm_squared=None*, *squared=False*, *X_norm_squared=None*)

Considering the rows of X (and Y=X) as vectors, compute the distance matrix between each pair of vectors.

For efficiency reasons, the euclidean distance between a pair of row vector x and y is computed as:

```
dist(x, y) = sqrt(dot(x, x) - 2 * dot(x, y) + dot(y, y))
```

This formulation has two advantages over other ways of computing distances. First, it is computationally efficient when dealing with sparse data. Second, if one argument varies but the other remains unchanged, then `dot(x, x)` and/or `dot(y, y)` can be pre-computed.

However, this is not the most precise way of doing this computation, and the distance matrix returned by this function may not be exactly symmetric as required by, e.g., `scipy.spatial.distance` functions.

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix}, shape (n_samples_1, n_features)]

Y [{array-like, sparse matrix}, shape (n_samples_2, n_features)]

Y_norm_squared [array-like, shape (n_samples_2,), optional] Pre-computed dot-products of vectors in Y (e.g., `(Y**2).sum(axis=1)`) May be ignored in some cases, see the note below.

squared [boolean, optional] Return squared Euclidean distances.

X_norm_squared [array-like of shape (n_samples,), optional] Pre-computed dot-products of vectors in X (e.g., `(X**2).sum(axis=1)`) May be ignored in some cases, see the note below.

Returns

distances [array, shape (n_samples_1, n_samples_2)]

See also:

paired_distances distances betweens pairs of elements of X and Y.

Notes

To achieve better accuracy, `X_norm_squared` and `Y_norm_squared` may be unused if they are passed as `float32`.

Examples

```
>>> from sklearn.metrics.pairwise import euclidean_distances
>>> X = [[0, 1], [1, 1]]
>>> # distance between rows of X
>>> euclidean_distances(X, X)
array([[0., 1.],
       [1., 0.]])
>>> # get distance to origin
>>> euclidean_distances(X, [[0, 0]])
array([[1.        ],
       [1.41421356]])
```

sklearn.metrics.pairwise.haversine_distances

sklearn.metrics.pairwise.**haversine_distances**(*X*, *Y=None*)

Compute the Haversine distance between samples in X and Y

The Haversine (or great circle) distance is the angular distance between two points on the surface of a sphere. The first distance of each point is assumed to be the latitude, the second is the longitude, given in radians. The dimension of the data must be 2.

$$D(x, y) = 2\arcsin[\sqrt{\sin^2((x1 - y1)/2) + \cos(x1)\cos(y1)\sin^2((x2 - y2)/2)}]$$

Parameters

X [array_like, shape (n_samples_1, 2)]

Y [array_like, shape (n_samples_2, 2), optional]

Returns

distance [{array}, shape (n_samples_1, n_samples_2)]

Notes

As the Earth is nearly spherical, the haversine formula provides a good approximation of the distance between two points of the Earth surface, with a less than 1% error on average.

Examples

We want to calculate the distance between the Ezeiza Airport (Buenos Aires, Argentina) and the Charles de Gaulle Airport (Paris, France)

```
>>> from sklearn.metrics.pairwise import haversine_distances
>>> from math import radians
>>> bsas = [-34.83333, -58.5166646]
>>> paris = [49.0083899664, 2.53844117956]
>>> bsas_in_radians = [radians(_) for _ in bsas]
>>> paris_in_radians = [radians(_) for _ in paris]
>>> result = haversine_distances([bsas_in_radians, paris_in_radians])
>>> result * 6371000/1000  # multiply by Earth radius to get kilometers
array([[    0.        , 11099.54035582],
       [11099.54035582,     0.        ]])
```

sklearn.metrics.pairwise.kernel_metrics

sklearn.metrics.pairwise.**kernel_metrics**()

Valid metrics for pairwise_kernels

This function simply returns the valid pairwise distance metrics. It exists, however, to allow for a verbose description of the mapping for each of the valid strings.

The valid distance metrics, and the function they map to, are:

metric	Function
'additive_chi2'	sklearn.pairwise.additive_chi2_kernel
'chi2'	sklearn.pairwise.chi2_kernel
'linear'	sklearn.pairwise.linear_kernel
'poly'	sklearn.pairwise.polynomial_kernel
'polynomial'	sklearn.pairwise.polynomial_kernel
'rbf'	sklearn.pairwise.rbf_kernel
'laplacian'	sklearn.pairwise.laplacian_kernel
'sigmoid'	sklearn.pairwise.sigmoid_kernel
'cosine'	sklearn.pairwise.cosine_similarity

Read more in the *User Guide*.

sklearn.metrics.pairwise.laplacian_kernel

sklearn.metrics.pairwise.**laplacian_kernel**(*X*, *Y=None*, *gamma=None*)

Compute the laplacian kernel between X and Y.

The laplacian kernel is defined as:

```
K(x, y) = exp(-gamma ||x-y||_1)
```

for each pair of rows x in X and y in Y. Read more in the *User Guide*.

New in version 0.17.

Parameters

X [array of shape (n_samples_X, n_features)]

Y [array of shape (n_samples_Y, n_features)]

gamma [float, default None] If None, defaults to 1.0 / n_features

Returns

kernel_matrix [array of shape (n_samples_X, n_samples_Y)]

sklearn.metrics.pairwise.linear_kernel

sklearn.metrics.pairwise.**linear_kernel**(*X*, *Y=None*, *dense_output=True*)

Compute the linear kernel between X and Y.

Read more in the *User Guide*.

Parameters

X [array of shape (n_samples_1, n_features)]

Y [array of shape (n_samples_2, n_features)]

dense_output [boolean (optional), default True] Whether to return dense output even when the input is sparse. If `False`, the output is sparse if both input arrays are sparse.

New in version 0.20.

Returns

Gram matrix [array of shape (n_samples_1, n_samples_2)]

sklearn.metrics.pairwise.manhattan_distances

sklearn.metrics.pairwise.**manhattan_distances**(*X*, *Y=None*, *sum_over_features=True*)

Compute the L1 distances between the vectors in X and Y.

With sum_over_features equal to False it returns the componentwise distances.

Read more in the *User Guide*.

Parameters

X [array_like] An array with shape (n_samples_X, n_features).

Y [array_like, optional] An array with shape (n_samples_Y, n_features).

sum_over_features [bool, default=True] If True the function returns the pairwise distance matrix else it returns the componentwise L1 pairwise-distances. Not supported for sparse matrix inputs.

Returns

D [array] If sum_over_features is False shape is (n_samples_X * n_samples_Y, n_features) and D contains the componentwise L1 pairwise-distances (ie. absolute difference), else shape is (n_samples_X, n_samples_Y) and D contains the pairwise L1 distances.

Notes

When X and/or Y are CSR sparse matrices and they are not already in canonical format, this function modifies them in-place to make them canonical.

Examples

```
>>> from sklearn.metrics.pairwise import manhattan_distances
>>> manhattan_distances([[3]], [[3]])
array([[0.]])
>>> manhattan_distances([[3]], [[2]])
array([[1.]])
>>> manhattan_distances([[2]], [[3]])
array([[1.]])
>>> manhattan_distances([[1, 2], [3, 4]],         [[1, 2], [0, 3]])
array([[0., 2.],
       [4., 4.]])
>>> import numpy as np
>>> X = np.ones((1, 2))
>>> y = np.full((2, 2), 2.)
>>> manhattan_distances(X, y, sum_over_features=False)
array([[1., 1.],
       [1., 1.]])
```

sklearn.metrics.pairwise.nan_euclidean_distances

sklearn.metrics.pairwise.**nan_euclidean_distances**(*X*, *Y=None*, *squared=False*, *missing_values=nan*, *copy=True*)

Calculate the euclidean distances in the presence of missing values.

Compute the euclidean distance between each pair of samples in X and Y, where Y=X is assumed if Y=None. When calculating the distance between a pair of samples, this formulation ignores feature coordinates with a missing value in either sample and scales up the weight of the remaining coordinates:

> dist(x,y) = sqrt(weight * sq. distance from present coordinates) where, weight = Total # of coordinates / # of present coordinates

For example, the distance between `[3, na, na, 6]` and `[1, na, 4, 5]` is:

$$\sqrt{\frac{4}{2}((3-1)^2+(6-5)^2)}$$

If all the coordinates are missing or if there are no common present coordinates then NaN is returned for that pair.

Read more in the *User Guide*.

New in version 0.22.

Parameters

X [array-like, shape=(n_samples_1, n_features)]

Y [array-like, shape=(n_samples_2, n_features)]

squared [bool, default=False] Return squared Euclidean distances.

missing_values [np.nan or int, default=np.nan] Representation of missing value

copy [boolean, default=True] Make and use a deep copy of X and Y (if Y exists)

Returns

distances [array, shape (n_samples_1, n_samples_2)]

See also:

paired_distances distances between pairs of elements of X and Y.

References

- John K. Dixon, "Pattern Recognition with Partly Missing Data", IEEE Transactions on Systems, Man, and Cybernetics, Volume: 9, Issue: 10, pp. 617 - 621, Oct. 1979. http://ieeexplore.ieee.org/abstract/document/4310090/

Examples

```
>>> from sklearn.metrics.pairwise import nan_euclidean_distances
>>> nan = float("NaN")
>>> X = [[0, 1], [1, nan]]
>>> nan_euclidean_distances(X, X) # distance between rows of X
array([[0.        , 1.41421356],
       [1.41421356, 0.        ]])
```

```
>>> # get distance to origin
>>> nan_euclidean_distances(X, [[0, 0]])
array([[1.        ],
       [1.41421356]])
```

sklearn.metrics.pairwise.pairwise_kernels

sklearn.metrics.pairwise.**pairwise_kernels**(*X*, *Y=None*, *metric='linear'*, *filter_params=False*, *n_jobs=None*, ***kwds*)

Compute the kernel between arrays X and optional array Y.

This method takes either a vector array or a kernel matrix, and returns a kernel matrix. If the input is a vector array, the kernels are computed. If the input is a kernel matrix, it is returned instead.

This method provides a safe way to take a kernel matrix as input, while preserving compatibility with many other algorithms that take a vector array.

If Y is given (default is None), then the returned matrix is the pairwise kernel between the arrays from both X and Y.

Valid values for metric are: ['additive_chi2', 'chi2', 'linear', 'poly', 'polynomial', 'rbf', 'laplacian', 'sigmoid', 'cosine']

Read more in the *User Guide*.

Parameters

X [array [n_samples_a, n_samples_a] if metric == "precomputed", or, [n_samples_a, n_features] otherwise] Array of pairwise kernels between samples, or a feature array.

Y [array [n_samples_b, n_features]] A second feature array only if X has shape [n_samples_a, n_features].

metric [string, or callable] The metric to use when calculating kernel between instances in a feature array. If metric is a string, it must be one of the metrics in pairwise.PAIRWISE_KERNEL_FUNCTIONS. If metric is "precomputed", X is assumed to be a kernel matrix. Alternatively, if metric is a callable function, it is called on each pair

of instances (rows) and the resulting value recorded. The callable should take two rows from X as input and return the corresponding kernel value as a single number. This means that callables from `sklearn.metrics.pairwise` are not allowed, as they operate on matrices, not single samples. Use the string identifying the kernel instead.

filter_params [boolean] Whether to filter invalid parameters or not.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by breaking down the pairwise matrix into n_jobs even slices and computing them in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

****kwds** [optional keyword parameters] Any further parameters are passed directly to the kernel function.

Returns

K [array [n_samples_a, n_samples_a] or [n_samples_a, n_samples_b]] A kernel matrix K such that K_{i, j} is the kernel between the ith and jth vectors of the given matrix X, if Y is None. If Y is not None, then K_{i, j} is the kernel between the ith array from X and the jth array from Y.

Notes

If metric is 'precomputed', Y is ignored and X is returned.

sklearn.metrics.pairwise.polynomial_kernel

sklearn.metrics.pairwise.**polynomial_kernel**(*X*, *Y=None*, *degree=3*, *gamma=None*, *coef0=1*)

Compute the polynomial kernel between X and Y:

```
K(X, Y) = (gamma <X, Y> + coef0)^degree
```

Read more in the *User Guide*.

Parameters

X [ndarray of shape (n_samples_1, n_features)]

Y [ndarray of shape (n_samples_2, n_features)]

degree [int, default 3]

gamma [float, default None] if None, defaults to 1.0 / n_features

coef0 [float, default 1]

Returns

Gram matrix [array of shape (n_samples_1, n_samples_2)]

sklearn.metrics.pairwise.rbf_kernel

sklearn.metrics.pairwise.**rbf_kernel**(*X*, *Y=None*, *gamma=None*)

Compute the rbf (gaussian) kernel between X and Y:

```
K(x, y) = exp(-gamma ||x-y||^2)
```

for each pair of rows x in X and y in Y.

Read more in the *User Guide*.

Parameters

X [array of shape (n_samples_X, n_features)]

Y [array of shape (n_samples_Y, n_features)]

gamma [float, default None] If None, defaults to 1.0 / n_features

Returns

kernel_matrix [array of shape (n_samples_X, n_samples_Y)]

sklearn.metrics.pairwise.sigmoid_kernel

sklearn.metrics.pairwise.**sigmoid_kernel**(*X*, *Y=None*, *gamma=None*, *coef0=1*)

Compute the sigmoid kernel between X and Y:

```
K(X, Y) = tanh(gamma <X, Y> + coef0)
```

Read more in the *User Guide*.

Parameters

X [ndarray of shape (n_samples_1, n_features)]

Y [ndarray of shape (n_samples_2, n_features)]

gamma [float, default None] If None, defaults to 1.0 / n_features

coef0 [float, default 1]

Returns

Gram matrix [array of shape (n_samples_1, n_samples_2)]

sklearn.metrics.pairwise.paired_euclidean_distances

sklearn.metrics.pairwise.**paired_euclidean_distances**(*X*, *Y*)

Computes the paired euclidean distances between X and Y

Read more in the *User Guide*.

Parameters

X [array-like, shape (n_samples, n_features)]

Y [array-like, shape (n_samples, n_features)]

Returns

distances [ndarray (n_samples,)]

sklearn.metrics.pairwise.paired_manhattan_distances

sklearn.metrics.pairwise.**paired_manhattan_distances**(*X*, *Y*)

Compute the L1 distances between the vectors in X and Y.

Read more in the *User Guide*.

Parameters

X [array-like, shape (n_samples, n_features)]

Y [array-like, shape (n_samples, n_features)]

Returns

distances [ndarray (n_samples,)]

sklearn.metrics.pairwise.paired_cosine_distances

sklearn.metrics.pairwise.**paired_cosine_distances**(*X*, *Y*)

Computes the paired cosine distances between X and Y

Read more in the *User Guide*.

Parameters

X [array-like, shape (n_samples, n_features)]

Y [array-like, shape (n_samples, n_features)]

Returns

distances [ndarray, shape (n_samples,)]

Notes

The cosine distance is equivalent to the half the squared euclidean distance if each sample is normalized to unit norm

sklearn.metrics.pairwise.paired_distances

sklearn.metrics.pairwise.**paired_distances**(*X*, *Y*, *metric='euclidean'*, ***kwds*)

Computes the paired distances between X and Y.

Computes the distances between (X[0], Y[0]), (X[1], Y[1]), etc...

Read more in the *User Guide*.

Parameters

X [ndarray (n_samples, n_features)] Array 1 for distance computation.

Y [ndarray (n_samples, n_features)] Array 2 for distance computation.

metric [string or callable] The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options specified in PAIRED_DISTANCES, including "euclidean", "manhattan", or "cosine". Alternatively, if metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays from X as input and return a value indicating the distance between them.

Returns

distances [ndarray (n_samples,)]

See also:

pairwise_distances Computes the distance between every pair of samples

Examples

```
>>> from sklearn.metrics.pairwise import paired_distances
>>> X = [[0, 1], [1, 1]]
>>> Y = [[0, 1], [2, 1]]
>>> paired_distances(X, Y)
array([0., 1.])
```

sklearn.metrics.pairwise_distances

sklearn.metrics.**pairwise_distances**(*X*, *Y=None*, *metric='euclidean'*, *n_jobs=None*, *force_all_finite=True*, ***kwds*)

Compute the distance matrix from a vector array X and optional Y.

This method takes either a vector array or a distance matrix, and returns a distance matrix. If the input is a vector array, the distances are computed. If the input is a distances matrix, it is returned instead.

This method provides a safe way to take a distance matrix as input, while preserving compatibility with many other algorithms that take a vector array.

If Y is given (default is None), then the returned matrix is the pairwise distance between the arrays from both X and Y.

Valid values for metric are:

- From scikit-learn: ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']. These metrics support sparse matrix inputs. ['nan_euclidean'] but it does not yet support sparse matrices.
- From scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule'] See the documentation for scipy.spatial.distance for details on these metrics. These metrics do not support sparse matrix inputs.

Note that in the case of 'cityblock', 'cosine' and 'euclidean' (which are valid scipy.spatial.distance metrics), the scikit-learn implementation will be used, which is faster and has support for sparse matrices (except for 'cityblock'). For a verbose description of the metrics from scikit-learn, see the __doc__ of the sklearn.pairwise.distance_metrics function.

Read more in the *User Guide*.

Parameters

X [array [n_samples_a, n_samples_a] if metric == "precomputed", or, [n_samples_a, n_features] otherwise] Array of pairwise distances between samples, or a feature array.

Y [array [n_samples_b, n_features], optional] An optional second feature array. Only allowed if metric != "precomputed".

metric [string, or callable] The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by scipy.spatial.distance.pdist for its metric parameter, or a metric listed in pairwise.PAIRWISE_DISTANCE_FUNCTIONS. If metric is "precomputed", X is assumed to be a distance matrix. Alternatively, if metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays from X as input and return a value indicating the distance between them.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by breaking down the pairwise matrix into n_jobs even slices and computing them in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

force_all_finite [boolean or 'allow-nan', (default=True)] Whether to raise an error on np.inf and np.nan in array. The possibilities are:

- True: Force all values of array to be finite.
- False: accept both np.inf and np.nan in array.
- 'allow-nan': accept only np.nan values in array. Values cannot be infinite.

New in version 0.22.

****kwds** [optional keyword parameters] Any further parameters are passed directly to the distance function. If using a scipy.spatial.distance metric, the parameters are still metric dependent. See the scipy docs for usage examples.

Returns

D [array [n_samples_a, n_samples_a] or [n_samples_a, n_samples_b]] A distance matrix D such that D_{i, j} is the distance between the ith and jth vectors of the given matrix X, if Y is None. If Y is not None, then D_{i, j} is the distance between the ith array from X and the jth array from Y.

See also:

pairwise_distances_chunked performs the same calculation as this function, but returns a generator of chunks of the distance matrix, in order to limit memory usage.

paired_distances Computes the distances between corresponding elements of two arrays

Examples using `sklearn.metrics.pairwise_distances`

- *Agglomerative clustering with different metrics*

sklearn.metrics.pairwise_distances_argmin

sklearn.metrics.**pairwise_distances_argmin**(*X*, *Y*, *axis=1*, *metric='euclidean'*, *metric_kwargs=None*)

Compute minimum distances between one point and a set of points.

This function computes for each row in X, the index of the row of Y which is closest (according to the specified distance).

This is mostly equivalent to calling:

pairwise_distances(X, Y=Y, metric=metric).argmin(axis=axis)

but uses much less memory, and is faster for large arrays.

This function works with dense 2D arrays only.

Parameters

X [array-like] Arrays containing points. Respective shapes (n_samples1, n_features) and (n_samples2, n_features)

Y [array-like] Arrays containing points. Respective shapes (n_samples1, n_features) and (n_samples2, n_features)

axis [int, optional, default 1] Axis along which the argmin and distances are to be computed.

metric [string or callable] metric to use for distance computation. Any metric from scikit-learn or scipy.spatial.distance can be used.

If metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays as input and return one value indicating the distance between them. This works for Scipy's metrics, but is less efficient than passing the metric name as a string.

Distance matrices are not supported.

Valid values for metric are:

- from scikit-learn: ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']
- from scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule']

See the documentation for scipy.spatial.distance for details on these metrics.

metric_kwargs [dict] keyword arguments to pass to specified metric function.

Returns

argmin [numpy.ndarray] Y[argmin[i], :] is the row in Y that is closest to X[i, :].

See also:

sklearn.metrics.pairwise_distances

sklearn.metrics.pairwise_distances_argmin_min

Examples using `sklearn.metrics.pairwise_distances_argmin`

- *Color Quantization using K-Means*
- *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*

`sklearn.metrics.`pairwise_distances_argmin_min

`sklearn.metrics.`**`pairwise_distances_argmin_min`**(*X*, *Y*, *axis=1*, *metric='euclidean'*, *metric_kwargs=None*)

Compute minimum distances between one point and a set of points.

This function computes for each row in X, the index of the row of Y which is closest (according to the specified distance). The minimal distances are also returned.

This is mostly equivalent to calling:

(pairwise_distances(X, Y=Y, metric=metric).argmin(axis=axis), pairwise_distances(X, Y=Y, metric=metric).min(axis=axis))

but uses much less memory, and is faster for large arrays.

Parameters

X [{array-like, sparse matrix}, shape (n_samples1, n_features)] Array containing points.

Y [{array-like, sparse matrix}, shape (n_samples2, n_features)] Arrays containing points.

axis [int, optional, default 1] Axis along which the argmin and distances are to be computed.

metric [string or callable, default 'euclidean'] metric to use for distance computation. Any metric from scikit-learn or scipy.spatial.distance can be used.

If metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays as input and return one value indicating the distance between them. This works for Scipy's metrics, but is less efficient than passing the metric name as a string.

Distance matrices are not supported.

Valid values for metric are:

- from scikit-learn: ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']
- from scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule']

See the documentation for scipy.spatial.distance for details on these metrics.

metric_kwargs [dict, optional] Keyword arguments to pass to specified metric function.

Returns

argmin [numpy.ndarray] Y[argmin[i], :] is the row in Y that is closest to X[i, :].

distances [numpy.ndarray] distances[i] is the distance between the i-th row in X and the argmin[i]-th row in Y.

See also:

sklearn.metrics.pairwise_distances

sklearn.metrics.pairwise_distances_argmin

sklearn.metrics.pairwise_distances_chunked

sklearn.metrics.**pairwise_distances_chunked**(*X*, *Y=None*, *reduce_func=None*, *metric='euclidean'*, *n_jobs=None*, *working_memory=None*, ***kwds*)

Generate a distance matrix chunk by chunk with optional reduction

In cases where not all of a pairwise distance matrix needs to be stored at once, this is used to calculate pairwise distances in `working_memory`-sized chunks. If `reduce_func` is given, it is run on each chunk and its return values are concatenated into lists, arrays or sparse matrices.

Parameters

X [array [n_samples_a, n_samples_a] if metric == "precomputed", or,] [n_samples_a, n_features] otherwise Array of pairwise distances between samples, or a feature array.

Y [array [n_samples_b, n_features], optional] An optional second feature array. Only allowed if metric != "precomputed".

reduce_func [callable, optional] The function which is applied on each chunk of the distance matrix, reducing it to needed values. `reduce_func(D_chunk, start)` is called repeatedly, where `D_chunk` is a contiguous vertical slice of the pairwise distance matrix, starting at row `start`. It should return one of: None; an array, a list, or a sparse matrix of length `D_chunk.shape[0]`; or a tuple of such objects. Returning None is useful for in-place operations, rather than reductions.

If None, pairwise_distances_chunked returns a generator of vertical chunks of the distance matrix.

metric [string, or callable] The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by scipy.spatial.distance.pdist for its metric parameter, or a metric listed in pairwise.PAIRWISE_DISTANCE_FUNCTIONS. If metric is "precomputed", X is assumed to be a distance matrix. Alternatively, if metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays from X as input and return a value indicating the distance between them.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by breaking down the pairwise matrix into n_jobs even slices and computing them in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

working_memory [int, optional] The sought maximum memory for temporary distance matrix chunks. When None (default), the value of `sklearn.get_config()['working_memory']` is used.

'kwds'** [optional keyword parameters] Any further parameters are passed directly to the distance function. If using a scipy.spatial.distance metric, the parameters are still metric dependent. See the scipy docs for usage examples.

Yields

D_chunk [array or sparse matrix] A contiguous slice of distance matrix, optionally processed by `reduce_func`.

Examples

Without reduce_func:

```
>>> import numpy as np
>>> from sklearn.metrics import pairwise_distances_chunked
>>> X = np.random.RandomState(0).rand(5, 3)
>>> D_chunk = next(pairwise_distances_chunked(X))
>>> D_chunk
array([[0.  ..., 0.29..., 0.41..., 0.19..., 0.57...],
       [0.29..., 0.  ..., 0.57..., 0.41..., 0.76...],
       [0.41..., 0.57..., 0.  ..., 0.44..., 0.90...],
       [0.19..., 0.41..., 0.44..., 0.  ..., 0.51...],
       [0.57..., 0.76..., 0.90..., 0.51..., 0.  ...]])
```

Retrieve all neighbors and average distance within radius r:

```
>>> r = .2
>>> def reduce_func(D_chunk, start):
...     neigh = [np.flatnonzero(d < r) for d in D_chunk]
...     avg_dist = (D_chunk * (D_chunk < r)).mean(axis=1)
...     return neigh, avg_dist
>>> gen = pairwise_distances_chunked(X, reduce_func=reduce_func)
>>> neigh, avg_dist = next(gen)
>>> neigh
[array([0, 3]), array([1]), array([2]), array([0, 3]), array([4])]
>>> avg_dist
array([0.039..., 0.        , 0.        , 0.039..., 0.        ])
```

Where r is defined per sample, we need to make use of `start`:

```
>>> r = [.2, .4, .4, .3, .1]
>>> def reduce_func(D_chunk, start):
...     neigh = [np.flatnonzero(d < r[i])
...              for i, d in enumerate(D_chunk, start)]
...     return neigh
>>> neigh = next(pairwise_distances_chunked(X, reduce_func=reduce_func))
>>> neigh
[array([0, 3]), array([0, 1]), array([2]), array([0, 3]), array([4])]
```

Force row-by-row generation by reducing `working_memory`:

```
>>> gen = pairwise_distances_chunked(X, reduce_func=reduce_func,
...                                  working_memory=0)
>>> next(gen)
[array([0, 3])]
>>> next(gen)
[array([0, 1])]
```

7.24.8 Plotting

See the *Visualizations* section of the user guide for further details.

metrics.plot_confusion_matrix(estimator, X, ...)	Plot Confusion Matrix.
metrics.plot_precision_recall_curve(..., [, ...])	Plot Precision Recall Curve for binary classifiers.
metrics.plot_roc_curve(estimator, X, y[, ...])	Plot Receiver operating characteristic (ROC) curve.

sklearn.metrics.plot_confusion_matrix

sklearn.metrics.**plot_confusion_matrix**(*estimator*, *X*, *y_true*, *labels=None*, *sample_weight=None*, *normalize=None*, *display_labels=None*, *include_values=True*, *xticks_rotation='horizontal'*, *values_format=None*, *cmap='viridis'*, *ax=None*)

Plot Confusion Matrix.

Read more in the *User Guide*.

Parameters

estimator [estimator instance] Trained classifier.

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Input values.

y [array-like of shape (n_samples,)] Target values.

labels [array-like of shape (n_classes,), default=None] List of labels to index the matrix. This may be used to reorder or select a subset of labels. If `None` is given, those that appear at least once in `y_true` or `y_pred` are used in sorted order.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

normalize [{'true', 'pred', 'all'}, default=None] Normalizes confusion matrix over the true (rows), predicted (columns) conditions or all the population. If None, confusion matrix will not be normalized.

display_labels [array-like of shape (n_classes,), default=None] Target names used for plotting. By default, `labels` will be used if it is defined, otherwise the unique labels of `y_true` and `y_pred` will be used.

include_values [bool, default=True] Includes values in confusion matrix.

xticks_rotation [{'vertical', 'horizontal'} or float, default='horizontal'] Rotation of xtick labels.

values_format [str, default=None] Format specification for values in confusion matrix. If `None`, the format specification is '.2g'.

cmap [str or matplotlib Colormap, default='viridis'] Colormap recognized by matplotlib.

ax [matplotlib Axes, default=None] Axes object to plot on. If `None`, a new figure and axes is created.

Returns

display [*ConfusionMatrixDisplay*]

Examples using `sklearn.metrics.plot_confusion_matrix`

- *Recognizing hand-written digits*
- *Confusion matrix*

`sklearn.metrics.plot_precision_recall_curve`

`sklearn.metrics.plot_precision_recall_curve`(*estimator*, *X*, *y*, *sample_weight=None*, *response_method='auto'*, *name=None*, *ax=None*, ***kwargs*)

Plot Precision Recall Curve for binary classifiers.

Extra keyword arguments will be passed to matplotlib's `plot`.

Read more in the *User Guide*.

Parameters

estimator [estimator instance] Trained classifier.

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Input values.

y [array-like of shape (n_samples,)] Binary target values.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

response_method [{'predict_proba', 'decision_function', 'auto'}, default='auto'] Specifies whether to use *predict_proba* or *decision_function* as the target response. If set to 'auto', *predict_proba* is tried first and if it does not exist *decision_function* is tried next.

name [str, default=None] Name for labeling curve. If `None`, the name of the estimator is used.

ax [matplotlib axes, default=None] Axes object to plot on. If `None`, a new figure and axes is created.

****kwargs** [dict] Keyword arguments to be passed to matplotlib's `plot`.

Returns

display [*PrecisionRecallDisplay*] Object that stores computed values.

Examples using `sklearn.metrics.plot_precision_recall_curve`

- *Precision-Recall*

`sklearn.metrics.plot_roc_curve`

`sklearn.metrics.plot_roc_curve`(*estimator*, *X*, *y*, *sample_weight=None*, *drop_intermediate=True*, *response_method='auto'*, *name=None*, *ax=None*, ***kwargs*)

Plot Receiver operating characteristic (ROC) curve.

Extra keyword arguments will be passed to matplotlib's `plot`.

Read more in the *User Guide*.

Parameters

estimator [estimator instance] Trained classifier.

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Input values.

y [array-like of shape (n_samples,)] Target values.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

drop_intermediate [boolean, default=True] Whether to drop some suboptimal thresholds which would not appear on a plotted ROC curve. This is useful in order to create lighter ROC curves.

response_method [{'predict_proba', 'decision_function', 'auto'} default='auto'] Specifies whether to use *predict_proba* or *decision_function* as the target response. If set to 'auto', *predict_proba* is tried first and if it does not exist *decision_function* is tried next.

name [str, default=None] Name of ROC Curve for labeling. If `None`, use the name of the estimator.

ax [matplotlib axes, default=None] Axes object to plot on. If `None`, a new figure and axes is created.

Returns

display [*RocCurveDisplay*] Object that stores computed values.

Examples

```
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> from sklearn import datasets, metrics, model_selection, svm
>>> X, y = datasets.make_classification(random_state=0)
>>> X_train, X_test, y_train, y_test = model_selection.train_test_split(
↪  X, y, random_state=0)
>>> clf = svm.SVC(random_state=0)
>>> clf.fit(X_train, y_train)
SVC(random_state=0)
>>> metrics.plot_roc_curve(clf, X_test, y_test)  # doctest: +SKIP
>>> plt.show()                                   # doctest: +SKIP
```

Examples using `sklearn.metrics.plot_roc_curve`

- *ROC Curve with Visualization API*
- *Receiver Operating Characteristic (ROC) with cross validation*
- *Release Highlights for scikit-learn 0.22*

metrics.ConfusionMatrixDisplay(...)	Confusion Matrix visualization.
metrics.PrecisionRecallDisplay(precision, ...)	Precision Recall visualization.
metrics.RocCurveDisplay(fpr, tpr, roc_auc, ...)	ROC Curve visualization.

sklearn.metrics.ConfusionMatrixDisplay

class sklearn.metrics.**ConfusionMatrixDisplay**(*confusion_matrix*, *display_labels*)

Confusion Matrix visualization.

It is recommend to use *plot_confusion_matrix* to create a *ConfusionMatrixDisplay*. All parameters are stored as attributes.

Read more in the *User Guide*.

Parameters

confusion_matrix [ndarray of shape (n_classes, n_classes)] Confusion matrix.

display_labels [ndarray of shape (n_classes,)] Display labels for plot.

Attributes

im_ [matplotlib AxesImage] Image representing the confusion matrix.

text_ [ndarray of shape (n_classes, n_classes), dtype=matplotlib Text, or None] Array of matplotlib axes. `None` if `include_values` is false.

ax_ [matplotlib Axes] Axes with confusion matrix.

figure_ [matplotlib Figure] Figure containing the confusion matrix.

Methods

plot(self[, include_values, cmap, ...])	Plot visualization.

__init__(*self*, *confusion_matrix*, *display_labels*)
Initialize self. See help(type(self)) for accurate signature.

plot(*self*, *include_values=True*, *cmap='viridis'*, *xticks_rotation='horizontal'*, *values_format=None*, *ax=None*)
Plot visualization.

Parameters

include_values [bool, default=True] Includes values in confusion matrix.

cmap [str or matplotlib Colormap, default='viridis'] Colormap recognized by matplotlib.

xticks_rotation [{'vertical', 'horizontal'} or float, default='horizontal'] Rotation of xtick labels.

values_format [str, default=None] Format specification for values in confusion matrix. If `None`, the format specification is '.2g'.

ax [matplotlib axes, default=None] Axes object to plot on. If `None`, a new figure and axes is created.

Returns

display [*ConfusionMatrixDisplay*]

sklearn.metrics.PrecisionRecallDisplay

class sklearn.metrics.**PrecisionRecallDisplay**(*precision*, *recall*, *average_precision*, *estimator_name*)
Precision Recall visualization.

It is recommend to use *plot_precision_recall_curve* to create a visualizer. All parameters are stored as attributes.

Read more in the *User Guide*.

Parameters

precision [ndarray] Precision values.

recall [ndarray] Recall values.

average_precision [float] Average precision.

estimator_name [str] Name of estimator.

Attributes

line_ [matplotlib Artist] Precision recall curve.

ax_ [matplotlib Axes] Axes with precision recall curve.

figure_ [matplotlib Figure] Figure containing the curve.

Methods

plot(self[, ax, name])	Plot visualization.

__init__ (*self*, *precision*, *recall*, *average_precision*, *estimator_name*)
Initialize self. See help(type(self)) for accurate signature.

plot (*self*, *ax=None*, *name=None*, ***kwargs*)
Plot visualization.

Extra keyword arguments will be passed to matplotlib's `plot`.

Parameters

ax [Matplotlib Axes, default=None] Axes object to plot on. If `None`, a new figure and axes is created.

name [str, default=None] Name of precision recall curve for labeling. If `None`, use the name of the estimator.

****kwargs** [dict] Keyword arguments to be passed to matplotlib's `plot`.

Returns

display [*PrecisionRecallDisplay*] Object that stores computed values.

sklearn.metrics.RocCurveDisplay

class sklearn.metrics.**RocCurveDisplay** (*fpr*, *tpr*, *roc_auc*, *estimator_name*)
ROC Curve visualization.

It is recommend to use *plot_roc_curve* to create a visualizer. All parameters are stored as attributes.

Read more in the *User Guide*.

Parameters

fpr [ndarray] False positive rate.

tpr [ndarray] True positive rate.

roc_auc [float] Area under ROC curve.

estimator_name [str] Name of estimator.

Attributes

line_ [matplotlib Artist] ROC Curve.

ax_ [matplotlib Axes] Axes with ROC Curve.

figure_ [matplotlib Figure] Figure containing the curve.

Examples

```
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> import numpy as np
>>> from sklearn import metrics
>>> y = np.array([0, 0, 1, 1])
>>> pred = np.array([0.1, 0.4, 0.35, 0.8])
>>> fpr, tpr, thresholds = metrics.roc_curve(y, pred)
>>> roc_auc = metrics.auc(fpr, tpr)
>>> display = metrics.RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc,
↪                                   estimator_name='example estimator')
>>> display.plot()  # doctest: +SKIP
>>> plt.show()      # doctest: +SKIP
```

Methods

plot(self[, ax, name])	Plot visualization

__init__ (*self*, *fpr*, *tpr*, *roc_auc*, *estimator_name*)
Initialize self. See help(type(self)) for accurate signature.

plot (*self*, *ax=None*, *name=None*, ***kwargs*)
Plot visualization

Extra keyword arguments will be passed to matplotlib's `plot`.

Parameters

ax [matplotlib axes, default=None] Axes object to plot on. If `None`, a new figure and axes is created.

name [str, default=None] Name of ROC Curve for labeling. If `None`, use the name of the estimator.

Returns

display [`RocCurveDisplay`] Object that stores computed values.

7.25 sklearn.mixture: Gaussian Mixture Models

The *sklearn.mixture* module implements mixture modeling algorithms.

User guide: See the *Gaussian mixture models* section for further details.

mixture.BayesianGaussianMixture([...])	Variational Bayesian estimation of a Gaussian mixture.
mixture.GaussianMixture([n_components, ...])	Gaussian Mixture.

7.25.1 sklearn.mixture.BayesianGaussianMixture

class sklearn.mixture.**BayesianGaussianMixture**(*n_components=1*, *covariance_type='full'*, *tol=0.001*, *reg_covar=1e-06*, *max_iter=100*, *n_init=1*, *init_params='kmeans'*, *weight_concentration_prior_type='dirichlet_process'*, *weight_concentration_prior=None*, *mean_precision_prior=None*, *mean_prior=None*, *degrees_of_freedom_prior=None*, *covariance_prior=None*, *random_state=None*, *warm_start=False*, *verbose=0*, *verbose_interval=10*)

Variational Bayesian estimation of a Gaussian mixture.

This class allows to infer an approximate posterior distribution over the parameters of a Gaussian mixture distribution. The effective number of components can be inferred from the data.

This class implements two types of prior for the weights distribution: a finite mixture model with Dirichlet distribution and an infinite mixture model with the Dirichlet Process. In practice Dirichlet Process inference

algorithm is approximated and uses a truncated distribution with a fixed maximum number of components (called the Stick-breaking representation). The number of components actually used almost always depends on the data.

New in version 0.18.

Read more in the *User Guide*.

Parameters

n_components [int, defaults to 1.] The number of mixture components. Depending on the data and the value of the `weight_concentration_prior` the model can decide to not use all the components by setting some component `weights_` to values very close to zero. The number of effective components is therefore smaller than n_components.

covariance_type [{'full', 'tied', 'diag', 'spherical'}, defaults to 'full'] String describing the type of covariance parameters to use. Must be one of:

```
'full' (each component has its own general covariance matrix),
'tied' (all components share the same general covariance matrix),
'diag' (each component has its own diagonal covariance matrix),
'spherical' (each component has its own single variance).
```

tol [float, defaults to 1e-3.] The convergence threshold. EM iterations will stop when the lower bound average gain on the likelihood (of the training data with respect to the model) is below this threshold.

reg_covar [float, defaults to 1e-6.] Non-negative regularization added to the diagonal of covariance. Allows to assure that the covariance matrices are all positive.

max_iter [int, defaults to 100.] The number of EM iterations to perform.

n_init [int, defaults to 1.] The number of initializations to perform. The result with the highest lower bound value on the likelihood is kept.

init_params [{'kmeans', 'random'}, defaults to 'kmeans'.] The method used to initialize the weights, the means and the covariances. Must be one of:

```
'kmeans' : responsibilities are initialized using kmeans.
'random' : responsibilities are initialized randomly.
```

weight_concentration_prior_type [str, defaults to 'dirichlet_process'.] String describing the type of the weight concentration prior. Must be one of:

```
'dirichlet_process' (using the Stick-breaking representation),
'dirichlet_distribution' (can favor more uniform weights).
```

weight_concentration_prior [float | None, optional.] The dirichlet concentration of each component on the weight distribution (Dirichlet). This is commonly called gamma in the literature. The higher concentration puts more mass in the center and will lead to more components being active, while a lower concentration parameter will lead to more mass at the edge of the mixture weights simplex. The value of the parameter must be greater than 0. If it is None, it's set to `1. / n_components`.

mean_precision_prior [float | None, optional.] The precision prior on the mean distribution (Gaussian). Controls the extent of where means can be placed. Larger values concentrate the cluster means around `mean_prior`. The value of the parameter must be greater than 0. If it is None, it is set to 1.

mean_prior [array-like, shape (n_features,), optional] The prior on the mean distribution (Gaussian). If it is None, it is set to the mean of X.

degrees_of_freedom_prior [float | None, optional.] The prior of the number of degrees of freedom on the covariance distributions (Wishart). If it is None, it's set to `n_features`.

covariance_prior [float or array-like, optional] The prior on the covariance distribution (Wishart). If it is None, the emiprical covariance prior is initialized using the covariance of X. The shape depends on `covariance_type`:

```
(n_features, n_features) if 'full',
(n_features, n_features) if 'tied',
(n_features)             if 'diag',
float                    if 'spherical'
```

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

warm_start [bool, default to False.] If 'warm_start' is True, the solution of the last fitting is used as initialization for the next call of fit(). This can speed up convergence when fit is called several times on similar problems. See *the Glossary*.

verbose [int, default to 0.] Enable verbose output. If 1 then it prints the current initialization and each iteration step. If greater than 1 then it prints also the log probability and the time needed for each step.

verbose_interval [int, default to 10.] Number of iteration done before the next print.

Attributes

weights_ [array-like, shape (n_components,)] The weights of each mixture components.

means_ [array-like, shape (n_components, n_features)] The mean of each mixture component.

covariances_ [array-like] The covariance of each mixture component. The shape depends on `covariance_type`:

```
(n_components,)                        if 'spherical',
(n_features, n_features)               if 'tied',
(n_components, n_features)             if 'diag',
(n_components, n_features, n_features) if 'full'
```

precisions_ [array-like] The precision matrices for each component in the mixture. A precision matrix is the inverse of a covariance matrix. A covariance matrix is symmetric positive definite so the mixture of Gaussian can be equivalently parameterized by the precision matrices. Storing the precision matrices instead of the covariance matrices makes it more efficient to compute the log-likelihood of new samples at test time. The shape depends on `covariance_type`:

```
(n_components,)                        if 'spherical',
(n_features, n_features)               if 'tied',
(n_components, n_features)             if 'diag',
(n_components, n_features, n_features) if 'full'
```

precisions_cholesky_ [array-like] The cholesky decomposition of the precision matrices of each mixture component. A precision matrix is the inverse of a covariance matrix. A covariance matrix is symmetric positive definite so the mixture of Gaussian can be equivalently parameterized by the precision matrices. Storing the precision matrices instead of the covariance matrices makes it more efficient to compute the log-likelihood of new samples at test time. The shape depends on `covariance_type`:

```
(n_components,)                        if 'spherical',
(n_features, n_features)               if 'tied',
(n_components, n_features)             if 'diag',
(n_components, n_features, n_features) if 'full'
```

converged_ [bool] True when convergence was reached in fit(), False otherwise.

n_iter_ [int] Number of step used by the best fit of inference to reach the convergence.

lower_bound_ [float] Lower bound value on the likelihood (of the training data with respect to the model) of the best fit of inference.

weight_concentration_prior_ [tuple or float] The dirichlet concentration of each component on the weight distribution (Dirichlet). The type depends on `weight_concentration_prior_type`:

```
(float, float) if 'dirichlet_process' (Beta parameters),
float          if 'dirichlet_distribution' (Dirichlet parameters).
```

The higher concentration puts more mass in the center and will lead to more components being active, while a lower concentration parameter will lead to more mass at the edge of the simplex.

weight_concentration_ [array-like, shape (n_components,)] The dirichlet concentration of each component on the weight distribution (Dirichlet).

mean_precision_prior_ [float] The precision prior on the mean distribution (Gaussian). Controls the extent of where means can be placed. Larger values concentrate the cluster means around `mean_prior`. If mean_precision_prior is set to None, `mean_precision_prior_` is set to 1.

mean_precision_ [array-like, shape (n_components,)] The precision of each components on the mean distribution (Gaussian).

mean_prior_ [array-like, shape (n_features,)] The prior on the mean distribution (Gaussian).

degrees_of_freedom_prior_ [float] The prior of the number of degrees of freedom on the covariance distributions (Wishart).

degrees_of_freedom_ [array-like, shape (n_components,)] The number of degrees of freedom of each components in the model.

covariance_prior_ [float or array-like] The prior on the covariance distribution (Wishart). The shape depends on `covariance_type`:

```
(n_features, n_features) if 'full',
(n_features, n_features) if 'tied',
(n_features)             if 'diag',
float                    if 'spherical'
```

See also:

GaussianMixture Finite Gaussian mixture fit with EM.

References

[R16529824bff2-1], [R16529824bff2-2], [R16529824bff2-3]

Methods

fit(self, X[, y])	Estimate model parameters with the EM algorithm.
fit_predict(self, X[, y])	Estimate model parameters using X and predict the labels for X.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the labels for the data samples in X using trained model.
predict_proba(self, X)	Predict posterior probability of each component given the data.
sample(self[, n_samples])	Generate random samples from the fitted Gaussian distribution.
score(self, X[, y])	Compute the per-sample average log-likelihood of the given data X.
score_samples(self, X)	Compute the weighted log probabilities for each sample.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_components=1*, *covariance_type='full'*, *tol=0.001*, *reg_covar=1e-06*, *max_iter=100*, *n_init=1*, *init_params='kmeans'*, *weight_concentration_prior_type='dirichlet_process'*, *weight_concentration_prior=None*, *mean_precision_prior=None*, *mean_prior=None*, *degrees_of_freedom_prior=None*, *covariance_prior=None*, *random_state=None*, *warm_start=False*, *verbose=0*, *verbose_interval=10*)

Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)

Estimate model parameters with the EM algorithm.

The method fits the model `n_init` times and sets the parameters with which the model has the largest likelihood or lower bound. Within each trial, the method iterates between E-step and M-step for `max_iter` times until the change of likelihood or lower bound is less than `tol`, otherwise, a `ConvergenceWarning` is raised. If `warm_start` is `True`, then `n_init` is ignored and a single initialization is performed upon the first call. Upon consecutive calls, training starts where it left off.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

self

fit_predict(*self*, *X*, *y=None*)

Estimate model parameters using X and predict the labels for X.

The method fits the model n_init times and sets the parameters with which the model has the largest likelihood or lower bound. Within each trial, the method iterates between E-step and M-step for `max_iter` times until the change of likelihood or lower bound is less than `tol`, otherwise, a *ConvergenceWarning* is raised. After fitting, it predicts the most probable label for the input data points.

New in version 0.20.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

labels [array, shape (n_samples,)] Component labels.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict the labels for the data samples in X using trained model.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

labels [array, shape (n_samples,)] Component labels.

predict_proba (*self*, *X*)
Predict posterior probability of each component given the data.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

resp [array, shape (n_samples, n_components)] Returns the probability each Gaussian (state) in the model given each sample.

sample (*self*, *n_samples=1*)
Generate random samples from the fitted Gaussian distribution.

Parameters

n_samples [int, optional] Number of samples to generate. Defaults to 1.

Returns

X [array, shape (n_samples, n_features)] Randomly generated sample

y [array, shape (nsamples,)] Component labels

score (*self*, *X*, *y=None*)
Compute the per-sample average log-likelihood of the given data X.

Parameters

X [array-like, shape (n_samples, n_dimensions)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

log_likelihood [float] Log likelihood of the Gaussian mixture given X.

score_samples (*self*, *X*)
Compute the weighted log probabilities for each sample.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

log_prob [array, shape (n_samples,)] Log probabilities of each data point in X.

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.mixture.BayesianGaussianMixture`

- *Gaussian Mixture Model Ellipsoids*
- *Gaussian Mixture Model Sine Curve*
- *Concentration Prior Type Analysis of Variation Bayesian Gaussian Mixture*

7.25.2 `sklearn.mixture`.GaussianMixture

class sklearn.mixture.**GaussianMixture**(*n_components=1*, *covariance_type='full'*, *tol=0.001*, *reg_covar=1e-06*, *max_iter=100*, *n_init=1*, *init_params='kmeans'*, *weights_init=None*, *means_init=None*, *precisions_init=None*, *random_state=None*, *warm_start=False*, *verbose=0*, *verbose_interval=10*)
Gaussian Mixture.

Representation of a Gaussian mixture model probability distribution. This class allows to estimate the parameters of a Gaussian mixture distribution.

Read more in the *User Guide*.

New in version 0.18.

Parameters

n_components [int, defaults to 1.] The number of mixture components.

covariance_type [{'full' (default), 'tied', 'diag', 'spherical'}] String describing the type of covariance parameters to use. Must be one of:

'full' each component has its own general covariance matrix

'tied' all components share the same general covariance matrix

'diag' each component has its own diagonal covariance matrix

'spherical' each component has its own single variance

tol [float, defaults to 1e-3.] The convergence threshold. EM iterations will stop when the lower bound average gain is below this threshold.

reg_covar [float, defaults to 1e-6.] Non-negative regularization added to the diagonal of covariance. Allows to assure that the covariance matrices are all positive.

max_iter [int, defaults to 100.] The number of EM iterations to perform.

n_init [int, defaults to 1.] The number of initializations to perform. The best results are kept.

init_params [{'kmeans', 'random'}, defaults to 'kmeans'.] The method used to initialize the weights, the means and the precisions. Must be one of:

```
'kmeans' : responsibilities are initialized using kmeans.
'random' : responsibilities are initialized randomly.
```

weights_init [array-like, shape (n_components,), optional] The user-provided initial weights, defaults to None. If it None, weights are initialized using the `init_params` method.

means_init [array-like, shape (n_components, n_features), optional] The user-provided initial means, defaults to None, If it None, means are initialized using the `init_params` method.

precisions_init [array-like, optional.] The user-provided initial precisions (inverse of the covariance matrices), defaults to None. If it None, precisions are initialized using the 'init_params' method. The shape depends on 'covariance_type':

```
(n_components,)                        if 'spherical',
(n_features, n_features)               if 'tied',
(n_components, n_features)             if 'diag',
(n_components, n_features, n_features) if 'full'
```

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

warm_start [bool, default to False.] If 'warm_start' is True, the solution of the last fitting is used as initialization for the next call of fit(). This can speed up convergence when fit is called several times on similar problems. In that case, 'n_init' is ignored and only a single initialization occurs upon the first call. See *the Glossary*.

verbose [int, default to 0.] Enable verbose output. If 1 then it prints the current initialization and each iteration step. If greater than 1 then it prints also the log probability and the time needed for each step.

verbose_interval [int, default to 10.] Number of iteration done before the next print.

Attributes

weights_ [array-like, shape (n_components,)] The weights of each mixture components.

means_ [array-like, shape (n_components, n_features)] The mean of each mixture component.

covariances_ [array-like] The covariance of each mixture component. The shape depends on `covariance_type`:

```
(n_components,)                        if 'spherical',
(n_features, n_features)               if 'tied',
(n_components, n_features)             if 'diag',
(n_components, n_features, n_features) if 'full'
```

precisions_ [array-like] The precision matrices for each component in the mixture. A precision matrix is the inverse of a covariance matrix. A covariance matrix is symmetric positive definite so the mixture of Gaussian can be equivalently parameterized by the precision matrices. Storing the precision matrices instead of the covariance matrices makes it more efficient to compute the log-likelihood of new samples at test time. The shape depends on `covariance_type`:

```
(n_components,)                        if 'spherical',
(n_features, n_features)               if 'tied',
(n_components, n_features)             if 'diag',
(n_components, n_features, n_features) if 'full'
```

precisions_cholesky_ [array-like] The cholesky decomposition of the precision matrices of each mixture component. A precision matrix is the inverse of a covariance matrix. A covariance matrix is symmetric positive definite so the mixture of Gaussian can be equivalently parameterized by the precision matrices. Storing the precision matrices instead of the covariance matrices makes it more efficient to compute the log-likelihood of new samples at test time. The shape depends on `covariance_type`:

```
(n_components,)                        if 'spherical',
(n_features, n_features)               if 'tied',
(n_components, n_features)             if 'diag',
(n_components, n_features, n_features) if 'full'
```

converged_ [bool] True when convergence was reached in fit(), False otherwise.

n_iter_ [int] Number of step used by the best fit of EM to reach the convergence.

lower_bound_ [float] Lower bound value on the log-likelihood (of the training data with respect to the model) of the best fit of EM.

See also:

BayesianGaussianMixture Gaussian mixture model fit with a variational inference.

Methods

aic(self, X)	Akaike information criterion for the current model on the input X.
bic(self, X)	Bayesian information criterion for the current model on the input X.
fit(self, X[, y])	Estimate model parameters with the EM algorithm.
fit_predict(self, X[, y])	Estimate model parameters using X and predict the labels for X.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the labels for the data samples in X using trained model.
predict_proba(self, X)	Predict posterior probability of each component given the data.
sample(self[, n_samples])	Generate random samples from the fitted Gaussian distribution.
score(self, X[, y])	Compute the per-sample average log-likelihood of the given data X.

Continued on next page

Table 202 – continued from previous page

score_samples(self, X)	Compute the weighted log probabilities for each sample.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_components=1*, *covariance_type='full'*, *tol=0.001*, *reg_covar=1e-06*, *max_iter=100*, *n_init=1*, *init_params='kmeans'*, *weights_init=None*, *means_init=None*, *precisions_init=None*, *random_state=None*, *warm_start=False*, *verbose=0*, *verbose_interval=10*)

Initialize self. See help(type(self)) for accurate signature.

aic(*self*, *X*)

Akaike information criterion for the current model on the input X.

Parameters

X [array of shape (n_samples, n_dimensions)]

Returns

aic [float] The lower the better.

bic(*self*, *X*)

Bayesian information criterion for the current model on the input X.

Parameters

X [array of shape (n_samples, n_dimensions)]

Returns

bic [float] The lower the better.

fit(*self*, *X*, *y=None*)

Estimate model parameters with the EM algorithm.

The method fits the model `n_init` times and sets the parameters with which the model has the largest likelihood or lower bound. Within each trial, the method iterates between E-step and M-step for `max_iter` times until the change of likelihood or lower bound is less than `tol`, otherwise, a `ConvergenceWarning` is raised. If `warm_start` is `True`, then `n_init` is ignored and a single initialization is performed upon the first call. Upon consecutive calls, training starts where it left off.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

self

fit_predict(*self*, *X*, *y=None*)

Estimate model parameters using X and predict the labels for X.

The method fits the model n_init times and sets the parameters with which the model has the largest likelihood or lower bound. Within each trial, the method iterates between E-step and M-step for `max_iter` times until the change of likelihood or lower bound is less than `tol`, otherwise, a *ConvergenceWarning* is raised. After fitting, it predicts the most probable label for the input data points.

New in version 0.20.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

labels [array, shape (n_samples,)] Component labels.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict the labels for the data samples in X using trained model.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

labels [array, shape (n_samples,)] Component labels.

predict_proba(*self*, *X*)
Predict posterior probability of each component given the data.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

resp [array, shape (n_samples, n_components)] Returns the probability each Gaussian (state) in the model given each sample.

sample(*self*, *n_samples=1*)
Generate random samples from the fitted Gaussian distribution.

Parameters

n_samples [int, optional] Number of samples to generate. Defaults to 1.

Returns

X [array, shape (n_samples, n_features)] Randomly generated sample

y [array, shape (nsamples,)] Component labels

score(*self*, *X*, *y=None*)
Compute the per-sample average log-likelihood of the given data X.

Parameters

X [array-like, shape (n_samples, n_dimensions)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

log_likelihood [float] Log likelihood of the Gaussian mixture given X.

score_samples(*self*, *X*)
Compute the weighted log probabilities for each sample.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

log_prob [array, shape (n_samples,)] Log probabilities of each data point in X.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.mixture.GaussianMixture`

- *Comparing different clustering algorithms on toy datasets*
- *Density Estimation for a Gaussian mixture*
- *Gaussian Mixture Model Ellipsoids*
- *Gaussian Mixture Model Selection*
- *GMM covariances*
- *Gaussian Mixture Model Sine Curve*

7.26 `sklearn.model_selection`: Model Selection

User guide: See the *Cross-validation: evaluating estimator performance*, *Tuning the hyper-parameters of an estimator* and *Learning curve* sections for further details.

7.26.1 Splitter Classes

model_selection.GroupKFold([n_splits])	K-fold iterator variant with non-overlapping groups.
model_selection.GroupShuffleSplit([...])	Shuffle-Group(s)-Out cross-validation iterator
model_selection.KFold([n_splits, shuffle, ...])	K-Folds cross-validator
model_selection.LeaveOneGroupOut	Leave One Group Out cross-validator
model_selection.LeavePGroupsOut(n_groups)	Leave P Group(s) Out cross-validator
model_selection.LeaveOneOut	Leave-One-Out cross-validator

Continued on next page

Table 203 – continued from previous page

model_selection.LeavePOut(p)	Leave-P-Out cross-validator
model_selection.PredefinedSplit(test_fold)	Predefined split cross-validator
model_selection.RepeatedKFold([n_splits, ...])	Repeated K-Fold cross validator.
model_selection.RepeatedStratifiedKFold([...])	Repeated Stratified K-Fold cross validator.
model_selection.ShuffleSplit([n_splits, ...])	Random permutation cross-validator
model_selection.StratifiedKFold([n_splits, ...])	Stratified K-Folds cross-validator
model_selection.StratifiedShuffleSplit([...])	Stratified ShuffleSplit cross-validator
model_selection.TimeSeriesSplit([n_splits, ...])	Time Series cross-validator

sklearn.model_selection.GroupKFold

class sklearn.model_selection.**GroupKFold**(*n_splits=5*)

K-fold iterator variant with non-overlapping groups.

The same group will not appear in two different folds (the number of distinct groups has to be at least equal to the number of folds).

The folds are approximately balanced in the sense that the number of distinct groups is approximately the same in each fold.

Parameters

n_splits [int, default=5] Number of folds. Must be at least 2.

Changed in version 0.22: `n_splits` default value changed from 3 to 5.

See also:

LeaveOneGroupOut For splitting the data according to explicit domain-specific stratification of the dataset.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import GroupKFold
>>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
>>> y = np.array([1, 2, 3, 4])
>>> groups = np.array([0, 0, 2, 2])
>>> group_kfold = GroupKFold(n_splits=2)
>>> group_kfold.get_n_splits(X, y, groups)
2
>>> print(group_kfold)
GroupKFold(n_splits=2)
>>> for train_index, test_index in group_kfold.split(X, y, groups):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...     print(X_train, X_test, y_train, y_test)
...
```

(continues on next page)

(continued from previous page)

```
TRAIN: [0 1] TEST: [2 3]
[[1 2]
 [3 4]] [[5 6]
 [7 8]] [1 2] [3 4]
TRAIN: [2 3] TEST: [0 1]
[[5 6]
 [7 8]] [[1 2]
 [3 4]] [3 4] [1 2]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_splits=5*)
Initialize self. See help(type(self)) for accurate signature.

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape (n_samples,), optional] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,)] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Examples using `sklearn.model_selection.GroupKFold`

- *Visualizing cross-validation behavior in scikit-learn*

sklearn.model_selection.GroupShuffleSplit

class sklearn.model_selection.**GroupShuffleSplit**(*n_splits=5*, *test_size=None*, *train_size=None*, *random_state=None*)

Shuffle-Group(s)-Out cross-validation iterator

Provides randomized train/test indices to split data according to a third-party provided group. This group information can be used to encode arbitrary domain specific stratifications of the samples as integers.

For instance the groups could be the year of collection of the samples and thus allow for cross-validation against time-based splits.

The difference between LeavePGroupsOut and GroupShuffleSplit is that the former generates splits using all subsets of size `p` unique groups, whereas GroupShuffleSplit generates a user-determined number of random test splits, each with a user-determined fraction of unique groups.

For example, a less computationally intensive alternative to `LeavePGroupsOut(p=10)` would be `GroupShuffleSplit(test_size=10, n_splits=100)`.

Note: The parameters `test_size` and `train_size` refer to groups, and not to samples, as in ShuffleSplit.

Parameters

n_splits [int (default 5)] Number of re-shuffling & splitting iterations.

test_size [float, int, None, optional (default=None)] If float, should be between 0.0 and 1.0 and represent the proportion of groups to include in the test split (rounded up). If int, represents the absolute number of test groups. If None, the value is set to the complement of the train size. By default, the value is set to 0.2. The default will change in version 0.21. It will remain 0.2 only if `train_size` is unspecified, otherwise it will complement the specified `train_size`.

train_size [float, int, or None, default is None] If float, should be between 0.0 and 1.0 and represent the proportion of the groups to include in the train split. If int, represents the absolute number of train groups. If None, the value is automatically set to the complement of the test size.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import GroupShuffleSplit
>>> X = np.ones(shape=(8, 2))
>>> y = np.ones(shape=(8, 1))
>>> groups = np.array([1, 1, 2, 2, 2, 3, 3, 3])
>>> print(groups.shape)
(8,)
>>> gss = GroupShuffleSplit(n_splits=2, train_size=.7, random_state=42)
>>> gss.get_n_splits()
2
>>> for train_idx, test_idx in gss.split(X, y, groups):
...     print("TRAIN:", train_idx, "TEST:", test_idx)
TRAIN: [2 3 4 5 6 7] TEST: [0 1]
TRAIN: [0 1 5 6 7] TEST: [2 3 4]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_splits=5*, *test_size=None*, *train_size=None*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape (n_samples,), optional] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,)] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Examples using `sklearn.model_selection.GroupShuffleSplit`

- *Visualizing cross-validation behavior in scikit-learn*

sklearn.model_selection.KFold

class sklearn.model_selection.**KFold**(*n_splits=5*, *shuffle=False*, *random_state=None*)

K-Folds cross-validator

Provides train/test indices to split data in train/test sets. Split dataset into k consecutive folds (without shuffling by default).

Each fold is then used once as a validation while the k - 1 remaining folds form the training set.

Read more in the *User Guide*.

Parameters

n_splits [int, default=5] Number of folds. Must be at least 2.

Changed in version 0.22: `n_splits` default value changed from 3 to 5.

shuffle [boolean, optional] Whether to shuffle the data before splitting into batches.

random_state [int, RandomState instance or None, optional, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Only used when `shuffle` is True. This should be left to None if `shuffle` is False.

See also:

StratifiedKFold Takes group information into account to avoid building folds with imbalanced class distributions (for binary or multiclass classification tasks).

GroupKFold K-fold iterator variant with non-overlapping groups.

RepeatedKFold Repeats K-Fold n times.

Notes

The first `n_samples % n_splits` folds have size `n_samples // n_splits + 1`, other folds have size `n_samples // n_splits`, where `n_samples` is the number of samples.

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import KFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([1, 2, 3, 4])
>>> kf = KFold(n_splits=2)
>>> kf.get_n_splits(X)
2
>>> print(kf)
KFold(n_splits=2, random_state=None, shuffle=False)
>>> for train_index, test_index in kf.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
```

(continues on next page)

(continued from previous page)

```
TRAIN: [2 3] TEST: [0 1]
TRAIN: [0 1] TEST: [2 3]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_splits=5*, *shuffle=False*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape (n_samples,)] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Examples using `sklearn.model_selection.KFold`

- *Feature agglomeration vs. univariate selection*
- *Gradient Boosting Out-of-Bag estimates*
- *Nested versus non-nested cross-validation*
- *Visualizing cross-validation behavior in scikit-learn*
- *Cross-validation on diabetes Dataset Exercise*

sklearn.model_selection.LeaveOneGroupOut

class sklearn.model_selection.**LeaveOneGroupOut**

Leave One Group Out cross-validator

Provides train/test indices to split data according to a third-party provided group. This group information can be used to encode arbitrary domain specific stratifications of the samples as integers.

For instance the groups could be the year of collection of the samples and thus allow for cross-validation against time-based splits.

Read more in the *User Guide*.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import LeaveOneGroupOut
>>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
>>> y = np.array([1, 2, 1, 2])
>>> groups = np.array([1, 1, 2, 2])
>>> logo = LeaveOneGroupOut()
>>> logo.get_n_splits(X, y, groups)
2
>>> logo.get_n_splits(groups=groups)  # 'groups' is always required
2
>>> print(logo)
LeaveOneGroupOut()
>>> for train_index, test_index in logo.split(X, y, groups):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...     print(X_train, X_test, y_train, y_test)
TRAIN: [2 3] TEST: [0 1]
[[5 6]
 [7 8]] [[1 2]
 [3 4]] [1 2] [1 2]
TRAIN: [0 1] TEST: [2 3]
[[1 2]
 [3 4]] [[5 6]
 [7 8]] [1 2] [1 2]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, */*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)

Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [array-like, with shape (n_samples,)] Group labels for the samples used while splitting the dataset into train/test set. This 'groups' parameter must always be specified to calculate the number of splits, though the other parameters can be omitted.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, of length n_samples, optional] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,)] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

sklearn.model_selection.LeavePGroupsOut

class sklearn.model_selection.**LeavePGroupsOut**(*n_groups*)
Leave P Group(s) Out cross-validator

Provides train/test indices to split data according to a third-party provided group. This group information can be used to encode arbitrary domain specific stratifications of the samples as integers.

For instance the groups could be the year of collection of the samples and thus allow for cross-validation against time-based splits.

The difference between LeavePGroupsOut and LeaveOneGroupOut is that the former builds the test sets with all the samples assigned to `p` different values of the groups while the latter uses samples all assigned the same groups.

Read more in the *User Guide*.

Parameters

n_groups [int] Number of groups (`p`) to leave out in the test split.

See also:

GroupKFold K-fold iterator variant with non-overlapping groups.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import LeavePGroupsOut
>>> X = np.array([[1, 2], [3, 4], [5, 6]])
>>> y = np.array([1, 2, 1])
>>> groups = np.array([1, 2, 3])
>>> lpgo = LeavePGroupsOut(n_groups=2)
>>> lpgo.get_n_splits(X, y, groups)
3
>>> lpgo.get_n_splits(groups=groups)  # 'groups' is always required
3
>>> print(lpgo)
LeavePGroupsOut(n_groups=2)
>>> for train_index, test_index in lpgo.split(X, y, groups):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...     print(X_train, X_test, y_train, y_test)
TRAIN: [2] TEST: [0 1]
[[5 6]] [[1 2]
 [3 4]] [1] [1 2]
TRAIN: [1] TEST: [0 2]
[[3 4]] [[1 2]
 [5 6]] [2] [1 1]
TRAIN: [0] TEST: [1 2]
[[1 2]] [[3 4]
 [5 6]] [1] [2 1]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_groups*)
Initialize self. See help(type(self)) for accurate signature.

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [array-like, with shape (n_samples,)] Group labels for the samples used while splitting the dataset into train/test set. This 'groups' parameter must always be specified to calculate the number of splits, though the other parameters can be omitted.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, of length n_samples, optional] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,)] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

sklearn.model_selection.LeaveOneOut

class sklearn.model_selection.**LeaveOneOut**

Leave-One-Out cross-validator

Provides train/test indices to split data in train/test sets. Each sample is used once as a test set (singleton) while the remaining samples form the training set.

Note: `LeaveOneOut()` is equivalent to `KFold(n_splits=n)` and `LeavePOut(p=1)` where `n` is the number of samples.

Due to the high number of test sets (which is the same as the number of samples) this cross-validation method can be very costly. For large datasets one should favor *KFold*, *ShuffleSplit* or *StratifiedKFold*.

Read more in the *User Guide*.

See also:

LeaveOneGroupOut For splitting the data according to explicit, domain-specific stratification of the dataset.

GroupKFold K-fold iterator variant with non-overlapping groups.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import LeaveOneOut
>>> X = np.array([[1, 2], [3, 4]])
>>> y = np.array([1, 2])
>>> loo = LeaveOneOut()
>>> loo.get_n_splits(X)
2
>>> print(loo)
LeaveOneOut()
>>> for train_index, test_index in loo.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...     print(X_train, X_test, y_train, y_test)
TRAIN: [1] TEST: [0]
[[3 4]] [[1 2]] [2] [1]
TRAIN: [0] TEST: [1]
[[1 2]] [[3 4]] [1] [2]
```

Methods

get_n_splits(self, X[, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

get_n_splits (*self*, *X*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, of length n_samples] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

sklearn.model_selection.LeavePOut

class sklearn.model_selection.**LeavePOut** (*p*)
Leave-P-Out cross-validator

Provides train/test indices to split data in train/test sets. This results in testing on all distinct samples of size p, while the remaining n - p samples form the training set in each iteration.

Note: `LeavePOut(p)` is NOT equivalent to `KFold(n_splits=n_samples // p)` which creates non-overlapping test sets.

Due to the high number of iterations which grows combinatorically with the number of samples this cross-validation method can be very costly. For large datasets one should favor *KFold*, *StratifiedKFold* or *ShuffleSplit*.

Read more in the *User Guide*.

Parameters

p [int] Size of the test sets. Must be strictly less than the number of samples.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import LeavePOut
>>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
>>> y = np.array([1, 2, 3, 4])
>>> lpo = LeavePOut(2)
>>> lpo.get_n_splits(X)
6
>>> print(lpo)
LeavePOut(p=2)
>>> for train_index, test_index in lpo.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
TRAIN: [2 3] TEST: [0 1]
TRAIN: [1 3] TEST: [0 2]
TRAIN: [1 2] TEST: [0 3]
TRAIN: [0 3] TEST: [1 2]
TRAIN: [0 2] TEST: [1 3]
TRAIN: [0 1] TEST: [2 3]
```

Methods

get_n_splits(self, X[, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *p*)
Initialize self. See help(type(self)) for accurate signature.

get_n_splits (*self*, *X*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, of length n_samples] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

sklearn.model_selection.PredefinedSplit

class sklearn.model_selection.**PredefinedSplit**(*test_fold*)

Predefined split cross-validator

Provides train/test indices to split data into train/test sets using a predefined scheme specified by the user with the `test_fold` parameter.

Read more in the *User Guide*.

New in version 0.16.

Parameters

test_fold [array-like, shape (n_samples,)] The entry `test_fold[i]` represents the index of the test set that sample `i` belongs to. It is possible to exclude sample `i` from any test set (i.e. include sample `i` in every training set) by setting `test_fold[i]` equal to -1.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import PredefinedSplit
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 1, 1])
>>> test_fold = [0, 1, -1, 1]
>>> ps = PredefinedSplit(test_fold)
>>> ps.get_n_splits()
2
>>> print(ps)
PredefinedSplit(test_fold=array([ 0,  1, -1,  1]))
>>> for train_index, test_index in ps.split():
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
TRAIN: [1 2 3] TEST: [0]
TRAIN: [0 2] TEST: [1 3]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self[, X, y, groups])	Generate indices to split data into training and test set.

__init__(*self*, *test_fold*)

Initialize self. See help(type(self)) for accurate signature.

get_n_splits(*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split(*self*, *X=None*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

sklearn.model_selection.RepeatedKFold

class sklearn.model_selection.**RepeatedKFold**(*n_splits=5*, *n_repeats=10*, *random_state=None*)
Repeated K-Fold cross validator.

Repeats K-Fold n times with different randomization in each repetition.

Read more in the *User Guide*.

Parameters

n_splits [int, default=5] Number of folds. Must be at least 2.

n_repeats [int, default=10] Number of times cross-validator needs to be repeated.

random_state [int, RandomState instance or None, optional, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

See also:

RepeatedStratifiedKFold Repeats Stratified K-Fold n times.

Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import RepeatedKFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 1, 1])
>>> rkf = RepeatedKFold(n_splits=2, n_repeats=2, random_state=2652124)
>>> for train_index, test_index in rkf.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...
TRAIN: [0 1] TEST: [2 3]
TRAIN: [2 3] TEST: [0 1]
TRAIN: [1 2] TEST: [0 3]
TRAIN: [0 3] TEST: [1 2]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generates indices to split data into training and test set.

__init__ (*self*, *n_splits=5*, *n_repeats=10*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility. `np.zeros(n_samples)` may be used as a placeholder.

y [object] Always ignored, exists for compatibility. `np.zeros(n_samples)` may be used as a placeholder.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generates indices to split data into training and test set.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, of length n_samples] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

sklearn.model_selection.RepeatedStratifiedKFold

class sklearn.model_selection.**RepeatedStratifiedKFold**(*n_splits=5*, *n_repeats=10*, *random_state=None*)

Repeated Stratified K-Fold cross validator.

Repeats Stratified K-Fold n times with different randomization in each repetition.

Read more in the *User Guide*.

Parameters

n_splits [int, default=5] Number of folds. Must be at least 2.

n_repeats [int, default=10] Number of times cross-validator needs to be repeated.

random_state [None, int or RandomState, default=None] Random state to be used to generate random state for each repetition.

See also:

RepeatedKFold Repeats K-Fold n times.

Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import RepeatedStratifiedKFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 1, 1])
>>> rskf = RepeatedStratifiedKFold(n_splits=2, n_repeats=2,
...     random_state=36851234)
>>> for train_index, test_index in rskf.split(X, y):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...
TRAIN: [1 2] TEST: [0 3]
TRAIN: [0 3] TEST: [1 2]
TRAIN: [1 3] TEST: [0 2]
TRAIN: [0 2] TEST: [1 3]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator

Continued on next page

Table 213 – continued from previous page

split(self, X[, y, groups])	Generates indices to split data into training and test set.

__init__ (*self*, *n_splits=5*, *n_repeats=10*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility. `np.zeros(n_samples)` may be used as a placeholder.

y [object] Always ignored, exists for compatibility. `np.zeros(n_samples)` may be used as a placeholder.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generates indices to split data into training and test set.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, of length n_samples] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

sklearn.model_selection.ShuffleSplit

class sklearn.model_selection.**ShuffleSplit**(*n_splits=10*, *test_size=None*, *train_size=None*, *random_state=None*)
Random permutation cross-validator

Yields indices to split data into training and test sets.

Note: contrary to other cross-validation strategies, random splits do not guarantee that all folds will be different, although this is still very likely for sizeable datasets.

Read more in the *User Guide*.

Parameters

n_splits [int, default 10] Number of re-shuffling & splitting iterations.

test_size [float, int, None, default=None] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute

number of test samples. If None, the value is set to the complement of the train size. If `train_size` is also None, it will be set to 0.1.

train_size [float, int, or None, default=None] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import ShuffleSplit
>>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [3, 4], [5, 6]])
>>> y = np.array([1, 2, 1, 2, 1, 2])
>>> rs = ShuffleSplit(n_splits=5, test_size=.25, random_state=0)
>>> rs.get_n_splits(X)
5
>>> print(rs)
ShuffleSplit(n_splits=5, random_state=0, test_size=0.25, train_size=None)
>>> for train_index, test_index in rs.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)
TRAIN: [1 3 0 4] TEST: [5 2]
TRAIN: [4 0 2 5] TEST: [1 3]
TRAIN: [1 2 4 0] TEST: [3 5]
TRAIN: [3 4 1 0] TEST: [5 2]
TRAIN: [3 5 1 0] TEST: [2 4]
>>> rs = ShuffleSplit(n_splits=5, train_size=0.5, test_size=.25,
...                   random_state=0)
>>> for train_index, test_index in rs.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)
TRAIN: [1 3 0] TEST: [5 2]
TRAIN: [4 0 2] TEST: [1 3]
TRAIN: [1 2 4] TEST: [3 5]
TRAIN: [3 4 1] TEST: [5 2]
TRAIN: [3 5 1] TEST: [2 4]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_splits=10*, *test_size=None*, *train_size=None*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape (n_samples,)] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Examples using `sklearn.model_selection.ShuffleSplit`

- *Visualizing cross-validation behavior in scikit-learn*
- *Plotting Learning Curves*
- *Scaling the regularization parameter for SVCs*

`sklearn.model_selection`.StratifiedKFold

class sklearn.model_selection.**StratifiedKFold**(*n_splits=5*, *shuffle=False*, *random_state=None*)
Stratified K-Folds cross-validator

Provides train/test indices to split data in train/test sets.

This cross-validation object is a variation of KFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

Read more in the *User Guide*.

Parameters

n_splits [int, default=5] Number of folds. Must be at least 2.

Changed in version 0.22: `n_splits` default value changed from 3 to 5.

shuffle [boolean, optional] Whether to shuffle each class's samples before splitting into batches.

random_state [int, RandomState instance or None, optional, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Only used when `shuffle` is True. This should be left to None if `shuffle` is False.

See also:

RepeatedStratifiedKFold Repeats Stratified K-Fold n times.

Notes

The implementation is designed to:

- Generate test sets such that all contain the same distribution of classes, or as close as possible.
- Be invariant to class label: relabelling `y = ["Happy", "Sad"]` to `y = [1, 0]` should not change the indices generated.
- Preserve order dependencies in the dataset ordering, when `shuffle=False`: all samples from class k in some test set were contiguous in y, or separated in y by samples from classes other than k.
- Generate test sets where the smallest and largest differ by at most one sample.

Changed in version 0.22: The previous implementation did not follow the last constraint.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import StratifiedKFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 1, 1])
>>> skf = StratifiedKFold(n_splits=2)
>>> skf.get_n_splits(X, y)
2
>>> print(skf)
StratifiedKFold(n_splits=2, random_state=None, shuffle=False)
>>> for train_index, test_index in skf.split(X, y):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
TRAIN: [1 3] TEST: [0 2]
TRAIN: [0 2] TEST: [1 3]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X, y[, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_splits=5*, *shuffle=False*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)

Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split(*self*, *X*, *y*, *groups=None*)

Generate indices to split data into training and test set.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

Note that providing `y` is sufficient to generate the splits and hence `np.zeros(n_samples)` may be used as a placeholder for `X` instead of actual training data.

y [array-like, shape (n_samples,)] The target variable for supervised learning problems. Stratification is done based on the y labels.

groups [object] Always ignored, exists for compatibility.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Examples using `sklearn.model_selection.StratifiedKFold`

- *Recursive feature elimination with cross-validation*
- *Test with permutations the significance of a classification score*
- *GMM covariances*
- *Receiver Operating Characteristic (ROC) with cross validation*
- *Visualizing cross-validation behavior in scikit-learn*

`sklearn.model_selection.StratifiedShuffleSplit`

class sklearn.model_selection.**StratifiedShuffleSplit**(*n_splits=10*, *test_size=None*, *train_size=None*, *random_state=None*)

Stratified ShuffleSplit cross-validator

Provides train/test indices to split data in train/test sets.

This cross-validation object is a merge of StratifiedKFold and ShuffleSplit, which returns stratified randomized folds. The folds are made by preserving the percentage of samples for each class.

Note: like the ShuffleSplit strategy, stratified random splits do not guarantee that all folds will be different, although this is still very likely for sizeable datasets.

Read more in the *User Guide*.

Parameters

n_splits [int, default 10] Number of re-shuffling & splitting iterations.

test_size [float, int, None, optional (default=None)] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. If `train_size` is also None, it will be set to 0.1.

train_size [float, int, or None, default is None] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import StratifiedShuffleSplit
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 0, 1, 1, 1])
>>> sss = StratifiedShuffleSplit(n_splits=5, test_size=0.5, random_state=0)
>>> sss.get_n_splits(X, y)
5
>>> print(sss)
StratifiedShuffleSplit(n_splits=5, random_state=0, ...)
>>> for train_index, test_index in sss.split(X, y):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
TRAIN: [5 2 3] TEST: [4 1 0]
TRAIN: [5 1 4] TEST: [0 2 3]
TRAIN: [5 0 2] TEST: [4 3 1]
TRAIN: [4 1 0] TEST: [2 3 5]
TRAIN: [0 5 1] TEST: [3 4 2]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X, y[, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_splits=10*, *test_size=None*, *train_size=None*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

Note that providing `y` is sufficient to generate the splits and hence `np.zeros(n_samples)` may be used as a placeholder for `X` instead of actual training data.

y [array-like, shape (n_samples,) or (n_samples, n_labels)] The target variable for supervised learning problems. Stratification is done based on the y labels.

groups [object] Always ignored, exists for compatibility.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Examples using `sklearn.model_selection.StratifiedShuffleSplit`

- *Visualizing cross-validation behavior in scikit-learn*
- *RBF SVM parameters*

sklearn.model_selection.TimeSeriesSplit

class sklearn.model_selection.**TimeSeriesSplit** (*n_splits=5*, *max_train_size=None*)
Time Series cross-validator

Provides train/test indices to split time series data samples that are observed at fixed time intervals, in train/test sets. In each split, test indices must be higher than before, and thus shuffling in cross validator is inappropriate.

This cross-validation object is a variation of *KFold*. In the kth split, it returns first k folds as train set and the (k+1)th fold as test set.

Note that unlike standard cross-validation methods, successive training sets are supersets of those that come before them.

Read more in the *User Guide*.

Parameters

n_splits [int, default=5] Number of splits. Must be at least 2.

Changed in version 0.22: `n_splits` default value changed from 3 to 5.

max_train_size [int, optional] Maximum size for a single training set.

Notes

The training set has size `i * n_samples // (n_splits + 1) + n_samples % (n_splits + 1)` in the `i``th split, with a test set of size ``n_samples//(n_splits + 1)`, where `n_samples` is the number of samples.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import TimeSeriesSplit
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> tscv = TimeSeriesSplit()
>>> print(tscv)
TimeSeriesSplit(max_train_size=None, n_splits=5)
>>> for train_index, test_index in tscv.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
TRAIN: [0] TEST: [1]
TRAIN: [0 1] TEST: [2]
TRAIN: [0 1 2] TEST: [3]
TRAIN: [0 1 2 3] TEST: [4]
TRAIN: [0 1 2 3 4] TEST: [5]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_splits=5*, *max_train_size=None*)
Initialize self. See help(type(self)) for accurate signature.

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape (n_samples,)] Always ignored, exists for compatibility.

groups [array-like, with shape (n_samples,)] Always ignored, exists for compatibility.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Examples using `sklearn.model_selection.TimeSeriesSplit`

- *Visualizing cross-validation behavior in scikit-learn*

7.26.2 Splitter Functions

model_selection.check_cv([cv, y, classifier])	Input checker utility for building a cross-validator
model_selection.train_test_split(*arrays, ...)	Split arrays or matrices into random train and test subsets

sklearn.model_selection.check_cv

sklearn.model_selection.**check_cv** (*cv=5*, *y=None*, *classifier=False*)
Input checker utility for building a cross-validator

Parameters

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if classifier is True and `y` is either binary or multiclass, *StratifiedKFold* is used. In all other cases, *KFold* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value changed from 3-fold to 5-fold.

y [array-like, optional] The target variable for supervised learning problems.

classifier [boolean, optional, default False] Whether the task is a classification task, in which case stratified KFold will be used.

Returns

checked_cv [a cross-validator instance.] The return value is a cross-validator which generates the train/test splits via the `split` method.

sklearn.model_selection.train_test_split

sklearn.model_selection.**train_test_split**(**arrays*, ***options*)

Split arrays or matrices into random train and test subsets

Quick utility that wraps input validation and `next(ShuffleSplit().split(X, y))` and application to input data into a single call for splitting (and optionally subsampling) data in a oneliner.

Read more in the *User Guide*.

Parameters

***arrays** [sequence of indexables with same length / shape[0]] Allowed inputs are lists, numpy arrays, scipy-sparse matrices or pandas dataframes.

test_size [float, int or None, optional (default=None)] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. If `train_size` is also None, it will be set to 0.25.

train_size [float, int, or None, (default=None)] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

shuffle [boolean, optional (default=True)] Whether or not to shuffle the data before splitting. If shuffle=False then stratify must be None.

stratify [array-like or None (default=None)] If not None, data is split in a stratified fashion, using this as the class labels.

Returns

splitting [list, length=2 * len(arrays)] List containing train-test split of inputs.

New in version 0.16: If the input is sparse, the output will be a `scipy.sparse.csr_matrix`. Else, output type is the same as the input type.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
>>> X, y = np.arange(10).reshape((5, 2)), range(5)
>>> X
array([[0, 1],
       [2, 3],
```

(continues on next page)

(continued from previous page)

```
       [4, 5],
       [6, 7],
       [8, 9]])
>>> list(y)
[0, 1, 2, 3, 4]
```

```
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.33, random_state=42)
...
>>> X_train
array([[4, 5],
       [0, 1],
       [6, 7]])
>>> y_train
[2, 0, 3]
>>> X_test
array([[2, 3],
       [8, 9]])
>>> y_test
[1, 4]
```

```
>>> train_test_split(y, shuffle=False)
[[0, 1, 2], [3, 4]]
```

Examples using `sklearn.model_selection.train_test_split`

- *ROC Curve with Visualization API*
- *Probability Calibration curves*
- *Probability calibration of classifiers*
- *Recognizing hand-written digits*
- *Classifier comparison*
- *Post pruning decision trees with cost complexity pruning*
- *Understanding the decision tree structure*
- *Comparing random forests and the multi-output meta estimator*
- *Early stopping of Gradient Boosting*
- *Feature transformations with ensembles of trees*
- *Gradient Boosting Out-of-Bag estimates*
- *Faces recognition example using eigenfaces and SVMs*
- *Prediction Latency*
- *Pipeline Anova SVM*
- *Univariate Feature Selection*
- *Comparing various online solvers*
- *MNIST classification using multinomial logistic + L1*
- *Multiclass sparse logistic regression on 20newgroups*

- *Early stopping of Stochastic Gradient Descent*
- *Permutation Importance with Multicollinear or Correlated Features*
- *Permutation Importance vs Random Forest Feature Importance (MDI)*
- *Partial Dependence Plots*
- *Confusion matrix*
- *Parameter estimation using grid search with cross-validation*
- *Receiver Operating Characteristic (ROC)*
- *Precision-Recall*
- *Classifier Chain*
- *Comparing Nearest Neighbors with and without Neighborhood Components Analysis*
- *Dimensionality Reduction with Neighborhood Components Analysis*
- *Restricted Boltzmann Machine features for digit classification*
- *Varying regularization in Multi-layer Perceptron*
- *Column Transformer with Mixed Types*
- *Effect of transforming the targets in regression model*
- *Using FunctionTransformer to select columns*
- *Importance of Feature Scaling*
- *Map data to a normal distribution*
- *Feature discretization*
- *Release Highlights for scikit-learn 0.22*

7.26.3 Hyper-parameter optimizers

model_selection.GridSearchCV(estimator, ...)	Exhaustive search over specified parameter values for an estimator.
model_selection.ParameterGrid(param_grid)	Grid of parameters with a discrete number of values for each.
model_selection.ParameterSampler(..., [, ...])	Generator on parameters sampled from given distributions.
model_selection.RandomizedSearchCV(..., [, ...])	Randomized search on hyper parameters.

sklearn.model_selection.GridSearchCV

class sklearn.model_selection.**GridSearchCV**(*estimator*, *param_grid*, *scoring=None*, *n_jobs=None*, *iid='deprecated'*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *error_score=nan*, *return_train_score=False*)

Exhaustive search over specified parameter values for an estimator.

Important members are fit, predict.

GridSearchCV implements a "fit" and a "score" method. It also implements "predict", "predict_proba", "deci-

sion_function", "transform" and "inverse_transform" if they are implemented in the estimator used.

The parameters of the estimator used to apply these methods are optimized by cross-validated grid-search over a parameter grid.

Read more in the *User Guide*.

Parameters

estimator [estimator object.] This is assumed to implement the scikit-learn estimator interface. Either estimator needs to provide a `score` function, or `scoring` must be passed.

param_grid [dict or list of dictionaries] Dictionary with parameters names (string) as keys and lists of parameter settings to try as values, or a list of such dictionaries, in which case the grids spanned by each dictionary in the list are explored. This enables searching over any sequence of parameter settings.

scoring [string, callable, list/tuple, dict or None, default: None] A single string (see *The scoring parameter: defining model evaluation rules*) or a callable (see *Defining your scoring strategy from metric functions*) to evaluate the predictions on the test set.

For evaluating multiple metrics, either give a list of (unique) strings or a dict with names as keys and callables as values.

NOTE that when using custom scorers, each scorer should return a single value. Metric functions returning a list/array of values can be wrapped into multiple scorers that return one value each.

See *Specifying multiple metrics for evaluation* for an example.

If None, the estimator's score method is used.

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

pre_dispatch [int, or string, optional] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A string, giving an expression as a function of n_jobs, as in '2*n_jobs'

iid [boolean, default=False] If True, return the average score across folds, weighted by the number of samples in each test set. In this case, the data is assumed to be identically distributed across the folds, and the loss minimized is the total loss per sample, and not the mean loss across the folds.

Deprecated since version 0.22: Parameter `iid` is deprecated in 0.22 and will be removed in 0.24

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,

- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, *`StratifiedKFold`* is used. In all other cases, *`KFold`* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

refit [boolean, string, or callable, default=True] Refit an estimator using the best found parameters on the whole dataset.

For multiple metric evaluation, this needs to be a string denoting the scorer that would be used to find the best parameters for refitting the estimator at the end.

Where there are considerations other than maximum score in choosing a best estimator, `refit` can be set to a function which returns the selected `best_index_` given `cv_results_`. In that case, the `best_estimator_` and `best_parameters_` will be set according to the returned `best_index_` while the `best_score_` attribute will not be available.

The refitted estimator is made available at the `best_estimator_` attribute and permits using `predict` directly on this `GridSearchCV` instance.

Also for multiple metric evaluation, the attributes `best_index_`, `best_score_` and `best_params_` will only be available if `refit` is set and all of them will be determined w.r.t this specific scorer.

See `scoring` parameter to know more about multiple metric evaluation.

Changed in version 0.20: Support for callable added.

verbose [integer] Controls the verbosity: the higher, the more messages.

error_score ['raise' or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error. Default is `np.nan`.

return_train_score [boolean, default=False] If `False`, the `cv_results_` attribute will not include training scores. Computing training scores is used to get insights on how different parameter settings impact the overfitting/underfitting trade-off. However computing the scores on the training set can be computationally expensive and is not strictly required to select the parameters that yield the best generalization performance.

Attributes

cv_results_ [dict of numpy (masked) ndarrays] A dict with keys as column headers and values as columns, that can be imported into a pandas `DataFrame`.

For instance the below given table

param_kernel	param_gamma	param_degree	split0_test_score	...	rank_t...
'poly'	–	2	0.80	...	2
'poly'	–	3	0.70	...	4
'rbf'	0.1	–	0.80	...	3
'rbf'	0.2	–	0.93	...	1

will be represented by a `cv_results_` dict of:

```
{
'param_kernel': masked_array(data = ['poly', 'poly', 'rbf', 'rbf'],
                             mask = [False False False False]...)
'param_gamma': masked_array(data = [-- -- 0.1 0.2],
                            mask = [ True  True False False]...),
'param_degree': masked_array(data = [2.0 3.0 -- --],
                             mask = [False False  True  True]...),
'split0_test_score'  : [0.80, 0.70, 0.80, 0.93],
'split1_test_score'  : [0.82, 0.50, 0.70, 0.78],
'mean_test_score'    : [0.81, 0.60, 0.75, 0.85],
'std_test_score'     : [0.01, 0.10, 0.05, 0.08],
'rank_test_score'    : [2, 4, 3, 1],
'split0_train_score' : [0.80, 0.92, 0.70, 0.93],
'split1_train_score' : [0.82, 0.55, 0.70, 0.87],
'mean_train_score'   : [0.81, 0.74, 0.70, 0.90],
'std_train_score'    : [0.01, 0.19, 0.00, 0.03],
'mean_fit_time'      : [0.73, 0.63, 0.43, 0.49],
'std_fit_time'       : [0.01, 0.02, 0.01, 0.01],
'mean_score_time'    : [0.01, 0.06, 0.04, 0.04],
'std_score_time'     : [0.00, 0.00, 0.00, 0.01],
'params'             : [{'kernel': 'poly', 'degree': 2}, ...],
}
```

NOTE

The key `'params'` is used to store a list of parameter settings dicts for all the parameter candidates.

The `mean_fit_time`, `std_fit_time`, `mean_score_time` and `std_score_time` are all in seconds.

For multi-metric evaluation, the scores for all the scorers are available in the `cv_results_` dict at the keys ending with that scorer's name (`'_<scorer_name>'`) instead of `'_score'` shown above. ('split0_test_precision', 'mean_train_precision' etc.)

best_estimator_ [estimator] Estimator that was chosen by the search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data. Not available if `refit=False`.

See `refit` parameter for more information on allowed values.

best_score_ [float] Mean cross-validated score of the best_estimator

For multi-metric evaluation, this is present only if `refit` is specified.

This attribute is not available if `refit` is a function.

best_params_ [dict] Parameter setting that gave the best results on the hold out data.

For multi-metric evaluation, this is present only if `refit` is specified.

best_index_ [int] The index (of the `cv_results_` arrays) which corresponds to the best candidate parameter setting.

The dict at `search.cv_results_['params'][search.best_index_]` gives the parameter setting for the best model, that gives the highest mean score (`search.best_score_`).

For multi-metric evaluation, this is present only if `refit` is specified.

scorer_ [function or a dict] Scorer function used on the held out data to choose the best parameters for the model.

For multi-metric evaluation, this attribute holds the validated `scoring` dict which maps the scorer key to the scorer callable.

n_splits_ [int] The number of cross-validation splits (folds/iterations).

refit_time_ [float] Seconds used for refitting the best model on the whole dataset.

This is present only if `refit` is not False.

See also:

ParameterGrid generates all the combinations of a hyperparameter grid.

sklearn.model_selection.train_test_split utility function to split the data into a development set usable for fitting a GridSearchCV instance and an evaluation set for its final evaluation.

sklearn.metrics.make_scorer Make a scorer from a performance metric or loss function.

Notes

The parameters selected are those that maximize the score of the left out data, unless an explicit score is passed in which case it is used instead.

If `n_jobs` was set to a value higher than one, the data is copied for each point in the grid (and not `n_jobs` times). This is done for efficiency reasons if individual jobs take very little time, but may raise errors if the dataset is large and not enough memory is available. A workaround in this case is to set `pre_dispatch`. Then, the memory is copied only `pre_dispatch` many times. A reasonable value for `pre_dispatch` is `2 * n_jobs`.

Examples

```
>>> from sklearn import svm, datasets
>>> from sklearn.model_selection import GridSearchCV
>>> iris = datasets.load_iris()
>>> parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
>>> svc = svm.SVC()
>>> clf = GridSearchCV(svc, parameters)
>>> clf.fit(iris.data, iris.target)
GridSearchCV(estimator=SVC(),
             param_grid={'C': [1, 10], 'kernel': ('linear', 'rbf')})
>>> sorted(clf.cv_results_.keys())
['mean_fit_time', 'mean_score_time', 'mean_test_score',...
 'param_C', 'param_kernel', 'params',...
 'rank_test_score', 'split0_test_score',...
 'split2_test_score', ...
 'std_fit_time', 'std_score_time', 'std_test_score']
```

Methods

decision_function(self, X)	Call decision_function on the estimator with the best found parameters.
fit(self, X[, y, groups])	Run fit with all sets of parameters.
get_params(self[, deep])	Get parameters for this estimator.

Continued on next page

Table 220 – continued from previous page

inverse_transform(self, Xt)	Call inverse_transform on the estimator with the best found params.
predict(self, X)	Call predict on the estimator with the best found parameters.
predict_log_proba(self, X)	Call predict_log_proba on the estimator with the best found parameters.
predict_proba(self, X)	Call predict_proba on the estimator with the best found parameters.
score(self, X[, y])	Returns the score on the given data, if the estimator has been refit.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Call transform on the estimator with the best found parameters.

__init__(*self*, *estimator*, *param_grid*, *scoring=None*, *n_jobs=None*, *iid='deprecated'*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *error_score=nan*, *return_train_score=False*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Call decision_function on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `decision_function`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

fit(*self*, *X*, *y=None*, *groups=None*, ***fit_params*)
Run fit with all sets of parameters.

Parameters

X [array-like of shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y [array-like of shape (n_samples, n_output) or (n_samples,), optional] Target relative to X for classification or regression; None for unsupervised learning.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., *GroupKFold*).

****fit_params** [dict of string -> object] Parameters passed to the `fit` method of the estimator

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *Xt*)
Call inverse_transform on the estimator with the best found params.

Only available if the underlying estimator implements `inverse_transform` and `refit=True`.

Parameters

Xt [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict(*self*, *X*)

Call predict on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict_log_proba(*self*, *X*)

Call predict_log_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_log_proba`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict_proba(*self*, *X*)

Call predict_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_proba`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

score(*self*, *X*, *y=None*)

Returns the score on the given data, if the estimator has been refit.

This uses the score defined by `scoring` where provided, and the `best_estimator_.score` method otherwise.

Parameters

X [array-like of shape (n_samples, n_features)] Input data, where n_samples is the number of samples and n_features is the number of features.

y [array-like of shape (n_samples, n_output) or (n_samples,), optional] Target relative to X for classification or regression; None for unsupervised learning.

Returns

score [float]

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Call transform on the estimator with the best found parameters.

Only available if the underlying estimator supports `transform` and `refit=True`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Examples using `sklearn.model_selection.GridSearchCV`

- *Sample pipeline for text feature extraction and evaluation*

sklearn.model_selection.ParameterGrid

class sklearn.model_selection.**ParameterGrid**(*param_grid*)
Grid of parameters with a discrete number of values for each.

Can be used to iterate over parameter value combinations with the Python built-in function iter.

Read more in the *User Guide*.

Parameters

param_grid [dict of string to sequence, or sequence of such] The parameter grid to explore, as a dictionary mapping estimator parameters to sequences of allowed values.

An empty dict signifies default parameters.

A sequence of dicts signifies a sequence of grids to search, and is useful to avoid exploring parameter combinations that make no sense or have no effect. See the examples below.

See also:

GridSearchCV Uses *ParameterGrid* to perform a full parallelized parameter search.

Examples

```
>>> from sklearn.model_selection import ParameterGrid
>>> param_grid = {'a': [1, 2], 'b': [True, False]}
>>> list(ParameterGrid(param_grid)) == (
...     [{'a': 1, 'b': True}, {'a': 1, 'b': False},
...      {'a': 2, 'b': True}, {'a': 2, 'b': False}])
True
```

```
>>> grid = [{'kernel': ['linear']}, {'kernel': ['rbf'], 'gamma': [1, 10]}]
>>> list(ParameterGrid(grid)) == [{'kernel': 'linear'},
...                               {'kernel': 'rbf', 'gamma': 1},
...                               {'kernel': 'rbf', 'gamma': 10}]
True
>>> ParameterGrid(grid)[1] == {'kernel': 'rbf', 'gamma': 1}
True
```

__init__(*self*, *param_grid*)
Initialize self. See help(type(self)) for accurate signature.

sklearn.model_selection.ParameterSampler

class sklearn.model_selection.**ParameterSampler**(*param_distributions*, *n_iter*, *random_state=None*)

Generator on parameters sampled from given distributions.

Non-deterministic iterable over random candidate combinations for hyper- parameter search. If all parameters are presented as a list, sampling without replacement is performed. If at least one parameter is given as a distribution, sampling with replacement is used. It is highly recommended to use continuous distributions for continuous parameters.

Read more in the User Guide.

Parameters

param_distributions [dict] Dictionary with parameters names (string) as keys and distributions or lists of parameters to try. Distributions must provide a `rvs` method for sampling (such as those from scipy.stats.distributions). If a list is given, it is sampled uniformly. If a list of dicts is given, first a dict is sampled uniformly, and then a parameter is sampled using that dict as above.

n_iter [integer] Number of parameter settings that are produced.

random_state [int, RandomState instance or None, optional (default=None)] Pseudo random number generator state used for random uniform sampling from lists of possible values instead of scipy.stats distributions. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Returns

params [dict of string to any] **Yields** dictionaries mapping each estimator parameter to as sampled value.

Examples

```
>>> from sklearn.model_selection import ParameterSampler
>>> from scipy.stats.distributions import expon
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> param_grid = {'a':[1, 2], 'b': expon()}
>>> param_list = list(ParameterSampler(param_grid, n_iter=4,
...                                    random_state=rng))
>>> rounded_list = [dict((k, round(v, 6)) for (k, v) in d.items())
...                 for d in param_list]
>>> rounded_list == [{'b': 0.89856, 'a': 1},
...                  {'b': 0.923223, 'a': 1},
...                  {'b': 1.878964, 'a': 2},
...                  {'b': 1.038159, 'a': 2}]
True
```

__init__(*self*, *param_distributions*, *n_iter*, *random_state=None*)

Initialize self. See help(type(self)) for accurate signature.

sklearn.model_selection.RandomizedSearchCV

class sklearn.model_selection.**RandomizedSearchCV**(*estimator*, *param_distributions*, *n_iter=10*, *scoring=None*, *n_jobs=None*, *iid='deprecated'*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *random_state=None*, *error_score=nan*, *return_train_score=False*)

Randomized search on hyper parameters.

RandomizedSearchCV implements a "fit" and a "score" method. It also implements "predict", "predict_proba", "decision_function", "transform" and "inverse_transform" if they are implemented in the estimator used.

The parameters of the estimator used to apply these methods are optimized by cross-validated search over parameter settings.

In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that are tried is given by n_iter.

If all parameters are presented as a list, sampling without replacement is performed. If at least one parameter is given as a distribution, sampling with replacement is used. It is highly recommended to use continuous distributions for continuous parameters.

Read more in the *User Guide*.

New in version 0.14.

Parameters

estimator [estimator object.] A object of that type is instantiated for each grid point. This is assumed to implement the scikit-learn estimator interface. Either estimator needs to provide a `score` function, or `scoring` must be passed.

param_distributions [dict or list of dicts] Dictionary with parameters names (string) as keys and distributions or lists of parameters to try. Distributions must provide a `rvs` method for sampling (such as those from scipy.stats.distributions). If a list is given, it is sampled uniformly. If a list of dicts is given, first a dict is sampled uniformly, and then a parameter is sampled using that dict as above.

n_iter [int, default=10] Number of parameter settings that are sampled. n_iter trades off runtime vs quality of the solution.

scoring [string, callable, list/tuple, dict or None, default: None] A single string (see *The scoring parameter: defining model evaluation rules*) or a callable (see *Defining your scoring strategy from metric functions*) to evaluate the predictions on the test set.

For evaluating multiple metrics, either give a list of (unique) strings or a dict with names as keys and callables as values.

NOTE that when using custom scorers, each scorer should return a single value. Metric functions returning a list/array of values can be wrapped into multiple scorers that return one value each.

See *Specifying multiple metrics for evaluation* for an example.

If None, the estimator's score method is used.

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

pre_dispatch [int, or string, optional] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A string, giving an expression as a function of n_jobs, as in '2*n_jobs'

iid [boolean, default=False] If True, return the average score across folds, weighted by the number of samples in each test set. In this case, the data is assumed to be identically distributed across the folds, and the loss minimized is the total loss per sample, and not the mean loss across the folds.

Deprecated since version 0.22: Parameter `iid` is deprecated in 0.22 and will be removed in 0.24

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, *`StratifiedKFold`* is used. In all other cases, *`KFold`* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

refit [boolean, string, or callable, default=True] Refit an estimator using the best found parameters on the whole dataset.

For multiple metric evaluation, this needs to be a string denoting the scorer that would be used to find the best parameters for refitting the estimator at the end.

Where there are considerations other than maximum score in choosing a best estimator, `refit` can be set to a function which returns the selected `best_index_` given the `cv_results`. In that case, the `best_estimator_` and `best_parameters_` will be set according to the returned `best_index_` while the `best_score_` attribute will not be available.

The refitted estimator is made available at the `best_estimator_` attribute and permits using `predict` directly on this `RandomizedSearchCV` instance.

Also for multiple metric evaluation, the attributes `best_index_`, `best_score_` and `best_params_` will only be available if `refit` is set and all of them will be determined w.r.t this specific scorer.

See `scoring` parameter to know more about multiple metric evaluation.

Changed in version 0.20: Support for callable added.

verbose [integer] Controls the verbosity: the higher, the more messages.

random_state [int, RandomState instance or None, optional, default=None] Pseudo random number generator state used for random uniform sampling from lists of possible values

instead of scipy.stats distributions. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

error_score ['raise' or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error. Default is `np.nan`.

return_train_score [boolean, default=False] If `False`, the `cv_results_` attribute will not include training scores. Computing training scores is used to get insights on how different parameter settings impact the overfitting/underfitting trade-off. However computing the scores on the training set can be computationally expensive and is not strictly required to select the parameters that yield the best generalization performance.

Attributes

cv_results_ [dict of numpy (masked) ndarrays] A dict with keys as column headers and values as columns, that can be imported into a pandas `DataFrame`.

For instance the below given table

param_kernel	param_gamma	split0_test_score	...	rank_test_score
'rbf'	0.1	0.80	...	2
'rbf'	0.2	0.90	...	1
'rbf'	0.3	0.70	...	1

will be represented by a `cv_results_` dict of:

```
{
'param_kernel' : masked_array(data = ['rbf', 'rbf', 'rbf'],
                              mask = False),
'param_gamma'  : masked_array(data = [0.1 0.2 0.3], mask = False),
'split0_test_score'  : [0.80, 0.90, 0.70],
'split1_test_score'  : [0.82, 0.50, 0.70],
'mean_test_score'    : [0.81, 0.70, 0.70],
'std_test_score'     : [0.01, 0.20, 0.00],
'rank_test_score'    : [3, 1, 1],
'split0_train_score' : [0.80, 0.92, 0.70],
'split1_train_score' : [0.82, 0.55, 0.70],
'mean_train_score'   : [0.81, 0.74, 0.70],
'std_train_score'    : [0.01, 0.19, 0.00],
'mean_fit_time'      : [0.73, 0.63, 0.43],
'std_fit_time'       : [0.01, 0.02, 0.01],
'mean_score_time'    : [0.01, 0.06, 0.04],
'std_score_time'     : [0.00, 0.00, 0.00],
'params'             : [{'kernel' : 'rbf', 'gamma' : 0.1}, ...],
}
```

NOTE

The key `'params'` is used to store a list of parameter settings dicts for all the parameter candidates.

The `mean_fit_time`, `std_fit_time`, `mean_score_time` and `std_score_time` are all in seconds.

For multi-metric evaluation, the scores for all the scorers are available in the `cv_results_` dict at the keys ending with that scorer's name (`'_<scorer_name>'`)

instead of '_score' shown above. ('split0_test_precision', 'mean_train_precision' etc.)

best_estimator_ [estimator] Estimator that was chosen by the search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data. Not available if refit=False.

For multi-metric evaluation, this attribute is present only if refit is specified.

See refit parameter for more information on allowed values.

best_score_ [float] Mean cross-validated score of the best_estimator.

For multi-metric evaluation, this is not available if refit is False. See refit parameter for more information.

This attribute is not available if refit is a function.

best_params_ [dict] Parameter setting that gave the best results on the hold out data.

For multi-metric evaluation, this is not available if refit is False. See refit parameter for more information.

best_index_ [int] The index (of the cv_results_ arrays) which corresponds to the best candidate parameter setting.

The dict at search.cv_results_['params'][search.best_index_] gives the parameter setting for the best model, that gives the highest mean score (search.best_score_).

For multi-metric evaluation, this is not available if refit is False. See refit parameter for more information.

scorer_ [function or a dict] Scorer function used on the held out data to choose the best parameters for the model.

For multi-metric evaluation, this attribute holds the validated scoring dict which maps the scorer key to the scorer callable.

n_splits_ [int] The number of cross-validation splits (folds/iterations).

refit_time_ [float] Seconds used for refitting the best model on the whole dataset.

This is present only if refit is not False.

See also:

GridSearchCV Does exhaustive search over a grid of parameters.

ParameterSampler A generator over parameter settings, constructed from param_distributions.

Notes

The parameters selected are those that maximize the score of the held-out data, according to the scoring parameter.

If n_jobs was set to a value higher than one, the data is copied for each parameter setting(and not n_jobs times). This is done for efficiency reasons if individual jobs take very little time, but may raise errors if the dataset is large and not enough memory is available. A workaround in this case is to set pre_dispatch. Then, the memory is copied only pre_dispatch many times. A reasonable value for pre_dispatch is 2 * n_jobs.

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.model_selection import RandomizedSearchCV
>>> from scipy.stats import uniform
>>> iris = load_iris()
>>> logistic = LogisticRegression(solver='saga', tol=1e-2, max_iter=200,
...                               random_state=0)
>>> distributions = dict(C=uniform(loc=0, scale=4),
...                      penalty=['l2', 'l1'])
>>> clf = RandomizedSearchCV(logistic, distributions, random_state=0)
>>> search = clf.fit(iris.data, iris.target)
>>> search.best_params_
{'C': 2..., 'penalty': 'l1'}
```

Methods

decision_function(self, X)	Call decision_function on the estimator with the best found parameters.
fit(self, X[, y, groups])	Run fit with all sets of parameters.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, Xt)	Call inverse_transform on the estimator with the best found params.
predict(self, X)	Call predict on the estimator with the best found parameters.
predict_log_proba(self, X)	Call predict_log_proba on the estimator with the best found parameters.
predict_proba(self, X)	Call predict_proba on the estimator with the best found parameters.
score(self, X[, y])	Returns the score on the given data, if the estimator has been refit.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Call transform on the estimator with the best found parameters.

__init__(*self*, *estimator*, *param_distributions*, *n_iter=10*, *scoring=None*, *n_jobs=None*, *iid='deprecated'*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *random_state=None*, *error_score=nan*, *return_train_score=False*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Call decision_function on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `decision_function`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

fit(*self*, *X*, *y=None*, *groups=None*, ***fit_params*)
Run fit with all sets of parameters.

Parameters

X [array-like of shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y [array-like of shape (n_samples, n_output) or (n_samples,), optional] Target relative to X for classification or regression; None for unsupervised learning.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., `GroupKFold`).

****fit_params** [dict of string -> object] Parameters passed to the `fit` method of the estimator

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *Xt*)

Call inverse_transform on the estimator with the best found params.

Only available if the underlying estimator implements `inverse_transform` and `refit=True`.

Parameters

Xt [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict(*self*, *X*)

Call predict on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict_log_proba(*self*, *X*)

Call predict_log_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_log_proba`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict_proba(*self*, *X*)

Call predict_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_proba`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

score(*self*, *X*, *y=None*)
Returns the score on the given data, if the estimator has been refit.

This uses the score defined by `scoring` where provided, and the `best_estimator_.score` method otherwise.

Parameters

X [array-like of shape (n_samples, n_features)] Input data, where n_samples is the number of samples and n_features is the number of features.

y [array-like of shape (n_samples, n_output) or (n_samples,), optional] Target relative to X for classification or regression; None for unsupervised learning.

Returns

score [float]

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Call transform on the estimator with the best found parameters.

Only available if the underlying estimator supports `transform` and `refit=True`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Examples using `sklearn.model_selection.RandomizedSearchCV`

- *Comparing randomized search and grid search for hyperparameter estimation*

model_selection.fit_grid_point(X, y, ... [, ...])	Run fit on one set of parameters.

sklearn.model_selection.fit_grid_point

sklearn.model_selection.**fit_grid_point**(*X*, *y*, *estimator*, *parameters*, *train*, *test*, *scorer*, *verbose*, *error_score=nan*, ***fit_params*)
Run fit on one set of parameters.

Parameters

X [array-like, sparse matrix or list] Input data.

y [array-like or None] Targets for input data.

estimator [estimator object] A object of that type is instantiated for each grid point. This is assumed to implement the scikit-learn estimator interface. Either estimator needs to provide a `score` function, or `scoring` must be passed.

parameters [dict] Parameters to be set on estimator for this grid point.

train [ndarray, dtype int or bool] Boolean mask or indices for training set.

test [ndarray, dtype int or bool] Boolean mask or indices for test set.

scorer [callable or None] The scorer callable object / function must have its signature as `scorer(estimator, X, y)`.

If `None` the estimator's score method is used.

verbose [int] Verbosity level.

****fit_params** [kwargs] Additional parameter passed to the fit function of the estimator.

error_score ['raise' or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error. Default is `np.nan`.

Returns

score [float] Score of this parameter setting on given test split.

parameters [dict] The parameters that have been evaluated.

n_samples_test [int] Number of test samples in this split.

7.26.4 Model validation

model_selection.cross_validate(estimator, X)	Evaluate metric(s) by cross-validation and also record fit/score times.
model_selection.cross_val_predict(estimator, X)	Generate cross-validated estimates for each input data point
model_selection.cross_val_score(estimator, X)	Evaluate a score by cross-validation
model_selection.learning_curve(estimator, X, y)	Learning curve.
model_selection.permutation_test_score(...)	Evaluate the significance of a cross-validated score with permutations
model_selection.validation_curve(estimator, ...)	Validation curve.

sklearn.model_selection.cross_validate

sklearn.model_selection.**cross_validate**(*estimator*, *X*, *y=None*, *groups=None*, *scoring=None*, *cv=None*, *n_jobs=None*, *verbose=0*, *fit_params=None*, *pre_dispatch='2*n_jobs'*, *return_train_score=False*, *return_estimator=False*, *error_score=nan*)

Evaluate metric(s) by cross-validation and also record fit/score times.

Read more in the *User Guide*.

Parameters

estimator [estimator object implementing 'fit'] The object to use to fit the data.

X [array-like] The data to fit. Can be for example a list, or an array.

y [array-like, optional, default: None] The target variable to try to predict in the case of supervised learning.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., `GroupKFold`).

scoring [string, callable, list/tuple, dict or None, default: None] A single string (see *The scoring parameter: defining model evaluation rules*) or a callable (see *Defining your scoring strategy from metric functions*) to evaluate the predictions on the test set.

For evaluating multiple metrics, either give a list of (unique) strings or a dict with names as keys and callables as values.

NOTE that when using custom scorers, each scorer should return a single value. Metric functions returning a list/array of values can be wrapped into multiple scorers that return one value each.

See *Specifying multiple metrics for evaluation* for an example.

If None, the estimator's score method is used.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

n_jobs [int or None, optional (default=None)] The number of CPUs to use to do the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [integer, optional] The verbosity level.

fit_params [dict, optional] Parameters to pass to the fit method of the estimator.

pre_dispatch [int, or string, optional] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A string, giving an expression as a function of n_jobs, as in '2*n_jobs'

return_train_score [boolean, default=False] Whether to include train scores. Computing training scores is used to get insights on how different parameter settings impact the overfitting/underfitting trade-off. However computing the scores on the training set can be computationally expensive and is not strictly required to select the parameters that yield the best generalization performance.

return_estimator [boolean, default False] Whether to return the estimators fitted on each split.

error_score ['raise' or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error.

Returns

scores [dict of float arrays of shape (n_splits,)] Array of scores of the estimator for each run of the cross validation.

A dict of arrays containing the score/time arrays for each scorer is returned. The possible keys for this `dict` are:

`test_score` The score array for test scores on each cv split. Suffix `_score` in `test_score` changes to a specific metric like `test_r2` or `test_auc` if there are multiple scoring metrics in the scoring parameter.

`train_score` The score array for train scores on each cv split. Suffix `_score` in `train_score` changes to a specific metric like `train_r2` or `train_auc` if there are multiple scoring metrics in the scoring parameter. This is available only if `return_train_score` parameter is `True`.

`fit_time` The time for fitting the estimator on the train set for each cv split.

`score_time` The time for scoring the estimator on the test set for each cv split. (Note time for scoring on the train set is not included even if `return_train_score` is set to `True`

`estimator` The estimator objects for each cv split. This is available only if `return_estimator` parameter is set to `True`.

See also:

`sklearn.model_selection.cross_val_score` Run cross-validation for single metric evaluation.

`sklearn.model_selection.cross_val_predict` Get predictions from each split of cross-validation for diagnostic purposes.

`sklearn.metrics.make_scorer` Make a scorer from a performance metric or loss function.

Examples

```
>>> from sklearn import datasets, linear_model
>>> from sklearn.model_selection import cross_validate
>>> from sklearn.metrics import make_scorer
>>> from sklearn.metrics import confusion_matrix
>>> from sklearn.svm import LinearSVC
>>> diabetes = datasets.load_diabetes()
>>> X = diabetes.data[:150]
>>> y = diabetes.target[:150]
>>> lasso = linear_model.Lasso()
```

Single metric evaluation using `cross_validate`

```
>>> cv_results = cross_validate(lasso, X, y, cv=3)
>>> sorted(cv_results.keys())
['fit_time', 'score_time', 'test_score']
>>> cv_results['test_score']
array([0.33150734, 0.08022311, 0.03531764])
```

Multiple metric evaluation using `cross_validate` (please refer the `scoring` parameter doc for more information)

```
>>> scores = cross_validate(lasso, X, y, cv=3,
...                         scoring=('r2', 'neg_mean_squared_error'),
...                         return_train_score=True)
>>> print(scores['test_neg_mean_squared_error'])
[-3635.5... -3573.3... -6114.7...]
>>> print(scores['train_r2'])
[0.28010158 0.39088426 0.22784852]
```

Examples using `sklearn.model_selection.cross_validate`

- *Combine predictors using stacking*

`sklearn.model_selection.`cross_val_predict

sklearn.model_selection.**cross_val_predict**(*estimator*, *X*, *y=None*, *groups=None*, *cv=None*, *n_jobs=None*, *verbose=0*, *fit_params=None*, *pre_dispatch='2*n_jobs'*, *method='predict'*)

Generate cross-validated estimates for each input data point

The data is split according to the cv parameter. Each sample belongs to exactly one test set, and its prediction is computed with an estimator fitted on the corresponding training set.

Passing these predictions into an evaluation metric may not be a valid way to measure generalization performance. Results can differ from *cross_validate* and *cross_val_score* unless all tests sets have equal size and the metric decomposes over samples.

Read more in the *User Guide*.

Parameters

estimator [estimator object implementing 'fit' and 'predict'] The object to use to fit the data.

X [array-like] The data to fit. Can be, for example a list, or an array at least 2d.

y [array-like, optional, default: None] The target variable to try to predict in the case of supervised learning.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., *GroupKFold*).

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,

- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, *`StratifiedKFold`* is used. In all other cases, *`KFold`* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

n_jobs [int or None, optional (default=None)] The number of CPUs to use to do the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [integer, optional] The verbosity level.

fit_params [dict, optional] Parameters to pass to the fit method of the estimator.

pre_dispatch [int, or string, optional] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A string, giving an expression as a function of n_jobs, as in '2*n_jobs'

method [string, optional, default: 'predict'] Invokes the passed method name of the passed estimator. For method='predict_proba', the columns correspond to the classes in sorted order.

Returns

predictions [ndarray] This is the result of calling `method`

See also:

cross_val_score calculate score for each CV split

cross_validate calculate one or more scores and timings for each CV split

Notes

In the case that one or more classes are absent in a training portion, a default score needs to be assigned to all instances for that class if `method` produces columns per class, as in {'decision_function', 'predict_proba', 'predict_log_proba'}. For `predict_proba` this value is 0. In order to ensure finite output, we approximate negative infinity by the minimum finite float value for the dtype in other cases.

Examples

```
>>> from sklearn import datasets, linear_model
>>> from sklearn.model_selection import cross_val_predict
>>> diabetes = datasets.load_diabetes()
>>> X = diabetes.data[:150]
>>> y = diabetes.target[:150]
>>> lasso = linear_model.Lasso()
>>> y_pred = cross_val_predict(lasso, X, y, cv=3)
```

Examples using `sklearn.model_selection.cross_val_predict`

- *Combine predictors using stacking*
- *Plotting Cross-Validated Predictions*

`sklearn.model_selection.cross_val_score`

`sklearn.model_selection.cross_val_score`(*estimator*, *X*, *y=None*, *groups=None*, *scoring=None*, *cv=None*, *n_jobs=None*, *verbose=0*, *fit_params=None*, *pre_dispatch='2*n_jobs'*, *error_score=nan*)

Evaluate a score by cross-validation

Read more in the *User Guide*.

Parameters

estimator [estimator object implementing 'fit'] The object to use to fit the data.

X [array-like] The data to fit. Can be for example a list, or an array.

y [array-like, optional, default: None] The target variable to try to predict in the case of supervised learning.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., *GroupKFold*).

scoring [string, callable or None, optional, default: None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)` which should return only a single value.

Similar to *cross_validate* but only a single metric is permitted.

If None, the estimator's default scorer (if available) is used.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, *StratifiedKFold* is used. In all other cases, *KFold* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

n_jobs [int or None, optional (default=None)] The number of CPUs to use to do the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [integer, optional] The verbosity level.

fit_params [dict, optional] Parameters to pass to the fit method of the estimator.

pre_dispatch [int, or string, optional] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A string, giving an expression as a function of n_jobs, as in '2*n_jobs'

error_score ['raise' or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error.

Returns

scores [array of float, shape=(len(list(cv)),)] Array of scores of the estimator for each run of the cross validation.

See also:

sklearn.model_selection.cross_validate To run cross-validation on multiple metrics and also to return train scores, fit times and score times.

sklearn.model_selection.cross_val_predict Get predictions from each split of cross-validation for diagnostic purposes.

sklearn.metrics.make_scorer Make a scorer from a performance metric or loss function.

Examples

```
>>> from sklearn import datasets, linear_model
>>> from sklearn.model_selection import cross_val_score
>>> diabetes = datasets.load_diabetes()
>>> X = diabetes.data[:150]
>>> y = diabetes.target[:150]
>>> lasso = linear_model.Lasso()
>>> print(cross_val_score(lasso, X, y, cv=3))
[0.33150734 0.08022311 0.03531764]
```

Examples using `sklearn.model_selection.cross_val_score`

- *Model selection with Probabilistic PCA and Factor Analysis (FA)*
- *Imputing missing values with variants of IterativeImputer*
- *Imputing missing values before building an estimator*
- *Underfitting vs. Overfitting*
- *Receiver Operating Characteristic (ROC) with cross validation*
- *Nested versus non-nested cross-validation*
- *SVM-Anova: SVM with univariate feature selection*
- *Cross-validation on Digits Dataset Exercise*

sklearn.model_selection.learning_curve

sklearn.model_selection.**learning_curve**(*estimator*, *X*, *y*, *groups=None*, *train_sizes=array([0.1, 0.325, 0.55, 0.775, 1.])*, *cv=None*, *scoring=None*, *exploit_incremental_learning=False*, *n_jobs=None*, *pre_dispatch='all'*, *verbose=0*, *shuffle=False*, *random_state=None*, *error_score=nan*, *return_times=False*)

Learning curve.

Determines cross-validated training and test scores for different training set sizes.

A cross-validation generator splits the whole dataset k times in training and test data. Subsets of the training set with varying sizes will be used to train the estimator and a score for each training subset size and the test set will be computed. Afterwards, the scores will be averaged over all k runs for each training subset size.

Read more in the *User Guide*.

Parameters

estimator [object type that implements the "fit" and "predict" methods] An object of that type which is cloned for each validation.

X [array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., *GroupKFold*).

train_sizes [array-like, shape (n_ticks,), dtype float or int] Relative or absolute numbers of training examples that will be used to generate the learning curve. If the dtype is float, it is regarded as a fraction of the maximum size of the training set (that is determined by the selected validation method), i.e. it has to be within (0, 1]. Otherwise it is interpreted as absolute sizes of the training sets. Note that for classification the number of samples usually have to be big enough to contain at least one sample from each class. (default: np.linspace(0.1, 1.0, 5))

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, *StratifiedKFold* is used. In all other cases, *KFold* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

scoring [string, callable or None, optional, default: None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`.

exploit_incremental_learning [boolean, optional, default: False] If the estimator supports incremental learning, this will be used to speed up fitting for different training set sizes.

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

pre_dispatch [integer or string, optional] Number of predispatched jobs for parallel execution (default is all). The option can reduce the allocated memory. The string can be an expression like '2*n_jobs'.

verbose [integer, optional] Controls the verbosity: the higher, the more messages.

shuffle [boolean, optional] Whether to shuffle training data before taking prefixes of it based on''train_sizes''.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `shuffle` is True.

error_score ['raise' or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error.

return_times [boolean, optional (default: False)] Whether to return the fit and score times.

Returns

train_sizes_abs [array, shape (n_unique_ticks,), dtype int] Numbers of training examples that has been used to generate the learning curve. Note that the number of ticks might be less than n_ticks because duplicate entries will be removed.

train_scores [array, shape (n_ticks, n_cv_folds)] Scores on training sets.

test_scores [array, shape (n_ticks, n_cv_folds)] Scores on test set.

fit_times [array, shape (n_ticks, n_cv_folds)] Times spent for fitting in seconds. Only present if `return_times` is True.

score_times [array, shape (n_ticks, n_cv_folds)] Times spent for scoring in seconds. Only present if `return_times` is True.

Notes

See *examples/model_selection/plot_learning_curve.py*

Examples using `sklearn.model_selection.learning_curve`

- *Comparison of kernel ridge regression and SVR*
- *Plotting Learning Curves*

sklearn.model_selection.permutation_test_score

sklearn.model_selection.**permutation_test_score**(*estimator*, *X*, *y*, *groups=None*, *cv=None*, *n_permutations=100*, *n_jobs=None*, *random_state=0*, *verbose=0*, *scoring=None*)

Evaluate the significance of a cross-validated score with permutations

Read more in the *User Guide*.

Parameters

estimator [estimator object implementing 'fit'] The object to use to fit the data.

X [array-like of shape at least 2D] The data to fit.

y [array-like] The target variable to try to predict in the case of supervised learning.

groups [array-like, with shape (n_samples,), optional] Labels to constrain permutation within groups, i.e. `y` values are permuted among samples with the same group identifier. When not specified, `y` values are permuted among all samples.

When a grouped cross-validator is used, the group labels are also passed on to the `split` method of the cross-validator. The cross-validator uses them for grouping the samples while splitting the dataset into train/test set.

scoring [string, callable or None, optional, default: None] A single string (see *The scoring parameter: defining model evaluation rules*) or a callable (see *Defining your scoring strategy from metric functions*) to evaluate the predictions on the test set.

If None the estimator's score method is used.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, *`StratifiedKFold`* is used. In all other cases, *`KFold`* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

n_permutations [integer, optional] Number of times to permute `y`.

n_jobs [int or None, optional (default=None)] The number of CPUs to use to do the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional (default=0)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

verbose [integer, optional] The verbosity level.

Returns

score [float] The true score without permuting targets.

permutation_scores [array, shape (n_permutations,)] The scores obtained for each permutations.

pvalue [float] The p-value, which approximates the probability that the score would be obtained by chance. This is calculated as:

```
(C + 1) / (n_permutations + 1)
```

Where C is the number of permutations whose score >= the true score.

The best possible p-value is 1/(n_permutations + 1), the worst is 1.0.

Notes

This function implements Test 1 in:

Ojala and Garriga. Permutation Tests for Studying Classifier Performance. The Journal of Machine Learning Research (2010) vol. 11

Examples using `sklearn.model_selection.permutation_test_score`

- *Test with permutations the significance of a classification score*

`sklearn.model_selection.validation_curve`

`sklearn.model_selection.validation_curve`(*estimator*, *X*, *y*, *param_name*, *param_range*, *groups=None*, *cv=None*, *scoring=None*, *n_jobs=None*, *pre_dispatch='all'*, *verbose=0*, *error_score=nan*)

Validation curve.

Determine training and test scores for varying parameter values.

Compute scores for an estimator with different values of a specified parameter. This is similar to grid search with one parameter. However, this will also compute training scores and is merely a utility for plotting the results.

Read more in the *User Guide*.

Parameters

estimator [object type that implements the "fit" and "predict" methods] An object of that type which is cloned for each validation.

X [array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

param_name [string] Name of the parameter that will be varied.

param_range [array-like, shape (n_values,)] The values of the parameter that will be evaluated.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., `GroupKFold`).

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, *`StratifiedKFold`* is used. In all other cases, *`KFold`* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

scoring [string, callable or None, optional, default: None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`.

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

pre_dispatch [integer or string, optional] Number of predispatched jobs for parallel execution (default is all). The option can reduce the allocated memory. The string can be an expression like '2*n_jobs'.

verbose [integer, optional] Controls the verbosity: the higher, the more messages.

error_score ['raise' or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error.

Returns

train_scores [array, shape (n_ticks, n_cv_folds)] Scores on training sets.

test_scores [array, shape (n_ticks, n_cv_folds)] Scores on test set.

Notes

See *Plotting Validation Curves*

Examples using `sklearn.model_selection.validation_curve`

- *Plotting Validation Curves*

7.27 `sklearn.multiclass`: Multiclass and multilabel classification

7.27.1 Multiclass and multilabel classification strategies

This module implements multiclass learning algorithms:

- one-vs-the-rest / one-vs-all
- one-vs-one
- error correcting output codes

The estimators provided in this module are meta-estimators: they require a base estimator to be provided in their constructor. For example, it is possible to use these estimators to turn a binary classifier or a regressor into a multiclass classifier. It is also possible to use these estimators with multiclass estimators in the hope that their accuracy or runtime performance improves.

All classifiers in scikit-learn implement multiclass classification; you only need to use this module if you want to experiment with custom multiclass strategies.

The one-vs-the-rest meta-classifier also implements a `predict_proba` method, so long as such a method is implemented by the base classifier. This method returns probabilities of class membership in both the single label and multilabel case. Note that in the multilabel case, probabilities are the marginal probability that a given sample falls in the given class. As such, in the multilabel case the sum of these probabilities over all possible labels for a given sample *will not* sum to unity, as they do in the single label case.

User guide: See the *Multiclass and multilabel algorithms* section for further details.

multiclass.OneVsRestClassifier(estimator[, ...])	One-vs-the-rest (OvR) multiclass/multilabel strategy
multiclass.OneVsOneClassifier(estimator[, ...])	One-vs-one multiclass strategy
multiclass.OutputCodeClassifier(estimator[, ...])	(Error-Correcting) Output-Code multiclass strategy

7.27.2 sklearn.multiclass.OneVsRestClassifier

class sklearn.multiclass.**OneVsRestClassifier**(*estimator*, *n_jobs=None*)

One-vs-the-rest (OvR) multiclass/multilabel strategy

Also known as one-vs-all, this strategy consists in fitting one classifier per class. For each classifier, the class is fitted against all the other classes. In addition to its computational efficiency (only `n_classes` classifiers are needed), one advantage of this approach is its interpretability. Since each class is represented by one and one classifier only, it is possible to gain knowledge about the class by inspecting its corresponding classifier. This is the most commonly used strategy for multiclass classification and is a fair default choice.

This strategy can also be used for multilabel learning, where a classifier is used to predict multiple labels for instance, by fitting on a 2-d matrix in which cell [i, j] is 1 if sample i has label j and 0 otherwise.

In the multilabel learning literature, OvR is also known as the binary relevance method.

Read more in the *User Guide*.

Parameters

estimator [estimator object] An estimator object implementing *fit* and one of *decision_function* or *predict_proba*.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

estimators_ [list of `n_classes` estimators] Estimators used for predictions.

classes_ [array, shape = [`n_classes`]] Class labels.

n_classes_ [int] Number of classes.

label_binarizer_ [LabelBinarizer object] Object used to transform multiclass labels to binary labels and vice-versa.

multilabel_ [boolean] Whether this is a multilabel classifier

Examples

```
>>> import numpy as np
>>> from sklearn.multiclass import OneVsRestClassifier
>>> from sklearn.svm import SVC
>>> X = np.array([
...     [10, 10],
...     [8, 10],
...     [-5, 5.5],
...     [-5.4, 5.5],
...     [-20, -20],
...     [-15, -20]
... ])
>>> y = np.array([0, 0, 1, 1, 2, 2])
>>> clf = OneVsRestClassifier(SVC()).fit(X, y)
>>> clf.predict([[-19, -20], [9, 9], [-5, 5]])
array([2, 0, 1])
```

Methods

decision_function(self, X)	Returns the distance of each sample from the decision boundary for each class.
fit(self, X, y)	Fit underlying estimators.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes])	Partially fit underlying estimators
predict(self, X)	Predict multi-class targets using underlying estimators.
predict_proba(self, X)	Probability estimates.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *estimator*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

decision_function (*self*, *X*)
Returns the distance of each sample from the decision boundary for each class. This can only be used with estimators which implement the decision_function method.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

T [array-like of shape (n_samples, n_classes)]

fit (*self*, *X*, *y*)
Fit underlying estimators.

Parameters

X [(sparse) array-like of shape (n_samples, n_features)] Data.

y [(sparse) array-like of shape (n_samples,) or (n_samples, n_classes)] Multi-class targets. An indicator matrix turns on multilabel classification.

Returns

self

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property multilabel_
Whether this is a multilabel classifier

partial_fit(*self*, *X*, *y*, *classes=None*)
Partially fit underlying estimators

Should be used when memory is inefficient to train all data. Chunks of data can be passed in several iteration.

Parameters

X [(sparse) array-like of shape (n_samples, n_features)] Data.

y [(sparse) array-like of shape (n_samples,) or (n_samples, n_classes)] Multi-class targets. An indicator matrix turns on multilabel classification.

classes [array, shape (n_classes,)] Classes across all calls to partial_fit. Can be obtained via `np.unique(y_all)`, where y_all is the target vector of the entire dataset. This argument is only required in the first call of partial_fit and can be omitted in the subsequent calls.

Returns

self

predict(*self*, *X*)
Predict multi-class targets using underlying estimators.

Parameters

X [(sparse) array-like of shape (n_samples, n_features)] Data.

Returns

y [(sparse) array-like of shape (n_samples,) or (n_samples, n_classes)] Predicted multi-class targets.

predict_proba(*self*, *X*)
Probability estimates.

The returned estimates for all classes are ordered by label of classes.

Note that in the multilabel case, each sample can have any number of labels. This returns the marginal probability that the given sample has the label in question. For example, it is entirely consistent that two labels both have a 90% probability of applying to a given sample.

In the single label multiclass case, the rows of the returned matrix sum to 1.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

T [(sparse) array-like of shape (n_samples, n_classes)] Returns the probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.multiclass.OneVsRestClassifier`

- *Multilabel classification*
- *Receiver Operating Characteristic (ROC)*
- *Precision-Recall*
- *Classifier Chain*

7.27.3 `sklearn.multiclass`.OneVsOneClassifier

class sklearn.multiclass.**OneVsOneClassifier** (*estimator*, *n_jobs=None*)

One-vs-one multiclass strategy

This strategy consists in fitting one classifier per class pair. At prediction time, the class which received the most votes is selected. Since it requires to fit `n_classes * (n_classes - 1) / 2` classifiers, this method is usually slower than one-vs-the-rest, due to its O(n_classes^2) complexity. However, this method may be advantageous for algorithms such as kernel algorithms which don't scale well with `n_samples`. This is because each individual learning problem only involves a small subset of the data whereas, with one-vs-the-rest, the complete dataset is used `n_classes` times.

Read more in the *User Guide*.

Parameters

estimator [estimator object] An estimator object implementing *fit* and one of *decision_function* or *predict_proba*.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

estimators_ [list of `n_classes * (n_classes - 1) / 2` estimators] Estimators used for predictions.

classes_ [numpy array of shape [n_classes]] Array containing labels.

n_classes_ [int] Number of classes

pairwise_indices_ [list, length = `len(estimators_)`, or `None`] Indices of samples used when training the estimators. `None` when `estimator` does not have `_pairwise` attribute.

Methods

decision_function(self, X)	Decision function for the OneVsOneClassifier.
fit(self, X, y)	Fit underlying estimators.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes])	Partially fit underlying estimators
predict(self, X)	Estimate the best class label for each sample in X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *estimator*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

decision_function (*self*, *X*)
Decision function for the OneVsOneClassifier.

The decision values for the samples are computed by adding the normalized sum of pair-wise classification confidence levels to the votes in order to disambiguate between the decision values when the votes for all the classes are equal leading to a tie.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

Y [array-like of shape (n_samples, n_classes)]

fit (*self*, *X*, *y*)
Fit underlying estimators.

Parameters

X [(sparse) array-like of shape (n_samples, n_features)] Data.

y [array-like of shape (n_samples,)] Multi-class targets.

Returns

self

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit (*self*, *X*, *y*, *classes=None*)
Partially fit underlying estimators

Should be used when memory is inefficient to train all data. Chunks of data can be passed in several iteration, where the first call should have an array of all target variables.

Parameters

X [(sparse) array-like of shape (n_samples, n_features)] Data.

y [array-like of shape (n_samples,)] Multi-class targets.

classes [array, shape (n_classes,)] Classes across all calls to partial_fit. Can be obtained via `np.unique(y_all)`, where y_all is the target vector of the entire dataset. This argument is only required in the first call of partial_fit and can be omitted in the subsequent calls.

Returns

self

predict (*self*, *X*)
Estimate the best class label for each sample in X.

This is implemented as `argmax(decision_function(X), axis=1)` which will return the label of the class with most votes by estimators predicting the outcome of a decision for each possible class pair.

Parameters

X [(sparse) array-like of shape (n_samples, n_features)] Data.

Returns

y [numpy array of shape [n_samples]] Predicted multi-class targets.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

7.27.4 sklearn.multiclass.OutputCodeClassifier

class sklearn.multiclass.**OutputCodeClassifier**(*estimator*, *code_size=1.5*, *random_state=None*, *n_jobs=None*)
(Error-Correcting) Output-Code multiclass strategy

Output-code based strategies consist in representing each class with a binary code (an array of 0s and 1s). At fitting time, one binary classifier per bit in the code book is fitted. At prediction time, the classifiers are used to project new points in the class space and the class closest to the points is chosen. The main advantage of these strategies is that the number of classifiers used can be controlled by the user, either for compressing the model (0 < code_size < 1) or for making the model more robust to errors (code_size > 1). See the documentation for more details.

Read more in the *User Guide*.

Parameters

estimator [estimator object] An estimator object implementing *fit* and one of *decision_function* or *predict_proba*.

code_size [float] Percentage of the number of classes to be used to create the code book. A number between 0 and 1 will require fewer classifiers than one-vs-the-rest. A number greater than 1 will require more classifiers than one-vs-the-rest.

random_state [int, RandomState instance or None, optional, default: None] The generator used to initialize the codebook. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

estimators_ [list of `int(n_classes * code_size)` estimators] Estimators used for predictions.

classes_ [numpy array of shape [n_classes]] Array containing labels.

code_book_ [numpy array of shape [n_classes, code_size]] Binary array containing the code of each class.

References

[R2eddaeec0849-1], [R2eddaeec0849-2], [R2eddaeec0849-3]

Examples

```
>>> from sklearn.multiclass import OutputCodeClassifier
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=100, n_features=4,
...                            n_informative=2, n_redundant=0,
...                            random_state=0, shuffle=False)
>>> clf = OutputCodeClassifier(
...     estimator=RandomForestClassifier(random_state=0),
...     random_state=0).fit(X, y)
>>> clf.predict([[0, 0, 0, 0]])
array([1])
```

Methods

fit(self, X, y)	Fit underlying estimators.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict multi-class targets using underlying estimators.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *estimator*, *code_size=1.5*, *random_state=None*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Fit underlying estimators.

Parameters

X [(sparse) array-like of shape (n_samples, n_features)] Data.

y [numpy array of shape [n_samples]] Multi-class targets.

Returns

self

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict multi-class targets using underlying estimators.

Parameters

X [(sparse) array-like of shape (n_samples, n_features)] Data.

Returns

y [numpy array of shape [n_samples]] Predicted multi-class targets.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

7.28 sklearn.multioutput: Multioutput regression and classification

This module implements multioutput regression and classification.

The estimators provided in this module are meta-estimators: they require a base estimator to be provided in their constructor. The meta-estimator extends single output estimators to multioutput estimators.

User guide: See the *Multiclass and multilabel algorithms* section for further details.

multioutput.ClassifierChain(base_estimator)	A multi-label model that arranges binary classifiers into a chain.
multioutput.MultiOutputRegressor(estimator)	Multi target regression
multioutput.MultiOutputClassifier(estimator)	Multi target classification
multioutput.RegressorChain(base_estimator[, ...])	A multi-label model that arranges regressions into a chain.

7.28.1 sklearn.multioutput.ClassifierChain

class sklearn.multioutput.**ClassifierChain**(*base_estimator*, *order=None*, *cv=None*, *random_state=None*)

A multi-label model that arranges binary classifiers into a chain.

Each model makes a prediction in the order specified by the chain using all of the available features provided to

the model plus the predictions of models that are earlier in the chain.

Read more in the *User Guide*.

New in version 0.19.

Parameters

base_estimator [estimator] The base estimator from which the classifier chain is built.

order [array-like of shape (n_outputs,) or 'random', optional] By default the order will be determined by the order of columns in the label matrix Y.:

```
order = [0, 1, 2, ..., Y.shape[1] - 1]
```

The order of the chain can be explicitly set by providing a list of integers. For example, for a chain of length 5.:

```
order = [1, 3, 2, 4, 0]
```

means that the first model in the chain will make predictions for column 1 in the Y matrix, the second model will make predictions for column 3, etc.

If order is 'random' a random ordering will be used.

cv [int, cross-validation generator or an iterable, optional (default=None)] Determines whether to use cross validated predictions or true labels for the results of previous estimators in the chain. If cv is None the true labels are used when fitting. Otherwise possible inputs for cv are:

- integer, to specify the number of folds in a (Stratified)KFold,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

The random number generator is used to generate random chain orders.

Attributes

classes_ [list] A list of arrays of length `len(estimators_)` containing the class labels for each estimator in the chain.

estimators_ [list] A list of clones of base_estimator.

order_ [list] The order of labels in the classifier chain.

See also:

RegressorChain Equivalent for regression

MultioutputClassifier Classifies each output independently rather than chaining.

References

Jesse Read, Bernhard Pfahringer, Geoff Holmes, Eibe Frank, "Classifier Chains for Multi-label Classification", 2009.

Methods

decision_function(self, X)	Evaluate the decision_function of the models in the chain.
fit(self, X, Y)	Fit the model to data matrix X and targets Y.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict on the data matrix X using the ClassifierChain model.
predict_proba(self, X)	Predict probability estimates.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *base_estimator*, *order=None*, *cv=None*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

decision_function (*self*, *X*)
Evaluate the decision_function of the models in the chain.

Parameters

X [array-like, shape (n_samples, n_features)]

Returns

Y_decision [array-like, shape (n_samples, n_classes)] Returns the decision function of the sample for each model in the chain.

fit (*self*, *X*, *Y*)
Fit the model to data matrix X and targets Y.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

Y [array-like, shape (n_samples, n_classes)] The target values.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict on the data matrix X using the ClassifierChain model.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

Returns

Y_pred [array-like, shape (n_samples, n_classes)] The predicted values.

predict_proba(*self*, *X*)
Predict probability estimates.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)]

Returns

Y_prob [array-like, shape (n_samples, n_classes)]

score(*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.multioutput.ClassifierChain`

- *Classifier Chain*

7.28.2 `sklearn.multioutput`.MultiOutputRegressor

class sklearn.multioutput.**MultiOutputRegressor**(*estimator*, *n_jobs=None*)
Multi target regression

This strategy consists of fitting one regressor per target. This is a simple strategy for extending regressors that do not natively support multi-target regression.

Parameters

estimator [estimator object] An estimator object implementing *fit* and *predict*.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for `fit`. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

When individual estimators are fast to train or predict using `n_jobs>1` can result in slower performance due to the overhead of spawning processes.

Attributes

estimators_ [list of `n_output` estimators] Estimators used for predictions.

Methods

fit(self, X, y[, sample_weight])	Fit the model to data.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, sample_weight])	Incrementally fit the model to data.
predict(self, X)	Predict multi-output variable using a model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *estimator*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit the model to data. Fit a separate model for each output variable.

Parameters

X [(sparse) array-like, shape (n_samples, n_features)] Data.

y [(sparse) array-like, shape (n_samples, n_outputs)] Multi-output targets. An indicator matrix turns on multilabel estimation.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Only supported if the underlying regressor supports sample weights.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit (*self*, *X*, *y*, *sample_weight=None*)
Incrementally fit the model to data. Fit a separate model for each output variable.

Parameters

X [(sparse) array-like, shape (n_samples, n_features)] Data.

y [(sparse) array-like, shape (n_samples, n_outputs)] Multi-output targets.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Only supported if the underlying regressor supports sample weights.

Returns

self [object]

predict (*self*, *X*)

Predict multi-output variable using a model trained for each target variable.

Parameters

X [(sparse) array-like, shape (n_samples, n_features)] Data.

Returns

y [(sparse) array-like, shape (n_samples, n_outputs)] Multi-output targets predicted across multiple predictors. Note: Separate models are generated for each predictor.

score (*self*, *X*, *y*, *sample_weight=None*)

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the regression sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like, shape (n_samples, n_features)] Test samples.

y [array-like, shape (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape [n_samples], optional] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

R^2 is calculated by weighting all the targets equally using `multioutput='uniform_average'`.

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.multioutput.MultiOutputRegressor`

- *Comparing random forests and the multi-output meta estimator*

7.28.3 `sklearn.multioutput`.MultiOutputClassifier

class sklearn.multioutput.**MultiOutputClassifier**(*estimator*, *n_jobs=None*)

Multi target classification

This strategy consists of fitting one classifier per target. This is a simple strategy for extending classifiers that do not natively support multi-target classification

Parameters

estimator [estimator object] An estimator object implementing *fit*, *score* and *predict_proba*.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. It does each target variable in y in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

estimators_ [list of `n_output` estimators] Estimators used for predictions.

Examples

```
>>> import numpy as np
>>> from sklearn.datasets import make_multilabel_classification
>>> from sklearn.multioutput import MultiOutputClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
```

```
>>> X, y = make_multilabel_classification(n_classes=3, random_state=0)
>>> clf = MultiOutputClassifier(KNeighborsClassifier()).fit(X, y)
>>> clf.predict(X[-2:])
array([[1, 1, 0], [1, 1, 1]])
```

Methods

fit(self, X, Y[, sample_weight])	Fit the model to data matrix X and targets Y.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Incrementally fit the model to data.
predict(self, X)	Predict multi-output variable using a model
score(self, X, y)	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *estimator*, *n_jobs=None*)

Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *Y*, *sample_weight=None*)

Fit the model to data matrix X and targets Y.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input data.

Y [array-like of shape (n_samples, n_classes)] The target values.

sample_weight [array-like of shape (n_samples,) or None] Sample weights. If None, then samples are equally weighted. Only supported if the underlying classifier supports sample weights.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit (*self*, *X*, *y*, *classes=None*, *sample_weight=None*)
Incrementally fit the model to data. Fit a separate model for each output variable.

Parameters

X [(sparse) array-like, shape (n_samples, n_features)] Data.

y [(sparse) array-like, shape (n_samples, n_outputs)] Multi-output targets.

classes [list of numpy arrays, shape (n_outputs)] Each array is unique classes for one output in str/int Can be obtained by via `[np.unique(y[:, i]) for i in range(y.shape[1])]`, where y is the target matrix of the entire dataset. This argument is required for the first call to partial_fit and can be omitted in the subsequent calls. Note that y doesn't need to contain all labels in `classes`.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Only supported if the underlying regressor supports sample weights.

Returns

self [object]

predict (*self*, *X*)

Predict multi-output variable using a model trained for each target variable.

Parameters

X [(sparse) array-like, shape (n_samples, n_features)] Data.

Returns

y [(sparse) array-like, shape (n_samples, n_outputs)] Multi-output targets predicted across multiple predictors. Note: Separate models are generated for each predictor.

property predict_proba
Probability estimates. Returns prediction probabilities for each class of each output.

This method will raise a `ValueError` if any of the estimators do not have `predict_proba`.

Parameters

X [array-like, shape (n_samples, n_features)] Data

Returns

p [array of shape (n_samples, n_classes), or a list of n_outputs such arrays if n_outputs > 1.] The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

score(*self*, *X*, *y*)

Returns the mean accuracy on the given test data and labels.

Parameters

X [array-like, shape [n_samples, n_features]] Test samples

y [array-like, shape [n_samples, n_outputs]] True values for X

Returns

scores [float] accuracy_score of self.predict(X) versus y

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

7.28.4 sklearn.multioutput.RegressorChain

class sklearn.multioutput.**RegressorChain**(*base_estimator*, *order=None*, *cv=None*, *random_state=None*)

A multi-label model that arranges regressions into a chain.

Each model makes a prediction in the order specified by the chain using all of the available features provided to the model plus the predictions of models that are earlier in the chain.

Read more in the *User Guide*.

Parameters

base_estimator [estimator] The base estimator from which the classifier chain is built.

order [array-like of shape (n_outputs,) or 'random', optional] By default the order will be determined by the order of columns in the label matrix Y.:

```
order = [0, 1, 2, ..., Y.shape[1] - 1]
```

The order of the chain can be explicitly set by providing a list of integers. For example, for a chain of length 5.:

```
order = [1, 3, 2, 4, 0]
```

means that the first model in the chain will make predictions for column 1 in the Y matrix, the second model will make predictions for column 3, etc.

If order is 'random' a random ordering will be used.

cv [int, cross-validation generator or an iterable, optional (default=None)] Determines whether to use cross validated predictions or true labels for the results of previous estimators in the chain. If cv is None the true labels are used when fitting. Otherwise possible inputs for cv are:

- integer, to specify the number of folds in a (Stratified)KFold,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

The random number generator is used to generate random chain orders.

Attributes

estimators_ [list] A list of clones of base_estimator.

order_ [list] The order of labels in the classifier chain.

See also:

ClassifierChain Equivalent for classification

MultioutputRegressor Learns each output independently rather than chaining.

Methods

fit(self, X, Y)	Fit the model to data matrix X and targets Y.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict on the data matrix X using the ClassifierChain model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *base_estimator*, *order=None*, *cv=None*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *Y*)
Fit the model to data matrix X and targets Y.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

Y [array-like, shape (n_samples, n_classes)] The target values.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict on the data matrix X using the ClassifierChain model.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

Returns

Y_pred [array-like, shape (n_samples, n_classes)] The predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

7.29 sklearn.naive_bayes: Naive Bayes

The *sklearn.naive_bayes* module implements Naive Bayes algorithms. These are supervised learning methods based on applying Bayes' theorem with strong (naive) feature independence assumptions.

User guide: See the *Naive Bayes* section for further details.

naive_bayes.BernoulliNB([alpha, binarize, ...])	Naive Bayes classifier for multivariate Bernoulli models.
naive_bayes.CategoricalNB([alpha, ...])	Naive Bayes classifier for categorical features
naive_bayes.ComplementNB([alpha, fit_prior, ...])	The Complement Naive Bayes classifier described in Rennie et al.
naive_bayes.GaussianNB([priors, var_smoothing])	Gaussian Naive Bayes (GaussianNB)
naive_bayes.MultinomialNB([alpha, ...])	Naive Bayes classifier for multinomial models

7.29.1 sklearn.naive_bayes.BernoulliNB

class sklearn.naive_bayes.**BernoulliNB**(*alpha=1.0*, *binarize=0.0*, *fit_prior=True*, *class_prior=None*)

Naive Bayes classifier for multivariate Bernoulli models.

Like MultinomialNB, this classifier is suitable for discrete data. The difference is that while MultinomialNB works with occurrence counts, BernoulliNB is designed for binary/boolean features.

Read more in the *User Guide*.

Parameters

alpha [float, optional (default=1.0)] Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

binarize [float or None, optional (default=0.0)] Threshold for binarizing (mapping to booleans) of sample features. If None, input is presumed to already consist of binary vectors.

fit_prior [bool, optional (default=True)] Whether to learn class prior probabilities or not. If false, a uniform prior will be used.

class_prior [array-like, size=[n_classes,], optional (default=None)] Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

Attributes

class_count_ [array, shape = [n_classes]] Number of samples encountered for each class during fitting. This value is weighted by the sample weight when provided.

class_log_prior_ [array, shape = [n_classes]] Log probability of each class (smoothed).

classes_ [array, shape (n_classes,)] Class labels known to the classifier

feature_count_ [array, shape = [n_classes, n_features]] Number of samples encountered for each (class, feature) during fitting. This value is weighted by the sample weight when provided.

feature_log_prob_ [array, shape = [n_classes, n_features]] Empirical log probability of features given a class, P(x_i|y).

n_features_ [int] Number of features of each sample.

References

C.D. Manning, P. Raghavan and H. Schuetze (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 234-265. https://nlp.stanford.edu/IR-book/html/htmledition/the-bernoulli-model-1.html

A. McCallum and K. Nigam (1998). A comparison of event models for naive Bayes text classification. Proc. AAAI/ICML-98 Workshop on Learning for Text Categorization, pp. 41-48.

V. Metsis, I. Androutsopoulos and G. Paliouras (2006). Spam filtering with naive Bayes – Which naive Bayes? 3rd Conf. on Email and Anti-Spam (CEAS).

Examples

```
>>> import numpy as np
>>> rng = np.random.RandomState(1)
>>> X = rng.randint(5, size=(6, 100))
>>> Y = np.array([1, 2, 3, 4, 4, 5])
>>> from sklearn.naive_bayes import BernoulliNB
>>> clf = BernoulliNB()
>>> clf.fit(X, Y)
BernoulliNB()
>>> print(clf.predict(X[2:3]))
[3]
```

Methods

fit(self, X, y[, sample_weight])	Fit Naive Bayes classifier according to X, y
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Incremental fit on a batch of samples.
predict(self, X)	Perform classification on an array of test vectors X.
predict_log_proba(self, X)	Return log-probability estimates for the test vector X.
predict_proba(self, X)	Return probability estimates for the test vector X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *binarize=0.0*, *fit_prior=True*, *class_prior=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit Naive Bayes classifier according to X, y

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like of shape (n_samples,)] Target values.

sample_weight [array-like of shape (n_samples,), default=None] Weights applied to individual samples (1. for unweighted).

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit(*self*, *X*, *y*, *classes=None*, *sample_weight=None*)
Incremental fit on a batch of samples.

This method is expected to be called several times consecutively on different chunks of a dataset so as to implement out-of-core or online learning.

This is especially useful when the whole dataset is too big to fit in memory at once.

This method has some performance overhead hence it is better to call partial_fit on chunks of data that are as large as possible (as long as fitting in the memory budget) to hide the overhead.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like of shape (n_samples,)] Target values.

classes [array-like of shape (n_classes) (default=None)] List of all the classes that can possibly appear in the y vector.

Must be provided at the first call to partial_fit, can be omitted in subsequent calls.

sample_weight [array-like of shape (n_samples,), default=None] Weights applied to individual samples (1. for unweighted).

Returns

self [object]

predict(*self*, *X*)
Perform classification on an array of test vectors X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [ndarray of shape (n_samples,)] Predicted target values for X

predict_log_proba(*self*, *X*)
Return log-probability estimates for the test vector X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [array-like of shape (n_samples, n_classes)] Returns the log-probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

predict_proba(*self*, *X*)
Return probability estimates for the test vector X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [array-like of shape (n_samples, n_classes)] Returns the probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.naive_bayes.BernoulliNB`

- *Hashing feature transformation using Totally Random Trees*
- *Classification of text documents using sparse features*

7.29.2 `sklearn.naive_bayes`.CategoricalNB

class sklearn.naive_bayes.**CategoricalNB**(*alpha=1.0*, *fit_prior=True*, *class_prior=None*)
Naive Bayes classifier for categorical features

The categorical Naive Bayes classifier is suitable for classification with discrete features that are categorically distributed. The categories of each feature are drawn from a categorical distribution.

Read more in the *User Guide*.

Parameters

alpha [float, optional (default=1.0)] Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

fit_prior [boolean, optional (default=True)] Whether to learn class prior probabilities or not. If false, a uniform prior will be used.

class_prior [array-like, size (n_classes,), optional (default=None)] Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

Attributes

category_count_ [list of arrays, len n_features] Holds arrays of shape (n_classes, n_categories of respective feature) for each feature. Each array provides the number of samples encountered for each class and category of the specific feature.

class_count_ [array, shape (n_classes,)] Number of samples encountered for each class during fitting. This value is weighted by the sample weight when provided.

class_log_prior_ [array, shape (n_classes,)] Smoothed empirical log probability for each class.

classes_ [array, shape (n_classes,)] Class labels known to the classifier

feature_log_prob_ [list of arrays, len n_features] Holds arrays of shape (n_classes, n_categories of respective feature) for each feature. Each array provides the empirical log probability of categories given the respective feature and class, `P(x_i|y)`.

n_features_ [int] Number of features of each sample.

Examples

```
>>> import numpy as np
>>> rng = np.random.RandomState(1)
>>> X = rng.randint(5, size=(6, 100))
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> from sklearn.naive_bayes import CategoricalNB
>>> clf = CategoricalNB()
>>> clf.fit(X, y)
CategoricalNB()
>>> print(clf.predict(X[2:3]))
[3]
```

Methods

fit(self, X, y[, sample_weight])	Fit Naive Bayes classifier according to X, y
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Incremental fit on a batch of samples.
predict(self, X)	Perform classification on an array of test vectors X.
predict_log_proba(self, X)	Return log-probability estimates for the test vector X.

Continued on next page

Table 235 – continued from previous page

predict_proba(self, X)	Return probability estimates for the test vector X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *alpha=1.0*, *fit_prior=True*, *class_prior=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit Naive Bayes classifier according to X, y

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of features. Here, each feature of X is assumed to be from a different categorical distribution. It is further assumed that all categories of each feature are represented by the numbers 0, ..., n - 1, where n refers to the total number of categories for the given feature. This can, for instance, be achieved with the help of OrdinalEncoder.

y [array-like, shape = [n_samples]] Target values.

sample_weight [array-like, shape = [n_samples], (default=None)] Weights applied to individual samples (1. for unweighted).

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit (*self*, *X*, *y*, *classes=None*, *sample_weight=None*)
Incremental fit on a batch of samples.

This method is expected to be called several times consecutively on different chunks of a dataset so as to implement out-of-core or online learning.

This is especially useful when the whole dataset is too big to fit in memory at once.

This method has some performance overhead hence it is better to call partial_fit on chunks of data that are as large as possible (as long as fitting in the memory budget) to hide the overhead.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of features. Here, each feature of X is assumed to be from a different categorical distribution. It is further assumed that all categories of each feature are represented by the numbers 0, ..., n - 1, where n refers to the total number of categories for the given feature. This can, for instance, be achieved with the help of OrdinalEncoder.

y [array-like, shape = [n_samples]] Target values.

classes [array-like, shape = [n_classes] (default=None)] List of all the classes that can possibly appear in the y vector.

Must be provided at the first call to partial_fit, can be omitted in subsequent calls.

sample_weight [array-like, shape = [n_samples], (default=None)] Weights applied to individual samples (1. for unweighted).

Returns

self [object]

predict (*self*, *X*)

Perform classification on an array of test vectors X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [ndarray of shape (n_samples,)] Predicted target values for X

predict_log_proba (*self*, *X*)

Return log-probability estimates for the test vector X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [array-like of shape (n_samples, n_classes)] Returns the log-probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

predict_proba (*self*, *X*)

Return probability estimates for the test vector X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [array-like of shape (n_samples, n_classes)] Returns the probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

7.29.3 sklearn.naive_bayes.ComplementNB

class sklearn.naive_bayes.**ComplementNB**(*alpha=1.0*, *fit_prior=True*, *class_prior=None*, *norm=False*)
The Complement Naive Bayes classifier described in Rennie et al. (2003).

The Complement Naive Bayes classifier was designed to correct the "severe assumptions" made by the standard Multinomial Naive Bayes classifier. It is particularly suited for imbalanced data sets.

Read more in the *User Guide*.

Parameters

alpha [float, optional (default=1.0)] Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

fit_prior [boolean, optional (default=True)] Only used in edge case with a single class in the training set.

class_prior [array-like, size (n_classes,), optional (default=None)] Prior probabilities of the classes. Not used.

norm [boolean, optional (default=False)] Whether or not a second normalization of the weights is performed. The default behavior mirrors the implementations found in Mahout and Weka, which do not follow the full algorithm described in Table 9 of the paper.

Attributes

class_count_ [array, shape (n_classes,)] Number of samples encountered for each class during fitting. This value is weighted by the sample weight when provided.

class_log_prior_ [array, shape (n_classes,)] Smoothed empirical log probability for each class. Only used in edge case with a single class in the training set.

classes_ [array, shape (n_classes,)] Class labels known to the classifier

feature_all_ [array, shape (n_features,)] Number of samples encountered for each feature during fitting. This value is weighted by the sample weight when provided.

feature_count_ [array, shape (n_classes, n_features)] Number of samples encountered for each (class, feature) during fitting. This value is weighted by the sample weight when provided.

feature_log_prob_ [array, shape (n_classes, n_features)] Empirical weights for class complements.

n_features_ [int] Number of features of each sample.

References

Rennie, J. D., Shih, L., Teevan, J., & Karger, D. R. (2003). Tackling the poor assumptions of naive bayes text classifiers. In ICML (Vol. 3, pp. 616-623). https://people.csail.mit.edu/jrennie/papers/icml03-nb.pdf

Examples

```
>>> import numpy as np
>>> rng = np.random.RandomState(1)
>>> X = rng.randint(5, size=(6, 100))
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> from sklearn.naive_bayes import ComplementNB
>>> clf = ComplementNB()
>>> clf.fit(X, y)
ComplementNB()
>>> print(clf.predict(X[2:3]))
[3]
```

Methods

fit(self, X, y[, sample_weight])	Fit Naive Bayes classifier according to X, y
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Incremental fit on a batch of samples.
predict(self, X)	Perform classification on an array of test vectors X.
predict_log_proba(self, X)	Return log-probability estimates for the test vector X.
predict_proba(self, X)	Return probability estimates for the test vector X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *fit_prior=True*, *class_prior=None*, *norm=False*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit Naive Bayes classifier according to X, y

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like of shape (n_samples,)] Target values.

sample_weight [array-like of shape (n_samples,), default=None] Weights applied to individual samples (1. for unweighted).

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit (*self*, *X*, *y*, *classes=None*, *sample_weight=None*)

Incremental fit on a batch of samples.

This method is expected to be called several times consecutively on different chunks of a dataset so as to implement out-of-core or online learning.

This is especially useful when the whole dataset is too big to fit in memory at once.

This method has some performance overhead hence it is better to call partial_fit on chunks of data that are as large as possible (as long as fitting in the memory budget) to hide the overhead.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like of shape (n_samples,)] Target values.

classes [array-like of shape (n_classes) (default=None)] List of all the classes that can possibly appear in the y vector.

Must be provided at the first call to partial_fit, can be omitted in subsequent calls.

sample_weight [array-like of shape (n_samples,), default=None] Weights applied to individual samples (1. for unweighted).

Returns

self [object]

predict (*self*, *X*)

Perform classification on an array of test vectors X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [ndarray of shape (n_samples,)] Predicted target values for X

predict_log_proba (*self*, *X*)

Return log-probability estimates for the test vector X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [array-like of shape (n_samples, n_classes)] Returns the log-probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

predict_proba (*self*, *X*)

Return probability estimates for the test vector X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [array-like of shape (n_samples, n_classes)] Returns the probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.naive_bayes.ComplementNB`

- *Classification of text documents using sparse features*

7.29.4 `sklearn.naive_bayes`.GaussianNB

class sklearn.naive_bayes.**GaussianNB** (*priors=None*, *var_smoothing=1e-09*)

Gaussian Naive Bayes (GaussianNB)

Can perform online updates to model parameters via `partial_fit`. For details on algorithm used to update feature means and variance online, see Stanford CS tech report STAN-CS-79-773 by Chan, Golub, and LeVeque:

http://i.stanford.edu/pub/cstr/reports/cs/tr/79/773/CS-TR-79-773.pdf

Read more in the *User Guide*.

Parameters

priors [array-like, shape (n_classes,)] Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

var_smoothing [float, optional (default=1e-9)] Portion of the largest variance of all features that is added to variances for calculation stability.

Attributes

class_count_ [array, shape (n_classes,)] number of training samples observed in each class.

class_prior_ [array, shape (n_classes,)] probability of each class.

classes_ [array, shape (n_classes,)] class labels known to the classifier

epsilon_ [float] absolute additive value to variances

sigma_ [array, shape (n_classes, n_features)] variance of each feature per class

theta_ [array, shape (n_classes, n_features)] mean of each feature per class

Examples

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> Y = np.array([1, 1, 1, 2, 2, 2])
>>> from sklearn.naive_bayes import GaussianNB
>>> clf = GaussianNB()
>>> clf.fit(X, Y)
GaussianNB()
>>> print(clf.predict([[-0.8, -1]]))
[1]
>>> clf_pf = GaussianNB()
>>> clf_pf.partial_fit(X, Y, np.unique(Y))
GaussianNB()
>>> print(clf_pf.predict([[-0.8, -1]]))
[1]
```

Methods

fit(self, X, y[, sample_weight])	Fit Gaussian Naive Bayes according to X, y
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Incremental fit on a batch of samples.
predict(self, X)	Perform classification on an array of test vectors X.
predict_log_proba(self, X)	Return log-probability estimates for the test vector X.
predict_proba(self, X)	Return probability estimates for the test vector X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *priors=None*, *var_smoothing=1e-09*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit Gaussian Naive Bayes according to X, y

Parameters

X [array-like, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape (n_samples,)] Target values.

sample_weight [array-like, shape (n_samples,), optional (default=None)] Weights applied to individual samples (1. for unweighted).

New in version 0.17: Gaussian Naive Bayes supports fitting with *sample_weight*.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit(*self*, *X*, *y*, *classes=None*, *sample_weight=None*)
Incremental fit on a batch of samples.

This method is expected to be called several times consecutively on different chunks of a dataset so as to implement out-of-core or online learning.

This is especially useful when the whole dataset is too big to fit in memory at once.

This method has some performance and numerical stability overhead, hence it is better to call partial_fit on chunks of data that are as large as possible (as long as fitting in the memory budget) to hide the overhead.

Parameters

X [array-like, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape (n_samples,)] Target values.

classes [array-like, shape (n_classes,), optional (default=None)] List of all the classes that can possibly appear in the y vector.

Must be provided at the first call to partial_fit, can be omitted in subsequent calls.

sample_weight [array-like, shape (n_samples,), optional (default=None)] Weights applied to individual samples (1. for unweighted).

New in version 0.17.

Returns

self [object]

predict(*self*, *X*)
Perform classification on an array of test vectors X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [ndarray of shape (n_samples,)] Predicted target values for X

predict_log_proba(*self*, *X*)
Return log-probability estimates for the test vector X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [array-like of shape (n_samples, n_classes)] Returns the log-probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

predict_proba(*self*, *X*)

Return probability estimates for the test vector X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [array-like of shape (n_samples, n_classes)] Returns the probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.naive_bayes.GaussianNB`

- *Comparison of Calibration of Classifiers*
- *Probability Calibration curves*
- *Probability calibration of classifiers*
- *Classifier comparison*

- *Plot class probabilities calculated by the VotingClassifier*
- *Plotting Learning Curves*
- *Importance of Feature Scaling*

7.29.5 sklearn.naive_bayes.MultinomialNB

class sklearn.naive_bayes.**MultinomialNB**(*alpha=1.0*, *fit_prior=True*, *class_prior=None*)

Naive Bayes classifier for multinomial models

The multinomial Naive Bayes classifier is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf may also work.

Read more in the *User Guide*.

Parameters

alpha [float, optional (default=1.0)] Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

fit_prior [boolean, optional (default=True)] Whether to learn class prior probabilities or not. If false, a uniform prior will be used.

class_prior [array-like, size (n_classes,), optional (default=None)] Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

Attributes

class_count_ [array, shape (n_classes,)] Number of samples encountered for each class during fitting. This value is weighted by the sample weight when provided.

class_log_prior_ [array, shape (n_classes,)] Smoothed empirical log probability for each class.

classes_ [array, shape (n_classes,)] Class labels known to the classifier

coef_ [array, shape (n_classes, n_features)] Mirrors `feature_log_prob_` for interpreting MultinomialNB as a linear model.

feature_count_ [array, shape (n_classes, n_features)] Number of samples encountered for each (class, feature) during fitting. This value is weighted by the sample weight when provided.

feature_log_prob_ [array, shape (n_classes, n_features)] Empirical log probability of features given a class, `P(x_i|y)`.

intercept_ [array, shape (n_classes,)] Mirrors `class_log_prior_` for interpreting MultinomialNB as a linear model.

n_features_ [int] Number of features of each sample.

Notes

For the rationale behind the names `coef_` and `intercept_`, i.e. naive Bayes as a linear classifier, see J. Rennie et al. (2003), Tackling the poor assumptions of naive Bayes text classifiers, ICML.

References

C.D. Manning, P. Raghavan and H. Schuetze (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 234-265. https://nlp.stanford.edu/IR-book/html/htmledition/naive-bayes-text-classification-1.html

Examples

```
>>> import numpy as np
>>> rng = np.random.RandomState(1)
>>> X = rng.randint(5, size=(6, 100))
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> from sklearn.naive_bayes import MultinomialNB
>>> clf = MultinomialNB()
>>> clf.fit(X, y)
MultinomialNB()
>>> print(clf.predict(X[2:3]))
[3]
```

Methods

fit(self, X, y[, sample_weight])	Fit Naive Bayes classifier according to X, y
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Incremental fit on a batch of samples.
predict(self, X)	Perform classification on an array of test vectors X.
predict_log_proba(self, X)	Return log-probability estimates for the test vector X.
predict_proba(self, X)	Return probability estimates for the test vector X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *alpha=1.0*, *fit_prior=True*, *class_prior=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit Naive Bayes classifier according to X, y

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like of shape (n_samples,)] Target values.

sample_weight [array-like of shape (n_samples,), default=None] Weights applied to individual samples (1. for unweighted).

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit(*self*, *X*, *y*, *classes=None*, *sample_weight=None*)

Incremental fit on a batch of samples.

This method is expected to be called several times consecutively on different chunks of a dataset so as to implement out-of-core or online learning.

This is especially useful when the whole dataset is too big to fit in memory at once.

This method has some performance overhead hence it is better to call partial_fit on chunks of data that are as large as possible (as long as fitting in the memory budget) to hide the overhead.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like of shape (n_samples,)] Target values.

classes [array-like of shape (n_classes) (default=None)] List of all the classes that can possibly appear in the y vector.

Must be provided at the first call to partial_fit, can be omitted in subsequent calls.

sample_weight [array-like of shape (n_samples,), default=None] Weights applied to individual samples (1. for unweighted).

Returns

self [object]

predict(*self*, *X*)

Perform classification on an array of test vectors X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [ndarray of shape (n_samples,)] Predicted target values for X

predict_log_proba(*self*, *X*)

Return log-probability estimates for the test vector X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [array-like of shape (n_samples, n_classes)] Returns the log-probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

predict_proba(*self*, *X*)

Return probability estimates for the test vector X.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [array-like of shape (n_samples, n_classes)] Returns the probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.naive_bayes.MultinomialNB`

- *Out-of-core classification of text documents*
- *Classification of text documents using sparse features*

7.30 `sklearn.neighbors`: Nearest Neighbors

The `sklearn.neighbors` module implements the k-nearest neighbors algorithm.

User guide: See the *Nearest Neighbors* section for further details.

`neighbors.BallTree`	BallTree for fast generalized N-point problems
`neighbors.DistanceMetric`	DistanceMetric class
`neighbors.KDTree`	KDTree for fast generalized N-point problems
`neighbors.KernelDensity`([bandwidth, ...])	Kernel Density Estimation.
`neighbors.KNeighborsClassifier`([...])	Classifier implementing the k-nearest neighbors vote.

Continued on next page

Table 239 – continued from previous page

neighbors.KNeighborsRegressor([n_neighbors, ...])	Regression based on k-nearest neighbors.
neighbors.KNeighborsTransformer([mode, ...])	Transform X into a (weighted) graph of k nearest neighbors
neighbors.LocalOutlierFactor([n_neighbors, ...])	Unsupervised Outlier Detection using Local Outlier Factor (LOF)
neighbors.RadiusNeighborsClassifier([...])	Classifier implementing a vote among neighbors within a given radius
neighbors.RadiusNeighborsRegressor([radius, ...])	Regression based on neighbors within a fixed radius.
neighbors.RadiusNeighborsTransformer([mode, ...])	Transform X into a (weighted) graph of neighbors nearer than a radius
neighbors.NearestCentroid([metric, ...])	Nearest centroid classifier.
neighbors.NearestNeighbors([n_neighbors, ...])	Unsupervised learner for implementing neighbor searches.
neighbors.NeighborhoodComponentsAnalysis([...])	Neighborhood Components Analysis

7.30.1 sklearn.neighbors.BallTree

class sklearn.neighbors.**BallTree**

BallTree for fast generalized N-point problems

BallTree(X, leaf_size=40, metric='minkowski', **kwargs)

Parameters

X [array-like of shape (n_samples, n_features)] n_samples is the number of points in the data set, and n_features is the dimension of the parameter space. Note: if X is a C-contiguous array of doubles then data will not be copied. Otherwise, an internal copy will be made.

leaf_size [positive integer (default = 40)] Number of points at which to switch to brute-force. Changing leaf_size will not affect the results of a query, but can significantly impact the speed of a query and the memory required to store the constructed tree. The amount of memory needed to store the tree scales as approximately n_samples / leaf_size. For a specified `leaf_size`, a leaf node is guaranteed to satisfy `leaf_size <= n_points <= 2 * leaf_size`, except in the case that `n_samples < leaf_size`.

metric [string or DistanceMetric object] the distance metric to use for the tree. Default='minkowski' with p=2 (that is, a euclidean metric). See the documentation of the DistanceMetric class for a list of available metrics. ball_tree.valid_metrics gives a list of the metrics which are valid for BallTree.

Additional keywords are passed to the distance metric class.

Note: Callable functions in the metric parameter are NOT supported for KDTree

and Ball Tree. Function call overhead will result in very poor performance.

Attributes

data [memory view] The training data

Examples

Query for k-nearest neighbors

```
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((10, 3))  # 10 points in 3 dimensions
>>> tree = BallTree(X, leaf_size=2)              # doctest: +SKIP
>>> dist, ind = tree.query(X[:1], k=3)                # doctest: +SKIP
>>> print(ind)  # indices of 3 closest neighbors
[0 3 1]
>>> print(dist)  # distances to 3 closest neighbors
[ 0.          0.19662693  0.29473397]
```

Pickle and Unpickle a tree. Note that the state of the tree is saved in the pickle operation: the tree needs not be rebuilt upon unpickling.

```
>>> import numpy as np
>>> import pickle
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((10, 3))  # 10 points in 3 dimensions
>>> tree = BallTree(X, leaf_size=2)        # doctest: +SKIP
>>> s = pickle.dumps(tree)                     # doctest: +SKIP
>>> tree_copy = pickle.loads(s)                # doctest: +SKIP
>>> dist, ind = tree_copy.query(X[:1], k=3)     # doctest: +SKIP
>>> print(ind)  # indices of 3 closest neighbors
[0 3 1]
>>> print(dist)  # distances to 3 closest neighbors
[ 0.          0.19662693  0.29473397]
```

Query for neighbors within a given radius

```
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((10, 3))  # 10 points in 3 dimensions
>>> tree = BallTree(X, leaf_size=2)     # doctest: +SKIP
>>> print(tree.query_radius(X[:1], r=0.3, count_only=True))
3
>>> ind = tree.query_radius(X[:1], r=0.3)  # doctest: +SKIP
>>> print(ind)  # indices of neighbors within distance 0.3
[3 0 1]
```

Compute a gaussian kernel density estimate:

```
>>> import numpy as np
>>> rng = np.random.RandomState(42)
>>> X = rng.random_sample((100, 3))
>>> tree = BallTree(X)                # doctest: +SKIP
>>> tree.kernel_density(X[:3], h=0.1, kernel='gaussian')
array([ 6.94114649,  7.83281226,  7.2071716 ])
```

Compute a two-point auto-correlation function

```
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((30, 3))
>>> r = np.linspace(0, 1, 5)
>>> tree = BallTree(X)                # doctest: +SKIP
>>> tree.two_point_correlation(X, r)
array([ 30,  62, 278, 580, 820])
```

Methods

kernel_density(self, X, h[, kernel, atol, ...])	Compute the kernel density estimate at points X with the given kernel, using the distance metric specified at tree creation.
query(X[, k, return_distance, dualtree, ...])	query the tree for the k nearest neighbors
query_radius()	query_radius(self, X, r, count_only = False):
two_point_correlation()	Compute the two-point correlation function

get_arrays	
get_n_calls	
get_tree_stats	
reset_n_calls	

__init__ (*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

kernel_density (*self*, *X*, *h*, *kernel='gaussian'*, *atol=0*, *rtol=1E-8*, *breadth_first=True*, *return_log=False*)
Compute the kernel density estimate at points X with the given kernel, using the distance metric specified at tree creation.

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query. Last dimension should match dimension of training data.

h [float] the bandwidth of the kernel

kernel [string] specify the kernel to use. Options are - 'gaussian' - 'tophat' - 'epanechnikov' - 'exponential' - 'linear' - 'cosine' Default is kernel = 'gaussian'

atol, rtol [float (default = 0)] Specify the desired relative and absolute tolerance of the result. If the true result is K_true, then the returned result K_ret satisfies `abs(K_true - K_ret) < atol + rtol * K_ret` The default is zero (i.e. machine precision) for both.

breadth_first [boolean (default = False)] if True, use a breadth-first search. If False (default) use a depth-first search. Breadth-first is generally faster for compact kernels and/or high tolerances.

return_log [boolean (default = False)] return the logarithm of the result. This can be more accurate than returning the result itself for narrow kernels.

Returns

density [ndarray] The array of (log)-density evaluations, shape = X.shape[:-1]

query (*X*, *k=1*, *return_distance=True*, *dualtree=False*, *breadth_first=False*)
query the tree for the k nearest neighbors

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query

k [integer (default = 1)] The number of nearest neighbors to return

return_distance [boolean (default = True)] if True, return a tuple (d, i) of distances and indices if False, return array i

dualtree [boolean (default = False)] if True, use the dual tree formalism for the query: a tree is built for the query points, and the pair of trees is used to efficiently search this space. This can lead to better performance as the number of points grows large.

breadth_first [boolean (default = False)] if True, then query the nodes in a breadth-first manner. Otherwise, query the nodes in a depth-first manner.

sort_results [boolean (default = True)] if True, then distances and indices of each point are sorted on return, so that the first column contains the closest points. Otherwise, neighbors are returned in an arbitrary order.

Returns

i [if return_distance == False]

(d,i) [if return_distance == True]

d [array of doubles - shape: x.shape[:-1] + (k,)] each entry gives the list of distances to the neighbors of the corresponding point

i [array of integers - shape: x.shape[:-1] + (k,)] each entry gives the list of indices of neighbors of the corresponding point

query_radius()

query_radius(self, X, r, count_only = False):

query the tree for neighbors within a radius r

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query

r [distance within which neighbors are returned] r can be a single value, or an array of values of shape x.shape[:-1] if different radii are desired for each point.

return_distance [boolean (default = False)] if True, return distances to neighbors of each point if False, return only neighbors Note that unlike the query() method, setting return_distance=True here adds to the computation time. Not all distances need to be calculated explicitly for return_distance=False. Results are not sorted by default: see `sort_results` keyword.

count_only [boolean (default = False)] if True, return only the count of points within distance r if False, return the indices of all points within distance r If return_distance==True, setting count_only=True will result in an error.

sort_results [boolean (default = False)] if True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. If return_distance == False, setting sort_results = True will result in an error.

Returns

count [if count_only == True]

ind [if count_only == False and return_distance == False]

(ind, dist) [if count_only == False and return_distance == True]

count [array of integers, shape = X.shape[:-1]] each entry gives the number of neighbors within a distance r of the corresponding point.

ind [array of objects, shape = X.shape[:-1]] each element is a numpy integer array listing the indices of neighbors of the corresponding point. Note that unlike the results of a k-neighbors query, the returned neighbors are not sorted by distance by default.

dist [array of objects, shape = X.shape[:-1]] each element is a numpy double array listing the distances corresponding to indices in i.

two_point_correlation()
Compute the two-point correlation function

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query. Last dimension should match dimension of training data.

r [array_like] A one-dimensional array of distances

dualtree [boolean (default = False)] If true, use a dualtree algorithm. Otherwise, use a single-tree algorithm. Dual tree algorithms can have better scaling for large N.

Returns

counts [ndarray] counts[i] contains the number of pairs of points with distance less than or equal to r[i]

7.30.2 sklearn.neighbors.DistanceMetric

class sklearn.neighbors.**DistanceMetric**
DistanceMetric class

This class provides a uniform interface to fast distance metric functions. The various metrics can be accessed via the *get_metric* class method and the metric string identifier (see below). For example, to use the Euclidean distance:

```
>>> dist = DistanceMetric.get_metric('euclidean')
>>> X = [[0, 1, 2],
         [3, 4, 5]]
>>> dist.pairwise(X)
array([[ 0.        ,  5.19615242],
       [ 5.19615242,  0.        ]])
```

Available Metrics

The following lists the string metric identifiers and the associated distance metric classes:

Metrics intended for real-valued vector spaces:

identifier	class name	args	distance function
“euclidean”	EuclideanDistance	•	`sqrt(sum((x - y)^2))`
“manhattan”	ManhattanDistance	•	`sum(|x - y|)`
“chebyshev”	ChebyshevDistance	•	`max(|x - y|)`
“minkowski”	MinkowskiDistance	p	`sum(|x - y|^p)^(1/p)`
“wminkowski”	WMinkowskiDistance	p, w	`sum(|w * (x - y)|^p)^(1/p)`
“seuclidean”	SEuclideanDistance	V	`sqrt(sum((x - y)^2 / V))`
“mahalanobis”	MahalanobisDistance	V or VI	`sqrt((x - y)' V^-1 (x - y))`

Metrics intended for two-dimensional vector spaces: Note that the haversine distance metric requires data in the form of [latitude, longitude] and both inputs and outputs are in units of radians.

identifier	class name	distance function
“haversine”	HaversineDistance	`2 arcsin(sqrt(sin^2(0.5*dx) + cos(x1)cos(x2)sin^2(0.5*dy)))`

Metrics intended for integer-valued vector spaces: Though intended for integer-valued vectors, these are also valid metrics in the case of real-valued vectors.

identifier	class name	distance function
“hamming”	HammingDistance	`N_unequal(x, y) / N_tot`
“canberra”	CanberraDistance	`sum(|x - y| / (|x| + |y|))`
“braycurtis”	BrayCurtisDistance	`sum(|x - y|) / (sum(|x|) + sum(|y|))`

Metrics intended for boolean-valued vector spaces: Any nonzero entry is evaluated to “True”. In the listings below, the following abbreviations are used:

- N : number of dimensions
- NTT : number of dims in which both values are True
- NTF : number of dims in which the first value is True, second is False
- NFT : number of dims in which the first value is False, second is True
- NFF : number of dims in which both values are False
- NNEQ : number of non-equal dimensions, NNEQ = NTF + NFT
- NNZ : number of nonzero dimensions, NNZ = NTF + NFT + NTT

identifier	class name	distance function
“jaccard”	JaccardDistance	NNEQ / NNZ
“matching”	MatchingDistance	NNEQ / N
“dice”	DiceDistance	NNEQ / (NTT + NNZ)
“kulsinski”	KulsinskiDistance	(NNEQ + N - NTT) / (NNEQ + N)
“rogerstanimoto”	RogersTanimotoDistance	2 * NNEQ / (N + NNEQ)
“russellrao”	RussellRaoDistance	NNZ / N
“sokalmichener”	SokalMichenerDistance	2 * NNEQ / (N + NNEQ)
“sokalsneath”	SokalSneathDistance	NNEQ / (NNEQ + 0.5 * NTT)

User-defined distance:

identifier	class name	args
“pyfunc”	PyFuncDistance	func

Here `func` is a function which takes two one-dimensional numpy arrays, and returns a distance. Note that in order to be used within the BallTree, the distance must be a true metric: i.e. it must satisfy the following properties

1) Non-negativity: d(x, y) >= 0
2) Identity: d(x, y) = 0 if and only if x == y
3) Symmetry: d(x, y) = d(y, x)
4) Triangle Inequality: d(x, y) + d(y, z) >= d(x, z)

Because of the Python object overhead involved in calling the python function, this will be fairly slow, but it will have the same scaling as other distances.

Methods

dist_to_rdist()	Convert the true distance to the reduced distance.
get_metric()	Get the given distance metric from the string identifier.
pairwise()	Compute the pairwise distances between X and Y
rdist_to_dist()	Convert the Reduced distance to the true distance.

__init__(*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

dist_to_rdist()
Convert the true distance to the reduced distance.

The reduced distance, defined for some metrics, is a computationally more efficient measure which preserves the rank of the true distance. For example, in the Euclidean distance metric, the reduced distance is the squared-euclidean distance.

get_metric()
Get the given distance metric from the string identifier.

See the docstring of DistanceMetric for a list of available metrics.

Parameters

metric [string or class name] The distance metric to use

****kwargs** additional arguments will be passed to the requested metric

pairwise()
Compute the pairwise distances between X and Y

This is a convenience routine for the sake of testing. For many metrics, the utilities in scipy.spatial.distance.cdist and scipy.spatial.distance.pdist will be faster.

Parameters

X [array_like] Array of shape (Nx, D), representing Nx points in D dimensions.

Y [array_like (optional)] Array of shape (Ny, D), representing Ny points in D dimensions. If not specified, then Y=X.

Returns

——-

dist [ndarray] The shape (Nx, Ny) array of pairwise distances between points in X and Y.

rdist_to_dist()
Convert the Reduced distance to the true distance.

The reduced distance, defined for some metrics, is a computationally more efficient measure which preserves the rank of the true distance. For example, in the Euclidean distance metric, the reduced distance is the squared-euclidean distance.

7.30.3 sklearn.neighbors.KDTree

class sklearn.neighbors.**KDTree**
KDTree for fast generalized N-point problems

KDTree(X, leaf_size=40, metric='minkowski', **kwargs)

Parameters

X [array-like of shape (n_samples, n_features)] n_samples is the number of points in the data set, and n_features is the dimension of the parameter space. Note: if X is a C-contiguous array of doubles then data will not be copied. Otherwise, an internal copy will be made.

leaf_size [positive integer (default = 40)] Number of points at which to switch to brute-force. Changing leaf_size will not affect the results of a query, but can significantly impact the speed of a query and the memory required to store the constructed tree. The amount of memory needed to store the tree scales as approximately n_samples / leaf_size. For a specified `leaf_size`, a leaf node is guaranteed to satisfy `leaf_size <= n_points <= 2 * leaf_size`, except in the case that `n_samples < leaf_size`.

metric [string or DistanceMetric object] the distance metric to use for the tree. Default='minkowski' with p=2 (that is, a euclidean metric). See the documentation of the DistanceMetric class for a list of available metrics. kd_tree.valid_metrics gives a list of the metrics which are valid for KDTree.

Additional keywords are passed to the distance metric class.

Note: Callable functions in the metric parameter are NOT supported for KDTree

and Ball Tree. Function call overhead will result in very poor performance.

Attributes

data [memory view] The training data

Examples

Query for k-nearest neighbors

```
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((10, 3))  # 10 points in 3 dimensions
>>> tree = KDTree(X, leaf_size=2)              # doctest: +SKIP
>>> dist, ind = tree.query(X[:1], k=3)                # doctest: +SKIP
>>> print(ind)  # indices of 3 closest neighbors
[0 3 1]
>>> print(dist)  # distances to 3 closest neighbors
[ 0.          0.19662693  0.29473397]
```

Pickle and Unpickle a tree. Note that the state of the tree is saved in the pickle operation: the tree needs not be rebuilt upon unpickling.

```
>>> import numpy as np
>>> import pickle
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((10, 3))  # 10 points in 3 dimensions
>>> tree = KDTree(X, leaf_size=2)        # doctest: +SKIP
>>> s = pickle.dumps(tree)                     # doctest: +SKIP
>>> tree_copy = pickle.loads(s)                # doctest: +SKIP
>>> dist, ind = tree_copy.query(X[:1], k=3)     # doctest: +SKIP
>>> print(ind)  # indices of 3 closest neighbors
[0 3 1]
>>> print(dist)  # distances to 3 closest neighbors
[ 0.          0.19662693  0.29473397]
```

Query for neighbors within a given radius

```
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((10, 3))  # 10 points in 3 dimensions
>>> tree = KDTree(X, leaf_size=2)     # doctest: +SKIP
>>> print(tree.query_radius(X[:1], r=0.3, count_only=True))
3
>>> ind = tree.query_radius(X[:1], r=0.3)  # doctest: +SKIP
>>> print(ind)  # indices of neighbors within distance 0.3
[3 0 1]
```

Compute a gaussian kernel density estimate:

```
>>> import numpy as np
>>> rng = np.random.RandomState(42)
>>> X = rng.random_sample((100, 3))
>>> tree = KDTree(X)                # doctest: +SKIP
>>> tree.kernel_density(X[:3], h=0.1, kernel='gaussian')
array([ 6.94114649,  7.83281226,  7.2071716 ])
```

Compute a two-point auto-correlation function

```
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((30, 3))
>>> r = np.linspace(0, 1, 5)
>>> tree = KDTree(X)                # doctest: +SKIP
```

(continues on next page)

(continued from previous page)

```
>>> tree.two_point_correlation(X, r)
array([ 30,  62, 278, 580, 820])
```

Methods

kernel_density(self, X, h[, kernel, atol, ...])	Compute the kernel density estimate at points X with the given kernel, using the distance metric specified at tree creation.
query(X[, k, return_distance, dualtree, ...])	query the tree for the k nearest neighbors
query_radius()	query_radius(self, X, r, count_only = False):
two_point_correlation()	Compute the two-point correlation function

get_arrays	
get_n_calls	
get_tree_stats	
reset_n_calls	

__init__ (*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

kernel_density (*self*, *X*, *h*, *kernel='gaussian'*, *atol=0*, *rtol=1E-8*, *breadth_first=True*, *return_log=False*)
Compute the kernel density estimate at points X with the given kernel, using the distance metric specified at tree creation.

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query. Last dimension should match dimension of training data.

h [float] the bandwidth of the kernel

kernel [string] specify the kernel to use. Options are - 'gaussian' - 'tophat' - 'epanechnikov' - 'exponential' - 'linear' - 'cosine' Default is kernel = 'gaussian'

atol, rtol [float (default = 0)] Specify the desired relative and absolute tolerance of the result. If the true result is K_true, then the returned result K_ret satisfies `abs(K_true - K_ret) < atol + rtol * K_ret` The default is zero (i.e. machine precision) for both.

breadth_first [boolean (default = False)] if True, use a breadth-first search. If False (default) use a depth-first search. Breadth-first is generally faster for compact kernels and/or high tolerances.

return_log [boolean (default = False)] return the logarithm of the result. This can be more accurate than returning the result itself for narrow kernels.

Returns

density [ndarray] The array of (log)-density evaluations, shape = X.shape[:-1]

query (*X*, *k=1*, *return_distance=True*, *dualtree=False*, *breadth_first=False*)
query the tree for the k nearest neighbors

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query

k [integer (default = 1)] The number of nearest neighbors to return

return_distance [boolean (default = True)] if True, return a tuple (d, i) of distances and indices if False, return array i

dualtree [boolean (default = False)] if True, use the dual tree formalism for the query: a tree is built for the query points, and the pair of trees is used to efficiently search this space. This can lead to better performance as the number of points grows large.

breadth_first [boolean (default = False)] if True, then query the nodes in a breadth-first manner. Otherwise, query the nodes in a depth-first manner.

sort_results [boolean (default = True)] if True, then distances and indices of each point are sorted on return, so that the first column contains the closest points. Otherwise, neighbors are returned in an arbitrary order.

Returns

i [if return_distance == False]

(d,i) [if return_distance == True]

d [array of doubles - shape: x.shape[:-1] + (k,)] each entry gives the list of distances to the neighbors of the corresponding point

i [array of integers - shape: x.shape[:-1] + (k,)] each entry gives the list of indices of neighbors of the corresponding point

query_radius()

query_radius(self, X, r, count_only = False):

query the tree for neighbors within a radius r

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query

r [distance within which neighbors are returned] r can be a single value, or an array of values of shape x.shape[:-1] if different radii are desired for each point.

return_distance [boolean (default = False)] if True, return distances to neighbors of each point if False, return only neighbors Note that unlike the query() method, setting return_distance=True here adds to the computation time. Not all distances need to be calculated explicitly for return_distance=False. Results are not sorted by default: see `sort_results` keyword.

count_only [boolean (default = False)] if True, return only the count of points within distance r if False, return the indices of all points within distance r If return_distance==True, setting count_only=True will result in an error.

sort_results [boolean (default = False)] if True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. If return_distance == False, setting sort_results = True will result in an error.

Returns

count [if count_only == True]

ind [if count_only == False and return_distance == False]

(ind, dist) [if count_only == False and return_distance == True]

count [array of integers, shape = X.shape[:-1]] each entry gives the number of neighbors within a distance r of the corresponding point.

ind [array of objects, shape = X.shape[:-1]] each element is a numpy integer array listing the indices of neighbors of the corresponding point. Note that unlike the results of a k-neighbors query, the returned neighbors are not sorted by distance by default.

dist [array of objects, shape = X.shape[:-1]] each element is a numpy double array listing the distances corresponding to indices in i.

two_point_correlation()
Compute the two-point correlation function

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query. Last dimension should match dimension of training data.

r [array_like] A one-dimensional array of distances

dualtree [boolean (default = False)] If true, use a dualtree algorithm. Otherwise, use a single-tree algorithm. Dual tree algorithms can have better scaling for large N.

Returns

counts [ndarray] counts[i] contains the number of pairs of points with distance less than or equal to r[i]

7.30.4 sklearn.neighbors.KernelDensity

class sklearn.neighbors.**KernelDensity**(*bandwidth=1.0*, *algorithm='auto'*, *kernel='gaussian'*, *metric='euclidean'*, *atol=0*, *rtol=0*, *breadth_first=True*, *leaf_size=40*, *metric_params=None*)
Kernel Density Estimation.

Read more in the *User Guide*.

Parameters

bandwidth [float] The bandwidth of the kernel.

algorithm [str] The tree algorithm to use. Valid options are ['kd_tree'|'ball_tree'|'auto']. Default is 'auto'.

kernel [str] The kernel to use. Valid kernels are ['gaussian'|'tophat'|'epanechnikov'|'exponential'|'linear'|'cosine'] Default is 'gaussian'.

metric [str] The distance metric to use. Note that not all metrics are valid with all algorithms. Refer to the documentation of *BallTree* and *KDTree* for a description of available algorithms. Note that the normalization of the density output is correct only for the Euclidean distance metric. Default is 'euclidean'.

atol [float] The desired absolute tolerance of the result. A larger tolerance will generally lead to faster execution. Default is 0.

rtol [float] The desired relative tolerance of the result. A larger tolerance will generally lead to faster execution. Default is 1E-8.

breadth_first [bool] If true (default), use a breadth-first approach to the problem. Otherwise use a depth-first approach.

leaf_size [int] Specify the leaf size of the underlying tree. See *BallTree* or *KDTree* for details. Default is 40.

metric_params [dict] Additional parameters to be passed to the tree for use with the metric. For more information, see the documentation of *BallTree* or *KDTree*.

See also:

sklearn.neighbors.KDTree K-dimensional tree for fast generalized N-point problems.

sklearn.neighbors.BallTree Ball tree for fast generalized N-point problems.

Examples

Compute a gaussian kernel density estimate with a fixed bandwidth. >>> import numpy as np >>> rng = np.random.RandomState(42) >>> X = rng.random_sample((100, 3)) >>> kde = KernelDensity(kernel='gaussian', bandwidth=0.5).fit(X) >>> log_density = kde.score_samples(X[:3]) >>> log_density array([-1.52955942, -1.51462041, -1.60244657])

Methods

fit(self, X[, y, sample_weight])	Fit the Kernel Density model on the data.
get_params(self[, deep])	Get parameters for this estimator.
sample(self[, n_samples, random_state])	Generate random samples from the model.
score(self, X[, y])	Compute the total log probability density under the model.
score_samples(self, X)	Evaluate the log density model on the data.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *bandwidth=1.0*, *algorithm='auto'*, *kernel='gaussian'*, *metric='euclidean'*, *atol=0*, *rtol=0*, *breadth_first=True*, *leaf_size=40*, *metric_params=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*, *sample_weight=None*)
Fit the Kernel Density model on the data.

Parameters

X [array_like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

y [None] Ignored. This parameter exists only for compatibility with *sklearn.pipeline.Pipeline*.

sample_weight [array_like, shape (n_samples,), optional] List of sample weights attached to the data X.

Returns

self [object] Returns instance of object.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

sample(*self*, *n_samples=1*, *random_state=None*)

Generate random samples from the model.

Currently, this is implemented only for gaussian and tophat kernels.

Parameters

n_samples [int, optional] Number of samples to generate. Defaults to 1.

random_state [int, RandomState instance or None. default to None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Returns

X [array_like, shape (n_samples, n_features)] List of samples.

score(*self*, *X*, *y=None*)

Compute the total log probability density under the model.

Parameters

X [array_like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

y [None] Ignored. This parameter exists only for compatibility with *sklearn.pipeline.Pipeline*.

Returns

logprob [float] Total log-likelihood of the data in X. This is normalized to be a probability density, so the value will be low for high-dimensional data.

score_samples(*self*, *X*)

Evaluate the log density model on the data.

Parameters

X [array_like, shape (n_samples, n_features)] An array of points to query. Last dimension should match dimension of training data (n_features).

Returns

density [ndarray, shape (n_samples,)] The array of log(density) evaluations. These are normalized to be probability densities, so values will be low for high-dimensional data.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.neighbors.KernelDensity`

- *Kernel Density Estimation*
- *Kernel Density Estimate of Species Distributions*
- *Simple 1D Kernel Density Estimation*

7.30.5 `sklearn.neighbors`.KNeighborsClassifier

class sklearn.neighbors.**KNeighborsClassifier**(*n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

Classifier implementing the k-nearest neighbors vote.

Read more in the *User Guide*.

Parameters

n_neighbors [int, optional (default = 5)] Number of neighbors to use by default for `kneighbors` queries.

weights [str or callable, optional (default = 'uniform')] weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, optional (default = 30)] Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p [integer, optional (default = 2)] Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric [string or callable, default 'minkowski'] the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics. If metric is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a *Glossary*, in which case only "nonzero" elements may be considered neighbors.

metric_params [dict, optional (default = None)] Additional keyword arguments for the metric function.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details. Doesn't affect *`fit`* method.

Attributes

classes_ [array of shape (n_classes,)] Class labels known to the classifier

effective_metric_ [string or callble] The distance metric used. It will be same as the `metric` parameter or a synonym of it, e.g. 'euclidean' if the `metric` parameter set to 'minkowski' and `p` parameter set to 2.

effective_metric_params_ [dict] Additional keyword arguments for the metric function. For most metrics will be same with `metric_params` parameter, but may also contain the `p` parameter value if the `effective_metric_` attribute is set to 'minkowski'.

outputs_2d_ [bool] False when `y`'s shape is (n_samples,) or (n_samples, 1) during fit otherwise True.

See also:

RadiusNeighborsClassifier

KNeighborsRegressor

RadiusNeighborsRegressor

NearestNeighbors

Notes

See *Nearest Neighbors* in the online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

Warning: Regarding the Nearest Neighbors algorithms, if it is found that two neighbors, neighbor `k+1` and `k`, have identical distances but different labels, the results will depend on the ordering of the training data.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import KNeighborsClassifier
>>> neigh = KNeighborsClassifier(n_neighbors=3)
>>> neigh.fit(X, y)
KNeighborsClassifier(...)
>>> print(neigh.predict([[1.1]]))
[0]
>>> print(neigh.predict_proba([[0.9]]))
[[0.66666667 0.33333333]]
```

Methods

fit(self, X, y)	Fit the model using X as training data and y as target values
get_params(self[, deep])	Get parameters for this estimator.
kneighbors(self[, X, n_neighbors, ...])	Finds the K-neighbors of a point.
kneighbors_graph(self[, X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
predict(self, X)	Predict the class labels for the provided data.
predict_proba(self, X)	Return probability estimates for the test data X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)
Fit the model using X as training data and y as target values

Parameters

X [{array-like, sparse matrix, BallTree, KDTree}] Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.

y [{array-like, sparse matrix}] Target values of shape = [n_samples] or [n_samples, n_outputs]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

kneighbors(*self*, *X=None*, *n_neighbors=None*, *return_distance=True*)
Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

neigh_dist [array, shape (n_queries, n_neighbors)] Array representing the lengths to points, only present if return_distance=True

neigh_ind [array, shape (n_queries, n_neighbors)] Indices of the nearest points in the population matrix.

Examples

In the following example, we construct a NearestNeighbors class from an array representing our data set and ask who's the closest point to [1,1,1]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(n_neighbors=1)
>>> print(neigh.kneighbors([[1., 1., 1.]]))
(array([[0.5]]), array([[2]]))
```

As you can see, it returns [[0.5]], and [[2]], which means that the element is at distance 0.5 and is the third element of samples (indexes start at 0). You can also query for multiple points:

```
>>> X = [[0., 1., 0.], [1., 0., 1.]]
>>> neigh.kneighbors(X, return_distance=False)
array([[1],
       [2]]...)
```

kneighbors_graph (*self*, *X=None*, *n_neighbors=None*, *mode='connectivity'*)
Computes the (weighted) graph of k-Neighbors for points in X

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors for each sample. (default is value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

NearestNeighbors.radius_neighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
```

(continues on next page)

(continued from previous page)

```
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

predict (*self*, *X*)

Predict the class labels for the provided data.

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] Test samples.

Returns

y [array of shape [n_queries] or [n_queries, n_outputs]] Class labels for each data sample.

predict_proba (*self*, *X*)

Return probability estimates for the test data X.

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] Test samples.

Returns

p [array of shape = [n_queries, n_classes], or a list of n_outputs] of such arrays if n_outputs > 1. The class probabilities of the input samples. Classes are ordered by lexicographic order.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.neighbors.KNeighborsClassifier`

- *Classifier comparison*
- *Plot the decision boundaries of a VotingClassifier*
- *Nearest Neighbors Classification*
- *Caching nearest neighbors*
- *Comparing Nearest Neighbors with and without Neighborhood Components Analysis*
- *Dimensionality Reduction with Neighborhood Components Analysis*
- *Digits Classification Exercise*
- *Classification of text documents using sparse features*

7.30.6 `sklearn.neighbors.KNeighborsRegressor`

class sklearn.neighbors.**KNeighborsRegressor**(*n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

Regression based on k-nearest neighbors.

The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

Read more in the *User Guide*.

New in version 0.9.

Parameters

n_neighbors [int, optional (default = 5)] Number of neighbors to use by default for *kneighbors* queries.

weights [str or callable] weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

Uniform weights are used by default.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use *BallTree*
- 'kd_tree' will use *KDTree*
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to *fit* method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, optional (default = 30)] Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p [integer, optional (default = 2)] Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric [string or callable, default 'minkowski'] the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics. If metric is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a *Glossary*, in which case only "nonzero" elements may be considered neighbors.

metric_params [dict, optional (default = None)] Additional keyword arguments for the metric function.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details. Doesn't affect *`fit`* method.

Attributes

effective_metric_ [string or callable] The distance metric to use. It will be same as the `metric` parameter or a synonym of it, e.g. 'euclidean' if the `metric` parameter set to 'minkowski' and `p` parameter set to 2.

effective_metric_params_ [dict] Additional keyword arguments for the metric function. For most metrics will be same with `metric_params` parameter, but may also contain the `p` parameter value if the `effective_metric_` attribute is set to 'minkowski'.

See also:

NearestNeighbors

RadiusNeighborsRegressor

KNeighborsClassifier

RadiusNeighborsClassifier

Notes

See *Nearest Neighbors* in the online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

Warning: Regarding the Nearest Neighbors algorithms, if it is found that two neighbors, neighbor `k+1` and `k`, have identical distances but different labels, the results will depend on the ordering of the training data.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import KNeighborsRegressor
```

(continues on next page)

(continued from previous page)

```
>>> neigh = KNeighborsRegressor(n_neighbors=2)
>>> neigh.fit(X, y)
KNeighborsRegressor(...)
>>> print(neigh.predict([[1.5]]))
[0.5]
```

Methods

fit(self, X, y)	Fit the model using X as training data and y as target values
get_params(self[, deep])	Get parameters for this estimator.
kneighbors(self[, X, n_neighbors, ...])	Finds the K-neighbors of a point.
kneighbors_graph(self[, X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
predict(self, X)	Predict the target for the provided data
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Fit the model using X as training data and y as target values

Parameters

X [{array-like, sparse matrix, BallTree, KDTree}] Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.

y [{array-like, sparse matrix}]

Target values, array of float values, shape = [n_samples] or [n_samples, n_outputs]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

kneighbors (*self*, *X=None*, *n_neighbors=None*, *return_distance=True*)
Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

neigh_dist [array, shape (n_queries, n_neighbors)] Array representing the lengths to points, only present if return_distance=True

neigh_ind [array, shape (n_queries, n_neighbors)] Indices of the nearest points in the population matrix.

Examples

In the following example, we construct a NearestNeighbors class from an array representing our data set and ask who's the closest point to [1,1,1]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(n_neighbors=1)
>>> print(neigh.kneighbors([[1., 1., 1.]]))
(array([[0.5]]), array([[2]]))
```

As you can see, it returns [[0.5]], and [[2]], which means that the element is at distance 0.5 and is the third element of samples (indexes start at 0). You can also query for multiple points:

```
>>> X = [[0., 1., 0.], [1., 0., 1.]]
>>> neigh.kneighbors(X, return_distance=False)
array([[1],
       [2]]...)
```

kneighbors_graph(*self*, *X=None*, *n_neighbors=None*, *mode='connectivity'*)

Computes the (weighted) graph of k-Neighbors for points in X

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors for each sample. (default is value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

NearestNeighbors.radius_neighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

predict (*self*, *X*)

Predict the target for the provided data

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] Test samples.

Returns

y [array of int, shape = [n_queries] or [n_queries, n_outputs]] Target values

score (*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.neighbors.KNeighborsRegressor`

- *Face completion with a multi-output estimators*
- *Imputing missing values with variants of IterativeImputer*
- *Nearest Neighbors regression*

7.30.7 `sklearn.neighbors`.KNeighborsTransformer

class sklearn.neighbors.**KNeighborsTransformer**(*mode='distance'*, *n_neighbors=5*, *algorithm='auto'*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *n_jobs=1*)

Transform X into a (weighted) graph of k nearest neighbors

The transformed data is a sparse graph as returned by kneighbors_graph.

Read more in the *User Guide*.

New in version 0.22.

Parameters

mode [{'distance', 'connectivity'}, default='distance'] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, and 'distance' will return the distances between neighbors according to the given metric.

n_neighbors [int, default=5] Number of neighbors for each sample in the transformed sparse graph. For compatibility reasons, as each sample is considered as its own neighbor, one extra neighbor will be computed when mode == 'distance'. In this case, the sparse graph contains (n_neighbors + 1) neighbors.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'] Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use *BallTree*
- 'kd_tree' will use *KDTree*
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to *fit* method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, default=30] Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

metric [string or callable, default='minkowski'] metric to use for distance computation. Any metric from scikit-learn or scipy.spatial.distance can be used.

If metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays as input and return one value indicating the distance between them. This works for Scipy's metrics, but is less efficient than passing the metric name as a string.

Distance matrices are not supported.

Valid values for metric are:

- from scikit-learn: ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']
- from scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule']

See the documentation for scipy.spatial.distance for details on these metrics.

p [int, default=2] Parameter for the Minkowski metric from sklearn.metrics.pairwise.pairwise_distances. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params [dict, default=None] Additional keyword arguments for the metric function.

n_jobs [int, default=1] The number of parallel jobs to run for neighbors search. If `-1`, then the number of jobs is set to the number of CPU cores.

Examples

```
>>> from sklearn.manifold import Isomap
>>> from sklearn.neighbors import KNeighborsTransformer
>>> from sklearn.pipeline import make_pipeline
>>> estimator = make_pipeline(
...     KNeighborsTransformer(n_neighbors=5, mode='distance'),
...     Isomap(neighbors_algorithm='precomputed'))
```

Methods

fit(self, X[, y])	Fit the model using X as training data
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
kneighbors(self[, X, n_neighbors, ...])	Finds the K-neighbors of a point.
kneighbors_graph(self[, X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Computes the (weighted) graph of Neighbors for points in X

__init__(*self*, *mode='distance'*, *n_neighbors=5*, *algorithm='auto'*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *n_jobs=1*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)

Fit the model using X as training data

Parameters

X [{array-like, sparse matrix, BallTree, KDTree}] Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.

fit_transform(*self*, *X*, *y=None*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [array-like of shape (n_samples, n_features)] Training set.

y [ignored]

Returns

Xt [CSR sparse graph of shape (n_samples, n_samples)] Xt[i, j] is assigned the weight of edge that connects i to j. Only the neighbors have an explicit value. The diagonal is always explicit.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

kneighbors(*self*, *X=None*, *n_neighbors=None*, *return_distance=True*)
Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

neigh_dist [array, shape (n_queries, n_neighbors)] Array representing the lengths to points, only present if return_distance=True

neigh_ind [array, shape (n_queries, n_neighbors)] Indices of the nearest points in the population matrix.

Examples

In the following example, we construct a NearestNeighbors class from an array representing our data set and ask who's the closest point to [1,1,1]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(n_neighbors=1)
>>> print(neigh.kneighbors([[1., 1., 1.]]))
(array([[0.5]]), array([[2]]))
```

As you can see, it returns [[0.5]], and [[2]], which means that the element is at distance 0.5 and is the third element of samples (indexes start at 0). You can also query for multiple points:

```
>>> X = [[0., 1., 0.], [1., 0., 1.]]
>>> neigh.kneighbors(X, return_distance=False)
array([[1],
       [2]]...)
```

kneighbors_graph(*self*, *X=None*, *n_neighbors=None*, *mode='connectivity'*)

Computes the (weighted) graph of k-Neighbors for points in X

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors for each sample. (default is value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

NearestNeighbors.radius_neighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)

Computes the (weighted) graph of Neighbors for points in X

Parameters

X [array-like of shape (n_samples_transform, n_features)] Sample data

Returns

Xt [CSR sparse graph of shape (n_samples_transform, n_samples_fit)] Xt[i, j] is assigned the weight of edge that connects i to j. Only the neighbors have an explicit value. The diagonal is always explicit.

Examples using `sklearn.neighbors.KNeighborsTransformer`

- *Approximate nearest neighbors in TSNE*

7.30.8 `sklearn.neighbors`.LocalOutlierFactor

class sklearn.neighbors.**LocalOutlierFactor**(*n_neighbors=20*, *algorithm='auto'*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *contamination='auto'*, *novelty=False*, *n_jobs=None*)

Unsupervised Outlier Detection using Local Outlier Factor (LOF)

The anomaly score of each sample is called Local Outlier Factor. It measures the local deviation of density of a given sample with respect to its neighbors. It is local in that the anomaly score depends on how isolated the object is with respect to the surrounding neighborhood. More precisely, locality is given by k-nearest neighbors, whose distance is used to estimate the local density. By comparing the local density of a sample to the local densities of its neighbors, one can identify samples that have a substantially lower density than their neighbors. These are considered outliers.

New in version 0.19.

Parameters

n_neighbors [int, optional (default=20)] Number of neighbors to use by default for *kneighbors* queries. If n_neighbors is larger than the number of samples provided, all samples will be used.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use *BallTree*
- 'kd_tree' will use *KDTree*
- 'brute' will use a brute-force search.

- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to *fit* method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, optional (default=30)] Leaf size passed to *BallTree* or *KDTree*. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

metric [string or callable, default 'minkowski'] metric used for the distance computation. Any metric from scikit-learn or scipy.spatial.distance can be used.

If metric is "precomputed", X is assumed to be a distance matrix and must be square. X may be a sparse matrix, in which case only "nonzero" elements may be considered neighbors.

If metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays as input and return one value indicating the distance between them. This works for Scipy's metrics, but is less efficient than passing the metric name as a string.

Valid values for metric are:

- from scikit-learn: ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']
- from scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule']

See the documentation for scipy.spatial.distance for details on these metrics: https://docs.scipy.org/doc/scipy/reference/spatial.distance.html

p [integer, optional (default=2)] Parameter for the Minkowski metric from `sklearn.metrics.pairwise.pairwise_distances`. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params [dict, optional (default=None)] Additional keyword arguments for the metric function.

contamination ['auto' or float, optional (default='auto')] The amount of contamination of the data set, i.e. the proportion of outliers in the data set. When fitting this is used to define the threshold on the scores of the samples.

- if 'auto', the threshold is determined as in the original paper,
- if a float, the contamination should be in the range [0, 0.5].

Changed in version 0.22: The default value of `contamination` changed from 0.1 to `'auto'`.

novelty [boolean, default False] By default, LocalOutlierFactor is only meant to be used for outlier detection (novelty=False). Set novelty to True if you want to use LocalOutlierFactor for novelty detection. In this case be aware that that you should only use predict, decision_function and score_samples on new unseen data and not on the training set.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

negative_outlier_factor_ [numpy array, shape (n_samples,)] The opposite LOF of the training samples. The higher, the more normal. Inliers tend to have a LOF score close to 1

(negative_outlier_factor_ close to -1), while outliers tend to have a larger LOF score.

The local outlier factor (LOF) of a sample captures its supposed 'degree of abnormality'. It is the average of the ratio of the local reachability density of a sample and those of its k-nearest neighbors.

n_neighbors_ [integer] The actual number of neighbors used for *kneighbors* queries.

offset_ [float] Offset used to obtain binary labels from the raw scores. Observations having a negative_outlier_factor smaller than offset_ are detected as abnormal. The offset is set to -1.5 (inliers score around -1), except when a contamination parameter different than "auto" is provided. In that case, the offset is defined in such a way we obtain the expected number of outliers in training.

References

[Rca479bb49841-1]

Examples

```
>>> import numpy as np
>>> from sklearn.neighbors import LocalOutlierFactor
>>> X = [[-1.1], [0.2], [101.1], [0.3]]
>>> clf = LocalOutlierFactor(n_neighbors=2)
>>> clf.fit_predict(X)
array([ 1,  1, -1,  1])
>>> clf.negative_outlier_factor_
array([ -0.9821...,  -1.0370..., -73.3697...,  -0.9821...])
```

Methods

fit(self, X[, y])	Fit the model using X as training data.
get_params(self[, deep])	Get parameters for this estimator.
kneighbors(self[, X, n_neighbors, ...])	Finds the K-neighbors of a point.
kneighbors_graph(self[, X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_neighbors=20*, *algorithm='auto'*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *contamination='auto'*, *novelty=False*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

property decision_function
Shifted opposite of the Local Outlier Factor of X.

Bigger is better, i.e. large values correspond to inliers.

The shift offset allows a zero threshold for being an outlier. Only available for novelty detection (when novelty is set to True). The argument X is supposed to contain *new data*: if X contains a point from training, it considers the later in its own neighborhood. Also, the samples in X are not considered in the neighborhood of any point.

Parameters

X [array-like, shape (n_samples, n_features)] The query sample or samples to compute the Local Outlier Factor w.r.t. the training samples.

Returns

shifted_opposite_lof_scores [array, shape (n_samples,)] The shifted opposite of the Local Outlier Factor of each input samples. The lower, the more abnormal. Negative scores represent outliers, positive scores represent inliers.

fit (*self*, *X*, *y=None*)

Fit the model using X as training data.

Parameters

X [{array-like, sparse matrix, BallTree, KDTree}] Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.

y [Ignored] not used, present for API consistency by convention.

Returns

self [object]

property fit_predict

"Fits the model to the training set X and returns the labels.

Label is 1 for an inlier and -1 for an outlier according to the LOF score and the contamination parameter.

Parameters

X [array-like, shape (n_samples, n_features), default=None] The query sample or samples to compute the Local Outlier Factor w.r.t. to the training samples.

y [Ignored] not used, present for API consistency by convention.

Returns

is_inlier [array, shape (n_samples,)] Returns -1 for anomalies/outliers and 1 for inliers.

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

kneighbors (*self*, *X=None*, *n_neighbors=None*, *return_distance=True*)

Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

neigh_dist [array, shape (n_queries, n_neighbors)] Array representing the lengths to points, only present if return_distance=True

neigh_ind [array, shape (n_queries, n_neighbors)] Indices of the nearest points in the population matrix.

Examples

In the following example, we construct a NearestNeighbors class from an array representing our data set and ask who's the closest point to [1,1,1]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(n_neighbors=1)
>>> print(neigh.kneighbors([[1., 1., 1.]]))
(array([[0.5]]), array([[2]]))
```

As you can see, it returns [[0.5]], and [[2]], which means that the element is at distance 0.5 and is the third element of samples (indexes start at 0). You can also query for multiple points:

```
>>> X = [[0., 1., 0.], [1., 0., 1.]]
>>> neigh.kneighbors(X, return_distance=False)
array([[1],
       [2]]...)
```

kneighbors_graph (*self*, *X=None*, *n_neighbors=None*, *mode='connectivity'*)
Computes the (weighted) graph of k-Neighbors for points in X

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors for each sample. (default is value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

NearestNeighbors.radius_neighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

property predict

Predict the labels (1 inlier, -1 outlier) of X according to LOF.

This method allows to generalize prediction to *new observations* (not in the training set). Only available for novelty detection (when novelty is set to True).

Parameters

X [array-like, shape (n_samples, n_features)] The query sample or samples to compute the Local Outlier Factor w.r.t. to the training samples.

Returns

is_inlier [array, shape (n_samples,)] Returns -1 for anomalies/outliers and +1 for inliers.

property score_samples

Opposite of the Local Outlier Factor of X.

It is the opposite as bigger is better, i.e. large values correspond to inliers.

Only available for novelty detection (when novelty is set to True). The argument X is supposed to contain *new data*: if X contains a point from training, it considers the later in its own neighborhood. Also, the samples in X are not considered in the neighborhood of any point. The score_samples on training data is available by considering the the `negative_outlier_factor_` attribute.

Parameters

X [array-like, shape (n_samples, n_features)] The query sample or samples to compute the Local Outlier Factor w.r.t. the training samples.

Returns

opposite_lof_scores [array, shape (n_samples,)] The opposite of the Local Outlier Factor of each input samples. The lower, the more abnormal.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.neighbors.LocalOutlierFactor`

- *Comparing anomaly detection algorithms for outlier detection on toy datasets*
- *Outlier detection with Local Outlier Factor (LOF)*
- *Novelty detection with Local Outlier Factor (LOF)*

7.30.9 `sklearn.neighbors`.RadiusNeighborsClassifier

class sklearn.neighbors.**RadiusNeighborsClassifier**(*radius=1.0*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *outlier_label=None*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

Classifier implementing a vote among neighbors within a given radius

Read more in the *User Guide*.

Parameters

radius [float, optional (default = 1.0)] Range of parameter space to use by default for `radius_neighbors` queries.

weights [str or callable] weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

Uniform weights are used by default.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, optional (default = 30)] Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p [integer, optional (default = 2)] Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric [string or callable, default 'minkowski'] the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics. If metric

is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a *Glossary*, in which case only "nonzero" elements may be considered neighbors.

outlier_label [{manual label, 'most_frequent'}, optional (default = None)] label for outlier samples (samples with no neighbors in given radius).

- manual label: str or int label (should be the same type as y) or list of manual labels if multi-output is used.
- 'most_frequent' : assign the most frequent label of y to outliers.
- None : when any outlier is detected, ValueError will be raised.

metric_params [dict, optional (default = None)] Additional keyword arguments for the metric function.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

classes_ [array of shape (n_classes,)] Class labels known to the classifier.

effective_metric_ [string or callble] The distance metric used. It will be same as the `metric` parameter or a synonym of it, e.g. 'euclidean' if the `metric` parameter set to 'minkowski' and `p` parameter set to 2.

effective_metric_params_ [dict] Additional keyword arguments for the metric function. For most metrics will be same with `metric_params` parameter, but may also contain the `p` parameter value if the `effective_metric_` attribute is set to 'minkowski'.

outputs_2d_ [bool] False when `y`'s shape is (n_samples,) or (n_samples, 1) during fit otherwise True.

See also:

KNeighborsClassifier

RadiusNeighborsRegressor

KNeighborsRegressor

NearestNeighbors

Notes

See *Nearest Neighbors* in the online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import RadiusNeighborsClassifier
>>> neigh = RadiusNeighborsClassifier(radius=1.0)
>>> neigh.fit(X, y)
RadiusNeighborsClassifier(...)
```

(continues on next page)

(continued from previous page)

```
>>> print(neigh.predict([[1.5]]))
[0]
>>> print(neigh.predict_proba([[1.0]]))
[[0.66666667 0.33333333]]
```

Methods

fit(self, X, y)	Fit the model using X as training data and y as target values
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the class labels for the provided data.
predict_proba(self, X)	Return probability estimates for the test data X.
radius_neighbors(self[, X, radius, ...])	Finds the neighbors within a given radius of a point or points.
radius_neighbors_graph(self[, X, radius, ...])	Computes the (weighted) graph of Neighbors for points in X
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *radius=1.0*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *outlier_label=None*, *metric_params=None*, *n_jobs=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)
Fit the model using X as training data and y as target values

Parameters

X [{array-like, sparse matrix, BallTree, KDTree}] Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.

y [{array-like, sparse matrix}] Target values of shape = [n_samples] or [n_samples, n_outputs]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict the class labels for the provided data.

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] Test samples.

Returns

y [array of shape [n_queries] or [n_queries, n_outputs]] Class labels for each data sample.

predict_proba(*self*, *X*)
Return probability estimates for the test data X.

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] Test samples.

Returns

p [array of shape = [n_queries, n_classes], or a list of n_outputs] of such arrays if n_outputs > 1. The class probabilities of the input samples. Classes are ordered by lexicographic order.

radius_neighbors(*self*, *X=None*, *radius=None*, *return_distance=True*, *sort_results=False*)
Finds the neighbors within a given radius of a point or points.

Return the indices and distances of each point from the dataset lying in a ball with size `radius` around the points of the query array. Points lying on the boundary are included in the results.

The result points are *not* necessarily sorted by distance to their query point.

Parameters

X [array-like, (n_samples, n_features), optional] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Limiting distance of neighbors to return. (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned.

sort_results [boolean, optional. Defaults to False.] If True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. If return_distance == False, setting sort_results = True will result in an error.

New in version 0.22.

Returns

neigh_dist [array, shape (n_samples,) of arrays] Array representing the distances to each point, only present if return_distance=True. The distance values are computed according to the `metric` constructor parameter.

neigh_ind [array, shape (n_samples,) of arrays] An array of arrays of indices of the approximate nearest points from the population matrix that lie within a ball of size `radius` around the query points.

Notes

Because the number of neighbors of each point is not necessarily equal, the results for multiple query points cannot be fit in a standard data array. For efficiency, `radius_neighbors` returns arrays of objects, where each object is a 1D array of indices or distances.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1, 1, 1]:

```
>>> import numpy as np
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.6)
>>> neigh.fit(samples)
NearestNeighbors(radius=1.6)
>>> rng = neigh.radius_neighbors([[1., 1., 1.]])
>>> print(np.asarray(rng[0][0]))
[1.5 0.5]
>>> print(np.asarray(rng[1][0]))
[1 2]
```

The first array returned contains the distances to all points which are closer than 1.6, while the second array returned contains their indices. In general, multiple points can be queried at the same time.

radius_neighbors_graph (*self*, *X=None*, *radius=None*, *mode='connectivity'*, *sort_results=False*)

Computes the (weighted) graph of Neighbors for points in X

Neighborhoods are restricted the points at a distance lower than radius.

Parameters

X [array-like of shape (n_samples, n_features), default=None] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Radius of neighborhoods. (default is the value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

sort_results [boolean, optional. Defaults to False.] If True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. Only used with mode='distance'.

New in version 0.22.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

kneighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.5)
>>> neigh.fit(X)
NearestNeighbors(radius=1.5)
>>> A = neigh.radius_neighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
```

(continues on next page)

(continued from previous page)

```
        [0., 1., 0.],
        [1., 0., 1.]])
```

score(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

7.30.10 sklearn.neighbors.RadiusNeighborsRegressor

class sklearn.neighbors.**RadiusNeighborsRegressor**(*radius=1.0*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

Regression based on neighbors within a fixed radius.

The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

Read more in the *User Guide*.

New in version 0.9.

Parameters

radius [float, optional (default = 1.0)] Range of parameter space to use by default for `radius_neighbors` queries.

weights [str or callable] weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

Uniform weights are used by default.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use *BallTree*
- 'kd_tree' will use *KDTree*
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to *fit* method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, optional (default = 30)] Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p [integer, optional (default = 2)] Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric [string or callable, default 'minkowski'] the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics. If metric is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a *Glossary*, in which case only "nonzero" elements may be considered neighbors.

metric_params [dict, optional (default = None)] Additional keyword arguments for the metric function.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

effective_metric_ [string or callable] The distance metric to use. It will be same as the `metric` parameter or a synonym of it, e.g. 'euclidean' if the `metric` parameter set to 'minkowski' and `p` parameter set to 2.

effective_metric_params_ [dict] Additional keyword arguments for the metric function. For most metrics will be same with `metric_params` parameter, but may also contain the `p` parameter value if the `effective_metric_` attribute is set to 'minkowski'.

See also:

NearestNeighbors

KNeighborsRegressor

KNeighborsClassifier

RadiusNeighborsClassifier

Notes

See *Nearest Neighbors* in the online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import RadiusNeighborsRegressor
>>> neigh = RadiusNeighborsRegressor(radius=1.0)
>>> neigh.fit(X, y)
RadiusNeighborsRegressor(...)
>>> print(neigh.predict([[1.5]]))
[0.5]
```

Methods

fit(self, X, y)	Fit the model using X as training data and y as target values
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the target for the provided data
radius_neighbors(self[, X, radius, ...])	Finds the neighbors within a given radius of a point or points.
radius_neighbors_graph(self[, X, radius, ...])	Computes the (weighted) graph of Neighbors for points in X
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *radius=1.0*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Fit the model using X as training data and y as target values

Parameters

X [{array-like, sparse matrix, BallTree, KDTree}] Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.

y [{array-like, sparse matrix}]

Target values, array of float values, shape = [n_samples] or [n_samples, n_outputs]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)

Predict the target for the provided data

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] Test samples.

Returns

y [array of float, shape = [n_queries] or [n_queries, n_outputs]] Target values

radius_neighbors (*self*, *X=None*, *radius=None*, *return_distance=True*, *sort_results=False*)

Finds the neighbors within a given radius of a point or points.

Return the indices and distances of each point from the dataset lying in a ball with size `radius` around the points of the query array. Points lying on the boundary are included in the results.

The result points are *not* necessarily sorted by distance to their query point.

Parameters

X [array-like, (n_samples, n_features), optional] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Limiting distance of neighbors to return. (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned.

sort_results [boolean, optional. Defaults to False.] If True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. If return_distance == False, setting sort_results = True will result in an error.

New in version 0.22.

Returns

neigh_dist [array, shape (n_samples,) of arrays] Array representing the distances to each point, only present if return_distance=True. The distance values are computed according to the `metric` constructor parameter.

neigh_ind [array, shape (n_samples,) of arrays] An array of arrays of indices of the approximate nearest points from the population matrix that lie within a ball of size `radius` around the query points.

Notes

Because the number of neighbors of each point is not necessarily equal, the results for multiple query points cannot be fit in a standard data array. For efficiency, `radius_neighbors` returns arrays of objects, where each object is a 1D array of indices or distances.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1, 1, 1]:

```
>>> import numpy as np
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.6)
>>> neigh.fit(samples)
NearestNeighbors(radius=1.6)
>>> rng = neigh.radius_neighbors([[1., 1., 1.]])
>>> print(np.asarray(rng[0][0]))
[1.5 0.5]
>>> print(np.asarray(rng[1][0]))
[1 2]
```

The first array returned contains the distances to all points which are closer than 1.6, while the second array returned contains their indices. In general, multiple points can be queried at the same time.

radius_neighbors_graph (*self*, *X=None*, *radius=None*, *mode='connectivity'*, *sort_results=False*)
Computes the (weighted) graph of Neighbors for points in X

Neighborhoods are restricted the points at a distance lower than radius.

Parameters

X [array-like of shape (n_samples, n_features), default=None] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Radius of neighborhoods. (default is the value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

sort_results [boolean, optional. Defaults to False.] If True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. Only used with mode='distance'.

New in version 0.22.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

kneighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.5)
>>> neigh.fit(X)
NearestNeighbors(radius=1.5)
>>> A = neigh.radius_neighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
```

(continues on next page)

(continued from previous page)

```
        [0., 1., 0.],
        [1., 0., 1.]])
```

score(*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

7.30.11 sklearn.neighbors.RadiusNeighborsTransformer

class sklearn.neighbors.**RadiusNeighborsTransformer**(*mode='distance'*, *radius=1.0*, *algorithm='auto'*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *n_jobs=1*)

Transform X into a (weighted) graph of neighbors nearer than a radius

The transformed data is a sparse graph as returned by radius_neighbors_graph.

Read more in the *User Guide*.

New in version 0.22.

Parameters

mode [{'distance', 'connectivity'}, default='distance'] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, and 'distance' will return the distances between neighbors according to the given metric.

radius [float, default=1.] Radius of neighborhood in the transformed sparse graph.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'] Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, default=30] Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

metric [string or callable, default='minkowski'] metric to use for distance computation. Any metric from scikit-learn or scipy.spatial.distance can be used.

If metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays as input and return one value indicating the distance between them. This works for Scipy's metrics, but is less efficient than passing the metric name as a string.

Distance matrices are not supported.

Valid values for metric are:

- from scikit-learn: ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']
- from scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule']

See the documentation for scipy.spatial.distance for details on these metrics.

p [int, default=2] Parameter for the Minkowski metric from sklearn.metrics.pairwise.pairwise_distances. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params [dict, default=None] Additional keyword arguments for the metric function.

n_jobs [int, default=1] The number of parallel jobs to run for neighbors search. If `-1`, then the number of jobs is set to the number of CPU cores.

Examples

```
>>> from sklearn.cluster import DBSCAN
>>> from sklearn.neighbors import RadiusNeighborsTransformer
>>> from sklearn.pipeline import make_pipeline
>>> estimator = make_pipeline(
...     RadiusNeighborsTransformer(radius=42.0, mode='distance'),
...     DBSCAN(min_samples=30, metric='precomputed'))
```

Methods

fit(self, X[, y])	Fit the model using X as training data
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
radius_neighbors(self[, X, radius, ...])	Finds the neighbors within a given radius of a point or points.
radius_neighbors_graph(self[, X, radius, ...])	Computes the (weighted) graph of Neighbors for points in X
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Computes the (weighted) graph of Neighbors for points in X

__init__(*self*, *mode='distance'*, *radius=1.0*, *algorithm='auto'*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *n_jobs=1*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the model using X as training data

Parameters

X [{array-like, sparse matrix, BallTree, KDTree}] Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.

fit_transform(*self*, *X*, *y=None*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [array-like of shape (n_samples, n_features)] Training set.

y [ignored]

Returns

Xt [CSR sparse graph, shape (n_samples, n_samples)] Xt[i, j] is assigned the weight of edge that connects i to j. Only the neighbors have an explicit value. The diagonal is always explicit.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

radius_neighbors (*self*, *X=None*, *radius=None*, *return_distance=True*, *sort_results=False*)
Finds the neighbors within a given radius of a point or points.

Return the indices and distances of each point from the dataset lying in a ball with size `radius` around the points of the query array. Points lying on the boundary are included in the results.

The result points are *not* necessarily sorted by distance to their query point.

Parameters

X [array-like, (n_samples, n_features), optional] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Limiting distance of neighbors to return. (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned.

sort_results [boolean, optional. Defaults to False.] If True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. If return_distance == False, setting sort_results = True will result in an error.

New in version 0.22.

Returns

neigh_dist [array, shape (n_samples,) of arrays] Array representing the distances to each point, only present if return_distance=True. The distance values are computed according to the `metric` constructor parameter.

neigh_ind [array, shape (n_samples,) of arrays] An array of arrays of indices of the approximate nearest points from the population matrix that lie within a ball of size `radius` around the query points.

Notes

Because the number of neighbors of each point is not necessarily equal, the results for multiple query points cannot be fit in a standard data array. For efficiency, `radius_neighbors` returns arrays of objects, where each object is a 1D array of indices or distances.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1, 1, 1]:

```
>>> import numpy as np
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.6)
>>> neigh.fit(samples)
NearestNeighbors(radius=1.6)
>>> rng = neigh.radius_neighbors([[1., 1., 1.]])
>>> print(np.asarray(rng[0][0]))
```

(continues on next page)

(continued from previous page)

```
[1.5 0.5]
>>> print(np.asarray(rng[1][0]))
[1 2]
```

The first array returned contains the distances to all points which are closer than 1.6, while the second array returned contains their indices. In general, multiple points can be queried at the same time.

radius_neighbors_graph(*self*, *X=None*, *radius=None*, *mode='connectivity'*, *sort_results=False*)
Computes the (weighted) graph of Neighbors for points in X

Neighborhoods are restricted the points at a distance lower than radius.

Parameters

X [array-like of shape (n_samples, n_features), default=None] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Radius of neighborhoods. (default is the value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

sort_results [boolean, optional. Defaults to False.] If True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. Only used with mode='distance'.

New in version 0.22.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

kneighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.5)
>>> neigh.fit(X)
NearestNeighbors(radius=1.5)
>>> A = neigh.radius_neighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 0.],
       [1., 0., 1.]])
```

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)

Computes the (weighted) graph of Neighbors for points in X

Parameters

X [array-like of shape (n_samples_transform, n_features)] Sample data

Returns

Xt [CSR sparse graph of shape (n_samples_transform, n_samples_fit)] Xt[i, j] is assigned the weight of edge that connects i to j. Only the neighbors have an explicit value. The diagonal is always explicit.

7.30.12 sklearn.neighbors.NearestCentroid

class sklearn.neighbors.**NearestCentroid**(*metric='euclidean'*, *shrink_threshold=None*)

Nearest centroid classifier.

Each class is represented by its centroid, with test samples classified to the class with the nearest centroid.

Read more in the *User Guide*.

Parameters

metric [string, or callable] The metric to use when calculating distance between instances in a feature array. If metric is a string or callable, it must be one of the options allowed by metrics.pairwise.pairwise_distances for its metric parameter. The centroids for the samples corresponding to each class is the point from which the sum of the distances (according to the metric) of all samples that belong to that particular class are minimized. If the "manhattan" metric is provided, this centroid is the median and for all other metrics, the centroid is now set to be the mean.

shrink_threshold [float, optional (default = None)] Threshold for shrinking centroids to remove features.

Attributes

centroids_ [array-like of shape (n_classes, n_features)] Centroid of each class.

classes_ [array of shape (n_classes,)] The unique classes labels.

See also:

sklearn.neighbors.KNeighborsClassifier nearest neighbors classifier

Notes

When used for text classification with tf-idf vectors, this classifier is also known as the Rocchio classifier.

References

Tibshirani, R., Hastie, T., Narasimhan, B., & Chu, G. (2002). Diagnosis of multiple cancer types by shrunken centroids of gene expression. Proceedings of the National Academy of Sciences of the United States of America, 99(10), 6567-6572. The National Academy of Sciences.

Examples

```
>>> from sklearn.neighbors import NearestCentroid
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> y = np.array([1, 1, 1, 2, 2, 2])
>>> clf = NearestCentroid()
>>> clf.fit(X, y)
NearestCentroid()
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

fit(self, X, y)	Fit the NearestCentroid model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Perform classification on an array of test vectors X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *metric='euclidean'*, *shrink_threshold=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Fit the NearestCentroid model according to the given training data.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features. Note that centroid shrinking cannot be used with sparse matrices.

y [array, shape = [n_samples]] Target values (integers)

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Perform classification on an array of test vectors X.

The predicted class C for each sample in X is returned.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

C [ndarray of shape (n_samples,)]

Notes

If the metric constructor parameter is "precomputed", X is assumed to be the distance matrix between the data to be predicted and `self.centroids_`.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.neighbors.NearestCentroid`

- *Nearest Centroid Classification*
- *Classification of text documents using sparse features*

7.30.13 `sklearn.neighbors`.NearestNeighbors

class sklearn.neighbors.**NearestNeighbors**(*n_neighbors=5*, *radius=1.0*, *algorithm='auto'*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *n_jobs=None*)
Unsupervised learner for implementing neighbor searches.

Read more in the *User Guide*.

New in version 0.9.

Parameters

n_neighbors [int, optional (default = 5)] Number of neighbors to use by default for *kneighbors* queries.

radius [float, optional (default = 1.0)] Range of parameter space to use by default for *radius_neighbors* queries.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use *BallTree*
- 'kd_tree' will use *KDTree*
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to *fit* method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, optional (default = 30)] Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

metric [string or callable, default 'minkowski'] the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics. If metric is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a *Glossary*, in which case only "nonzero" elements may be considered neighbors.

p [integer, optional (default = 2)] Parameter for the Minkowski metric from sklearn.metrics.pairwise.pairwise_distances. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params [dict, optional (default = None)] Additional keyword arguments for the metric function.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

effective_metric_ [string] Metric used to compute distances to neighbors.

effective_metric_params_ [dict] Parameters for the metric used to compute distances to neighbors.

See also:

KNeighborsClassifier

RadiusNeighborsClassifier

KNeighborsRegressor

RadiusNeighborsRegressor

BallTree

Notes

See *Nearest Neighbors* in the online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Examples

```
>>> import numpy as np
>>> from sklearn.neighbors import NearestNeighbors
>>> samples = [[0, 0, 2], [1, 0, 0], [0, 0, 1]]
```

```
>>> neigh = NearestNeighbors(2, 0.4)
>>> neigh.fit(samples)
NearestNeighbors(...)
```

```
>>> neigh.kneighbors([[0, 0, 1.3]], 2, return_distance=False)
array([[2, 0]]...)
```

```
>>> nbrs = neigh.radius_neighbors([[0, 0, 1.3]], 0.4, return_distance=False)
>>> np.asarray(nbrs[0][0])
array(2)
```

Methods

fit(self, X[, y])	Fit the model using X as training data
get_params(self[, deep])	Get parameters for this estimator.
kneighbors(self[, X, n_neighbors, ...])	Finds the K-neighbors of a point.
kneighbors_graph(self[, X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
radius_neighbors(self[, X, radius, ...])	Finds the neighbors within a given radius of a point or points.
radius_neighbors_graph(self[, X, radius, ...])	Computes the (weighted) graph of Neighbors for points in X
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_neighbors=5*, *radius=1.0*, *algorithm='auto'*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the model using X as training data

Parameters

X [{array-like, sparse matrix, BallTree, KDTree}] Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

kneighbors (*self*, *X=None*, *n_neighbors=None*, *return_distance=True*)
Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

neigh_dist [array, shape (n_queries, n_neighbors)] Array representing the lengths to points, only present if return_distance=True

neigh_ind [array, shape (n_queries, n_neighbors)] Indices of the nearest points in the population matrix.

Examples

In the following example, we construct a NearestNeighbors class from an array representing our data set and ask who's the closest point to [1,1,1]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(n_neighbors=1)
>>> print(neigh.kneighbors([[1., 1., 1.]]))
(array([[0.5]]), array([[2]]))
```

As you can see, it returns [[0.5]], and [[2]], which means that the element is at distance 0.5 and is the third element of samples (indexes start at 0). You can also query for multiple points:

```
>>> X = [[0., 1., 0.], [1., 0., 1.]]
>>> neigh.kneighbors(X, return_distance=False)
array([[1],
       [2]]...)
```

kneighbors_graph (*self*, *X=None*, *n_neighbors=None*, *mode='connectivity'*)
Computes the (weighted) graph of k-Neighbors for points in X

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors for each sample. (default is value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

NearestNeighbors.radius_neighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

radius_neighbors(*self*, *X=None*, *radius=None*, *return_distance=True*, *sort_results=False*)

Finds the neighbors within a given radius of a point or points.

Return the indices and distances of each point from the dataset lying in a ball with size `radius` around the points of the query array. Points lying on the boundary are included in the results.

The result points are *not* necessarily sorted by distance to their query point.

Parameters

X [array-like, (n_samples, n_features), optional] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Limiting distance of neighbors to return. (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned.

sort_results [boolean, optional. Defaults to False.] If True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. If return_distance == False, setting sort_results = True will result in an error.

New in version 0.22.

Returns

neigh_dist [array, shape (n_samples,) of arrays] Array representing the distances to each point, only present if return_distance=True. The distance values are computed according to the `metric` constructor parameter.

neigh_ind [array, shape (n_samples,) of arrays] An array of arrays of indices of the approximate nearest points from the population matrix that lie within a ball of size `radius` around the query points.

Notes

Because the number of neighbors of each point is not necessarily equal, the results for multiple query points cannot be fit in a standard data array. For efficiency, `radius_neighbors` returns arrays of objects, where each object is a 1D array of indices or distances.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1, 1, 1]:

```
>>> import numpy as np
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.6)
>>> neigh.fit(samples)
NearestNeighbors(radius=1.6)
>>> rng = neigh.radius_neighbors([[1., 1., 1.]])
>>> print(np.asarray(rng[0][0]))
[1.5 0.5]
>>> print(np.asarray(rng[1][0]))
[1 2]
```

The first array returned contains the distances to all points which are closer than 1.6, while the second array returned contains their indices. In general, multiple points can be queried at the same time.

radius_neighbors_graph(*self*, *X=None*, *radius=None*, *mode='connectivity'*, *sort_results=False*)
Computes the (weighted) graph of Neighbors for points in X

Neighborhoods are restricted the points at a distance lower than radius.

Parameters

X [array-like of shape (n_samples, n_features), default=None] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Radius of neighborhoods. (default is the value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

sort_results [boolean, optional. Defaults to False.] If True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. Only used with mode='distance'.

New in version 0.22.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

kneighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.5)
>>> neigh.fit(X)
NearestNeighbors(radius=1.5)
>>> A = neigh.radius_neighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 0.],
       [1., 0., 1.]])
```

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

7.30.14 sklearn.neighbors.NeighborhoodComponentsAnalysis

class sklearn.neighbors.**NeighborhoodComponentsAnalysis**(*n_components=None*, *init='auto'*, *warm_start=False*, *max_iter=50*, *tol=1e-05*, *callback=None*, *verbose=0*, *random_state=None*)

Neighborhood Components Analysis

Neighborhood Component Analysis (NCA) is a machine learning algorithm for metric learning. It learns a linear transformation in a supervised fashion to improve the classification accuracy of a stochastic nearest neighbors rule in the transformed space.

Read more in the *User Guide*.

Parameters

n_components [int, optional (default=None)] Preferred dimensionality of the projected space. If None it will be set to `n_features`.

init [string or numpy array, optional (default='auto')] Initialization of the linear transformation. Possible options are 'auto', 'pca', 'lda', 'identity', 'random', and a numpy array of shape (n_features_a, n_features_b).

'auto' Depending on `n_components`, the most reasonable initialization will be chosen. If `n_components <= n_classes` we use 'lda', as it uses labels information. If not, but `n_components < min(n_features, n_samples)`, we use 'pca', as it projects data in meaningful directions (those of higher variance). Otherwise, we just use 'identity'.

'pca' `n_components` principal components of the inputs passed to *fit* will be used to initialize the transformation. (See *PCA*)

'lda' `min(n_components, n_classes)` most discriminative components of the inputs passed to *fit* will be used to initialize the transformation. (If `n_components > n_classes`, the rest of the components will be zero.) (See *LinearDiscriminantAnalysis*)

'identity' If `n_components` is strictly smaller than the dimensionality of the inputs passed to *fit*, the identity matrix will be truncated to the first `n_components` rows.

'random' The initial transformation will be a random array of shape `(n_components, n_features)`. Each value is sampled from the standard normal distribution.

numpy array n_features_b must match the dimensionality of the inputs passed to *fit* and n_features_a must be less than or equal to that. If `n_components` is not None, n_features_a must match it.

warm_start [bool, optional, (default=False)] If True and *fit* has been called before, the solution of the previous call to *fit* is used as the initial linear transformation (`n_components` and `init` will be ignored).

max_iter [int, optional (default=50)] Maximum number of iterations in the optimization.

tol [float, optional (default=1e-5)] Convergence tolerance for the optimization.

callback [callable, optional (default=None)] If not None, this function is called after every iteration of the optimizer, taking as arguments the current solution (flattened transformation matrix) and the number of iterations. This might be useful in case one wants to examine or store the transformation found after each iteration.

verbose [int, optional (default=0)] If 0, no progress messages will be printed. If 1, progress messages will be printed to stdout. If > 1, progress messages will be printed and the `disp` parameter of `scipy.optimize.minimize` will be set to `verbose - 2`.

random_state [int or numpy.RandomState or None, optional (default=None)] A pseudo random number generator object or a seed for it if int. If `init='random'`, `random_state` is used to initialize the random transformation. If `init='pca'`, `random_state` is passed as an argument to PCA when initializing the transformation.

Attributes

components_ [array, shape (n_components, n_features)] The linear transformation learned during fitting.

n_iter_ [int] Counts the number of iterations performed by the optimizer.

random_state_ [numpy.RandomState] Pseudo random number generator object used during initialization.

References

[Rf9b6baee8229-1], [Rf9b6baee8229-2]

Examples

```
>>> from sklearn.neighbors import NeighborhoodComponentsAnalysis
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
... stratify=y, test_size=0.7, random_state=42)
>>> nca = NeighborhoodComponentsAnalysis(random_state=42)
>>> nca.fit(X_train, y_train)
NeighborhoodComponentsAnalysis(...)
>>> knn = KNeighborsClassifier(n_neighbors=3)
>>> knn.fit(X_train, y_train)
KNeighborsClassifier(...)
>>> print(knn.score(X_test, y_test))
0.933333...
>>> knn.fit(nca.transform(X_train), y_train)
KNeighborsClassifier(...)
>>> print(knn.score(nca.transform(X_test), y_test))
0.961904...
```

Methods

fit(self, X, y)	Fit the model according to the given training data.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Applies the learned transformation to the given data.

__init__(*self*, *n_components=None*, *init='auto'*, *warm_start=False*, *max_iter=50*, *tol=1e-05*, *callback=None*, *verbose=0*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)
Fit the model according to the given training data.

Parameters

X [array-like, shape (n_samples, n_features)] The training samples.

y [array-like, shape (n_samples,)] The corresponding training labels.

Returns

self [object] returns a trained NeighborhoodComponentsAnalysis model.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Applies the learned transformation to the given data.

Parameters

X [array-like, shape (n_samples, n_features)] Data samples.

Returns

X_embedded: array, shape (n_samples, n_components) The data samples transformed.

Raises

NotFittedError If *fit* has not been called before.

Examples using `sklearn.neighbors.NeighborhoodComponentsAnalysis`

- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*
- *Neighborhood Components Analysis Illustration*
- *Comparing Nearest Neighbors with and without Neighborhood Components Analysis*
- *Dimensionality Reduction with Neighborhood Components Analysis*

neighbors.kneighbors_graph(X, n_neighbors[, ...])	Computes the (weighted) graph of k-Neighbors for points in X
neighbors.radius_neighbors_graph(X, radius)	Computes the (weighted) graph of Neighbors for points in X

7.30.15 sklearn.neighbors.kneighbors_graph

sklearn.neighbors.**kneighbors_graph**(*X*, *n_neighbors*, *mode='connectivity'*, *metric='minkowski'*, *p=2*, *metric_params=None*, *include_self=False*, *n_jobs=None*)

Computes the (weighted) graph of k-Neighbors for points in X

Read more in the *User Guide*.

Parameters

X [array-like of shape (n_samples, n_features) or BallTree] Sample data, in the form of a numpy array or a precomputed *BallTree*.

n_neighbors [int] Number of neighbors for each sample.

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, and 'distance' will return the distances between neighbors according to the given metric.

metric [string, default 'minkowski'] The distance metric used to calculate the k-Neighbors for each sample point. The DistanceMetric class gives a list of available metrics. The default distance is 'euclidean' ('minkowski' metric with the p param equal to 2.)

p [int, default 2] Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params [dict, optional] additional keyword arguments for the metric function.

include_self [bool or 'auto', default=False] Whether or not to mark each sample as the first nearest neighbor to itself. If 'auto', then True is used for mode='connectivity' and False for mode='distance'.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Returns

A [sparse graph in CSR format, shape = [n_samples, n_samples]] A[i, j] is assigned the weight of edge that connects i to j.

See also:

radius_neighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import kneighbors_graph
>>> A = kneighbors_graph(X, 2, mode='connectivity', include_self=True)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

Examples using `sklearn.neighbors.kneighbors_graph`

- *Agglomerative clustering with and without structure*
- *Hierarchical clustering: structured vs unstructured ward*
- *Comparing different clustering algorithms on toy datasets*

7.30.16 `sklearn.neighbors`.radius_neighbors_graph

`sklearn.neighbors.`**`radius_neighbors_graph`**(*X*, *radius*, *mode='connectivity'*, *metric='minkowski'*, *p=2*, *metric_params=None*, *include_self=False*, *n_jobs=None*)

Computes the (weighted) graph of Neighbors for points in X

Neighborhoods are restricted the points at a distance lower than radius.

Read more in the *User Guide*.

Parameters

X [array-like of shape (n_samples, n_features) or BallTree] Sample data, in the form of a numpy array or a precomputed `BallTree`.

radius [float] Radius of neighborhoods.

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, and 'distance' will return the distances between neighbors according to the given metric.

metric [string, default 'minkowski'] The distance metric used to calculate the neighbors within a given radius for each sample point. The DistanceMetric class gives a list of available metrics. The default distance is 'euclidean' ('minkowski' metric with the param equal to 2.)

p [int, default 2] Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params [dict, optional] additional keyword arguments for the metric function.

include_self [bool or 'auto', default=False] Whether or not to mark each sample as the first nearest neighbor to itself. If 'auto', then True is used for mode='connectivity' and False for mode='distance'.

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Returns

A [sparse graph in CSR format, shape = [n_samples, n_samples]] A[i, j] is assigned the weight of edge that connects i to j.

See also:

kneighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import radius_neighbors_graph
>>> A = radius_neighbors_graph(X, 1.5, mode='connectivity',
...                            include_self=True)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 0.],
       [1., 0., 1.]])
```

7.31 sklearn.neural_network: Neural network models

The *sklearn.neural_network* module includes models based on neural networks.

User guide: See the *Neural network models (supervised)* and *Neural network models (unsupervised)* sections for further details.

neural_network.BernoulliRBM([n_components, ...])	Bernoulli Restricted Boltzmann Machine (RBM).
neural_network.MLPClassifier([...])	Multi-layer Perceptron classifier.
neural_network.MLPRegressor([...])	Multi-layer Perceptron regressor.

7.31.1 sklearn.neural_network.BernoulliRBM

class sklearn.neural_network.**BernoulliRBM**(*n_components=256*, *learning_rate=0.1*, *batch_size=10*, *n_iter=10*, *verbose=0*, *random_state=None*)

Bernoulli Restricted Boltzmann Machine (RBM).

A Restricted Boltzmann Machine with binary visible units and binary hidden units. Parameters are estimated using Stochastic Maximum Likelihood (SML), also known as Persistent Contrastive Divergence (PCD) [2].

The time complexity of this implementation is `O(d ** 2)` assuming d ~ n_features ~ n_components.

Read more in the *User Guide*.

Parameters

n_components [int, default=256] Number of binary hidden units.

learning_rate [float, default=0.1] The learning rate for weight updates. It is *highly* recommended to tune this hyper-parameter. Reasonable values are in the 10**[0., -3.] range.

batch_size [int, default=10] Number of examples per minibatch.

n_iter [int, default=10] Number of iterations/sweeps over the training dataset to perform during training.

verbose [int, default=0] The verbosity level. The default, zero, means silent mode.

random_state [integer or RandomState, default=None] A random number generator instance to define the state of the random permutations generator. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

Attributes

intercept_hidden_ [array-like, shape (n_components,)] Biases of the hidden units.

intercept_visible_ [array-like, shape (n_features,)] Biases of the visible units.

components_ [array-like, shape (n_components, n_features)] Weight matrix, where n_features in the number of visible units and n_components is the number of hidden units.

h_samples_ [array-like, shape (batch_size, n_components)] Hidden Activation sampled from the model distribution, where batch_size in the number of examples per minibatch and n_components is the number of hidden units.

References

[1] Hinton, G. E., Osindero, S. and Teh, Y. A fast learning algorithm for deep belief nets. Neural Computation 18, pp 1527-1554. https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf

[2] Tieleman, T. Training Restricted Boltzmann Machines using Approximations to the Likelihood Gradient. International Conference on Machine Learning (ICML) 2008

Examples

```
>>> import numpy as np
>>> from sklearn.neural_network import BernoulliRBM
>>> X = np.array([[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 1]])
>>> model = BernoulliRBM(n_components=2)
>>> model.fit(X)
BernoulliRBM(n_components=2)
```

Methods

fit(self, X[, y])	Fit the model to the data X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
gibbs(self, v)	Perform one Gibbs sampling step.
partial_fit(self, X[, y])	Fit the model to the data X which should contain a partial segment of the data.
score_samples(self, X)	Compute the pseudo-likelihood of X.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Compute the hidden layer activation probabilities, P(h=1|v=X).

__init__(*self*, *n_components=256*, *learning_rate=0.1*, *batch_size=10*, *n_iter=10*, *verbose=0*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the model to the data X.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training data.

Returns

self [BernoulliRBM] The fitted model.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

gibbs(*self*, *v*)
Perform one Gibbs sampling step.

Parameters

v [ndarray of shape (n_samples, n_features)] Values of the visible layer to start from.

Returns

v_new [ndarray of shape (n_samples, n_features)] Values of the visible layer after one Gibbs step.

partial_fit(*self*, *X*, *y=None*)
Fit the model to the data X which should contain a partial segment of the data.

Parameters

X [ndarray of shape (n_samples, n_features)] Training data.

Returns

self [BernoulliRBM] The fitted model.

score_samples(*self*, *X*)
Compute the pseudo-likelihood of X.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Values of the visible layer. Must be all-boolean (not checked).

Returns

pseudo_likelihood [ndarray of shape (n_samples,)] Value of the pseudo-likelihood (proxy for likelihood).

Notes

This method is not deterministic: it computes a quantity called the free energy on X, then on a randomly corrupted version of X, and returns the log of the logistic function of the difference.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Compute the hidden layer activation probabilities, P(h=1|v=X).

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The data to be transformed.

Returns

h [ndarray of shape (n_samples, n_components)] Latent representations of the data.

Examples using `sklearn.neural_network.BernoulliRBM`

- *Restricted Boltzmann Machine features for digit classification*

7.31.2 `sklearn.neural_network`.MLPClassifier

class sklearn.neural_network.**MLPClassifier**(*hidden_layer_sizes=(100,)*, *activation='relu'*, *solver='adam'*, *alpha=0.0001*, *batch_size='auto'*, *learning_rate='constant'*, *learning_rate_init=0.001*, *power_t=0.5*, *max_iter=200*, *shuffle=True*, *random_state=None*, *tol=0.0001*, *verbose=False*, *warm_start=False*, *momentum=0.9*, *nesterovs_momentum=True*, *early_stopping=False*, *validation_fraction=0.1*, *beta_1=0.9*, *beta_2=0.999*, *epsilon=1e-08*, *n_iter_no_change=10*, *max_fun=15000*)
Multi-layer Perceptron classifier.

This model optimizes the log-loss function using LBFGS or stochastic gradient descent.

New in version 0.18.

Parameters

hidden_layer_sizes [tuple, length = n_layers - 2, default=(100,)] The ith element represents the number of neurons in the ith hidden layer.

activation [{'identity', 'logistic', 'tanh', 'relu'}, default='relu'] Activation function for the hidden layer.

- 'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x
- 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
- 'tanh', the hyperbolic tan function, returns f(x) = tanh(x).
- 'relu', the rectified linear unit function, returns f(x) = max(0, x)

solver [{'lbfgs', 'sgd', 'adam'}, default='adam'] The solver for weight optimization.

- 'lbfgs' is an optimizer in the family of quasi-Newton methods.
- 'sgd' refers to stochastic gradient descent.
- 'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba

Note: The default solver 'adam' works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, 'lbfgs' can converge faster and perform better.

alpha [float, default=0.0001] L2 penalty (regularization term) parameter.

batch_size [int, default='auto'] Size of minibatches for stochastic optimizers. If the solver is 'lbfgs', the classifier will not use minibatch. When set to "auto", `batch_size=min(200, n_samples)`

learning_rate [{'constant', 'invscaling', 'adaptive'}, default='constant'] Learning rate schedule for weight updates.

- 'constant' is a constant learning rate given by 'learning_rate_init'.
- 'invscaling' gradually decreases the learning rate at each time step 't' using an inverse scaling exponent of 'power_t'. effective_learning_rate = learning_rate_init / pow(t, power_t)
- 'adaptive' keeps the learning rate constant to 'learning_rate_init' as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5.

Only used when `solver='sgd'`.

learning_rate_init [double, default=0.001] The initial learning rate used. It controls the stepsize in updating the weights. Only used when solver='sgd' or 'adam'.

power_t [double, default=0.5] The exponent for inverse scaling learning rate. It is used in updating effective learning rate when the learning_rate is set to 'invscaling'. Only used when solver='sgd'.

max_iter [int, default=200] Maximum number of iterations. The solver iterates until convergence (determined by 'tol') or this number of iterations. For stochastic solvers ('sgd', 'adam'), note that this determines the number of epochs (how many times each data point will be used), not the number of gradient steps.

shuffle [bool, default=True] Whether to shuffle samples in each iteration. Only used when solver='sgd' or 'adam'.

random_state [int, RandomState instance or None, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the

random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

tol [float, default=1e-4] Tolerance for the optimization. When the loss or score is not improving by at least `tol` for `n_iter_no_change` consecutive iterations, unless `learning_rate` is set to 'adaptive', convergence is considered to be reached and training stops.

verbose [bool, default=False] Whether to print progress messages to stdout.

warm_start [bool, default=False] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

momentum [float, default=0.9] Momentum for gradient descent update. Should be between 0 and 1. Only used when solver='sgd'.

nesterovs_momentum [boolean, default=True] Whether to use Nesterov's momentum. Only used when solver='sgd' and momentum > 0.

early_stopping [bool, default=False] Whether to use early stopping to terminate training when validation score is not improving. If set to true, it will automatically set aside 10% of training data as validation and terminate training when validation score is not improving by at least tol for `n_iter_no_change` consecutive epochs. The split is stratified, except in a multilabel setting. Only effective when solver='sgd' or 'adam'

validation_fraction [float, default=0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True

beta_1 [float, default=0.9] Exponential decay rate for estimates of first moment vector in adam, should be in [0, 1). Only used when solver='adam'

beta_2 [float, default=0.999] Exponential decay rate for estimates of second moment vector in adam, should be in [0, 1). Only used when solver='adam'

epsilon [float, default=1e-8] Value for numerical stability in adam. Only used when solver='adam'

n_iter_no_change [int, default=10] Maximum number of epochs to not meet `tol` improvement. Only effective when solver='sgd' or 'adam'

New in version 0.20.

max_fun [int, default=15000] Only used when solver='lbfgs'. Maximum number of loss function calls. The solver iterates until convergence (determined by 'tol'), number of iterations reaches max_iter, or this number of loss function calls. Note that number of loss function calls will be greater than or equal to the number of iterations for the `MLPClassifier`.

New in version 0.22.

Attributes

classes_ [ndarray or list of ndarray of shape (n_classes,)] Class labels for each output.

loss_ [float] The current loss computed with the loss function.

coefs_ [list, length n_layers - 1] The ith element in the list represents the weight matrix corresponding to layer i.

intercepts_ [list, length n_layers - 1] The ith element in the list represents the bias vector corresponding to layer i + 1.

n_iter_ [int,] The number of iterations the solver has ran.

n_layers_ [int] Number of layers.

n_outputs_ [int] Number of outputs.

out_activation_ [string] Name of the output activation function.

Notes

MLPClassifier trains iteratively since at each time step the partial derivatives of the loss function with respect to the model parameters are computed to update the parameters.

It can also have a regularization term added to the loss function that shrinks model parameters to prevent overfitting.

This implementation works with data represented as dense numpy arrays or sparse scipy arrays of floating point values.

References

Hinton, Geoffrey E. "Connectionist learning procedures." Artificial intelligence 40.1 (1989): 185-234.

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." International Conference on Artificial Intelligence and Statistics. 2010.

He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." arXiv preprint arXiv:1502.01852 (2015).

Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

Methods

fit(self, X, y)	Fit the model to data matrix X and target(s) y.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the multi-layer perceptron classifier
predict_log_proba(self, X)	Return the log of probability estimates.
predict_proba(self, X)	Probability estimates.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *hidden_layer_sizes=(100,)*, *activation='relu'*, *solver='adam'*, *alpha=0.0001*, *batch_size='auto'*, *learning_rate='constant'*, *learning_rate_init=0.001*, *power_t=0.5*, *max_iter=200*, *shuffle=True*, *random_state=None*, *tol=0.0001*, *verbose=False*, *warm_start=False*, *momentum=0.9*, *nesterovs_momentum=True*, *early_stopping=False*, *validation_fraction=0.1*, *beta_1=0.9*, *beta_2=0.999*, *epsilon=1e-08*, *n_iter_no_change=10*, *max_fun=15000*)

Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)

Fit the model to data matrix X and target(s) y.

Parameters

X [ndarray or sparse matrix of shape (n_samples, n_features)] The input data.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels in classification, real numbers in regression).

Returns

self [returns a trained MLP model.]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property partial_fit
Update the model with a single iteration over the given data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

y [array-like, shape (n_samples,)] The target values.

classes [array, shape (n_classes), default None] Classes across all calls to partial_fit. Can be obtained via `np.unique(y_all)`, where y_all is the target vector of the entire dataset. This argument is required for the first call to partial_fit and can be omitted in the subsequent calls. Note that y doesn't need to contain all labels in `classes`.

Returns

self [returns a trained MLP model.]

predict (*self*, *X*)
Predict using the multi-layer perceptron classifier

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input data.

Returns

y [ndarray, shape (n_samples,) or (n_samples, n_classes)] The predicted classes.

predict_log_proba (*self*, *X*)
Return the log of probability estimates.

Parameters

X [ndarray of shape (n_samples, n_features)] The input data.

Returns

log_y_prob [ndarray of shape (n_samples, n_classes)] The predicted log-probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`. Equivalent to log(predict_proba(X))

predict_proba (*self*, *X*)
Probability estimates.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input data.

Returns

y_prob [ndarray of shape (n_samples, n_classes)] The predicted probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.neural_network.MLPClassifier`

- *Classifier comparison*
- *Visualization of MLP weights on MNIST*
- *Varying regularization in Multi-layer Perceptron*
- *Compare Stochastic learning strategies for MLPClassifier*

7.31.3 `sklearn.neural_network`.MLPRegressor

class sklearn.neural_network.**MLPRegressor**(*hidden_layer_sizes=(100,)*, *activation='relu'*, *solver='adam'*, *alpha=0.0001*, *batch_size='auto'*, *learning_rate='constant'*, *learning_rate_init=0.001*, *power_t=0.5*, *max_iter=200*, *shuffle=True*, *random_state=None*, *tol=0.0001*, *verbose=False*, *warm_start=False*, *momentum=0.9*, *nesterovs_momentum=True*, *early_stopping=False*, *validation_fraction=0.1*, *beta_1=0.9*, *beta_2=0.999*, *epsilon=1e-08*, *n_iter_no_change=10*, *max_fun=15000*)

Multi-layer Perceptron regressor.

This model optimizes the squared-loss using LBFGS or stochastic gradient descent.

New in version 0.18.

Parameters

hidden_layer_sizes [tuple, length = n_layers - 2, default=(100,)] The ith element represents the number of neurons in the ith hidden layer.

activation [{'identity', 'logistic', 'tanh', 'relu'}, default='relu'] Activation function for the hidden layer.

- 'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x
- 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
- 'tanh', the hyperbolic tan function, returns f(x) = tanh(x).
- 'relu', the rectified linear unit function, returns f(x) = max(0, x)

solver [{'lbfgs', 'sgd', 'adam'}, default='adam'] The solver for weight optimization.

- 'lbfgs' is an optimizer in the family of quasi-Newton methods.
- 'sgd' refers to stochastic gradient descent.
- 'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba

Note: The default solver 'adam' works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, 'lbfgs' can converge faster and perform better.

alpha [float, default=0.0001] L2 penalty (regularization term) parameter.

batch_size [int, default='auto'] Size of minibatches for stochastic optimizers. If the solver is 'lbfgs', the classifier will not use minibatch. When set to "auto", `batch_size=min(200, n_samples)`

learning_rate [{'constant', 'invscaling', 'adaptive'}, default='constant'] Learning rate schedule for weight updates.

- 'constant' is a constant learning rate given by 'learning_rate_init'.
- 'invscaling' gradually decreases the learning rate `learning_rate_` at each time step 't' using an inverse scaling exponent of 'power_t'. effective_learning_rate = learning_rate_init / pow(t, power_t)
- 'adaptive' keeps the learning rate constant to 'learning_rate_init' as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5.

Only used when solver='sgd'.

learning_rate_init [double, default=0.001] The initial learning rate used. It controls the step-size in updating the weights. Only used when solver='sgd' or 'adam'.

power_t [double, default=0.5] The exponent for inverse scaling learning rate. It is used in updating effective learning rate when the learning_rate is set to 'invscaling'. Only used when solver='sgd'.

max_iter [int, default=200] Maximum number of iterations. The solver iterates until convergence (determined by 'tol') or this number of iterations. For stochastic solvers ('sgd', 'adam'), note that this determines the number of epochs (how many times each data point will be used), not the number of gradient steps.

shuffle [bool, default=True] Whether to shuffle samples in each iteration. Only used when solver='sgd' or 'adam'.

random_state [int, RandomState instance or None, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

tol [float, default=1e-4] Tolerance for the optimization. When the loss or score is not improving by at least `tol` for `n_iter_no_change` consecutive iterations, unless `learning_rate` is set to 'adaptive', convergence is considered to be reached and training stops.

verbose [bool, default=False] Whether to print progress messages to stdout.

warm_start [bool, default=False] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

momentum [float, default=0.9] Momentum for gradient descent update. Should be between 0 and 1. Only used when solver='sgd'.

nesterovs_momentum [boolean, default=True] Whether to use Nesterov's momentum. Only used when solver='sgd' and momentum > 0.

early_stopping [bool, default=False] Whether to use early stopping to terminate training when validation score is not improving. If set to true, it will automatically set aside 10% of training data as validation and terminate training when validation score is not improving by at least `tol` for `n_iter_no_change` consecutive epochs. Only effective when solver='sgd' or 'adam'

validation_fraction [float, default=0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True

beta_1 [float, default=0.9] Exponential decay rate for estimates of first moment vector in adam, should be in [0, 1). Only used when solver='adam'

beta_2 [float, default=0.999] Exponential decay rate for estimates of second moment vector in adam, should be in [0, 1). Only used when solver='adam'

epsilon [float, default=1e-8] Value for numerical stability in adam. Only used when solver='adam'

n_iter_no_change [int, default=10] Maximum number of epochs to not meet `tol` improvement. Only effective when solver='sgd' or 'adam'

New in version 0.20.

max_fun [int, default=15000] Only used when solver='lbfgs'. Maximum number of function calls. The solver iterates until convergence (determined by 'tol'), number of iterations reaches max_iter, or this number of function calls. Note that number of function calls will be greater than or equal to the number of iterations for the MLPRegressor.

New in version 0.22.

Attributes

loss_ [float] The current loss computed with the loss function.

coefs_ [list, length n_layers - 1] The ith element in the list represents the weight matrix corresponding to layer i.

intercepts_ [list, length n_layers - 1] The ith element in the list represents the bias vector corresponding to layer i + 1.

n_iter_ [int,] The number of iterations the solver has ran.

n_layers_ [int] Number of layers.

n_outputs_ [int] Number of outputs.

out_activation_ [string] Name of the output activation function.

Notes

MLPRegressor trains iteratively since at each time step the partial derivatives of the loss function with respect to the model parameters are computed to update the parameters.

It can also have a regularization term added to the loss function that shrinks model parameters to prevent overfitting.

This implementation works with data represented as dense and sparse numpy arrays of floating point values.

References

Hinton, Geoffrey E. "Connectionist learning procedures." Artificial intelligence 40.1 (1989): 185-234.

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." International Conference on Artificial Intelligence and Statistics. 2010.

He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." arXiv preprint arXiv:1502.01852 (2015).

Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

Methods

fit(self, X, y)	Fit the model to data matrix X and target(s) y.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the multi-layer perceptron model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *hidden_layer_sizes=(100,)*, *activation='relu'*, *solver='adam'*, *alpha=0.0001*, *batch_size='auto'*, *learning_rate='constant'*, *learning_rate_init=0.001*, *power_t=0.5*, *max_iter=200*, *shuffle=True*, *random_state=None*, *tol=0.0001*, *verbose=False*, *warm_start=False*, *momentum=0.9*, *nesterovs_momentum=True*, *early_stopping=False*, *validation_fraction=0.1*, *beta_1=0.9*, *beta_2=0.999*, *epsilon=1e-08*, *n_iter_no_change=10*, *max_fun=15000*)

Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)

Fit the model to data matrix X and target(s) y.

Parameters

X [ndarray or sparse matrix of shape (n_samples, n_features)] The input data.

y [ndarray of shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels in classification, real numbers in regression).

Returns

self [returns a trained MLP model.]

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property partial_fit

Update the model with a single iteration over the given data.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input data.

y [ndarray of shape (n_samples,)] The target values.

Returns

self [returns a trained MLP model.]

predict (*self*, *X*)

Predict using the multi-layer perceptron model.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input data.

Returns

y [ndarray of shape (n_samples, n_outputs)] The predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.neural_network.MLPRegressor`

- *Advanced Plotting With Partial Dependence*
- *Partial Dependence Plots*

7.32 `sklearn.pipeline`: Pipeline

The *sklearn.pipeline* module implements utilities to build a composite estimator, as a chain of transforms and estimators.

pipeline.FeatureUnion(transformer_list[, ...])	Concatenates results of multiple transformer objects.
pipeline.Pipeline(steps[, memory, verbose])	Pipeline of transforms with a final estimator.

7.32.1 `sklearn.pipeline`.FeatureUnion

class sklearn.pipeline.**FeatureUnion**(*transformer_list*, *n_jobs=None*, *transformer_weights=None*, *verbose=False*)
Concatenates results of multiple transformer objects.

This estimator applies a list of transformer objects in parallel to the input data, then concatenates the results. This is useful to combine several feature extraction mechanisms into a single transformer.

Parameters of the transformers may be set using its name and the parameter name separated by a '__'. A transformer may be replaced entirely by setting the parameter with its name to another transformer, or removed by setting to 'drop'.

Read more in the *User Guide*.

New in version 0.13.

Parameters

transformer_list [list of (string, transformer) tuples] List of transformer objects to be applied to the data. The first half of each tuple is the name of the transformer.

Changed in version 0.22: Deprecated `None` as a transformer in favor of 'drop'.

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

transformer_weights [dict, optional] Multiplicative weights for features per transformer. Keys are transformer names, values the weights.

verbose [boolean, optional(default=False)] If True, the time elapsed while fitting each transformer will be printed as it is completed.

See also:

sklearn.pipeline.make_union Convenience function for simplified feature union construction.

Examples

```
>>> from sklearn.pipeline import FeatureUnion
>>> from sklearn.decomposition import PCA, TruncatedSVD
>>> union = FeatureUnion([("pca", PCA(n_components=1)),
...                       ("svd", TruncatedSVD(n_components=2))])
>>> X = [[0., 1., 3], [2., 2., 5]]
>>> union.fit_transform(X)
array([[ 1.5       ,  3.0...,  0.8...],
       [-1.5       ,  5.7..., -0.4...]])
```

Methods

fit(self, X[, y])	Fit all transformers using X.
fit_transform(self, X[, y])	Fit all transformers, transform the data and concatenate results.
get_feature_names(self)	Get feature names from all transformers.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **kwargs)	Set the parameters of this estimator.
transform(self, X)	Transform X separately by each transformer, concatenate results.

__init__ (*self*, *transformer_list*, *n_jobs=None*, *transformer_weights=None*, *verbose=False*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*, ***fit_params*)
Fit all transformers using X.

Parameters

X [iterable or array-like, depending on transformers] Input data, used to fit transformers.

y [array-like, shape (n_samples, ...), optional] Targets for supervised learning.

Returns

self [FeatureUnion] This estimator

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit all transformers, transform the data and concatenate results.

Parameters

X [iterable or array-like, depending on transformers] Input data to be transformed.

y [array-like, shape (n_samples, ...), optional] Targets for supervised learning.

Returns

X_t [array-like or sparse matrix, shape (n_samples, sum_n_components)] hstack of results of transformers. sum_n_components is the sum of n_components (output dimension) over transformers.

get_feature_names(*self*)
Get feature names from all transformers.

Returns

feature_names [list of strings] Names of the features produced by transform.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***kwargs*)
Set the parameters of this estimator.

Valid parameter keys can be listed with `get_params()`.

Returns

self

transform(*self*, *X*)
Transform X separately by each transformer, concatenate results.

Parameters

X [iterable or array-like, depending on transformers] Input data to be transformed.

Returns

X_t [array-like or sparse matrix, shape (n_samples, sum_n_components)] hstack of results of transformers. sum_n_components is the sum of n_components (output dimension) over transformers.

Examples using `sklearn.pipeline.FeatureUnion`

- *Concatenating multiple feature extraction methods*

7.32.2 sklearn.pipeline.Pipeline

class sklearn.pipeline.**Pipeline**(*steps*, *memory=None*, *verbose=False*)

Pipeline of transforms with a final estimator.

Sequentially apply a list of transforms and a final estimator. Intermediate steps of the pipeline must be 'transforms', that is, they must implement fit and transform methods. The final estimator only needs to implement fit. The transformers in the pipeline can be cached using `memory` argument.

The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters. For this, it enables setting parameters of the various steps using their names and the parameter name separated by a '__', as in the example below. A step's estimator may be replaced entirely by setting the parameter with its name to another estimator, or a transformer removed by setting it to 'passthrough' or `None`.

Read more in the *User Guide*.

New in version 0.5.

Parameters

steps [list] List of (name, transform) tuples (implementing fit/transform) that are chained, in the order in which they are chained, with the last object an estimator.

memory [None, str or object with the joblib.Memory interface, optional] Used to cache the fitted transformers of the pipeline. By default, no caching is performed. If a string is given, it is the path to the caching directory. Enabling caching triggers a clone of the transformers before fitting. Therefore, the transformer instance given to the pipeline cannot be inspected directly. Use the attribute `named_steps` or `steps` to inspect estimators within the pipeline. Caching the transformers is advantageous when fitting is time consuming.

verbose [bool, default=False] If True, the time elapsed while fitting each step will be printed as it is completed.

Attributes

named_steps [bunch object, a dictionary with attribute access] Read-only attribute to access any step parameter by user given name. Keys are step names and values are steps parameters.

See also:

sklearn.pipeline.make_pipeline Convenience function for simplified pipeline construction.

Examples

```
>>> from sklearn import svm
>>> from sklearn.datasets import make_classification
>>> from sklearn.feature_selection import SelectKBest
>>> from sklearn.feature_selection import f_regression
>>> from sklearn.pipeline import Pipeline
>>> # generate some data to play with
>>> X, y = make_classification(
...     n_informative=5, n_redundant=0, random_state=42)
>>> # ANOVA SVM-C
>>> anova_filter = SelectKBest(f_regression, k=5)
>>> clf = svm.SVC(kernel='linear')
>>> anova_svm = Pipeline([('anova', anova_filter), ('svc', clf)])
>>> # You can set the parameters using the names issued
>>> # For instance, fit using a k of 10 in the SelectKBest
>>> # and a parameter 'C' of the svm
```

(continues on next page)

(continued from previous page)

```
>>> anova_svm.set_params(anova__k=10, svc__C=.1).fit(X, y)
Pipeline(steps=[('anova', SelectKBest(...)), ('svc', SVC(...))])
>>> prediction = anova_svm.predict(X)
>>> anova_svm.score(X, y)
0.83
>>> # getting the selected features chosen by anova_filter
>>> anova_svm['anova'].get_support()
array([False, False,  True,  True, False, False,  True,  True, False,
       True, False,  True,  True, False,  True, False,  True,  True,
       False, False])
>>> # Another way to get selected features chosen by anova_filter
>>> anova_svm.named_steps.anova.get_support()
array([False, False,  True,  True, False, False,  True,  True, False,
       True, False,  True,  True, False,  True, False,  True,  True,
       False, False])
>>> # Indexing can also be used to extract a sub-pipeline.
>>> sub_pipeline = anova_svm[:1]
>>> sub_pipeline
Pipeline(steps=[('anova', SelectKBest(...))])
>>> coef = anova_svm[-1].coef_
>>> anova_svm['svc'] is anova_svm[-1]
True
>>> coef.shape
(1, 10)
>>> sub_pipeline.inverse_transform(coef).shape
(1, 20)
```

Methods

decision_function(self, X)	Apply transforms, and decision_function of the final estimator
fit(self, X[, y])	Fit the model
fit_predict(self, X[, y])	Applies fit_predict of last step in pipeline after transforms.
fit_transform(self, X[, y])	Fit the model and transform with the final estimator
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X, **predict_params)	Apply transforms to the data, and predict with the final estimator
predict_log_proba(self, X)	Apply transforms, and predict_log_proba of the final estimator
predict_proba(self, X)	Apply transforms, and predict_proba of the final estimator
score(self, X[, y, sample_weight])	Apply transforms, and score with the final estimator
score_samples(self, X)	Apply transforms, and score_samples of the final estimator.
set_params(self, **kwargs)	Set the parameters of this estimator.

__init__(*self*, *steps*, *memory=None*, *verbose=False*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Apply transforms, and decision_function of the final estimator

Parameters

X [iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

Returns

y_score [array-like of shape (n_samples, n_classes)]

fit (*self*, *X*, *y=None*, ***fit_params*)
Fit the model

Fit all the transforms one after the other and transform the data, then fit the transformed data using the final estimator.

Parameters

X [iterable] Training data. Must fulfill input requirements of first step of the pipeline.

y [iterable, default=None] Training targets. Must fulfill label requirements for all steps of the pipeline.

****fit_params** [dict of string -> object] Parameters passed to the `fit` method of each step, where each parameter name is prefixed such that parameter `p` for step `s` has key `s__p`.

Returns

self [Pipeline] This estimator

fit_predict (*self*, *X*, *y=None*, ***fit_params*)
Applies fit_predict of last step in pipeline after transforms.

Applies fit_transforms of a pipeline to the data, followed by the fit_predict method of the final estimator in the pipeline. Valid only if the final estimator implements fit_predict.

Parameters

X [iterable] Training data. Must fulfill input requirements of first step of the pipeline.

y [iterable, default=None] Training targets. Must fulfill label requirements for all steps of the pipeline.

****fit_params** [dict of string -> object] Parameters passed to the `fit` method of each step, where each parameter name is prefixed such that parameter `p` for step `s` has key `s__p`.

Returns

y_pred [array-like]

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit the model and transform with the final estimator

Fits all the transforms one after the other and transforms the data, then uses fit_transform on transformed data with the final estimator.

Parameters

X [iterable] Training data. Must fulfill input requirements of first step of the pipeline.

y [iterable, default=None] Training targets. Must fulfill label requirements for all steps of the pipeline.

****fit_params** [dict of string -> object] Parameters passed to the `fit` method of each step, where each parameter name is prefixed such that parameter `p` for step `s` has key `s__p`.

Returns

Xt [array-like of shape (n_samples, n_transformed_features)] Transformed samples

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

property inverse_transform
Apply inverse transformations in reverse order

All estimators in the pipeline must support `inverse_transform`.

Parameters

Xt [array-like of shape (n_samples, n_transformed_features)] Data samples, where `n_samples` is the number of samples and `n_features` is the number of features. Must fulfill input requirements of last step of pipeline's `inverse_transform` method.

Returns

Xt [array-like of shape (n_samples, n_features)]

predict(*self*, *X*, ***predict_params*)
Apply transforms to the data, and predict with the final estimator

Parameters

X [iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

****predict_params** [dict of string -> object] Parameters to the `predict` called at the end of all transformations in the pipeline. Note that while this may be used to return uncertainties from some models with return_std or return_cov, uncertainties that are generated by the transformations in the pipeline are not propagated to the final estimator.

Returns

y_pred [array-like]

predict_log_proba(*self*, *X*)
Apply transforms, and predict_log_proba of the final estimator

Parameters

X [iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

Returns

y_score [array-like of shape (n_samples, n_classes)]

predict_proba(*self*, *X*)
Apply transforms, and predict_proba of the final estimator

Parameters

X [iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

Returns

y_proba [array-like of shape (n_samples, n_classes)]

score(*self*, *X*, *y=None*, *sample_weight=None*)
Apply transforms, and score with the final estimator

Parameters

X [iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

y [iterable, default=None] Targets used for scoring. Must fulfill label requirements for all steps of the pipeline.

sample_weight [array-like, default=None] If not None, this argument is passed as `sample_weight` keyword argument to the `score` method of the final estimator.

Returns

score [float]

score_samples(*self*, *X*)
Apply transforms, and score_samples of the final estimator.

Parameters

X [iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

Returns

y_score [ndarray, shape (n_samples,)]

set_params(*self*, ***kwargs*)
Set the parameters of this estimator.

Valid parameter keys can be listed with `get_params()`.

Returns

self

property transform
Apply transforms, and transform with the final estimator

This also works where final estimator is `None`: all prior transformations are applied.

Parameters

X [iterable] Data to transform. Must fulfill input requirements of first step of the pipeline.

Returns

Xt [array-like of shape (n_samples, n_transformed_features)]

Examples using `sklearn.pipeline.Pipeline`

- *Sample pipeline for text feature extraction and evaluation*

pipeline.make_pipeline(*steps, **kwargs)	Construct a Pipeline from the given estimators.
pipeline.make_union(*transformers, **kwargs)	Construct a FeatureUnion from the given transformers.

7.32.3 sklearn.pipeline.make_pipeline

sklearn.pipeline.**make_pipeline**(**steps*, ***kwargs*)
Construct a Pipeline from the given estimators.

This is a shorthand for the Pipeline constructor; it does not require, and does not permit, naming the estimators. Instead, their names will be set to the lowercase of their types automatically.

Parameters

***steps** [list of estimators.]

memory [None, str or object with the joblib.Memory interface, optional] Used to cache the fitted transformers of the pipeline. By default, no caching is performed. If a string is given, it is the path to the caching directory. Enabling caching triggers a clone of the transformers before fitting. Therefore, the transformer instance given to the pipeline cannot be inspected directly. Use the attribute `named_steps` or `steps` to inspect estimators within the pipeline. Caching the transformers is advantageous when fitting is time consuming.

verbose [boolean, default=False] If True, the time elapsed while fitting each step will be printed as it is completed.

Returns

p [Pipeline]

See also:

sklearn.pipeline.Pipeline Class for creating a pipeline of transforms with a final estimator.

Examples

```
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.preprocessing import StandardScaler
>>> make_pipeline(StandardScaler(), GaussianNB(priors=None))
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('gaussiannb', GaussianNB())])
```

Examples using `sklearn.pipeline.make_pipeline`

- *Approximate nearest neighbors in TSNE*

7.32.4 `sklearn.pipeline`.make_union

`sklearn.pipeline.`**`make_union`**(**transformers*, ***kwargs*)

Construct a FeatureUnion from the given transformers.

This is a shorthand for the FeatureUnion constructor; it does not require, and does not permit, naming the transformers. Instead, they will be given names automatically based on their types. It also does not allow weighting.

Parameters

***transformers** [list of estimators]

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [boolean, optional(default=False)] If True, the time elapsed while fitting each transformer will be printed as it is completed.

Returns

f [FeatureUnion]

See also:

sklearn.pipeline.FeatureUnion Class for concatenating the results of multiple transformer objects.

Examples

```
>>> from sklearn.decomposition import PCA, TruncatedSVD
>>> from sklearn.pipeline import make_union
>>> make_union(PCA(), TruncatedSVD())
 FeatureUnion(transformer_list=[('pca', PCA()),
                                ('truncatedsvd', TruncatedSVD())])
```

Examples using `sklearn.pipeline.make_union`

- *Imputing missing values before building an estimator*

7.33 `sklearn.preprocessing`: Preprocessing and Normalization

The *sklearn.preprocessing* module includes scaling, centering, normalization, binarization methods.

User guide: See the *Preprocessing data* section for further details.

preprocessing.Binarizer([threshold, copy])	Binarize data (set feature values to 0 or 1) according to a threshold
preprocessing.FunctionTransformer([func, ...])	Constructs a transformer from an arbitrary callable.
preprocessing.KBinsDiscretizer([n_bins, ...])	Bin continuous data into intervals.
preprocessing.KernelCenterer()	Center a kernel matrix
preprocessing.LabelBinarizer([neg_label, ...])	Binarize labels in a one-vs-all fashion
preprocessing.LabelEncoder	Encode target labels with value between 0 and n_classes-1.
preprocessing.MultiLabelBinarizer([classes, ...])	Transform between iterable of iterables and a multilabel format
preprocessing.MaxAbsScaler([copy])	Scale each feature by its maximum absolute value.
preprocessing.MinMaxScaler([feature_range, copy])	Transform features by scaling each feature to a given range.
preprocessing.Normalizer([norm, copy])	Normalize samples individually to unit norm.
preprocessing.OneHotEncoder([categories, ...])	Encode categorical features as a one-hot numeric array.
preprocessing.OrdinalEncoder([categories, dtype])	Encode categorical features as an integer array.
preprocessing.PolynomialFeatures([degree, ...])	Generate polynomial and interaction features.
preprocessing.PowerTransformer([method, ...])	Apply a power transform featurewise to make data more Gaussian-like.
preprocessing.QuantileTransformer([...])	Transform features using quantiles information.

Continued on next page

Table 263 – continued from previous page

preprocessing.RobustScaler([with_centering, ...])	Scale features using statistics that are robust to outliers.
preprocessing.StandardScaler([copy, ...])	Standardize features by removing the mean and scaling to unit variance

7.33.1 sklearn.preprocessing.Binarizer

class sklearn.preprocessing.**Binarizer**(*threshold=0.0*, *copy=True*)

Binarize data (set feature values to 0 or 1) according to a threshold

Values greater than the threshold map to 1, while values less than or equal to the threshold map to 0. With the default threshold of 0, only positive values map to 1.

Binarization is a common operation on text count data where the analyst can decide to only consider the presence or absence of a feature rather than a quantified number of occurrences for instance.

It can also be used as a pre-processing step for estimators that consider boolean random variables (e.g. modelled using the Bernoulli distribution in a Bayesian setting).

Read more in the *User Guide*.

Parameters

threshold [float, optional (0.0 by default)] Feature values below or equal to this are replaced by 0, above it by 1. Threshold may not be less than 0 for operations on sparse matrices.

copy [boolean, optional, default True] set to False to perform inplace binarization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR matrix).

See also:

binarize Equivalent function without the estimator API.

Notes

If the input is a sparse matrix, only the non-zero values are subject to update by the Binarizer class.

This estimator is stateless (besides constructor parameters), the fit method does nothing but is useful when used in a pipeline.

Examples

```
>>> from sklearn.preprocessing import Binarizer
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]
>>> transformer = Binarizer().fit(X)  # fit does nothing.
>>> transformer
Binarizer()
>>> transformer.transform(X)
array([[1., 0., 1.],
       [1., 0., 0.],
       [0., 1., 0.]])
```

Methods

fit(self, X[, y])	Do nothing and return the estimator unchanged
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, copy])	Binarize each element of X

__init__(*self*, *threshold=0.0*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Do nothing and return the estimator unchanged

This method is just there to implement the usual API and hence work in pipelines.

Parameters

X [array-like]

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*, *copy=None*)
Binarize each element of X

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data to binarize, element by element. scipy.sparse matrices should be in CSR format to avoid an un-necessary copy.

copy [bool] Copy the input X or not.

7.33.2 sklearn.preprocessing.FunctionTransformer

class sklearn.preprocessing.**FunctionTransformer**(*func=None*, *inverse_func=None*, *validate=False*, *accept_sparse=False*, *check_inverse=True*, *kw_args=None*, *inv_kw_args=None*)

Constructs a transformer from an arbitrary callable.

A FunctionTransformer forwards its X (and optionally y) arguments to a user-defined function or function object and returns the result of this function. This is useful for stateless transformations such as taking the log of frequencies, doing custom scaling, etc.

Note: If a lambda is used as the function, then the resulting transformer will not be pickleable.

New in version 0.17.

Read more in the *User Guide*.

Parameters

func [callable, optional default=None] The callable to use for the transformation. This will be passed the same arguments as transform, with args and kwargs forwarded. If func is None, then func will be the identity function.

inverse_func [callable, optional default=None] The callable to use for the inverse transformation. This will be passed the same arguments as inverse transform, with args and kwargs forwarded. If inverse_func is None, then inverse_func will be the identity function.

validate [bool, optional default=False] Indicate that the input X array should be checked before calling `func`. The possibilities are:

- If False, there is no input validation.
- If True, then X will be converted to a 2-dimensional NumPy array or sparse matrix. If the conversion is not possible an exception is raised.

Changed in version 0.22: The default of `validate` changed from True to False.

accept_sparse [boolean, optional] Indicate that func accepts a sparse matrix as input. If validate is False, this has no effect. Otherwise, if accept_sparse is false, sparse matrix inputs will cause an exception to be raised.

check_inverse [bool, default=True] Whether to check that or `func` followed by `inverse_func` leads to the original inputs. It can be used for a sanity check, raising a warning when the condition is not fulfilled.

New in version 0.20.

kw_args [dict, optional] Dictionary of additional keyword arguments to pass to func.

inv_kw_args [dict, optional] Dictionary of additional keyword arguments to pass to inverse_func.

Examples

```
>>> import numpy as np
>>> from sklearn.preprocessing import FunctionTransformer
>>> transformer = FunctionTransformer(np.log1p)
>>> X = np.array([[0, 1], [2, 3]])
>>> transformer.transform(X)
array([[0.       , 0.6931...],
       [1.0986..., 1.3862...]])
```

Methods

fit(self, X[, y])	Fit transformer by checking X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Transform X using the inverse function.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform X using the forward function.

__init__(*self*, *func=None*, *inverse_func=None*, *validate=False*, *accept_sparse=False*, *check_inverse=True*, *kw_args=None*, *inv_kw_args=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit transformer by checking X.

If `validate` is `True`, `X` will be checked.

Parameters

X [array-like, shape (n_samples, n_features)] Input array.

Returns

self

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*)
Transform X using the inverse function.

Parameters

X [array-like, shape (n_samples, n_features)] Input array.

Returns

X_out [array-like, shape (n_samples, n_features)] Transformed input.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Transform X using the forward function.

Parameters

X [array-like, shape (n_samples, n_features)] Input array.

Returns

X_out [array-like, shape (n_samples, n_features)] Transformed input.

Examples using `sklearn.preprocessing.FunctionTransformer`

- *Using FunctionTransformer to select columns*

7.33.3 `sklearn.preprocessing`.KBinsDiscretizer

class sklearn.preprocessing.**KBinsDiscretizer**(*n_bins=5*, *encode='onehot'*, *strategy='quantile'*)

Bin continuous data into intervals.

Read more in the *User Guide*.

Parameters

n_bins [int or array-like, shape (n_features,) (default=5)] The number of bins to produce. Raises ValueError if `n_bins < 2`.

encode [{'onehot', 'onehot-dense', 'ordinal'}, (default='onehot')] Method used to encode the transformed result.

onehot Encode the transformed result with one-hot encoding and return a sparse matrix. Ignored features are always stacked to the right.

onehot-dense Encode the transformed result with one-hot encoding and return a dense array. Ignored features are always stacked to the right.

ordinal Return the bin identifier encoded as an integer value.

strategy [{'uniform', 'quantile', 'kmeans'}, (default='quantile')] Strategy used to define the widths of the bins.

uniform All bins in each feature have identical widths.

quantile All bins in each feature have the same number of points.

kmeans Values in each bin have the same nearest center of a 1D k-means cluster.

Attributes

n_bins_ [int array, shape (n_features,)] Number of bins per feature. Bins whose width are too small (i.e., <= 1e-8) are removed with a warning.

bin_edges_ [array of arrays, shape (n_features,)] The edges of each bin. Contain arrays of varying shapes `(n_bins_, )` Ignored features will have empty arrays.

See also:

sklearn.preprocessing.Binarizer Class used to bin values as `0` or `1` based on a parameter `threshold`.

Notes

In bin edges for feature `i`, the first and last values are used only for `inverse_transform`. During transform, bin edges are extended to:

```
np.concatenate([-np.inf, bin_edges_[i][1:-1], np.inf])
```

You can combine `KBinsDiscretizer` with *sklearn.compose.ColumnTransformer* if you only want to preprocess part of the features.

`KBinsDiscretizer` might produce constant features (e.g., when `encode = 'onehot'` and certain bins do not contain any data). These features can be removed with feature selection algorithms (e.g., *sklearn.feature_selection.VarianceThreshold*).

Examples

```
>>> X = [[-2, 1, -4,   -1],
...      [-1, 2, -3, -0.5],
...      [ 0, 3, -2,  0.5],
...      [ 1, 4, -1,    2]]
>>> est = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='uniform')
>>> est.fit(X)
KBinsDiscretizer(...)
>>> Xt = est.transform(X)
>>> Xt  # doctest: +SKIP
array([[ 0., 0., 0., 0.],
       [ 1., 1., 1., 0.],
       [ 2., 2., 2., 1.],
       [ 2., 2., 2., 2.]])
```

Sometimes it may be useful to convert the data back into the original feature space. The `inverse_transform` function converts the binned data into the original feature space. Each value will be equal to the mean of the two bin edges.

```
>>> est.bin_edges_[0]
array([-2., -1.,  0.,  1.])
>>> est.inverse_transform(Xt)
array([[-1.5,  1.5, -3.5, -0.5],
       [-0.5,  2.5, -2.5, -0.5],
       [ 0.5,  3.5, -1.5,  0.5],
       [ 0.5,  3.5, -1.5,  1.5]])
```

Methods

fit(self, X[, y])	Fit the estimator.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, Xt)	Transform discretized data back to original feature space.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Discretize the data.

__init__ (*self*, *n_bins=5*, *encode='onehot'*, *strategy='quantile'*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*)
Fit the estimator.

Parameters

X [numeric array-like, shape (n_samples, n_features)] Data to be discretized.

y [None] Ignored. This parameter exists only for compatibility with *sklearn.pipeline.Pipeline*.

Returns

self

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *Xt*)
Transform discretized data back to original feature space.

Note that this function does not regenerate the original data due to discretization rounding.

Parameters

Xt [numeric array-like, shape (n_sample, n_features)] Transformed data in the binned space.

Returns

Xinv [numeric array-like] Data in the original feature space.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Discretize the data.

Parameters

X [numeric array-like, shape (n_samples, n_features)] Data to be discretized.

Returns

Xt [numeric array-like or sparse matrix] Data in the binned space.

Examples using `sklearn.preprocessing.KBinsDiscretizer`

- *Using KBinsDiscretizer to discretize continuous features*
- *Demonstrating the different strategies of KBinsDiscretizer*
- *Feature discretization*

7.33.4 `sklearn.preprocessing`.KernelCenterer

class sklearn.preprocessing.**KernelCenterer**
Center a kernel matrix

Let K(x, z) be a kernel defined by phi(x)^T phi(z), where phi is a function mapping x to a Hilbert space. KernelCenterer centers (i.e., normalize to have zero mean) the data without explicitly computing phi(x). It is equivalent to centering phi(x) with sklearn.preprocessing.StandardScaler(with_std=False).

Read more in the *User Guide*.

Attributes

K_fit_rows_ [array, shape (n_samples,)] Average of each column of kernel matrix

K_fit_all_ [float] Average of kernel matrix

Examples

```
>>> from sklearn.preprocessing import KernelCenterer
>>> from sklearn.metrics.pairwise import pairwise_kernels
>>> X = [[ 1., -2.,  2.],
...      [ -2.,  1.,  3.],
...      [ 4.,  1., -2.]]
>>> K = pairwise_kernels(X, metric='linear')
>>> K
array([[  9.,   2.,  -2.],
       [  2.,  14., -13.],
       [ -2., -13.,  21.]])
>>> transformer = KernelCenterer().fit(K)
>>> transformer
KernelCenterer()
>>> transformer.transform(K)
array([[  5.,   0.,  -5.],
       [  0.,  14., -14.],
       [ -5., -14.,  19.]])
```

Methods

fit(self, K[, y])	Fit KernelCenterer
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, K[, copy])	Center kernel matrix.

__init__(*self*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *K*, *y=None*)
Fit KernelCenterer

Parameters

K [numpy array of shape [n_samples, n_samples]] Kernel matrix.

Returns

self [returns an instance of self.]

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *K*, *copy=True*)
Center kernel matrix.

Parameters

K [numpy array of shape [n_samples1, n_samples2]] Kernel matrix.

copy [boolean, optional, default True] Set to False to perform inplace computation.

Returns

K_new [numpy array of shape [n_samples1, n_samples2]]

7.33.5 sklearn.preprocessing.LabelBinarizer

class sklearn.preprocessing.**LabelBinarizer**(*neg_label=0*, *pos_label=1*, *sparse_output=False*)
Binarize labels in a one-vs-all fashion

Several regression and binary classification algorithms are available in scikit-learn. A simple way to extend these algorithms to the multi-class classification case is to use the so-called one-vs-all scheme.

At learning time, this simply consists in learning one regressor or binary classifier per class. In doing so, one needs to convert multi-class labels to binary labels (belong or does not belong to the class). LabelBinarizer makes this process easy with the transform method.

At prediction time, one assigns the class for which the corresponding model gave the greatest confidence. LabelBinarizer makes this easy with the inverse_transform method.

Read more in the *User Guide*.

Parameters

neg_label [int (default: 0)] Value with which negative labels must be encoded.

pos_label [int (default: 1)] Value with which positive labels must be encoded.

sparse_output [boolean (default: False)] True if the returned array from transform is desired to be in sparse CSR format.

Attributes

classes_ [array of shape [n_class]] Holds the label for each class.

y_type_ [str,] Represents the type of the target data as evaluated by utils.multiclass.type_of_target. Possible type are 'continuous', 'continuous-multioutput', 'binary', 'multiclass', 'multiclass-multioutput', 'multilabel-indicator', and 'unknown'.

sparse_input_ [boolean,] True if the input data to transform is given as a sparse matrix, False otherwise.

See also:

label_binarize function to perform the transform operation of LabelBinarizer with fixed classes.

sklearn.preprocessing.OneHotEncoder encode categorical features using a one-hot aka one-of-K scheme.

Examples

```
>>> from sklearn import preprocessing
>>> lb = preprocessing.LabelBinarizer()
>>> lb.fit([1, 2, 6, 4, 2])
LabelBinarizer()
>>> lb.classes_
array([1, 2, 4, 6])
>>> lb.transform([1, 6])
array([[1, 0, 0, 0],
       [0, 0, 0, 1]])
```

Binary targets transform to a column vector

```
>>> lb = preprocessing.LabelBinarizer()
>>> lb.fit_transform(['yes', 'no', 'no', 'yes'])
array([[1],
       [0],
       [0],
       [1]])
```

Passing a 2D matrix for multilabel classification

```
>>> import numpy as np
>>> lb.fit(np.array([[0, 1, 1], [1, 0, 0]]))
LabelBinarizer()
>>> lb.classes_
array([0, 1, 2])
>>> lb.transform([0, 1, 2, 1])
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1],
       [0, 1, 0]])
```

Methods

fit(self, y)	Fit label binarizer
fit_transform(self, y)	Fit label binarizer and transform multi-class labels to binary labels.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, Y[, threshold])	Transform binary labels back to multi-class labels
set_params(self, **params)	Set the parameters of this estimator.
transform(self, y)	Transform multi-class labels to binary labels

__init__ (*self*, *neg_label=0*, *pos_label=1*, *sparse_output=False*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *y*)
Fit label binarizer

Parameters

y [array of shape [n_samples,] or [n_samples, n_classes]] Target values. The 2-d matrix should only contain 0 and 1, represents multilabel classification.

Returns

self [returns an instance of self.]

fit_transform (*self*, *y*)
Fit label binarizer and transform multi-class labels to binary labels.

The output of transform is sometimes referred to as the 1-of-K coding scheme.

Parameters

y [array or sparse matrix of shape [n_samples,] or [n_samples, n_classes]] Target values. The 2-d matrix should only contain 0 and 1, represents multilabel classification. Sparse matrix can be CSR, CSC, COO, DOK, or LIL.

Returns

Y [array or CSR matrix of shape [n_samples, n_classes]] Shape will be [n_samples, 1] for binary problems.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform (*self*, *Y*, *threshold=None*)
Transform binary labels back to multi-class labels

Parameters

Y [numpy array or sparse matrix with shape [n_samples, n_classes]] Target values. All sparse matrices are converted to CSR before inverse transformation.

threshold [float or None] Threshold used in the binary and multi-label cases.

Use 0 when `Y` contains the output of decision_function (classifier). Use 0.5 when `Y` contains the output of predict_proba.

If None, the threshold is assumed to be half way between neg_label and pos_label.

Returns

y [numpy array or CSR matrix of shape [n_samples] Target values.]

Notes

In the case when the binary labels are fractional (probabilistic), inverse_transform chooses the class with the greatest value. Typically, this allows to use the output of a linear model's decision_function method directly as the input of inverse_transform.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *y*)
Transform multi-class labels to binary labels

The output of transform is sometimes referred to by some authors as the 1-of-K coding scheme.

Parameters

y [array or sparse matrix of shape [n_samples,] or [n_samples, n_classes]] Target values. The 2-d matrix should only contain 0 and 1, represents multilabel classification. Sparse matrix can be CSR, CSC, COO, DOK, or LIL.

Returns

Y [numpy array or CSR matrix of shape [n_samples, n_classes]] Shape will be [n_samples, 1] for binary problems.

7.33.6 sklearn.preprocessing.LabelEncoder

class sklearn.preprocessing.**LabelEncoder**
Encode target labels with value between 0 and n_classes-1.

This transformer should be used to encode target values, *i.e.* `y`, and not the input `X`.

Read more in the *User Guide*.

New in version 0.12.

Attributes

classes_ [array of shape (n_class,)] Holds the label for each class.

See also:

sklearn.preprocessing.OrdinalEncoder Encode categorical features using an ordinal encoding scheme.

sklearn.preprocessing.OneHotEncoder Encode categorical features as a one-hot numeric array.

Examples

`LabelEncoder` can be used to normalize labels.

```
>>> from sklearn import preprocessing
>>> le = preprocessing.LabelEncoder()
>>> le.fit([1, 2, 2, 6])
LabelEncoder()
>>> le.classes_
array([1, 2, 6])
>>> le.transform([1, 1, 2, 6])
array([0, 0, 1, 2]...)
>>> le.inverse_transform([0, 0, 1, 2])
array([1, 1, 2, 6])
```

It can also be used to transform non-numerical labels (as long as they are hashable and comparable) to numerical labels.

```
>>> le = preprocessing.LabelEncoder()
>>> le.fit(["paris", "paris", "tokyo", "amsterdam"])
LabelEncoder()
>>> list(le.classes_)
['amsterdam', 'paris', 'tokyo']
>>> le.transform(["tokyo", "tokyo", "paris"])
array([2, 2, 1]...)
>>> list(le.inverse_transform([2, 2, 1]))
['tokyo', 'tokyo', 'paris']
```

Methods

fit(self, y)	Fit label encoder
fit_transform(self, y)	Fit label encoder and return encoded labels
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, y)	Transform labels back to original encoding.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, y)	Transform labels to normalized encoding.

__init__ (*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *y*)
Fit label encoder

Parameters

y [array-like of shape (n_samples,)] Target values.

Returns

self [returns an instance of self.]

fit_transform(*self*, *y*)
Fit label encoder and return encoded labels

Parameters

y [array-like of shape [n_samples]] Target values.

Returns

y [array-like of shape [n_samples]]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *y*)
Transform labels back to original encoding.

Parameters

y [numpy array of shape [n_samples]] Target values.

Returns

y [numpy array of shape [n_samples]]

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *y*)
Transform labels to normalized encoding.

Parameters

y [array-like of shape [n_samples]] Target values.

Returns

y [array-like of shape [n_samples]]

7.33.7 sklearn.preprocessing.MultiLabelBinarizer

class sklearn.preprocessing.**MultiLabelBinarizer**(*classes=None*, *sparse_output=False*)

Transform between iterable of iterables and a multilabel format

Although a list of sets or tuples is a very intuitive format for multilabel data, it is unwieldy to process. This transformer converts between this intuitive format and the supported multilabel format: a (samples x classes) binary matrix indicating the presence of a class label.

Parameters

classes [array-like of shape [n_classes] (optional)] Indicates an ordering for the class labels. All entries should be unique (cannot contain duplicate classes).

sparse_output [boolean (default: False),] Set to true if output binary array is desired in CSR sparse format

Attributes

classes_ [array of labels] A copy of the `classes` parameter where provided, or otherwise, the sorted set of classes found when fitting.

See also:

sklearn.preprocessing.OneHotEncoder encode categorical features using a one-hot aka one-of-K scheme.

Examples

```
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> mlb = MultiLabelBinarizer()
>>> mlb.fit_transform([(1, 2), (3,)])
array([[1, 1, 0],
       [0, 0, 1]])
>>> mlb.classes_
array([1, 2, 3])
```

```
>>> mlb.fit_transform([{'sci-fi', 'thriller'}, {'comedy'}])
array([[0, 1, 1],
       [1, 0, 0]])
>>> list(mlb.classes_)
['comedy', 'sci-fi', 'thriller']
```

A common mistake is to pass in a list, which leads to the following issue:

```
>>> mlb = MultiLabelBinarizer()
>>> mlb.fit(['sci-fi', 'thriller', 'comedy'])
MultiLabelBinarizer()
>>> mlb.classes_
array(['-', 'c', 'd', 'e', 'f', 'h', 'i', 'l', 'm', 'o', 'r', 's', 't',
    'y'], dtype=object)
```

To correct this, the list of labels should be passed in as:

```
>>> mlb = MultiLabelBinarizer()
>>> mlb.fit([['sci-fi', 'thriller', 'comedy']])
MultiLabelBinarizer()
```

(continues on next page)

(continued from previous page)

```
>>> mlb.classes_
array(['comedy', 'sci-fi', 'thriller'], dtype=object)
```

Methods

fit(self, y)	Fit the label sets binarizer, storing *classes_*
fit_transform(self, y)	Fit the label sets binarizer and transform the given label sets
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, yt)	Transform the given indicator matrix into label sets
set_params(self, **params)	Set the parameters of this estimator.
transform(self, y)	Transform the given label sets

__init__ (*self*, *classes=None*, *sparse_output=False*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *y*)
Fit the label sets binarizer, storing *classes_*

Parameters

y [iterable of iterables] A set of labels (any orderable and hashable object) for each sample. If the `classes` parameter is set, `y` will not be iterated.

Returns

self [returns this MultiLabelBinarizer instance]

fit_transform (*self*, *y*)
Fit the label sets binarizer and transform the given label sets

Parameters

y [iterable of iterables] A set of labels (any orderable and hashable object) for each sample. If the `classes` parameter is set, `y` will not be iterated.

Returns

y_indicator [array or CSR matrix, shape (n_samples, n_classes)] A matrix such that `y_indicator[i, j] = 1` iff `classes_[j]` is in `y[i]`, and 0 otherwise.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform (*self*, *yt*)
Transform the given indicator matrix into label sets

Parameters

yt [array or sparse matrix of shape (n_samples, n_classes)] A matrix containing only 1s ands 0s.

Returns

y [list of tuples] The set of labels for each sample such that `y[i]` consists of `classes_[j]` for each `yt[i, j] == 1`.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *y*)
Transform the given label sets

Parameters

y [iterable of iterables] A set of labels (any orderable and hashable object) for each sample. If the `classes` parameter is set, `y` will not be iterated.

Returns

y_indicator [array or CSR matrix, shape (n_samples, n_classes)] A matrix such that `y_indicator[i, j] = 1` iff `classes_[j]` is in `y[i]`, and 0 otherwise.

7.33.8 sklearn.preprocessing.MaxAbsScaler

class sklearn.preprocessing.**MaxAbsScaler**(*copy=True*)
Scale each feature by its maximum absolute value.

This estimator scales and translates each feature individually such that the maximal absolute value of each feature in the training set will be 1.0. It does not shift/center the data, and thus does not destroy any sparsity.

This scaler can also be applied to sparse CSR or CSC matrices.

New in version 0.17.

Parameters

copy [boolean, optional, default is True] Set to False to perform inplace scaling and avoid a copy (if the input is already a numpy array).

Attributes

scale_ [ndarray, shape (n_features,)] Per feature relative scaling of the data.

New in version 0.17: *scale_* attribute.

max_abs_ [ndarray, shape (n_features,)] Per feature maximum absolute value.

n_samples_seen_ [int] The number of samples processed by the estimator. Will be reset on new calls to fit, but increments across `partial_fit` calls.

See also:

maxabs_scale Equivalent function without the estimator API.

Notes

NaNs are treated as missing values: disregarded in fit, and maintained in transform.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples

```
>>> from sklearn.preprocessing import MaxAbsScaler
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]
>>> transformer = MaxAbsScaler().fit(X)
>>> transformer
MaxAbsScaler()
>>> transformer.transform(X)
array([[ 0.5, -1. ,  1. ],
       [ 1. ,  0. ,  0. ],
       [ 0. ,  1. , -0.5]])
```

Methods

fit(self, X[, y])	Compute the maximum absolute value to be used for later scaling.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Scale back the data to the original representation
partial_fit(self, X[, y])	Online computation of max absolute value of X for later scaling.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Scale the data

__init__ (*self*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*)
Compute the maximum absolute value to be used for later scaling.

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data used to compute the per-feature minimum and maximum used for later scaling along the features axis.

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*)
Scale back the data to the original representation

Parameters

X [{array-like, sparse matrix}] The data that should be transformed back.

partial_fit(*self*, *X*, *y=None*)
Online computation of max absolute value of X for later scaling.

All of X is processed as a single batch. This is intended for cases when *fit* is not feasible due to very large number of `n_samples` or because X is read from a continuous stream.

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data used to compute the mean and standard deviation used for later scaling along the features axis.

y [None] Ignored.

Returns

self [object] Transformer instance.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Scale the data

Parameters

X [{array-like, sparse matrix}] The data that should be scaled.

Examples using `sklearn.preprocessing.MaxAbsScaler`

- *Compare the effect of different scalers on data with outliers*

7.33.9 `sklearn.preprocessing`.MinMaxScaler

class sklearn.preprocessing.**MinMaxScaler**(*feature_range=(0, 1)*, *copy=True*)

Transform features by scaling each feature to a given range.

This estimator scales and translates each feature individually such that it is in the given range on the training set, e.g. between zero and one.

The transformation is given by:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

where min, max = feature_range.

The transformation is calculated as:

```
X_scaled = scale * X + min - X.min(axis=0) * scale
where scale = (max - min) / (X.max(axis=0) - X.min(axis=0))
```

This transformation is often used as an alternative to zero mean, unit variance scaling.

Read more in the *User Guide*.

Parameters

feature_range [tuple (min, max), default=(0, 1)] Desired range of transformed data.

copy [bool, default=True] Set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array).

Attributes

min_ [ndarray of shape (n_features,)] Per feature adjustment for minimum. Equivalent to `min - X.min(axis=0) * self.scale_`

scale_ [ndarray of shape (n_features,)] Per feature relative scaling of the data. Equivalent to `(max - min) / (X.max(axis=0) - X.min(axis=0))`

New in version 0.17: *scale_* attribute.

data_min_ [ndarray of shape (n_features,)] Per feature minimum seen in the data

New in version 0.17: *data_min_*

data_max_ [ndarray of shape (n_features,)] Per feature maximum seen in the data

New in version 0.17: *data_max_*

data_range_ [ndarray of shape (n_features,)] Per feature range `(data_max_ - data_min_)` seen in the data

New in version 0.17: *data_range_*

n_samples_seen_ [int] The number of samples processed by the estimator. It will be reset on new calls to fit, but increments across `partial_fit` calls.

See also:

minmax_scale Equivalent function without the estimator API.

Notes

NaNs are treated as missing values: disregarded in fit, and maintained in transform.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]
>>> scaler = MinMaxScaler()
>>> print(scaler.fit(data))
MinMaxScaler()
>>> print(scaler.data_max_)
[ 1. 18.]
>>> print(scaler.transform(data))
[[0.   0.  ]
 [0.25 0.25]
 [0.5  0.5 ]
 [1.   1.  ]]
>>> print(scaler.transform([[2, 2]]))
[[1.5 0. ]]
```

Methods

fit(self, X[, y])	Compute the minimum and maximum to be used for later scaling.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Undo the scaling of X according to feature_range.
partial_fit(self, X[, y])	Online computation of min and max on X for later scaling.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Scale features of X according to feature_range.

__init__ (*self*, *feature_range=(0, 1)*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*)
Compute the minimum and maximum to be used for later scaling.

Parameters

X [array-like of shape (n_samples, n_features)] The data used to compute the per-feature minimum and maximum used for later scaling along the features axis.

y [None] Ignored.

Returns

self [object] Fitted scaler.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*)
Undo the scaling of X according to feature_range.

Parameters

X [array-like of shape (n_samples, n_features)] Input data that will be transformed. It cannot be sparse.

Returns

Xt [array-like of shape (n_samples, n_features)] Transformed data.

partial_fit(*self*, *X*, *y=None*)
Online computation of min and max on X for later scaling.

All of X is processed as a single batch. This is intended for cases when `fit` is not feasible due to very large number of `n_samples` or because X is read from a continuous stream.

Parameters

X [array-like of shape (n_samples, n_features)] The data used to compute the mean and standard deviation used for later scaling along the features axis.

y [None] Ignored.

Returns

self [object] Transformer instance.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Scale features of X according to feature_range.

Parameters

X [array-like of shape (n_samples, n_features)] Input data that will be transformed.

Returns

Xt [array-like of shape (n_samples, n_features)] Transformed data.

Examples using `sklearn.preprocessing.MinMaxScaler`

- *Univariate Feature Selection*
- *Compare Stochastic learning strategies for MLPClassifier*
- *Compare the effect of different scalers on data with outliers*

7.33.10 `sklearn.preprocessing`.Normalizer

class sklearn.preprocessing.**Normalizer**(*norm='l2'*, *copy=True*)
Normalize samples individually to unit norm.

Each sample (i.e. each row of the data matrix) with at least one non zero component is rescaled independently of other samples so that its norm (l1 or l2) equals one.

This transformer is able to work both with dense numpy arrays and scipy.sparse matrix (use CSR format if you want to avoid the burden of a copy / conversion).

Scaling inputs to unit norms is a common operation for text classification or clustering for instance. For instance the dot product of two l2-normalized TF-IDF vectors is the cosine similarity of the vectors and is the base similarity metric for the Vector Space Model commonly used by the Information Retrieval community.

Read more in the *User Guide*.

Parameters

norm ['l1', 'l2', or 'max', optional ('l2' by default)] The norm to use to normalize each non zero sample.

copy [boolean, optional, default True] set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR matrix).

See also:

normalize Equivalent function without the estimator API.

Notes

This estimator is stateless (besides constructor parameters), the fit method does nothing but is useful when used in a pipeline.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples

```
>>> from sklearn.preprocessing import Normalizer
>>> X = [[4, 1, 2, 2],
...      [1, 3, 9, 3],
...      [5, 7, 5, 1]]
>>> transformer = Normalizer().fit(X)  # fit does nothing.
>>> transformer
Normalizer()
>>> transformer.transform(X)
array([[0.8, 0.2, 0.4, 0.4],
       [0.1, 0.3, 0.9, 0.3],
       [0.5, 0.7, 0.5, 0.1]])
```

Methods

fit(self, X[, y])	Do nothing and return the estimator unchanged
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, copy])	Scale each non zero row of X to unit norm

__init__ (*self*, *norm='l2'*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*)
Do nothing and return the estimator unchanged

This method is just there to implement the usual API and hence work in pipelines.

Parameters

X [array-like]

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*, *copy=None*)
Scale each non zero row of X to unit norm

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data to normalize, row by row. scipy.sparse matrices should be in CSR format to avoid an un-necessary copy.

copy [bool, optional (default: None)] Copy the input X or not.

Examples using `sklearn.preprocessing.Normalizer`

- *Compare the effect of different scalers on data with outliers*
- *Clustering text documents using k-means*

7.33.11 `sklearn.preprocessing`.OneHotEncoder

class sklearn.preprocessing.**OneHotEncoder**(*categories='auto'*, *drop=None*, *sparse=True*, *dtype=<class 'numpy.float64'>*, *handle_unknown='error'*)
Encode categorical features as a one-hot numeric array.

The input to this transformer should be an array-like of integers or strings, denoting the values taken on by categorical (discrete) features. The features are encoded using a one-hot (aka 'one-of-K' or 'dummy') encoding scheme. This creates a binary column for each category and returns a sparse matrix or dense array (depending on the `sparse` parameter)

By default, the encoder derives the categories based on the unique values in each feature. Alternatively, you can also specify the `categories` manually.

This encoding is needed for feeding categorical data to many scikit-learn estimators, notably linear models and SVMs with the standard kernels.

Note: a one-hot encoding of y labels should use a LabelBinarizer instead.

Read more in the *User Guide*.

Changed in version 0.20.

Parameters

categories ['auto' or a list of array-like, default='auto'] Categories (unique values) per feature:

- 'auto' : Determine categories automatically from the training data.

- list : `categories[i]` holds the categories expected in the ith column. The passed categories should not mix strings and numeric values within a single feature, and should be sorted in case of numeric values.

The used categories can be found in the `categories_` attribute.

drop ['first' or a array-like of shape (n_features,), default=None] Specifies a methodology to use to drop one of the categories per feature. This is useful in situations where perfectly collinear features cause problems, such as when feeding the resulting data into a neural network or an unregularized regression.

- None : retain all features (the default).
- 'first' : drop the first category in each feature. If only one category is present, the feature will be dropped entirely.
- array : `drop[i]` is the category in feature `X[:, i]` that should be dropped.

sparse [bool, default=True] Will return sparse matrix if set True else will return an array.

dtype [number type, default=np.float] Desired dtype of output.

handle_unknown [{'error', 'ignore'}, default='error'] Whether to raise an error or ignore if an unknown categorical feature is present during transform (default is to raise). When this parameter is set to 'ignore' and an unknown category is encountered during transform, the resulting one-hot encoded columns for this feature will be all zeros. In the inverse transform, an unknown category will be denoted as None.

Attributes

categories_ [list of arrays] The categories of each feature determined during fitting (in order of the features in X and corresponding with the output of `transform`). This includes the category specified in `drop` (if any).

drop_idx_ [array of shape (n_features,)] `drop_idx_[i]` is the index in `categories_[i]` of the category to be dropped for each feature. None if all the transformed features will be retained.

See also:

sklearn.preprocessing.OrdinalEncoder Performs an ordinal (integer) encoding of the categorical features.

sklearn.feature_extraction.DictVectorizer Performs a one-hot encoding of dictionary items (also handles string-valued features).

sklearn.feature_extraction.FeatureHasher Performs an approximate one-hot encoding of dictionary items or strings.

sklearn.preprocessing.LabelBinarizer Binarizes labels in a one-vs-all fashion.

sklearn.preprocessing.MultiLabelBinarizer Transforms between iterable of iterables and a multilabel format, e.g. a (samples x classes) binary matrix indicating the presence of a class label.

Examples

Given a dataset with two features, we let the encoder find the unique values per feature and transform the data to a binary one-hot encoding.

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> enc = OneHotEncoder(handle_unknown='ignore')
>>> X = [['Male', 1], ['Female', 3], ['Female', 2]]
>>> enc.fit(X)
OneHotEncoder(handle_unknown='ignore')
>>> enc.categories_
[array(['Female', 'Male'], dtype=object), array([1, 2, 3], dtype=object)]
>>> enc.transform([['Female', 1], ['Male', 4]]).toarray()
array([[1., 0., 1., 0., 0.],
       [0., 1., 0., 0., 0.]])
>>> enc.inverse_transform([[0, 1, 1, 0, 0], [0, 0, 0, 1, 0]])
array([['Male', 1],
       [None, 2]], dtype=object)
>>> enc.get_feature_names(['gender', 'group'])
array(['gender_Female', 'gender_Male', 'group_1', 'group_2', 'group_3'],
  dtype=object)
>>> drop_enc = OneHotEncoder(drop='first').fit(X)
>>> drop_enc.categories_
[array(['Female', 'Male'], dtype=object), array([1, 2, 3], dtype=object)]
>>> drop_enc.transform([['Female', 1], ['Male', 2]]).toarray()
array([[0., 0., 0.],
       [1., 1., 0.]])
```

Methods

fit(self, X[, y])	Fit OneHotEncoder to X.
fit_transform(self, X[, y])	Fit OneHotEncoder to X, then transform X.
get_feature_names(self[, input_features])	Return feature names for output features.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Convert the data back to the original representation.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform X using one-hot encoding.

__init__(*self*, *categories='auto'*, *drop=None*, *sparse=True*, *dtype=<class 'numpy.float64'>*, *handle_unknown='error'*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit OneHotEncoder to X.

Parameters

X [array-like, shape [n_samples, n_features]] The data to determine the categories of each feature.

y [None] Ignored. This parameter exists only for compatibility with *sklearn.pipeline.Pipeline*.

Returns

self

fit_transform(*self*, *X*, *y=None*)
Fit OneHotEncoder to X, then transform X.

Equivalent to fit(X).transform(X) but more convenient.

Parameters

X [array-like, shape [n_samples, n_features]] The data to encode.

y [None] Ignored. This parameter exists only for compatibility with *sklearn.pipeline.Pipeline*.

Returns

X_out [sparse matrix if sparse=True else a 2-d array] Transformed input.

get_feature_names(*self*, *input_features=None*)
Return feature names for output features.

Parameters

input_features [list of str of shape (n_features,)] String names for input features if available. By default, "x0", "x1", ... "xn_features" is used.

Returns

output_feature_names [ndarray of shape (n_output_features,)] Array of feature names.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*)
Convert the data back to the original representation.

In case unknown categories are encountered (all zeros in the one-hot encoding), `None` is used to represent this category.

Parameters

X [array-like or sparse matrix, shape [n_samples, n_encoded_features]] The transformed data.

Returns

X_tr [array-like, shape [n_samples, n_features]] Inverse transformed array.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Transform X using one-hot encoding.

Parameters

X [array-like, shape [n_samples, n_features]] The data to encode.

Returns

X_out [sparse matrix if sparse=True else a 2-d array] Transformed input.

Examples using `sklearn.preprocessing.OneHotEncoder`

- *Feature transformations with ensembles of trees*
- *Permutation Importance vs Random Forest Feature Importance (MDI)*
- *Column Transformer with Mixed Types*

7.33.12 `sklearn.preprocessing`.OrdinalEncoder

class sklearn.preprocessing.**OrdinalEncoder**(*categories='auto'*, *dtype=<class 'numpy.float64'>*)

Encode categorical features as an integer array.

The input to this transformer should be an array-like of integers or strings, denoting the values taken on by categorical (discrete) features. The features are converted to ordinal integers. This results in a single column of integers (0 to n_categories - 1) per feature.

Read more in the *User Guide*.

Changed in version 0.20.1.

Parameters

categories ['auto' or a list of array-like, default='auto'] Categories (unique values) per feature:

- 'auto' : Determine categories automatically from the training data.
- list : `categories[i]` holds the categories expected in the ith column. The passed categories should not mix strings and numeric values, and should be sorted in case of numeric values.

The used categories can be found in the `categories_` attribute.

dtype [number type, default np.float64] Desired dtype of output.

Attributes

categories_ [list of arrays] The categories of each feature determined during fitting (in order of the features in X and corresponding with the output of `transform`).

See also:

sklearn.preprocessing.OneHotEncoder Performs a one-hot encoding of categorical features.

sklearn.preprocessing.LabelEncoder Encodes target labels with values between 0 and n_classes-1.

Examples

Given a dataset with two features, we let the encoder find the unique values per feature and transform the data to an ordinal encoding.

```
>>> from sklearn.preprocessing import OrdinalEncoder
>>> enc = OrdinalEncoder()
>>> X = [['Male', 1], ['Female', 3], ['Female', 2]]
>>> enc.fit(X)
OrdinalEncoder()
>>> enc.categories_
[array(['Female', 'Male'], dtype=object), array([1, 2, 3], dtype=object)]
>>> enc.transform([['Female', 3], ['Male', 1]])
array([[0., 2.],
       [1., 0.]])
```

```
>>> enc.inverse_transform([[1, 0], [0, 1]])
array([['Male', 1],
       ['Female', 2]], dtype=object)
```

Methods

fit(self, X[, y])	Fit the OrdinalEncoder to X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Convert the data back to the original representation.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform X to ordinal codes.

__init__(*self*, *categories='auto'*, *dtype=<class 'numpy.float64'>*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Fit the OrdinalEncoder to X.

Parameters

X [array-like, shape [n_samples, n_features]] The data to determine the categories of each feature.

y [None] Ignored. This parameter exists only for compatibility with *sklearn.pipeline.Pipeline*.

Returns

self

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*)
Convert the data back to the original representation.

Parameters

X [array-like or sparse matrix, shape [n_samples, n_encoded_features]] The transformed data.

Returns

X_tr [array-like, shape [n_samples, n_features]] Inverse transformed array.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Transform X to ordinal codes.

Parameters

X [array-like, shape [n_samples, n_features]] The data to encode.

Returns

X_out [sparse matrix or a 2-d array] Transformed input.

7.33.13 sklearn.preprocessing.PolynomialFeatures

class sklearn.preprocessing.**PolynomialFeatures**(*degree=2*, *interaction_only=False*, *include_bias=True*, *order='C'*)
Generate polynomial and interaction features.

Generate a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to the specified degree. For example, if an input sample is two dimensional and of the form [a, b], the degree-2 polynomial features are [1, a, b, a^2, ab, b^2].

Parameters

degree [integer] The degree of the polynomial features. Default = 2.

interaction_only [boolean, default = False] If true, only interaction features are produced: features that are products of at most `degree` *distinct* input features (so not `x[1] ** 2`, `x[0] * x[2] ** 3`, etc.).

include_bias [boolean] If True (default), then include a bias column, the feature in which all polynomial powers are zero (i.e. a column of ones - acts as an intercept term in a linear model).

order [str in {'C', 'F'}, default 'C'] Order of output array in the dense case. 'F' order is faster to compute, but may slow down subsequent estimators.

New in version 0.21.

Attributes

powers_ [array, shape (n_output_features, n_input_features)] powers_[i, j] is the exponent of the jth input in the ith output.

n_input_features_ [int] The total number of input features.

n_output_features_ [int] The total number of polynomial output features. The number of output features is computed by iterating over all suitably sized combinations of input features.

Notes

Be aware that the number of features in the output array scales polynomially in the number of features of the input array, and exponentially in the degree. High degrees can cause overfitting.

See *examples/linear_model/plot_polynomial_interpolation.py*

Examples

```
>>> import numpy as np
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> poly = PolynomialFeatures(2)
>>> poly.fit_transform(X)
array([[ 1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  2.,  3.,  4.,  6.,  9.],
       [ 1.,  4.,  5., 16., 20., 25.]])
>>> poly = PolynomialFeatures(interaction_only=True)
>>> poly.fit_transform(X)
array([[ 1.,  0.,  1.,  0.],
       [ 1.,  2.,  3.,  6.],
       [ 1.,  4.,  5., 20.]])
```

Methods

fit(self, X[, y])	Compute number of output features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_feature_names(self[, input_features])	Return feature names for output features

Continued on next page

Table 276 – continued from previous page

get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform data to polynomial features

__init__ (*self*, *degree=2*, *interaction_only=False*, *include_bias=True*, *order='C'*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*)
Compute number of output features.

Parameters

X [array-like, shape (n_samples, n_features)] The data.

Returns

self [instance]

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_feature_names (*self*, *input_features=None*)
Return feature names for output features

Parameters

input_features [list of string, length n_features, optional] String names for input features if available. By default, "x0", "x1", ... "xn_features" is used.

Returns

output_feature_names [list of string, length n_output_features]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)

Transform data to polynomial features

Parameters

X [array-like or CSR/CSC sparse matrix, shape [n_samples, n_features]] The data to transform, row by row.

Prefer CSR over CSC for sparse input (for speed), but CSC is required if the degree is 4 or higher. If the degree is less than 4 and the input format is CSC, it will be converted to CSR, have its polynomial features generated, then converted back to CSC.

If the degree is 2 or 3, the method described in "Leveraging Sparsity to Speed Up Polynomial Feature Expansions of CSR Matrices Using K-Simplex Numbers" by Andrew Nystrom and John Hughes is used, which is much faster than the method used on CSC input. For this reason, a CSC input will be converted to CSR, and the output will be converted back to CSC prior to being returned, hence the preference of CSR.

Returns

XP [np.ndarray or CSR/CSC sparse matrix, shape [n_samples, NP]] The matrix of features, where NP is the number of polynomial features generated from the combination of inputs.

Examples using sklearn.preprocessing.PolynomialFeatures

- *Polynomial interpolation*
- *Robust linear estimator fitting*
- *Underfitting vs. Overfitting*

7.33.14 sklearn.preprocessing.PowerTransformer

class sklearn.preprocessing.**PowerTransformer**(*method='yeo-johnson'*, *standardize=True*, *copy=True*)

Apply a power transform featurewise to make data more Gaussian-like.

Power transforms are a family of parametric, monotonic transformations that are applied to make data more Gaussian-like. This is useful for modeling issues related to heteroscedasticity (non-constant variance), or other situations where normality is desired.

Currently, PowerTransformer supports the Box-Cox transform and the Yeo-Johnson transform. The optimal parameter for stabilizing variance and minimizing skewness is estimated through maximum likelihood.

Box-Cox requires input data to be strictly positive, while Yeo-Johnson supports both positive or negative data.

By default, zero-mean, unit-variance normalization is applied to the transformed data.

Read more in the *User Guide*.

New in version 0.20.

Parameters

method [str, (default='yeo-johnson')] The power transform method. Available methods are:

- 'yeo-johnson' [Rf3e1504535de-1], works with positive and negative values
- 'box-cox' [Rf3e1504535de-2], only works with strictly positive values

standardize [boolean, default=True] Set to True to apply zero-mean, unit-variance normalization to the transformed output.

copy [boolean, optional, default=True] Set to False to perform inplace computation during transformation.

Attributes

lambdas_ [array of float, shape (n_features,)] The parameters of the power transformation for the selected features.

See also:

power_transform Equivalent function without the estimator API.

QuantileTransformer Maps data to a standard normal distribution with the parameter `output_distribution='normal'`.

Notes

NaNs are treated as missing values: disregarded in `fit`, and maintained in `transform`.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

References

[Rf3e1504535de-1], [Rf3e1504535de-2]

Examples

```
>>> import numpy as np
>>> from sklearn.preprocessing import PowerTransformer
>>> pt = PowerTransformer()
>>> data = [[1, 2], [3, 2], [4, 5]]
>>> print(pt.fit(data))
PowerTransformer()
>>> print(pt.lambdas_)
[ 1.386... -3.100...]
>>> print(pt.transform(data))
[[-1.316... -0.707...]
 [ 0.209... -0.707...]
 [ 1.106...  1.414...]]
```

Methods

fit(self, X[, y])	Estimate the optimal parameter lambda for each feature.
fit_transform(self, X[, y])	Fit to data, then transform it.

Continued on next page

Table 277 – continued from previous page

get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Apply the inverse power transformation using the fitted lambdas.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply the power transform to each feature using the fitted lambdas.

__init__ (*self*, *method='yeo-johnson'*, *standardize=True*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*)
Estimate the optimal parameter lambda for each feature.

The optimal lambda parameter for minimizing skewness is estimated on each feature independently using maximum likelihood.

Parameters

X [array-like, shape (n_samples, n_features)] The data used to estimate the optimal transformation parameters.

y [Ignored]

Returns

self [object]

fit_transform (*self*, *X*, *y=None*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform (*self*, *X*)
Apply the inverse power transformation using the fitted lambdas.

The inverse of the Box-Cox transformation is given by:

```
if lambda_ == 0:
    X = exp(X_trans)
else:
    X = (X_trans * lambda_ + 1) ** (1 / lambda_)
```

The inverse of the Yeo-Johnson transformation is given by:

```
if X >= 0 and lambda_ == 0:
    X = exp(X_trans) - 1
elif X >= 0 and lambda_ != 0:
    X = (X_trans * lambda_ + 1) ** (1 / lambda_) - 1
elif X < 0 and lambda_ != 2:
    X = 1 - (-(2 - lambda_) * X_trans + 1) ** (1 / (2 - lambda_))
elif X < 0 and lambda_ == 2:
    X = 1 - exp(-X_trans)
```

Parameters

X [array-like, shape (n_samples, n_features)] The transformed data.

Returns

X [array-like, shape (n_samples, n_features)] The original data

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)

Apply the power transform to each feature using the fitted lambdas.

Parameters

X [array-like, shape (n_samples, n_features)] The data to be transformed using a power transformation.

Returns

X_trans [array-like, shape (n_samples, n_features)] The transformed data.

Examples using `sklearn.preprocessing.PowerTransformer`

- *Map data to a normal distribution*
- *Compare the effect of different scalers on data with outliers*

7.33.15 sklearn.preprocessing.QuantileTransformer

class sklearn.preprocessing.**QuantileTransformer**(*n_quantiles=1000*, *output_distribution='uniform'*, *ignore_implicit_zeros=False*, *subsample=100000*, *random_state=None*, *copy=True*)

Transform features using quantiles information.

This method transforms the features to follow a uniform or a normal distribution. Therefore, for a given feature, this transformation tends to spread out the most frequent values. It also reduces the impact of (marginal) outliers: this is therefore a robust preprocessing scheme.

The transformation is applied on each feature independently. First an estimate of the cumulative distribution function of a feature is used to map the original values to a uniform distribution. The obtained values are then mapped to the desired output distribution using the associated quantile function. Features values of new/unseen data that fall below or above the fitted range will be mapped to the bounds of the output distribution. Note that this transform is non-linear. It may distort linear correlations between variables measured at the same scale but renders variables measured at different scales more directly comparable.

Read more in the *User Guide*.

New in version 0.19.

Parameters

n_quantiles [int, optional (default=1000 or n_samples)] Number of quantiles to be computed. It corresponds to the number of landmarks used to discretize the cumulative distribution function. If n_quantiles is larger than the number of samples, n_quantiles is set to the number of samples as a larger number of quantiles does not give a better approximation of the cumulative distribution function estimator.

output_distribution [str, optional (default='uniform')] Marginal distribution for the transformed data. The choices are 'uniform' (default) or 'normal'.

ignore_implicit_zeros [bool, optional (default=False)] Only applies to sparse matrices. If True, the sparse entries of the matrix are discarded to compute the quantile statistics. If False, these entries are treated as zeros.

subsample [int, optional (default=1e5)] Maximum number of samples used to estimate the quantiles for computational efficiency. Note that the subsampling procedure may differ for value-identical sparse and dense matrices.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. Note that this is used by subsampling and smoothing noise.

copy [boolean, optional, (default=True)] Set to False to perform inplace transformation and avoid a copy (if the input is already a numpy array).

Attributes

n_quantiles_ [integer] The actual number of quantiles used to discretize the cumulative distribution function.

quantiles_ [ndarray, shape (n_quantiles, n_features)] The values corresponding the quantiles of reference.

references_ [ndarray, shape(n_quantiles,)] Quantiles of references.

See also:

quantile_transform Equivalent function without the estimator API.

PowerTransformer Perform mapping to a normal distribution using a power transform.

StandardScaler Perform standardization that is faster, but less robust to outliers.

RobustScaler Perform robust standardization that removes the influence of outliers but does not put outliers and inliers on the same scale.

Notes

NaNs are treated as missing values: disregarded in fit, and maintained in transform.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples

```
>>> import numpy as np
>>> from sklearn.preprocessing import QuantileTransformer
>>> rng = np.random.RandomState(0)
>>> X = np.sort(rng.normal(loc=0.5, scale=0.25, size=(25, 1)), axis=0)
>>> qt = QuantileTransformer(n_quantiles=10, random_state=0)
>>> qt.fit_transform(X)
array([...])
```

Methods

fit(self, X[, y])	Compute the quantiles used for transforming.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Back-projection to the original space.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Feature-wise transformation of the data.

__init__(*self*, *n_quantiles=1000*, *output_distribution='uniform'*, *ignore_implicit_zeros=False*, *subsample=100000*, *random_state=None*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Compute the quantiles used for transforming.

Parameters

X [ndarray or sparse matrix, shape (n_samples, n_features)] The data used to scale along the features axis. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`. Additionally, the sparse matrix needs to be nonnegative if `ignore_implicit_zeros` is False.

Returns

self [object]

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*)
Back-projection to the original space.

Parameters

X [ndarray or sparse matrix, shape (n_samples, n_features)] The data used to scale along the features axis. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`. Additionally, the sparse matrix needs to be nonnegative if `ignore_implicit_zeros` is False.

Returns

Xt [ndarray or sparse matrix, shape (n_samples, n_features)] The projected data.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Feature-wise transformation of the data.

Parameters

X [ndarray or sparse matrix, shape (n_samples, n_features)] The data used to scale along the features axis. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`. Additionally, the sparse matrix needs to be nonnegative if `ignore_implicit_zeros` is False.

Returns

Xt [ndarray or sparse matrix, shape (n_samples, n_features)] The projected data.

Examples using `sklearn.preprocessing.QuantileTransformer`

- *Partial Dependence Plots*
- *Effect of transforming the targets in regression model*
- *Map data to a normal distribution*
- *Compare the effect of different scalers on data with outliers*

7.33.16 `sklearn.preprocessing`.RobustScaler

class sklearn.preprocessing.**RobustScaler**(*with_centering=True*, *with_scaling=True*, *quantile_range=(25.0, 75.0)*, *copy=True*)

Scale features using statistics that are robust to outliers.

This Scaler removes the median and scales the data according to the quantile range (defaults to IQR: Interquartile Range). The IQR is the range between the 1st quartile (25th quantile) and the 3rd quartile (75th quantile).

Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Median and interquartile range are then stored to be used on later data using the `transform` method.

Standardization of a dataset is a common requirement for many machine learning estimators. Typically this is done by removing the mean and scaling to unit variance. However, outliers can often influence the sample mean / variance in a negative way. In such cases, the median and the interquartile range often give better results.

New in version 0.17.

Read more in the *User Guide*.

Parameters

with_centering [boolean, True by default] If True, center the data before scaling. This will cause `transform` to raise an exception when attempted on sparse matrices, because centering them entails building a dense matrix which in common use cases is likely to be too large to fit in memory.

with_scaling [boolean, True by default] If True, scale the data to interquartile range.

quantile_range [tuple (q_min, q_max), 0.0 < q_min < q_max < 100.0] Default: (25.0, 75.0) = (1st quantile, 3rd quantile) = IQR Quantile range used to calculate `scale_`.

New in version 0.18.

copy [boolean, optional, default is True] If False, try to avoid a copy and do inplace scaling instead. This is not guaranteed to always work inplace; e.g. if the data is not a NumPy array or scipy.sparse CSR matrix, a copy may still be returned.

Attributes

center_ [array of floats] The median value for each feature in the training set.

scale_ [array of floats] The (scaled) interquartile range for each feature in the training set.

New in version 0.17: *scale_* attribute.

See also:

robust_scale Equivalent function without the estimator API.

sklearn.decomposition.PCA Further removes the linear correlation across features with 'whiten=True'.

Notes

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

https://en.wikipedia.org/wiki/Median https://en.wikipedia.org/wiki/Interquartile_range

Examples

```
>>> from sklearn.preprocessing import RobustScaler
>>> X = [[ 1., -2.,  2.],
...      [ -2.,  1.,  3.],
...      [ 4.,  1., -2.]]
>>> transformer = RobustScaler().fit(X)
>>> transformer
RobustScaler()
>>> transformer.transform(X)
array([[ 0. , -2. ,  0. ],
       [-1. ,  0. ,  0.4],
       [ 1. ,  0. , -1.6]])
```

Methods

fit(self, X[, y])	Compute the median and quantiles to be used for scaling.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Scale back the data to the original representation
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Center and scale the data.

__init__(*self*, *with_centering=True*, *with_scaling=True*, *quantile_range=(25.0, 75.0)*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Compute the median and quantiles to be used for scaling.

Parameters

X [array-like, shape [n_samples, n_features]] The data used to compute the median and quantiles used for later scaling along the features axis.

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*)
Scale back the data to the original representation

Parameters

X [array-like] The data used to scale along the specified axis.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Center and scale the data.

Parameters

X [{array-like, sparse matrix}] The data used to scale along the specified axis.

Examples using `sklearn.preprocessing.RobustScaler`

- *Compare the effect of different scalers on data with outliers*

7.33.17 sklearn.preprocessing.StandardScaler

class sklearn.preprocessing.**StandardScaler**(*copy=True*, *with_mean=True*, *with_std=True*)
Standardize features by removing the mean and scaling to unit variance

The standard score of a sample x is calculated as:

z = (x - u) / s

where `u` is the mean of the training samples or zero if `with_mean=False`, and `s` is the standard deviation of the training samples or one if `with_std=False`.

Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Mean and standard deviation are then stored to be used on later data using *`transform`*.

Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual features do not more or less look like standard normally distributed data (e.g. Gaussian with 0 mean and unit variance).

For instance many elements used in the objective function of a learning algorithm (such as the RBF kernel of Support Vector Machines or the L1 and L2 regularizers of linear models) assume that all features are centered around 0 and have variance in the same order. If a feature has a variance that is orders of magnitude larger that others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

This scaler can also be applied to sparse CSR or CSC matrices by passing `with_mean=False` to avoid breaking the sparsity structure of the data.

Read more in the *User Guide*.

Parameters

copy [boolean, optional, default True] If False, try to avoid a copy and do inplace scaling instead. This is not guaranteed to always work inplace; e.g. if the data is not a NumPy array or scipy.sparse CSR matrix, a copy may still be returned.

with_mean [boolean, True by default] If True, center the data before scaling. This does not work (and will raise an exception) when attempted on sparse matrices, because centering them entails building a dense matrix which in common use cases is likely to be too large to fit in memory.

with_std [boolean, True by default] If True, scale the data to unit variance (or equivalently, unit standard deviation).

Attributes

scale_ [ndarray or None, shape (n_features,)] Per feature relative scaling of the data. This is calculated using `np.sqrt(var_)`. Equal to `None` when `with_std=False`.

New in version 0.17: *scale_*

mean_ [ndarray or None, shape (n_features,)] The mean value for each feature in the training set. Equal to `None` when `with_mean=False`.

var_ [ndarray or None, shape (n_features,)] The variance for each feature in the training set. Used to compute `scale_`. Equal to `None` when `with_std=False`.

n_samples_seen_ [int or array, shape (n_features,)] The number of samples processed by the estimator for each feature. If there are not missing samples, the `n_samples_seen` will be an integer, otherwise it will be an array. Will be reset on new calls to fit, but increments across `partial_fit` calls.

See also:

`scale` Equivalent function without the estimator API.

`sklearn.decomposition.PCA` Further removes the linear correlation across features with 'whiten=True'.

Notes

NaNs are treated as missing values: disregarded in fit, and maintained in transform.

We use a biased estimator for the standard deviation, equivalent to `numpy.std(x, ddof=0)`. Note that the choice of `ddof` is unlikely to affect model performance.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples

```
>>> from sklearn.preprocessing import StandardScaler
>>> data = [[0, 0], [0, 0], [1, 1], [1, 1]]
>>> scaler = StandardScaler()
>>> print(scaler.fit(data))
StandardScaler()
>>> print(scaler.mean_)
[0.5 0.5]
>>> print(scaler.transform(data))
[[-1. -1.]
 [-1. -1.]
 [ 1.  1.]
 [ 1.  1.]]
>>> print(scaler.transform([[2, 2]]))
[[3. 3.]]
```

Methods

fit(self, X[, y])	Compute the mean and std to be used for later scaling.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X[, copy])	Scale back the data to the original representation
partial_fit(self, X[, y])	Online computation of mean and std on X for later scaling.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, copy])	Perform standardization by centering and scaling

__init__(*self*, *copy=True*, *with_mean=True*, *with_std=True*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Compute the mean and std to be used for later scaling.

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data used to compute the mean and standard deviation used for later scaling along the features axis.

y Ignored

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*, *copy=None*)
Scale back the data to the original representation

Parameters

X [array-like, shape [n_samples, n_features]] The data used to scale along the features axis.

copy [bool, optional (default: None)] Copy the input X or not.

Returns

X_tr [array-like, shape [n_samples, n_features]] Transformed array.

partial_fit(*self*, *X*, *y=None*)
Online computation of mean and std on X for later scaling.

All of X is processed as a single batch. This is intended for cases when `fit` is not feasible due to very large number of `n_samples` or because X is read from a continuous stream.

The algorithm for incremental mean and std is given in Equation 1.5a,b in Chan, Tony F., Gene H. Golub, and Randall J. LeVeque. "Algorithms for computing the sample variance: Analysis and recommendations." The American Statistician 37.3 (1983): 242-247:

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data used to compute the mean and standard deviation used for later scaling along the features axis.

y [None] Ignored.

Returns

self [object] Transformer instance.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*, *copy=None*)
Perform standardization by centering and scaling

Parameters

X [array-like, shape [n_samples, n_features]] The data used to scale along the features axis.

copy [bool, optional (default: None)] Copy the input X or not.

Examples using `sklearn.preprocessing.StandardScaler`

- *Advanced Plotting With Partial Dependence*
- *Classifier comparison*
- *Demo of DBSCAN clustering algorithm*
- *Comparing different hierarchical linkage methods on toy datasets*
- *Comparing different clustering algorithms on toy datasets*
- *Prediction Latency*
- *MNIST classification using multinomial logistic + L1*
- *L1 Penalty and Sparsity in Logistic Regression*
- *Comparing Nearest Neighbors with and without Neighborhood Components Analysis*
- *Dimensionality Reduction with Neighborhood Components Analysis*
- *Varying regularization in Multi-layer Perceptron*
- *Column Transformer with Mixed Types*
- *Importance of Feature Scaling*
- *Feature discretization*
- *Compare the effect of different scalers on data with outliers*
- *Release Highlights for scikit-learn 0.22*
- *SVM-Anova: SVM with univariate feature selection*
- *RBF SVM parameters*

`preprocessing.add_dummy_feature`(X[, value])	Augment dataset with an additional dummy feature.
`preprocessing.binarize`(X[, threshold, copy])	Boolean thresholding of array-like or scipy.sparse matrix
`preprocessing.label_binarize`(y, classes[, ...])	Binarize labels in a one-vs-all fashion
`preprocessing.maxabs_scale`(X[, axis, copy])	Scale each feature to the [-1, 1] range without breaking the sparsity.
`preprocessing.minmax_scale`(X[, ...])	Transform features by scaling each feature to a given range.

Continued on next page

Table 281 – continued from previous page

preprocessing.normalize(X[, norm, axis, ...])	Scale input vectors individually to unit norm (vector length).
preprocessing.quantile_transform(X[, axis, ...])	Transform features using quantiles information.
preprocessing.robust_scale(X[, axis, ...])	Standardize a dataset along any axis
preprocessing.scale(X[, axis, with_mean, ...])	Standardize a dataset along any axis
preprocessing.power_transform(X[, method, ...])	Power transforms are a family of parametric, monotonic transformations that are applied to make data more Gaussian-like.

7.33.18 sklearn.preprocessing.add_dummy_feature

sklearn.preprocessing.**add_dummy_feature**(*X*, *value=1.0*)

Augment dataset with an additional dummy feature.

This is useful for fitting an intercept term with implementations which cannot otherwise fit it directly.

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] Data.

value [float] Value to use for the dummy feature.

Returns

X [{array, sparse matrix}, shape [n_samples, n_features + 1]] Same data with dummy feature added as first column.

Examples

```
>>> from sklearn.preprocessing import add_dummy_feature
>>> add_dummy_feature([[0, 1], [1, 0]])
array([[1., 0., 1.],
       [1., 1., 0.]])
```

7.33.19 sklearn.preprocessing.binarize

sklearn.preprocessing.**binarize**(*X*, *threshold=0.0*, *copy=True*)

Boolean thresholding of array-like or scipy.sparse matrix

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data to binarize, element by element. scipy.sparse matrices should be in CSR or CSC format to avoid an un-necessary copy.

threshold [float, optional (0.0 by default)] Feature values below or equal to this are replaced by 0, above it by 1. Threshold may not be less than 0 for operations on sparse matrices.

copy [boolean, optional, default True] set to False to perform inplace binarization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR / CSC matrix and if axis is 1).

See also:

Binarizer Performs binarization using the `Transformer` API (e.g. as part of a preprocessing *sklearn.pipeline.Pipeline*).

7.33.20 sklearn.preprocessing.label_binarize

sklearn.preprocessing.**label_binarize**(*y*, *classes*, *neg_label=0*, *pos_label=1*, *sparse_output=False*)

Binarize labels in a one-vs-all fashion

Several regression and binary classification algorithms are available in scikit-learn. A simple way to extend these algorithms to the multi-class classification case is to use the so-called one-vs-all scheme.

This function makes it possible to compute this transformation for a fixed set of class labels known ahead of time.

Parameters

y [array-like] Sequence of integer labels or multilabel data to encode.

classes [array-like of shape [n_classes]] Uniquely holds the label for each class.

neg_label [int (default: 0)] Value with which negative labels must be encoded.

pos_label [int (default: 1)] Value with which positive labels must be encoded.

sparse_output [boolean (default: False),] Set to true if output binary array is desired in CSR sparse format

Returns

Y [numpy array or CSR matrix of shape [n_samples, n_classes]] Shape will be [n_samples, 1] for binary problems.

See also:

LabelBinarizer class used to wrap the functionality of label_binarize and allow for fitting to classes independently of the transform operation

Examples

```
>>> from sklearn.preprocessing import label_binarize
>>> label_binarize([1, 6], classes=[1, 2, 4, 6])
array([[1, 0, 0, 0],
       [0, 0, 0, 1]])
```

The class ordering is preserved:

```
>>> label_binarize([1, 6], classes=[1, 6, 4, 2])
array([[1, 0, 0, 0],
       [0, 1, 0, 0]])
```

Binary targets transform to a column vector

```
>>> label_binarize(['yes', 'no', 'no', 'yes'], classes=['no', 'yes'])
array([[1],
       [0],
       [0],
       [1]])
```

Examples using `sklearn.preprocessing.label_binarize`

- *Receiver Operating Characteristic (ROC)*
- *Precision-Recall*

7.33.21 `sklearn.preprocessing`.maxabs_scale

`sklearn.preprocessing.`**`maxabs_scale`**(*X*, *axis=0*, *copy=True*)

Scale each feature to the [-1, 1] range without breaking the sparsity.

This estimator scales each feature individually such that the maximal absolute value of each feature in the training set will be 1.0.

This scaler can also be applied to sparse CSR or CSC matrices.

Parameters

X [array-like, shape (n_samples, n_features)] The data.

axis [int (0 by default)] axis used to scale along. If 0, independently scale each feature, otherwise (if 1) scale each sample.

copy [boolean, optional, default is True] Set to False to perform inplace scaling and avoid a copy (if the input is already a numpy array).

See also:

`MaxAbsScaler` Performs scaling to the [-1, 1] range using the‘‘Transformer‘‘ API (e.g. as part of a preprocessing *`sklearn.pipeline.Pipeline`*).

Notes

NaNs are treated as missing values: disregarded to compute the statistics, and maintained during the data transformation.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

7.33.22 `sklearn.preprocessing`.minmax_scale

`sklearn.preprocessing.`**`minmax_scale`**(*X*, *feature_range=(0, 1)*, *axis=0*, *copy=True*)

Transform features by scaling each feature to a given range.

This estimator scales and translates each feature individually such that it is in the given range on the training set, i.e. between zero and one.

The transformation is given by (when `axis=0`):

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

where min, max = feature_range.

The transformation is calculated as (when `axis=0`):

```
X_scaled = scale * X + min - X.min(axis=0) * scale
where scale = (max - min) / (X.max(axis=0) - X.min(axis=0))
```

This transformation is often used as an alternative to zero mean, unit variance scaling.

Read more in the *User Guide*.

New in version 0.17: *minmax_scale* function interface to `sklearn.preprocessing.MinMaxScaler`.

Parameters

X [array-like of shape (n_samples, n_features)] The data.

feature_range [tuple (min, max), default=(0, 1)] Desired range of transformed data.

axis [int, default=0] Axis used to scale along. If 0, independently scale each feature, otherwise (if 1) scale each sample.

copy [bool, default=True] Set to False to perform inplace scaling and avoid a copy (if the input is already a numpy array).

See also:

MinMaxScaler Performs scaling to a given range using the‘‘Transformer‘‘ API (e.g. as part of a preprocessing `sklearn.pipeline.Pipeline`).

Notes

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples using `sklearn.preprocessing.minmax_scale`

- *Compare the effect of different scalers on data with outliers*

7.33.23 `sklearn.preprocessing`.normalize

`sklearn.preprocessing.`**`normalize`**(*X*, *norm='l2'*, *axis=1*, *copy=True*, *return_norm=False*)

Scale input vectors individually to unit norm (vector length).

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data to normalize, element by element. scipy.sparse matrices should be in CSR format to avoid an un-necessary copy.

norm [‘l1’, ‘l2’, or ‘max’, optional (‘l2’ by default)] The norm to use to normalize each non zero sample (or each non-zero feature if axis is 0).

axis [0 or 1, optional (1 by default)] axis used to normalize the data along. If 1, independently normalize each sample, otherwise (if 0) normalize each feature.

copy [boolean, optional, default True] set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR matrix and if axis is 1).

return_norm [boolean, default False] whether to return the computed norms

Returns

X [{array-like, sparse matrix}, shape [n_samples, n_features]] Normalized input X.

norms [array, shape [n_samples] if axis=1 else [n_features]] An array of norms along given axis for X. When X is sparse, a NotImplementedError will be raised for norm 'l1' or 'l2'.

See also:

Normalizer Performs normalization using the `Transformer` API (e.g. as part of a preprocessing *sklearn.pipeline.Pipeline*).

Notes

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

7.33.24 sklearn.preprocessing.quantile_transform

sklearn.preprocessing.**quantile_transform**(*X*, *axis=0*, *n_quantiles=1000*, *output_distribution='uniform'*, *ignore_implicit_zeros=False*, *subsample=100000*, *random_state=None*, *copy='warn'*)

Transform features using quantiles information.

This method transforms the features to follow a uniform or a normal distribution. Therefore, for a given feature, this transformation tends to spread out the most frequent values. It also reduces the impact of (marginal) outliers: this is therefore a robust preprocessing scheme.

The transformation is applied on each feature independently. First an estimate of the cumulative distribution function of a feature is used to map the original values to a uniform distribution. The obtained values are then mapped to the desired output distribution using the associated quantile function. Features values of new/unseen data that fall below or above the fitted range will be mapped to the bounds of the output distribution. Note that this transform is non-linear. It may distort linear correlations between variables measured at the same scale but renders variables measured at different scales more directly comparable.

Read more in the *User Guide*.

Parameters

X [array-like, sparse matrix] The data to transform.

axis [int, (default=0)] Axis used to compute the means and standard deviations along. If 0, transform each feature, otherwise (if 1) transform each sample.

n_quantiles [int, optional (default=1000 or n_samples)] Number of quantiles to be computed. It corresponds to the number of landmarks used to discretize the cumulative distribution function. If n_quantiles is larger than the number of samples, n_quantiles is set to the number of samples as a larger number of quantiles does not give a better approximation of the cumulative distribution function estimator.

output_distribution [str, optional (default='uniform')] Marginal distribution for the transformed data. The choices are 'uniform' (default) or 'normal'.

ignore_implicit_zeros [bool, optional (default=False)] Only applies to sparse matrices. If True, the sparse entries of the matrix are discarded to compute the quantile statistics. If False, these entries are treated as zeros.

subsample [int, optional (default=1e5)] Maximum number of samples used to estimate the quantiles for computational efficiency. Note that the subsampling procedure may differ for value-identical sparse and dense matrices.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. Note that this is used by subsampling and smoothing noise.

copy [boolean, optional, (default="warn")] Set to False to perform inplace transformation and avoid a copy (if the input is already a numpy array). If True, a copy of `X` is transformed, leaving the original `X` unchanged

Deprecated since version 0.21: The default value of parameter `copy` will be changed from False to True in 0.23. The current default of False is being changed to make it more consistent with the default `copy` values of other functions in *`sklearn.preprocessing`*. Furthermore, the current default of False may have unexpected side effects by modifying the value of `X` inplace

Returns

Xt [ndarray or sparse matrix, shape (n_samples, n_features)] The transformed data.

See also:

`QuantileTransformer` Performs quantile-based scaling using the `Transformer` API (e.g. as part of a preprocessing *`sklearn.pipeline.Pipeline`*).

`power_transform` Maps data to a normal distribution using a power transformation.

`scale` Performs standardization that is faster, but less robust to outliers.

`robust_scale` Performs robust standardization that removes the influence of outliers but does not put outliers and inliers on the same scale.

Notes

NaNs are treated as missing values: disregarded in fit, and maintained in transform.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples

```
>>> import numpy as np
>>> from sklearn.preprocessing import quantile_transform
>>> rng = np.random.RandomState(0)
>>> X = np.sort(rng.normal(loc=0.5, scale=0.25, size=(25, 1)), axis=0)
>>> quantile_transform(X, n_quantiles=10, random_state=0, copy=True)
array([...])
```

Examples using `sklearn.preprocessing.quantile_transform`

- *Effect of transforming the targets in regression model*

7.33.25 sklearn.preprocessing.robust_scale

sklearn.preprocessing.**robust_scale**(*X*, *axis=0*, *with_centering=True*, *with_scaling=True*, *quantile_range=(25.0, 75.0)*, *copy=True*)

Standardize a dataset along any axis

Center to the median and component wise scale according to the interquartile range.

Read more in the *User Guide*.

Parameters

X [array-like] The data to center and scale.

axis [int (0 by default)] axis used to compute the medians and IQR along. If 0, independently scale each feature, otherwise (if 1) scale each sample.

with_centering [boolean, True by default] If True, center the data before scaling.

with_scaling [boolean, True by default] If True, scale the data to unit variance (or equivalently, unit standard deviation).

quantile_range [tuple (q_min, q_max), 0.0 < q_min < q_max < 100.0] Default: (25.0, 75.0) = (1st quantile, 3rd quantile) = IQR Quantile range used to calculate `scale_`.

New in version 0.18.

copy [boolean, optional, default is True] set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR matrix and if axis is 1).

See also:

RobustScaler Performs centering and scaling using the `Transformer` API (e.g. as part of a preprocessing *sklearn.pipeline.Pipeline*).

Notes

This implementation will refuse to center scipy.sparse matrices since it would make them non-sparse and would potentially crash the program with memory exhaustion problems.

Instead the caller is expected to either set explicitly `with_centering=False` (in that case, only variance scaling will be performed on the features of the CSR matrix) or to call `X.toarray()` if he/she expects the materialized dense array to fit in memory.

To avoid memory copy the caller should pass a CSR matrix.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

7.33.26 sklearn.preprocessing.scale

sklearn.preprocessing.**scale**(*X*, *axis=0*, *with_mean=True*, *with_std=True*, *copy=True*)

Standardize a dataset along any axis

Center to the mean and component wise scale to unit variance.

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix}] The data to center and scale.

axis [int (0 by default)] axis used to compute the means and standard deviations along. If 0, independently standardize each feature, otherwise (if 1) standardize each sample.

with_mean [boolean, True by default] If True, center the data before scaling.

with_std [boolean, True by default] If True, scale the data to unit variance (or equivalently, unit standard deviation).

copy [boolean, optional, default True] set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSC matrix and if axis is 1).

See also:

StandardScaler Performs scaling to unit variance using the‘‘Transformer‘‘ API (e.g. as part of a preprocessing *sklearn.pipeline.Pipeline*).

Notes

This implementation will refuse to center scipy.sparse matrices since it would make them non-sparse and would potentially crash the program with memory exhaustion problems.

Instead the caller is expected to either set explicitly `with_mean=False` (in that case, only variance scaling will be performed on the features of the CSC matrix) or to call `X.toarray()` if he/she expects the materialized dense array to fit in memory.

To avoid memory copy the caller should pass a CSC matrix.

NaNs are treated as missing values: disregarded to compute the statistics, and maintained during the data transformation.

We use a biased estimator for the standard deviation, equivalent to `numpy.std(x, ddof=0)`. Note that the choice of `ddof` is unlikely to affect model performance.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples using `sklearn.preprocessing.scale`

- *A demo of K-Means clustering on the handwritten digits data*

7.33.27 `sklearn.preprocessing`.power_transform

`sklearn.preprocessing.`**`power_transform`**(*X*, *method='warn'*, *standardize=True*, *copy=True*)

Power transforms are a family of parametric, monotonic transformations that are applied to make data more Gaussian-like. This is useful for modeling issues related to heteroscedasticity (non-constant variance), or other situations where normality is desired.

Currently, power_transform supports the Box-Cox transform and the Yeo-Johnson transform. The optimal parameter for stabilizing variance and minimizing skewness is estimated through maximum likelihood.

Box-Cox requires input data to be strictly positive, while Yeo-Johnson supports both positive or negative data.

By default, zero-mean, unit-variance normalization is applied to the transformed data.

Read more in the *User Guide*.

Parameters

X [array-like, shape (n_samples, n_features)] The data to be transformed using a power transformation.

method [str] The power transform method. Available methods are:

- 'yeo-johnson' [1], works with positive and negative values
- 'box-cox' [2], only works with strictly positive values

The default method will be changed from 'box-cox' to 'yeo-johnson' in version 0.23. To suppress the FutureWarning, explicitly set the parameter.

standardize [boolean, default=True] Set to True to apply zero-mean, unit-variance normalization to the transformed output.

copy [boolean, optional, default=True] Set to False to perform inplace computation during transformation.

Returns

X_trans [array-like, shape (n_samples, n_features)] The transformed data.

See also:

PowerTransformer Equivalent transformation with the `Transformer` API (e.g. as part of a preprocessing *sklearn.pipeline.Pipeline*).

quantile_transform Maps data to a standard normal distribution with the parameter `output_distribution='normal'`.

Notes

NaNs are treated as missing values: disregarded in `fit`, and maintained in `transform`.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

References

[1], [2]

Examples

```
>>> import numpy as np
>>> from sklearn.preprocessing import power_transform
>>> data = [[1, 2], [3, 2], [4, 5]]
>>> print(power_transform(data, method='box-cox'))
[[-1.332... -0.707...]
 [ 0.256... -0.707...]
 [ 1.076...  1.414...]]
```

7.34 sklearn.random_projection: Random projection

Random Projection transformers

Random Projections are a simple and computationally efficient way to reduce the dimensionality of the data by trading a controlled amount of accuracy (as additional variance) for faster processing times and smaller model sizes.

The dimensions and distribution of Random Projections matrices are controlled so as to preserve the pairwise distances between any two samples of the dataset.

The main theoretical result behind the efficiency of random projection is the Johnson-Lindenstrauss lemma (quoting Wikipedia):

> In mathematics, the Johnson-Lindenstrauss lemma is a result concerning low-distortion embeddings of points from high-dimensional into low-dimensional Euclidean space. The lemma states that a small set of points in a high-dimensional space can be embedded into a space of much lower dimension in such a way that distances between the points are nearly preserved. The map used for the embedding is at least Lipschitz, and can even be taken to be an orthogonal projection.

User guide: See the *Random Projection* section for further details.

random_projection.GaussianRandomProjection([...])	Reduce dimensionality through Gaussian random projection
random_projection.SparseRandomProjection([...])	Reduce dimensionality through sparse random projection

7.34.1 sklearn.random_projection.GaussianRandomProjection

class sklearn.random_projection.**GaussianRandomProjection**(*n_components='auto'*, *eps=0.1*, *random_state=None*)

Reduce dimensionality through Gaussian random projection

The components of the random matrix are drawn from N(0, 1 / n_components).

Read more in the *User Guide*.

New in version 0.13.

Parameters

n_components [int or 'auto', optional (default = 'auto')] Dimensionality of the target projection space.

n_components can be automatically adjusted according to the number of samples in the dataset and the bound given by the Johnson-Lindenstrauss lemma. In that case the quality of the embedding is controlled by the `eps` parameter.

It should be noted that Johnson-Lindenstrauss lemma can yield very conservative estimated of the required number of components as it makes no assumption on the structure of the dataset.

eps [strictly positive float, optional (default=0.1)] Parameter to control the quality of the embedding according to the Johnson-Lindenstrauss lemma when n_components is set to 'auto'.

Smaller values lead to better embedding and higher number of dimensions (n_components) in the target projection space.

random_state [int, RandomState instance or None, optional (default=None)] Control the pseudo random number generator used to generate the matrix at fit time. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Attributes

n_components_ [int] Concrete number of components computed when n_components="auto".

components_ [numpy array of shape [n_components, n_features]] Random matrix used for the projection.

See also:

SparseRandomProjection

Examples

```
>>> import numpy as np
>>> from sklearn.random_projection import GaussianRandomProjection
>>> rng = np.random.RandomState(42)
>>> X = rng.rand(100, 10000)
>>> transformer = GaussianRandomProjection(random_state=rng)
>>> X_new = transformer.fit_transform(X)
>>> X_new.shape
(100, 3947)
```

Methods

fit(self, X[, y])	Generate a sparse random projection matrix
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Project the data by using matrix product with the random matrix

__init__ (*self*, *n_components='auto'*, *eps=0.1*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y=None*)
Generate a sparse random projection matrix

Parameters

X [numpy array or scipy.sparse of shape [n_samples, n_features]] Training set: only the shape is used to find optimal random matrix dimensions based on the theory referenced in the afore mentioned papers.

y Ignored

Returns

self

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Project the data by using matrix product with the random matrix

Parameters

X [numpy array or scipy.sparse of shape [n_samples, n_features]] The input data to project into a smaller dimensional space.

Returns

X_new [numpy array or scipy sparse of shape [n_samples, n_components]] Projected array.

7.34.2 sklearn.random_projection.SparseRandomProjection

class sklearn.random_projection.**SparseRandomProjection**(*n_components='auto'*, *density='auto'*, *eps=0.1*, *dense_output=False*, *random_state=None*)

Reduce dimensionality through sparse random projection

Sparse random matrix is an alternative to dense random projection matrix that guarantees similar embedding quality while being much more memory efficient and allowing faster computation of the projected data.

If we note `s = 1 / density` the components of the random matrix are drawn from:

- -sqrt(s) / sqrt(n_components) with probability 1 / 2s
- 0 with probability 1 - 1 / s
- +sqrt(s) / sqrt(n_components) with probability 1 / 2s

Read more in the *User Guide*.

New in version 0.13.

Parameters

n_components [int or 'auto', optional (default = 'auto')] Dimensionality of the target projection space.

n_components can be automatically adjusted according to the number of samples in the dataset and the bound given by the Johnson-Lindenstrauss lemma. In that case the quality of the embedding is controlled by the `eps` parameter.

It should be noted that Johnson-Lindenstrauss lemma can yield very conservative estimated of the required number of components as it makes no assumption on the structure of the dataset.

density [float in range]0, 1], optional (default='auto')] Ratio of non-zero component in the random projection matrix.

If density = 'auto', the value is set to the minimum density as recommended by Ping Li et al.: 1 / sqrt(n_features).

Use density = 1 / 3.0 if you want to reproduce the results from Achlioptas, 2001.

eps [strictly positive float, optional, (default=0.1)] Parameter to control the quality of the embedding according to the Johnson-Lindenstrauss lemma when n_components is set to 'auto'.

Smaller values lead to better embedding and higher number of dimensions (n_components) in the target projection space.

dense_output [boolean, optional (default=False)] If True, ensure that the output of the random projection is a dense numpy array even if the input and random projection matrix are both sparse. In practice, if the number of components is small the number of zero components in the projected data will be very small and it will be more CPU and memory efficient to use a dense representation.

If False, the projected data uses a sparse representation if the input is sparse.

random_state [int, RandomState instance or None, optional (default=None)] Control the pseudo random number generator used to generate the matrix at fit time. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Attributes

n_components_ [int] Concrete number of components computed when n_components="auto".

components_ [CSR matrix with shape [n_components, n_features]] Random matrix used for the projection.

density_ [float in range 0.0 - 1.0] Concrete density computed from when density = "auto".

See also:

GaussianRandomProjection

References

[R0fecf191e4b8-1], [R0fecf191e4b8-2]

Examples

```
>>> import numpy as np
>>> from sklearn.random_projection import SparseRandomProjection
>>> rng = np.random.RandomState(42)
>>> X = rng.rand(100, 10000)
>>> transformer = SparseRandomProjection(random_state=rng)
>>> X_new = transformer.fit_transform(X)
>>> X_new.shape
(100, 3947)
>>> # very few components are non-zero
>>> np.mean(transformer.components_ != 0)
0.0100...
```

Methods

fit(self, X[, y])	Generate a sparse random projection matrix
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Project the data by using matrix product with the random matrix

__init__(*self*, *n_components='auto'*, *density='auto'*, *eps=0.1*, *dense_output=False*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y=None*)
Generate a sparse random projection matrix

Parameters

X [numpy array or scipy.sparse of shape [n_samples, n_features]] Training set: only the shape is used to find optimal random matrix dimensions based on the theory referenced in the afore mentioned papers.

y Ignored

Returns

self

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*self*, *X*)
Project the data by using matrix product with the random matrix

Parameters

X [numpy array or scipy.sparse of shape [n_samples, n_features]] The input data to project into a smaller dimensional space.

Returns

X_new [numpy array or scipy sparse of shape [n_samples, n_components]] Projected array.

Examples using `sklearn.random_projection.SparseRandomProjection`

- *The Johnson-Lindenstrauss bound for embedding with random projections*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

random_projection.johnson_lindenstrauss_min_dim(...)	Find a 'safe' number of components to randomly project to

7.34.3 sklearn.random_projection.johnson_lindenstrauss_min_dim

sklearn.random_projection.**johnson_lindenstrauss_min_dim**(*n_samples*, *eps=0.1*)
Find a 'safe' number of components to randomly project to

The distortion introduced by a random projection `p` only changes the distance between two points by a factor (1 +- eps) in an euclidean space with good probability. The projection `p` is an eps-embedding as defined by:

(1 - eps) ||u - v||^2 < ||p(u) - p(v)||^2 < (1 + eps) ||u - v||^2

Where u and v are any rows taken from a dataset of shape [n_samples, n_features], eps is in]0, 1[and p is a projection by a random Gaussian N(0, 1) matrix with shape [n_components, n_features] (or a sparse Achlioptas matrix).

The minimum number of components to guarantee the eps-embedding is given by:

n_components >= 4 log(n_samples) / (eps^2 / 2 - eps^3 / 3)

Note that the number of dimensions is independent of the original number of features but instead depends on the size of the dataset: the larger the dataset, the higher is the minimal dimensionality of an eps-embedding.

Read more in the *User Guide*.

Parameters

n_samples [int or numpy array of int greater than 0,] Number of samples. If an array is given, it will compute a safe number of components array-wise.

eps [float or numpy array of float in]0,1[, optional (default=0.1)] Maximum distortion rate as defined by the Johnson-Lindenstrauss lemma. If an array is given, it will compute a safe number of components array-wise.

Returns

n_components [int or numpy array of int,] The minimal number of components to guarantee with good probability an eps-embedding with n_samples.

References

[1], [2]

Examples

```
>>> johnson_lindenstrauss_min_dim(1e6, eps=0.5)
663
```

```
>>> johnson_lindenstrauss_min_dim(1e6, eps=[0.5, 0.1, 0.01])
array([    663,   11841, 1112658])
```

```
>>> johnson_lindenstrauss_min_dim([1e4, 1e5, 1e6], eps=0.1)
array([ 7894,  9868, 11841])
```

Examples using `sklearn.random_projection.johnson_lindenstrauss_min_dim`

- *The Johnson-Lindenstrauss bound for embedding with random projections*

7.35 `sklearn.semi_supervised` Semi-Supervised Learning

The *`sklearn.semi_supervised`* module implements semi-supervised learning algorithms. These algorithms utilized small amounts of labeled data and large amounts of unlabeled data for classification tasks. This module includes Label Propagation.

User guide: See the *Semi-Supervised* section for further details.

semi_supervised.LabelPropagation([kernel, ...])	Label Propagation classifier
semi_supervised.LabelSpreading([kernel, ...])	LabelSpreading model for semi-supervised learning

7.35.1 sklearn.semi_supervised.LabelPropagation

class sklearn.semi_supervised.**LabelPropagation**(*kernel='rbf'*, *gamma=20*, *n_neighbors=7*, *max_iter=1000*, *tol=0.001*, *n_jobs=None*)

Label Propagation classifier

Read more in the *User Guide*.

Parameters

kernel [{'knn', 'rbf', callable}] String identifier for kernel function to use or the kernel function itself. Only 'rbf' and 'knn' strings are valid inputs. The function passed should take two inputs, each of shape [n_samples, n_features], and return a [n_samples, n_samples] shaped weight matrix.

gamma [float] Parameter for rbf kernel

n_neighbors [integer > 0] Parameter for knn kernel

max_iter [integer] Change maximum number of iterations allowed

tol [float] Convergence tolerance: threshold to consider the system at steady state

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

X_ [array, shape = [n_samples, n_features]] Input array.

classes_ [array, shape = [n_classes]] The distinct labels used in classifying instances.

label_distributions_ [array, shape = [n_samples, n_classes]] Categorical distribution for each item.

transduction_ [array, shape = [n_samples]] Label assigned to each item via the transduction.

n_iter_ [int] Number of iterations run.

See also:

LabelSpreading Alternate label propagation strategy more robust to noise

References

Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical Report CMU-CALD-02-107, Carnegie Mellon University, 2002 http://pages.cs.wisc.edu/~jerryzhu/pub/CMU-CALD-02-107.pdf

Examples

```
>>> import numpy as np
>>> from sklearn import datasets
>>> from sklearn.semi_supervised import LabelPropagation
>>> label_prop_model = LabelPropagation()
>>> iris = datasets.load_iris()
>>> rng = np.random.RandomState(42)
>>> random_unlabeled_points = rng.rand(len(iris.target)) < 0.3
>>> labels = np.copy(iris.target)
>>> labels[random_unlabeled_points] = -1
>>> label_prop_model.fit(iris.data, labels)
LabelPropagation(...)
```

Methods

fit(self, X, y)	Fit a semi-supervised label propagation model based
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Performs inductive inference across the model.
predict_proba(self, X)	Predict probability for each possible outcome.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *kernel='rbf'*, *gamma=20*, *n_neighbors=7*, *max_iter=1000*, *tol=0.001*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*)
Fit a semi-supervised label propagation model based

All the input data is provided matrix X (labeled and unlabeled) and corresponding label matrix y with a dedicated marker value for unlabeled samples.

Parameters

X [array-like of shape (n_samples, n_features)] A {n_samples by n_samples} size matrix will be created from this

y [array_like, shape = [n_samples]] n_labeled_samples (unlabeled points are marked as -1) All unlabeled samples will be transductively assigned labels

Returns

self [returns an instance of self.]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Performs inductive inference across the model.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

y [array_like, shape = [n_samples]] Predictions for input data

predict_proba(*self*, *X*)

Predict probability for each possible outcome.

Compute the probability estimates for each single sample in X and each possible outcome seen during training (categorical distribution).

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

probabilities [array, shape = [n_samples, n_classes]] Normalized probability distributions across class labels

score(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

7.35.2 sklearn.semi_supervised.LabelSpreading

class sklearn.semi_supervised.**LabelSpreading**(*kernel='rbf'*, *gamma=20*, *n_neighbors=7*, *alpha=0.2*, *max_iter=30*, *tol=0.001*, *n_jobs=None*)

LabelSpreading model for semi-supervised learning

This model is similar to the basic Label Propagation algorithm, but uses affinity matrix based on the normalized graph Laplacian and soft clamping across the labels.

Read more in the *User Guide*.

Parameters

kernel [{'knn', 'rbf', callable}] String identifier for kernel function to use or the kernel function itself. Only 'rbf' and 'knn' strings are valid inputs. The function passed should take two inputs, each of shape [n_samples, n_features], and return a [n_samples, n_samples] shaped weight matrix

gamma [float] parameter for rbf kernel

n_neighbors [integer > 0] parameter for knn kernel

alpha [float] Clamping factor. A value in (0, 1) that specifies the relative amount that an instance should adopt the information from its neighbors as opposed to its initial label. alpha=0 means keeping the initial label information; alpha=1 means replacing all initial information.

max_iter [integer] maximum number of iterations allowed

tol [float] Convergence tolerance: threshold to consider the system at steady state

n_jobs [int or None, optional (default=None)] The number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

X_ [array, shape = [n_samples, n_features]] Input array.

classes_ [array, shape = [n_classes]] The distinct labels used in classifying instances.

label_distributions_ [array, shape = [n_samples, n_classes]] Categorical distribution for each item.

transduction_ [array, shape = [n_samples]] Label assigned to each item via the transduction.

n_iter_ [int] Number of iterations run.

See also:

LabelPropagation Unregularized graph based semi-supervised learning

References

Dengyong Zhou, Olivier Bousquet, Thomas Navin Lal, Jason Weston, Bernhard Schoelkopf. Learning with local and global consistency (2004) http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.115.3219

Examples

```
>>> import numpy as np
>>> from sklearn import datasets
>>> from sklearn.semi_supervised import LabelSpreading
>>> label_prop_model = LabelSpreading()
>>> iris = datasets.load_iris()
>>> rng = np.random.RandomState(42)
>>> random_unlabeled_points = rng.rand(len(iris.target)) < 0.3
>>> labels = np.copy(iris.target)
>>> labels[random_unlabeled_points] = -1
>>> label_prop_model.fit(iris.data, labels)
LabelSpreading(...)
```

Methods

fit(self, X, y)	Fit a semi-supervised label propagation model based
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Performs inductive inference across the model.
predict_proba(self, X)	Predict probability for each possible outcome.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *kernel='rbf'*, *gamma=20*, *n_neighbors=7*, *alpha=0.2*, *max_iter=30*, *tol=0.001*, *n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*)
Fit a semi-supervised label propagation model based

All the input data is provided matrix X (labeled and unlabeled) and corresponding label matrix y with a dedicated marker value for unlabeled samples.

Parameters

X [array-like of shape (n_samples, n_features)] A {n_samples by n_samples} size matrix will be created from this

y [array_like, shape = [n_samples]] n_labeled_samples (unlabeled points are marked as -1) All unlabeled samples will be transductively assigned labels

Returns

self [returns an instance of self.]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Performs inductive inference across the model.

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

y [array_like, shape = [n_samples]] Predictions for input data

predict_proba(*self*, *X*)
Predict probability for each possible outcome.

Compute the probability estimates for each single sample in X and each possible outcome seen during training (categorical distribution).

Parameters

X [array-like of shape (n_samples, n_features)]

Returns

probabilities [array, shape = [n_samples, n_classes]] Normalized probability distributions across class labels

score (*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.semi_supervised.LabelSpreading`

- *Decision boundary of label propagation versus SVM on the Iris dataset*
- *Label Propagation learning a complex structure*
- *Label Propagation digits: Demonstrating performance*
- *Label Propagation digits active learning*

7.36 `sklearn.svm`: Support Vector Machines

The `sklearn.svm` module includes Support Vector Machine algorithms.

User guide: See the *Support Vector Machines* section for further details.

7.36.1 Estimators

`svm.LinearSVC`([penalty, loss, dual, tol, C, ...])	Linear Support Vector Classification.

Continued on next page

Table 289 – continued from previous page

svm.LinearSVR([epsilon, tol, C, loss, ...])	Linear Support Vector Regression.
svm.NuSVC([nu, kernel, degree, gamma, ...])	Nu-Support Vector Classification.
svm.NuSVR([nu, C, kernel, degree, gamma, ...])	Nu Support Vector Regression.
svm.OneClassSVM([kernel, degree, gamma, ...])	Unsupervised Outlier Detection.
svm.SVC([C, kernel, degree, gamma, coef0, ...])	C-Support Vector Classification.
svm.SVR([kernel, degree, gamma, coef0, tol, ...])	Epsilon-Support Vector Regression.

sklearn.svm.LinearSVC

class sklearn.svm.**LinearSVC**(*penalty='l2'*, *loss='squared_hinge'*, *dual=True*, *tol=0.0001*, *C=1.0*, *multi_class='ovr'*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *verbose=0*, *random_state=None*, *max_iter=1000*)

Linear Support Vector Classification.

Similar to SVC with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

This class supports both dense and sparse input and the multiclass support is handled according to a one-vs-the-rest scheme.

Read more in the *User Guide*.

Parameters

penalty [str, 'l1' or 'l2' (default='l2')] Specifies the norm used in the penalization. The 'l2' penalty is the standard used in SVC. The 'l1' leads to `coef_` vectors that are sparse.

loss [str, 'hinge' or 'squared_hinge' (default='squared_hinge')] Specifies the loss function. 'hinge' is the standard SVM loss (used e.g. by the SVC class) while 'squared_hinge' is the square of the hinge loss.

dual [bool, (default=True)] Select the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n_samples > n_features.

tol [float, optional (default=1e-4)] Tolerance for stopping criteria.

C [float, optional (default=1.0)] Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive.

multi_class [str, 'ovr' or 'crammer_singer' (default='ovr')] Determines the multi-class strategy if `y` contains more than two classes. `"ovr"` trains n_classes one-vs-rest classifiers, while `"crammer_singer"` optimizes a joint objective over all classes. While `crammer_singer` is interesting from a theoretical perspective as it is consistent, it is seldom used in practice as it rarely leads to better accuracy and is more expensive to compute. If `"crammer_singer"` is chosen, the options loss, penalty and dual will be ignored.

fit_intercept [bool, optional (default=True)] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be already centered).

intercept_scaling [float, optional (default=1)] When self.fit_intercept is True, instance vector x becomes `[x, self.intercept_scaling]`, i.e. a "synthetic" feature with constant value equals to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.

class_weight [{dict, 'balanced'}, optional] Set the parameter C of class i to `class_weight[i]*C` for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

verbose [int, (default=0)] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in liblinear that, if enabled, may not work properly in a multithreaded context.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data for the dual coordinate descent (if `dual=True`). When `dual=False` the underlying implementation of *LinearSVC* is not random and `random_state` has no effect on the results. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

max_iter [int, (default=1000)] The maximum number of iterations to be run.

Attributes

coef_ [array, shape = [1, n_features] if n_classes == 2 else [n_classes, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

`coef_` is a readonly property derived from `raw_coef_` that follows the internal memory layout of liblinear.

intercept_ [array, shape = [1] if n_classes == 2 else [n_classes]] Constants in decision function.

classes_ [array of shape (n_classes,)] The unique classes labels.

n_iter_ [int] Maximum number of iterations run across all classes.

See also:

SVC Implementation of Support Vector Machine classifier using libsvm: the kernel can be non-linear but its SMO algorithm does not scale to large number of samples as LinearSVC does. Furthermore SVC multi-class mode is implemented using one vs one scheme while LinearSVC uses one vs the rest. It is possible to implement one vs the rest with SVC by using the *sklearn.multiclass.OneVsRestClassifier* wrapper. Finally SVC can fit dense data without memory copy if the input is C-contiguous. Sparse data will still incur memory copy though.

sklearn.linear_model.SGDClassifier SGDClassifier can optimize the same cost function as LinearSVC by adjusting the penalty and loss parameters. In addition it requires less memory, allows incremental (online) learning, and implements various loss functions and regularization regimes.

Notes

The underlying C implementation uses a random number generator to select features when fitting the model. It is thus not uncommon to have slightly different results for the same input data. If that happens, try with a smaller `tol` parameter.

The underlying implementation, liblinear, uses a sparse internal representation for the data that will incur a memory copy.

Predict output may not match that of standalone liblinear in certain cases. See *differences from liblinear* in the narrative documentation.

References

LIBLINEAR: A Library for Large Linear Classification

Examples

```
>>> from sklearn.svm import LinearSVC
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_features=4, random_state=0)
>>> clf = LinearSVC(random_state=0, tol=1e-5)
>>> clf.fit(X, y)
LinearSVC(random_state=0, tol=1e-05)
>>> print(clf.coef_)
[[0.085... 0.394... 0.498... 0.375...]]
>>> print(clf.intercept_)
[0.284...]
>>> print(clf.predict([[0, 0, 0, 0]]))
[1]
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
densify(self)	Convert coefficient matrix to dense array format.
fit(self, X, y[, sample_weight])	Fit the model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class labels for samples in X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.
sparsify(self)	Convert coefficient matrix to sparse format.

__init__ (*self*, *penalty='l2'*, *loss='squared_hinge'*, *dual=True*, *tol=0.0001*, *C=1.0*, *multi_class='ovr'*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *verbose=0*, *random_state=None*, *max_iter=1000*)

Initialize self. See help(type(self)) for accurate signature.

decision_function (*self*, *X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

densify (*self*)

Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is

required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns

self Fitted estimator.

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit the model according to the given training data.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [array-like of shape (n_samples,)] Target vector relative to X.

sample_weight [array-like of shape (n_samples,), default=None] Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight.

Returns

self [object] An instance of the estimator.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

score (*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params (*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

sparsify(*self*)

Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual numpy.ndarray representation.

The `intercept_` member is not converted.

Returns

self Fitted estimator.

Notes

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the partial_fit method (if any) will not work until you call densify.

Examples using `sklearn.svm.LinearSVC`

- *Explicit feature map approximation for RBF kernels*
- *Comparison of Calibration of Classifiers*
- *Probability Calibration curves*
- *Pipeline Anova SVM*
- *Univariate Feature Selection*
- *Balance model complexity and cross-validated score*
- *Precision-Recall*
- *Selecting dimensionality reduction with Pipeline and GridSearchCV*
- *Column Transformer with Heterogeneous Data Sources*
- *Feature discretization*
- *Release Highlights for scikit-learn 0.22*
- *Plot the support vectors in LinearSVC*
- *Plot different SVM classifiers in the iris dataset*
- *Scaling the regularization parameter for SVCs*
- *Classification of text documents using sparse features*

sklearn.svm.LinearSVR

class sklearn.svm.**LinearSVR**(*epsilon=0.0*, *tol=0.0001*, *C=1.0*, *loss='epsilon_insensitive'*, *fit_intercept=True*, *intercept_scaling=1.0*, *dual=True*, *verbose=0*, *random_state=None*, *max_iter=1000*)

Linear Support Vector Regression.

Similar to SVR with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

This class supports both dense and sparse input.

Read more in the *User Guide*.

New in version 0.16.

Parameters

epsilon [float, optional (default=0.0)] Epsilon parameter in the epsilon-insensitive loss function. Note that the value of this parameter depends on the scale of the target variable y. If unsure, set `epsilon=0`.

tol [float, optional (default=1e-4)] Tolerance for stopping criteria.

C [float, optional (default=1.0)] Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive.

loss [string, optional (default='epsilon_insensitive')] Specifies the loss function. The epsilon-insensitive loss (standard SVR) is the L1 loss, while the squared epsilon-insensitive loss ('squared_epsilon_insensitive') is the L2 loss.

fit_intercept [boolean, optional (default=True)] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be already centered).

intercept_scaling [float, optional (default=1)] When self.fit_intercept is True, instance vector x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equals to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.

dual [bool, (default=True)] Select the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n_samples > n_features.

verbose [int, (default=0)] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in liblinear that, if enabled, may not work properly in a multithreaded context.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

max_iter [int, (default=1000)] The maximum number of iterations to be run.

Attributes

coef_ [array, shape = [n_features] if n_classes == 2 else [n_classes, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

`coef_` is a readonly property derived from `raw_coef_` that follows the internal memory layout of liblinear.

intercept_ [array, shape = [1] if n_classes == 2 else [n_classes]] Constants in decision function.

n_iter_ [int] Maximum number of iterations run across all classes.

See also:

LinearSVC Implementation of Support Vector Machine classifier using the same library as this class (liblinear).

SVR Implementation of Support Vector Machine regression using libsvm: the kernel can be non-linear but its SMO algorithm does not scale to large number of samples as LinearSVC does.

sklearn.linear_model.SGDRegressor SGDRegressor can optimize the same cost function as LinearSVR by adjusting the penalty and loss parameters. In addition it requires less memory, allows incremental (online) learning, and implements various loss functions and regularization regimes.

Examples

```
>>> from sklearn.svm import LinearSVR
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(n_features=4, random_state=0)
>>> regr = LinearSVR(random_state=0, tol=1e-5)
>>> regr.fit(X, y)
LinearSVR(random_state=0, tol=1e-05)
>>> print(regr.coef_)
[16.35... 26.91... 42.30... 60.47...]
>>> print(regr.intercept_)
[-4.29...]
>>> print(regr.predict([[0, 0, 0, 0]]))
[-4.29...]
```

Methods

fit(self, X, y[, sample_weight])	Fit the model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *epsilon=0.0*, *tol=0.0001*, *C=1.0*, *loss='epsilon_insensitive'*, *fit_intercept=True*, *intercept_scaling=1.0*, *dual=True*, *verbose=0*, *random_state=None*, *max_iter=1000*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit the model according to the given training data.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [array-like of shape (n_samples,)] Target vector relative to X

sample_weight [array-like of shape (n_samples,), default=None] Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Predict using the linear model.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

sklearn.svm.NuSVC

class sklearn.svm.**NuSVC**(*nu=0.5*, *kernel='rbf'*, *degree=3*, *gamma='scale'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=-1*, *decision_function_shape='ovr'*, *break_ties=False*, *random_state=None*)

Nu-Support Vector Classification.

Similar to SVC but uses a parameter to control the number of support vectors.

The implementation is based on libsvm.

Read more in the *User Guide*.

Parameters

nu [float, optional (default=0.5)] An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1].

kernel [string, optional (default='rbf')] Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to precompute the kernel matrix.

degree [int, optional (default=3)] Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma [{'scale', 'auto'} or float, optional (default='scale')] Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

- if `gamma='scale'` (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
- if 'auto', uses 1 / n_features.

Changed in version 0.22: The default value of `gamma` changed from 'auto' to 'scale'.

coef0 [float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

shrinking [boolean, optional (default=True)] Whether to use the shrinking heuristic.

probability [boolean, optional (default=False)] Whether to enable probability estimates. This must be enabled prior to calling `fit`, will slow down that method as it internally uses 5-fold cross-validation, and `predict_proba` may be inconsistent with `predict`. Read more in the *User Guide*.

tol [float, optional (default=1e-3)] Tolerance for stopping criterion.

cache_size [float, optional] Specify the size of the kernel cache (in MB).

class_weight [{dict, 'balanced'}, optional] Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies as `n_samples / (n_classes * np.bincount(y))`

verbose [bool, default: False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter [int, optional (default=-1)] Hard limit on iterations within solver, or -1 for no limit.

decision_function_shape ['ovo', 'ovr', default='ovr'] Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2).

Changed in version 0.19: decision_function_shape is 'ovr' by default.

New in version 0.17: *decision_function_shape='ovr'* is recommended.

Changed in version 0.17: Deprecated *decision_function_shape='ovo' and None*.

break_ties [bool, optional (default=False)] If true, `decision_function_shape='ovr'`, and number of classes > 2, *predict* will break ties according to the confidence values of *decision_function*; otherwise the first class among the tied classes is returned. Please note that breaking ties comes at a relatively high computational cost compared to a simple predict.

New in version 0.22.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator used when shuffling the data for probability estimates. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Attributes

support_ [array-like of shape (n_SV)] Indices of support vectors.

support_vectors_ [array-like of shape (n_SV, n_features)] Support vectors.

n_support_ [array-like, dtype=int32, shape = [n_class]] Number of support vectors for each class.

dual_coef_ [array, shape = [n_class-1, n_SV]] Coefficients of the support vector in the decision function. For multiclass, coefficient for all 1-vs-1 classifiers. The layout of the coefficients in the multiclass case is somewhat non-trivial. See the section about multi-class classification in the SVM section of the User Guide for details.

coef_ [array, shape = [n_class * (n_class-1) / 2, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

`coef_` is readonly property derived from `dual_coef_` and `support_vectors_`.

intercept_ [ndarray of shape (n_class * (n_class-1) / 2,)] Constants in decision function.

classes_ [array of shape (n_classes,)] The unique classes labels.

fit_status_ [int] 0 if correctly fitted, 1 if the algorithm did not converge.

probA_ [ndarray, shape of (n_class * (n_class-1) / 2,)]

probB_ [ndarray of shape (n_class * (n_class-1) / 2,)] If `probability=True`, it corresponds to the parameters learned in Platt scaling to produce probability estimates from decision values. If `probability=False`, it's an empty array. Platt scaling uses the logistic function `1 / (1 + exp(decision_value * probA_ + probB_))` where `probA_` and `probB_` are learned from the dataset [R9709ce4a60d3-2]. For more information on the multiclass case and training procedure see section 8 of [R9709ce4a60d3-1].

class_weight_ [ndarray of shape (n_class,)] Multipliers of parameter C of each class. Computed based on the `class_weight` parameter.

shape_fit_ [tuple of int of shape (n_dimensions_of_X,)] Array dimensions of training vector `X`.

See also:

SVC Support Vector Machine for classification using libsvm.

LinearSVC Scalable linear Support Vector Machine for classification using liblinear.

References

[R9709ce4a60d3-1], [R9709ce4a60d3-2]

Examples

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> y = np.array([1, 1, 2, 2])
>>> from sklearn.svm import NuSVC
>>> clf = NuSVC()
>>> clf.fit(X, y)
NuSVC()
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

decision_function(self, X)	Evaluates the decision function for the samples in X.
fit(self, X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Perform classification on samples in X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *nu=0.5*, *kernel='rbf'*, *degree=3*, *gamma='scale'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=-1*, *decision_function_shape='ovr'*, *break_ties=False*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Evaluates the decision function for the samples in X.

Parameters

X [array-like, shape (n_samples, n_features)]

Returns

X [array-like, shape (n_samples, n_classes * (n_classes-1) / 2)] Returns the decision function of the sample for each class in the model. If decision_function_shape='ovr', the shape is (n_samples, n_classes).

Notes

If decision_function_shape='ovo', the function values are proportional to the distance of the samples X to the separating hyperplane. If the exact distances are required, divide the function values by the norm of the weight vector (`coef_`). See also this question for further details. If decision_function_shape='ovr', the decision function is a monotonic transformation of ovo decision function.

fit(*self*, *X*, *y*, *sample_weight=None*)

Fit the SVM model according to the given training data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features. For kernel="precomputed", the expected shape of X is (n_samples, n_samples).

y [array-like, shape (n_samples,)] Target values (class labels in classification, real numbers in regression)

sample_weight [array-like, shape (n_samples,)] Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

Returns

self [object]

Notes

If X and y are not C-ordered and contiguous arrays of np.float64 and X is not a scipy.sparse.csr_matrix, X and/or y may be copied.

If X is a dense array, then the other methods will not support sparse matrices as input.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)

Perform classification on samples in X.

For an one-class model, +1 or -1 is returned.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returns

y_pred [array, shape (n_samples,)] Class labels for samples in X.

property predict_log_proba

Compute log probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute `probability` set to True.

Parameters

X [array-like, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returns

T [array-like, shape (n_samples, n_classes)] Returns the log-probabilities of the sample for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will produce meaningless results on very small datasets.

property predict_proba

Compute probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute `probability` set to True.

Parameters

X [array-like, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returns

T [array-like, shape (n_samples, n_classes)] Returns the probability of the sample for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will produce meaningless results on very small datasets.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.svm.NuSVC`

- *Non-linear SVM*

`sklearn.svm`.NuSVR

class sklearn.svm.**NuSVR**(*nu=0.5*, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma='scale'*, *coef0=0.0*, *shrinking=True*, *tol=0.001*, *cache_size=200*, *verbose=False*, *max_iter=-1*)

Nu Support Vector Regression.

Similar to NuSVC, for regression, uses a parameter nu to control the number of support vectors. However, unlike NuSVC, where nu replaces C, here nu replaces the parameter epsilon of epsilon-SVR.

The implementation is based on libsvm.

Read more in the *User Guide*.

Parameters

nu [float, optional] An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1]. By default 0.5 will be taken.

C [float, optional (default=1.0)] Penalty parameter C of the error term.

kernel [string, optional (default='rbf')] Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to precompute the kernel matrix.

degree [int, optional (default=3)] Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma [{'scale', 'auto'} or float, optional (default='scale')] Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

- if `gamma='scale'` (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
- if 'auto', uses 1 / n_features.

Changed in version 0.22: The default value of `gamma` changed from 'auto' to 'scale'.

coef0 [float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

shrinking [boolean, optional (default=True)] Whether to use the shrinking heuristic.

tol [float, optional (default=1e-3)] Tolerance for stopping criterion.

cache_size [float, optional] Specify the size of the kernel cache (in MB).

verbose [bool, default: False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter [int, optional (default=-1)] Hard limit on iterations within solver, or -1 for no limit.

Attributes

support_ [array-like of shape (n_SV)] Indices of support vectors.

support_vectors_ [array-like of shape (n_SV, n_features)] Support vectors.

dual_coef_ [array, shape = [1, n_SV]] Coefficients of the support vector in the decision function.

coef_ [array, shape = [1, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

`coef_` is readonly property derived from `dual_coef_` and `support_vectors_`.

intercept_ [array, shape = [1]] Constants in decision function.

See also:

NuSVC Support Vector Machine for classification implemented with libsvm with a parameter to control the number of support vectors.

SVR epsilon Support Vector Machine for regression implemented with libsvm.

Notes

References: LIBSVM: A Library for Support Vector Machines

Examples

```
>>> from sklearn.svm import NuSVR
>>> import numpy as np
>>> n_samples, n_features = 10, 5
>>> np.random.seed(0)
>>> y = np.random.randn(n_samples)
>>> X = np.random.randn(n_samples, n_features)
>>> clf = NuSVR(C=1.0, nu=0.1)
>>> clf.fit(X, y)
NuSVR(nu=0.1)
```

Methods

fit(self, X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.

Continued on next page

Table 293 – continued from previous page

predict(self, X)	Perform regression on samples in X.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *nu=0.5*, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma='scale'*, *coef0=0.0*, *shrinking=True*, *tol=0.001*, *cache_size=200*, *verbose=False*, *max_iter=-1*)
Initialize self. See help(type(self)) for accurate signature.

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit the SVM model according to the given training data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features. For kernel="precomputed", the expected shape of X is (n_samples, n_samples).

y [array-like, shape (n_samples,)] Target values (class labels in classification, real numbers in regression)

sample_weight [array-like, shape (n_samples,)] Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

Returns

self [object]

Notes

If X and y are not C-ordered and contiguous arrays of np.float64 and X is not a scipy.sparse.csr_matrix, X and/or y may be copied.

If X is a dense array, then the other methods will not support sparse matrices as input.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Perform regression on samples in X.

For an one-class model, +1 (inlier) or -1 (outlier) is returned.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is (n_samples_test, n_samples_train).

Returns

y_pred [array, shape (n_samples,)]

score(*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.svm.NuSVR`

- *Model Complexity Influence*

`sklearn.svm`.OneClassSVM

class sklearn.svm.**OneClassSVM**(*kernel='rbf'*, *degree=3*, *gamma='scale'*, *coef0=0.0*, *tol=0.001*, *nu=0.5*, *shrinking=True*, *cache_size=200*, *verbose=False*, *max_iter=-1*)

Unsupervised Outlier Detection.

Estimate the support of a high-dimensional distribution.

The implementation is based on libsvm.

Read more in the *User Guide*.

Parameters

kernel [string, optional (default='rbf')] Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to precompute the kernel matrix.

degree [int, optional (default=3)] Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma [{'scale', 'auto'} or float, optional (default='scale')] Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

- if `gamma='scale'` (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
- if 'auto', uses 1 / n_features.

Changed in version 0.22: The default value of `gamma` changed from 'auto' to 'scale'.

coef0 [float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

tol [float, optional] Tolerance for stopping criterion.

nu [float, optional] An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1]. By default 0.5 will be taken.

shrinking [boolean, optional] Whether to use the shrinking heuristic.

cache_size [float, optional] Specify the size of the kernel cache (in MB).

verbose [bool, default: False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter [int, optional (default=-1)] Hard limit on iterations within solver, or -1 for no limit.

Attributes

support_ [array-like of shape (n_SV)] Indices of support vectors.

support_vectors_ [array-like of shape (n_SV, n_features)] Support vectors.

dual_coef_ [array, shape = [1, n_SV]] Coefficients of the support vectors in the decision function.

coef_ [array, shape = [1, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

`coef_` is readonly property derived from `dual_coef_` and `support_vectors_`

intercept_ [array, shape = [1,]] Constant in the decision function.

offset_ [float] Offset used to define the decision function from the raw scores. We have the relation: decision_function = score_samples - `offset_`. The offset is the opposite of `intercept_` and is provided for consistency with other outlier detection algorithms.

fit_status_ [int] 0 if correctly fitted, 1 otherwise (will raise warning)

Examples

```
>>> from sklearn.svm import OneClassSVM
>>> X = [[0], [0.44], [0.45], [0.46], [1]]
>>> clf = OneClassSVM(gamma='auto').fit(X)
>>> clf.predict(X)
array([-1,  1,  1,  1, -1])
>>> clf.score_samples(X)  # doctest: +ELLIPSIS
array([1.7798..., 2.0547..., 2.0556..., 2.0561..., 1.7332...])
```

Methods

decision_function(self, X)	Signed distance to the separating hyperplane.
fit(self, X[, y, sample_weight])	Detects the soft boundary of the set of samples X.
fit_predict(self, X[, y])	Perform fit on X and returns labels for X.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Perform classification on samples in X.
score_samples(self, X)	Raw scoring function of the samples.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *kernel='rbf'*, *degree=3*, *gamma='scale'*, *coef0=0.0*, *tol=0.001*, *nu=0.5*, *shrinking=True*, *cache_size=200*, *verbose=False*, *max_iter=-1*)
Initialize self. See help(type(self)) for accurate signature.

decision_function(*self*, *X*)
Signed distance to the separating hyperplane.

Signed distance is positive for an inlier and negative for an outlier.

Parameters

X [array-like, shape (n_samples, n_features)]

Returns

dec [array-like, shape (n_samples,)] Returns the decision function of the samples.

fit(*self*, *X*, *y=None*, *sample_weight=None*, ***params*)
Detects the soft boundary of the set of samples X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Set of samples, where n_samples is the number of samples and n_features is the number of features.

sample_weight [array-like, shape (n_samples,)] Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

y [Ignored] not used, present for API consistency by convention.

Returns

self [object]

Notes

If X is not a C-ordered contiguous array it is copied.

fit_predict(*self*, *X*, *y=None*)
Perform fit on X and returns labels for X.

Returns -1 for outliers and 1 for inliers.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

y [Ignored] Not used, present for API consistency by convention.

Returns

y [ndarray, shape (n_samples,)] 1 for inliers, -1 for outliers.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Perform classification on samples in X.

For a one-class model, +1 or -1 is returned.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returns

y_pred [array, shape (n_samples,)] Class labels for samples in X.

score_samples(*self*, *X*)
Raw scoring function of the samples.

Parameters

X [array-like, shape (n_samples, n_features)]

Returns

score_samples [array-like, shape (n_samples,)] Returns the (unshifted) scoring function of the samples.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.svm.OneClassSVM`

- *Libsvm GUI*

`sklearn.svm.SVC`

class sklearn.svm.**SVC**(*C=1.0*, *kernel='rbf'*, *degree=3*, *gamma='scale'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=-1*, *decision_function_shape='ovr'*, *break_ties=False*, *random_state=None*)

C-Support Vector Classification.

The implementation is based on libsvm. The fit time scales at least quadratically with the number of samples and may be impractical beyond tens of thousands of samples. For large datasets consider using *sklearn.svm.LinearSVC* or *sklearn.linear_model.SGDClassifier* instead, possibly after a *sklearn.kernel_approximation.Nystroem* transformer.

The multiclass support is handled according to a one-vs-one scheme.

For details on the precise mathematical formulation of the provided kernel functions and how `gamma`, `coef0` and `degree` affect each other, see the corresponding section in the narrative documentation: *Kernel functions*.

Read more in the *User Guide*.

Parameters

C [float, optional (default=1.0)] Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.

kernel [string, optional (default='rbf')] Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape `(n_samples, n_samples)`.

degree [int, optional (default=3)] Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma [{'scale', 'auto'} or float, optional (default='scale')] Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

- if `gamma='scale'` (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
- if 'auto', uses 1 / n_features.

Changed in version 0.22: The default value of `gamma` changed from 'auto' to 'scale'.

coef0 [float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

shrinking [boolean, optional (default=True)] Whether to use the shrinking heuristic.

probability [boolean, optional (default=False)] Whether to enable probability estimates. This must be enabled prior to calling `fit`, will slow down that method as it internally uses 5-fold cross-validation, and `predict_proba` may be inconsistent with `predict`. Read more in the *User Guide*.

tol [float, optional (default=1e-3)] Tolerance for stopping criterion.

cache_size [float, optional] Specify the size of the kernel cache (in MB).

class_weight [{dict, 'balanced'}, optional] Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

verbose [bool, default: False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter [int, optional (default=-1)] Hard limit on iterations within solver, or -1 for no limit.

decision_function_shape ['ovo', 'ovr', default='ovr'] Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2). However, one-vs-one ('ovo') is always used as multi-class strategy.

Changed in version 0.19: decision_function_shape is 'ovr' by default.

New in version 0.17: *decision_function_shape='ovr'* is recommended.

Changed in version 0.17: Deprecated *decision_function_shape='ovo' and None*.

break_ties [bool, optional (default=False)] If true, `decision_function_shape='ovr'`, and number of classes > 2, *predict* will break ties according to the confidence values of *decision_function*; otherwise the first class among the tied classes is returned. Please note that breaking ties comes at a relatively high computational cost compared to a simple predict.

New in version 0.22.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator used when shuffling the data for probability estimates. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Attributes

support_ [array-like of shape (n_SV)] Indices of support vectors.

support_vectors_ [array-like of shape (n_SV, n_features)] Support vectors.

n_support_ [array-like, dtype=int32, shape = [n_class]] Number of support vectors for each class.

dual_coef_ [array, shape = [n_class-1, n_SV]] Coefficients of the support vector in the decision function. For multiclass, coefficient for all 1-vs-1 classifiers. The layout of the coefficients in the multiclass case is somewhat non-trivial. See the section about multi-class classification in the SVM section of the User Guide for details.

coef_ [array, shape = [n_class * (n_class-1) / 2, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

`coef_` is a readonly property derived from `dual_coef_` and `support_vectors_`.

intercept_ [ndarray of shape (n_class * (n_class-1) / 2,)] Constants in decision function.

fit_status_ [int] 0 if correctly fitted, 1 otherwise (will raise warning)

classes_ [array of shape (n_classes,)] The classes labels.

probA_ [array, shape = [n_class * (n_class-1) / 2]]

probB_ [array, shape = [n_class * (n_class-1) / 2]] If `probability=True`, it corresponds to the parameters learned in Platt scaling to produce probability estimates from decision values.

If `probability=False`, it's an empty array. Platt scaling uses the logistic function `1 / (1 + exp(decision_value * probA_ + probB_))` where `probA_` and `probB_` are learned from the dataset [R20c70293ef72-2]. For more information on the multiclass case and training procedure see section 8 of [R20c70293ef72-1].

class_weight_ [ndarray of shape (n_class,)] Multipliers of parameter C for each class. Computed based on the `class_weight` parameter.

shape_fit_ [tuple of int of shape (n_dimensions_of_X,)] Array dimensions of training vector `X`.

See also:

SVR Support Vector Machine for Regression implemented using libsvm.

LinearSVC Scalable Linear Support Vector Machine for classification implemented using liblinear. Check the See also section of LinearSVC for more comparison element.

References

[R20c70293ef72-1], [R20c70293ef72-2]

Examples

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> y = np.array([1, 1, 2, 2])
>>> from sklearn.svm import SVC
>>> clf = SVC(gamma='auto')
>>> clf.fit(X, y)
SVC(gamma='auto')
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

decision_function(self, X)	Evaluates the decision function for the samples in X.
fit(self, X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Perform classification on samples in X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma='scale'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=-1*, *decision_function_shape='ovr'*, *break_ties=False*, *random_state=None*)

Initialize self. See help(type(self)) for accurate signature.

decision_function (*self*, *X*)

Evaluates the decision function for the samples in X.

Parameters

X [array-like, shape (n_samples, n_features)]

Returns

X [array-like, shape (n_samples, n_classes * (n_classes-1) / 2)] Returns the decision function of the sample for each class in the model. If decision_function_shape='ovr', the shape is (n_samples, n_classes).

Notes

If decision_function_shape='ovo', the function values are proportional to the distance of the samples X to the separating hyperplane. If the exact distances are required, divide the function values by the norm of the weight vector (`coef_`). See also this question for further details. If decision_function_shape='ovr', the decision function is a monotonic transformation of ovo decision function.

fit(*self*, *X*, *y*, *sample_weight=None*)

Fit the SVM model according to the given training data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features. For kernel="precomputed", the expected shape of X is (n_samples, n_samples).

y [array-like, shape (n_samples,)] Target values (class labels in classification, real numbers in regression)

sample_weight [array-like, shape (n_samples,)] Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

Returns

self [object]

Notes

If X and y are not C-ordered and contiguous arrays of np.float64 and X is not a scipy.sparse.csr_matrix, X and/or y may be copied.

If X is a dense array, then the other methods will not support sparse matrices as input.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)

Perform classification on samples in X.

For an one-class model, +1 or -1 is returned.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returns

y_pred [array, shape (n_samples,)] Class labels for samples in X.

property predict_log_proba

Compute log probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute `probability` set to True.

Parameters

X [array-like, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returns

T [array-like, shape (n_samples, n_classes)] Returns the log-probabilities of the sample for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will produce meaningless results on very small datasets.

property predict_proba

Compute probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute `probability` set to True.

Parameters

X [array-like, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returns

T [array-like, shape (n_samples, n_classes)] Returns the probability of the sample for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will produce meaningless results on very small datasets.

score (*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.svm.SVC`

- *Libsvm GUI*

`sklearn.svm.SVR`

class sklearn.svm.**SVR**(*kernel='rbf'*, *degree=3*, *gamma='scale'*, *coef0=0.0*, *tol=0.001*, *C=1.0*, *epsilon=0.1*, *shrinking=True*, *cache_size=200*, *verbose=False*, *max_iter=-1*)

Epsilon-Support Vector Regression.

The free parameters in the model are C and epsilon.

The implementation is based on libsvm. The fit time complexity is more than quadratic with the number of samples which makes it hard to scale to datasets with more than a couple of 10000 samples. For large datasets consider using *sklearn.svm.LinearSVR* or *sklearn.linear_model.SGDRegressor* instead, possibly after a *sklearn.kernel_approximation.Nystroem* transformer.

Read more in the *User Guide*.

Parameters

kernel [string, optional (default='rbf')] Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to precompute the kernel matrix.

degree [int, optional (default=3)] Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma [{'scale', 'auto'} or float, optional (default='scale')] Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

- if `gamma='scale'` (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
- if 'auto', uses 1 / n_features.

Changed in version 0.22: The default value of `gamma` changed from 'auto' to 'scale'.

coef0 [float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

tol [float, optional (default=1e-3)] Tolerance for stopping criterion.

C [float, optional (default=1.0)] Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.

epsilon [float, optional (default=0.1)] Epsilon in the epsilon-SVR model. It specifies the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value.

shrinking [boolean, optional (default=True)] Whether to use the shrinking heuristic.

cache_size [float, optional] Specify the size of the kernel cache (in MB).

verbose [bool, default: False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter [int, optional (default=-1)] Hard limit on iterations within solver, or -1 for no limit.

Attributes

support_ [array-like of shape (n_SV)] Indices of support vectors.

support_vectors_ [array-like of shape (n_SV, n_features)] Support vectors.

dual_coef_ [array, shape = [1, n_SV]] Coefficients of the support vector in the decision function.

coef_ [array, shape = [1, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

`coef_` is readonly property derived from `dual_coef_` and `support_vectors_`.

fit_status_ [int] 0 if correctly fitted, 1 otherwise (will raise warning)

intercept_ [array, shape = [1]] Constants in decision function.

See also:

NuSVR Support Vector Machine for regression implemented using libsvm using a parameter to control the number of support vectors.

LinearSVR Scalable Linear Support Vector Machine for regression implemented using liblinear.

Notes

References: LIBSVM: A Library for Support Vector Machines

Examples

```
>>> from sklearn.svm import SVR
>>> import numpy as np
>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
>>> y = rng.randn(n_samples)
>>> X = rng.randn(n_samples, n_features)
>>> clf = SVR(C=1.0, epsilon=0.2)
>>> clf.fit(X, y)
SVR(epsilon=0.2)
```

Methods

fit(self, X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Perform regression on samples in X.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *kernel='rbf'*, *degree=3*, *gamma='scale'*, *coef0=0.0*, *tol=0.001*, *C=1.0*, *epsilon=0.1*, *shrinking=True*, *cache_size=200*, *verbose=False*, *max_iter=-1*)
Initialize self. See help(type(self)) for accurate signature.

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit the SVM model according to the given training data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features. For kernel="precomputed", the expected shape of X is (n_samples, n_samples).

y [array-like, shape (n_samples,)] Target values (class labels in classification, real numbers in regression)

sample_weight [array-like, shape (n_samples,)] Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

Returns

self [object]

Notes

If X and y are not C-ordered and contiguous arrays of np.float64 and X is not a scipy.sparse.csr_matrix, X and/or y may be copied.

If X is a dense array, then the other methods will not support sparse matrices as input.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)
Perform regression on samples in X.

For an one-class model, +1 (inlier) or -1 (outlier) is returned.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is (n_samples_test, n_samples_train).

Returns

y_pred [array, shape (n_samples,)]

score(*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.svm.SVR`

- *Comparison of kernel ridge regression and SVR*
- *Prediction Latency*
- *Support Vector Regression (SVR) using linear and non-linear kernels*

svm.l1_min_c(X, y[, loss, fit_intercept, ...])	Return the lowest bound for C such that for C in (l1_min_C, infinity) the model is guaranteed not to be empty.

sklearn.svm.l1_min_c

sklearn.svm.**l1_min_c**(*X*, *y*, *loss='squared_hinge'*, *fit_intercept=True*, *intercept_scaling=1.0*)

Return the lowest bound for C such that for C in (l1_min_C, infinity) the model is guaranteed not to be empty. This applies to l1 penalized classifiers, such as LinearSVC with penalty='l1' and linear_model.LogisticRegression with penalty='l1'.

This value is valid if class_weight parameter in fit() is not set.

Parameters

X [{array-like or sparse matrix} of shape (n_samples, n_features)] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [array, shape = [n_samples]] Target vector relative to X

loss [{'squared_hinge', 'log'}, default 'squared_hinge'] Specifies the loss function. With 'squared_hinge' it is the squared hinge loss (a.k.a. L2 loss). With 'log' it is the loss of logistic regression models.

fit_intercept [bool, default: True] Specifies if the intercept should be fitted by the model. It must match the fit() method parameter.

intercept_scaling [float, default: 1] when fit_intercept is True, instance vector x becomes [x, intercept_scaling], i.e. a "synthetic" feature with constant value equals to intercept_scaling is appended to the instance vector. It must match the fit() method parameter.

Returns

l1_min_c [float] minimum value for C

Examples using sklearn.svm.l1_min_c

- *Regularization path of L1- Logistic Regression*

7.37 sklearn.tree: Decision Trees

The *sklearn.tree* module includes decision tree-based models for classification and regression.

User guide: See the *Decision Trees* section for further details.

tree.DecisionTreeClassifier([criterion, ...])	A decision tree classifier.
tree.DecisionTreeRegressor([criterion, ...])	A decision tree regressor.
tree.ExtraTreeClassifier([criterion, ...])	An extremely randomized tree classifier.
tree.ExtraTreeRegressor([criterion, ...])	An extremely randomized tree regressor.

7.37.1 sklearn.tree.DecisionTreeClassifier

class sklearn.tree.**DecisionTreeClassifier**(*criterion='gini'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *class_weight=None*, *presort='deprecated'*, *ccp_alpha=0.0*)

A decision tree classifier.

Read more in the *User Guide*.

Parameters

criterion [{"gini", "entropy"}, default="gini"] The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

splitter [{"best", "random"}, default="best"] The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_depth [int, default=None] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int or float, default=2] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int or float, default=1] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, default=0.0] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features [int, float or {"auto", "sqrt", "log2"}, default=None] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=sqrt(n_features)`.

- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

random_state [int or RandomState, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

max_leaf_nodes [int, default=None] Grow a tree with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, default=0.0] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, default=1e-7] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

class_weight [dict, list of dict or "balanced", default=None] Weights associated with classes in the form `{class_label: weight}`. If None, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

presort [deprecated, default='deprecated'] This parameter is deprecated and will be removed in v0.24.

Deprecated since version 0.22.

ccp_alpha [non-negative float, default=0.0] Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than `ccp_alpha` will be chosen. By default, no pruning is performed. See *Minimal Cost-Complexity Pruning* for details.

New in version 0.22.

Attributes

classes_ [ndarray of shape (n_classes,) or list of ndarray] The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

feature_importances_ [ndarray of shape (n_features,)] Return the feature importances.

max_features_ [int] The inferred value of max_features.

n_classes_ [int or list of int] The number of classes (for single output problems), or a list containing the number of classes for each output (for multi-output problems).

n_features_ [int] The number of features when `fit` is performed.

n_outputs_ [int] The number of outputs when `fit` is performed.

tree_ [Tree] The underlying Tree object. Please refer to `help(sklearn.tree._tree.Tree)` for attributes of Tree object and *Understanding the decision tree structure* for basic usage of these attributes.

See also:

DecisionTreeRegressor A decision tree regressor.

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data and `max_features=n_features`, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed.

References

[Rb1ec977cd307-1], [Rb1ec977cd307-2], [Rb1ec977cd307-3], [Rb1ec977cd307-4]

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.tree import DecisionTreeClassifier
>>> clf = DecisionTreeClassifier(random_state=0)
>>> iris = load_iris()
>>> cross_val_score(clf, iris.data, iris.target, cv=10)
```

(continues on next page)

(continued from previous page)

```
...                                # doctest: +SKIP
...
array([ 1.     ,  0.93...,  0.86...,  0.93...,  0.93...,
        0.93...,  0.93...,  1.     ,  0.93...,  1.      ])
```

Methods

apply(self, X[, check_input])	Return the index of the leaf that each sample is predicted as.
cost_complexity_pruning_path(self, X, y[, ...])	Compute the pruning path during Minimal Cost-Complexity Pruning.
decision_path(self, X[, check_input])	Return the decision path in the tree.
fit(self, X, y[, sample_weight, ...])	Build a decision tree classifier from the training set (X, y).
get_depth(self)	Return the depth of the decision tree.
get_n_leaves(self)	Return the number of leaves of the decision tree.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, check_input])	Predict class or regression value for X.
predict_log_proba(self, X)	Predict class log-probabilities of the input samples X.
predict_proba(self, X[, check_input])	Predict class probabilities of the input samples X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *criterion='gini'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *class_weight=None*, *presort='deprecated'*, *ccp_alpha=0.0*)

Initialize self. See help(type(self)) for accurate signature.

apply(*self*, *X*, *check_input=True*)

Return the index of the leaf that each sample is predicted as.

New in version 0.17.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

X_leaves [array-like of shape (n_samples,)] For each datapoint x in X, return the index of the leaf x ends up in. Leaves are numbered within `[0; self.tree_.node_count)`, possibly with gaps in the numbering.

cost_complexity_pruning_path(*self*, *X*, *y*, *sample_weight=None*)

Compute the pruning path during Minimal Cost-Complexity Pruning.

See *Minimal Cost-Complexity Pruning* for details on the pruning process.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels) as integers or strings.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. Splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

ccp_path [Bunch] Dictionary-like object, with attributes:

ccp_alphas [ndarray] Effective alphas of subtree during pruning.

impurities [ndarray] Sum of the impurities of the subtree leaves for the corresponding alpha value in `ccp_alphas`.

decision_path(*self*, *X*, *check_input=True*)

Return the decision path in the tree.

New in version 0.18.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

indicator [sparse matrix of shape (n_samples, n_nodes)] Return a node indicator CSR matrix where non zero elements indicates that the samples goes through the nodes.

property feature_importances_

Return the feature importances.

The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance.

Returns

feature_importances_ [ndarray of shape (n_features,)] Normalized total reduction of criteria by feature (Gini importance).

fit(*self*, *X*, *y*, *sample_weight=None*, *check_input=True*, *X_idx_sorted=None*)

Build a decision tree classifier from the training set (X, y).

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels) as integers or strings.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. Splits are also ignored if they would result in any single class carrying a negative weight in either child node.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

X_idx_sorted [array-like of shape (n_samples, n_features), default=None] The indexes of the sorted training input samples. If many tree are grown on the same dataset, this allows the ordering to be cached between trees. If None, the data will be sorted here. Don't use this parameter unless you know what to do.

Returns

self [DecisionTreeClassifier] Fitted estimator.

get_depth(*self*)
Return the depth of the decision tree.

The depth of a tree is the maximum distance between the root and any leaf.

Returns

self.tree_.max_depth [int] The maximum depth of the tree.

get_n_leaves(*self*)
Return the number of leaves of the decision tree.

Returns

self.tree_.n_leaves [int] Number of leaves.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*, *check_input=True*)
Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The predicted classes, or the predict values.

predict_log_proba(*self*, *X*)
Predict class log-probabilities of the input samples X.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

proba [ndarray of shape (n_samples, n_classes) or list of n_outputs such arrays if n_outputs > 1] The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

predict_proba(*self*, *X*, *check_input=True*)
Predict class probabilities of the input samples X.

The predicted class probability is the fraction of samples of the same class in a leaf.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

proba [ndarray of shape (n_samples, n_classes) or list of n_outputs such arrays if n_outputs > 1] The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.tree.DecisionTreeClassifier`

- *Classifier comparison*
- *Plot the decision surface of a decision tree on the iris dataset*
- *Post pruning decision trees with cost complexity pruning*
- *Understanding the decision tree structure*
- *Plot the decision boundaries of a VotingClassifier*
- *Two-class AdaBoost*
- *Multi-class AdaBoosted Decision Trees*
- *Discrete versus Real AdaBoost*
- *Plot the decision surfaces of ensembles of trees on the iris dataset*
- *Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV*

7.37.2 `sklearn.tree`.DecisionTreeRegressor

class sklearn.tree.**DecisionTreeRegressor**(*criterion='mse'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *presort='deprecated'*, *ccp_alpha=0.0*)

A decision tree regressor.

Read more in the *User Guide*.

Parameters

criterion [{"mse", "friedman_mse", "mae"}, default="mse"] The function to measure the quality of a split. Supported criteria are "mse" for the mean squared error, which is equal to variance reduction as feature selection criterion and minimizes the L2 loss using the mean of each terminal node, "friedman_mse", which uses mean squared error with Friedman's improvement score for potential splits, and "mae" for the mean absolute error, which minimizes the L1 loss using the median of each terminal node.

New in version 0.18: Mean Absolute Error (MAE) criterion.

splitter [{"best", "random"}, default="best"] The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_depth [int, default=None] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int or float, default=2] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int or float, default=1] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, default=0.0] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features [int, float or {"auto", "sqrt", "log2"}, default=None] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=n_features`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

random_state [int or RandomState, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

max_leaf_nodes [int, default=None] Grow a tree with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, default=0.0] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

presort [deprecated, default='deprecated'] This parameter is deprecated and will be removed in v0.24.

Deprecated since version 0.22.

ccp_alpha [non-negative float, default=0.0] Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than `ccp_alpha` will be chosen. By default, no pruning is performed. See *Minimal Cost-Complexity Pruning* for details.

New in version 0.22.

Attributes

feature_importances_ [ndarray of shape (n_features,)] Return the feature importances.

max_features_ [int] The inferred value of max_features.

n_features_ [int] The number of features when `fit` is performed.

n_outputs_ [int] The number of outputs when `fit` is performed.

tree_ [Tree] The underlying Tree object. Please refer to `help(sklearn.tree._tree.Tree)` for attributes of Tree object and *Understanding the decision tree structure* for basic usage of these attributes.

See also:

DecisionTreeClassifier A decision tree classifier.

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data and `max_features=n_features`, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed.

References

[Ra37b7e3adb19-1], [Ra37b7e3adb19-2], [Ra37b7e3adb19-3], [Ra37b7e3adb19-4]

Examples

```
>>> from sklearn.datasets import load_boston
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.tree import DecisionTreeRegressor
>>> X, y = load_boston(return_X_y=True)
>>> regressor = DecisionTreeRegressor(random_state=0)
>>> cross_val_score(regressor, X, y, cv=10)
...                    # doctest: +SKIP
...
array([ 0.61..., 0.57..., -0.34..., 0.41..., 0.75...,
        0.07..., 0.29..., 0.33..., -1.42..., -1.77...])
```

Methods

apply(self, X[, check_input])	Return the index of the leaf that each sample is predicted as.
cost_complexity_pruning_path(self, X, y[, ...])	Compute the pruning path during Minimal Cost-Complexity Pruning.
decision_path(self, X[, check_input])	Return the decision path in the tree.
fit(self, X, y[, sample_weight, ...])	Build a decision tree regressor from the training set (X, y).
get_depth(self)	Return the depth of the decision tree.
get_n_leaves(self)	Return the number of leaves of the decision tree.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, check_input])	Predict class or regression value for X.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *criterion='mse'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *presort='deprecated'*, *ccp_alpha=0.0*)

Initialize self. See help(type(self)) for accurate signature.

apply(*self*, *X*, *check_input=True*)

Return the index of the leaf that each sample is predicted as.

New in version 0.17.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

X_leaves [array-like of shape (n_samples,)] For each datapoint x in X, return the index of the leaf x ends up in. Leaves are numbered within `[0; self.tree_.node_count)`, possibly with gaps in the numbering.

cost_complexity_pruning_path(*self*, *X*, *y*, *sample_weight=None*)
Compute the pruning path during Minimal Cost-Complexity Pruning.

See *Minimal Cost-Complexity Pruning* for details on the pruning process.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels) as integers or strings.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. Splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

ccp_path [Bunch] Dictionary-like object, with attributes:

ccp_alphas [ndarray] Effective alphas of subtree during pruning.

impurities [ndarray] Sum of the impurities of the subtree leaves for the corresponding alpha value in `ccp_alphas`.

decision_path(*self*, *X*, *check_input=True*)
Return the decision path in the tree.

New in version 0.18.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

indicator [sparse matrix of shape (n_samples, n_nodes)] Return a node indicator CSR matrix where non zero elements indicates that the samples goes through the nodes.

property feature_importances_
Return the feature importances.

The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance.

Returns

feature_importances_ [ndarray of shape (n_features,)] Normalized total reduction of criteria by feature (Gini importance).

fit(*self*, *X*, *y*, *sample_weight=None*, *check_input=True*, *X_idx_sorted=None*)
Build a decision tree regressor from the training set (X, y).

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The target values (real numbers). Use `dtype=np.float64` and `order='C'` for maximum efficiency.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

X_idx_sorted [array-like of shape (n_samples, n_features), default=None] The indexes of the sorted training input samples. If many tree are grown on the same dataset, this allows the ordering to be cached between trees. If None, the data will be sorted here. Don't use this parameter unless you know what to do.

Returns

self [DecisionTreeRegressor] Fitted estimator.

get_depth(*self*)
Return the depth of the decision tree.

The depth of a tree is the maximum distance between the root and any leaf.

Returns

self.tree_.max_depth [int] The maximum depth of the tree.

get_n_leaves(*self*)
Return the number of leaves of the decision tree.

Returns

self.tree_.n_leaves [int] Number of leaves.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*, *check_input=True*)
Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The predicted classes, or the predict values.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `sklearn.tree.DecisionTreeRegressor`

- *Advanced Plotting With Partial Dependence*
- *Decision Tree Regression*
- *Multi-output Decision Tree Regression*
- *Decision Tree Regression with AdaBoost*
- *Single estimator versus bagging: bias-variance decomposition*

- *Imputing missing values with variants of IterativeImputer*
- *Using KBinsDiscretizer to discretize continuous features*
- *Release Highlights for scikit-learn 0.22*

7.37.3 sklearn.tree.ExtraTreeClassifier

class sklearn.tree.**ExtraTreeClassifier**(*criterion='gini'*, *splitter='random'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *class_weight=None*, *ccp_alpha=0.0*)

An extremely randomized tree classifier.

Extra-trees differ from classic decision trees in the way they are built. When looking for the best split to separate the samples of a node into two groups, random splits are drawn for each of the `max_features` randomly selected features and the best split among those is chosen. When `max_features` is set 1, this amounts to building a totally random decision tree.

Warning: Extra-trees should only be used within ensemble methods.

Read more in the *User Guide*.

Parameters

criterion [{"gini", "entropy"}, default="gini"] The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

splitter [{"random", "best"}, default="random"] The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_depth [int, default=None] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int or float, default=2] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int or float, default=1] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, default=0.0] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features [int, float, {"auto", "sqrt", "log2"} or None, default="auto"] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=sqrt(n_features)`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

random_state [int or RandomState, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

max_leaf_nodes [int, default=None] Grow a tree with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, default=0.0] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

class_weight [dict, list of dict or "balanced", default=None] Weights associated with classes in the form `{class_label: weight}`. If None, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights

should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

ccp_alpha [non-negative float, default=0.0] Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than `ccp_alpha` will be chosen. By default, no pruning is performed. See *Minimal Cost-Complexity Pruning* for details.

New in version 0.22.

Attributes

classes_ [ndarray of shape (n_classes,) or list of ndarray] The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

max_features_ [int] The inferred value of max_features.

n_classes_ [int or list of int] The number of classes (for single output problems), or a list containing the number of classes for each output (for multi-output problems).

feature_importances_ [ndarray of shape (n_features,)] Return the feature importances.

n_features_ [int] The number of features when `fit` is performed.

n_outputs_ [int] The number of outputs when `fit` is performed.

tree_ [Tree] The underlying Tree object. Please refer to `help(sklearn.tree._tree.Tree)` for attributes of Tree object and *Understanding the decision tree structure* for basic usage of these attributes.

See also:

ExtraTreeRegressor An extremely randomized tree regressor.

sklearn.ensemble.ExtraTreesClassifier An extra-trees classifier.

sklearn.ensemble.ExtraTreesRegressor An extra-trees regressor.

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

References

[Rdd99a0224c6e-1]

Methods

apply(self, X[, check_input])	Return the index of the leaf that each sample is predicted as.
cost_complexity_pruning_path(self, X, y[, ...])	Compute the pruning path during Minimal Cost-Complexity Pruning.
decision_path(self, X[, check_input])	Return the decision path in the tree.
fit(self, X, y[, sample_weight, ...])	Build a decision tree classifier from the training set (X, y).
get_depth(self)	Return the depth of the decision tree.
get_n_leaves(self)	Return the number of leaves of the decision tree.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, check_input])	Predict class or regression value for X.
predict_log_proba(self, X)	Predict class log-probabilities of the input samples X.
predict_proba(self, X[, check_input])	Predict class probabilities of the input samples X.
score(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *criterion='gini'*, *splitter='random'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *class_weight=None*, *ccp_alpha=0.0*)
Initialize self. See help(type(self)) for accurate signature.

apply(*self*, *X*, *check_input=True*)
Return the index of the leaf that each sample is predicted as.

New in version 0.17.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

X_leaves [array-like of shape (n_samples,)] For each datapoint x in X, return the index of the leaf x ends up in. Leaves are numbered within `[0; self.tree_.node_count)`, possibly with gaps in the numbering.

cost_complexity_pruning_path(*self*, *X*, *y*, *sample_weight=None*)
Compute the pruning path during Minimal Cost-Complexity Pruning.

See *Minimal Cost-Complexity Pruning* for details on the pruning process.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels) as integers or strings.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. Splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

ccp_path [Bunch] Dictionary-like object, with attributes:

ccp_alphas [ndarray] Effective alphas of subtree during pruning.

impurities [ndarray] Sum of the impurities of the subtree leaves for the corresponding alpha value in `ccp_alphas`.

decision_path(*self*, *X*, *check_input=True*)
Return the decision path in the tree.

New in version 0.18.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

indicator [sparse matrix of shape (n_samples, n_nodes)] Return a node indicator CSR matrix where non zero elements indicates that the samples goes through the nodes.

property feature_importances_
Return the feature importances.

The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance.

Returns

feature_importances_ [ndarray of shape (n_features,)] Normalized total reduction of criteria by feature (Gini importance).

fit(*self*, *X*, *y*, *sample_weight=None*, *check_input=True*, *X_idx_sorted=None*)
Build a decision tree classifier from the training set (X, y).

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels) as integers or strings.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. Splits are also ignored if they would result in any single class carrying a negative weight in either child node.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

X_idx_sorted [array-like of shape (n_samples, n_features), default=None] The indexes of the sorted training input samples. If many tree are grown on the same dataset, this allows the ordering to be cached between trees. If None, the data will be sorted here. Don't use this parameter unless you know what to do.

Returns

self [DecisionTreeClassifier] Fitted estimator.

get_depth(*self*)
Return the depth of the decision tree.

The depth of a tree is the maximum distance between the root and any leaf.

Returns

self.tree_.max_depth [int] The maximum depth of the tree.

get_n_leaves(*self*)
Return the number of leaves of the decision tree.

Returns

self.tree_.n_leaves [int] Number of leaves.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*, *check_input=True*)
Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The predicted classes, or the predict values.

predict_log_proba(*self*, *X*)
Predict class log-probabilities of the input samples X.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

proba [ndarray of shape (n_samples, n_classes) or list of n_outputs such arrays if n_outputs > 1] The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

predict_proba(*self*, *X*, *check_input=True*)

Predict class probabilities of the input samples X.

The predicted class probability is the fraction of samples of the same class in a leaf.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

proba [ndarray of shape (n_samples, n_classes) or list of n_outputs such arrays if n_outputs > 1] The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

score(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

7.37.4 sklearn.tree.ExtraTreeRegressor

class sklearn.tree.**ExtraTreeRegressor**(*criterion='mse'*, *splitter='random'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *random_state=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *max_leaf_nodes=None*, *ccp_alpha=0.0*)

An extremely randomized tree regressor.

Extra-trees differ from classic decision trees in the way they are built. When looking for the best split to separate the samples of a node into two groups, random splits are drawn for each of the `max_features` randomly selected features and the best split among those is chosen. When `max_features` is set 1, this amounts to building a totally random decision tree.

Warning: Extra-trees should only be used within ensemble methods.

Read more in the *User Guide*.

Parameters

criterion [{"mse", "friedman_mse", "mae"}, default="mse"] The function to measure the quality of a split. Supported criteria are "mse" for the mean squared error, which is equal to variance reduction as feature selection criterion, and "mae" for the mean absolute error.

New in version 0.18: Mean Absolute Error (MAE) criterion.

splitter [{"random", "best"}, default="random"] The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_depth [int, default=None] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int or float, default=2] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int or float, default=1] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, default=0.0] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features [int, float, {"auto", "sqrt", "log2"} or None, default="auto"] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=n_features`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

random_state [int or RandomState, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

min_impurity_decrease [float, default=0.0] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

max_leaf_nodes [int, default=None] Grow a tree with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

ccp_alpha [non-negative float, default=0.0] Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than `ccp_alpha` will be chosen. By default, no pruning is performed. See *Minimal Cost-Complexity Pruning* for details.

New in version 0.22.

Attributes

max_features_ [int] The inferred value of max_features.

n_features_ [int] The number of features when `fit` is performed.

n_outputs_ [int] The number of outputs when `fit` is performed.

tree_ [Tree] The underlying Tree object. Please refer to `help(sklearn.tree._tree.Tree)` for attributes of Tree object and *Understanding the decision tree structure* for basic usage of these attributes.

See also:

ExtraTreeClassifier An extremely randomized tree classifier.

sklearn.ensemble.ExtraTreesClassifier An extra-trees classifier.

sklearn.ensemble.ExtraTreesRegressor An extra-trees regressor.

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

References

[R4939d63d5a49-1]

Examples

```
>>> from sklearn.datasets import load_boston
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.ensemble import BaggingRegressor
>>> from sklearn.tree import ExtraTreeRegressor
>>> X, y = load_boston(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, random_state=0)
>>> extra_tree = ExtraTreeRegressor(random_state=0)
>>> reg = BaggingRegressor(extra_tree, random_state=0).fit(
...     X_train, y_train)
>>> reg.score(X_test, y_test)
0.7823...
```

Methods

apply(self, X[, check_input])	Return the index of the leaf that each sample is predicted as.
cost_complexity_pruning_path(self, X, y[, ...])	Compute the pruning path during Minimal Cost-Complexity Pruning.
decision_path(self, X[, check_input])	Return the decision path in the tree.
fit(self, X, y[, sample_weight, ...])	Build a decision tree regressor from the training set (X, y).
get_depth(self)	Return the depth of the decision tree.
get_n_leaves(self)	Return the number of leaves of the decision tree.

Continued on next page

Table 302 – continued from previous page

get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, check_input])	Predict class or regression value for X.
score(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *criterion='mse'*, *splitter='random'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *random_state=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *max_leaf_nodes=None*, *ccp_alpha=0.0*)
Initialize self. See help(type(self)) for accurate signature.

apply(*self*, *X*, *check_input=True*)
Return the index of the leaf that each sample is predicted as.

New in version 0.17.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

X_leaves [array-like of shape (n_samples,)] For each datapoint x in X, return the index of the leaf x ends up in. Leaves are numbered within `[0; self.tree_.node_count)`, possibly with gaps in the numbering.

cost_complexity_pruning_path(*self*, *X*, *y*, *sample_weight=None*)
Compute the pruning path during Minimal Cost-Complexity Pruning.

See *Minimal Cost-Complexity Pruning* for details on the pruning process.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels) as integers or strings.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. Splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

ccp_path [Bunch] Dictionary-like object, with attributes:

ccp_alphas [ndarray] Effective alphas of subtree during pruning.

impurities [ndarray] Sum of the impurities of the subtree leaves for the corresponding alpha value in `ccp_alphas`.

decision_path(*self*, *X*, *check_input=True*)
Return the decision path in the tree.

New in version 0.18.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

indicator [sparse matrix of shape (n_samples, n_nodes)] Return a node indicator CSR matrix where non zero elements indicates that the samples goes through the nodes.

property feature_importances_

Return the feature importances.

The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance.

Returns

feature_importances_ [ndarray of shape (n_features,)] Normalized total reduction of criteria by feature (Gini importance).

fit (*self*, *X*, *y*, *sample_weight=None*, *check_input=True*, *X_idx_sorted=None*)

Build a decision tree regressor from the training set (X, y).

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The target values (real numbers). Use `dtype=np.float64` and `order='C'` for maximum efficiency.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

X_idx_sorted [array-like of shape (n_samples, n_features), default=None] The indexes of the sorted training input samples. If many tree are grown on the same dataset, this allows the ordering to be cached between trees. If None, the data will be sorted here. Don't use this parameter unless you know what to do.

Returns

self [DecisionTreeRegressor] Fitted estimator.

get_depth (*self*)

Return the depth of the decision tree.

The depth of a tree is the maximum distance between the root and any leaf.

Returns

self.tree_.max_depth [int] The maximum depth of the tree.

get_n_leaves(*self*)
Return the number of leaves of the decision tree.

Returns

self.tree_.n_leaves [int] Number of leaves.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*, *check_input=True*)
Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool, default=True] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The predicted classes, or the predict values.

score(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *r2_score*. This will influence the `score` method of all the multioutput regressors (except for *MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *r2_score* directly or make a custom scorer with *make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

tree.export_graphviz(decision_tree[, ...])	Export a decision tree in DOT format.
tree.export_text(decision_tree[, ...])	Build a text report showing the rules of a decision tree.

7.37.5 sklearn.tree.export_graphviz

sklearn.tree.**export_graphviz**(*decision_tree*, *out_file=None*, *max_depth=None*, *feature_names=None*, *class_names=None*, *label='all'*, *filled=False*, *leaves_parallel=False*, *impurity=True*, *node_ids=False*, *proportion=False*, *rotate=False*, *rounded=False*, *special_characters=False*, *precision=3*)

Export a decision tree in DOT format.

This function generates a GraphViz representation of the decision tree, which is then written into `out_file`. Once exported, graphical renderings can be generated using, for example:

```
$ dot -Tps tree.dot -o tree.ps      (PostScript format)
$ dot -Tpng tree.dot -o tree.png    (PNG format)
```

The sample counts that are shown are weighted with any sample_weights that might be present.

Read more in the *User Guide*.

Parameters

decision_tree [decision tree classifier] The decision tree to be exported to GraphViz.

out_file [file object or string, optional (default=None)] Handle or name of the output file. If `None`, the result is returned as a string.

Changed in version 0.20: Default of out_file changed from "tree.dot" to None.

max_depth [int, optional (default=None)] The maximum depth of the representation. If None, the tree is fully generated.

feature_names [list of strings, optional (default=None)] Names of each of the features.

class_names [list of strings, bool or None, optional (default=None)] Names of each of the target classes in ascending numerical order. Only relevant for classification and not supported for multi-output. If `True`, shows a symbolic representation of the class name.

label [{'all', 'root', 'none'}, optional (default='all')] Whether to show informative labels for impurity, etc. Options include 'all' to show at every node, 'root' to show only at the top root node, or 'none' to not show at any node.

filled [bool, optional (default=False)] When set to `True`, paint nodes to indicate majority class for classification, extremity of values for regression, or purity of node for multi-output.

leaves_parallel [bool, optional (default=False)] When set to `True`, draw all leaf nodes at the bottom of the tree.

impurity [bool, optional (default=True)] When set to `True`, show the impurity at each node.

node_ids [bool, optional (default=False)] When set to `True`, show the ID number on each node.

proportion [bool, optional (default=False)] When set to `True`, change the display of 'values' and/or 'samples' to be proportions and percentages respectively.

rotate [bool, optional (default=False)] When set to `True`, orient tree left to right rather than top-down.

rounded [bool, optional (default=False)] When set to `True`, draw node boxes with rounded corners and use Helvetica fonts instead of Times-Roman.

special_characters [bool, optional (default=False)] When set to `False`, ignore special characters for PostScript compatibility.

precision [int, optional (default=3)] Number of digits of precision for floating point in the values of impurity, threshold and value attributes of each node.

Returns

dot_data [string] String representation of the input tree in GraphViz dot format. Only returned if `out_file` is None.

New in version 0.18.

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn import tree
```

```
>>> clf = tree.DecisionTreeClassifier()
>>> iris = load_iris()
```

```
>>> clf = clf.fit(iris.data, iris.target)
>>> tree.export_graphviz(clf)
'digraph Tree {...
```

7.37.6 sklearn.tree.export_text

sklearn.tree.**export_text**(*decision_tree*, *feature_names=None*, *max_depth=10*, *spacing=3*, *decimals=2*, *show_weights=False*)

Build a text report showing the rules of a decision tree.

Note that backwards compatibility may not be supported.

Parameters

decision_tree [object] The decision tree estimator to be exported. It can be an instance of DecisionTreeClassifier or DecisionTreeRegressor.

feature_names [list, optional (default=None)] A list of length n_features containing the feature names. If None generic names will be used ("feature_0", "feature_1", ...).

max_depth [int, optional (default=10)] Only the first max_depth levels of the tree are exported. Truncated branches will be marked with "...".

spacing [int, optional (default=3)] Number of spaces between edges. The higher it is, the wider the result.

decimals [int, optional (default=2)] Number of decimal digits to display.

show_weights [bool, optional (default=False)] If true the classification weights will be exported on each leaf. The classification weights are the number of samples each class.

Returns

report [string] Text summary of all the rules in the decision tree.

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.tree import export_text
>>> iris = load_iris()
>>> X = iris['data']
>>> y = iris['target']
>>> decision_tree = DecisionTreeClassifier(random_state=0, max_depth=2)
>>> decision_tree = decision_tree.fit(X, y)
>>> r = export_text(decision_tree, feature_names=iris['feature_names'])
>>> print(r)
|--- petal width (cm) <= 0.80
|   |--- class: 0
|--- petal width (cm) >  0.80
|   |--- petal width (cm) <= 1.75
|   |   |--- class: 1
|   |--- petal width (cm) >  1.75
|   |   |--- class: 2
```

7.37.7 Plotting

tree.plot_tree(decision_tree[, max_depth, ...])	Plot a decision tree.

sklearn.tree.plot_tree

sklearn.tree.**plot_tree**(*decision_tree*, *max_depth=None*, *feature_names=None*, *class_names=None*, *label='all'*, *filled=False*, *impurity=True*, *node_ids=False*, *proportion=False*, *rotate=False*, *rounded=False*, *precision=3*, *ax=None*, *fontsize=None*)

Plot a decision tree.

The sample counts that are shown are weighted with any sample_weights that might be present.

The visualization is fit automatically to the size of the axis. Use the `figsize` or `dpi` arguments of `plt.figure` to control the size of the rendering.

Read more in the *User Guide*.

New in version 0.21.

Parameters

decision_tree [decision tree regressor or classifier] The decision tree to be plotted.

max_depth [int, optional (default=None)] The maximum depth of the representation. If None, the tree is fully generated.

feature_names [list of strings, optional (default=None)] Names of each of the features.

class_names [list of strings, bool or None, optional (default=None)] Names of each of the target classes in ascending numerical order. Only relevant for classification and not supported for multi-output. If `True`, shows a symbolic representation of the class name.

label [{'all', 'root', 'none'}, optional (default='all')] Whether to show informative labels for impurity, etc. Options include 'all' to show at every node, 'root' to show only at the top root node, or 'none' to not show at any node.

filled [bool, optional (default=False)] When set to `True`, paint nodes to indicate majority class for classification, extremity of values for regression, or purity of node for multi-output.

impurity [bool, optional (default=True)] When set to `True`, show the impurity at each node.

node_ids [bool, optional (default=False)] When set to `True`, show the ID number on each node.

proportion [bool, optional (default=False)] When set to `True`, change the display of 'values' and/or 'samples' to be proportions and percentages respectively.

rotate [bool, optional (default=False)] When set to `True`, orient tree left to right rather than top-down.

rounded [bool, optional (default=False)] When set to `True`, draw node boxes with rounded corners and use Helvetica fonts instead of Times-Roman.

precision [int, optional (default=3)] Number of digits of precision for floating point in the values of impurity, threshold and value attributes of each node.

ax [matplotlib axis, optional (default=None)] Axes to plot to. If None, use current axis. Any previous content is cleared.

fontsize [int, optional (default=None)] Size of text font. If None, determined automatically to fit figure.

Returns

annotations [list of artists] List containing the artists for the annotation boxes making up the tree.

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn import tree
```

```
>>> clf = tree.DecisionTreeClassifier(random_state=0)
>>> iris = load_iris()
```

```
>>> clf = clf.fit(iris.data, iris.target)
>>> tree.plot_tree(clf)  # doctest: +SKIP
[Text(251.5,345.217,'X[3] <= 0.8...
```

Examples using `sklearn.tree.plot_tree`

- *Plot the decision surface of a decision tree on the iris dataset*

7.38 `sklearn.utils`: Utilities

The *sklearn.utils* module includes various utilities.

Developer guide: See the *Utilities for Developers* page for further details.

utils.arrayfuncs.min_pos()	Find the minimum value of an array over positive values
utils.as_float_array(X[, copy, force_all_finite])	Converts an array-like to an array of floats.
utils.assert_all_finite(X[, allow_nan])	Throw a ValueError if X contains NaN or infinity.
utils.check_X_y(X, y[, accept_sparse, ...])	Input validation for standard estimators.
utils.check_array(array[, accept_sparse, ...])	Input validation on an array, list, sparse matrix or similar.
utils.check_scalar(x, name, target_type[, ...])	Validate scalar parameters type and value.
utils.check_consistent_length(*arrays)	Check that all arrays have consistent first dimensions.
utils.check_random_state(seed)	Turn seed into a np.random.RandomState instance
utils.class_weight.compute_class_weight(...)	Estimate class weights for unbalanced datasets.
utils.class_weight.compute_sample_weight(...)	Estimate sample weights by class for unbalanced datasets.
utils.deprecated([extra])	Decorator to mark a function or class as deprecated.
utils.estimator_checks.check_estimator(Estimator)	Check if estimator adheres to scikit-learn conventions.
utils.estimator_checks.parametrize_with_checks(...)	Pytest specific decorator for parametrizing estimator checks.
utils.extmath.safe_sparse_dot(a, b[, ...])	Dot product that handle the sparse matrix case correctly
utils.extmath.randomized_range_finder(A, ...)	Computes an orthonormal matrix whose range approximates the range of A.
utils.extmath.randomized_svd(M, n_components)	Computes a truncated randomized SVD
utils.extmath.fast_logdet(A)	Compute log(det(A)) for A symmetric
utils.extmath.density(w, **kwargs)	Compute density of a sparse vector
utils.extmath.weighted_mode(a, w[, axis])	Returns an array of the weighted modal (most common) value in a
utils.gen_even_slices(n, n_packs[, n_samples])	Generator to create n_packs slices going up to n.
utils.graph.single_source_shortest_path_length(...)	Return the shortest path length from source to all reachable nodes.

Continued on next page

Table 305 – continued from previous page

utils.graph_shortest_path.graph_shortest_path()	Perform a shortest-path graph search on a positive directed or undirected graph.
utils.indexable(*iterables)	Make arrays indexable for cross-validation.
utils.metaestimators.if_delegate_has_method(...)	Create a decorator for methods that are delegated to a sub-estimator
utils.multiclass.type_of_target(y)	Determine the type of data indicated by the target.
utils.multiclass.is_multilabel(y)	Check if y is in a multilabel format.
utils.multiclass.unique_labels(*ys)	Extract an ordered array of unique labels
utils.murmurhash3_32()	Compute the 32bit murmurhash3 of key at seed.
utils.resample(*arrays, **options)	Resample arrays or sparse matrices in a consistent way
utils._safe_indexing(X, indices[, axis])	Return rows, items or columns of X using indices.
utils.safe_mask(X, mask)	Return a mask which is safe to use on X.
utils.safe_sqr(X[, copy])	Element wise squaring of array-likes and sparse matrices.
utils.shuffle(*arrays, **options)	Shuffle arrays or sparse matrices in a consistent way
utils.sparsefuncs.incr_mean_variance_axis(X, ...)	Compute incremental mean and variance along an axix on a CSR or CSC matrix.
utils.sparsefuncs.inplace_column_scale(X, scale)	Inplace column scaling of a CSC/CSR matrix.
utils.sparsefuncs.inplace_row_scale(X, scale)	Inplace row scaling of a CSR or CSC matrix.
utils.sparsefuncs.inplace_swap_row(X, m, n)	Swaps two rows of a CSC/CSR matrix in-place.
utils.sparsefuncs.inplace_swap_column(X, m, n)	Swaps two columns of a CSC/CSR matrix in-place.
utils.sparsefuncs.mean_variance_axis(X, axis)	Compute mean and variance along an axix on a CSR or CSC matrix
utils.sparsefuncs.inplace_csr_column_scale(X, ...)	Inplace column scaling of a CSR matrix.
utils.sparsefuncs_fast.inplace_csr_row_normalize_l1()	Inplace row normalize using the l1 norm
utils.sparsefuncs_fast.inplace_csr_row_normalize_l2()	Inplace row normalize using the l2 norm
utils.random.sample_without_replacement()	Sample integers without replacement.
utils.validation.check_is_fitted(estimator)	Perform is_fitted validation for estimator.
utils.validation.check_memory(memory)	Check that memory is joblib.Memory-like.
utils.validation.check_symmetric(array[, ...])	Make sure that array is 2D, square and symmetric.
utils.validation.column_or_1d(y[, warn])	Ravel column or 1d numpy array, else raises an error
utils.validation.has_fit_parameter(...)	Checks whether the estimator's fit method supports the given parameter.
utils.all_estimators([...])	Get a list of all estimators from sklearn.

7.38.1 sklearn.utils.arrayfuncs.min_pos

sklearn.utils.arrayfuncs.**min_pos**()

Find the minimum value of an array over positive values

Returns a huge value if none of the values are positive

7.38.2 sklearn.utils.as_float_array

sklearn.utils.**as_float_array**(*X*, *copy=True*, *force_all_finite=True*)

Converts an array-like to an array of floats.

The new dtype will be np.float32 or np.float64, depending on the original type. The function can create a copy or modify the argument depending on the argument copy.

Parameters

X [{array-like, sparse matrix}]

copy [bool, optional] If True, a copy of X will be created. If False, a copy may still be returned if X's dtype is not a floating point type.

force_all_finite [boolean or 'allow-nan', (default=True)] Whether to raise an error on np.inf and np.nan in X. The possibilities are:

- True: Force all values of X to be finite.
- False: accept both np.inf and np.nan in X.
- 'allow-nan': accept only np.nan values in X. Values cannot be infinite.

New in version 0.20: `force_all_finite` accepts the string `'allow-nan'`.

Returns

XT [{array, sparse matrix}] An array of type np.float

7.38.3 sklearn.utils.assert_all_finite

sklearn.utils.**assert_all_finite**(*X*, *allow_nan=False*)

Throw a ValueError if X contains NaN or infinity.

Parameters

X [array or sparse matrix]

allow_nan [bool]

7.38.4 sklearn.utils.check_X_y

sklearn.utils.**check_X_y**(*X*, *y*, *accept_sparse=False*, *accept_large_sparse=True*, *dtype='numeric'*, *order=None*, *copy=False*, *force_all_finite=True*, *ensure_2d=True*, *allow_nd=False*, *multi_output=False*, *ensure_min_samples=1*, *ensure_min_features=1*, *y_numeric=False*, *warn_on_dtype=None*, *estimator=None*)

Input validation for standard estimators.

Checks X and y for consistent length, enforces X to be 2D and y 1D. By default, X is checked to be non-empty and containing only finite values. Standard input checks are also applied to y, such as checking that y does not have np.nan or np.inf targets. For multi-label y, set multi_output=True to allow 2D and sparse y. If the dtype of X is object, attempt converting to float, raising on failure.

Parameters

X [nd-array, list or sparse matrix] Input data.

y [nd-array, list or sparse matrix] Labels.

accept_sparse [string, boolean or list of string (default=False)] String[s] representing allowed sparse matrix formats, such as 'csc', 'csr', etc. If the input is sparse but not in the allowed format, it will be converted to the first listed format. True allows the input to be any format. False means that a sparse matrix input will raise an error.

accept_large_sparse [bool (default=True)] If a CSR, CSC, COO or BSR sparse matrix is supplied and accepted by accept_sparse, accept_large_sparse will cause it to be accepted only if its indices are stored with a 32-bit dtype.

New in version 0.20.

dtype [string, type, list of types or None (default="numeric")] Data type of result. If None, the dtype of the input is preserved. If "numeric", dtype is preserved unless array.dtype is object. If dtype is a list of types, conversion on the first type is only performed if the dtype of the input is not in the list.

order ['F', 'C' or None (default=None)] Whether an array will be forced to be fortran or c-style.

copy [boolean (default=False)] Whether a forced copy will be triggered. If copy=False, a copy might be triggered by a conversion.

force_all_finite [boolean or 'allow-nan', (default=True)] Whether to raise an error on np.inf and np.nan in X. This parameter does not influence whether y can have np.inf or np.nan values. The possibilities are:

- True: Force all values of X to be finite.
- False: accept both np.inf and np.nan in X.
- 'allow-nan': accept only np.nan values in X. Values cannot be infinite.

New in version 0.20: `force_all_finite` accepts the string `'allow-nan'`.

ensure_2d [boolean (default=True)] Whether to raise a value error if X is not 2D.

allow_nd [boolean (default=False)] Whether to allow X.ndim > 2.

multi_output [boolean (default=False)] Whether to allow 2D y (array or sparse matrix). If false, y will be validated as a vector. y cannot have np.nan or np.inf values if multi_output=True.

ensure_min_samples [int (default=1)] Make sure that X has a minimum number of samples in its first axis (rows for a 2D array).

ensure_min_features [int (default=1)] Make sure that the 2D array has some minimum number of features (columns). The default value of 1 rejects empty datasets. This check is only enforced when X has effectively 2 dimensions or is originally 1D and `ensure_2d` is True. Setting to 0 disables this check.

y_numeric [boolean (default=False)] Whether to ensure that y has a numeric type. If dtype of y is object, it is converted to float64. Should only be used for regression algorithms.

warn_on_dtype [boolean or None, optional (default=None)] Raise DataConversionWarning if the dtype of the input data structure does not match the requested dtype, causing a memory copy.

Deprecated since version 0.21: `warn_on_dtype` is deprecated in version 0.21 and will be removed in 0.23.

estimator [str or estimator instance (default=None)] If passed, include the name of the estimator in warning messages.

Returns

X_converted [object] The converted and validated X.

y_converted [object] The converted and validated y.

7.38.5 sklearn.utils.check_array

sklearn.utils.**check_array**(*array*, *accept_sparse=False*, *accept_large_sparse=True*, *dtype='numeric'*, *order=None*, *copy=False*, *force_all_finite=True*, *ensure_2d=True*, *allow_nd=False*, *ensure_min_samples=1*, *ensure_min_features=1*, *warn_on_dtype=None*, *estimator=None*)

Input validation on an array, list, sparse matrix or similar.

By default, the input is checked to be a non-empty 2D array containing only finite values. If the dtype of the array is object, attempt converting to float, raising on failure.

Parameters

array [object] Input object to check / convert.

accept_sparse [string, boolean or list/tuple of strings (default=False)] String[s] representing allowed sparse matrix formats, such as 'csc', 'csr', etc. If the input is sparse but not in the allowed format, it will be converted to the first listed format. True allows the input to be any format. False means that a sparse matrix input will raise an error.

accept_large_sparse [bool (default=True)] If a CSR, CSC, COO or BSR sparse matrix is supplied and accepted by accept_sparse, accept_large_sparse=False will cause it to be accepted only if its indices are stored with a 32-bit dtype.

New in version 0.20.

dtype [string, type, list of types or None (default="numeric")] Data type of result. If None, the dtype of the input is preserved. If "numeric", dtype is preserved unless array.dtype is object. If dtype is a list of types, conversion on the first type is only performed if the dtype of the input is not in the list.

order ['F', 'C' or None (default=None)] Whether an array will be forced to be fortran or c-style. When order is None (default), then if copy=False, nothing is ensured about the memory layout of the output array; otherwise (copy=True) the memory layout of the returned array is kept as close as possible to the original array.

copy [boolean (default=False)] Whether a forced copy will be triggered. If copy=False, a copy might be triggered by a conversion.

force_all_finite [boolean or 'allow-nan', (default=True)] Whether to raise an error on np.inf and np.nan in array. The possibilities are:

- True: Force all values of array to be finite.
- False: accept both np.inf and np.nan in array.
- 'allow-nan': accept only np.nan values in array. Values cannot be infinite.

For object dtyped data, only np.nan is checked and not np.inf.

New in version 0.20: `force_all_finite` accepts the string `'allow-nan'`.

ensure_2d [boolean (default=True)] Whether to raise a value error if array is not 2D.

allow_nd [boolean (default=False)] Whether to allow array.ndim > 2.

ensure_min_samples [int (default=1)] Make sure that the array has a minimum number of samples in its first axis (rows for a 2D array). Setting to 0 disables this check.

ensure_min_features [int (default=1)] Make sure that the 2D array has some minimum number of features (columns). The default value of 1 rejects empty datasets. This check is only enforced when the input data has effectively 2 dimensions or is originally 1D and `ensure_2d` is True. Setting to 0 disables this check.

warn_on_dtype [boolean or None, optional (default=None)] Raise DataConversionWarning if the dtype of the input data structure does not match the requested dtype, causing a memory copy.

Deprecated since version 0.21: `warn_on_dtype` is deprecated in version 0.21 and will be removed in 0.23.

estimator [str or estimator instance (default=None)] If passed, include the name of the estimator in warning messages.

Returns

array_converted [object] The converted and validated array.

7.38.6 sklearn.utils.check_scalar

sklearn.utils.**check_scalar**(*x*, *name*, *target_type*, *min_val=None*, *max_val=None*)

Validate scalar parameters type and value.

Parameters

x [object] The scalar parameter to validate.

name [str] The name of the parameter to be printed in error messages.

target_type [type or tuple] Acceptable data types for the parameter.

min_val [float or int, optional (default=None)] The minimum valid value the parameter can take. If None (default) it is implied that the parameter does not have a lower bound.

max_val [float or int, optional (default=None)] The maximum valid value the parameter can take. If None (default) it is implied that the parameter does not have an upper bound.

Raises

TypeError If the parameter's type does not match the desired type.

ValueError If the parameter's value violates the given bounds.

7.38.7 sklearn.utils.check_consistent_length

sklearn.utils.**check_consistent_length**(**arrays*)

Check that all arrays have consistent first dimensions.

Checks whether all objects in arrays have the same shape or length.

Parameters

***arrays** [list or tuple of input objects.] Objects that will be checked for consistent length.

7.38.8 sklearn.utils.check_random_state

sklearn.utils.**check_random_state**(*seed*)

Turn seed into a np.random.RandomState instance

Parameters

seed [None | int | instance of RandomState] If seed is None, return the RandomState singleton used by np.random. If seed is an int, return a new RandomState instance seeded with seed. If seed is already a RandomState instance, return it. Otherwise raise ValueError.

Examples using `sklearn.utils.check_random_state`

- *Isotonic Regression*
- *Face completion with a multi-output estimators*
- *Empirical evaluation of the impact of k-means initialization*
- *MNIST classification using multinomial logistic + L1*
- *Manifold Learning methods on a severed sphere*
- *Scaling the regularization parameter for SVCs*

7.38.9 `sklearn.utils.class_weight`.compute_class_weight

`sklearn.utils.class_weight.`**`compute_class_weight`**(*class_weight*, *classes*, *y*)

Estimate class weights for unbalanced datasets.

Parameters

class_weight [dict, 'balanced' or None] If 'balanced', class weights will be given by `n_samples / (n_classes * np.bincount(y))`. If a dictionary is given, keys are classes and values are corresponding class weights. If None is given, the class weights will be uniform.

classes [ndarray] Array of the classes occurring in the data, as given by `np.unique(y_org)` with `y_org` the original class labels.

y [array-like, shape (n_samples,)] Array of original class labels per sample;

Returns

class_weight_vect [ndarray, shape (n_classes,)] Array with class_weight_vect[i] the weight for i-th class

References

The "balanced" heuristic is inspired by Logistic Regression in Rare Events Data, King, Zen, 2001.

7.38.10 `sklearn.utils.class_weight`.compute_sample_weight

`sklearn.utils.class_weight.`**`compute_sample_weight`**(*class_weight*, *y*, *indices=None*)

Estimate sample weights by class for unbalanced datasets.

Parameters

class_weight [dict, list of dicts, "balanced", or None, optional] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data: `n_samples / (n_classes * np.bincount(y))`.

For multi-output, the weights of each column of y will be multiplied.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] Array of original class labels per sample.

indices [array-like, shape (n_subsample,), or None] Array of indices to be used in a subsample. Can be of length less than n_samples in the case of a subsample, or equal to n_samples in the case of a bootstrap subsample with repeated indices. If None, the sample weight will be calculated over the full sample. Only "balanced" is supported for class_weight if this is provided.

Returns

sample_weight_vect [ndarray, shape (n_samples,)] Array with sample weights as applied to the original y

7.38.11 sklearn.utils.deprecated

sklearn.utils.**deprecated**(*extra=''*)

Decorator to mark a function or class as deprecated.

Issue a warning when the function is called/the class is instantiated and adds a warning to the docstring.

The optional extra argument will be appended to the deprecation message and the docstring. Note: to use this with the default value for extra, put in an empty of parentheses:

```
>>> from sklearn.utils import deprecated
>>> deprecated()
<sklearn.utils.deprecation.deprecated object at ...>
```

```
>>> @deprecated()
... def some_function(): pass
```

Parameters

extra [string] to be added to the deprecation messages

7.38.12 sklearn.utils.estimator_checks.check_estimator

sklearn.utils.estimator_checks.**check_estimator**(*Estimator*, *generate_only=False*)

Check if estimator adheres to scikit-learn conventions.

This estimator will run an extensive test-suite for input validation, shapes, etc. Additional tests for classifiers, regressors, clustering or transformers will be run if the Estimator class inherits from the corresponding mixin from sklearn.base.

This test can be applied to classes or instances. Classes currently have some additional tests that related to construction, while passing instances allows the testing of multiple options.

Read more in *Rolling your own estimator*.

Parameters

estimator [estimator object or class] Estimator to check. Estimator is a class object or instance.

generate_only [bool, optional (default=False)] When `False`, checks are evaluated when `check_estimator` is called. When `True`, `check_estimator` returns a generator that yields (estimator, check) tuples. The check is run by calling `check(estimator)`.

New in version 0.22.

Returns

checks_generator [generator] Generator that yields (estimator, check) tuples. Returned when `generate_only=True`.

7.38.13 sklearn.utils.estimator_checks.parametrize_with_checks

sklearn.utils.estimator_checks.**parametrize_with_checks**(*estimators*)

Pytest specific decorator for parametrizing estimator checks.

The `id` of each test is set to be a pprint version of the estimator and the name of the check with its keyword arguments.

Read more in the *User Guide*.

Parameters

estimators [list of estimators objects or classes] Estimators to generated checks for.

Returns

decorator [`pytest.mark.parametrize`]

Examples using sklearn.utils.estimator_checks.parametrize_with_checks

- *Release Highlights for scikit-learn 0.22*

7.38.14 sklearn.utils.extmath.safe_sparse_dot

sklearn.utils.extmath.**safe_sparse_dot**(*a*, *b*, *dense_output=False*)

Dot product that handle the sparse matrix case correctly

Parameters

a [array or sparse matrix]

b [array or sparse matrix]

dense_output [boolean, (default=False)] When False, `a` and `b` both being sparse will yield sparse output. When True, output will always be a dense array.

Returns

dot_product [array or sparse matrix] sparse if `a` and `b` are sparse and `dense_output=False`.

7.38.15 sklearn.utils.extmath.randomized_range_finder

sklearn.utils.extmath.**randomized_range_finder**(*A*, *size*, *n_iter*, *power_iteration_normalizer='auto'*, *random_state=None*)

Computes an orthonormal matrix whose range approximates the range of A.

Parameters

A [2D array] The input data matrix

size [integer] Size of the return array

n_iter [integer] Number of power iterations used to stabilize the result

power_iteration_normalizer ['auto' (default), 'QR', 'LU', 'none'] Whether the power iterations are normalized with step-by-step QR factorization (the slowest but most accurate), 'none' (the fastest but numerically unstable when `n_iter` is large, e.g. typically 5 or larger), or 'LU' factorization (numerically stable but can lose slightly in accuracy). The 'auto' mode applies no normalization if `n_iter` <= 2 and switches to LU otherwise.

New in version 0.18.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Returns

Q [2D array] A (size x size) projection matrix, the range of which approximates well the range of the input matrix A.

Notes

Follows Algorithm 4.3 of Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions Halko, et al., 2009 (arXiv:909) https://arxiv.org/pdf/0909.4061.pdf

An implementation of a randomized algorithm for principal component analysis A. Szlam et al. 2014

7.38.16 sklearn.utils.extmath.randomized_svd

sklearn.utils.extmath.**randomized_svd**(*M*, *n_components*, *n_oversamples=10*, *n_iter='auto'*, *power_iteration_normalizer='auto'*, *transpose='auto'*, *flip_sign=True*, *random_state=0*)

Computes a truncated randomized SVD

Parameters

M [ndarray or sparse matrix] Matrix to decompose

n_components [int] Number of singular values and vectors to extract.

n_oversamples [int (default is 10)] Additional number of random vectors to sample the range of M so as to ensure proper conditioning. The total number of random vectors used to find the range of M is n_components + n_oversamples. Smaller number can improve speed but can negatively impact the quality of approximation of singular vectors and singular values.

n_iter [int or 'auto' (default is 'auto')] Number of power iterations. It can be used to deal with very noisy problems. When 'auto', it is set to 4, unless `n_components` is small (< .1 * min(X.shape)) `n_iter` in which case is set to 7. This improves precision with few components.

Changed in version 0.18.

power_iteration_normalizer ['auto' (default), 'QR', 'LU', 'none'] Whether the power iterations are normalized with step-by-step QR factorization (the slowest but most accurate), 'none' (the fastest but numerically unstable when `n_iter` is large, e.g. typically 5 or larger), or 'LU' factorization (numerically stable but can lose slightly in accuracy). The 'auto' mode applies no normalization if `n_iter` <= 2 and switches to LU otherwise.

New in version 0.18.

transpose [True, False or 'auto' (default)] Whether the algorithm should be applied to M.T instead of M. The result should approximately be the same. The 'auto' mode will trigger the transposition if M.shape[1] > M.shape[0] since this implementation of randomized SVD tend to be a little faster in that case.

Changed in version 0.18.

flip_sign [boolean, (True by default)] The output of a singular value decomposition is only unique up to a permutation of the signs of the singular vectors. If `flip_sign` is set to `True`, the sign ambiguity is resolved by making the largest loadings for each component in the left singular vectors positive.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

Notes

This algorithm finds a (usually very good) approximate truncated singular value decomposition using randomization to speed up the computations. It is particularly fast on large matrices on which you wish to extract only a small number of components. In order to obtain further speed up, `n_iter` can be set <=2 (at the cost of loss of precision).

References

- Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions Halko, et al., 2009 https://arxiv.org/abs/0909.4061
- A randomized algorithm for the decomposition of matrices Per-Gunnar Martinsson, Vladimir Rokhlin and Mark Tygert
- An implementation of a randomized algorithm for principal component analysis A. Szlam et al. 2014

7.38.17 sklearn.utils.extmath.fast_logdet

sklearn.utils.extmath.**fast_logdet**(*A*)

Compute log(det(A)) for A symmetric

Equivalent to : np.log(nl.det(A)) but more robust. It returns -Inf if det(A) is non positive or is not defined.

Parameters

A [array_like] The matrix

7.38.18 sklearn.utils.extmath.density

sklearn.utils.extmath.**density**(*w*, ***kwargs*)

Compute density of a sparse vector

Parameters

w [array_like] The sparse vector

Returns

float The density of w, between 0 and 1

Examples using sklearn.utils.extmath.density

- *Classification of text documents using sparse features*

7.38.19 sklearn.utils.extmath.weighted_mode

sklearn.utils.extmath.**weighted_mode**(*a*, *w*, *axis=0*)

Returns an array of the weighted modal (most common) value in a

If there is more than one such value, only the first is returned. The bin-count for the modal bins is also returned.

This is an extension of the algorithm in scipy.stats.mode.

Parameters

a [array_like] n-dimensional array of which to find mode(s).

w [array_like] n-dimensional array of weights for each value

axis [int, optional] Axis along which to operate. Default is 0, i.e. the first axis.

Returns

vals [ndarray] Array of modal values.

score [ndarray] Array of weighted counts for each mode.

See also:

scipy.stats.mode

Examples

```
>>> from sklearn.utils.extmath import weighted_mode
>>> x = [4, 1, 4, 2, 4, 2]
>>> weights = [1, 1, 1, 1, 1, 1]
>>> weighted_mode(x, weights)
(array([4.]), array([3.]))
```

The value 4 appears three times: with uniform weights, the result is simply the mode of the distribution.

```
>>> weights = [1, 3, 0.5, 1.5, 1, 2]  # deweight the 4's
>>> weighted_mode(x, weights)
(array([2.]), array([3.5]))
```

The value 2 has the highest score: it appears twice with weights of 1.5 and 2: the sum of these is 3.5.

7.38.20 sklearn.utils.gen_even_slices

sklearn.utils.**gen_even_slices**(*n*, *n_packs*, *n_samples=None*)

Generator to create n_packs slices going up to n.

Parameters

n [int]

n_packs [int] Number of slices to generate.

n_samples [int or None (default = None)] Number of samples. Pass n_samples when the slices are to be used for sparse matrix indexing; slicing off-the-end raises an exception, while it works for NumPy arrays.

Yields

slice

Examples

```
>>> from sklearn.utils import gen_even_slices
>>> list(gen_even_slices(10, 1))
[slice(0, 10, None)]
>>> list(gen_even_slices(10, 10))
[slice(0, 1, None), slice(1, 2, None), ..., slice(9, 10, None)]
>>> list(gen_even_slices(10, 5))
[slice(0, 2, None), slice(2, 4, None), ..., slice(8, 10, None)]
>>> list(gen_even_slices(10, 3))
[slice(0, 4, None), slice(4, 7, None), slice(7, 10, None)]
```

7.38.21 sklearn.utils.graph.single_source_shortest_path_length

sklearn.utils.graph.**single_source_shortest_path_length**(*graph*, *source*, *cutoff=None*)

Return the shortest path length from source to all reachable nodes.

Returns a dictionary of shortest path lengths keyed by target.

Parameters

graph [sparse matrix or 2D array (preferably LIL matrix)] Adjacency matrix of the graph

source [integer] Starting node for path

cutoff [integer, optional] Depth to stop the search - only paths of length <= cutoff are returned.

Examples

```
>>> from sklearn.utils.graph import single_source_shortest_path_length
>>> import numpy as np
>>> graph = np.array([[ 0, 1, 0, 0],
...                   [ 1, 0, 1, 0],
...                   [ 0, 1, 0, 1],
...                   [ 0, 0, 1, 0]])
>>> list(sorted(single_source_shortest_path_length(graph, 0).items()))
[(0, 0), (1, 1), (2, 2), (3, 3)]
>>> graph = np.ones((6, 6))
>>> list(sorted(single_source_shortest_path_length(graph, 2).items()))
[(0, 1), (1, 1), (2, 0), (3, 1), (4, 1), (5, 1)]
```

7.38.22 sklearn.utils.graph_shortest_path.graph_shortest_path

sklearn.utils.graph_shortest_path.**graph_shortest_path**()

Perform a shortest-path graph search on a positive directed or undirected graph.

Parameters

dist_matrix [arraylike or sparse matrix, shape = (N,N)] Array of positive distances. If vertex i is connected to vertex j, then dist_matrix[i,j] gives the distance between the vertices. If vertex i is not connected to vertex j, then dist_matrix[i,j] = 0

directed [boolean] if True, then find the shortest path on a directed graph: only progress from a point to its neighbors, not the other way around. if False, then find the shortest path on an undirected graph: the algorithm can progress from a point to its neighbors and vice versa.

method [string ['auto'|'FW'|'D']] method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]

Returns

G [np.ndarray, float, shape = [N,N]] G[i,j] gives the shortest distance from point i to point j along the graph.

Notes

As currently implemented, Dijkstra's algorithm does not work for graphs with direction-dependent distances when directed == False. i.e., if dist_matrix[i,j] and dist_matrix[j,i] are not equal and both are nonzero, method='D' will not necessarily yield the correct result.

Also, these routines have not been tested for graphs with negative distances. Negative distances can lead to infinite cycles that must be handled by specialized algorithms.

7.38.23 sklearn.utils.indexable

sklearn.utils.**indexable**(**iterables*)

Make arrays indexable for cross-validation.

Checks consistent length, passes through None, and ensures that everything can be indexed by converting sparse matrices to csr and converting non-interable objects to arrays.

Parameters

***iterables** [lists, dataframes, arrays, sparse matrices] List of objects to ensure sliceability.

7.38.24 sklearn.utils.metaestimators.if_delegate_has_method

sklearn.utils.metaestimators.**if_delegate_has_method**(*delegate*)

Create a decorator for methods that are delegated to a sub-estimator

This enables ducktyping by hasattr returning True according to the sub-estimator.

Parameters

delegate [string, list of strings or tuple of strings] Name of the sub-estimator that can be accessed as an attribute of the base object. If a list or a tuple of names are provided, the first sub-estimator that is an attribute of the base object will be used.

Examples using sklearn.utils.metaestimators.if_delegate_has_method

- *Inductive Clustering*

7.38.25 sklearn.utils.multiclass.type_of_target

sklearn.utils.multiclass.**type_of_target**(*y*)

Determine the type of data indicated by the target.

Note that this type is the most specific type that can be inferred. For example:

- `binary` is more specific but compatible with `multiclass`.
- `multiclass` of integers is more specific but compatible with `continuous`.
- `multilabel-indicator` is more specific but compatible with `multiclass-multioutput`.

Parameters

y [array-like]

Returns

target_type [string] One of:

- 'continuous': `y` is an array-like of floats that are not all integers, and is 1d or a column vector.
- 'continuous-multioutput': `y` is a 2d array of floats that are not all integers, and both dimensions are of size > 1.
- 'binary': `y` contains <= 2 discrete values and is 1d or a column vector.
- 'multiclass': `y` contains more than two discrete values, is not a sequence of sequences, and is 1d or a column vector.
- 'multiclass-multioutput': `y` is a 2d array that contains more than two discrete values, is not a sequence of sequences, and both dimensions are of size > 1.
- 'multilabel-indicator': `y` is a label indicator matrix, an array of two dimensions with at least two columns, and at most 2 unique values.
- 'unknown': `y` is array-like but none of the above, such as a 3d array, sequence of sequences, or an array of non-sequence objects.

Examples

```
>>> import numpy as np
>>> type_of_target([0.1, 0.6])
'continuous'
>>> type_of_target([1, -1, -1, 1])
'binary'
>>> type_of_target(['a', 'b', 'a'])
'binary'
>>> type_of_target([1.0, 2.0])
'binary'
>>> type_of_target([1, 0, 2])
'multiclass'
>>> type_of_target([1.0, 0.0, 3.0])
'multiclass'
>>> type_of_target(['a', 'b', 'c'])
'multiclass'
>>> type_of_target(np.array([[1, 2], [3, 1]]))
'multiclass-multioutput'
>>> type_of_target([[1, 2]])
'multilabel-indicator'
>>> type_of_target(np.array([[1.5, 2.0], [3.0, 1.6]]))
'continuous-multioutput'
>>> type_of_target(np.array([[0, 1], [1, 1]]))
'multilabel-indicator'
```

7.38.26 sklearn.utils.multiclass.is_multilabel

sklearn.utils.multiclass.**is_multilabel**(*y*)

Check if `y` is in a multilabel format.

Parameters

y [numpy array of shape [n_samples]] Target values.

Returns

out [bool,] Return `True`, if `y` is in a multilabel format, else ``False`.

Examples

```
>>> import numpy as np
>>> from sklearn.utils.multiclass import is_multilabel
>>> is_multilabel([0, 1, 0, 1])
False
>>> is_multilabel([[1], [0, 2], []])
False
>>> is_multilabel(np.array([[1, 0], [0, 0]]))
True
>>> is_multilabel(np.array([[1], [0], [0]]))
False
>>> is_multilabel(np.array([[1, 0, 0]]))
True
```

7.38.27 sklearn.utils.multiclass.unique_labels

sklearn.utils.multiclass.**unique_labels**(**ys*)

Extract an ordered array of unique labels

We don't allow:

- mix of multilabel and multiclass (single label) targets
- mix of label indicator matrix and anything else, because there are no explicit labels)
- mix of label indicator matrices of different sizes
- mix of string and integer labels

At the moment, we also don't allow "multiclass-multioutput" input type.

Parameters

***ys** [array-likes]

Returns

out [numpy array of shape [n_unique_labels]] An ordered array of unique labels.

Examples

```
>>> from sklearn.utils.multiclass import unique_labels
>>> unique_labels([3, 5, 5, 5, 7, 7])
array([3, 5, 7])
>>> unique_labels([1, 2, 3, 4], [2, 2, 3, 4])
array([1, 2, 3, 4])
>>> unique_labels([1, 2, 10], [5, 11])
array([ 1,  2,  5, 10, 11])
```

7.38.28 sklearn.utils.murmurhash3_32

sklearn.utils.**murmurhash3_32**()

Compute the 32bit murmurhash3 of key at seed.

The underlying implementation is MurmurHash3_x86_32 generating low latency 32bits hash suitable for implementing lookup tables, Bloom filters, count min sketch or feature hashing.

Parameters

key [int32, bytes, unicode or ndarray with dtype int32] the physical object to hash

seed [int, optional default is 0] integer seed for the hashing algorithm.

positive [boolean, optional default is False]

True: the results is casted to an unsigned int from 0 to 2 ** 32 - 1

False: the results is casted to a signed int from -(2 ** 31) to 2 ** 31 - 1

7.38.29 sklearn.utils.resample

sklearn.utils.**resample**(**arrays*, ***options*)

Resample arrays or sparse matrices in a consistent way

The default strategy implements one step of the bootstrapping procedure.

Parameters

***arrays** [sequence of indexable data-structures] Indexable data-structures can be arrays, lists, dataframes or scipy sparse matrices with consistent first dimension.

Returns

resampled_arrays [sequence of indexable data-structures] Sequence of resampled copies of the collections. The original arrays are not impacted.

Other Parameters

replace [boolean, True by default] Implements resampling with replacement. If False, this will implement (sliced) random permutations.

n_samples [int, None by default] Number of samples to generate. If left to None this is automatically set to the first dimension of the arrays. If replace is False it should not be larger than the length of arrays.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

stratify [array-like or None (default=None)] If not None, data is split in a stratified fashion, using this as the class labels.

See also:

sklearn.utils.shuffle

Examples

It is possible to mix sparse and dense arrays in the same run:

```
>>> X = np.array([[1., 0.], [2., 1.], [0., 0.]])
>>> y = np.array([0, 1, 2])

>>> from scipy.sparse import coo_matrix
>>> X_sparse = coo_matrix(X)

>>> from sklearn.utils import resample
>>> X, X_sparse, y = resample(X, X_sparse, y, random_state=0)
>>> X
array([[1., 0.],
       [2., 1.],
       [1., 0.]])

>>> X_sparse
<3x2 sparse matrix of type '<... 'numpy.float64'>'
    with 4 stored elements in Compressed Sparse Row format>

>>> X_sparse.toarray()
array([[1., 0.],
       [2., 1.],
       [1., 0.]])
```

(continues on next page)

(continued from previous page)

```
>>> y
array([0, 1, 0])

>>> resample(y, n_samples=2, random_state=0)
array([0, 1])
```

Example using stratification:

```
>>> y = [0, 0, 1, 1, 1, 1, 1, 1, 1]
>>> resample(y, n_samples=5, replace=False, stratify=y,
...          random_state=0)
[1, 1, 1, 0, 1]
```

7.38.30 sklearn.utils._safe_indexing

sklearn.utils.**_safe_indexing**(*X*, *indices*, *axis=0*)

Return rows, items or columns of X using indices.

> **Warning:** This utility is documented, but **private**. This means that backward compatibility might be broken without any deprecation cycle.

Parameters

X [array-like, sparse-matrix, list, pandas.DataFrame, pandas.Series] Data from which to sample rows, items or columns. `list` are only supported when `axis=0`.

indices [bool, int, str, slice, array-like]

- If `axis=0`, boolean and integer array-like, integer slice, and scalar integer are supported.
- **If `axis=1`:**
  - to select a single column, `indices` can be of `int` type for all `X` types and `str` only for dataframe. The selected subset will be 1D, unless `X` is a sparse matrix in which case it will be 2D.
  - to select multiples columns, `indices` can be one of the following: `list`, `array`, `slice`. The type used in these containers can be one of the following: `int`, 'bool' and `str`. However, `str` is only supported when `X` is a dataframe. The selected subset will be 2D.

axis [int, default=0] The axis along which `X` will be subsampled. `axis=0` will select rows while `axis=1` will select columns.

Returns

subset Subset of X on axis 0 or 1.

Notes

CSR, CSC, and LIL sparse matrices are supported. COO sparse matrices are not supported.

7.38.31 sklearn.utils.safe_mask

sklearn.utils.**safe_mask**(*X*, *mask*)

Return a mask which is safe to use on X.

Parameters

X [{array-like, sparse matrix}] Data on which to apply mask.

mask [array] Mask to be used on X.

Returns

mask

7.38.32 sklearn.utils.safe_sqr

sklearn.utils.**safe_sqr**(*X*, *copy=True*)

Element wise squaring of array-likes and sparse matrices.

Parameters

X [array like, matrix, sparse matrix]

copy [boolean, optional, default True] Whether to create a copy of X and operate on it or to perform inplace computation (default behaviour).

Returns

X ** 2 [element wise square]

7.38.33 sklearn.utils.shuffle

sklearn.utils.**shuffle**(**arrays*, ***options*)

Shuffle arrays or sparse matrices in a consistent way

This is a convenience alias to `resample(*arrays, replace=False)` to do random permutations of the collections.

Parameters

***arrays** [sequence of indexable data-structures] Indexable data-structures can be arrays, lists, dataframes or scipy sparse matrices with consistent first dimension.

Returns

shuffled_arrays [sequence of indexable data-structures] Sequence of shuffled copies of the collections. The original arrays are not impacted.

Other Parameters

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

n_samples [int, None by default] Number of samples to generate. If left to None this is automatically set to the first dimension of the arrays.

See also:

sklearn.utils.resample

Examples

It is possible to mix sparse and dense arrays in the same run:

```
>>> X = np.array([[1., 0.], [2., 1.], [0., 0.]])
>>> y = np.array([0, 1, 2])

>>> from scipy.sparse import coo_matrix
>>> X_sparse = coo_matrix(X)

>>> from sklearn.utils import shuffle
>>> X, X_sparse, y = shuffle(X, X_sparse, y, random_state=0)
>>> X
array([[0., 0.],
       [2., 1.],
       [1., 0.]])

>>> X_sparse
<3x2 sparse matrix of type '<... 'numpy.float64'>'
    with 3 stored elements in Compressed Sparse Row format>

>>> X_sparse.toarray()
array([[0., 0.],
       [2., 1.],
       [1., 0.]])

>>> y
array([2, 1, 0])

>>> shuffle(y, n_samples=2, random_state=0)
array([0, 1])
```

Examples using `sklearn.utils.shuffle`

- *Approximate nearest neighbors in TSNE*

7.38.34 `sklearn.utils.sparsefuncs.incr_mean_variance_axis`

`sklearn.utils.sparsefuncs.`**`incr_mean_variance_axis`**(*X*, *axis*, *last_mean*, *last_var*, *last_n*)

Compute incremental mean and variance along an axix on a CSR or CSC matrix.

last_mean, last_var are the statistics computed at the last step by this function. Both must be initialized to 0-arrays of the proper size, i.e. the number of features in X. last_n is the number of samples encountered until now.

Parameters

X [CSR or CSC sparse matrix, shape (n_samples, n_features)] Input data.

axis [int (either 0 or 1)] Axis along which the axis should be computed.

last_mean [float array with shape (n_features,)] Array of feature-wise means to update with the new data X.

last_var [float array with shape (n_features,)] Array of feature-wise var to update with the new data X.

last_n [int with shape (n_features,)] Number of samples seen so far, excluded X.

Returns

means [float array with shape (n_features,)] Updated feature-wise means.

variances [float array with shape (n_features,)] Updated feature-wise variances.

n [int with shape (n_features,)] Updated number of seen samples.

Notes

NaNs are ignored in the algorithm.

7.38.35 sklearn.utils.sparsefuncs.inplace_column_scale

sklearn.utils.sparsefuncs.**inplace_column_scale**(*X*, *scale*)

Inplace column scaling of a CSC/CSR matrix.

Scale each feature of the data matrix by multiplying with specific scale provided by the caller assuming a (n_samples, n_features) shape.

Parameters

X [CSC or CSR matrix with shape (n_samples, n_features)] Matrix to normalize using the variance of the features.

scale [float array with shape (n_features,)] Array of precomputed feature-wise values to use for scaling.

7.38.36 sklearn.utils.sparsefuncs.inplace_row_scale

sklearn.utils.sparsefuncs.**inplace_row_scale**(*X*, *scale*)

Inplace row scaling of a CSR or CSC matrix.

Scale each row of the data matrix by multiplying with specific scale provided by the caller assuming a (n_samples, n_features) shape.

Parameters

X [CSR or CSC sparse matrix, shape (n_samples, n_features)] Matrix to be scaled.

scale [float array with shape (n_features,)] Array of precomputed sample-wise values to use for scaling.

7.38.37 sklearn.utils.sparsefuncs.inplace_swap_row

sklearn.utils.sparsefuncs.**inplace_swap_row**(*X*, *m*, *n*)

Swaps two rows of a CSC/CSR matrix in-place.

Parameters

X [CSR or CSC sparse matrix, shape=(n_samples, n_features)] Matrix whose two rows are to be swapped.

m [int] Index of the row of X to be swapped.

n [int] Index of the row of X to be swapped.

7.38.38 sklearn.utils.sparsefuncs.inplace_swap_column

sklearn.utils.sparsefuncs.**inplace_swap_column**(*X*, *m*, *n*)

Swaps two columns of a CSC/CSR matrix in-place.

Parameters

X [CSR or CSC sparse matrix, shape=(n_samples, n_features)] Matrix whose two columns are to be swapped.

m [int] Index of the column of X to be swapped.

n [int] Index of the column of X to be swapped.

7.38.39 sklearn.utils.sparsefuncs.mean_variance_axis

sklearn.utils.sparsefuncs.**mean_variance_axis**(*X*, *axis*)

Compute mean and variance along an axix on a CSR or CSC matrix

Parameters

X [CSR or CSC sparse matrix, shape (n_samples, n_features)] Input data.

axis [int (either 0 or 1)] Axis along which the axis should be computed.

Returns

means [float array with shape (n_features,)] Feature-wise means

variances [float array with shape (n_features,)] Feature-wise variances

7.38.40 sklearn.utils.sparsefuncs.inplace_csr_column_scale

sklearn.utils.sparsefuncs.**inplace_csr_column_scale**(*X*, *scale*)

Inplace column scaling of a CSR matrix.

Scale each feature of the data matrix by multiplying with specific scale provided by the caller assuming a (n_samples, n_features) shape.

Parameters

X [CSR matrix with shape (n_samples, n_features)] Matrix to normalize using the variance of the features.

scale [float array with shape (n_features,)] Array of precomputed feature-wise values to use for scaling.

7.38.41 sklearn.utils.sparsefuncs_fast.inplace_csr_row_normalize_l1

sklearn.utils.sparsefuncs_fast.**inplace_csr_row_normalize_l1**()

Inplace row normalize using the l1 norm

7.38.42 sklearn.utils.sparsefuncs_fast.inplace_csr_row_normalize_l2

sklearn.utils.sparsefuncs_fast.**inplace_csr_row_normalize_l2**()
Inplace row normalize using the l2 norm

7.38.43 sklearn.utils.random.sample_without_replacement

sklearn.utils.random.**sample_without_replacement**()
Sample integers without replacement.

Select n_samples integers from the set [0, n_population) without replacement.

Parameters

n_population [int,] The size of the set to sample from.

n_samples [int,] The number of integer to sample.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

method ["auto", "tracking_selection", "reservoir_sampling" or "pool"] If method == "auto", the ratio of n_samples / n_population is used to determine which algorithm to use: If ratio is between 0 and 0.01, tracking selection is used. If ratio is between 0.01 and 0.99, numpy.random.permutation is used. If ratio is greater than 0.99, reservoir sampling is used. The order of the selected integers is undefined. If a random order is desired, the selected subset should be shuffled.

If method =="tracking_selection", a set based implementation is used which is suitable for `n_samples` <<< `n_population`.

If method == "reservoir_sampling", a reservoir sampling algorithm is used which is suitable for high memory constraint or when O(`n_samples`) ~ O(`n_population`). The order of the selected integers is undefined. If a random order is desired, the selected subset should be shuffled.

If method == "pool", a pool based algorithm is particularly fast, even faster than the tracking selection method. Hovewer, a vector containing the entire population has to be initialized. If n_samples ~ n_population, the reservoir sampling method is faster.

Returns

out [array of size (n_samples,)] The sampled subsets of integer. The subset of selected integer might not be randomized, see the method argument.

7.38.44 sklearn.utils.validation.check_is_fitted

sklearn.utils.validation.**check_is_fitted**(*estimator*, *attributes=None*, *msg=None*, *all_or_any=<built-in function all>*)
Perform is_fitted validation for estimator.

Checks if the estimator is fitted by verifying the presence of fitted attributes (ending with a trailing underscore) and otherwise raises a NotFittedError with the given message.

This utility is meant to be used internally by estimators themselves, typically in their own predict / transform methods.

Parameters

estimator [estimator instance.] estimator instance for which the check is performed.

attributes [str, list or tuple of str, default=None] Attribute name(s) given as string or a list/tuple of strings Eg.: `["coef_", "estimator_", ...], "coef_"`

If `None`, `estimator` is considered fitted if there exist an attribute that ends with a underscore and does not start with double underscore.

msg [string] The default error message is, "This %(name)s instance is not fitted yet. Call 'fit' with appropriate arguments before using this estimator."

For custom messages if "%(name)s" is present in the message string, it is substituted for the estimator name.

Eg. : "Estimator, %(name)s, must be fitted before sparsifying".

all_or_any [callable, {all, any}, default all] Specify whether all or any of the given attributes must exist.

Returns

None

Raises

NotFittedError If the attributes are not found.

7.38.45 sklearn.utils.validation.check_memory

sklearn.utils.validation.**check_memory**(*memory*)

Check that `memory` is joblib.Memory-like.

joblib.Memory-like means that `memory` can be converted into a joblib.Memory instance (typically a str denoting the `location`) or has the same interface (has a `cache` method).

Parameters

memory [None, str or object with the joblib.Memory interface]

Returns

memory [object with the joblib.Memory interface]

Raises

ValueError If `memory` is not joblib.Memory-like.

7.38.46 sklearn.utils.validation.check_symmetric

sklearn.utils.validation.**check_symmetric**(*array*, *tol=1e-10*, *raise_warning=True*, *raise_exception=False*)

Make sure that array is 2D, square and symmetric.

If the array is not symmetric, then a symmetrized version is returned. Optionally, a warning or exception is raised if the matrix is not symmetric.

Parameters

array [nd-array or sparse matrix] Input object to check / convert. Must be two-dimensional and square, otherwise a ValueError will be raised.

tol [float] Absolute tolerance for equivalence of arrays. Default = 1E-10.

raise_warning [boolean (default=True)] If True then raise a warning if conversion is required.

raise_exception [boolean (default=False)] If True then raise an exception if array is not symmetric.

Returns

array_sym [ndarray or sparse matrix] Symmetrized version of the input array, i.e. the average of array and array.transpose(). If sparse, then duplicate entries are first summed and zeros are eliminated.

7.38.47 sklearn.utils.validation.column_or_1d

sklearn.utils.validation.**column_or_1d**(*y*, *warn=False*)

Ravel column or 1d numpy array, else raises an error

Parameters

y [array-like]

warn [boolean, default False] To control display of warnings.

Returns

y [array]

7.38.48 sklearn.utils.validation.has_fit_parameter

sklearn.utils.validation.**has_fit_parameter**(*estimator*, *parameter*)

Checks whether the estimator's fit method supports the given parameter.

Parameters

estimator [object] An estimator to inspect.

parameter [str] The searched parameter.

Returns

is_parameter: bool Whether the parameter was found to be a named parameter of the estimator's fit method.

Examples

```
>>> from sklearn.svm import SVC
>>> has_fit_parameter(SVC(), "sample_weight")
True
```

7.38.49 sklearn.utils.all_estimators

sklearn.utils.**all_estimators**(*include_meta_estimators=None*, *include_other=None*, *type_filter=None*, *include_dont_test=None*)

Get a list of all estimators from sklearn.

This function crawls the module and gets all classes that inherit from BaseEstimator. Classes that are defined in test-modules are not included. By default meta_estimators such as GridSearchCV are also not included.

Parameters

include_meta_estimators [boolean, default=False] Deprecated, ignored.

Deprecated since version 0.21: `include_meta_estimators` has been deprecated and has no effect in 0.21 and will be removed in 0.23.

include_other [boolean, default=False] Deprecated, ignored.

Deprecated since version 0.21: `include_other` has been deprecated and has not effect in 0.21 and will be removed in 0.23.

type_filter [string, list of string, or None, default=None] Which kind of estimators should be returned. If None, no filter is applied and all estimators are returned. Possible values are 'classifier', 'regressor', 'cluster' and 'transformer' to get estimators only of these specific types, or a list of these to get the estimators that fit at least one of the types.

include_dont_test [boolean, default=False] Deprecated, ignored.

Deprecated since version 0.21: `include_dont_test` has been deprecated and has no effect in 0.21 and will be removed in 0.23.

Returns

estimators [list of tuples] List of (name, class), where `name` is the class name as string and `class` is the actuall type of the class.

Utilities from joblib:

utils.parallel_backend(backend[, n_jobs, ...])	Change the default backend used by Parallel inside a with block.
utils.register_parallel_backend(name, factory)	Register a new Parallel backend factory.

7.38.50 sklearn.utils.parallel_backend

sklearn.utils.**parallel_backend**(*backend*, *n_jobs=-1*, *inner_max_num_threads=None*, ***backend_params*)

Change the default backend used by Parallel inside a with block.

If `backend` is a string it must match a previously registered implementation using the `register_parallel_backend` function.

By default the following backends are available:

- 'loky': single-host, process-based parallelism (used by default),
- 'threading': single-host, thread-based parallelism,
- 'multiprocessing': legacy single-host, process-based parallelism.

'loky' is recommended to run functions that manipulate Python objects. 'threading' is a low-overhead alternative that is most efficient for functions that release the Global Interpreter Lock: e.g. I/O-bound code or CPU-bound code in a few calls to native code that explicitly releases the GIL.

In addition, if the `dask` and `distributed` Python packages are installed, it is possible to use the 'dask' backend for better scheduling of nested parallel calls without over-subscription and potentially distribute parallel calls over a networked cluster of several hosts.

Alternatively the backend can be passed directly as an instance.

By default all available workers will be used (`n_jobs=-1`) unless the caller passes an explicit value for the `n_jobs` parameter.

This is an alternative to passing a `backend='backend_name'` argument to the `Parallel` class constructor. It is particularly useful when calling into library code that uses joblib internally but does not expose the backend argument in its own API.

```
>>> from operator import neg
>>> with parallel_backend('threading'):
...     print(Parallel()(delayed(neg)(i + 1) for i in range(5)))
...
[-1, -2, -3, -4, -5]
```

Warning: this function is experimental and subject to change in a future version of joblib.

Joblib also tries to limit the oversubscription by limiting the number of threads usable in some third-party library threadpools like OpenBLAS, MKL or OpenMP. The default limit in each worker is set to `max(cpu_count() // effective_n_jobs, 1)` but this limit can be overwritten with the `inner_max_num_threads` argument which will be used to set this limit in the child processes.

New in version 0.10.

7.38.51 sklearn.utils.register_parallel_backend

sklearn.utils.**register_parallel_backend**(*name*, *factory*, *make_default=False*)

Register a new Parallel backend factory.

The new backend can then be selected by passing its name as the backend argument to the Parallel class. Moreover, the default backend can be overwritten globally by setting make_default=True.

The factory can be any callable that takes no argument and return an instance of `ParallelBackendBase`.

Warning: this function is experimental and subject to change in a future version of joblib.

New in version 0.10.

7.39 Recently deprecated

7.39.1 To be removed in 0.23

utils.Memory(**kwargs)

Attributes

utils.Parallel(**kwargs)

Methods

sklearn.utils.Memory

Warning: DEPRECATED

class sklearn.utils.**Memory**(***kwargs*)

Attributes

cachedir

Methods

cache(self[, func, ignore, verbose, mmap_mode])	Decorates the given function func to only compute its return value for input arguments not cached on disk.
clear(self[, warn])	Erase the complete cache directory.
eval(self, func, *args, **kwargs)	Eval function func with arguments `*args` and `**kwargs`, in the context of the memory.
format(self, obj[, indent])	Return the formatted representation of the object.
reduce_size(self)	Remove cache elements to make cache size fit in `bytes_limit`.

debug	
warn	

__init__(**args*, ***kwargs*)
DEPRECATED: deprecated in version 0.20.1 to be removed in version 0.23. Please import this functionality directly from joblib, which can be installed with: pip install joblib.

cache(*self*, *func=None*, *ignore=None*, *verbose=None*, *mmap_mode=False*)
Decorates the given function func to only compute its return value for input arguments not cached on disk.

Parameters

func: callable, optional The function to be decorated

ignore: list of strings A list of arguments name to ignore in the hashing

verbose: integer, optional The verbosity mode of the function. By default that of the memory object is used.

mmap_mode: {None, 'r+', 'r', 'w+', 'c'}, optional The memmapping mode used when loading from cache numpy arrays. See numpy.load for the meaning of the arguments. By default that of the memory object is used.

Returns

decorated_func: MemorizedFunc object The returned object is a MemorizedFunc object, that is callable (behaves like a function), but offers extra methods for cache lookup and management. See the documentation for `joblib.memory.MemorizedFunc`.

clear(*self*, *warn=True*)
Erase the complete cache directory.

eval(*self*, *func*, **args*, ***kwargs*)
Eval function func with arguments `*args` and `**kwargs`, in the context of the memory.

This method works similarly to the builtin `apply`, except that the function is called only if the cache is not up to date.

format(*self*, *obj*, *indent=0*)
Return the formatted representation of the object.

reduce_size(*self*)
Remove cache elements to make cache size fit in `bytes_limit`.

sklearn.utils.Parallel

Warning: DEPRECATED

class sklearn.utils.**Parallel**(***kwargs*)

Methods

`__call__`(self, iterable)	Call self as a function.
dispatch_next(self)	Dispatch more data for parallel processing
dispatch_one_batch(self, iterator)	Prefetch the tasks for the next batch and dispatch them.
format(self, obj[, indent])	Return the formatted representation of the object.
print_progress(self)	Display the process of the parallel execution only a fraction of time, controlled by self.verbose.

debug	
retrieve	
warn	

__init__(**args*, ***kwargs*)
DEPRECATED: deprecated in version 0.20.1 to be removed in version 0.23. Please import this functionality directly from joblib, which can be installed with: pip install joblib.

dispatch_next(*self*)
Dispatch more data for parallel processing

This method is meant to be called concurrently by the multiprocessing callback. We rely on the thread-safety of dispatch_one_batch to protect against concurrent consumption of the unprotected iterator.

dispatch_one_batch(*self*, *iterator*)
Prefetch the tasks for the next batch and dispatch them.

The effective size of the batch is computed here. If there are no more jobs to dispatch, return False, else return True.

The iterator consumption and dispatching is protected by the same lock so calling this function should be thread safe.

format(*self*, *obj*, *indent=0*)
Return the formatted representation of the object.

print_progress(*self*)
Display the process of the parallel execution only a fraction of time, controlled by self.verbose.

utils.cpu_count()	DEPRECATED: deprecated in version 0.20.1 to be removed in version 0.23.
utils.delayed(function[, check_pickle])	DEPRECATED: deprecated in version 0.20.1 to be removed in version 0.23.
metrics.calinski_harabaz_score(X, labels)	DEPRECATED: Function 'calinski_harabaz_score' has been renamed to 'calinski_harabasz_score' and will be removed in version 0.23.
metrics.jaccard_similarity_score(y_true, y_pred)	Jaccard similarity coefficient score
linear_model.logistic_regression_path(X, y)	DEPRECATED: logistic_regression_path was deprecated in version 0.21 and will be removed in version 0.23.0
utils.safe_indexing(X, indices[, axis])	DEPRECATED: safe_indexing is deprecated in version 0.22 and will be removed in version 0.24.

sklearn.utils.cpu_count

Warning: DEPRECATED

sklearn.utils.**cpu_count**()
DEPRECATED: deprecated in version 0.20.1 to be removed in version 0.23. Please import this functionality directly from joblib, which can be installed with: pip install joblib.

Return the number of CPUs.

sklearn.utils.delayed

Warning: DEPRECATED

sklearn.utils.**delayed**(*function*, *check_pickle=None*)
DEPRECATED: deprecated in version 0.20.1 to be removed in version 0.23. Please import this functionality directly from joblib, which can be installed with: pip install joblib.

Decorator used to capture the arguments of a function.

sklearn.metrics.calinski_harabaz_score

Warning: DEPRECATED

sklearn.metrics.**calinski_harabaz_score**(*X*, *labels*)
DEPRECATED: Function 'calinski_harabaz_score' has been renamed to 'calinski_harabasz_score' and will be removed in version 0.23.

sklearn.metrics.jaccard_similarity_score

Warning: DEPRECATED

sklearn.metrics.**jaccard_similarity_score**(*y_true*, *y_pred*, *normalize=True*, *sample_weight=None*)

Jaccard similarity coefficient score

Deprecated since version 0.21: This is deprecated to be removed in 0.23, since its handling of binary and multiclass inputs was broken. `jaccard_score` has an API that is consistent with precision_score, f_score, etc.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) labels.

y_pred [1d array-like, or label indicator array / sparse matrix] Predicted labels, as returned by a classifier.

normalize [bool, optional (default=True)] If `False`, return the sum of the Jaccard similarity coefficient over the sample set. Otherwise, return the average of Jaccard similarity coefficient.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] If `normalize == True`, return the average Jaccard similarity coefficient, else it returns the sum of the Jaccard similarity coefficient over the sample set.

The best performance is 1 with `normalize == True` and the number of samples with `normalize == False`.

See also:

accuracy_score*, *hamming_loss*, *zero_one_loss

Notes

In binary and multiclass classification, this function is equivalent to the `accuracy_score`. It differs in the multilabel classification problem.

References

[1]

sklearn.linear_model.logistic_regression_path

Warning: DEPRECATED

sklearn.linear_model.**logistic_regression_path**(*X*, *y*, *pos_class=None*, *Cs=10*, *fit_intercept=True*, *max_iter=100*, *tol=0.0001*, *verbose=0*, *solver='lbfgs'*, *coef=None*, *class_weight=None*, *dual=False*, *penalty='l2'*, *intercept_scaling=1.0*, *multi_class='auto'*, *random_state=None*, *check_input=True*, *max_squared_sum=None*, *sample_weight=None*, *l1_ratio=None*)

DEPRECATED: logistic_regression_path was deprecated in version 0.21 and will be removed in version 0.23.0

Compute a Logistic Regression model for a list of regularization parameters.

This is an implementation that uses the result of the previous model to speed up computations along the set of solutions, making it faster than sequentially calling LogisticRegression for the different parameters. Note that there will be no speedup with liblinear solver, since it does not handle warm-starting.

Deprecated since version 0.21: `logistic_regression_path` was deprecated in version 0.21 and will be removed in 0.23.

Read more in the *User Guide*.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)]

Input data.

y [array-like, shape (n_samples,) or (n_samples, n_targets)] Input data, target values.

pos_class [int, None] The class with respect to which we perform a one-vs-all fit. If None, then it is assumed that the given problem is binary.

Cs [int | array-like, shape (n_cs,)] List of values for the regularization parameter or integer specifying the number of regularization parameters that should be used. In this case, the parameters will be chosen in a logarithmic scale between 1e-4 and 1e4.

fit_intercept [bool] Whether to fit an intercept for the model. In this case the shape of the returned array is (n_cs, n_features + 1).

max_iter [int] Maximum number of iterations for the solver.

tol [float] Stopping criterion. For the newton-cg and lbfgs solvers, the iteration will stop when `max{|g_i | i = 1, ..., n} <= tol` where `g_i` is the i-th component of the gradient.

verbose [int] For the liblinear and lbfgs solvers set verbose to any positive number for verbosity.

solver [{'lbfgs', 'newton-cg', 'liblinear', 'sag', 'saga'}] Numerical solver to use.

coef [array-like, shape (n_features,), default None] Initialization value for coefficients of logistic regression. Useless for liblinear solver.

class_weight [dict or 'balanced', optional] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

dual [bool] Dual or primal formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer dual=False when n_samples > n_features.

penalty [str, 'l1', 'l2', or 'elasticnet'] Used to specify the norm used in the penalization. The 'newton-cg', 'sag' and 'lbfgs' solvers support only l2 penalties. 'elasticnet' is only supported by the 'saga' solver.

intercept_scaling [float, default 1.] Useful only when the solver 'liblinear' is used and self.fit_intercept is set to True. In this case, x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equal to intercept_scaling is appended to the instance vector. The intercept becomes `intercept_scaling * synthetic_feature_weight`.

Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.

multi_class [{'ovr', 'multinomial', 'auto'}, default='auto'] If the option chosen is 'ovr', then a binary problem is fit for each label. For 'multinomial' the loss minimised is the multinomial loss fit across the entire probability distribution, *even when the data is binary*. 'multinomial' is unavailable when solver='liblinear'. 'auto' selects 'ovr' if the data is binary, or if solver='liblinear', and otherwise selects 'multinomial'.

New in version 0.18: Stochastic Average Gradient descent solver for 'multinomial' case.

Changed in version 0.22: Default changed from 'ovr' to 'auto' in 0.22.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'sag' or 'liblinear'.

check_input [bool, default True] If False, the input arrays X and y will not be checked.

max_squared_sum [float, default None] Maximum squared sum of X over samples. Used only in SAG solver. If None, it will be computed, going through all the samples. The value should be precomputed to speed up cross validation.

sample_weight [array-like, shape(n_samples,) optional] Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight.

l1_ratio [float or None, optional (default=None)] The Elastic-Net mixing parameter, with `0 <= l1_ratio <= 1`. Only used if `penalty='elasticnet'`. Setting `l1_ratio=0` is equivalent to using `penalty='l2'`, while setting `l1_ratio=1` is equivalent to using `penalty='l1'`. For `0 < l1_ratio <1`, the penalty is a combination of L1 and L2.

Returns

coefs [ndarray, shape (n_cs, n_features) or (n_cs, n_features + 1)]

List of coefficients for the Logistic Regression model. If fit_intercept is set to True then the second dimension will be n_features + 1, where the last item represents the intercept. For `multiclass='multinomial'`, the shape is (n_classes, n_cs, n_features) or (n_classes, n_cs, n_features + 1).

Cs [ndarray] Grid of Cs used for cross-validation.

n_iter [array, shape (n_cs,)] Actual number of iteration for each Cs.

Notes

You might get slightly different results with the solver liblinear than with the others since this uses LIBLINEAR which penalizes the intercept.

Changed in version 0.19: The "copy" parameter was removed.

sklearn.utils.safe_indexing

Warning: DEPRECATED

sklearn.utils.**safe_indexing**(*X*, *indices*, *axis=0*)

DEPRECATED: safe_indexing is deprecated in version 0.22 and will be removed in version 0.24.

Return rows, items or columns of X using indices.

Deprecated since version 0.22: This function was deprecated in version 0.22 and will be removed in version 0.24.

Parameters

X [array-like, sparse-matrix, list, pandas.DataFrame, pandas.Series]

Data from which to sample rows, items or columns. `list` are only supported when `axis=0`.

indices : bool, int, str, slice, array-like

- If `axis=0`, boolean and integer array-like, integer slice, and scalar integer are supported.
- If `axis=1`:
  - to select a single column, `indices` can be of `int` type for all `X` types and `str` only for dataframe. The selected subset will be 1D, unless `X` is a sparse matrix in which case it will be 2D.
  - to select multiples columns, `indices` can be one of the following: `list`, `array`, `slice`. The type used in these containers can be one of the following: `int`, 'bool' and `str`. However, `str` is only supported when `X` is a dataframe. The selected subset will be 2D.

axis [int, default=0] The axis along which `X` will be subsampled. `axis=0` will select rows while `axis=1` will select columns.

Returns

subset Subset of X on axis 0 or 1.

Notes

CSR, CSC, and LIL sparse matrices are supported. COO sparse matrices are not supported.

ensemble.partial_dependence.partial_dependence(...)	DEPRECATED: The function ensemble.partial_dependence has been deprecated in favour of inspection.partial_dependence in 0.21 and will be removed in 0.23.
ensemble.partial_dependence.plot_partial_dependence(...)	DEPRECATED: The function ensemble.plot_partial_dependence has been deprecated in favour of sklearn.inspection.plot_partial_dependence in 0.21 and will be removed in 0.23.

sklearn.ensemble.partial_dependence.partial_dependence

sklearn.ensemble.partial_dependence.**partial_dependence**(*gbrt*, *target_variables*, *grid=None*, *X=None*, *percentiles=(0.05*, *0.95)*, *grid_resolution=100*)

DEPRECATED: The function ensemble.partial_dependence has been deprecated in favour of inspection.partial_dependence in 0.21 and will be removed in 0.23.

Partial dependence of `target_variables`.

Partial dependence plots show the dependence between the joint values of the `target_variables` and the function represented by the `gbrt`.

Read more in the *User Guide*.

Deprecated since version 0.21: This function was deprecated in version 0.21 in favor of *sklearn.inspection.partial_dependence* and will be removed in 0.23.

Parameters

gbrt [BaseGradientBoosting]

A fitted gradient boosting model.

target_variables [array-like, dtype=int] The target features for which the partial dependency should be computed (size should be smaller than 3 for visual renderings).

grid [array-like of shape (n_points, n_target_variables)] The grid of `target_variables` values for which the partial dependency should be evaluated (either `grid` or `X` must be specified).

X [array-like of shape (n_samples, n_features)] The data on which `gbrt` was trained. It is used to generate a `grid` for the `target_variables`. The `grid` comprises `grid_resolution` equally spaced points between the two `percentiles`.

percentiles [(low, high), default=(0.05, 0.95)] The lower and upper percentile used create the extreme values for the `grid`. Only if `X` is not None.

grid_resolution [int, default=100] The number of equally spaced points on the `grid`.

Returns

pdp [array, shape=(n_classes, n_points)]

The partial dependence function evaluated on the `grid`. For regression and binary classification `n_classes==1`.

axes [seq of ndarray or None] The axes with which the grid has been created or None if the grid has been given.

Examples

```
>>> samples = [[0, 0, 2], [1, 0, 0]]
>>> labels = [0, 1]
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> gb = GradientBoostingClassifier(random_state=0).fit(samples, labels)
>>> kwargs = dict(X=samples, percentiles=(0, 1), grid_resolution=2)
>>> partial_dependence(gb, [0], **kwargs) # doctest: +SKIP
(array([[-4.52...,  4.52...]]), [array([ 0.,  1.])])
```

sklearn.ensemble.partial_dependence.plot_partial_dependence

sklearn.ensemble.partial_dependence.**plot_partial_dependence**(*gbrt*, *X*, *features*, *feature_names=None*, *label=None*, *n_cols=3*, *grid_resolution=100*, *percentiles=(0.05, 0.95)*, *n_jobs=None*, *verbose=0*, *ax=None*, *line_kw=None*, *contour_kw=None*, ***fig_kw*)

DEPRECATED: The function ensemble.plot_partial_dependence has been deprecated in favour of sklearn.inspection.plot_partial_dependence in 0.21 and will be removed in 0.23.

Partial dependence plots for `features`.

The `len(features)` plots are arranged in a grid with `n_cols` columns. Two-way partial dependence plots are plotted as contour plots.

Read more in the *User Guide*.

Deprecated since version 0.21: This function was deprecated in version 0.21 in favor of *sklearn.inspection.plot_partial_dependence* and will be removed in 0.23.

Parameters

gbrt [BaseGradientBoosting]

A fitted gradient boosting model.

X [array-like of shape (n_samples, n_features)] The data on which `gbrt` was trained.

features [seq of ints, strings, or tuples of ints or strings] If seq[i] is an int or a tuple with one int value, a one-way PDP is created; if seq[i] is a tuple of two ints, a two-way PDP is created. If feature_names is specified and seq[i] is an int, seq[i] must be < len(feature_names). If seq[i] is a string, feature_names must be specified, and seq[i] must be in feature_names.

feature_names [seq of str] Name of each feature; feature_names[i] holds the name of the feature with index i.

label [object] The class label for which the PDPs should be computed. Only if gbrt is a multi-class model. Must be in `gbrt.classes_`.

n_cols [int] The number of columns in the grid plot (default: 3).

grid_resolution [int, default=100] The number of equally spaced points on the axes.

percentiles [(low, high), default=(0.05, 0.95)] The lower and upper percentile used to create the extreme values for the PDP axes.

n_jobs [int or None, optional (default=None)] `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [int] Verbose output during PD computations. Defaults to 0.

ax [Matplotlib axis object, default None] An axis object onto which the plots will be drawn.

line_kw [dict] Dict with keywords passed to the `matplotlib.pyplot.plot` call. For one-way partial dependence plots.

contour_kw [dict] Dict with keywords passed to the `matplotlib.pyplot.plot` call. For two-way partial dependence plots.

****fig_kw** [dict] Dict with keywords passed to the figure() call. Note that all keywords not recognized above will be automatically included here.

Returns

fig [figure]

The Matplotlib Figure object.

axs [seq of Axis objects] A seq of Axis objects, one for each subplot.

Examples

```
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> X, y = make_friedman1()
>>> clf = GradientBoostingRegressor(n_estimators=10).fit(X, y)
>>> fig, axs = plot_partial_dependence(clf, X, [0, (0, 1)]) #doctest: +SKIP
...
```

CHAPTER

EIGHT

DEVELOPER'S GUIDE

8.1 Contributing

This project is a community effort, and everyone is welcome to contribute.

The project is hosted on https://github.com/scikit-learn/scikit-learn

The decision making process and governance structure of scikit-learn is laid out in the governance document: *Scikit-learn governance and decision-making*.

Scikit-learn is somewhat *selective* when it comes to adding new algorithms, and the best way to contribute and to help the project is to start working on known issues. See *Issues for New Contributors* to get started.

Our community, our values

We are a community based on openness and friendly, didactic, discussions.

We aspire to treat everybody equally, and value their contributions.

Decisions are made based on technical merit and consensus.

Code is not the only way to help the project. Reviewing pull requests, answering questions to help others on mailing lists or issues, organizing and teaching tutorials, working on the website, improving the documentation, are all priceless contributions.

We abide by the principles of openness, respect, and consideration of others of the Python Software Foundation: https://www.python.org/psf/codeofconduct/

In case you experience issues using this package, do not hesitate to submit a ticket to the GitHub issue tracker. You are also welcome to post feature requests or pull requests.

8.1.1 Ways to contribute

There are many ways to contribute to scikit-learn, with the most common ones being contribution of code or documentation to the project. Improving the documentation is no less important than improving the library itself. If you find a typo in the documentation, or have made improvements, do not hesitate to send an email to the mailing list or preferably submit a GitHub pull request. Full documentation can be found under the doc/ directory.

But there are many other ways to help. In particular answering queries on the issue tracker, investigating bugs, and *reviewing other developers' pull requests* are very valuable contributions that decrease the burden on the project maintainers.

Another way to contribute is to report issues you're facing, and give a "thumbs up" on issues that others reported and that are relevant to you. It also helps us if you spread the word: reference the project from your blog and articles, link to it from your website, or simply star to say "I use it":

In case a contribution/issue involves changes to the API principles or changes to dependencies or supported versions, it must be backed by a *Enhancement proposals (SLEPs)*, where a SLEP must be submitted as a pull-request to enhancement proposals using the SLEP template and follows the decision-making process outlined in *Scikit-learn governance and decision-making*.

Contributing to related projects

Scikit-learn thrives in an ecosystem of several related projects, which also may have relevant issues to work on, including smaller projects such as:

- scikit-learn-contrib
- joblib
- sphinx-gallery
- numpydoc
- liac-arff

and larger projects:

- numpy
- scipy
- matplotlib
- and so on.

Look for issues marked "help wanted" or similar. Helping these projects may help Scikit-learn too. See also *Related Projects*.

8.1.2 Submitting a bug report or a feature request

We use GitHub issues to track all bugs and feature requests; feel free to open an issue if you have found a bug or wish to see a feature implemented.

In case you experience issues using this package, do not hesitate to submit a ticket to the Bug Tracker. You are also welcome to post feature requests or pull requests.

It is recommended to check that your issue complies with the following rules before submitting:

- Verify that your issue is not being currently addressed by other issues or pull requests.
- If you are submitting an algorithm or feature request, please verify that the algorithm fulfills our new algorithm requirements.
- If you are submitting a bug report, we strongly encourage you to follow the guidelines in *How to make a good bug report*.

How to make a good bug report

When you submit an issue to Github, please do your best to follow these guidelines! This will make it a lot easier to provide you with good feedback:

- The ideal bug report contains a **short reproducible code snippet**, this way anyone can try to reproduce the bug easily (see this for more details). If your snippet is longer than around 50 lines, please link to a gist or a github repo.
- If not feasible to include a reproducible snippet, please be specific about what **estimators and/or functions are involved and the shape of the data**.
- If an exception is raised, please **provide the full traceback**.
- Please include your **operating system type and version number**, as well as your **Python, scikit-learn, numpy, and scipy versions**. This information can be found by running the following code snippet:

```
>>> import sklearn
>>> sklearn.show_versions()
```

Note: This utility function is only available in scikit-learn v0.20+. For previous versions, one has to explicitly run:

```
import platform; print(platform.platform())
import sys; print("Python", sys.version)
import numpy; print("NumPy", numpy.__version__)
import scipy; print("SciPy", scipy.__version__)
import sklearn; print("Scikit-Learn", sklearn.__version__)
```

- Please ensure all **code snippets and error messages are formatted in appropriate code blocks**. See Creating and highlighting code blocks for more details.

8.1.3 Contributing code

Note: To avoid duplicating work, it is highly advised that you search through the issue tracker and the PR list. If in doubt about duplicated work, or if you want to work on a non-trivial feature, it's recommended to first open an issue in the issue tracker to get some feedbacks from core developers.

How to contribute

The preferred way to contribute to scikit-learn is to fork the main repository on GitHub, then submit a "pull request" (PR).

In the first few steps, we explain how to locally install scikit-learn, and how to set up your git repository:

1. Create an account on GitHub if you do not already have one.
2. Fork the project repository: click on the 'Fork' button near the top of the page. This creates a copy of the code under your account on the GitHub user account. For more details on how to fork a repository see this guide.
3. Clone your fork of the scikit-learn repo from your GitHub account to your local disk:

```
$ git clone git@github.com:YourLogin/scikit-learn.git
$ cd scikit-learn
```

4. Install the development dependencies:

```
$ pip install cython pytest pytest-cov flake8
```

5. Install scikit-learn in editable mode:

```
$ pip install --editable .
```

   for more details about advanced installation, see the *Building from source* section.

6. Add the `upstream` remote. This saves a reference to the main scikit-learn repository, which you can use to keep your repository synchronized with the latest changes:

```
$ git remote add upstream https://github.com/scikit-learn/scikit-learn.git
```

You should now have a working installation of scikit-learn, and your git repository properly configured. The next steps now describe the process of modifying code and submitting a PR:

7. Synchronize your master branch with the upstream master branch:

```
$ git checkout master
$ git pull upstream master
```

8. Create a feature branch to hold your development changes:

```
$ git checkout -b my_feature
```

   and start making changes. Always use a feature branch. It's good practice to never work on the `master` branch!

9. Develop the feature on your feature branch on your computer, using Git to do the version control. When you're done editing, add changed files using `git add` and then `git commit`:

```
$ git add modified_files
$ git commit
```

   to record your changes in Git, then push the changes to your GitHub account with:

```
$ git push -u origin my_feature
```

10. Follow these instructions to create a pull request from your fork. This will send an email to the committers. You may want to consider sending an email to the mailing list for more visibility.

Note: If you are modifying a Cython module, you have to re-run step 5 after modifications and before testing them.

It is often helpful to keep your local feature branch synchronized with the latest changes of the main scikit-learn repository:

```
$ git fetch upstream
$ git merge upstream/master
```

Subsequently, you might need to solve the conflicts. You can refer to the Git documentation related to resolving merge conflict using the command line.

Learning git:

The Git documentation and http://try.github.io are excellent resources to get started with git, and understanding all of the commands shown here.

Pull request checklist

Before a PR can be merged, it needs to be approved by two core developers. Please prefix the title of your pull request with `[MRG]` if the contribution is complete and should be subjected to a detailed review. An incomplete contribution – where you expect to do more work before receiving a full review – should be prefixed `[WIP]` (to indicate a work in progress) and changed to `[MRG]` when it matures. WIPs may be useful to: indicate you are working on something to avoid duplicated work, request broad review of functionality or API, or seek collaborators. WIPs often benefit from the inclusion of a task list in the PR description.

In order to ease the reviewing process, we recommend that your contribution complies with the following rules before marking a PR as `[MRG]`. The **bolded** ones are especially important:

1. **Give your pull request a helpful title** that summarises what your contribution does. This title will often become the commit message once merged so it should summarise your contribution for posterity. In some cases "Fix <ISSUE TITLE>" is enough. "Fix #<ISSUE NUMBER>" is never a good title.

2. **Make sure your code passes the tests**. The whole test suite can be run with `pytest`, but it is usually not recommended since it takes a long time. It is often enough to only run the test related to your changes: for example, if you changed something in `sklearn/linear_model/logistic.py`, running the following commands will usually be enough:

   - `pytest sklearn/linear_model/logistic.py` to make sure the doctest examples are correct
   - `pytest sklearn/linear_model/tests/test_logistic.py` to run the tests specific to the file
   - `pytest sklearn/linear_model` to test the whole *`linear_model`* module
   - `pytest doc/modules/linear_model.rst` to make sure the user guide examples are correct.
   - `pytest sklearn/tests/test_common.py -k LogisticRegression` to run all our estimator checks (specifically for `LogisticRegression`, if that's the estimator you changed).

   There may be other failing tests, but they will be caught by the CI so you don't need to run the whole test suite locally. For guidelines on how to use `pytest` efficiently, see the *Useful pytest aliases and flags*.

3. **Make sure your code is properly commented and documented**, and **make sure the documentation renders properly**. To build the documentation, please refer to our *Documentation* guidelines. The CI will also build the docs: please refer to *Generated documentation on CircleCI*.

4. **Tests are necessary for enhancements to be accepted**. Bug-fixes or new features should be provided with non-regression tests. These tests verify the correct behavior of the fix or feature. In this manner, further modifications on the code base are granted to be consistent with the desired behavior. In the case of bug fixes, at the time of the PR, the non-regression tests should fail for the code base in the master branch and pass for the PR code.

5. **Make sure that your PR does not add PEP8 violations**. On a Unix-like system, you can run `make flake8-diff`. `flake8 path_to_file`, would work for any system, but please avoid reformatting parts of the file that your pull request doesn't change, as it distracts from code review.

6. Follow the *Coding guidelines*.

7. When applicable, use the validation tools and scripts in the `sklearn.utils` submodule. A list of utility routines available for developers can be found in the *Utilities for Developers* page.

8. Often pull requests resolve one or more other issues (or pull requests). If merging your pull request means that some other issues/PRs should be closed, you should use keywords to create link to them (e.g., `Fixes #1234`; multiple issues/PRs are allowed as long as each one is preceded by a keyword). Upon merging, those issues/PRs will automatically be closed by GitHub. If your pull request is simply related to some other issues/PRs, create a link to them without using the keywords (e.g., `See also #1234`).

9. PRs should often substantiate the change, through benchmarks of performance and efficiency or through examples of usage. Examples also illustrate the features and intricacies of the library to users. Have a look at other

examples in the examples/ directory for reference. Examples should demonstrate why the new functionality is useful in practice and, if possible, compare it to other methods available in scikit-learn.

10. New features often need to be illustrated with narrative documentation in the user guide, with small code snippets. If relevant, please also add references in the literature, with PDF links when possible.
11. The user guide should also include expected time and space complexity of the algorithm and scalability, e.g. "this algorithm can scale to a large number of samples > 100000, but does not scale in dimensionality: n_features is expected to be lower than 100".

You can also check our *Code Review Guidelines* to get an idea of what reviewers will expect.

You can check for common programming errors with the following tools:

- Code with a good unittest coverage (at least 80%, better 100%), check with:

```
$ pip install pytest pytest-cov
$ pytest --cov sklearn path/to/tests_for_package
```

see also *Testing and improving test coverage*

Bonus points for contributions that include a performance analysis with a benchmark script and profiling output (please report on the mailing list or on the GitHub issue).

Also check out the *How to optimize for speed* guide for more details on profiling and Cython optimizations.

Note: The current state of the scikit-learn code base is not compliant with all of those guidelines, but we expect that enforcing those constraints on all new contributions will get the overall code base quality in the right direction.

Note: For two very well documented and more detailed guides on development workflow, please pay a visit to the Scipy Development Workflow - and the Astropy Workflow for Developers sections.

Continuous Integration (CI)

- Azure pipelines are used for testing scikit-learn on Linux, Mac and Windows, with different dependencies and settings.
- CircleCI is used to build the docs for viewing, for linting with flake8, and for testing with PyPy on Linux

Please note that if one of the following markers appear in the latest commit message, the following actions are taken.

Commit Message Marker	Action Taken by CI
[scipy-dev]	Add a Travis build with our dependencies (numpy, scipy, etc ...) development builds
[ci skip]	CI is skipped completely
[lint skip]	Azure pipeline skips linting
[doc skip]	Docs are not built
[doc quick]	Docs built, but excludes example gallery plots
[doc build]	Docs built including example gallery plots

Stalled pull requests

As contributing a feature can be a lengthy process, some pull requests appear inactive but unfinished. In such a case, taking them over is a great service for the project.

A good etiquette to take over is:

- **Determine if a PR is stalled**
  - A pull request may have the label "stalled" or "help wanted" if we have already identified it as a candidate for other contributors.
  - To decide whether an inactive PR is stalled, ask the contributor if she/he plans to continue working on the PR in the near future. Failure to respond within 2 weeks with an activity that moves the PR forward suggests that the PR is stalled and will result in tagging that PR with "help wanted".

    Note that if a PR has received earlier comments on the contribution that have had no reply in a month, it is safe to assume that the PR is stalled and to shorten the wait time to one day.

    After a sprint, follow-up for un-merged PRs opened during sprint will be communicated to participants at the sprint, and those PRs will be tagged "sprint". PRs tagged with "sprint" can be reassigned or declared stalled by sprint leaders.
- **Taking over a stalled PR**: To take over a PR, it is important to comment on the stalled PR that you are taking over and to link from the new PR to the old one. The new PR should be created by pulling from the old one.

Issues for New Contributors

New contributors should look for the following tags when looking for issues. We strongly recommend that new contributors tackle "easy" issues first: this helps the contributor become familiar with the contribution workflow, and for the core devs to become acquainted with the contributor; besides which, we frequently underestimate how easy an issue is to solve!

good first issue tag

A great way to start contributing to scikit-learn is to pick an item from the list of good first issues in the issue tracker. Resolving these issues allow you to start contributing to the project without much prior knowledge. If you have already contributed to scikit-learn, you should look at Easy issues instead.

Easy tag

If you have already contributed to scikit-learn, another great way to contribute to scikit-learn is to pick an item from the list of Easy issues in the issue tracker. Your assistance in this area will be greatly appreciated by the more experienced developers as it helps free up their time to concentrate on other issues.

help wanted tag

We often use the help wanted tag to mark issues regardless of difficulty. Additionally, we use the help wanted tag to mark Pull Requests which have been abandoned by their original contributor and are available for someone to pick up where the original contributor left off. The list of issues with the help wanted tag can be found here .

Note that not all issues which need contributors will have this tag.

8.1.4 Documentation

We are glad to accept any sort of documentation: function docstrings, reStructuredText documents (like this one), tutorials, etc. reStructuredText documents live in the source code repository under the `doc/` directory.

You can edit the documentation using any text editor, and then generate the HTML output by typing `make` from the `doc/` directory. Alternatively, `make html` may be used to generate the documentation **with** the example gallery (which takes quite some time). The resulting HTML files will be placed in `_build/html/stable` and are viewable in a web browser.

Building the documentation

First, make sure you have *properly installed* the development version.

Building the documentation requires installing some additional packages:

```
pip install sphinx sphinx-gallery numpydoc matplotlib Pillow pandas scikit-image␣
↪packaging
```

To build the documentation, you need to be in the `doc` folder:

```
cd doc
```

In the vast majority of cases, you only need to generate the full web site, without the example gallery:

```
make
```

The documentation will be generated in the `_build/html/stable` directory. To also generate the example gallery you can use:

```
make html
```

This will run all the examples, which takes a while. If you only want to generate a few examples, you can use:

```
EXAMPLES_PATTERN=your_regex_goes_here make html
```

This is particularly useful if you are modifying a few examples.

Set the environment variable `NO_MATHJAX=1` if you intend to view the documentation in an offline setting.

To build the PDF manual, run:

```
make latexpdf
```

Warning: Sphinx version

While we do our best to have the documentation build under as many versions of Sphinx as possible, the different versions tend to behave slightly differently. To get the best results, you should use the same version as the one we used on CircleCI. Look at this github search to know the exact version.

Guidelines for writing documentation

It is important to keep a good compromise between mathematical and algorithmic details, and give intuition to the reader on what the algorithm does.

Basically, to elaborate on the above, it is best to always start with a small paragraph with a hand-waving explanation of what the method does to the data. Then, it is very helpful to point out why the feature is useful and when it should be used - the latter also including "big O" ($O\left(g\left(n\right)\right)$) complexities of the algorithm, as opposed to just *rules of thumb*, as the latter can be very machine-dependent. If those complexities are not available, then rules of thumb may be provided instead.

Secondly, a generated figure from an example (as mentioned in the previous paragraph) should then be included to further provide some intuition.

Next, one or two small code examples to show its use can be added.

Next, any math and equations, followed by references, can be added to further the documentation. Not starting the documentation with the maths makes it more friendly towards users that are just interested in what the feature will do, as opposed to how it works "under the hood".

Finally, follow the formatting rules below to make it consistently good:

- Add "See also" in docstrings for related classes/functions.
- "See also" in docstrings should be one line per reference, with a colon and an explanation, for example:

```
See also
--------
SelectKBest : Select features based on the k highest scores.
SelectFpr : Select features based on a false positive rate test.
```

- When documenting the parameters and attributes, here is a list of some well-formatted examples:

```
n_clusters : int, default=3
    The number of clusters detected by the algorithm.

some_param : {'hello', 'goodbye'}, bool or int, default=True
    The parameter description goes here, which can be either a string
    literal (either `hello` or `goodbye`), a bool, or an int. The default
    value is True.

array_parameter : {array-like, sparse matrix, dataframe} of shape (n_samples, n_
↪features) or (n_samples,)
    This parameter accepts data in either of the mentioned forms, with one
    of the mentioned shapes. The default value is
    `np.ones(shape=(n_samples,))`.

list_param : list of int

typed_ndarray : ndarray of shape (n_samples,), dtype=np.int32

sample_weight : array-like of shape (n_samples,), default=None
```

In general have the following in mind:

1. Use Python basic types. (`bool` instead of `boolean`)
2. Use parenthesis for defining shapes: `array-like of shape (n_samples,)` or `array-like of shape (n_samples, n_features)`
3. For strings with multiple options, use brackets: `input: {'log', 'squared', 'multinomial'}`
4. 1D or 2D data can be a subset of `{array-like, ndarray, sparse matrix, dataframe}`. Note that `array-like` can also be a `list`, while `ndarray` is explicitly only a `numpy.ndarray`.
5. When specifying the data type of a list, use `of` as a delimiter: `list of int`.

6. When specifying the dtype of an ndarray, use e.g. `dtype=np.int32` after defining the shape: `ndarray of shape (n_samples,), dtype=np.int32`.
7. When the default is `None`, `None` only needs to be specified at the end with `default=None`. Be sure to include in the docstring, what it means for the parameter or attribute to be `None`.

- For unwritten formatting rules, try to follow existing good works:
  - For "References" in docstrings, see the Silhouette Coefficient (*`sklearn.metrics.silhouette_score`*).
- When editing reStructuredText (`.rst`) files, try to keep line length under 80 characters when possible (exceptions include links and tables).
- Before submitting you pull request check if your modifications have introduced new sphinx warnings and try to fix them.

Generated documentation on CircleCI

When you change the documentation in a pull request, CircleCI automatically builds it. To view the documentation generated by CircleCI:

- navigate to the bottom of your pull request page to see the CI statuses. You may need to click on "Show all checks" to see all the CI statuses.
- click on the CircleCI status with "doc" in the title.
- add `#artifacts` at the end of the URL. Note: you need to wait for the CircleCI build to finish before being able to look at the artifacts.
- once the artifacts are visible, navigate to `doc/_changed.html` to see a list of documentation pages that are likely to be affected by your pull request. Navigate to `doc/index.html` to see the full generated html documentation.

If you often need to look at the documentation generated by CircleCI, e.g. when reviewing pull requests, you may find *this tip* very handy.

8.1.5 Testing and improving test coverage

High-quality unit testing is a corner-stone of the scikit-learn development process. For this purpose, we use the pytest package. The tests are functions appropriately named, located in `tests` subdirectories, that check the validity of the algorithms and the different options of the code.

Running `pytest` in a folder will run all the tests of the corresponding subpackages. For a more detailed `pytest` workflow, please refer to the *Pull request checklist*.

We expect code coverage of new features to be at least around 90%.

Writing matplotlib related tests

Test fixtures ensure that a set of tests will be executing with the appropriate initialization and cleanup. The scikit-learn test suite implements a fixture which can be used with `matplotlib`.

`pyplot` The `pyplot` fixture should be used when a test function is dealing with `matplotlib`. `matplotlib` is a soft dependency and is not required. This fixture is in charge of skipping the tests if `matplotlib` is not installed. In addition, figures created during the tests will be automatically closed once the test function has been executed.

To use this fixture in a test function, one needs to pass it as an argument:

```
def test_requiring_mpl_fixture(pyplot):
    # you can now safely use matplotlib
```

Workflow to improve test coverage

To test code coverage, you need to install the coverage package in addition to pytest.

1. **Run 'make test-coverage'. The output lists for each file the line** numbers that are not tested.
2. **Find a low hanging fruit, looking at which lines are not tested,** write or adapt a test specifically for these lines.
3. Loop.

8.1.6 Issue Tracker Tags

All issues and pull requests on the GitHub issue tracker should have (at least) one of the following tags:

Bug / Crash Something is happening that clearly shouldn't happen. Wrong results as well as unexpected errors from estimators go here.

Cleanup / Enhancement Improving performance, usability, consistency.

Documentation Missing, incorrect or sub-standard documentations and examples.

New Feature Feature requests and pull requests implementing a new feature.

There are four other tags to help new contributors:

good first issue This issue is ideal for a first contribution to scikit-learn. Ask for help if the formulation is unclear. If you have already contributed to scikit-learn, look at Easy issues instead.

Easy This issue can be tackled without much prior experience.

Moderate Might need some knowledge of machine learning or the package, but is still approachable for someone new to the project.

help wanted This tag marks an issue which currently lacks a contributor or a PR that needs another contributor to take over the work. These issues can range in difficulty, and may not be approachable for new contributors. Note that not all issues which need contributors will have this tag.

8.1.7 Maintaining backwards compatibility

Deprecation

If any publicly accessible method, function, attribute or parameter is renamed, we still support the old one for two releases and issue a deprecation warning when it is called/passed/accessed. E.g., if the function `zero_one` is renamed to `zero_one_loss`, we add the decorator `deprecated` (from `sklearn.utils`) to `zero_one` and call `zero_one_loss` from that function:

```
from ..utils import deprecated

def zero_one_loss(y_true, y_pred, normalize=True):
    # actual implementation
    pass
```

(continues on next page)

(continued from previous page)

```
@deprecated("Function 'zero_one' was renamed to 'zero_one_loss' "
            "in version 0.13 and will be removed in release 0.15. "
            "Default behavior is changed from 'normalize=False' to "
            "'normalize=True'")
def zero_one(y_true, y_pred, normalize=False):
    return zero_one_loss(y_true, y_pred, normalize)
```

If an attribute is to be deprecated, use the decorator `deprecated` on a property. Please note that the `property` decorator should be placed before the `deprecated` decorator for the docstrings to be rendered properly. E.g., renaming an attribute `labels_` to `classes_` can be done as:

```
@deprecated("Attribute labels_ was deprecated in version 0.13 and "
            "will be removed in 0.15. Use 'classes_' instead")
@property
def labels_(self):
    return self.classes_
```

If a parameter has to be deprecated, a `FutureWarning` warning must be raised too. In the following example, k is deprecated and renamed to n_clusters:

```
import warnings

def example_function(n_clusters=8, k='deprecated'):
    if k != 'deprecated':
        warnings.warn("'k' was renamed to n_clusters in version 0.13 and "
                      "will be removed in 0.15.",
                      FutureWarning)
        n_clusters = k
```

When the change is in a class, we validate and raise warning in `fit`:

```
import warnings

class ExampleEstimator(BaseEstimator):
    def __init__(self, n_clusters=8, k='deprecated'):
        self.n_clusters = n_clusters
        self.k = k

    def fit(self, X, y):
        if self.k != 'deprecated':
            warnings.warn("'k' was renamed to n_clusters in version 0.13 and "
                          "will be removed in 0.15.",
                          FutureWarning)
            self._n_clusters = self.k
        else:
            self._n_clusters = self.n_clusters
```

As in these examples, the warning message should always give both the version in which the deprecation happened and the version in which the old behavior will be removed. If the deprecation happened in version 0.x-dev, the message should say deprecation occurred in version 0.x and the removal will be in 0.(x+2), so that users will have enough time to adapt their code to the new behaviour. For example, if the deprecation happened in version 0.18-dev, the message should say it happened in version 0.18 and the old behavior will be removed in version 0.20.

In addition, a deprecation note should be added in the docstring, recalling the same information as the deprecation warning as explained above. Use the `.. deprecated::` directive:

```
.. deprecated:: 0.13
   ``k`` was renamed to ``n_clusters`` in version 0.13 and will be removed
   in 0.15.
```

What's more, a deprecation requires a test which ensures that the warning is raised in relevant cases but not in other cases. The warning should be caught in all other tests (using e.g., `@pytest.mark.filterwarnings`), and there should be no warning in the examples.

Change the default value of a parameter

If the default value of a parameter needs to be changed, please replace the default value with a specific value (e.g., `warn`) and raise `FutureWarning` when users are using the default value. In the following example, we change the default value of `n_clusters` from 5 to 10 (current version is 0.20):

```
import warnings

def example_function(n_clusters='warn'):
    if n_clusters == 'warn':
        warnings.warn("The default value of n_clusters will change from "
                      "5 to 10 in 0.22.", FutureWarning)
        n_clusters = 5
```

When the change is in a class, we validate and raise warning in `fit`:

```
import warnings

class ExampleEstimator:
    def __init__(self, n_clusters='warn'):
        self.n_clusters = n_clusters

    def fit(self, X, y):
        if self.n_clusters == 'warn':
          warnings.warn("The default value of n_clusters will change from "
                        "5 to 10 in 0.22.", FutureWarning)
          self._n_clusters = 5
```

Similar to deprecations, the warning message should always give both the version in which the change happened and the version in which the old behavior will be removed. The docstring needs to be updated accordingly. We need a test which ensures that the warning is raised in relevant cases but not in other cases. The warning should be caught in all other tests (using e.g., `@pytest.mark.filterwarnings`), and there should be no warning in the examples.

8.1.8 Code Review Guidelines

Reviewing code contributed to the project as PRs is a crucial component of scikit-learn development. We encourage anyone to start reviewing code of other developers. The code review process is often highly educational for everybody involved. This is particularly appropriate if it is a feature you would like to use, and so can respond critically about whether the PR meets your needs. While each pull request needs to be signed off by two core developers, you can speed up this process by providing your feedback.

Here are a few important aspects that need to be covered in any code review, from high-level questions to a more detailed check-list.

- Do we want this in the library? Is it likely to be used? Do you, as a scikit-learn user, like the change and intend to use it? Is it in the scope of scikit-learn? Will the cost of maintaining a new feature be worth its benefits?

- Is the code consistent with the API of scikit-learn? Are public functions/classes/parameters well named and intuitively designed?
- Are all public functions/classes and their parameters, return types, and stored attributes named according to scikit-learn conventions and documented clearly?
- Is any new functionality described in the user-guide and illustrated with examples?
- Is every public function/class tested? Are a reasonable set of parameters, their values, value types, and combinations tested? Do the tests validate that the code is correct, i.e. doing what the documentation says it does? If the change is a bug-fix, is a non-regression test included? Look at this to get started with testing in Python.
- Do the tests pass in the continuous integration build? If appropriate, help the contributor understand why tests failed.
- Do the tests cover every line of code (see the coverage report in the build log)? If not, are the lines missing coverage good exceptions?
- Is the code easy to read and low on redundancy? Should variable names be improved for clarity or consistency? Should comments be added? Should comments be removed as unhelpful or extraneous?
- Could the code easily be rewritten to run much more efficiently for relevant settings?
- Is the code backwards compatible with previous versions? (or is a deprecation cycle necessary?)
- Will the new code add any dependencies on other libraries? (this is unlikely to be accepted)
- Does the documentation render properly (see the *Documentation* section for more details), and are the plots instructive?

Standard replies for reviewing includes some frequent comments that reviewers may make.

8.1.9 Reading the existing code base

Reading and digesting an existing code base is always a difficult exercise that takes time and experience to master. Even though we try to write simple code in general, understanding the code can seem overwhelming at first, given the sheer size of the project. Here is a list of tips that may help make this task easier and faster (in no particular order).

- Get acquainted with the *APIs of scikit-learn objects*: understand what *fit*, *predict*, *transform*, etc. are used for.
- Before diving into reading the code of a function / class, go through the docstrings first and try to get an idea of what each parameter / attribute is doing. It may also help to stop a minute and think *how would I do this myself if I had to?*
- The trickiest thing is often to identify which portions of the code are relevant, and which are not. In scikit-learn **a lot** of input checking is performed, especially at the beginning of the *fit* methods. Sometimes, only a very small portion of the code is doing the actual job. For example looking at the `fit()` method of *`sklearn.linear_model.LinearRegression`*, what you're looking for might just be the call the `scipy.linalg.lstsq`, but it is buried into multiple lines of input checking and the handling of different kinds of parameters.
- Due to the use of Inheritance, some methods may be implemented in parent classes. All estimators inherit at least from *`BaseEstimator`*, and from a `Mixin` class (e.g. *`ClassifierMixin`*) that enables default behaviour depending on the nature of the estimator (classifier, regressor, transformer, etc.).
- Sometimes, reading the tests for a given function will give you an idea of what its intended purpose is. You can use `git grep` (see below) to find all the tests written for a function. Most tests for a specific function/class are placed under the `tests/` folder of the module
- You'll often see code looking like this: `out = Parallel(...)(delayed(some_function)(param) for param in some_iterable)`. This runs

`some_function` in parallel using Joblib. `out` is then an iterable containing the values returned by `some_function` for each call.

- We use Cython to write fast code. Cython code is located in `.pyx` and `.pxd` files. Cython code has a more C-like flavor: we use pointers, perform manual memory allocation, etc. Having some minimal experience in C / C++ is pretty much mandatory here.
- Master your tools.
  - With such a big project, being efficient with your favorite editor or IDE goes a long way towards digesting the code base. Being able to quickly jump (or *peek*) to a function/class/attribute definition helps a lot. So does being able to quickly see where a given name is used in a file.
  - git also has some built-in killer features. It is often useful to understand how a file changed over time, using e.g. `git blame` (manual). This can also be done directly on GitHub. `git grep` (examples) is also extremely useful to see every occurrence of a pattern (e.g. a function call or a variable) in the code base.

8.2 Developing scikit-learn estimators

Whether you are proposing an estimator for inclusion in scikit-learn, developing a separate package compatible with scikit-learn, or implementing custom components for your own projects, this chapter details how to develop objects that safely interact with scikit-learn Pipelines and model selection tools.

8.2.1 APIs of scikit-learn objects

To have a uniform API, we try to have a common basic API for all the objects. In addition, to avoid the proliferation of framework code, we try to adopt simple conventions and limit to a minimum the number of methods an object must implement.

Elements of the scikit-learn API are described more definitively in the *Glossary of Common Terms and API Elements*.

Different objects

The main objects in scikit-learn are (one class can implement multiple interfaces):

Estimator The base object, implements a `fit` method to learn from data, either:

```
estimator = estimator.fit(data, targets)
```

or:

```
estimator = estimator.fit(data)
```

Predictor For supervised learning, or some unsupervised problems, implements:

```
prediction = predictor.predict(data)
```

Classification algorithms usually also offer a way to quantify certainty of a prediction, either using `decision_function` or `predict_proba`:

```
probability = predictor.predict_proba(data)
```

Transformer For filtering or modifying the data, in a supervised or unsupervised way, implements:

```
new_data = transformer.transform(data)
```

When fitting and transforming can be performed much more efficiently together than separately, implements:

```
new_data = transformer.fit_transform(data)
```

Model A model that can give a goodness of fit measure or a likelihood of unseen data, implements (higher is better):

```
score = model.score(data)
```

Estimators

The API has one predominant object: the estimator. A estimator is an object that fits a model based on some training data and is capable of inferring some properties on new data. It can be, for instance, a classifier or a regressor. All estimators implement the fit method:

```
estimator.fit(X, y)
```

All built-in estimators also have a `set_params` method, which sets data-independent parameters (overriding previous parameter values passed to `__init__`).

All estimators in the main scikit-learn codebase should inherit from `sklearn.base.BaseEstimator`.

Instantiation

This concerns the creation of an object. The object's `__init__` method might accept constants as arguments that determine the estimator's behavior (like the C constant in SVMs). It should not, however, take the actual training data as an argument, as this is left to the `fit()` method:

```
clf2 = SVC(C=2.3)
clf3 = SVC([[1, 2], [2, 3]], [-1, 1]) # WRONG!
```

The arguments accepted by `__init__` should all be keyword arguments with a default value. In other words, a user should be able to instantiate an estimator without passing any arguments to it. The arguments should all correspond to hyperparameters describing the model or the optimisation problem the estimator tries to solve. These initial arguments (or parameters) are always remembered by the estimator. Also note that they should not be documented under the "Attributes" section, but rather under the "Parameters" section for that estimator.

In addition, **every keyword argument accepted by `__init__` should correspond to an attribute on the instance**. Scikit-learn relies on this to find the relevant attributes to set on an estimator when doing model selection.

To summarize, an `__init__` should look like:

```
def __init__(self, param1=1, param2=2):
    self.param1 = param1
    self.param2 = param2
```

There should be no logic, not even input validation, and the parameters should not be changed. The corresponding logic should be put where the parameters are used, typically in `fit`. The following is wrong:

```
def __init__(self, param1=1, param2=2, param3=3):
    # WRONG: parameters should not be modified
    if param1 > 1:
        param2 += 1
    self.param1 = param1
    # WRONG: the object's attributes should have exactly the name of
    # the argument in the constructor
    self.param3 = param2
```

The reason for postponing the validation is that the same validation would have to be performed in `set_params`, which is used in algorithms like `GridSearchCV`.

Fitting

The next thing you will probably want to do is to estimate some parameters in the model. This is implemented in the `fit()` method.

The `fit()` method takes the training data as arguments, which can be one array in the case of unsupervised learning, or two arrays in the case of supervised learning.

Note that the model is fitted using X and y, but the object holds no reference to X and y. There are, however, some exceptions to this, as in the case of precomputed kernels where this data must be stored for use by the predict method.

Parameters	
X	array-like, shape (n_samples, n_features)
y	array, shape (n_samples,)
kwargs	optional data-dependent parameters.

`X.shape[0]` should be the same as `y.shape[0]`. If this requisite is not met, an exception of type `ValueError` should be raised.

`y` might be ignored in the case of unsupervised learning. However, to make it possible to use the estimator as part of a pipeline that can mix both supervised and unsupervised transformers, even unsupervised estimators need to accept a `y=None` keyword argument in the second position that is just ignored by the estimator. For the same reason, `fit_predict`, `fit_transform`, `score` and `partial_fit` methods need to accept a `y` argument in the second place if they are implemented.

The method should return the object (`self`). This pattern is useful to be able to implement quick one liners in an IPython session such as:

```
y_predicted = SVC(C=100).fit(X_train, y_train).predict(X_test)
```

Depending on the nature of the algorithm, `fit` can sometimes also accept additional keywords arguments. However, any parameter that can have a value assigned prior to having access to the data should be an `__init__` keyword argument. **fit parameters should be restricted to directly data dependent variables**. For instance a Gram matrix or an affinity matrix which are precomputed from the data matrix `X` are data dependent. A tolerance stopping criterion `tol` is not directly data dependent (although the optimal value according to some scoring function probably is).

When `fit` is called, any previous call to `fit` should be ignored. In general, calling `estimator.fit(X1)` and then `estimator.fit(X2)` should be the same as only calling `estimator.fit(X2)`. However, this may not be true in practice when `fit` depends on some random process, see *random_state*. Another exception to this rule is when the hyper-parameter `warm_start` is set to `True` for estimators that support it. `warm_start=True` means that the previous state of the trainable parameters of the estimator are reused instead of using the default initialization strategy.

Estimated Attributes

Attributes that have been estimated from the data must always have a name ending with trailing underscore, for example the coefficients of some regression estimator would be stored in a `coef_` attribute after `fit` has been called.

The estimated attributes are expected to be overridden when you call `fit` a second time.

Optional Arguments

In iterative algorithms, the number of iterations should be specified by an integer called `n_iter`.

Pairwise Attributes

An estimator that accept `X` of shape `(n_samples, n_samples)` and defines a *_pairwise* property equal to `True` allows for cross-validation of the dataset, e.g. when `X` is a precomputed kernel matrix. Specifically, the *_pairwise* property is used by `utils.metaestimators._safe_split` to slice rows and columns.

8.2.2 Rolling your own estimator

If you want to implement a new estimator that is scikit-learn-compatible, whether it is just for you or for contributing it to scikit-learn, there are several internals of scikit-learn that you should be aware of in addition to the scikit-learn API outlined above. You can check whether your estimator adheres to the scikit-learn interface and standards by running *utils.estimator_checks.check_estimator* on the class:

```
>>> from sklearn.utils.estimator_checks import check_estimator
>>> from sklearn.svm import LinearSVC
>>> check_estimator(LinearSVC)  # passes
```

The main motivation to make a class compatible to the scikit-learn estimator interface might be that you want to use it together with model evaluation and selection tools such as *model_selection.GridSearchCV* and *pipeline.Pipeline*.

Setting `generate_only=True` returns a generator that yields (estimator, check) tuples where the check can be called independently from each other, i.e. `check(estimator)`. This allows all checks to be run independently and report the checks that are failing. scikit-learn provides a pytest specific decorator, `parametrize_with_checks`, making it easier to test multiple estimators:

```
from sklearn.utils.estimator_checks import parametrize_with_checks
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeRegressor

@parametrize_with_checks([LogisticRegression, DecisionTreeRegressor])
def test_sklearn_compatible_estimator(estimator, check):
    check(estimator)
```

This decorator sets the `id` keyword in `pytest.mark.parameterize` exposing the name of the underlying estimator and check in the test name. This allows `pytest -k` to be used to specify which tests to run.

Before detailing the required interface below, we describe two ways to achieve the correct interface more easily.

Project template:

We provide a project template which helps in the creation of Python packages containing scikit-learn compatible estimators. It provides:

- an initial git repository with Python package directory structure
- a template of a scikit-learn estimator
- an initial test suite including use of `check_estimator`
- directory structures and scripts to compile documentation and example galleries
- scripts to manage continuous integration (testing on Linux and Windows)
- instructions from getting started to publishing on PyPi

`BaseEstimator` and mixins:

We tend to use "duck typing", so building an estimator which follows the API suffices for compatibility, without needing to inherit from or even import any scikit-learn classes.

However, if a dependency on scikit-learn is acceptable in your code, you can prevent a lot of boilerplate code by deriving a class from `BaseEstimator` and optionally the mixin classes in `sklearn.base`. For example, below is a custom classifier, with more examples included in the scikit-learn-contrib project template.

```
>>> import numpy as np
>>> from sklearn.base import BaseEstimator, ClassifierMixin
>>> from sklearn.utils.validation import check_X_y, check_array, check_is_fitted
>>> from sklearn.utils.multiclass import unique_labels
>>> from sklearn.metrics import euclidean_distances
>>> class TemplateClassifier(BaseEstimator, ClassifierMixin):
...
...     def __init__(self, demo_param='demo'):
...         self.demo_param = demo_param
...
...     def fit(self, X, y):
...
...         # Check that X and y have correct shape
...         X, y = check_X_y(X, y)
...         # Store the classes seen during fit
...         self.classes_ = unique_labels(y)
...
...         self.X_ = X
...         self.y_ = y
...         # Return the classifier
...         return self
...
...     def predict(self, X):
...
...         # Check is fit had been called
...         check_is_fitted(self)
...
...         # Input validation
...         X = check_array(X)
...
...         closest = np.argmin(euclidean_distances(X, self.X_), axis=1)
...         return self.y_[closest]
```

get_params and set_params

All scikit-learn estimators have `get_params` and `set_params` functions. The `get_params` function takes no arguments and returns a dict of the `__init__` parameters of the estimator, together with their values. It must take one keyword argument, `deep`, which receives a boolean value that determines whether the method should return the parameters of sub-estimators (for most estimators, this can be ignored). The default value for `deep` should be true.

The `set_params` on the other hand takes as input a dict of the form `'parameter': value` and sets the parameter of the estimator using this dict. Return value must be estimator itself.

While the `get_params` mechanism is not essential (see *Cloning* below), the `set_params` function is necessary as it is used to set parameters during grid searches.

The easiest way to implement these functions, and to get a sensible `__repr__` method, is to inherit from `sklearn.base.BaseEstimator`. If you do not want to make your code dependent on scikit-learn, the easiest way to implement the interface is:

```
def get_params(self, deep=True):
    # suppose this estimator has parameters "alpha" and "recursive"
    return {"alpha": self.alpha, "recursive": self.recursive}

def set_params(self, **parameters):
    for parameter, value in parameters.items():
        setattr(self, parameter, value)
    return self
```

Parameters and init

As *model_selection.GridSearchCV* uses `set_params` to apply parameter setting to estimators, it is essential that calling `set_params` has the same effect as setting parameters using the `__init__` method. The easiest and recommended way to accomplish this is to **not do any parameter validation in `__init__`**. All logic behind estimator parameters, like translating string arguments into functions, should be done in `fit`.

Also it is expected that parameters with trailing _ are **not to be set inside the `__init__` method**. All and only the public attributes set by fit have a trailing _. As a result the existence of parameters with trailing _ is used to check if the estimator has been fitted.

Cloning

For use with the `model_selection` module, an estimator must support the `base.clone` function to replicate an estimator. This can be done by providing a `get_params` method. If `get_params` is present, then `clone(estimator)` will be an instance of `type(estimator)` on which `set_params` has been called with clones of the result of `estimator.get_params()`.

Objects that do not provide this method will be deep-copied (using the Python standard function `copy.deepcopy`) if `safe=False` is passed to `clone`.

Pipeline compatibility

For an estimator to be usable together with `pipeline.Pipeline` in any but the last step, it needs to provide a `fit` or `fit_transform` function. To be able to evaluate the pipeline on any data but the training set, it also needs to provide a `transform` function. There are no special requirements for the last step in a pipeline, except that it has a `fit` function. All `fit` and `fit_transform` functions must take arguments `X, y`, even if y is not used. Similarly, for `score` to be usable, the last step of the pipeline needs to have a `score` function that accepts an optional `y`.

Estimator types

Some common functionality depends on the kind of estimator passed. For example, cross-validation in *model_selection.GridSearchCV* and *model_selection.cross_val_score* defaults to being stratified when used on a classifier, but not otherwise. Similarly, scorers for average precision that take a continuous prediction need to call `decision_function` for classifiers, but `predict` for regressors. This distinction between classifiers and regressors is implemented using the `_estimator_type` attribute, which takes a string value. It should be `"classifier"` for classifiers and `"regressor"` for regressors and `"clusterer"` for clustering methods, to work as expected. Inheriting from `ClassifierMixin`, `RegressorMixin` or `ClusterMixin` will set the attribute automatically. When a meta-estimator needs to distinguish among estimator types, instead of checking `_estimator_type` directly, helpers like *base.is_classifier* should be used.

Specific models

Classifiers should accept `y` (target) arguments to `fit` that are sequences (lists, arrays) of either strings or integers. They should not assume that the class labels are a contiguous range of integers; instead, they should store a list of classes in a `classes_` attribute or property. The order of class labels in this attribute should match the order in which `predict_proba`, `predict_log_proba` and `decision_function` return their values. The easiest way to achieve this is to put:

```
self.classes_, y = np.unique(y, return_inverse=True)
```

in `fit`. This returns a new `y` that contains class indexes, rather than labels, in the range [0, `n_classes`).

A classifier's `predict` method should return arrays containing class labels from `classes_`. In a classifier that implements `decision_function`, this can be achieved with:

```
def predict(self, X):
    D = self.decision_function(X)
    return self.classes_[np.argmax(D, axis=1)]
```

In linear models, coefficients are stored in an array called `coef_`, and the independent term is stored in `intercept_`. `sklearn.linear_model._base` contains a few base classes and mixins that implement common linear model patterns.

The `sklearn.utils.multiclass` module contains useful functions for working with multiclass and multilabel problems.

Estimator Tags

Warning: The estimator tags are experimental and the API is subject to change.

Scikit-learn introduced estimator tags in version 0.21. These are annotations of estimators that allow programmatic inspection of their capabilities, such as sparse matrix support, supported output types and supported methods. The estimator tags are a dictionary returned by the method `_get_tags()`. These tags are used by the common tests and the *sklearn.utils.estimator_checks.check_estimator* function to decide what tests to run and what input data is appropriate. Tags can depend on estimator parameters or even system architecture and can in general only be determined at runtime.

The default value of all tags except for `X_types` and `requires_fit` is `False`. These are defined in the `BaseEstimator` class.

The current set of estimator tags are:

non_deterministic whether the estimator is not deterministic given a fixed `random_state`

requires_positive_X whether the estimator requires positive X.

requires_positive_y whether the estimator requires a positive y (only applicable for regression).

no_validation whether the estimator skips input-validation. This is only meant for stateless and dummy transformers!

multioutput - unused for now whether a regressor supports multi-target outputs or a classifier supports multi-class multi-output.

multilabel whether the estimator supports multilabel output

stateless whether the estimator needs access to data for fitting. Even though an estimator is stateless, it might still need a call to `fit` for initialization.

requires_fit whether the estimator requires to be fitted before calling one of `transform`, `predict`, `predict_proba`, or `decision_function`.

allow_nan whether the estimator supports data with missing values encoded as np.NaN

poor_score whether the estimator fails to provide a "reasonable" test-set score, which currently for regression is an R2 of 0.5 on a subset of the boston housing dataset, and for classification an accuracy of 0.83 on `make_blobs(n_samples=300, random_state=0)`. These datasets and values are based on current estimators in sklearn and might be replaced by something more systematic.

multioutput_only whether estimator supports only multi-output classification or regression.

binary_only whether estimator supports binary classification but lacks multi-class classification support.

_skip_test whether to skip common tests entirely. Don't use this unless you have a *very good* reason.

X_types Supported input types for X as list of strings. Tests are currently only run if '2darray' is contained in the list, signifying that the estimator takes continuous 2d numpy arrays as input. The default value is ['2darray']. Other possible types are `'string'`, `'sparse'`, `'categorical'`, `dict`, `'1dlabels'` and `'2dlabels'`. The goal is that in the future the supported input type will determine the data used during testing, in particular for `'string'`, `'sparse'` and `'categorical'` data. For now, the test for sparse data do not make use of the `'sparse'` tag.

To override the tags of a child class, one must define the `_more_tags()` method and return a dict with the desired tags, e.g:

```
class MyMultiOutputEstimator(BaseEstimator):

    def _more_tags(self):
        return {'multioutput_only': True,
                'non_deterministic': True}
```

In addition to the tags, estimators also need to declare any non-optional parameters to `__init__` in the `_required_parameters` class attribute, which is a list or tuple. If `_required_parameters` is only `["estimator"]` or `["base_estimator"]`, then the estimator will be instantiated with an instance of `LinearDiscriminantAnalysis` (or `RidgeRegression` if the estimator is a regressor) in the tests. The choice of these two models is somewhat idiosyncratic but both should provide robust closed-form solutions.

8.2.3 Coding guidelines

The following are some guidelines on how new code should be written for inclusion in scikit-learn, and which may be appropriate to adopt in external projects. Of course, there are special cases and there will be exceptions to these rules. However, following these rules when submitting new code makes the review easier so new code can be integrated in less time.

Uniformly formatted code makes it easier to share code ownership. The scikit-learn project tries to closely follow the official Python guidelines detailed in PEP8 that detail how code should be formatted and indented. Please read it and follow it.

In addition, we add the following guidelines:

- Use underscores to separate words in non class names: `n_samples` rather than `nsamples`.
- Avoid multiple statements on one line. Prefer a line return after a control flow statement (`if`/`for`).
- Use relative imports for references inside scikit-learn.
- Unit tests are an exception to the previous rule; they should use absolute imports, exactly as client code would. A corollary is that, if `sklearn.foo` exports a class or function that is implemented in `sklearn.foo.bar.baz`, the test should import it from `sklearn.foo`.
- **Please don't use** `import *` **in any case**. It is considered harmful by the official Python recommendations. It makes the code harder to read as the origin of symbols is no longer explicitly referenced, but most important, it prevents using a static analysis tool like pyflakes to automatically find bugs in scikit-learn.
- Use the numpy docstring standard in all your docstrings.

A good example of code that we like can be found here.

Input validation

The module *`sklearn.utils`* contains various functions for doing input validation and conversion. Sometimes, `np.asarray` suffices for validation; do *not* use `np.asanyarray` or `np.atleast_2d`, since those let NumPy's `np.matrix` through, which has a different API (e.g., `*` means dot product on `np.matrix`, but Hadamard product on `np.ndarray`).

In other cases, be sure to call *`check_array`* on any array-like argument passed to a scikit-learn API function. The exact parameters to use depends mainly on whether and which `scipy.sparse` matrices must be accepted.

For more information, refer to the *Utilities for Developers* page.

Random Numbers

If your code depends on a random number generator, do not use `numpy.random.random()` or similar routines. To ensure repeatability in error checking, the routine should accept a keyword `random_state` and use this to construct a `numpy.random.RandomState` object. See *`sklearn.utils.check_random_state`* in *Utilities for Developers*.

Here's a simple example of code using some of the above guidelines:

```
from sklearn.utils import check_array, check_random_state

def choose_random_sample(X, random_state=0):
    """
    Choose a random point from X

    Parameters
    ----------
    X : array-like, shape (n_samples, n_features)
        array representing the data
    random_state : RandomState or an int seed (0 by default)
        A random number generator instance to define the state of the
        random permutations generator.
```

(continues on next page)

(continued from previous page)

```
    Returns
    -------
    x : numpy array, shape (n_features,)
        A random point selected from X
    """
    X = check_array(X)
    random_state = check_random_state(random_state)
    i = random_state.randint(X.shape[0])
    return X[i]
```

If you use randomness in an estimator instead of a freestanding function, some additional guidelines apply.

First off, the estimator should take a `random_state` argument to its `__init__` with a default value of `None`. It should store that argument's value, **unmodified**, in an attribute `random_state`. `fit` can call `check_random_state` on that attribute to get an actual random number generator. If, for some reason, randomness is needed after `fit`, the RNG should be stored in an attribute `random_state_`. The following example should make this clear:

```
class GaussianNoise(BaseEstimator, TransformerMixin):
    """This estimator ignores its input and returns random Gaussian noise.

    It also does not adhere to all scikit-learn conventions,
    but showcases how to handle randomness.
    """

    def __init__(self, n_components=100, random_state=None):
        self.random_state = random_state

    # the arguments are ignored anyway, so we make them optional
    def fit(self, X=None, y=None):
        self.random_state_ = check_random_state(self.random_state)

    def transform(self, X):
        n_samples = X.shape[0]
        return self.random_state_.randn(n_samples, n_components)
```

The reason for this setup is reproducibility: when an estimator is `fit` twice to the same data, it should produce an identical model both times, hence the validation in `fit`, not `__init__`.

8.3 Developers' Tips and Tricks

8.3.1 Productivity and sanity-preserving tips

In this section we gather some useful advice and tools that may increase your quality-of-life when reviewing pull requests, running unit tests, and so forth. Some of these tricks consist of userscripts that require a browser extension such as TamperMonkey or GreaseMonkey; to set up userscripts you must have one of these extensions installed, enabled and running. We provide userscripts as GitHub gists; to install them, click on the "Raw" button on the gist page.

Viewing the rendered HTML documentation for a pull request

We use CircleCI to build the HTML documentation for every pull request. To access that documentation, instructions are provided in the *documentation section of the contributor guide*. To save you a few clicks, we provide a userscript

that adds a button to every PR. After installing the userscript, navigate to any GitHub PR; a new button labeled "See CircleCI doc for this PR" should appear in the top-right area.

Folding and unfolding outdated diffs on pull requests

GitHub hides discussions on PRs when the corresponding lines of code have been changed in the mean while. This userscript provides a shortcut (Control-Alt-P at the time of writing but look at the code to be sure) to unfold all such hidden discussions at once, so you can catch up.

Checking out pull requests as remote-tracking branches

In your local fork, add to your `.git/config`, under the `[remote "upstream"]` heading, the line:

```
fetch = +refs/pull/*/head:refs/remotes/upstream/pr/*
```

You may then use `git checkout pr/PR_NUMBER` to navigate to the code of the pull-request with the given number. (Read more in this gist.)

Display code coverage in pull requests

To overlay the code coverage reports generated by the CodeCov continuous integration, consider this browser extension. The coverage of each line will be displayed as a color background behind the line number.

Useful pytest aliases and flags

The full test suite takes fairly long to run. For faster iterations, it is possibly to select a subset of tests using pytest selectors. In particular, one can run a single test based on its node ID:

```
pytest -v sklearn/linear_model/tests/test_logistic.py::test_sparsify
```

or use the -k pytest parameter to select tests based on their name. For instance,:

```
pytest sklearn/tests/test_common.py -v -k LogisticRegression
```

will run all *common tests* for the `LogisticRegression` estimator.

When a unit test fails, the following tricks can make debugging easier:

1. The command line argument `pytest -l` instructs pytest to print the local variables when a failure occurs.
2. The argument `pytest --pdb` drops into the Python debugger on failure. To instead drop into the rich IPython debugger `ipdb`, you may set up a shell alias to:

   ```
   pytest --pdbcls=IPython.terminal.debugger:TerminalPdb --capture no
   ```

Other `pytest` options that may become useful include:

- `-x` which exits on the first failed test
- `--lf` to rerun the tests that failed on the previous run
- `--ff` to rerun all previous tests, running the ones that failed first
- `-s` so that pytest does not capture the output of `print()` statements
- `--tb=short` or `--tb=line` to control the length of the logs

Since our continuous integration tests will error if `FutureWarning` isn't properly caught, it is also recommended to run `pytest` along with the `-Werror::FutureWarning` flag.

Standard replies for reviewing

It may be helpful to store some of these in GitHub's saved replies for reviewing:

Issue: Usage questions

```
You're asking a usage question. The issue tracker is mainly for bugs and new␣
↪features. For usage questions, it is recommended to try [Stack Overflow](https:/
↪/stackoverflow.com/questions/tagged/scikit-learn) or [the Mailing List](https://
↪mail.python.org/mailman/listinfo/scikit-learn).
```

Issue: You're welcome to update the docs

```
Please feel free to offer a pull request updating the documentation if you feel␣
↪it could be improved.
```

Issue: Self-contained example for bug

```
Please provide [self-contained example code](https://stackoverflow.com/help/mcve),
↪ including imports and data (if possible), so that other contributors can just␣
↪run it and reproduce your issue. Ideally your example code should be minimal.
```

Issue: Software versions

```
To help diagnose your issue, please paste the output of:
```py
import sklearn; sklearn.show_versions()
```
Thanks.
```

Issue: Code blocks

```
Readability can be greatly improved if you [format](https://help.github.com/
↪articles/creating-and-highlighting-code-blocks/) your code snippets and␣
↪complete error messages appropriately. For example:

    ```python
    print(something)
    ```
generates:
```python
print(something)
```
And:

    ```pytb
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    ImportError: No module named 'hello'
    ```
generates:
```pytb
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(continues on next page)
```

(continued from previous page)

```
ImportError: No module named 'hello'
```
You can edit your issue descriptions and comments at any time to improve␣
↪readability. This helps maintainers a lot. Thanks!
```

**Issue/Comment: Linking to code**

```
Friendly advice: for clarity's sake, you can link to code like [this](https://
↪help.github.com/articles/creating-a-permanent-link-to-a-code-snippet/).
```

Issue/Comment: Linking to comments

```
Please use links to comments, which make it a lot easier to see what you are␣
↪referring to, rather than just linking to the issue. See [this](https://
↪stackoverflow.com/questions/25163598/how-do-i-reference-a-specific-issue-
↪comment-on-github) for more details.
```

PR-NEW: Better description and title

```
Thanks for the pull request! Please make the title of the PR more descriptive.␣
↪The title will become the commit message when this is merged. You should state␣
↪what issue (or PR) it fixes/resolves in the description using the syntax␣
↪described [here](http://scikit-learn.org/dev/developers/contributing.html
↪#contributing-pull-requests).
```

PR-NEW: Fix #

```
Please use "Fix #issueNumber" in your PR description (and you can do it more than␣
↪once). This way the associated issue gets closed automatically when the PR is␣
↪merged. For more details, look at [this](https://github.com/blog/1506-closing-
↪issues-via-pull-requests).
```

PR-NEW or Issue: Maintenance cost

```
Every feature we include has a [maintenance cost](http://scikit-learn.org/dev/faq.
↪html#why-are-you-so-selective-on-what-algorithms-you-include-in-scikit-learn).␣
↪Our maintainers are mostly volunteers. For a new feature to be included, we␣
↪need evidence that it is often useful and, ideally, [well-established](http://
↪scikit-learn.org/dev/faq.html#what-are-the-inclusion-criteria-for-new-
↪algorithms) in the literature or in practice. That doesn't stop you␣
↪implementing it for yourself and publishing it in a separate repository, or␣
↪even [scikit-learn-contrib](https://scikit-learn-contrib.github.io).
```

PR-WIP: What's needed before merge?

```
Please clarify (perhaps as a TODO list in the PR description) what work you␣
↪believe still needs to be done before it can be reviewed for merge. When it is␣
↪ready, please prefix the PR title with `[MRG]`.
```

PR-WIP: Regression test needed

```
Please add a [non-regression test](https://en.wikipedia.org/wiki/Non-regression_
↪testing) that would fail at master but pass in this PR.
```

PR-WIP: PEP8
```

```
You have some [PEP8](https://www.python.org/dev/peps/pep-0008/) violations, whose␣
↪details you can see in the Circle CI `lint` job. It might be worth configuring␣
↪your code editor to check for such errors on the fly, so you can catch them␣
↪before committing.
```

PR-MRG: Patience

```
Before merging, we generally require two core developers to agree that your pull␣
↪request is desirable and ready. [Please be patient](http://scikit-learn.org/dev/
↪faq.html#why-is-my-pull-request-not-getting-any-attention), as we mostly rely␣
↪on volunteered time from busy core developers. (You are also welcome to help us␣
↪out with [reviewing other PRs](http://scikit-learn.org/dev/developers/
↪contributing.html#code-review-guidelines).)
```

PR-MRG: Add to what's new

```
Please add an entry to the change log at `doc/whats_new/v*.rst`. Like the other␣
↪entries there, please reference this pull request with `:pr:` and credit␣
↪yourself (and other contributors if applicable) with `:user:`.
```

PR: Don't change unrelated

```
Please do not change unrelated lines. It makes your contribution harder to review␣
↪and may introduce merge conflicts to other pull requests.
```

8.3.2 Debugging memory errors in Cython with valgrind

While python/numpy's built-in memory management is relatively robust, it can lead to performance penalties for some routines. For this reason, much of the high-performance code in scikit-learn in written in cython. This performance gain comes with a tradeoff, however: it is very easy for memory bugs to crop up in cython code, especially in situations where that code relies heavily on pointer arithmetic.

Memory errors can manifest themselves a number of ways. The easiest ones to debug are often segmentation faults and related glibc errors. Uninitialized variables can lead to unexpected behavior that is difficult to track down. A very useful tool when debugging these sorts of errors is valgrind.

Valgrind is a command-line tool that can trace memory errors in a variety of code. Follow these steps:

1. Install valgrind on your system.
2. Download the python valgrind suppression file: valgrind-python.supp.
3. Follow the directions in the README.valgrind file to customize your python suppressions. If you don't, you will have spurious output coming related to the python interpreter instead of your own code.
4. Run valgrind as follows:

```
$> valgrind -v --suppressions=valgrind-python.supp python my_test_script.py
```

The result will be a list of all the memory-related errors, which reference lines in the C-code generated by cython from your .pyx file. If you examine the referenced lines in the .c file, you will see comments which indicate the corresponding location in your .pyx source file. Hopefully the output will give you clues as to the source of your memory error.

For more information on valgrind and the array of options it has, see the tutorials and documentation on the valgrind web site.

8.4 Utilities for Developers

Scikit-learn contains a number of utilities to help with development. These are located in *sklearn.utils*, and include tools in a number of categories. All the following functions and classes are in the module *sklearn.utils*.

Warning: These utilities are meant to be used internally within the scikit-learn package. They are not guaranteed to be stable between versions of scikit-learn. Backports, in particular, will be removed as the scikit-learn dependencies evolve.

8.4.1 Validation Tools

These are tools used to check and validate input. When you write a function which accepts arrays, matrices, or sparse matrices as arguments, the following should be used when applicable.

- *assert_all_finite*: Throw an error if array contains NaNs or Infs.
- *as_float_array*: convert input to an array of floats. If a sparse matrix is passed, a sparse matrix will be returned.
- *check_array*: check that input is a 2D array, raise error on sparse matrices. Allowed sparse matrix formats can be given optionally, as well as allowing 1D or N-dimensional arrays. Calls *assert_all_finite* by default.
- *check_X_y*: check that X and y have consistent length, calls check_array on X, and column_or_1d on y. For multilabel classification or multitarget regression, specify multi_output=True, in which case check_array will be called on y.
- *indexable*: check that all input arrays have consistent length and can be sliced or indexed using safe_index. This is used to validate input for cross-validation.
- *validation.check_memory* checks that input is `joblib.Memory`-like, which means that it can be converted into a `sklearn.utils.Memory` instance (typically a str denoting the `cachedir`) or has the same interface.

If your code relies on a random number generator, it should never use functions like `numpy.random.random` or `numpy.random.normal`. This approach can lead to repeatability issues in unit tests. Instead, a `numpy.random.RandomState` object should be used, which is built from a `random_state` argument passed to the class or function. The function *check_random_state*, below, can then be used to create a random number generator object.

- *check_random_state*: create a `np.random.RandomState` object from a parameter `random_state`.
  - If `random_state` is `None` or `np.random`, then a randomly-initialized `RandomState` object is returned.
  - If `random_state` is an integer, then it is used to seed a new `RandomState` object.
  - If `random_state` is a `RandomState` object, then it is passed through.

For example:

```
>>> from sklearn.utils import check_random_state
>>> random_state = 0
>>> random_state = check_random_state(random_state)
>>> random_state.rand(4)
array([0.5488135 , 0.71518937, 0.60276338, 0.54488318])
```

When developing your own scikit-learn compatible estimator, the following helpers are available.

- *validation.check_is_fitted*: check that the estimator has been fitted before calling `transform`, `predict`, or similar methods. This helper allows to raise a standardized error message across estimator.
- *validation.has_fit_parameter*: check that a given parameter is supported in the `fit` method of a given estimator.

8.4.2 Efficient Linear Algebra & Array Operations

- *extmath.randomized_range_finder*: construct an orthonormal matrix whose range approximates the range of the input. This is used in *extmath.randomized_svd*, below.
- *extmath.randomized_svd*: compute the k-truncated randomized SVD. This algorithm finds the exact truncated singular values decomposition using randomization to speed up the computations. It is particularly fast on large matrices on which you wish to extract only a small number of components.
- `arrayfuncs.cholesky_delete`: (used in *sklearn.linear_model.lars_path*) Remove an item from a cholesky factorization.
- *arrayfuncs.min_pos*: (used in `sklearn.linear_model.least_angle`) Find the minimum of the positive values within an array.
- *extmath.fast_logdet*: efficiently compute the log of the determinant of a matrix.
- *extmath.density*: efficiently compute the density of a sparse vector
- *extmath.safe_sparse_dot*: dot product which will correctly handle `scipy.sparse` inputs. If the inputs are dense, it is equivalent to `numpy.dot`.
- *extmath.weighted_mode*: an extension of `scipy.stats.mode` which allows each item to have a real-valued weight.
- *resample*: Resample arrays or sparse matrices in a consistent way. used in *shuffle*, below.
- *shuffle*: Shuffle arrays or sparse matrices in a consistent way. Used in *sklearn.cluster.k_means*.

8.4.3 Efficient Random Sampling

- *random.sample_without_replacement*: implements efficient algorithms for sampling `n_samples` integers from a population of size `n_population` without replacement.

8.4.4 Efficient Routines for Sparse Matrices

The `sklearn.utils.sparsefuncs` cython module hosts compiled extensions to efficiently process `scipy.sparse` data.

- *sparsefuncs.mean_variance_axis*: compute the means and variances along a specified axis of a CSR matrix. Used for normalizing the tolerance stopping criterion in *sklearn.cluster.KMeans*.
- *sparsefuncs_fast.inplace_csr_row_normalize_l1* and *sparsefuncs_fast.inplace_csr_row_normalize_l2*: can be used to normalize individual sparse samples to unit L1 or L2 norm as done in *sklearn.preprocessing.Normalizer*.
- *sparsefuncs.inplace_csr_column_scale*: can be used to multiply the columns of a CSR matrix by a constant scale (one scale per column). Used for scaling features to unit standard deviation in *sklearn.preprocessing.StandardScaler*.

8.4.5 Graph Routines

- *graph.single_source_shortest_path_length*: (not currently used in scikit-learn) Return the shortest path from a single source to all connected nodes on a graph. Code is adapted from networkx. If this is ever needed again, it would be far faster to use a single iteration of Dijkstra's algorithm from graph_shortest_path.
- *graph_shortest_path.graph_shortest_path*: (used in *sklearn.manifold.Isomap*) Return the shortest path between all pairs of connected points on a directed or undirected graph. Both the Floyd-Warshall algorithm and Dijkstra's algorithm are available. The algorithm is most efficient when the connectivity matrix is a scipy.sparse.csr_matrix.

8.4.6 Testing Functions

- *all_estimators* : returns a list of all estimators in scikit-learn to test for consistent behavior and interfaces.

8.4.7 Multiclass and multilabel utility function

- *multiclass.is_multilabel*: Helper function to check if the task is a multi-label classification one.
- *multiclass.unique_labels*: Helper function to extract an ordered array of unique labels from different formats of target.

8.4.8 Helper Functions

- *gen_even_slices*: generator to create n-packs of slices going up to n. Used in *sklearn.decomposition.dict_learning* and *sklearn.cluster.k_means*.
- *safe_mask*: Helper function to convert a mask to the format expected by the numpy array or scipy sparse matrix on which to use it (sparse matrices support integer indices only while numpy arrays support both boolean masks and integer indices).
- *safe_sqr*: Helper function for unified squaring (**2) of array-likes, matrices and sparse matrices.

8.4.9 Hash Functions

- *murmurhash3_32* provides a python wrapper for the MurmurHash3_x86_32 C++ non cryptographic hash function. This hash function is suitable for implementing lookup tables, Bloom filters, Count Min Sketch, feature hashing and implicitly defined sparse random projections:

```
>>> from sklearn.utils import murmurhash3_32
>>> murmurhash3_32("some feature", seed=0) == -384616559
True

>>> murmurhash3_32("some feature", seed=0, positive=True) == 3910350737
True
```

The sklearn.utils.murmurhash module can also be "cimported" from other cython modules so as to benefit from the high performance of MurmurHash while skipping the overhead of the Python interpreter.

8.4.10 Warnings and Exceptions

- *deprecated*: Decorator to mark a function or class as deprecated.
- *sklearn.exceptions.ConvergenceWarning*: Custom warning to catch convergence problems. Used in `sklearn.covariance.graphical_lasso`.

8.5 How to optimize for speed

The following gives some practical guidelines to help you write efficient code for the scikit-learn project.

Note: While it is always useful to profile your code so as to **check performance assumptions**, it is also highly recommended to **review the literature** to ensure that the implemented algorithm is the state of the art for the task before investing into costly implementation optimization.

Times and times, hours of efforts invested in optimizing complicated implementation details have been rendered irrelevant by the subsequent discovery of simple **algorithmic tricks**, or by using another algorithm altogether that is better suited to the problem.

The section *A simple algorithmic trick: warm restarts* gives an example of such a trick.

8.5.1 Python, Cython or C/C++?

In general, the scikit-learn project emphasizes the **readability** of the source code to make it easy for the project users to dive into the source code so as to understand how the algorithm behaves on their data but also for ease of maintainability (by the developers).

When implementing a new algorithm is thus recommended to **start implementing it in Python using Numpy and Scipy** by taking care of avoiding looping code using the vectorized idioms of those libraries. In practice this means trying to **replace any nested for loops by calls to equivalent Numpy array methods**. The goal is to avoid the CPU wasting time in the Python interpreter rather than crunching numbers to fit your statistical model. It's generally a good idea to consider NumPy and SciPy performance tips: https://scipy.github.io/old-wiki/pages/PerformanceTips

Sometimes however an algorithm cannot be expressed efficiently in simple vectorized Numpy code. In this case, the recommended strategy is the following:

1. **Profile** the Python implementation to find the main bottleneck and isolate it in a **dedicated module level function**. This function will be reimplemented as a compiled extension module.
2. If there exists a well maintained BSD or MIT **C/C++** implementation of the same algorithm that is not too big, you can write a **Cython wrapper** for it and include a copy of the source code of the library in the scikit-learn source tree: this strategy is used for the classes *svm.LinearSVC*, *svm.SVC* and *linear_model.LogisticRegression* (wrappers for liblinear and libsvm).
3. Otherwise, write an optimized version of your Python function using **Cython** directly. This strategy is used for the *linear_model.ElasticNet* and *linear_model.SGDClassifier* classes for instance.
4. **Move the Python version of the function in the tests** and use it to check that the results of the compiled extension are consistent with the gold standard, easy to debug Python version.
5. Once the code is optimized (not simple bottleneck spottable by profiling), check whether it is possible to have **coarse grained parallelism** that is amenable to **multi-processing** by using the `joblib.Parallel` class.

When using Cython, use either

$ python setup.py build_ext -i $ python setup.py install

to generate C files. You are responsible for adding .c/.cpp extensions along with build parameters in each submodule `setup.py`.

C/C++ generated files are embedded in distributed stable packages. The goal is to make it possible to install scikit-learn stable version on any machine with Python, Numpy, Scipy and C/C++ compiler.

8.5.2 Profiling Python code

In order to profile Python code we recommend to write a script that loads and prepare you data and then use the IPython integrated profiler for interactively exploring the relevant part for the code.

Suppose we want to profile the Non Negative Matrix Factorization module of scikit-learn. Let us setup a new IPython session and load the digits dataset and as in the *Recognizing hand-written digits* example:

```
In [1]: from sklearn.decomposition import NMF

In [2]: from sklearn.datasets import load_digits

In [3]: X, _ = load_digits(return_X_y=True)
```

Before starting the profiling session and engaging in tentative optimization iterations, it is important to measure the total execution time of the function we want to optimize without any kind of profiler overhead and save it somewhere for later reference:

```
In [4]: %timeit NMF(n_components=16, tol=1e-2).fit(X)
1 loops, best of 3: 1.7 s per loop
```

To have a look at the overall performance profile using the `%prun` magic command:

```
In [5]: %prun -l nmf.py NMF(n_components=16, tol=1e-2).fit(X)
         14496 function calls in 1.682 CPU seconds

   Ordered by: internal time
   List reduced from 90 to 9 due to restriction <'nmf.py'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       36    0.609    0.017    1.499    0.042 nmf.py:151(_nls_subproblem)
     1263    0.157    0.000    0.157    0.000 nmf.py:18(_pos)
        1    0.053    0.053    1.681    1.681 nmf.py:352(fit_transform)
      673    0.008    0.000    0.057    0.000 nmf.py:28(norm)
        1    0.006    0.006    0.047    0.047 nmf.py:42(_initialize_nmf)
       36    0.001    0.000    0.010    0.000 nmf.py:36(_sparseness)
       30    0.001    0.000    0.001    0.000 nmf.py:23(_neg)
        1    0.000    0.000    0.000    0.000 nmf.py:337(__init__)
        1    0.000    0.000    1.681    1.681 nmf.py:461(fit)
```

The `tottime` column is the most interesting: it gives to total time spent executing the code of a given function ignoring the time spent in executing the sub-functions. The real total time (local code + sub-function calls) is given by the `cumtime` column.

Note the use of the `-l nmf.py` that restricts the output to lines that contains the "nmf.py" string. This is useful to have a quick look at the hotspot of the nmf Python module it-self ignoring anything else.

Here is the beginning of the output of the same command without the `-l nmf.py` filter:

```
In [5] %prun NMF(n_components=16, tol=1e-2).fit(X)
         16159 function calls in 1.840 CPU seconds
```

(continues on next page)

(continued from previous page)

```
   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     2833    0.653    0.000    0.653    0.000 {numpy.core._dotblas.dot}
       46    0.651    0.014    1.636    0.036 nmf.py:151(_nls_subproblem)
     1397    0.171    0.000    0.171    0.000 nmf.py:18(_pos)
     2780    0.167    0.000    0.167    0.000 {method 'sum' of 'numpy.ndarray'␣
↪objects}
        1    0.064    0.064    1.840    1.840 nmf.py:352(fit_transform)
     1542    0.043    0.000    0.043    0.000 {method 'flatten' of 'numpy.ndarray'␣
↪objects}
      337    0.019    0.000    0.019    0.000 {method 'all' of 'numpy.ndarray'␣
↪objects}
     2734    0.011    0.000    0.181    0.000 fromnumeric.py:1185(sum)
        2    0.010    0.005    0.010    0.005 {numpy.linalg.lapack_lite.dgesdd}
      748    0.009    0.000    0.065    0.000 nmf.py:28(norm)
...
```

The above results show that the execution is largely dominated by dot products operations (delegated to blas). Hence there is probably no huge gain to expect by rewriting this code in Cython or C/C++: in this case out of the 1.7s total execution time, almost 0.7s are spent in compiled code we can consider optimal. By rewriting the rest of the Python code and assuming we could achieve a 1000% boost on this portion (which is highly unlikely given the shallowness of the Python loops), we would not gain more than a 2.4x speed-up globally.

Hence major improvements can only be achieved by **algorithmic improvements** in this particular example (e.g. trying to find operation that are both costly and useless to avoid computing then rather than trying to optimize their implementation).

It is however still interesting to check what's happening inside the `_nls_subproblem` function which is the hotspot if we only consider Python code: it takes around 100% of the accumulated time of the module. In order to better understand the profile of this specific function, let us install `line_profiler` and wire it to IPython:

```
$ pip install line_profiler
```

- **Under IPython 0.13+**, first create a configuration profile:

  ```
  $ ipython profile create
  ```

  Then register the line_profiler extension in `~/.ipython/profile_default/ipython_config.py`:

  ```
  c.TerminalIPythonApp.extensions.append('line_profiler')
  c.InteractiveShellApp.extensions.append('line_profiler')
  ```

  This will register the `%lprun` magic command in the IPython terminal application and the other frontends such as qtconsole and notebook.

Now restart IPython and let us use this new toy:

```
In [1]: from sklearn.datasets import load_digits

In [2]: from sklearn.decomposition import NMF
  ... : from sklearn.decomposition._nmf import _nls_subproblem

In [3]: X, _ = load_digits(return_X_y=True)

In [4]: %lprun -f _nls_subproblem NMF(n_components=16, tol=1e-2).fit(X)
```

(continues on next page)

(continued from previous page)

```
Timer unit: 1e-06 s

File: sklearn/decomposition/nmf.py
Function: _nls_subproblem at line 137
Total time: 1.73153 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   137                                           def _nls_subproblem(V, W, H_init,␣
↪tol, max_iter):
   138                                               """Non-negative least square␣
↪solver
   ...
   170                                               """
   171        48         5863    122.1      0.3      if (H_init < 0).any():
   172                                                   raise ValueError("Negative␣
↪values in H_init passed to NLS solver.")
   173
   174        48          139      2.9      0.0      H = H_init
   175        48       112141   2336.3      5.8      WtV = np.dot(W.T, V)
   176        48        16144    336.3      0.8      WtW = np.dot(W.T, W)
   177
   178                                               # values justified in the paper
   179        48          144      3.0      0.0      alpha = 1
   180        48          113      2.4      0.0      beta = 0.1
   181       638         1880      2.9      0.1      for n_iter in range(1, max_iter␣
↪+ 1):
   182       638       195133    305.9     10.2          grad = np.dot(WtW, H) - WtV
   183       638       495761    777.1     25.9          proj_gradient = norm(grad[np.
↪logical_or(grad < 0, H > 0)])
   184       638         2449      3.8      0.1          if proj_gradient < tol:
   185        48          130      2.7      0.0              break
   186
   187      1474         4474      3.0      0.2          for inner_iter in range(1,␣
↪20):
   188      1474        83833     56.9      4.4              Hn = H - alpha * grad
   189                                                       # Hn = np.where(Hn > 0,␣
↪Hn, 0)
   190      1474       194239    131.8     10.1              Hn = _pos(Hn)
   191      1474        48858     33.1      2.5              d = Hn - H
   192      1474       150407    102.0      7.8              gradd = np.sum(grad * d)
   193      1474       515390    349.7     26.9              dQd = np.sum(np.dot(WtW,␣
↪d) * d)
   ...
```

By looking at the top values of the `% Time` column it is really easy to pin-point the most expensive expressions that would deserve additional care.

8.5.3 Memory usage profiling

You can analyze in detail the memory usage of any Python code with the help of memory_profiler. First, install the latest version:

```
$ pip install -U memory_profiler
```

Then, setup the magics in a manner similar to `line_profiler`.

- **Under IPython 0.11+**, first create a configuration profile:

```
$ ipython profile create
```

Then register the extension in `~/.ipython/profile_default/ipython_config.py` alongside the line profiler:

```
c.TerminalIPythonApp.extensions.append('memory_profiler')
c.InteractiveShellApp.extensions.append('memory_profiler')
```

This will register the `%memit` and `%mprun` magic commands in the IPython terminal application and the other frontends such as qtconsole and notebook.

`%mprun` is useful to examine, line-by-line, the memory usage of key functions in your program. It is very similar to `%lprun`, discussed in the previous section. For example, from the `memory_profiler examples` directory:

```
In [1] from example import my_func

In [2] %mprun -f my_func my_func()
Filename: example.py

Line #    Mem usage  Increment   Line Contents
================================================
     3                           @profile
     4      5.97 MB    0.00 MB   def my_func():
     5     13.61 MB    7.64 MB       a = [1] * (10 ** 6)
     6    166.20 MB  152.59 MB       b = [2] * (2 * 10 ** 7)
     7     13.61 MB -152.59 MB       del b
     8     13.61 MB    0.00 MB       return a
```

Another useful magic that `memory_profiler` defines is `%memit`, which is analogous to `%timeit`. It can be used as follows:

```
In [1]: import numpy as np

In [2]: %memit np.zeros(1e7)
maximum of 3: 76.402344 MB per loop
```

For more details, see the docstrings of the magics, using `%memit?` and `%mprun?`.

8.5.4 Performance tips for the Cython developer

If profiling of the Python code reveals that the Python interpreter overhead is larger by one order of magnitude or more than the cost of the actual numerical computation (e.g. `for` loops over vector components, nested evaluation of conditional expression, scalar arithmetic...), it is probably adequate to extract the hotspot portion of the code as a standalone function in a `.pyx` file, add static type declarations and then use Cython to generate a C program suitable to be compiled as a Python extension module.

The official documentation available at http://docs.cython.org/ contains a tutorial and reference guide for developing such a module. In the following we will just highlight a couple of tricks that we found important in practice on the existing cython codebase in the scikit-learn project.

TODO: html report, type declarations, bound checks, division by zero checks, memory alignment, direct blas calls...

- https://www.youtube.com/watch?v=gMvkiQ-gOW8
- http://conference.scipy.org/proceedings/SciPy2009/paper_1/
- http://conference.scipy.org/proceedings/SciPy2009/paper_2/

Using OpenMP

Since scikit-learn can be built without OpenMP, it's necessary to protect each direct call to OpenMP. This can be done using the following syntax:

```
# importing OpenMP
IF SKLEARN_OPENMP_PARALLELISM_ENABLED:
    cimport openmp

# calling OpenMP
IF SKLEARN_OPENMP_PARALLELISM_ENABLED:
    max_threads = openmp.omp_get_max_threads()
ELSE:
    max_threads = 1
```

Note: Protecting the parallel loop, `prange`, is already done by cython.

8.5.5 Profiling compiled extensions

When working with compiled extensions (written in C/C++ with a wrapper or directly as Cython extension), the default Python profiler is useless: we need a dedicated tool to introspect what's happening inside the compiled extension itself.

Using yep and gperftools

Easy profiling without special compilation options use yep:

- https://pypi.org/project/yep/
- http://fa.bianp.net/blog/2011/a-profiler-for-python-extensions

Using gprof

In order to profile compiled Python extensions one could use `gprof` after having recompiled the project with `gcc -pg` and using the `python-dbg` variant of the interpreter on debian / ubuntu: however this approach requires to also have `numpy` and `scipy` recompiled with `-pg` which is rather complicated to get working.

Fortunately there exist two alternative profilers that don't require you to recompile everything.

Using valgrind / callgrind / kcachegrind

kcachegrind

`yep` can be used to create a profiling report. `kcachegrind` provides a graphical environment to visualize this report:

```
# Run yep to profile some python script
python -m yep -c my_file.py

# open my_file.py.callgrin with kcachegrind
kcachegrind my_file.py.prof
```

Note: `yep` can be executed with the argument `--lines` or `-l` to compile a profiling report 'line by line'.

8.5.6 Multi-core parallelism using `joblib.Parallel`

See joblib documentation

8.5.7 A simple algorithmic trick: warm restarts

See the glossary entry for warm_start

8.6 Installing the development version of scikit-learn

This section introduces how to install the **master branch** of scikit-learn. This can be done by either installing a nightly build or building from source.

8.6.1 Installing nightly builds

The continuous integration servers of the scikit-learn project build, test and upload wheel packages for the most recent Python version on a nightly basis.

Installing a nightly build is the quickest way to:

- try a new feature that will be shipped in the next release (that is, a feature from a pull-request that was recently merged to the master branch);
- check whether a bug you encountered has been fixed since the last release.

```
pip install --pre --extra-index https://pypi.anaconda.org/scipy-wheels-nightly/simple␣
↪scikit-learn
```

8.6.2 Building from source

Building from source is required to work on a contribution (bug fix, new feature, code or documentation improvement).

1. Use Git to check out the latest source from the scikit-learn repository on Github.:

   ```
   git clone git://github.com/scikit-learn/scikit-learn.git
   cd scikit-learn
   ```

   If you plan on submitting a pull-request, you should clone from your fork instead.

2. Install a compiler with OpenMP support for your platform. See instructions for *Windows*, *macOS*, *Linux* and *FreeBSD*.

3. Optional (but recommended): create and activate a dedicated virtualenv or conda environment.

4. Install Cython and build the project with pip in *Editable mode*:

   ```
   pip install cython
   pip install --verbose --editable .
   ```

5. Check that the installed scikit-learn has a version number ending with `.dev0`:

```
python -c "import sklearn; sklearn.show_versions()"
```

6. Please refer to the *Developer's Guide* and *Useful pytest aliases and flags* to run the tests on the module of your choice.

Note: You will have to re-run the `pip install --editable .` command every time the source code of a Cython file is updated (ending in `.pyx` or `.pxd`).

Dependencies

Runtime dependencies

Scikit-learn requires the following dependencies both at build time and at runtime:

- Python (>= 3.5),
- NumPy (>= 1.11),
- SciPy (>= 0.17),
- Joblib (>= 0.11).

Those dependencies are **automatically installed by pip** if they were missing when building scikit-learn from source.

Note: For running on PyPy, PyPy3-v5.10+, Numpy 1.14.0+, and scipy 1.1.0+ are required. For PyPy, only installation instructions with pip apply.

Build dependencies

Building Scikit-learn also requires:

- Cython >= 0.28.5
- A C/C++ compiler and a matching OpenMP runtime library. See the *platform system specific instructions* for more details.

Note: If OpenMP is not supported by the compiler, the build will be done with OpenMP functionalities disabled. This is not recommended since it will force some estimators to run in sequential mode instead of leveraging thread-based parallelism. Setting the `SKLEARN_FAIL_NO_OPENMP` environment variable (before cythonization) will force the build to fail if OpenMP is not supported.

Since version 0.21, scikit-learn automatically detects and use the linear algebrea library used by SciPy **at runtime**. Scikit-learn has therefore no build dependency on BLAS/LAPACK implementations such as OpenBlas, Atlas, Blis or MKL.

Test dependencies

Running tests requires:

- pytest >=4.6.2

Some tests also require pandas.

Building a specific version from a tag

If you want to build a stable version, you can `git checkout <VERSION>` to get the code for that particular version, or download an zip archive of the version from github.

Editable mode

If you run the development version, it is cumbersome to reinstall the package each time you update the sources. Therefore it is recommended that you install in with the `pip install --editable .` command, which allows you to edit the code in-place. This builds the extension in place and creates a link to the development directory (see the pip docs).

This is fundamentally similar to using the command `python setup.py develop` (see the setuptool docs). It is however preferred to use pip.

On Unix-like systems, you can equivalently type `make in` from the top-level folder. Have a look at the `Makefile` for additional utilities.

8.6.3 Platform-specific instructions

Here are instructions to install a working C/C++ compiler with OpenMP support to build scikit-learn Cython extensions for each supported platform.

Windows

First, install Build Tools for Visual Studio 2019.

> **Warning:** You DO NOT need to install Visual Studio 2019. You only need the "Build Tools for Visual Studio 2019", under "All downloads" -> "Tools for Visual Studio 2019".

Secondly, find out if you are running 64-bit or 32-bit Python. The building command depends on the architecture of the Python interpreter. You can check the architecture by running the following in `cmd` or `powershell` console:

```
python -c "import struct; print(struct.calcsize('P') * 8)"
```

For 64-bit Python, configure the build environment with:

```
SET DISTUTILS_USE_SDK=1
"C:\Program Files (x86)\Microsoft Visual␣
↪Studio\2019\BuildTools\VC\Auxiliary\Build\vcvarsall.bat" x64
```

Replace `x64` by `x86` to build for 32-bit Python.

Please be aware that the path above might be different from user to user. The aim is to point to the "vcvarsall.bat" file that will set the necessary environment variables in the current command prompt.

Finally, build scikit-learn from this command prompt:

```
pip install --verbose --editable .
```

macOS

The default C compiler on macOS, Apple clang (confusingly aliased as `/usr/bin/gcc`), does not directly support OpenMP. We present two alternatives to enable OpenMP support:

- either install `conda-forge::compilers` with conda;
- or install `libomp` with Homebrew to extend the default Apple clang compiler.

macOS compilers from conda-forge

If you use the conda package manager (version >= 4.7), you can install the `compilers` meta-package from the conda-forge channel, which provides OpenMP-enabled C/C++ compilers based on the llvm toolchain.

First install the macOS command line tools:

```
xcode-select --install
```

It is recommended to use a dedicated conda environment to build scikit-learn from source:

```
conda create -n sklearn-dev python numpy scipy cython joblib pytest \
    conda-forge::compilers conda-forge::llvm-openmp
conda activate sklearn-dev
make clean
pip install --verbose --editable .
```

Note: If you get any conflicting dependency error message, try commenting out any custom conda configuration in the `$HOME/.condarc` file. In particular the `channel_priority: strict` directive is known to cause problems for this setup.

You can check that the custom compilers are properly installed from conda forge using the following command:

```
conda list compilers llvm-openmp
```

The compilers meta-package will automatically set custom environment variables:

```
echo $CC
echo $CXX
echo $CFLAGS
echo $CXXFLAGS
echo $LDFLAGS
```

They point to files and folders from your `sklearn-dev` conda environment (in particular in the bin/, include/ and lib/ subfolders). For instance `-L/path/to/conda/envs/sklearn-dev/lib` should appear in `LDFLAGS`.

In the log, you should see the compiled extension being built with the clang and clang++ compilers installed by conda with the `-fopenmp` command line flag.

macOS compilers from Homebrew

Another solution is to enable OpenMP support for the clang compiler shipped by default on macOS.

First install the macOS command line tools:

```
xcode-select --install
```

Install the Homebrew package manager for macOS.

Install the LLVM OpenMP library:

```
brew install libomp
```

Set the following environment variables:

```
export CC=/usr/bin/clang
export CXX=/usr/bin/clang++
export CPPFLAGS="$CPPFLAGS -Xpreprocessor -fopenmp"
export CFLAGS="$CFLAGS -I/usr/local/opt/libomp/include"
export CXXFLAGS="$CXXFLAGS -I/usr/local/opt/libomp/include"
export LDFLAGS="$LDFLAGS -Wl,-rpath,/usr/local/opt/libomp/lib -L/usr/local/opt/libomp/
↪lib -lomp"
```

Finally, build scikit-learn in verbose mode (to check for the presence of the `-fopenmp` flag in the compiler commands):

```
make clean
pip install --verbose --editable .
```

Linux

Linux compilers from the system

Installing scikit-learn from source without using conda requires you to have installed the scikit-learn Python development headers and a working C/C++ compiler with OpenMP support (typically the GCC toolchain).

Install build dependencies for Debian-based operating systems, e.g. Ubuntu:

```
sudo apt-get install build-essential python3-dev python3-pip
```

then proceed as usual:

```
pip3 install cython
pip3 install --verbose --editable .
```

Cython and the pre-compiled wheels for the runtime dependencies (numpy, scipy and joblib) should automatically be installed in `$HOME/.local/lib/pythonX.Y/site-packages`. Alternatively you can run the above commands from a virtualenv or a conda environment to get full isolation from the Python packages installed via the system packager. When using an isolated environment, `pip3` should be replaced by `pip` in the above commands.

When precompiled wheels of the runtime dependencies are not avalaible for your architecture (e.g. ARM), you can install the system versions:

```
sudo apt-get install cython3 python3-numpy python3-scipy
```

On Red Hat and clones (e.g. CentOS), install the dependencies using:

```
sudo yum -y install gcc gcc-c++ python3-devel numpy scipy
```

Linux compilers from conda-forge

Alternatively, install a recent version of the GNU C Compiler toolchain (GCC) in the user folder using conda:

```
conda create -n sklearn-dev numpy scipy joblib cython conda-forge::compilers
conda activate sklearn-dev
pip install --verbose --editable .
```

FreeBSD

The clang compiler included in FreeBSD 12.0 and 11.2 base systems does not include OpenMP support. You need to install the `openmp` library from packages (or ports):

```
sudo pkg install openmp
```

This will install header files in `/usr/local/include` and libs in `/usr/local/lib`. Since these directories are not searched by default, you can set the environment variables to these locations:

```
export CFLAGS="$CFLAGS -I/usr/local/include"
export CXXFLAGS="$CXXFLAGS -I/usr/local/include"
export LDFLAGS="$LDFLAGS -Wl,-rpath,/usr/local/lib -L/usr/local/lib -lomp"
```

Finally, build the package using the standard command:

```
pip install --verbose --editable .
```

For the upcoming FreeBSD 12.1 and 11.3 versions, OpenMP will be included in the base system and these steps will not be necessary.

8.7 Maintainer / core-developer information

8.7.1 Before a release

1. Update authors table:

   ```
   $ cd build_tools; make authors; cd ..
   ```

   and commit.

2. Confirm any blockers tagged for the milestone are resolved, and that other issues tagged for the milestone can be postponed.

3. Ensure the change log and commits correspond (within reason!), and that the change log is reasonably well curated. Some tools for these tasks include:

   - `maint_tools/sort_whats_new.py` can put what's new entries into sections.
   - The `maint_tools/whats_missing.sh` script may be used to identify pull requests that were merged but likely missing from What's New.

Preparing a bug-fix-release

Since any commits to a released branch (e.g. 0.999.X) will automatically update the web site documentation, it is best to develop a bug-fix release with a pull request in which 0.999.X is the base. It also allows you to keep track of any tasks towards release with a TO DO list.

Most development of the bug fix release, and its documentation, should happen in master to avoid asynchrony. To select commits from master for use in the bug fix (version 0.999.3), you can use:

```
$ git checkout -b release-0.999.3 master
$ git rebase -i 0.999.X
```

Then pick the commits for release and resolve any issues, and create a pull request with 0.999.X as base. Add a commit updating `sklearn.__version__`. Additional commits can be cherry-picked into the `release-0.999.3` branch while preparing the release.

8.7.2 Making a release

1. Update docs:
   - Edit the doc/whats_new.rst file to add release title and commit statistics. You can retrieve commit statistics with:

     ```
     $ git shortlog -s 0.99.33.. | cut -f2- | sort --ignore-case | tr '\n' ';' |␣
     ↪sed 's/;/, /g;s/, $//'
     ```

   - Update the release date in whats_new.rst
   - Edit the doc/index.rst to change the 'News' entry of the front page.
   - Note that these changes should be made in master and cherry-picked into the release branch.
2. On the branch for releasing, update the version number in sklearn/__init__.py, the `__version__` variable by removing `dev*` only when ready to release. On master, increment the version in the same place (when branching for release).
3. Create the tag and push it:

   ```
   $ git tag -a 0.999

   $ git push git@github.com:scikit-learn/scikit-learn.git --tags
   ```

4. Create the source tarball:
   - Wipe clean your repo:

     ```
     $ git clean -xfd
     ```

   - Generate the tarball:

     ```
     $ python setup.py sdist
     ```

   The result should be in the `dist/` folder. We will upload it later with the wheels. Check that you can install it in a new virtualenv and that the tests pass.
5. Update the dependency versions and set `BUILD_COMMIT` variable to the release tag at:

   https://github.com/MacPython/scikit-learn-wheels

Once the CI has completed successfully, collect the generated binary wheel packages and upload them to PyPI by running the following commands in the scikit-learn source folder (checked out at the release tag):

```
$ rm -r dist
$ pip install -U wheelhouse_uploader twine
$ python setup.py fetch_artifacts
```

6. Check the content of the `dist/` folder: it should contain all the wheels along with the source tarball ("scikit-learn-XXX.tar.gz").

   Make sure that you do not have developer versions or older versions of the scikit-learn package in that folder.

   Upload everything at once to https://pypi.org:

```
$ twine upload dist/*
```

7. For major/minor (not bug-fix release), update the symlink for `stable` and the `latestStable` variable in https://github.com/scikit-learn/scikit-learn.github.io:

```
$ cd /tmp
$ git clone --depth 1 --no-checkout git@github.com:scikit-learn/scikit-learn.
↪github.io.git
$ cd scikit-learn.github.io
$ echo stable > .git/info/sparse-checkout
$ git checkout master
$ rm stable
$ ln -s 0.999 stable
$ sed -i "s/latestStable = '.*/latestStable = '0.999';" versionwarning.js
$ git commit -m "Update stable to point to 0.999" stable
$ git push origin master
```

The following GitHub checklist might be helpful in a release PR:

```
* [ ] update news and what's new date in master and release branch
* [ ] create tag
* [ ] update dependencies and release tag at https://github.com/MacPython/scikit-
↪learn-wheels
* [ ] twine the wheels to PyPI when that's green
* [ ] https://github.com/scikit-learn/scikit-learn/releases draft
* [ ] confirm bot detected at https://github.com/conda-forge/scikit-learn-feedstock␣
↪and wait for merge
* [ ] https://github.com/scikit-learn/scikit-learn/releases publish
* [ ] announce on mailing list
* [ ] (regenerate Dash docs: https://github.com/Kapeli/Dash-User-Contributions/tree/
↪master/docsets/Scikit)
```

8.7.3 The scikit-learn.org web site

The scikit-learn web site (http://scikit-learn.org) is hosted at GitHub, but should rarely be updated manually by pushing to the https://github.com/scikit-learn/scikit-learn.github.io repository. Most updates can be made by pushing to master (for /dev) or a release branch like 0.99.X, from which Circle CI builds and uploads the documentation automatically.

8.7.4 Travis Cron jobs

From https://docs.travis-ci.com/user/cron-jobs: Travis CI cron jobs work similarly to the cron utility, they run builds at regular scheduled intervals independently of whether any commits were pushed to the repository. Cron jobs always

fetch the most recent commit on a particular branch and build the project at that state. Cron jobs can run daily, weekly or monthly, which in practice means up to an hour after the selected time span, and you cannot set them to run at a specific time.

For scikit-learn, Cron jobs are used for builds that we do not want to run in each PR. As an example the build with the dev versions of numpy and scipy is run as a Cron job. Most of the time when this numpy-dev build fail, it is related to a numpy change and not a scikit-learn one, so it would not make sense to blame the PR author for the Travis failure.

The definition of what gets run in the Cron job is done in the .travis.yml config file, exactly the same way as the other Travis jobs. We use a `if: type = cron` filter in order for the build to be run only in Cron jobs.

The branch targeted by the Cron job and the frequency of the Cron job is set via the web UI at https://www.travis-ci.org/scikit-learn/scikit-learn/settings.

8.7.5 Experimental features

The *sklearn.experimental* module was introduced in 0.21 and contains experimental features / estimators that are subject to change without deprecation cycle.

To create an experimental module, you can just copy and modify the content of enable_hist_gradient_boosting.py, or enable_iterative_imputer.py.

Note that the public import path must be to a public subpackage (like `sklearn/ensemble` or `sklearn/impute`), not just a `.py` module. Also, the (private) experimental features that are imported must be in a submodule/subpackage of the public subpackage, e.g. `sklearn/ensemble/_hist_gradient_boosting/` or `sklearn/impute/_iterative.py`. This is needed so that pickles still work in the future when the features aren't experimental anymore

Please also write basic tests following those in test_enable_hist_gradient_boosting.py.

Make sure every user-facing code you write explicitly mentions that the feature is experimental, and add a `# noqa` comment to avoid pep8-related warnings:

```
# To use this experimental feature, we need to explicitly ask for it:
from sklearn.experimental import enable_hist_gradient_boosting  # noqa
from sklearn.ensemble import HistGradientBoostingRegressor
```

For the docs to render properly, please also import `enable_my_experimental_feature` in `doc/conf.py`, else sphinx won't be able to import the corresponding modules. Note that using `from sklearn.experimental import *` **does not work**.

Note that some experimental classes / functions are not included in the *sklearn.experimental* module: `sklearn.datasets.fetch_openml`.

8.8 Developing with the Plotting API

Scikit-learn defines a simple API for creating visualizations for machine learning. The key features of this API is to run calculations once and to have the flexibility to adjust the visualizations after the fact. This section is intended for developers who wish to develop or maintain plotting tools. For usage, users should refer to the :ref‘User Guide <visualizations>‘.

8.8.1 Plotting API Overview

This logic is encapsulated into a display object where the computed data is stored and the plotting is done in a `plot` method. The display object's `__init__` method contains only the data needed to create the visualization. The `plot`

method takes in parameters that only have to do with visualization, such as a matplotlib axes. The `plot` method will store the matplotlib artists as attributes allowing for style adjustments through the display object. A `plot_*` helper function accepts parameters to do the computation and the parameters used for plotting. After the helper function creates the display object with the computed values, it calls the display's plot method. Note that the `plot` method defines attributes related to matplotlib, such as the line artist. This allows for customizations after calling the `plot` method.

For example, the `RocCurveDisplay` defines the following methods and attributes:

```
class RocCurveDisplay:
    def __init__(self, fpr, tpr, roc_auc, estimator_name):
        ...
        self.fpr = fpr
        self.tpr = tpr
        self.roc_auc = roc_auc
        self.estimator_name = estimator_name

    def plot(self, ax=None, name=None, **kwargs):
        ...
        self.line_ = ...
        self.ax_ = ax
        self.figure_ = ax.figure_

def plot_roc_curve(estimator, X, y, pos_label=None, sample_weight=None,
                   drop_intermediate=True, response_method="auto",
                   name=None, ax=None, **kwargs):
    # do computation
    viz = RocCurveDisplay(fpr, tpr, roc_auc,
                          estimator.__class__.__name__)
    return viz.plot(ax=ax, name=name, **kwargs)
```

Read more in *ROC Curve with Visualization API* and the *User Guide*.

8.8.2 Plotting with Multiple Axes

Some of the plotting tools like *plot_partial_dependence* and *PartialDependenceDisplay* support plottong on multiple axes. Two different scenarios are supported:

1. If a list of axes is passed in, `plot` will check if the number of axes is consistent with the number of axes it expects and then draws on those axes. 2. If a single axes is passed in, that axes defines a space for multiple axes to be placed. In this case, we suggest using matplotlib's `~matplotlib.gridspec.GridSpecFromSubplotSpec` to split up the space:

```
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpecFromSubplotSpec

fig, ax = plt.subplots()
gs = GridSpecFromSubplotSpec(2, 2, subplot_spec=ax.get_subplotspec())

ax_top_left = fig.add_subplot(gs[0, 0])
ax_top_right = fig.add_subplot(gs[0, 1])
ax_bottom = fig.add_subplot(gs[1, :])
```

By default, the `ax` keyword in `plot` is `None`. In this case, the single axes is created and the gridspec api is used to create the regions to plot in.

See for example, *plot_partial_dependence* which plots multiple lines and contours using this API. The axes defining the bounding box is saved in a `bounding_ax_` attribute. The individual axes created are stored in an `axes_`

ndarray, corresponding to the axes position on the grid. Positions that are not used are set to `None`. Furthermore, the matplotlib Artists are stored in `lines_` and `contours_` where the key is the position on the grid. When a list of axes is passed in, the `axes_`, `lines_`, and `contours_` is a 1d ndarray corresponding to the list of axes passed in.

BIBLIOGRAPHY

[M2012] "Machine Learning: A Probabilistic Perspective" Murphy, K. P. - chapter 14.4.3, pp. 492-493, The MIT Press, 2012

[RW2006] Carl Eduard Rasmussen and Christopher K.I. Williams, "Gaussian Processes for Machine Learning", MIT Press 2006, Link to an official complete PDF version of the book here .

[BRE] L. Breiman, J. Friedman, R. Olshen, and C. Stone. Classification and Regression Trees. Wadsworth, Belmont, CA, 1984.

[B1999] L. Breiman, "Pasting small votes for classification in large databases and on-line", Machine Learning, 36(1), 85-103, 1999.

[B1996] L. Breiman, "Bagging predictors", Machine Learning, 24(2), 123-140, 1996.

[H1998] T. Ho, "The random subspace method for constructing decision forests", Pattern Analysis and Machine Intelligence, 20(8), 832-844, 1998.

[LG2012] G. Louppe and P. Geurts, "Ensembles on Random Patches", Machine Learning and Knowledge Discovery in Databases, 346-361, 2012.

[B2001] L. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001.

[B1998] L. Breiman, "Arcing Classifiers", Annals of Statistics 1998.

[L2014] G. Louppe, "Understanding Random Forests: From Theory to Practice", PhD Thesis, U. of Liege, 2014.

[FS1995] Y. Freund, and R. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting", 1997.

[ZZRH2009] J. Zhu, H. Zou, S. Rosset, T. Hastie. "Multi-class AdaBoost", 2009.

[D1997] H. Drucker. "Improving Regressors using Boosting Techniques", 1997.

[HTF] T. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning Ed. 2", Springer, 2009.

[F1999] Friedmann, Jerome H., 2007, "Stochastic Gradient Boosting"

[R2007] G. Ridgeway, "Generalized Boosted Models: A guide to the gbm package", 2007

[XGBoost] Tianqi Chen, Carlos Guestrin, "XGBoost: A Scalable Tree Boosting System"

[LightGBM] Ke et. al. "LightGBM: A Highly Efficient Gradient BoostingDecision Tree"

[W1992] Wolpert, David H. "Stacked generalization." Neural networks 5.2 (1992): 241-259.

[VEB2009] Vinh, Epps, and Bailey, (2009). "Information theoretic measures for clusterings comparison". Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09. doi:10.1145/1553374.1553511. ISBN 9781605585161.

[VEB2010] Vinh, Epps, and Bailey, (2010). "Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance". JMLR <http://jmlr.csail.mit.edu/papers/volume11/vinh10a/vinh10a.pdf>

[YAT2016] Yang, Algesheimer, and Tessone, (2016). "A comparative analysis of community detection algorithms on artificial networks". Scientific Reports 6: 30750. doi:10.1038/srep30750.

[B2011] Identication and Characterization of Events in Social Media, Hila Becker, PhD Thesis.

[Mrl09] "Online Dictionary Learning for Sparse Coding" J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009

[Jen09] "Structured Sparse Principal Component Analysis" R. Jenatton, G. Obozinski, F. Bach, 2009

[R45f14345c000-1] L. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001.

[Rf91cab2dc427-1] L. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001.

[Rf91cab2dc427-2] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.

[Rc8f28bfad63f-1] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.

[Ra7d0c8995fbc-1] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.

[Guyon2015] I. Guyon, K. Bennett, G. Cawley, H.J. Escalante, S. Escalera, T.K. Ho, N. Macià, B. Ray, M. Saeed, A.R. Statnikov, E. Viegas, Design of the 2015 ChaLearn AutoML Challenge, IJCNN 2015.

[Mosley2013] L. Mosley, A balanced approach to the multi-class imbalance problem, IJCV 2010.

[Kelleher2015] John. D. Kelleher, Brian Mac Namee, Aoife D'Arcy, Fundamentals of Machine Learning for Predictive Data Analytics: Algorithms, Worked Examples, and Case Studies, 2015.

[Urbanowicz2015] Urbanowicz R.J., Moore, J.H. ExSTraCS 2.0: description and evaluation of a scalable learning classifier system, Evol. Intel. (2015) 8: 89.

[Manning2008] C.D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, 2008.

[Everingham2010] M. Everingham, L. Van Gool, C.K.I. Williams, J. Winn, A. Zisserman, The Pascal Visual Object Classes (VOC) Challenge, IJCV 2010.

[Davis2006] J. Davis, M. Goadrich, The Relationship Between Precision-Recall and ROC Curves, ICML 2006.

[Flach2015] P.A. Flach, M. Kull, Precision-Recall-Gain Curves: PR Analysis Done Right, NIPS 2015.

[HT2001] Hand, D.J. and Till, R.J., (2001). A simple generalisation of the area under the ROC curve for multiple class classification problems. Machine learning, 45(2), pp.171-186.

[FC2009] Ferri, Cèsar & Hernandez-Orallo, Jose & Modroiu, R. (2009). An Experimental Comparison of Performance Measures for Classification. Pattern Recognition Letters. 30. 27-38.

[PD2000] Provost, F., Domingos, P. (2000). Well-trained PETs: Improving probability estimation trees (Section 6.2), CeDER Working Paper #IS-00-04, Stern School of Business, New York University.

[F2006] Fawcett, T., 2006. An introduction to ROC analysis. Pattern Recognition Letters, 27(8), pp. 861-874.

[F2001] Fawcett, T., 2001. Using rule sets to maximize ROC performance In Data Mining, 2001. Proceedings IEEE International Conference, pp. 131-138.

[NQY18] J. Nothman, H. Qin and R. Yurchak (2018). "Stop Word Lists in Free Open-source Software Packages". In *Proc. Workshop for NLP Open Source Software*.

[OL2001] Olga Troyanskaya, Michael Cantor, Gavin Sherlock, Pat Brown, Trevor Hastie, Robert Tibshirani, David Botstein and Russ B. Altman, Missing value estimation methods for DNA microarrays, BIOINFORMATICS Vol. 17 no. 6, 2001 Pages 520-525.

[RR2007] "Random features for large-scale kernel machines" Rahimi, A. and Recht, B. - Advances in neural information processing 2007,

[LS2010] "Random Fourier approximations for skewed multiplicative histogram kernels" Random Fourier approximations for skewed multiplicative histogram kernels - Lecture Notes for Computer Sciencd (DAGM)

[VZ2010] "Efficient additive kernels via explicit feature maps" Vedaldi, A. and Zisserman, A. - Computer Vision and Pattern Recognition 2010

[VVZ2010] "Generalized RBF feature maps for Efficient Detection" Vempati, S. and Vedaldi, A. and Zisserman, A. and Jawahar, CV - 2010

[R57cf438d7060-1] Obtaining calibrated probability estimates from decision trees and naive Bayesian classifiers, B. Zadrozny & C. Elkan, ICML 2001

[R57cf438d7060-2] Transforming Classifier Scores into Accurate Multiclass Probability Estimates, B. Zadrozny & C. Elkan, (KDD 2002)

[R57cf438d7060-3] Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods, J. Platt, (1999)

[R57cf438d7060-4] Predicting Good Probabilities with Supervised Learning, A. Niculescu-Mizil & R. Caruana, ICML 2005

[R2c55e37003fe-1] Ankerst, Mihael, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. "OPTICS: ordering points to identify the clustering structure." ACM SIGMOD Record 28, no. 2 (1999): 49-60.

[R2c55e37003fe-2] Schubert, Erich, Michael Gertz. "Improving the Cluster Structure Extracted from OPTICS Plots." Proc. of the Conference "Lernen, Wissen, Daten, Analysen" (LWDA) (2018): 318-329.

[1] Ankerst, Mihael, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. "OPTICS: ordering points to identify the clustering structure." ACM SIGMOD Record 28, no. 2 (1999): 49-60.

[R68ae096da0e4-1] Rousseeuw, P.J., Van Driessen, K. "A fast algorithm for the minimum covariance determinant estimator" Technometrics 41(3), 212 (1999)

[RVD] A Fast Algorithm for the Minimum Covariance Determinant Estimator, 1999, American Statistical Association and the American Society for Quality, TECHNOMETRICS

[RVDriessen] A Fast Algorithm for the Minimum Covariance Determinant Estimator, 1999, American Statistical Association and the American Society for Quality, TECHNOMETRICS

[R9f63e655f7bd-Rouseeuw1984] P. J. Rousseeuw. Least median of squares regression. J. Am Stat Ass, 79:871, 1984.

[R9f63e655f7bd-Rousseeuw] A Fast Algorithm for the Minimum Covariance Determinant Estimator, 1999, American Statistical Association and the American Society for Quality, TECHNOMETRICS

[R9f63e655f7bd-ButlerDavies] R. W. Butler, P. L. Davies and M. Jhun, Asymptotics For The Minimum Covariance Determinant Estimator, The Annals of Statistics, 1993, Vol. 21, No. 3, 1385-1400

[RVD] A Fast Algorithm for the Minimum Covariance Determinant Estimator, 1999, American Statistical Association and the American Society for Quality, TECHNOMETRICS

[RVDriessen] A Fast Algorithm for the Minimum Covariance Determinant Estimator, 1999, American Statistical Association and the American Society for Quality, TECHNOMETRICS

[1] Dhillon, I. S. (2001, August). Co-clustering documents and words using bipartite spectral graph partitioning. In Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 269-274). ACM.

[1] Kluger, Y., Basri, R., Chang, J. T., & Gerstein, M. (2003). Spectral biclustering of microarray data: coclustering genes and conditions. Genome research, 13(4), 703-716.

[1] I. Guyon, "Design of experiments for the NIPS 2003 variable selection benchmark", 2003.

[1] J. Friedman, "Multivariate adaptive regression splines", The Annals of Statistics 19 (1), pages 1-67, 1991.

[2] L. Breiman, "Bagging predictors", Machine Learning 24, pages 123-140, 1996.

[1] J. Friedman, "Multivariate adaptive regression splines", The Annals of Statistics 19 (1), pages 1-67, 1991.

[2] L. Breiman, "Bagging predictors", Machine Learning 24, pages 123-140, 1996.

[1] J. Friedman, "Multivariate adaptive regression splines", The Annals of Statistics 19 (1), pages 1-67, 1991.

[2] L. Breiman, "Bagging predictors", Machine Learning 24, pages 123-140, 1996.

[1] J. Zhu, H. Zou, S. Rosset, T. Hastie, "Multi-class AdaBoost", 2009.

[1] T. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning Ed. 2", Springer, 2009.

[1] G. Celeux, M. El Anbari, J.-M. Marin, C. P. Robert, "Regularization in regression: comparing Bayesian and frequentist methods in a poorly informative situation", 2009.

[1] S. Marsland, "Machine Learning: An Algorithmic Perspective", Chapter 10, 2009. http://seat.massey.ac.nz/personal/s.r.marsland/Code/10/lle.py

[Re25e5648fc37-1] "Online Learning for Latent Dirichlet Allocation", Matthew D. Hoffman, David M. Blei, Francis Bach, 2010

[R33e4ec8c4ad5-1] Y. Freund, R. Schapire, "A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting", 1995.

[R33e4ec8c4ad5-2] J. Zhu, H. Zou, S. Rosset, T. Hastie, "Multi-class AdaBoost", 2009.

[R0c261b7dee9d-1] Y. Freund, R. Schapire, "A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting", 1995.

[R0c261b7dee9d-2] H. Drucker, "Improving Regressors using Boosting Techniques", 1997.

[Rb1846455d0e5-1] L. Breiman, "Pasting small votes for classification in large databases and on-line", Machine Learning, 36(1), 85-103, 1999.

[Rb1846455d0e5-2] L. Breiman, "Bagging predictors", Machine Learning, 24(2), 123-140, 1996.

[Rb1846455d0e5-3] T. Ho, "The random subspace method for constructing decision forests", Pattern Analysis and Machine Intelligence, 20(8), 832-844, 1998.

[Rb1846455d0e5-4] G. Louppe and P. Geurts, "Ensembles on Random Patches", Machine Learning and Knowledge Discovery in Databases, 346-361, 2012.

[R4d113ba76fc0-1] L. Breiman, "Pasting small votes for classification in large databases and on-line", Machine Learning, 36(1), 85-103, 1999.

[R4d113ba76fc0-2] L. Breiman, "Bagging predictors", Machine Learning, 24(2), 123-140, 1996.

[R4d113ba76fc0-3] T. Ho, "The random subspace method for constructing decision forests", Pattern Analysis and Machine Intelligence, 20(8), 832-844, 1998.

[R4d113ba76fc0-4] G. Louppe and P. Geurts, "Ensembles on Random Patches", Machine Learning and Knowledge Discovery in Databases, 346-361, 2012.

[Rd7ae0a2ae688-1] Liu, Fei Tony, Ting, Kai Ming and Zhou, Zhi-Hua. "Isolation forest." Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on.

[Rd7ae0a2ae688-2] Liu, Fei Tony, Ting, Kai Ming and Zhou, Zhi-Hua. "Isolation-based anomaly detection." ACM Transactions on Knowledge Discovery from Data (TKDD) 6.1 (2012): 3.

[R6e47e53bacbd-1] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.

[R6e47e53bacbd-2] Moosmann, F. and Triggs, B. and Jurie, F. "Fast discriminative visual codebooks using randomized clustering forests" NIPS 2007

[Rb91ed47a817e-1] Wolpert, David H. "Stacked generalization." Neural networks 5.2 (1992): 241-259.

[R606df7ffad02-1] Wolpert, David H. "Stacked generalization." Neural networks 5.2 (1992): 241-259.

[R1b90ac3ca370-Yates2011] R. Baeza-Yates and B. Ribeiro-Neto (2011). Modern Information Retrieval. Addison Wesley, pp. 68-74.

[R1b90ac3ca370-MRS2008] C.D. Manning, P. Raghavan and H. Schütze (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 118-120.

[Re310f679c81e-1] Guyon, I., Weston, J., Barnhill, S., & Vapnik, V., "Gene selection for cancer classification using support vector machines", Mach. Learn., 46(1-3), 389–422, 2002.

[R6f4d61ceb411-1] Guyon, I., Weston, J., Barnhill, S., & Vapnik, V., "Gene selection for cancer classification using support vector machines", Mach. Learn., 46(1-3), 389–422, 2002.

[1] Mutual Information on Wikipedia.

[2] A. Kraskov, H. Stogbauer and P. Grassberger, "Estimating mutual information". Phys. Rev. E 69, 2004.

[3] B. C. Ross "Mutual Information between Discrete and Continuous Data Sets". PLoS ONE 9(2), 2014.

[4] L. F. Kozachenko, N. N. Leonenko, "Sample Estimate of the Entropy of a Random Vector:, Probl. Peredachi Inf., 23:2 (1987), 9-16

[1] Mutual Information on Wikipedia.

[2] A. Kraskov, H. Stogbauer and P. Grassberger, "Estimating mutual information". Phys. Rev. E 69, 2004.

[3] B. C. Ross "Mutual Information between Discrete and Continuous Data Sets". PLoS ONE 9(2), 2014.

[4] L. F. Kozachenko, N. N. Leonenko, "Sample Estimate of the Entropy of a Random Vector", Probl. Peredachi Inf., 23:2 (1987), 9-16

[Rcd31b817a31e-1] Stef van Buuren, Karin Groothuis-Oudshoorn (2011). "mice: Multivariate Imputation by Chained Equations in R". Journal of Statistical Software 45: 1-67.

[Rcd31b817a31e-2] S. F. Buck, (1960). "A Method of Estimation of Missing Values in Multivariate Data Suitable for use with an Electronic Computer". Journal of the Royal Statistical Society 22(2): 302-306.

[BRE] L. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001. https://doi.org/10.1023/A:1010933404324

[Re4616ef910fb-1] Peter J. Huber, Elvezio M. Ronchetti, Robust Statistics Concomitant scale estimates, pg 172

[Re4616ef910fb-2] Art B. Owen (2006), A robust hybrid of lasso and ridge regression. https://statweb.stanford.edu/~owen/reports/hhu.pdf

[R80ce5b25cf9d-1] https://en.wikipedia.org/wiki/RANSAC

[R80ce5b25cf9d-2] https://www.sri.com/sites/default/files/publications/ransac-publication.pdf

[R80ce5b25cf9d-3] http://www.bmva.org/bmvc/2009/Papers/Paper355/Paper355.pdf

[1] "Least Angle Regression", Efron et al. http://statweb.stanford.edu/~tibs/ftp/lars.pdf

[2] Wikipedia entry on the Least-angle regression

[3] Wikipedia entry on the Lasso

[1] "Least Angle Regression", Efron et al. http://statweb.stanford.edu/~tibs/ftp/lars.pdf

[2] Wikipedia entry on the Least-angle regression

[3] Wikipedia entry on the Lasso

[R7f4d308f5054-1] Tenenbaum, J.B.; De Silva, V.; & Langford, J.C. A global geometric framework for nonlinear dimensionality reduction. Science 290 (5500)

[R62e36dd1b056-1] Roweis, S. & Saul, L. Nonlinear dimensionality reduction by locally linear embedding. Science 290:2323 (2000).

[R62e36dd1b056-2] Donoho, D. & Grimes, C. Hessian eigenmaps: Locally linear embedding techniques for high-dimensional data. Proc Natl Acad Sci U S A. 100:5591 (2003).

[R62e36dd1b056-3] Zhang, Z. & Wang, J. MLLE: Modified Locally Linear Embedding Using Multiple Weights. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.70.382

[R62e36dd1b056-4] Zhang, Z. & Zha, H. Principal manifolds and nonlinear dimensionality reduction via tangent space alignment. Journal of Shanghai Univ. 8:406 (2004)

[1] Roweis, S. & Saul, L. Nonlinear dimensionality reduction by locally linear embedding. Science 290:2323 (2000).

[2] Donoho, D. & Grimes, C. Hessian eigenmaps: Locally linear embedding techniques for high-dimensional data. Proc Natl Acad Sci U S A. 100:5591 (2003).

[3] Zhang, Z. & Wang, J. MLLE: Modified Locally Linear Embedding Using Multiple Weights. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.70.382

[4] Zhang, Z. & Zha, H. Principal manifolds and nonlinear dimensionality reduction via tangent space alignment. Journal of Shanghai Univ. 8:406 (2004)

[1] Wikipedia entry for the Average precision

[1] Brodersen, K.H.; Ong, C.S.; Stephan, K.E.; Buhmann, J.M. (2010). The balanced accuracy and its posterior distribution. Proceedings of the 20th International Conference on Pattern Recognition, 3121-24.

[2] John. D. Kelleher, Brian Mac Namee, Aoife D'Arcy, (2015). Fundamentals of Machine Learning for Predictive Data Analytics: Algorithms, Worked Examples, and Case Studies.

[1] Wikipedia entry for the Brier score.

[1] J. Cohen (1960). "A coefficient of agreement for nominal scales". Educational and Psychological Measurement 20(1):37-46. doi:10.1177/001316446002000104.

[2] R. Artstein and M. Poesio (2008). "Inter-coder agreement for computational linguistics". Computational Linguistics 34(4):555-596.

[3] Wikipedia entry for the Cohen's kappa.

[1] Wikipedia entry for the Confusion matrix (Wikipedia and other references may use a different convention for axes)

[1] Wikipedia entry for the F1-score

[1] R. Baeza-Yates and B. Ribeiro-Neto (2011). Modern Information Retrieval. Addison Wesley, pp. 327-328.

[2] Wikipedia entry for the F1-score

[1] Grigorios Tsoumakas, Ioannis Katakis. Multi-Label Classification: An Overview. International Journal of Data Warehousing & Mining, 3(3), 1-13, July-September 2007.

[2] Wikipedia entry on the Hamming distance

[1] Wikipedia entry on the Hinge loss

[2] Koby Crammer, Yoram Singer. On the Algorithmic Implementation of Multiclass Kernel-based Vector Machines. Journal of Machine Learning Research 2, (2001), 265-292

[3] L1 AND L2 Regularization for Multiclass Hinge Loss Models by Robert C. Moore, John DeNero.

[1] Wikipedia entry for the Jaccard index

[1] Baldi, Brunak, Chauvin, Andersen and Nielsen, (2000). Assessing the accuracy of prediction algorithms for classification: an overview

[2] Wikipedia entry for the Matthews Correlation Coefficient

[3] Gorodkin, (2004). Comparing two K-category assignments by a K-category correlation coefficient

[4] Jurman, Riccadonna, Furlanello, (2012). A Comparison of MCC and CEN Error Measures in MultiClass Prediction

[1] Wikipedia entry for the Precision and recall

[2] Wikipedia entry for the F1-score

[3] Discriminative Methods for Multi-labeled Classification Advances in Knowledge Discovery and Data Mining (2004), pp. 22-30 by Shantanu Godbole, Sunita Sarawagi

[1] Wikipedia entry for the Receiver operating characteristic

[2] Analyzing a portion of the ROC curve. McClish, 1989

[3] Provost, F., Domingos, P. (2000). Well-trained PETs: Improving probability estimation trees (Section 6.2), CeDER Working Paper #IS-00-04, Stern School of Business, New York University.

[4] Fawcett, T. (2006). An introduction to ROC analysis. Pattern Recognition Letters, 27(8), 861-874.

[5] Hand, D.J., Till, R.J. (2001). A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems. Machine Learning, 45(2), 171-186.

[1] Wikipedia entry for the Receiver operating characteristic

[2] Fawcett T. An introduction to ROC analysis[J]. Pattern Recognition Letters, 2006, 27(8):861-874.

[1] Wikipedia entry on the Coefficient of determination

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

[1] Vinh, Epps, and Bailey, (2010). Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance, JMLR

[2] Wikipedia entry for the Adjusted Mutual Information

[Hubert1985] L. Hubert and P. Arabie, Comparing Partitions, Journal of Classification 1985 https://link.springer.com/article/10.1007%2FBF01908075

[wk] https://en.wikipedia.org/wiki/Rand_index#Adjusted_Rand_index

[1] T. Calinski and J. Harabasz, 1974. "A dendrite method for cluster analysis". Communications in Statistics

[1] Davies, David L.; Bouldin, Donald W. (1979). "A Cluster Separation Measure". IEEE Transactions on Pattern Analysis and Machine Intelligence. PAMI-1 (2): 224-227

[1] Andrew Rosenberg and Julia Hirschberg, 2007. V-Measure: A conditional entropy-based external cluster evaluation measure

[1] E. B. Fowkles and C. L. Mallows, 1983. "A method for comparing two hierarchical clusterings". Journal of the American Statistical Association

[2] Wikipedia entry for the Fowlkes-Mallows Index

[1] Andrew Rosenberg and Julia Hirschberg, 2007. V-Measure: A conditional entropy-based external cluster evaluation measure

[1] Peter J. Rousseeuw (1987). "Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis". Computational and Applied Mathematics 20: 53-65.

[2] Wikipedia entry on the Silhouette Coefficient

[1] Peter J. Rousseeuw (1987). "Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis". Computational and Applied Mathematics 20: 53-65.

[2] Wikipedia entry on the Silhouette Coefficient

[1] Andrew Rosenberg and Julia Hirschberg, 2007. V-Measure: A conditional entropy-based external cluster evaluation measure

[R16529824bff2-1] Bishop, Christopher M. (2006). "Pattern recognition and machine learning". Vol. 4 No. 4. New York: Springer.

[R16529824bff2-2] Hagai Attias. (2000). "A Variational Bayesian Framework for Graphical Models". In Advances in Neural Information Processing Systems 12.

[R16529824bff2-3] Blei, David M. and Michael I. Jordan. (2006). "Variational inference for Dirichlet process mixtures". Bayesian analysis 1.1

[R2eddaeec0849-1] "Solving multiclass learning problems via error-correcting output codes", Dietterich T., Bakiri G., Journal of Artificial Intelligence Research 2, 1995.

[R2eddaeec0849-2] "The error coding method and PICTs", James G., Hastie T., Journal of Computational and Graphical statistics 7, 1998.

[R2eddaeec0849-3] "The Elements of Statistical Learning", Hastie T., Tibshirani R., Friedman J., page 606 (second-edition) 2008.

[Rca479bb49841-1] Breunig, M. M., Kriegel, H. P., Ng, R. T., & Sander, J. (2000, May). LOF: identifying density-based local outliers. In ACM sigmod record.

[Rf9b6baee8229-1] J. Goldberger, G. Hinton, S. Roweis, R. Salakhutdinov. "Neighbourhood Components Analysis". Advances in Neural Information Processing Systems. 17, 513-520, 2005. http://www.cs.nyu.edu/~roweis/papers/ncanips.pdf

[Rf9b6baee8229-2] Wikipedia entry on Neighborhood Components Analysis https://en.wikipedia.org/wiki/Neighbourhood_components_analysis

[Rf3e1504535de-1] I.K. Yeo and R.A. Johnson, "A new family of power transformations to improve normality or symmetry." Biometrika, 87(4), pp.954-959, (2000).

[Rf3e1504535de-2] G.E.P. Box and D.R. Cox, "An Analysis of Transformations", Journal of the Royal Statistical Society B, 26, 211-252 (1964).

[1] I.K. Yeo and R.A. Johnson, "A new family of power transformations to improve normality or symmetry." Biometrika, 87(4), pp.954-959, (2000).

[2] G.E.P. Box and D.R. Cox, "An Analysis of Transformations", Journal of the Royal Statistical Society B, 26, 211-252 (1964).

[R0fecf191e4b8-1] Ping Li, T. Hastie and K. W. Church, 2006, "Very Sparse Random Projections". https://web.stanford.edu/~hastie/Papers/Ping/KDD06_rp.pdf

[R0fecf191e4b8-2] D. Achlioptas, 2001, "Database-friendly random projections", https://users.soe.ucsc.edu/~optas/papers/jl.pdf

[1] https://en.wikipedia.org/wiki/Johnson%E2%80%93Lindenstrauss_lemma

[2] Sanjoy Dasgupta and Anupam Gupta, 1999, "An elementary proof of the Johnson-Lindenstrauss Lemma." http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.3654

[R9709ce4a60d3-1] LIBSVM: A Library for Support Vector Machines

[R9709ce4a60d3-2] Platt, John (1999). "Probabilistic outputs for support vector machines and comparison to regularizedlikelihood methods."

[R20c70293ef72-1] LIBSVM: A Library for Support Vector Machines

[R20c70293ef72-2] Platt, John (1999). "Probabilistic outputs for support vector machines and comparison to regularizedlikelihood methods."

[Rb1ec977cd307-1] https://en.wikipedia.org/wiki/Decision_tree_learning

[Rb1ec977cd307-2] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification and Regression Trees", Wadsworth, Belmont, CA, 1984.

[Rb1ec977cd307-3] T. Hastie, R. Tibshirani and J. Friedman. "Elements of Statistical Learning", Springer, 2009.

[Rb1ec977cd307-4] L. Breiman, and A. Cutler, "Random Forests", https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

[Ra37b7e3adb19-1] https://en.wikipedia.org/wiki/Decision_tree_learning

[Ra37b7e3adb19-2] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification and Regression Trees", Wadsworth, Belmont, CA, 1984.

[Ra37b7e3adb19-3] T. Hastie, R. Tibshirani and J. Friedman. "Elements of Statistical Learning", Springer, 2009.

[Ra37b7e3adb19-4] L. Breiman, and A. Cutler, "Random Forests", https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

[Rdd99a0224c6e-1] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.

[R4939d63d5a49-1] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.

[1] Wikipedia entry for the Jaccard index

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z