scikit-learn user guide

Release 0.16.1

scikit-learn developers

April 15, 2015

CHAPTER

ONE

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.

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), is it said to have several attributes or **features**.

We can separate learning problems in a few large 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 classification problem would be the handwritten digit recognition example, 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 applying them to new data. This is why a common practice in machine learning to evaluate an algorithm is to split the data at hand into two sets, one that we call the **training set** on which we learn data properties and one that we call the **testing set** on which we test these properties.

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.

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` that implements support vector classification. The constructor of an estimator takes as arguments the parameters of the model, but for the time being, 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. It is possible to automatically find good values for the parameters by using tools such as *grid search* and *cross validation*.

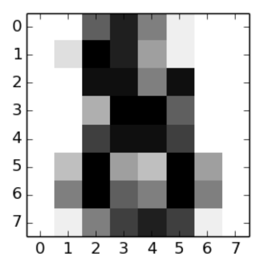
We call our estimator instance `clf`, as it is a classifier. It now must be fitted to the model, that is, it must *learn* from the model. This is done by passing our training set to the `fit` method. As a training set, let us use all the images of our dataset apart from the last one. We select this training set with the `[:-1]` Python syntax, which produces a new array that contains all but the last entry of `digits.data`:

```
>>> clf.fit(digits.data[:-1], digits.target[:-1])
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
  gamma=0.001, kernel='rbf', max_iter=-1, probability=False,
  random_state=None, shrinking=True, tol=0.001, verbose=False)
```

Now you can predict new values, in particular, we can ask to the classifier what is the digit of our last image in the `digits` dataset, which we have not used to train the classifier:

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

The corresponding image is the following:



As you can see, it is a challenging task: 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*.

1.4 Model persistence

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

```
>>> from sklearn import svm
>>> from sklearn import datasets
>>> clf = svm.SVC()
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> clf.fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3, gamma=0.0,
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)

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

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

```
>>> from sklearn.externals import joblib
>>> joblib.dump(clf, 'filename.pkl')
```

Later you can load back the pickled model (possibly in another Python process) with:

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

Note: joblib.dump returns a list of filenames. Each individual numpy array contained in the `clf` object is serialized as a separate file on the filesystem. All files are required in the same folder when reloading the model with joblib.load.

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.

CHAPTER

TWO

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).

2.1 Statistical learning: the setting and the estimator object in scikit-learn

2.1.1 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.

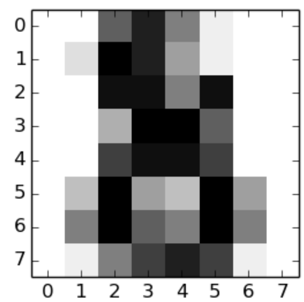
A simple example shipped with the scikit: iris dataset

```
>>> 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 pylab as pl
>>> pl.imshow(digits.images[-1], cmap=pl.cm.gray_r)
<matplotlib.image.AxesImage object at ...>
```

To use this dataset with the scikit, we transform each 8x8 image into a feature vector of length 64

```
>>> data = digits.images.reshape((digits.images.shape[0], -1))
```

2.1.2 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 Supervised learning: predicting an output variable from high-dimensional observations

The problem solved in supervised learning

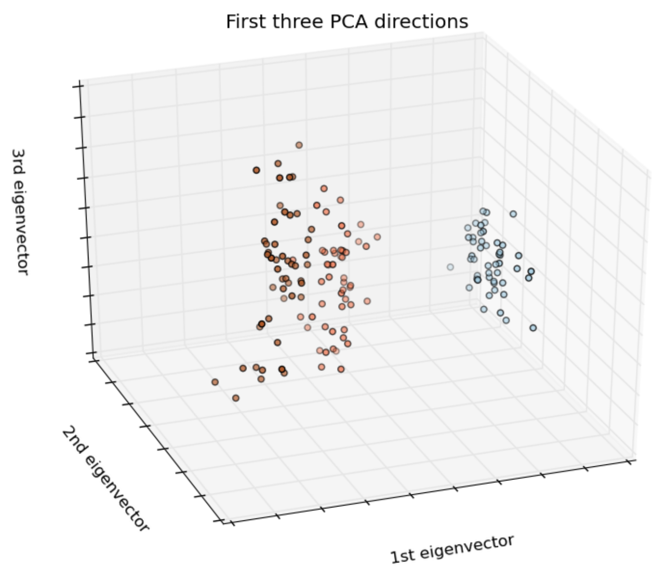
Supervised learning consists in learning the link between two datasets: the observed data X and an external variable y that we are trying to predict, usually called "target" or "labels". Most often, y is a 1D array of length n_samples.
All supervised estimators in scikit-learn implement a fit(X, y) method 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.

2.2.1 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:

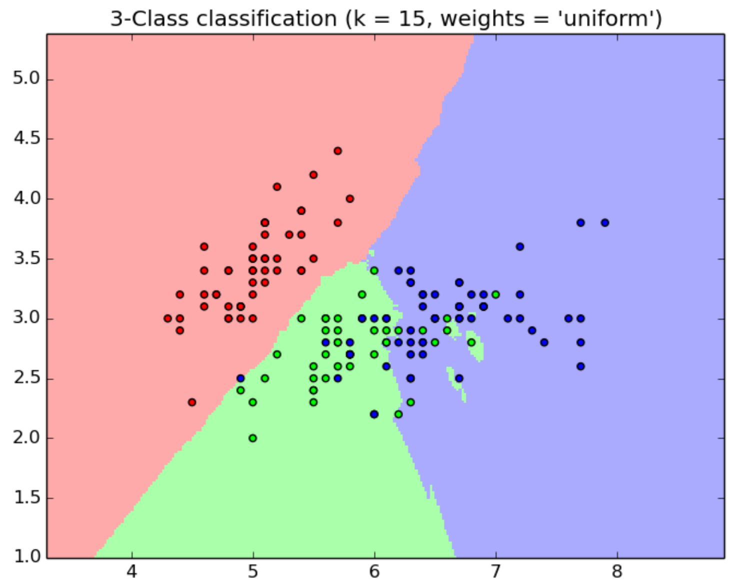
```
>>> import numpy as np
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> iris_X = iris.data
>>> iris_y = iris.target
>>> np.unique(iris_y)
array([0, 1, 2])
```

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(algorithm='auto', leaf_size=30, metric='minkowski',
               metric_params=None, n_neighbors=5, p=2, weights='uniform')
>>> 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 $n \; 1/d$ 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 $n \; 1/d^p$ 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 20 dimensions would require more training data than the current estimated size of the entire internet (±1000 Exabytes or so).

This is called the curse of dimensionality and is a core problem that machine learning addresses.

2.2.2 Linear model: from regression to sparsity

Diabetes dataset

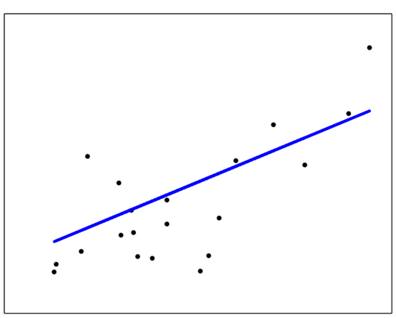
The diabetes dataset consists of 10 physiological variables (age, sex, weight, blood pressure) measure on 442 patients, and an indication of disease progression after one year:

```
>>> diabetes = datasets.load_diabetes()
>>> diabetes_X_train = diabetes.data[:-20]
>>> diabetes_X_test  = diabetes.data[-20:]
>>> diabetes_y_train = diabetes.target[:-20]
>>> diabetes_y_test  = diabetes.target[-20:]
```

The task at hand is to predict disease progression from physiological variables.

Linear regression

`LinearRegression`, in it's 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: $y = X\beta + \epsilon$

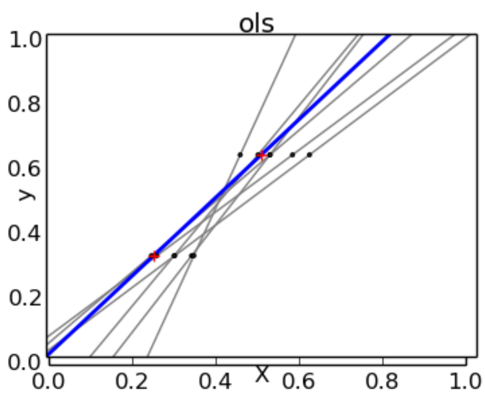
- X: data
- y: target variable
- β: Coefficients

- ϵ: Observation noise

```
>>> from sklearn import linear_model
>>> regr = linear_model.LinearRegression()
>>> regr.fit(diabetes_X_train, diabetes_y_train)
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
>>> 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...
```

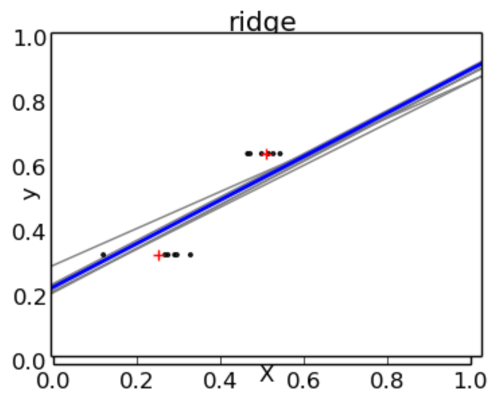
Shrinkage

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 pylab as pl
>>> pl.figure()

>>> np.random.seed(0)
>>> for _ in range(6):
...     this_X = .1*np.random.normal(size=(2, 1)) + X
...     regr.fit(this_X, y)
...     pl.plot(test, regr.predict(test))
...     pl.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)

>>> pl.figure()

>>> np.random.seed(0)
>>> for _ in range(6):
...     this_X = .1*np.random.normal(size=(2, 1)) + X
...     regr.fit(this_X, y)
...     pl.plot(test, regr.predict(test))
...     pl.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.

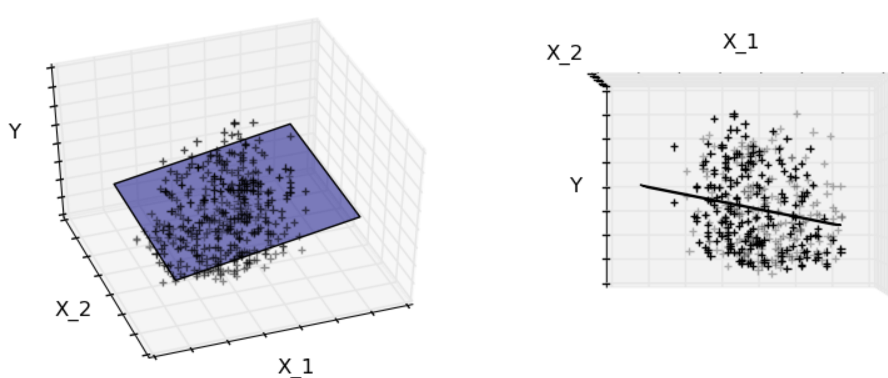
We can choose `alpha` to minimize left out error, this time using the diabetes dataset rather than our synthetic data:

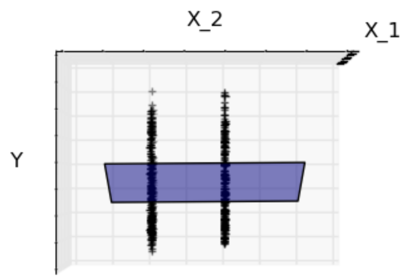
```
>>> alphas = np.logspace(-4, -1, 6)
>>> from __future__ import print_function
>>> print([regr.set_params(alpha=alpha
...             ).fit(diabetes_X_train, diabetes_y_train,
...             ).score(diabetes_X_test, diabetes_y_test) for alpha in alphas])
[0.5851110683883..., 0.5852073015444..., 0.5854677540698..., 0.5855512036503..., 0.5830717085554...,
```

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.

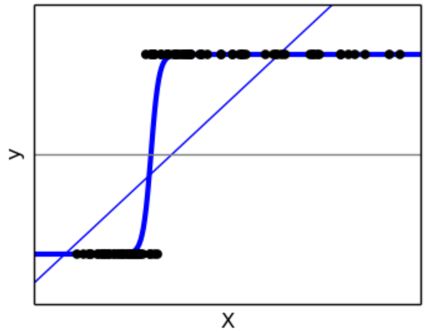
To improve the conditioning of the problem (i.e. mitigating the *The curse of dimensionality*), it would be interesting to select only the informative features and set non-informative ones, like feature 2 to 0. Ridge regression will decrease their contribution, but not set them to zero. Another penalization 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, copy_X=True, fit_intercept=True,
   max_iter=1000, normalize=False, positive=False, precompute=False,
   random_state=None, selection='cyclic', tol=0.0001, warm_start=False)
>>> 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 decent method, that is efficient on large datasets. However, scikit-learn also provides the `LassoLars` object using the *LARS* algorthm, which is very efficient for problems in which the weight vector estimated is very sparse (i.e. problems with very few observations).

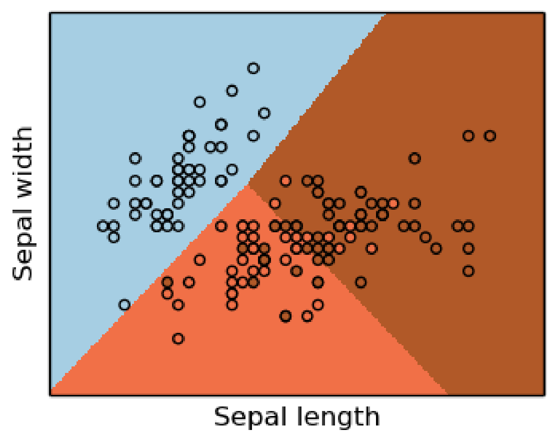
Classification



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:

$$y = \textrm{sigmoid}(X\beta - \textrm{offset}) + \epsilon = \frac{1}{1 + \textrm{exp}(-X\beta + \textrm{offset})} + \epsilon$$

```
>>> logistic = linear_model.LogisticRegression(C=1e5)
>>> logistic.fit(iris_X_train, iris_y_train)
LogisticRegression(C=100000.0, class_weight=None, dual=False,
          fit_intercept=True, intercept_scaling=1, max_iter=100,
          multi_class='ovr', penalty='l2', random_state=None,
          solver='liblinear', tol=0.0001, verbose=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.

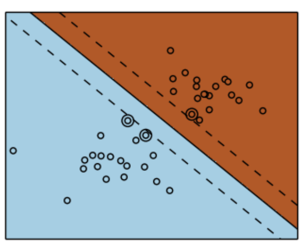
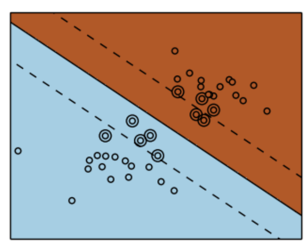
```
from sklearn import datasets, neighbors, linear_model

digits = datasets.load_digits()
X_digits = digits.data
y_digits = digits.target
```

Solution: `../../auto_examples/exercises/digits_classification_exercise.py`

2.2.3 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).

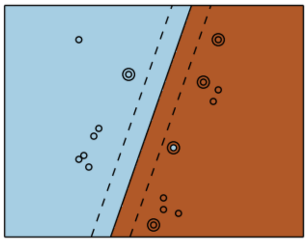
```
>>> from sklearn import svm
>>> svc = svm.SVC(kernel='linear')
>>> svc.fit(iris_X_train, iris_y_train)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3, gamma=0.0,
  kernel='linear', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
```

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

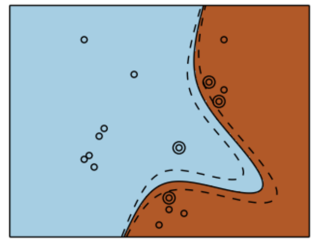
Classes are not always linearly separable in feature space. The solution is to build a decision function that is not linear but may be polynomial instead. This is done using the *kernel trick* that can be seen as creating a decision energy by positioning *kernels* on observations:

Linear kernel



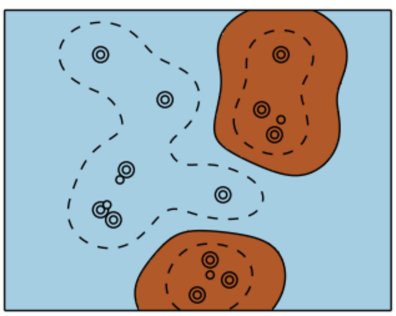
```
>>> svc = svm.SVC(kernel='linear')
```

Polynomial kernel



```
>>> svc = svm.SVC(kernel='poly',
...               degree=3)
>>> # degree: polynomial degree
```

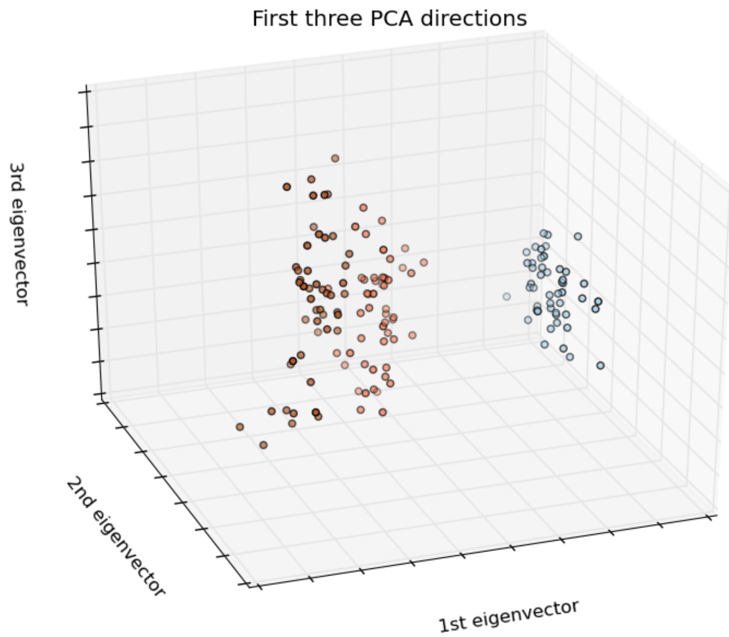
RBF kernel (Radial Basis Function)



```
>>> 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`

2.3 Model selection: choosing estimators and their parameters

2.3.1 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
>>> digits = datasets.load_digits()
>>> X_digits = digits.data
>>> y_digits = digits.target
>>> svc = svm.SVC(C=1, kernel='linear')
>>> svc.fit(X_digits[:-100], y_digits[:-100]).score(X_digits[-100:], y_digits[-100:])
0.97999999999999998
```

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)
[0.93489148580968284, 0.95659432387312182, 0.93989983305509184]
```

This is called a `KFold` cross validation

2.3.2 Cross-validation generators

The code above to split data in train and test sets is tedious to write. Scikit-learn exposes cross-validation generators to generate list of indices for this purpose:

```
>>> from sklearn import cross_validation
>>> k_fold = cross_validation.KFold(n=6, n_folds=3)
>>> for train_indices, test_indices in k_fold:
...      print('Train: %s | test: %s' % (train_indices, test_indices))
Train: [2 3 4 5] | test: [0 1]
Train: [0 1 4 5] | test: [2 3]
Train: [0 1 2 3] | test: [4 5]
```

The cross-validation can then be implemented easily:

```
>>> kfold = cross_validation.KFold(len(X_digits), n_folds=3)
>>> [svc.fit(X_digits[train], y_digits[train]).score(X_digits[test], y_digits[test])
...          for train, test in kfold]
[0.93489148580968284, 0.95659432387312182, 0.93989983305509184]
```

To compute the `score` method of an estimator, the sklearn exposes a helper function:

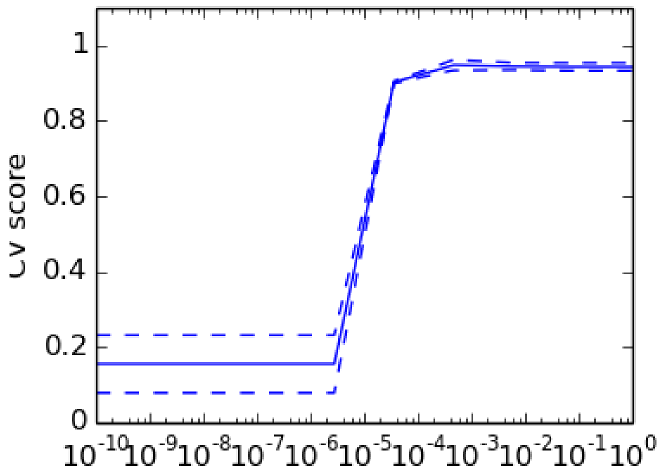
```
>>> cross_validation.cross_val_score(svc, X_digits, y_digits, cv=kfold, n_jobs=-1)
array([ 0.93489149,  0.95659432,  0.93989983])
```

n_jobs=-1 means that the computation will be dispatched on all the CPUs of the computer.

Cross-validation generators

`KFold` **(n, k)**	`StratifiedKFold` **(y, k)**	`LeaveOneOut` **(n)**	`LeaveOneLabelOut` **(labels)**
Split it K folds, train on K-1 and then test on left-out	It preserves the class ratios / label distribution within each fold.	Leave one observation out	Takes a label array to group observations

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 import cross_validation, datasets, svm

digits = datasets.load_digits()
X = digits.data
y = digits.target

svc = svm.SVC(kernel='linear')
C_s = np.logspace(-10, 0, 10)
```

Solution: *Cross-validation on Digits Dataset Exercise*

2.3.3 Grid-search and cross-validated estimators

Grid-search

The sklearn 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.grid_search import GridSearchCV
>>> 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.

Nested cross-validation

```
>>> 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 the sklearn 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()
>>> diabetes = datasets.load_diabetes()
>>> X_diabetes = diabetes.data
>>> y_diabetes = diabetes.target
>>> lasso.fit(X_diabetes, y_diabetes)
LassoCV(alphas=None, copy_X=True, cv=None, eps=0.001, fit_intercept=True,
    max_iter=1000, n_alphas=100, n_jobs=1, normalize=False, positive=False,
    precompute='auto', random_state=None, selection='cyclic', tol=0.0001,
    verbose=False)
>>> # The estimator chose automatically its lambda:
>>> lasso.alpha_
0.01229...
```

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 cross_validation, datasets, linear_model

diabetes = datasets.load_diabetes()
X = diabetes.data[:150]
y = diabetes.target[:150]

lasso = linear_model.Lasso()
alphas = np.logspace(-4, -.5, 30)
```

Solution: *Cross-validation on diabetes Dataset Exercise*

2.4 Unsupervised learning: seeking representations of the data

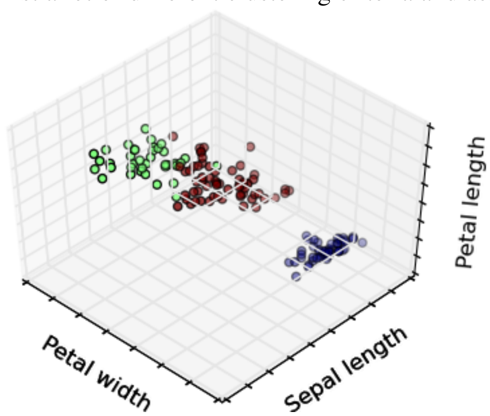
2.4.1 Clustering: grouping observations together

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

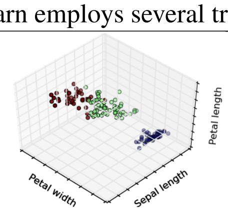
Note that there exist a lot of different clustering criteria and associated algorithms. The simplest clustering algorithm



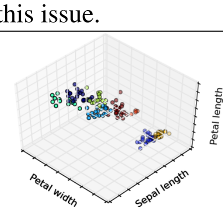
is *K-means*.

```
>>> from sklearn import cluster, datasets
>>> iris = datasets.load_iris()
>>> X_iris = iris.data
>>> y_iris = iris.target

>>> k_means = cluster.KMeans(n_clusters=3)
>>> k_means.fit(X_iris)
KMeans(copy_x=True, init='k-means++', ...
>>> 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]
```

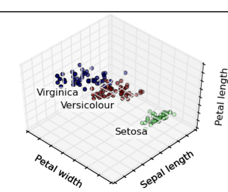
Warning: There is absolutely no guarantee of recovering a ground truth. First, choosing the right number of clusters is hard. Second, the algorithm is sensitive to initialization, and can fall into local minima, although scikit-learn employs several tricks to mitigate this issue.



Bad initialization



8 clusters



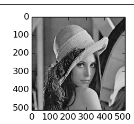
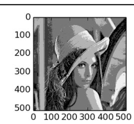
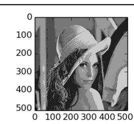
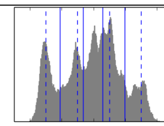
Ground truth

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:
...     lena = sp.lena()
... except AttributeError:
...     from scipy import misc
...     lena = misc.lena()
>>> X = lena.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(copy_x=True, init='k-means++', ...
>>> values = k_means.cluster_centers_.squeeze()
>>> labels = k_means.labels_
>>> lena_compressed = np.choose(labels, values)
>>> lena_compressed.shape = lena.shape
```

 Raw image	 K-means quantization	 Equal bins	 Image histogram

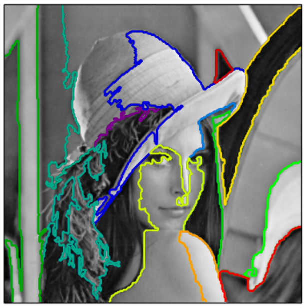
Hierarchical agglomerative clustering: Ward

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 the scikit 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

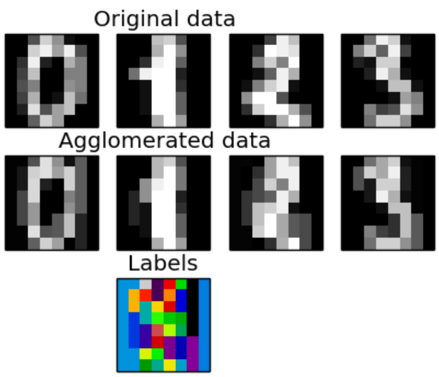


clustering an image:

```
from sklearn.feature_extraction.image import grid_to_graph
from sklearn.cluster import AgglomerativeClustering

###############################################################################
# Generate data
lena = sp.misc.lena()
# Downsample the image by a factor of 4
lena = lena[::2, ::2] + lena[1::2, ::2] + lena[::2, 1::2] + lena[1::2, 1::2]
X = np.reshape(lena, (-1, 1))

###############################################################################
# Define the structure A of the data. Pixels connected to their neighbors.
connectivity = grid_to_graph(*lena.shape)

###############################################################################
# Compute clustering
print("Compute structured hierarchical clustering...")
st = time.time()
n_clusters = 15  # number of regions
ward = AgglomerativeClustering(n_clusters=n_clusters,
        linkage='ward', connectivity=connectivity).fit(X)
label = np.reshape(ward.labels_, lena.shape)
print("Elapsed time: ", time.time() - st)
print("Number of pixels: ", label.size)
print("Number of clusters: ", np.unique(label).size)
```

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(affinity='euclidean', compute_full_tree='auto',...
>>> 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.

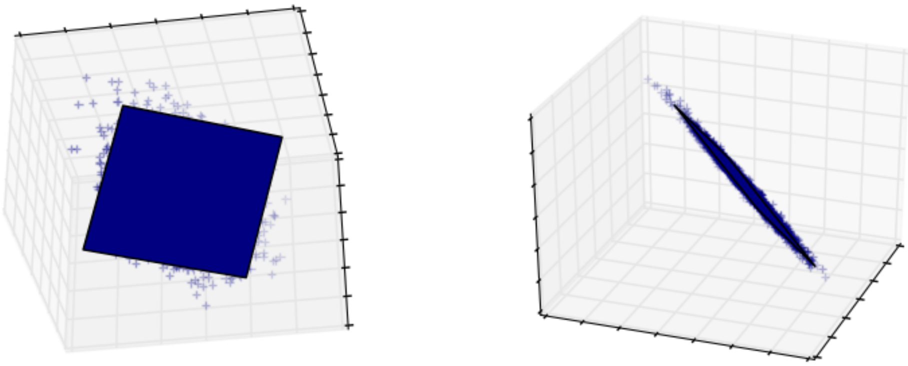
2.4.2 Decompositions: from a signal to components and loadings

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.



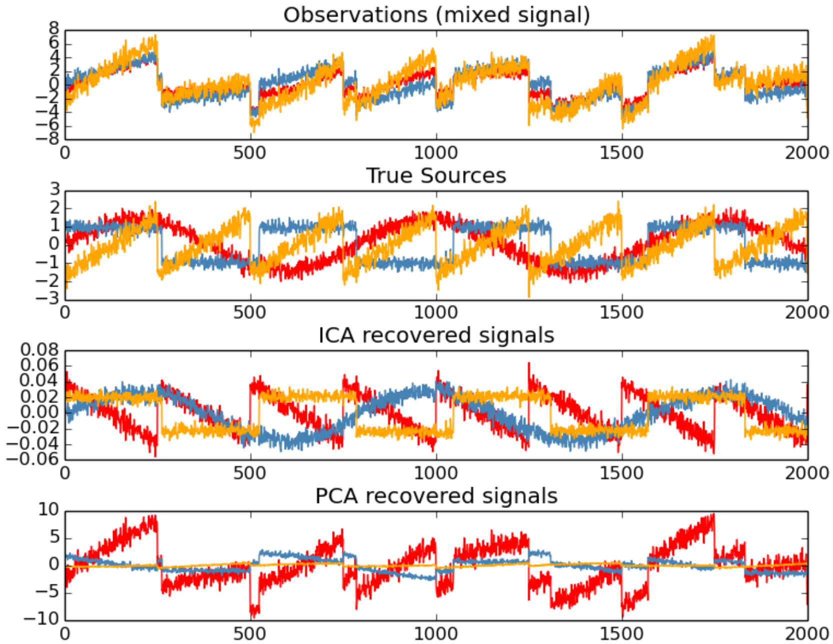
The point cloud spanned by the observations above is very flat in one direction: one of the three univariate features can almost be exactly computed using the other two. PCA finds the directions in which the data is not *flat*

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(copy=True, n_components=None, whiten=False)
>>> 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:

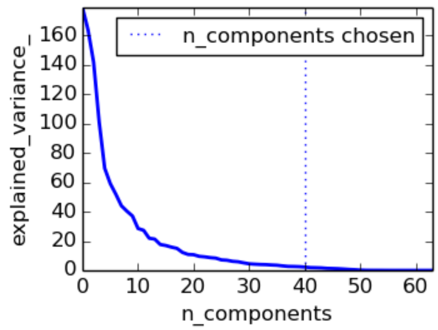



```
>>> # Generate sample data
>>> time = np.linspace(0, 10, 2000)
>>> s1 = np.sin(2 * time)  # Signal 1 : sinusoidal signal
>>> s2 = np.sign(np.sin(3 * time))  # Signal 2 : square signal
>>> S = np.c_[s1, s2]
>>> S += 0.2 * np.random.normal(size=S.shape)  # Add noise
>>> S /= S.std(axis=0)  # Standardize data
>>> # Mix data
>>> A = np.array([[1, 1], [0.5, 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
```

2.5 Putting it all together

2.5.1 Pipelining

We have seen that some estimators can transform data and that some estimators can predict variables. We can also



create combined estimators:

```
from sklearn import linear_model, decomposition, datasets
from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV

logistic = linear_model.LogisticRegression()

pca = decomposition.PCA()
pipe = Pipeline(steps=[('pca', pca), ('logistic', logistic)])

digits = datasets.load_digits()
X_digits = digits.data
y_digits = digits.target

###############################################################################
# Plot the PCA spectrum
pca.fit(X_digits)

plt.figure(1, figsize=(4, 3))
plt.clf()
plt.axes([.2, .2, .7, .7])
plt.plot(pca.explained_variance_, linewidth=2)
plt.axis('tight')
plt.xlabel('n_components')
plt.ylabel('explained_variance_')

###############################################################################
# Prediction

n_components = [20, 40, 64]
Cs = np.logspace(-4, 4, 3)

#Parameters of pipelines can be set using '__' separated parameter names:

estimator = GridSearchCV(pipe,
                         dict(pca__n_components=n_components,
                              logistic__C=Cs))
estimator.fit(X_digits, y_digits)
```

```
plt.axvline(estimator.best_estimator_.named_steps['pca'].n_components,
            linestyle=':', label='n_components chosen')
plt.legend(prop=dict(size=12))
```

2.5.2 Face recognition with eigenfaces

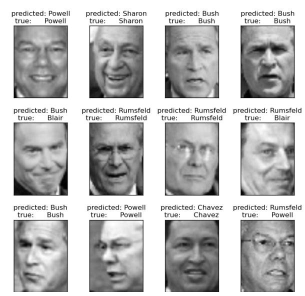
The dataset used in this example is a preprocessed excerpt of the “Labeled Faces in the Wild”, also known as LFW:

> http://vis-www.cs.umass.edu/lfw/lfw-funneled.tgz (233MB)

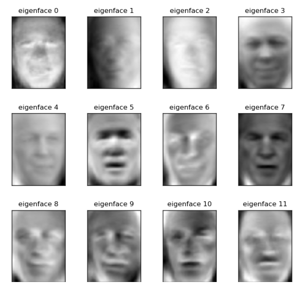
```
"""
===================================================
Faces recognition example using eigenfaces and SVMs
===================================================

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

  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



"""
from __future__ import print_function

from time import time
import logging
import matplotlib.pyplot as plt

from sklearn.cross_validation import train_test_split
from sklearn.datasets import fetch_lfw_people
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.decomposition import RandomizedPCA
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)


###############################################################################
# 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 = RandomizedPCA(n_components=n_components, 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='auto'), 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)
                     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()
```



Prediction



Eigenfaces

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
```

2.5.3 Open problem: Stock Market Structure

Can we predict the variation in stock prices for Google over a given time frame?

Learning a graph structure

2.6 Finding help

2.6.1 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

2.6.2 Q&A communities with Machine Learning practitioners

Metaoptimize/QA A forum for Machine Learning, Natural Language Processing and other Data Analytics discussions (similar to what Stackoverflow is for developers): http://metaoptimize.com/qa

A good starting point is the discussion on good freely available textbooks on machine learning

Quora.com Quora has a topic for Machine Learning related questions that also features some interesting discussions: http://quora.com/Machine-Learning

Have a look at the best questions section, eg: What are some good resources for learning about machine learning.

– _'An excellent free online course for Machine Learning taught by Professor Andrew Ng of Stanford': https://www.coursera.org/course/ml

– _'Another excellent free online course that takes a more general approach to Artificial Intelligence':http://www.udacity.com/overview/Course/cs271/CourseRev/1

CHAPTER

THREE

WORKING WITH TEXT DATA

The goal of this guide is to explore some of the main `scikit-learn` tools on a single practical task: analysing 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

3.1 Tutorial setup

To get started with this tutorial, you firstly must have the *scikit-learn* and all of its required dependencies installed.

Please refer to the *installation instructions* page for more information and for per-system instructions.

The source of this tutorial can be found within your scikit-learn folder:

```
scikit-learn/doc/tutorial/text_analytics/
```

The tutorial folder, should contain the following 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
```

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 web-site and use the `sklearn.datasets.load_files` function by pointing it to the `20news-bydate-train` subfolder 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?
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 can notice that the samples have been shuffled randomly (with a fixed RNG seed): this is useful if you select only the first samples to quickly train a model and get a first idea of the results before re-training on the complete dataset later.

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.

3.3.1 Bags of words

The most intuitive way to do so is the 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 that 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 couple thousands of 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.

3.3.2 Tokenizing text with `scikit-learn`

Text preprocessing, tokenizing and filtering of stopwords are included in a high level component that is able to build a dictionary of features and transform 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 consequective 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.

3.3.3 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:

```
>>> 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)
```

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
```

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 shall see their use in the section on grid search, below. We can now train the model with a single command:

```
>>> text_clf = text_clf.fit(twenty_train.data, twenty_train.target)
```

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)
0.834...
```

I.e., we achieved 83.4% 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 just 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, n_iter=5, random_state=42)),
... ])
>>> _ = text_clf.fit(twenty_train.data, twenty_train.target)
>>> predicted = text_clf.predict(docs_test)
>>> np.mean(predicted == twenty_test.target)
0.912...
```

`scikit-learn` further provides 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      0.81      0.87       319
         comp.graphics       0.88      0.97      0.92       389
               sci.med       0.94      0.90      0.92       396
soc.religion.christian       0.90      0.95      0.93       398

           avg / total       0.92      0.91      0.91      1502


>>> metrics.confusion_matrix(twenty_test.target, predicted)
array([[258,  11,  15,  35],
       [  4, 379,   3,   3],
       [  5,  33, 355,   3],
       [  5,  10,   4, 379]])
```

As expected the confusion matrix shows that posts from the newsgroups on atheism and christian are more often confused for one another than with computer graphics.

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.grid_search 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 uses them all:

```
>>> gs_clf = GridSearchCV(text_clf, parameters, 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'])]
'soc.religion.christian'
```

but otherwise, it's a pretty large and clumsy object. We can, however, get the optimal parameters out by inspecting the object's `grid_scores_` attribute, which is a list of parameters/score pairs. To get the best scoring attributes, we can do:

```
>>> best_parameters, score, _ = max(gs_clf.grid_scores_, key=lambda x: x[1])
>>> for param_name in sorted(parameters.keys()):
...     print("%s: %r" % (param_name, best_parameters[param_name]))
...
clf__alpha: 0.001
tfidf__use_idf: True
vect__ngram_range: (1, 1)

>>> score
0.900...
```

3.7.1 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 loosing 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 accurracy of the model.

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/
```

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/
```

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.

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`.

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

FOUR

SUPERVISED LEARNING

4.1 Generalized 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 input variables. In mathematical notion, if $\hat{y}$ is the predicted value.

$$\hat{y}(w, x) = w_0 + w_1 x_1 + ... + w_p x_p$$

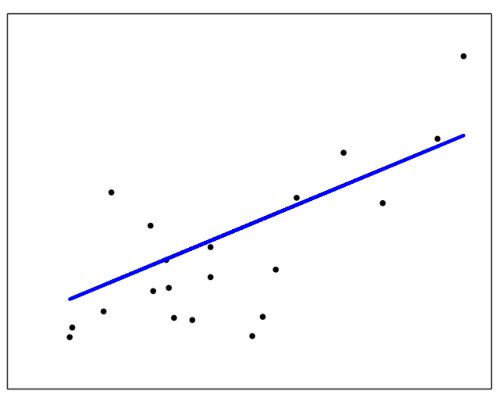
Across the module, we designate the vector $w = (w_1, ..., w_p)$ as `coef_` and w_0 as `intercept_`.

To perform classification with generalized linear models, see *Logistic regression*.

4.1.1 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 responses in the dataset, and the responses predicted by the linear approximation. Mathematically it solves a problem of the form:

$$\underset{w}{min\,} {|| X w - y||_2}^2$$



`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
>>> clf = linear_model.LinearRegression()
>>> clf.fit ([[0, 0], [1, 1], [2, 2]], [0, 1, 2])
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
>>> clf.coef_
array([ 0.5,  0.5])
```

However, coefficient estimates for Ordinary Least Squares rely on the independence of the model terms. When terms 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 response, 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

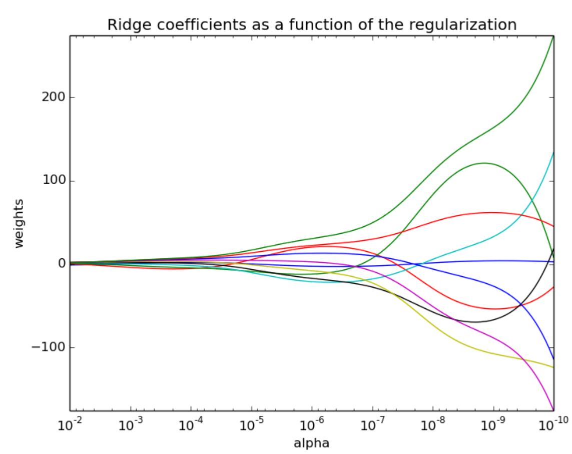
This method computes the least squares solution using a singular value decomposition of X. If X is a matrix of size (n, p) this method has a cost of $O(np^2)$, assuming that $n \geq p$.

4.1.2 Ridge Regression

`Ridge` regression addresses some of the problems of *Ordinary Least Squares* by imposing a penalty on the size of coefficients. The ridge coefficients minimize a penalized residual sum of squares,

$$\underset{w}{min}\,{{|| X w - y||_2}^2 + \alpha {||w||_2}^2}$$

Here, $\alpha \geq 0$ is a complexity parameter that 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
>>> clf = linear_model.Ridge (alpha = .5)
>>> clf.fit ([[0, 0], [0, 0], [1, 1]], [0, .1, 1])
Ridge(alpha=0.5, copy_X=True, fit_intercept=True, max_iter=None,
      normalize=False, solver='auto', tol=0.001)
>>> clf.coef_
array([ 0.34545455,  0.34545455])
>>> clf.intercept_
0.13636...
```

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 than an *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:

```
>>> from sklearn import linear_model
>>> clf = linear_model.RidgeCV(alphas=[0.1, 1.0, 10.0])
>>> clf.fit([[0, 0], [0, 0], [1, 1]], [0, .1, 1])
RidgeCV(alphas=[0.1, 1.0, 10.0], cv=None, fit_intercept=True, scoring=None,
    normalize=False)
>>> clf.alpha_
0.1
```

References

- "Notes on Regularized Least Squares", Rifkin & Lippert (technical report, course slides).

4.1.3 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 parameter values, effectively reducing the number of variables upon which the given solution is dependent. For this reason, the Lasso and its variants are fundamental to the field of compressed sensing. Under certain conditions, it can recover the exact set of non-zero weights (see *Compressive sensing: tomography reconstruction with L1 prior (Lasso)*).

Mathematically, it consists of a linear model trained with ℓ_1 prior as regularizer. The objective function to minimize is:

$$\underset{w}{min\,} { \frac{1}{2n_{samples}} ||X w - y||_2 ^ 2 + \alpha ||w||_1}$$

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 parameter vector.

The implementation in the class `Lasso` uses coordinate descent as the algorithm to fit the coefficients. See *Least Angle Regression* for another implementation:

```
>>> clf = linear_model.Lasso(alpha = 0.1)
>>> clf.fit([[0, 0], [1, 1]], [0, 1])
Lasso(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=None,
   selection='cyclic', tol=0.0001, warm_start=False)
>>> clf.predict([[1, 1]])
array([ 0.8])
```

Also useful for lower-level tasks is the function `lasso_path` that 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*.

Note: Randomized sparsity

For feature selection or sparse recovery, it may be interesting to use *Randomized sparse models*.

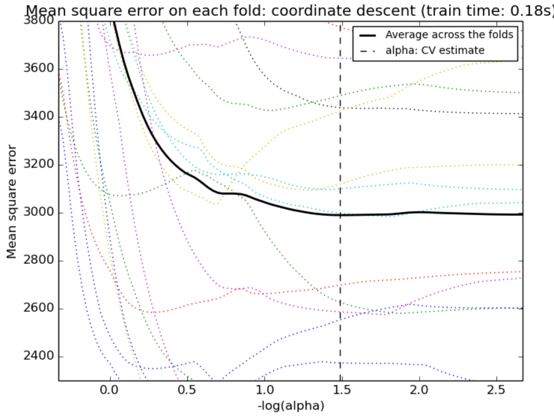
Setting regularization parameter

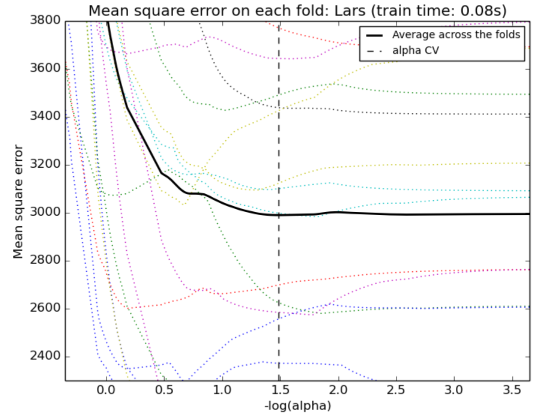
The `alpha` parameter controls the degree of sparsity of the coefficients estimated.

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.

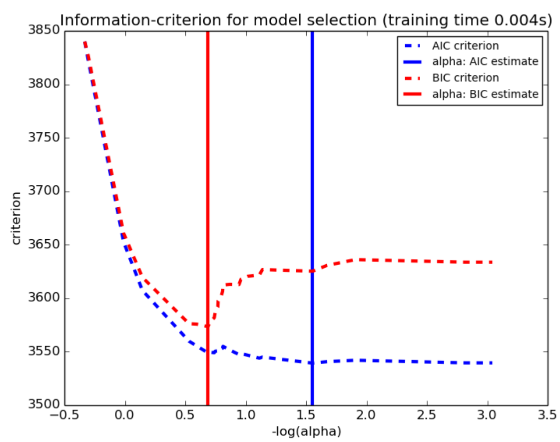
For high-dimensional datasets with many collinear regressors, `LassoCV` is most often preferable. However, `LassoLarsCV` has the advantage of exploring more relevant values of *alpha* parameter, and if the number of samples is very small compared to the number of observations, it is often faster than `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*

4.1.4 Elastic Net

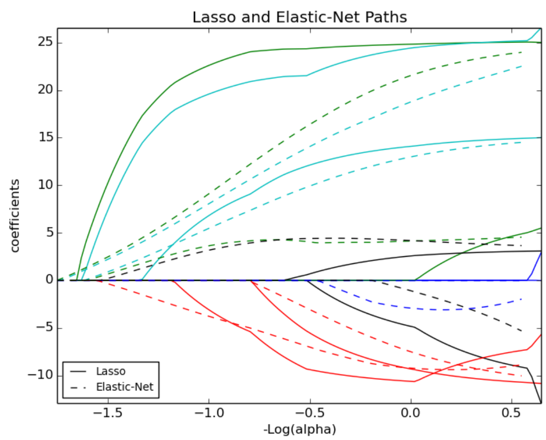
ElasticNet is a linear regression model trained with L1 and L2 prior as regularizer. This combination allows for learning a sparse model where few of the weights are non-zero like Lasso, while still maintaining the regularization properties of Ridge. We control the convex combination of L1 and L2 using 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 it allows Elastic-Net to inherit some of Ridge's stability under rotation.

The objective function to minimize is in this case

$$\underset{w}{min\,} { \frac{1}{2n_{samples}} ||X w - y||_2 ^ 2 + \alpha \rho ||w||_1 + \frac{\alpha(1-\rho)}{2} ||w||_2 ^ 2}$$



The class `ElasticNetCV` can be used to set the parameters `alpha` (α) and `l1_ratio` (ρ) by cross-validation.

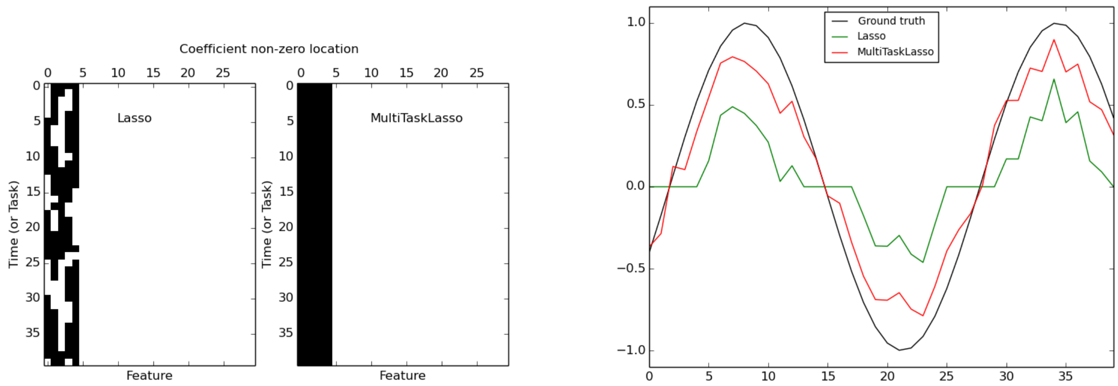
Examples:

- *Lasso and Elastic Net for Sparse Signals*
- *Lasso and Elastic Net*

4.1.5 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-zeros in W obtained with a simple Lasso or a MultiTaskLasso. The Lasso estimates yields 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 prior as regularizer. The objective function to minimize is:

$$\underset{w}{min} \frac{1}{2n_{samples}} ||XW - Y||_2^2 + \alpha ||W||_{21}$$

where;

$$||W||_{21} = \sum_i \sqrt{\sum_j w_{ij}^2}$$

The implementation in the class `MultiTaskLasso` uses coordinate descent as the algorithm to fit the coefficients.

4.1.6 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.

The advantages of LARS are:

- It is numerically efficient in contexts where p >> n (i.e., when the number of dimensions is significantly greater than the number of points)
- It is computationally just as fast as forward selection and has the same order of complexity as an 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 variables are almost equally correlated with the response, 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.

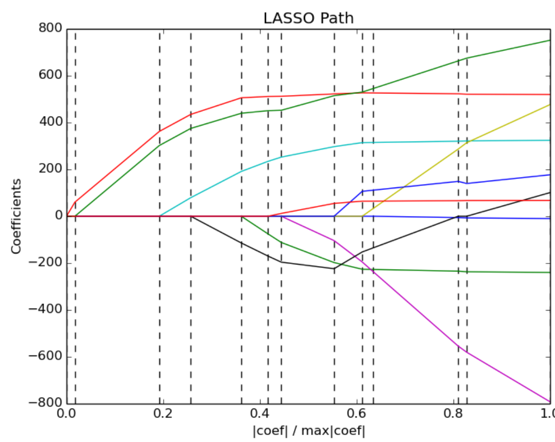
The disadvantages of the LARS method include:

- 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`.

4.1.7 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
>>> clf = linear_model.LassoLars(alpha=.1)
>>> clf.fit([[0, 0], [1, 1]], [0, 1])
LassoLars(alpha=0.1, copy_X=True, eps=..., fit_intercept=True,
     fit_path=True, max_iter=500, normalize=True, precompute='auto',
     verbose=False)
>>> clf.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 consist of retrieving the path with function `lars_path`

Mathematical formulation

The algorithm is similar to forward stepwise regression, but instead of including variables at each step, the estimated parameters 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 L1 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.

4.1.8 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:

$$\text{arg min} \, ||y - X\gamma||_2^2 \text{ subject to } \, ||\gamma||_0 \leq n_{nonzero\_coefs}$$

Alternatively, orthogonal matching pursuit can target a specific error instead of a specific number of non-zero coefficients. This can be expressed as:

$$\text{arg min} \, ||\gamma||_0 \text{ subject to } ||y - X\gamma||_2^2 \, \leq \text{tol}$$

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:

- http://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf
- Matching pursuits with time-frequency dictionaries, S. G. Mallat, Z. Zhang,

4.1.9 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 is equivalent to finding a maximum a-postiori solution under a Gaussian prior over the parameters 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:

$$p(y|X, w, \alpha) = \mathcal{N}(y|Xw, \alpha)$$

Alpha 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

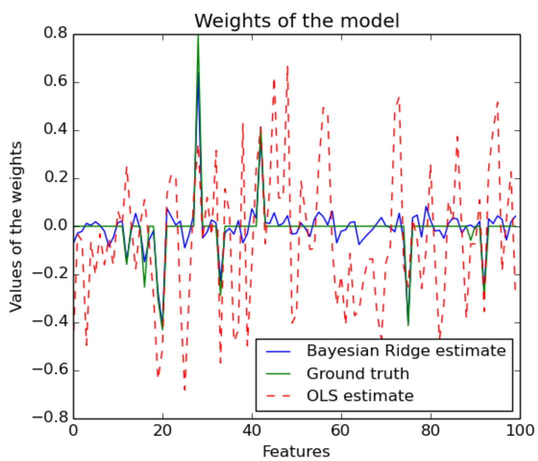
`BayesianRidge` estimates a probabilistic model of the regression problem as described above. The prior for the parameter w is given by a spherical Gaussian:

$$p(w|\lambda) = \mathcal{N}(w|0, \lambda^{-1}\mathbf{I_p})$$

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 remaining hyperparameters are the parameters of the gamma priors over α and λ. These are usually chosen to be *non-informative*. The parameters are estimated by maximizing the *marginal log likelihood*.

By default $\alpha_1 = \alpha_2 = \lambda_1 = \lambda_2 = 1.e^{-6}$.



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.]
>>> clf = linear_model.BayesianRidge()
>>> clf.fit(X, Y)
BayesianRidge(alpha_1=1e-06, alpha_2=1e-06, compute_score=False, copy_X=True,
       fit_intercept=True, lambda_1=1e-06, lambda_2=1e-06, n_iter=300,
       normalize=False, tol=0.001, verbose=False)
```

After being fitted, the model can then be used to predict new values:

```
>>> clf.predict ([[1, 0.]])
array([ 0.50000013])
```

The weights w of the model can be access:

```
>>> clf.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 problem.

Examples:

- *Bayesian Ridge Regression*

References

- More details can be found in the article Bayesian Interpolation by MacKay, David J. C.

Automatic Relevance Determination - ARD

`ARDRegression` is very similar to Bayesian Ridge Regression, but can lead to sparser weights w [1] [2]. `ARDRegression` poses a different prior over w, by dropping the assumption of the Gaussian being spherical.

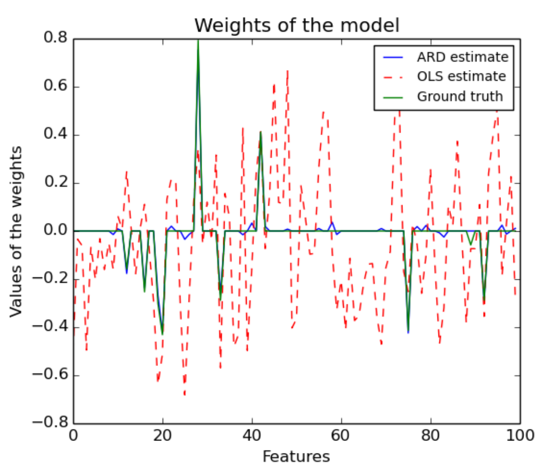
Instead, the distribution over w is assumed to be an axis-parallel, elliptical Gaussian distribution.

This means each weight 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 $diag\ (A) = \lambda = \{\lambda_1, ..., \lambda_p\}$.

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.



[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.

Examples:

- *Automatic Relevance Determination Regression (ARD)*

References:

4.1.10 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.

The implementation of logistic regression in scikit-learn can be accessed from class `LogisticRegression`. This implementation can fit a multiclass (one-vs-rest) logistic regression with optional L2 or L1 regularization.

As an optimization problem, binary class L2 penalized logistic regression minimizes the following cost function:

$$\underset{w, c}{min\,} \frac{1}{2}w^T w + C \sum_{i=1}^n \log(\exp(- y_i (X_i^T w + c)) + 1) .$$

Similarly, L1 regularized logistic regression solves the following optimization problem

$$\underset{w, c}{min\,} \|w\|_1 + C \sum_{i=1}^n \log(\exp(- y_i (X_i^T w + c)) + 1) .$$

The solvers implemented in the class `LogisticRegression` are "liblinear" (which is a wrapper around the C++ library, LIBLINEAR), "newton-cg" and "lbfgs".

The lbfgs and newton-cg solvers only support L2 penalization and are found to converge faster for some high dimensional data. L1 penalization yields sparse predicting weights.

Several estimators are available for logistic regression.

`LogisticRegression` has an option of using three solvers, "liblinear", "lbfgs" and "newton-cg".

The solver "liblinear" uses a coordinate descent (CD) algorithm based on Liblinear. For L1 penalization `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. This relies on the excellent 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.

Setting *multi_class* to "multinomial" with the "lbfgs" or "newton-cg" solver in `LogisticRegression` learns a true multinomial logistic regression model, which means that its probability estimates should be better calibrated than the default "one-vs-rest" setting. L-BFGS and newton-cg cannot optimize L1-penalized models, though, so the "multinomial" setting does not learn sparse models.

Examples:

- *L1 Penalty and Sparsity in Logistic Regression*
- *Path with L1- Logistic Regression*

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 L1 penalty yields sparse models, and can thus be used to perform feature selection, as detailed in *L1-based feature selection*.

`LogisticRegressionCV` implements Logistic Regression with builtin cross-validation to find out the optimal C parameter. In general the "newton-cg" and "lbfgs" solvers are found to be faster due to warm-starting. For the multiclass case, if *multi_class* option is set to "ovr", an optimal C is obtained for each class and if the *multi_class* option is set to "multinomial", an optimal C is obtained that minimizes the cross-entropy loss.

4.1.11 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 only/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*

4.1.12 Perceptron

The `Perceptron` is another simple 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.

4.1.13 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`.

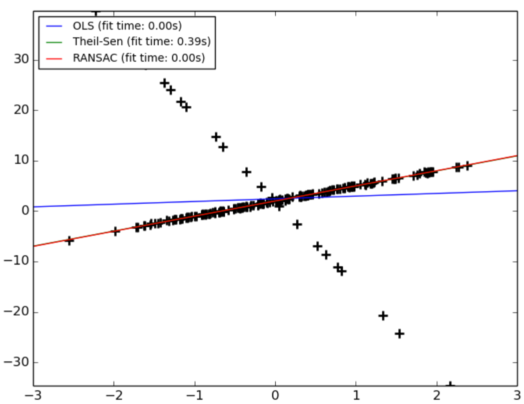
For classification, `PassiveAggressiveClassifier` can be used with `loss='hinge'` (PA-I) or `loss='squared_hinge'` (PA-II). For regression, `PassiveAggressiveRegressor` can be used with `loss='epsilon_insensitive'` (PA-I) or `loss='squared_epsilon_insensitive'` (PA-II).

References:

- "Online Passive-Aggressive Algorithms" K. Crammer, O. Dekel, J. Keshat, S. Shalev-Shwartz, Y. Singer - JMLR 7 (2006)

4.1.14 Robustness regression: outliers and modeling errors

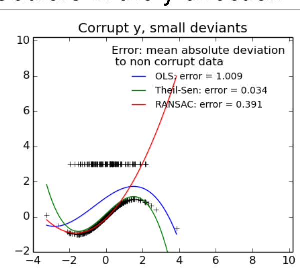
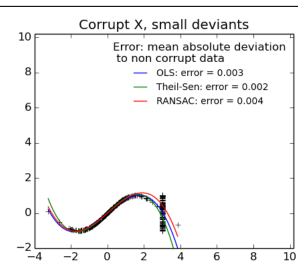
Robust regression is interested in fitting a regression model in the presence of corrupt data: either outliers, or error in the model.



Different scenario and useful concepts

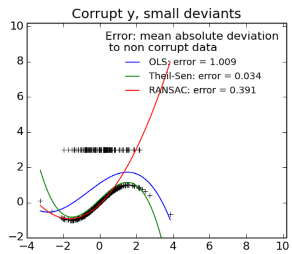
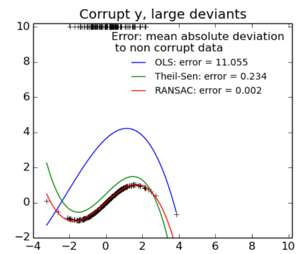
There are different things to keep in mind when dealing with data corrupted by outliers:

- **Outliers in X or in y**?

<table>
<tr><th>Outliers in the y direction</th><th>Outliers in the X direction</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

- **Fraction of outliers versus amplitude of error**

  The number of outlying points matters, but also how much they are outliers.

<table>
<tr><th>Small outliers</th><th>Large outliers</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

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 2 robust regression estimators: *RANSAC* and *Theil Sen*

- *RANSAC* is faster, 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 large dimensional settings.

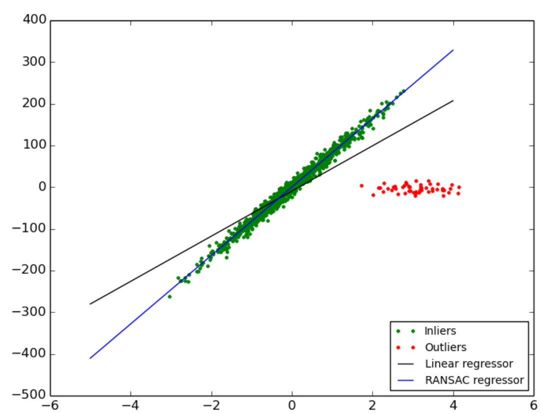
When in doubt, use *RANSAC*

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 fields 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:

- http://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 looses its robustness properties and becomes no better than an ordinary least squares in high dimension.

Examples:

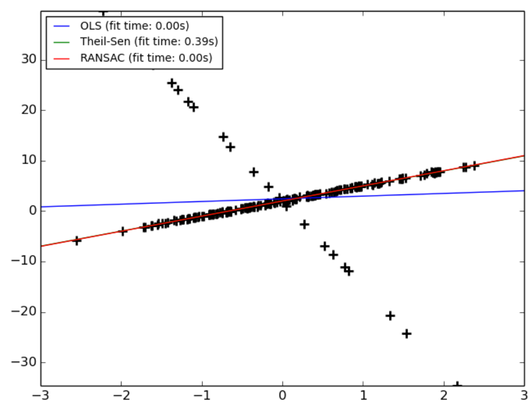
- *Theil-Sen Regression*
- *Robust linear estimator fitting*

References:

- http://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 [3] using the spatial median which is a generalization of the median to multiple dimensions [4].

In terms of time and space complexity, Theil-Sen scales according to

$$\binom{n_{samples}}{n_{subsamples}}$$

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:

4.1.15 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.

[3] Xin Dang, Hanxiang Peng, Xueqin Wang and Heping Zhang: Theil-Sen Estimators in a Multiple Linear Regression Model.

[4]

20. Kärkkäinen and S. Äyrämö: On Computation of Spatial Median for Robust Data Mining.

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 variable

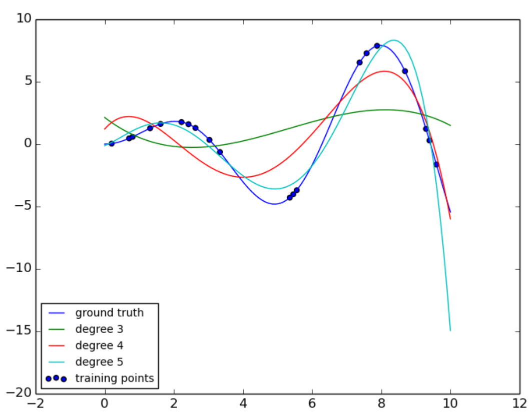
$$z = [x_1, x_2, x_1 x_2, x_1^2, x_2^2]$$

With this re-labeling of the data, our problem can be written

$$\hat{y}(w, x) = w_0 + w_1 z_1 + w_2 z_2 + w_3 z_3 + w_4 z_4 + w_5 z_5$$

We see that the resulting *polynomial regression* is in the same class of linear models we'd 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` preprocessor. This preprocessor 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 $[1, x_1, x_2, x_1^2, x_1 x_2, x_2^2]$, 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
>>> 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`.

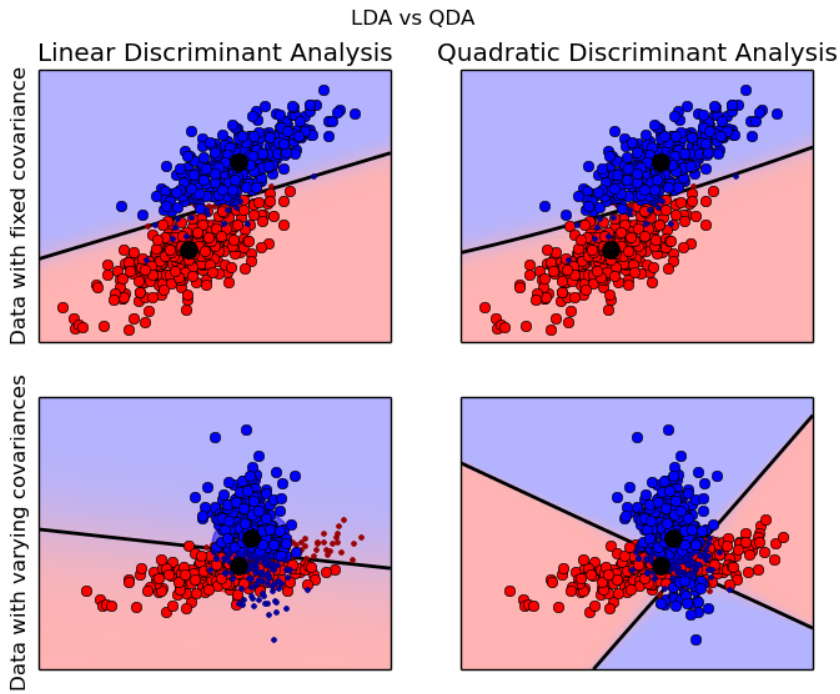
For example, when dealing with boolean features, $x_i^n = x_i$ for all n and is therefore useless; but $x_i x_j$ represents the conjunction of two booleans. This way, we can solve the XOR problem with a linear classifier:

```
>>> from sklearn.linear_model import Perceptron
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
>>> y = X[:, 0] ^ X[:, 1]
>>> X = PolynomialFeatures(interaction_only=True).fit_transform(X)
>>> X
array([[1, 0, 0, 0],
       [1, 0, 1, 0],
       [1, 1, 0, 0],
       [1, 1, 1, 1]])
>>> clf = Perceptron(fit_intercept=False, n_iter=10, shuffle=False).fit(X, y)
>>> clf.score(X, y)
1.0
```

4.2 Linear and quadratic discriminant analysis

Linear discriminant analysis (`lda.LDA`) and quadratic discriminant analysis (`qda.QDA`) 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, and have proven to work well in practice. Also there are no parameters to tune for these algorithms.



The plot shows decision boundaries for LDA and QDA. The bottom row demonstrates that LDA can only learn linear boundaries, while QDA can learn quadratic boundaries and is therefore more flexible.

Examples:

Linear and Quadratic Discriminant Analysis with confidence ellipsoid: Comparison of LDA and QDA on synthetic data.

4.2.1 Dimensionality reduction using LDA

`lda.LDA` can be used to perform supervised dimensionality reduction by projecting the input data to a subspace consisting of the most discriminant directions. This is implemented in `lda.LDA.transform`. The desired dimensionality can be set using the `n_components` constructor parameter. This parameter has no influence on `lda.LDA.fit` or `lda.LDA.predict`.

4.2.2 Mathematical Idea

Both methods work by modeling the class conditional distribution of the data $P(X|y = k)$ for each class k. Predictions can be obtained by using Bayes' rule:

$$P(y|X) = P(X|y) \cdot P(y)/P(X) = P(X|y) \cdot P(Y)/(\sum_{y'} P(X|y') \cdot p(y'))$$

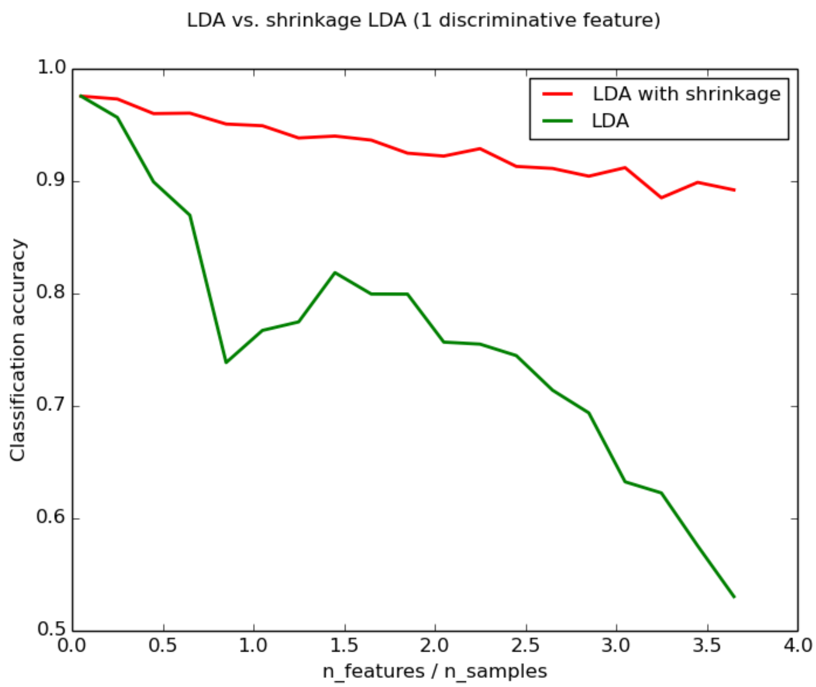
In linear and quadratic discriminant analysis, $P(X|y)$ is modelled as a Gaussian distribution. In the case of LDA, the Gaussians for each class are assumed to share the same covariance matrix. This leads to a linear decision surface, as can be seen by comparing the the log-probability rations $log[P(y = k|X)/P(y = l|X)]$.

In the case of QDA, there are no assumptions on the covariance matrices of the Gaussians, leading to a quadratic decision surface.

4.2.3 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 `lda.LDA` class to 'auto'. This automatically determines the optimal shrinkage parameter in an analytic way following the lemma introduced by Ledoit and Wolf. 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.



4.2.4 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:

Hastie T, Tibshirani R, Friedman J. The Elements of Statistical Learning. Springer, 2009.
Ledoit O, Wolf M. Honey, I Shrunk the Sample Covariance Matrix. The Journal of Portfolio Management 30(4), 110-119, 2004.

4.3 Kernel ridge regression

Kernel ridge regression (KRR) [M2012] 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 `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:

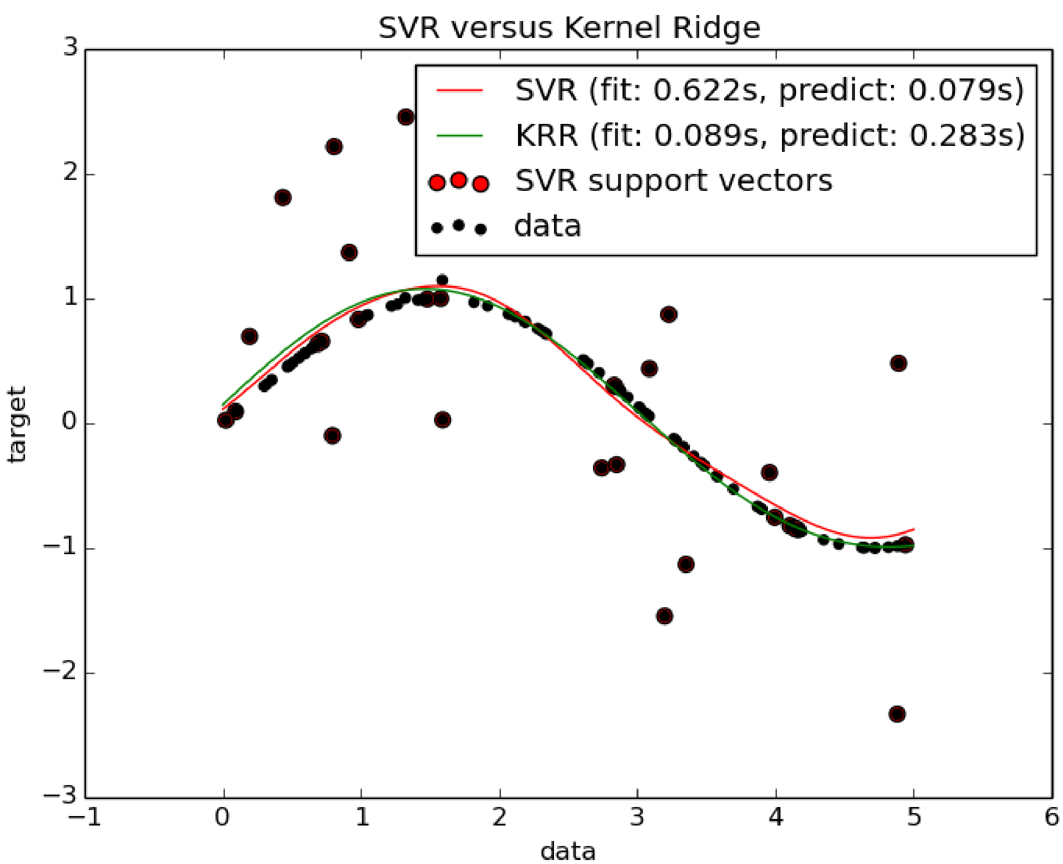
4.4 Support Vector Machines

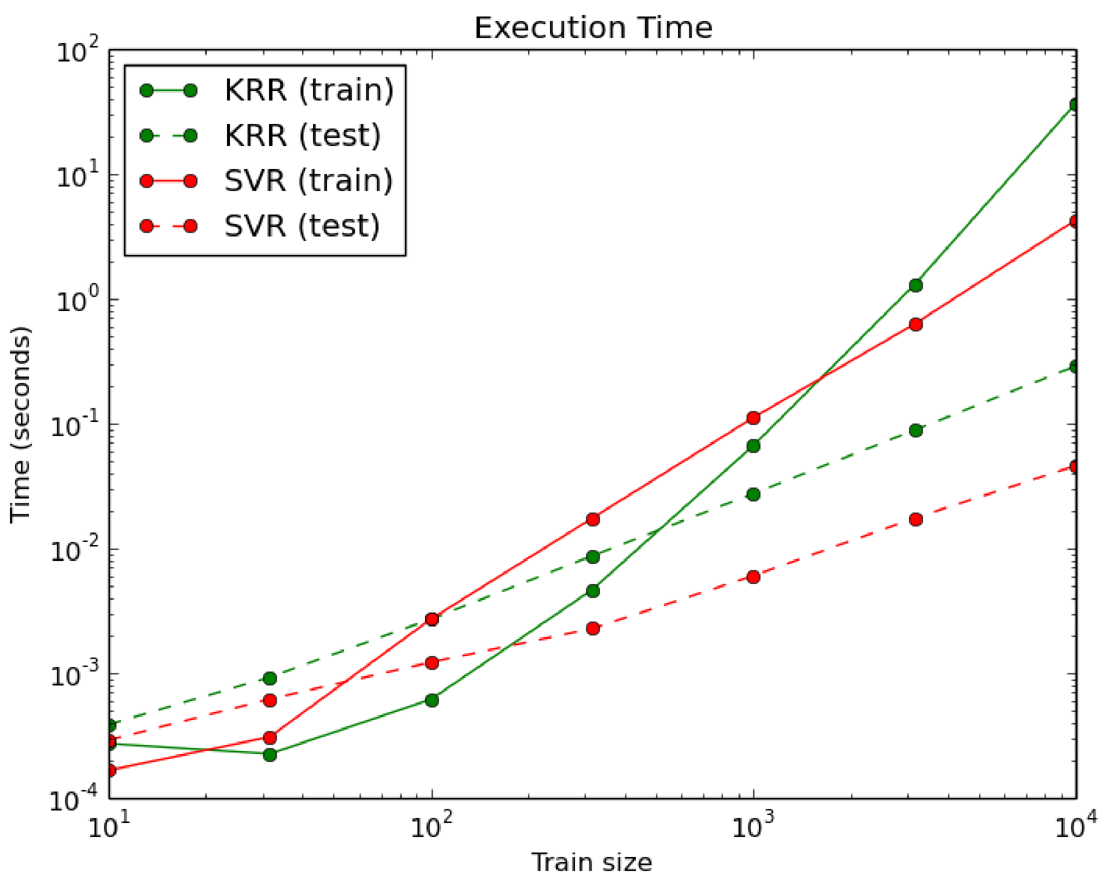
Support vector machines (SVMs) are a set of supervised learning methods used for *classification*, *regression* and *outliers detection*.

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:


SVR versus Kernel Ridge
SVR (fit: 0.622s, predict: 0.079s)
KRR (fit: 0.089s, predict: 0.283s)
SVR support vectors
data
target
data
3
2
1
0
−1
−2
−3
−1
0
1
2
3
4
5
6

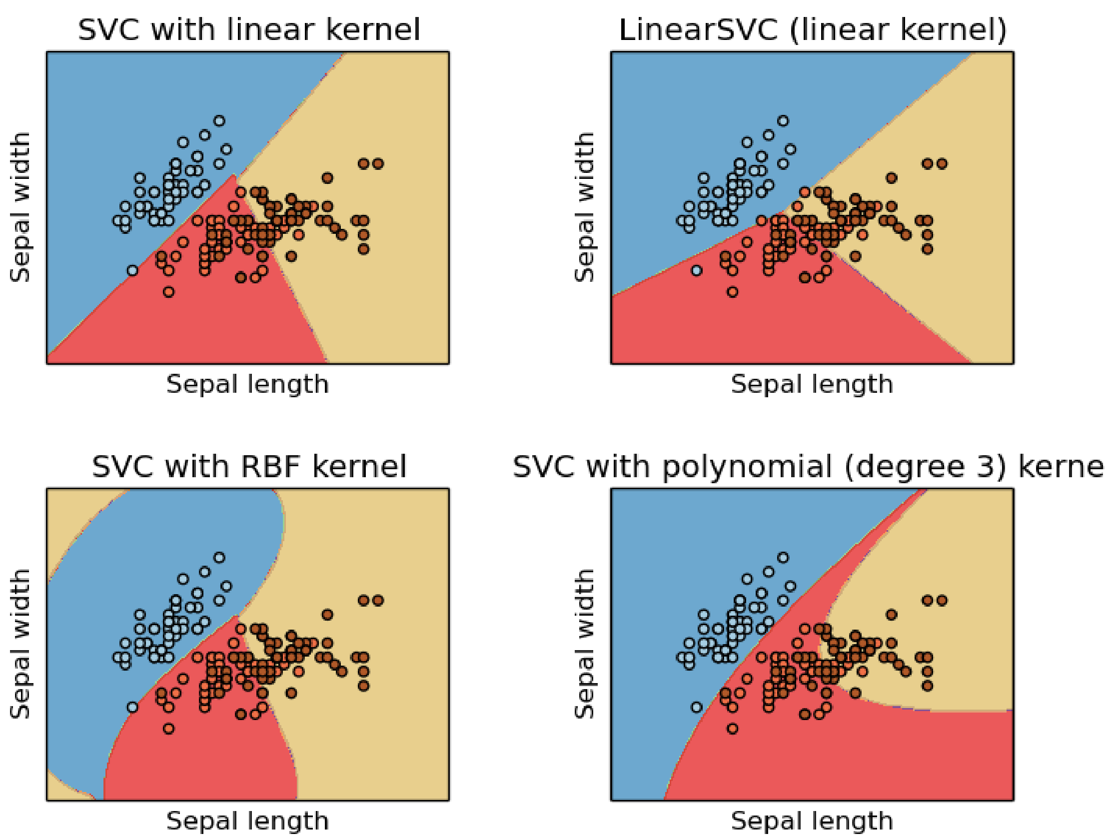

Execution Time
KRR (train)
KRR (test)
SVR (train)
SVR (test)
Time (seconds)
Train size
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{1}
10^{2}
10^{3}
10^{4}

- If the number of features is much greater than the number of samples, the method is likely to give poor performances.
- 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`.

4.4.1 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(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
gamma=0.0, kernel='rbf', max_iter=-1, probability=False, random_state=None,
shrinking=True, tol=0.001, verbose=False)
```

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:

```
>>> X = [[0], [1], [2], [3]]
>>> Y = [0, 1, 2, 3]
>>> clf = svm.SVC()
>>> clf.fit(X, Y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
gamma=0.0, kernel='rbf', max_iter=-1, probability=False, random_state=None,
shrinking=True, tol=0.001, verbose=False)
>>> dec = clf.decision_function([[1]])
>>> dec.shape[1] # 4 classes: 4*3/2 = 6
6
```

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(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
>>> 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 layout of `coef_` and `intercept_` is similar to the one described for `LinearSVC` described above, except that the shape of `coef_` is `[n_class * (n_class - 1) / 2, n_features]`, corresponding to as many binary classifiers. 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 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:

$\alpha_{0,1}^0$	$\alpha_{0,2}^0$	
$\alpha_{0,1}^1$	$\alpha_{0,2}^1$	Coefficients for SVs of class 0
$\alpha_{0,1}^2$	$\alpha_{0,2}^2$	
$\alpha_{1,0}^0$	$\alpha_{1,2}^0$	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 `SVC` method `decision_function` 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`.

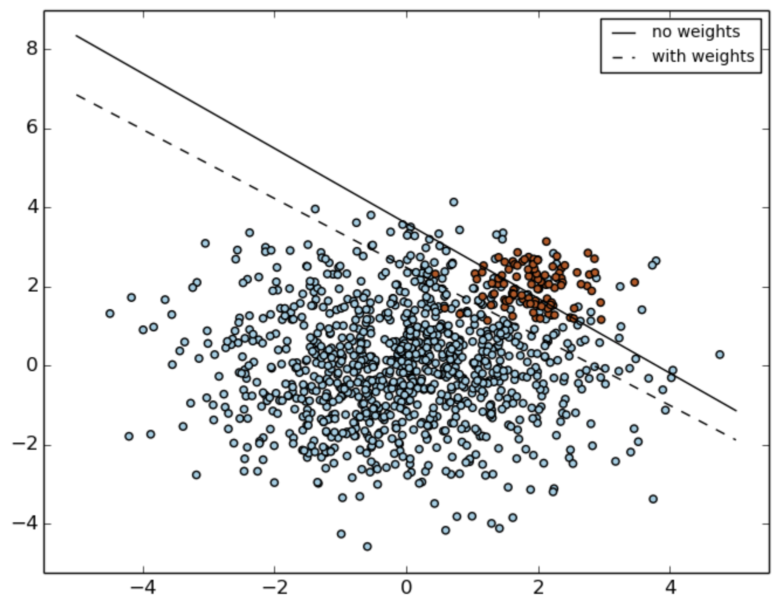
References:

- Wu, Lin and Weng, "Probability estimates for multi-class classification by pairwise coupling". JMLR 5:975-1005, 2004.

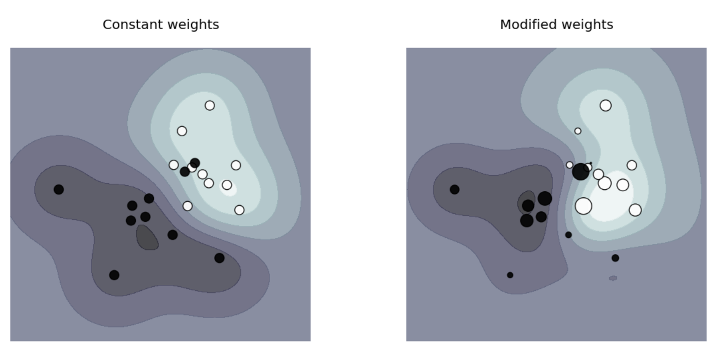
Unbalanced problems

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` 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*,

4.4.2 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.

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(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma=0.0,
    kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
>>> clf.predict([[1, 1]])
array([ 1.5])
```

Examples:

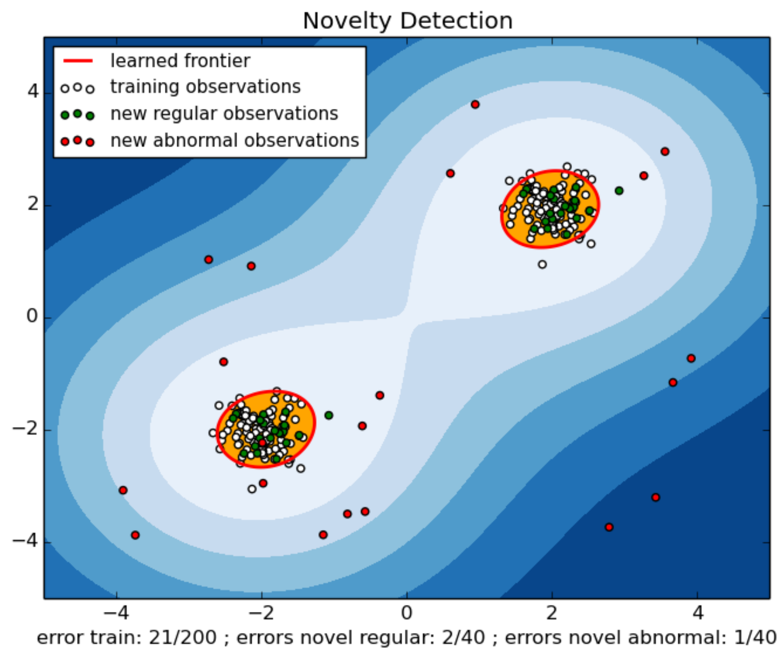
- *Support Vector Regression (SVR) using linear and non-linear kernels*

4.4.3 Density estimation, novelty detection

One-class SVM is used for novelty detection, that is, given a set of samples, it will detect the soft boundary of that set so as to classify new points as belonging to that set or not. The class that implements this is called `OneClassSVM`.

In this case, as it is a type of unsupervised learning, the fit method will only take as input an array X, as there are no class labels.

See, section *Novelty and Outlier Detection* for more details on this usage.



Examples:

- *One-class SVM with non-linear kernel (RBF)*
- *Species distribution modeling*

4.4.4 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.

4.4.5 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 give 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.
- 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='auto'` and/or try different penalty parameters `C`.
- The underlying LinearSVC 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.
- 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`.

4.4.6 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
'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 and return a third matrix.

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(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
gamma=0.0, kernel='precomputed', max_iter=-1, probability=False,
random_state=None, shrinking=True, tol=0.001, verbose=False)
>>> # predict on training examples
>>> 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.

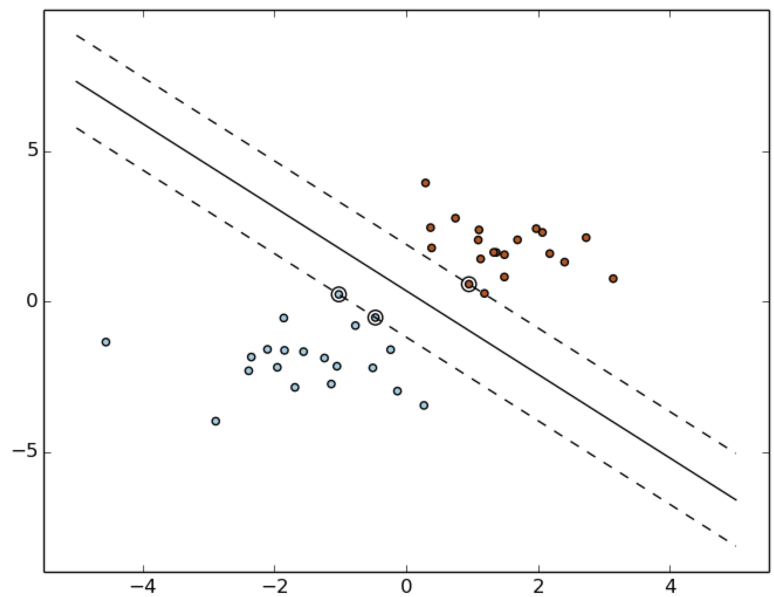
Proper choice of `C` and `gamma` is critical to the SVM's performance. One is advised to use `sklearn.grid_search.GridSearchCV` with `C` and `gamma` spaced exponentially far apart to choose good values.

Examples:

- *RBF SVM parameters*

4.4.7 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, $Q_{ij} \equiv K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the kernel. Here training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function ϕ.

The decision function is:

$$\operatorname{sgn}(\sum_{i=1}^n y_i \alpha_i K(x_i, x) + \rho)$$

Note: While SVM models derived from libsvm and liblinear use `C` as regularization parameter, most other estimators use `alpha`. The relation between both is $C = \frac{n\_samples}{alpha}$.

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 Leaming, 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 reparametrization 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:

$$\begin{aligned}
&\min_{w, b, \zeta, \zeta^*} \frac{1}{2} w^T w + C \sum_{i=1}^{n} (\zeta_i + \zeta_i^*) \\
\textrm{subject to } & y_i - w^T \phi(x_i) - b \leq \varepsilon + \zeta_i, \\
& w^T \phi(x_i) + b - y_i \leq \varepsilon + \zeta_i^*, \\
& \zeta_i, \zeta_i^* \geq 0, i=1, ..., n
\end{aligned}$$

Its dual is

$$\begin{aligned}
&\min_{\alpha, \alpha^*} \frac{1}{2} (\alpha - \alpha^*)^T Q (\alpha - \alpha^*) + \varepsilon e^T (\alpha + \alpha^*) - y^T (\alpha - \alpha^*) \\
\textrm{subject to } & e^T (\alpha - \alpha^*) = 0 \\
& 0 \leq \alpha_i, \alpha_i^* \leq C, i=1, ..., n
\end{aligned}$$

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

4.4.8 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

4.5 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).

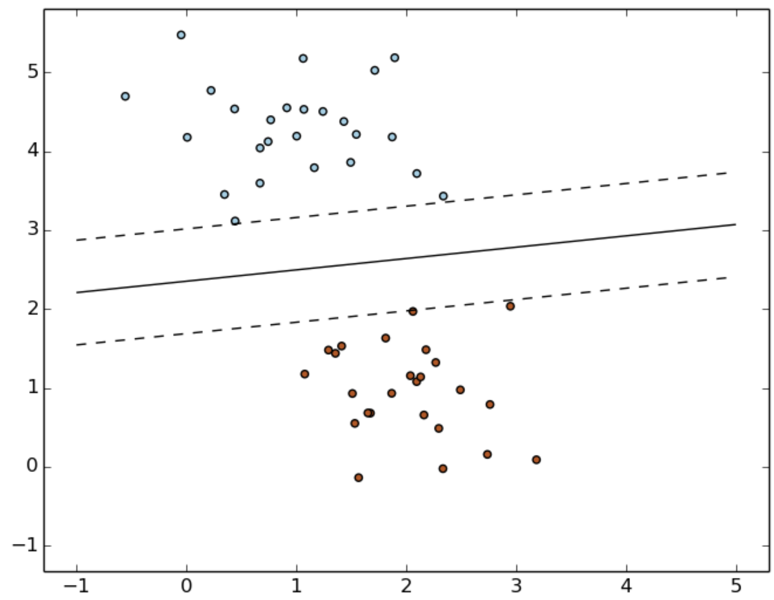
The disadvantages of Stochastic Gradient Descent include:

- SGD requires a number of hyperparameters such as the regularization parameter and the number of iterations.
- SGD is sensitive to feature scaling.

4.5.1 Classification

Warning: Make sure you permute (shuffle) your training data before fitting the model or use `shuffle=True` to shuffle after each iterations.

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.]]
>>> y = [0, 1]
>>> clf = SGDClassifier(loss="hinge", penalty="l2")
>>> clf.fit(X, y)
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', n_iter=5, n_jobs=1,
       penalty='l2', power_t=0.5, random_state=None, shuffle=True,
       verbose=0, warm_start=False)
```

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").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.

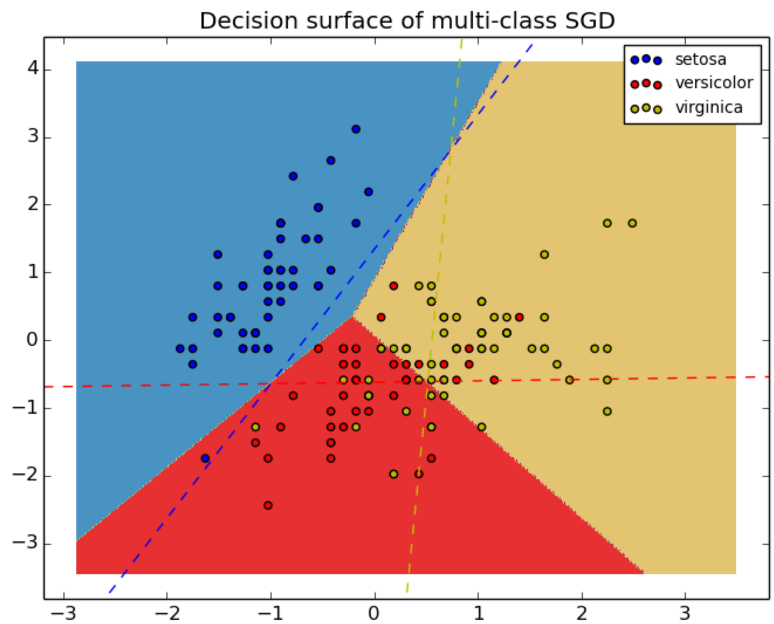
`SGDClassifier` supports multi-class classification by combining multiple binary classifiers in a "one versus all" (OVA) scheme. For each of the K classes, a binary classifier is learned that discriminates between that and all other $K - 1$ classes. At testing time, we compute the confidence score (i.e. 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-dimensionally 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 doc string of `SGDClassifier.fit` for further information.

Examples:

- *SGD: Maximum margin separating hyperplane*,
- *Plot multi-class SGD on the iris dataset*
- *SGD: Weighted samples*
- *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.

4.5.2 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`

4.5.3 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*

4.5.4 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.

4.5.5 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 `n_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.

4.5.6 Mathematical formulation

Given a set of training examples $(x_1, y_1), \ldots, (x_n, y_n)$ where $x_i \in \mathbf{R}^n$ 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

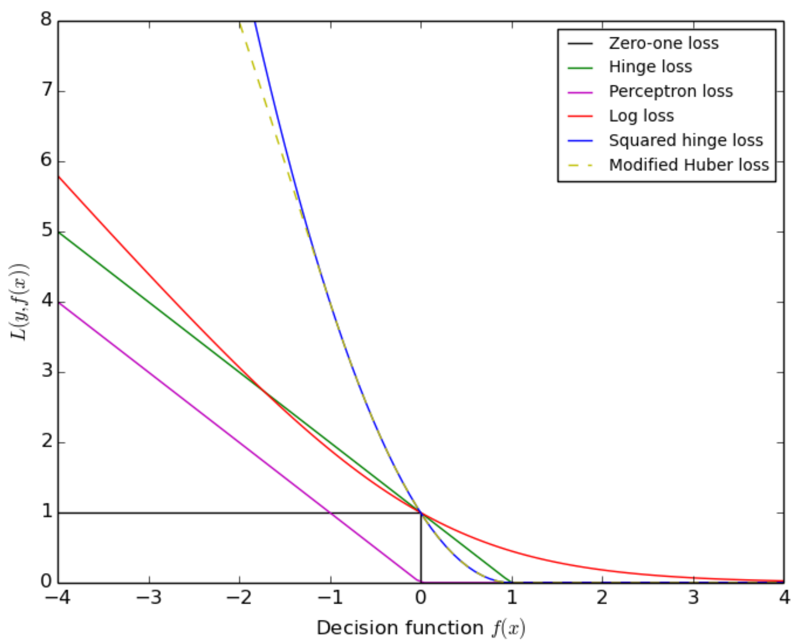
$$E(w,b) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, f(x_i)) + \alpha R(w)$$

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.

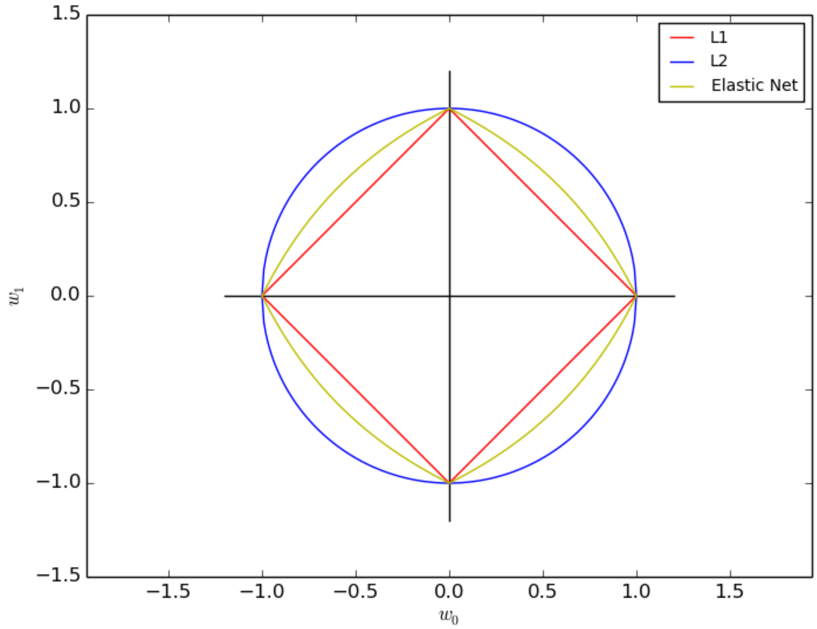
All of the above loss functions can be regarded as an upper bound on the misclassification error (Zero-one loss) as shown in the Figure below.



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.

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.

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

4.5.7 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.

4.6 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 or 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.

4.6.1 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, neighbor $k+1$ and k, have identical distances but different labels, the results 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]])
>>> 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.]])
```

Our 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 neighbors searches, including specification of query strategies, of various distance metrics, etc. For a list of available metrics, see the documentation of the `DistanceMetric` class.

4.6.2 Nearest Neighbors Classification

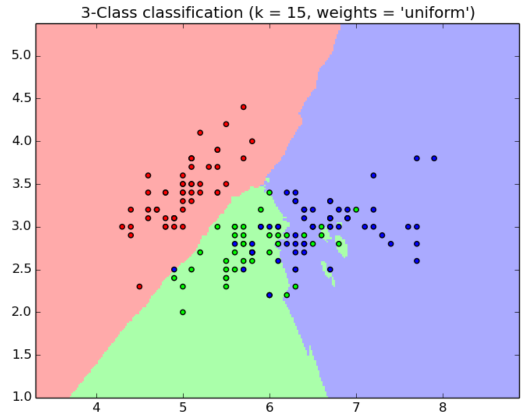
Neighbors-based classification is a type of *instance-based learning* or *non-generalizing learning*: it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point.

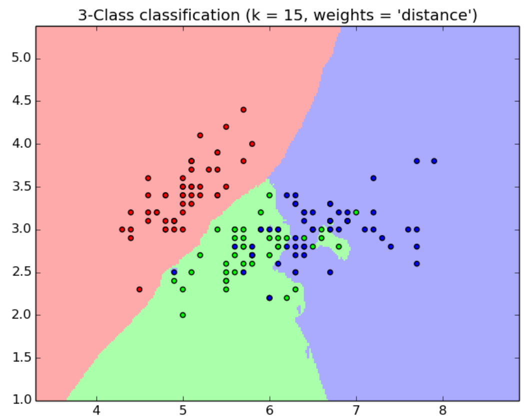
scikit-learn implements two different nearest neighbors classifiers: `KNeighborsClassifier` implements learning based on the k nearest neighbors of each query point, where k is an integer value specified by the user. `RadiusNeighborsClassifier` implements learning based on the number of neighbors within 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 more commonly used of the two techniques. 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 which is used to compute the weights.





Examples:

- *Nearest Neighbors Classification*: an example of classification using nearest neighbors.

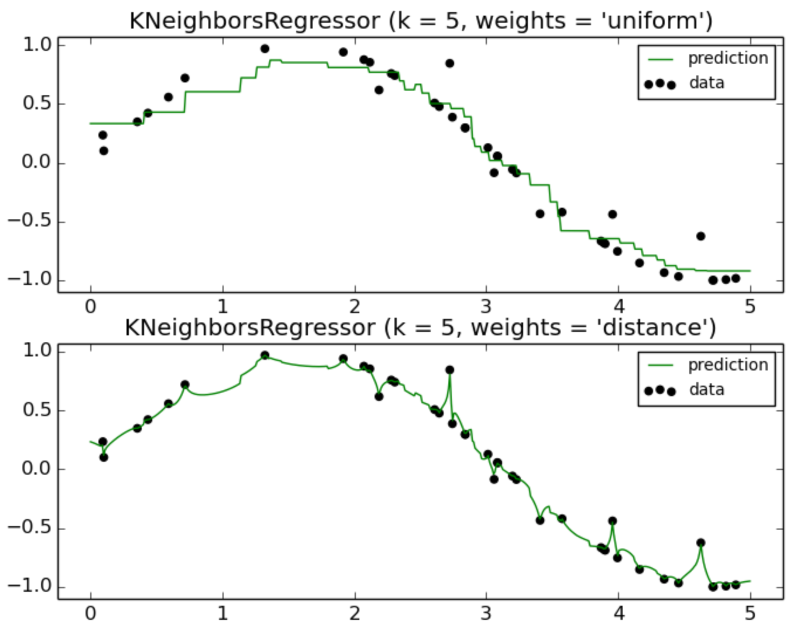
4.6.3 Nearest Neighbors Regression

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 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.

4.6.4 Nearest Neighbor Algorithms

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.


Face completion with multi-output estimators
true faces
Extra trees
K-nn
Linear regression
Ridge

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 orthotopic 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 $d \leq D$ 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 `'kd_tree'` if $k < N/2$ and the `'effective_metric_'` is in the `'VALID_METRICS'` list of `'kd_tree'`. It selects `'ball_tree'` if $k < N/2$ and the `'effective_metric_'` is not in the `'VALID_METRICS'` list of `'kd_tree'`. It selects `'brute'` if $k >= N/2$. 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.

4.6.5 Nearest Centroid Classifier

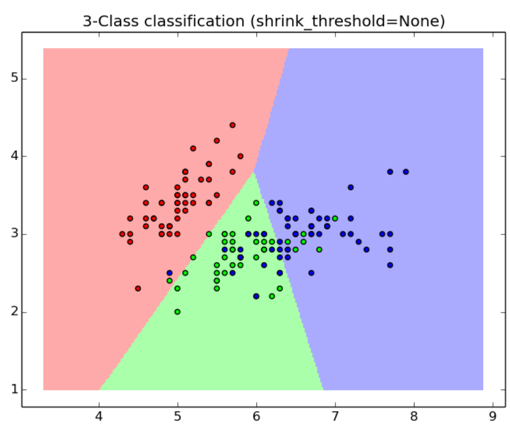
The `NearestCentroid` classifier is a simple algorithm that represents each class by the centroid of its members. In effect, this makes it similar to the label updating phase of the `sklearn.KMeans` algorithm. It also has no parameters to choose, making it a good baseline classifier. It does, however, 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.lda.LDA`) and Quadratic Discriminant Analysis (`sklearn.qda.QDA`) for more complex methods that do not make this assumption. Usage of the default `NearestCentroid` is simple:

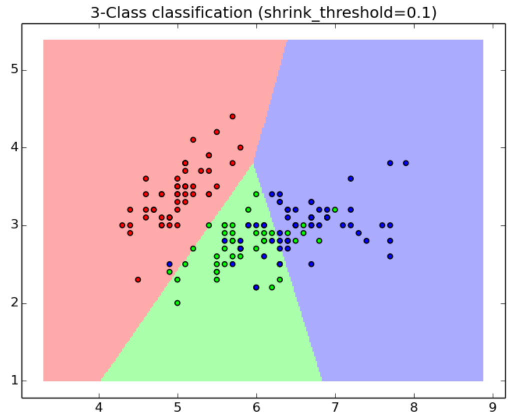
```
>>> from sklearn.neighbors.nearest_centroid import NearestCentroid
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> y = np.array([1, 1, 1, 2, 2, 2])
>>> clf = NearestCentroid()
>>> clf.fit(X, y)
NearestCentroid(metric='euclidean', shrink_threshold=None)
>>> 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.

4.6.6 Approximate Nearest Neighbors

There are many efficient exact nearest neighbor search algorithms for low dimensions d (approximately 50). However these algorithms perform poorly with respect to space and query time when d increases. These algorithms are not any better than comparing query point to each point from the database in a high dimension (see *Brute Force*). This is a well-known consequence of the phenomenon called "The Curse of Dimensionality".

There are certain applications where we do not need the exact nearest neighbors but having a "good guess" would suffice. When answers do not have to be exact, the `LSHForest` class implements an approximate nearest neighbor search. Approximate nearest neighbor search methods have been designed to try to speedup query time with high dimensional data. These techniques are useful when the aim is to characterize the neighborhood rather than identifying the exact neighbors themselves (eg: k-nearest neighbors classification and regression). Some of the most popular approximate nearest neighbor search techniques are locality sensitive hashing, best bin fit and balanced box-decomposition tree based search.

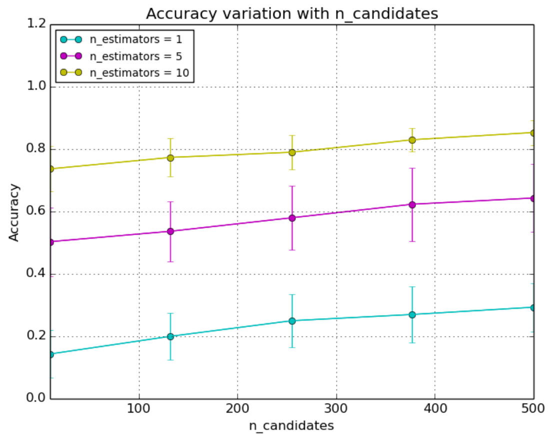
Locality Sensitive Hashing Forest

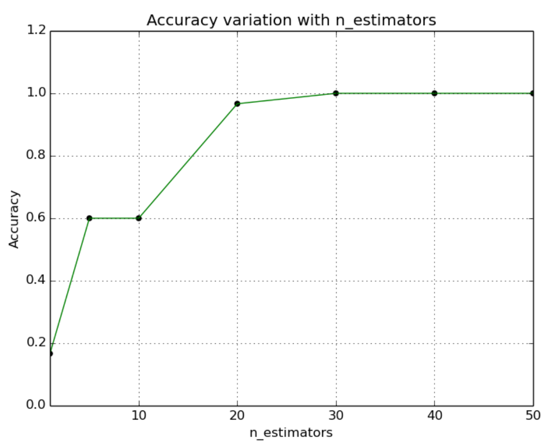
The vanilla implementation of locality sensitive hashing has a hyper-parameter that is hard to tune in practice, therefore scikit-learn implements a variant called `LSHForest` that has more reasonable hyperparameters. Both methods use internally random hyperplanes to index the samples into buckets and actual cosine similarities are only computed for samples that collide with the query hence achieving sublinear scaling. (see *Mathematical description of Locality Sensitive Hashing*).

`LSHForest` has two main hyper-parameters: `n_estimators` and `n_candidates`. The accuracy of queries can be controlled using these parameters as demonstrated in the following plots:

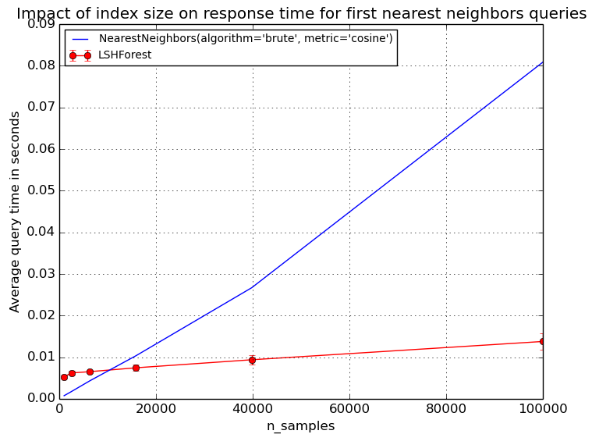
As a rule of thumb, a user can set `n_estimators` to a large enough value (e.g. between 10 and 50) and then adjust `n_candidates` to trade off accuracy for query time.

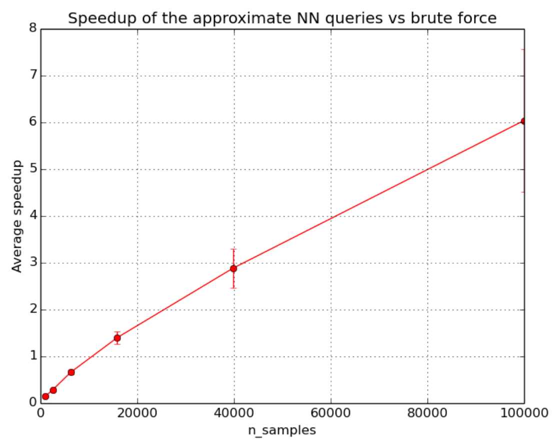
For small data sets, the brute force method for exact nearest neighbor search can be faster than LSH Forest. However LSH Forest has a sub-linear query time scalability with the index size. The exact break even point where LSH Forest queries become faster than brute force depends on the dimensionality, structure of the dataset, required level


Accuracy variation with n_candidates
n_estimators = 1
n_estimators = 5
n_estimators = 10
Accuracy
n_candidates
0.0
0.2
0.4
0.6
0.8
1.0
1.2
100
200
300
400
500


Accuracy variation with n_estimators
Accuracy
n_estimators
0.0
0.2
0.4
0.6
0.8
1.0
1.2
10
20
30
40
50

of precision, characteristics of the runtime environment such as availability of BLAS optimizations, number of CPU cores and size of the CPU caches. Following graphs depict scalability of LSHForest queries with index size.





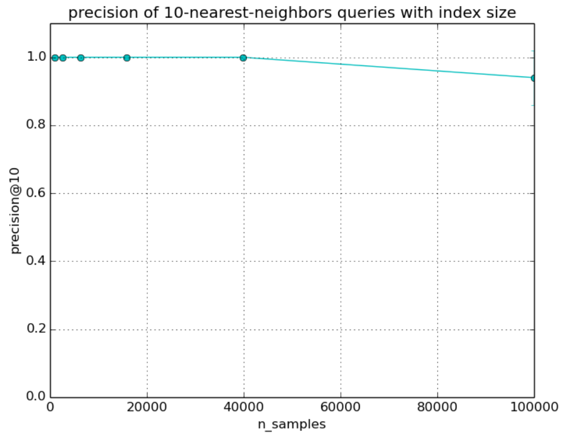
For fixed `LSHForest` parameters, the accuracy of queries tends to slowly decrease with larger datasets. The error bars on the previous plots represent standard deviation across different queries.

Examples:

- *Hyper-parameters of Approximate Nearest Neighbors*: an example of the behavior of hyperparameters of approximate nearest neighbor search using LSH Forest.
- *Scalability of Approximate Nearest Neighbors*: an example of scalability of approximate nearest neighbor search using LSH Forest.

Mathematical description of Locality Sensitive Hashing

Locality sensitive hashing (LSH) techniques have been used in many areas where nearest neighbor search is performed in high dimensions. The main concept behind LSH is to hash each data point in the database using multiple (often simple) hash functions to form a digest (also called a *hash*). At this point the probability of collision - where two objects have similar digests - is much higher for the points which are close to each other than that of the distant points. We describe the requirements for a hash function family to be locality sensitive as follows.



A family H of functions from a domain S to a range U is called $(r, e, p1, p2)$-sensitive, with $r, e > 0$, $p_1 > p_2 > 0$, if for any $p, q \in S$, the following conditions hold (D is the distance function):

- If $D(p, q) <= r$ then $P_H[h(p) = h(q)] >= p_1$,
- If $D(p, q) > r(1 + e)$ then $P_H[h(p) = h(q)] <= p_2$.

As defined, nearby points within a distance of r to each other are likely to collide with probability p_1. In contrast, distant points which are located with the distance more than $r(1 + e)$ have a small probability of p_2 of collision. Suppose there is a family of LSH function H. An *LSH index* is built as follows:

1. Choose k functions $h_1, h_2, \ldots h_k$ uniformly at random (with replacement) from H. For any $p \in S$, place p in the bucket with label $g(p) = (h_1(p), h_2(p), \ldots h_k(p))$. Observe that if each h_i outputs one "digit", each bucket has a k-digit label.
2. Independently perform step 1 l times to construct l separate estimators, with hash functions $g_1, g_2, \ldots g_l$.

The reason to concatenate hash functions in the step 1 is to decrease the probability of the collision of distant points as much as possible. The probability drops from p_2 to p_2^k which is negligibly small for large k. The choice of k is strongly dependent on the data set size and structure and is therefore hard to tune in practice. There is a side effect of having a large k; it has the potential of decreasing the chance of nearby points getting collided. To address this issue, multiple estimators are constructed in step 2.

The requirement to tune k for a given dataset makes classical LSH cumbersome to use in practice. The LSH Forest variant has benn designed to alleviate this requirement by automatically adjusting the number of digits used to hash the samples.

LSH Forest is formulated with prefix trees with each leaf of a tree corresponding to an actual data point in the database. There are l such trees which compose the forest and they are constructed using independently drawn random sequence of hash functions from H. In this implementation, "Random Projections" is being used as the LSH technique which is an approximation for the cosine distance. The length of the sequence of hash functions is kept fixed at 32. Moreover, a prefix tree is implemented using sorted arrays and binary search.

There are two phases of tree traversals used in order to answer a query to find the m nearest neighbors of a point q. First, a top-down traversal is performed using a binary search to identify the leaf having the longest prefix match (maximum depth) with q's label after subjecting q to the same hash functions. $M >> m$ points (total candidates) are extracted from the forest, moving up from the previously found maximum depth towards the root synchronously across all trees in the bottom-up traversal. *M* is set to cl where c, the number of candidates extracted from each tree, is a constant. Finally, the similarity of each of these M points against point q is calculated and the top m points are returned as the nearest neighbors of q. Since most of the time in these queries is spent calculating the distances to candidates, the speedup compared to brute force search is approximately N/M, where N is the number of points in database.

References:

- "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions", Alexandr, A., Indyk, P., Foundations of Computer Science, 2006. FOCS '06. 47th Annual IEEE Symposium
- "LSH Forest: Self-Tuning Indexes for Similarity Search", Bawa, M., Condie, T., Ganesan, P., WWW '05 Proceedings of the 14th international conference on World Wide Web Pages 651-660

4.7 Gaussian Processes

Gaussian Processes for Machine Learning (GPML) is a generic supervised learning method primarily designed to solve *regression* problems. It has also been extended to *probabilistic classification*, but in the present implementation, this is only a post-processing of the *regression* exercise.

The advantages of Gaussian Processes for Machine Learning are:

- The prediction interpolates the observations (at least for regular correlation models).
- The prediction is probabilistic (Gaussian) so that one can compute empirical confidence intervals and exceedance probabilities that might be used to refit (online fitting, adaptive fitting) the prediction in some region of interest.
- Versatile: different *linear regression models* and *correlation models* can be specified. Common models are provided, but it is also possible to specify custom models provided they are stationary.

The disadvantages of Gaussian Processes for Machine Learning include:

- It is not sparse. It uses the whole samples/features information to perform the prediction.
- It loses efficiency in high dimensional spaces – namely when the number of features exceeds a few dozens. It might indeed give poor performance and it loses computational efficiency.
- Classification is only a post-processing, meaning that one first need to solve a regression problem by providing the complete scalar float precision output y of the experiment one attempt to model.

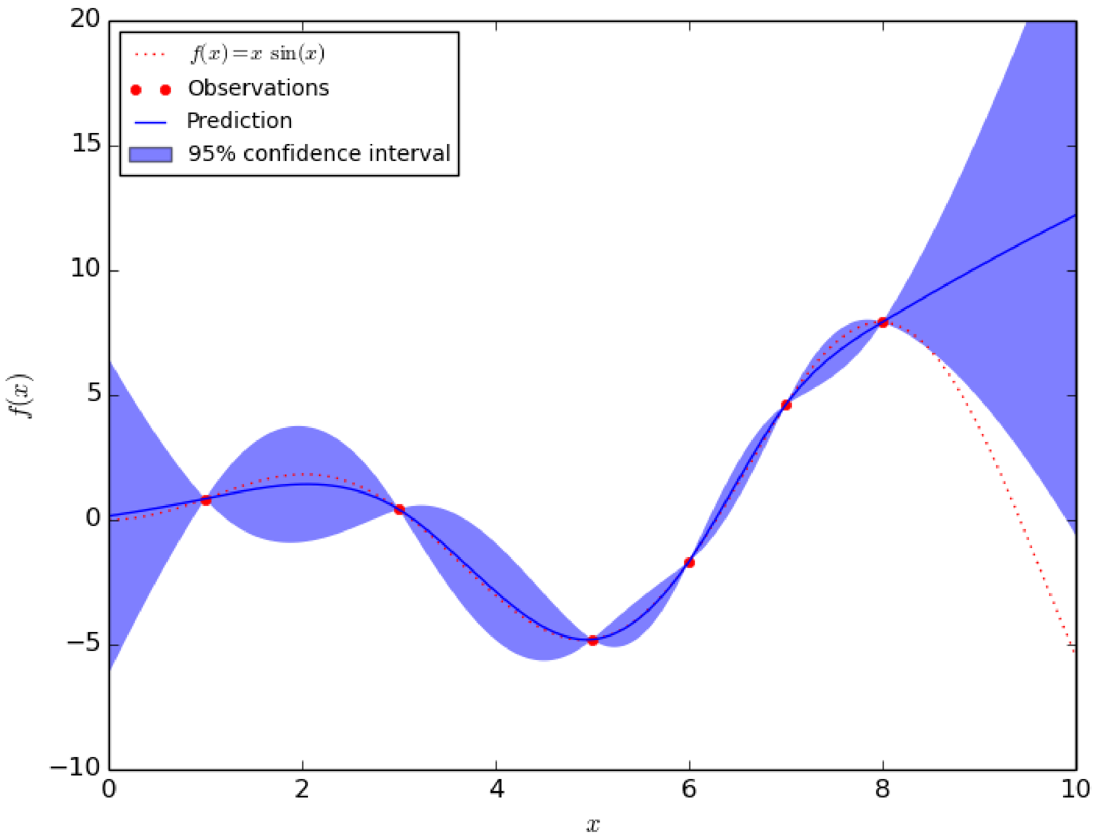
Thanks to the Gaussian property of the prediction, it has been given varied applications: e.g. for global optimization, probabilistic classification.

4.7.1 Examples

An introductory regression example

Say we want to surrogate the function $g(x) = x \sin(x)$. To do so, the function is evaluated onto a design of experiments. Then, we define a GaussianProcess model whose regression and correlation models might be specified using additional kwargs, and ask for the model to be fitted to the data. Depending on the number of parameters provided at instantiation, the fitting procedure may recourse to maximum likelihood estimation for the parameters or alternatively it uses the given parameters.

```
>>> import numpy as np
>>> from sklearn import gaussian_process
>>> def f(x):
...     return x * np.sin(x)
>>> X = np.atleast_2d([1., 3., 5., 6., 7., 8.]).T
>>> y = f(X).ravel()
>>> x = np.atleast_2d(np.linspace(0, 10, 1000)).T
>>> gp = gaussian_process.GaussianProcess(theta0=1e-2, thetaL=1e-4, thetaU=1e-1)
```


$f(x) = x\,\sin(x)$
Observations
Prediction
95% confidence interval
$f(x)$
x
20
15
10
5
0
−5
−10
0
2
4
6
8
10

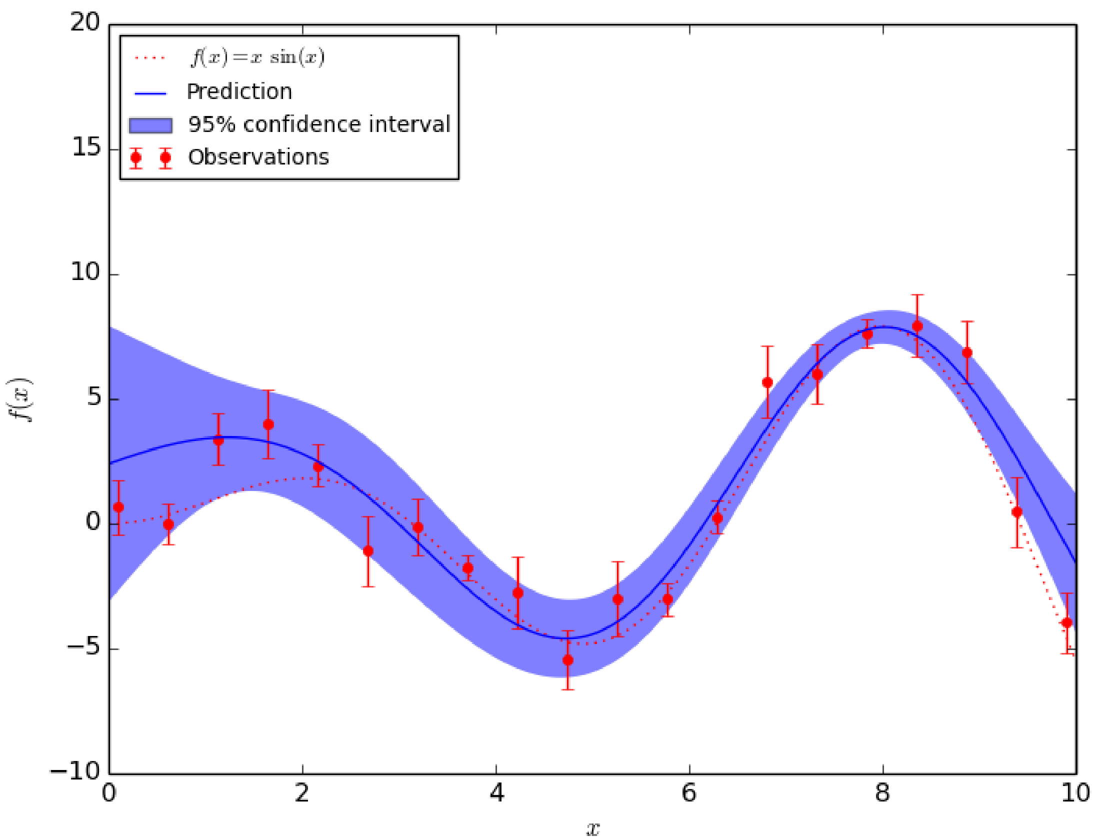
```
>>> gp.fit(X, y)
GaussianProcess(beta0=None, corr=<function squared_exponential at 0x...>,
        normalize=True, nugget=array(2.22...-15),
        optimizer='fmin_cobyla', random_start=1, random_state=...
        regr=<function constant at 0x...>, storage_mode='full',
        theta0=array([[ 0.01]]), thetaL=array([[ 0.0001]]),
        thetaU=array([[ 0.1]]), verbose=False)
>>> y_pred, sigma2_pred = gp.predict(x, eval_MSE=True)
```

Fitting Noisy Data

When the data to be fit includes noise, the Gaussian process model can be used by specifying the variance of the noise for each point. `GaussianProcess` takes a parameter `nugget` which is added to the diagonal of the correlation matrix between training points: in general this is a type of Tikhonov regularization. In the special case of a squared-exponential correlation function, this normalization is equivalent to specifying a fractional variance in the input. That is

$$\mathrm{nugget}_i = \left[\frac{\sigma_i}{y_i}\right]^2$$

With `nugget` and `corr` properly set, Gaussian Processes can be used to robustly recover an underlying function from noisy data:



Other examples

- *Gaussian Processes classification example: exploiting the probabilistic output*

4.7.2 Mathematical formulation

The initial assumption

Suppose one wants to model the output of a computer experiment, say a mathematical function:

$$\begin{aligned} g : & \mathbb{R}^{n_{\text{features}}} \rightarrow \mathbb{R} \\ & X \mapsto y = g(X) \end{aligned}$$

GPML starts with the assumption that this function is *a* conditional sample path of *a* Gaussian process G which is additionally assumed to read as follows:

$$G(X) = f(X)^T \beta + Z(X)$$

where $f(X)^T \beta$ is a linear regression model and $Z(X)$ is a zero-mean Gaussian process with a fully stationary covariance function:

$$C(X, X') = \sigma^2 R(|X - X'|)$$

σ^2 being its variance and R being the correlation function which solely depends on the absolute relative distance between each sample, possibly featurewise (this is the stationarity assumption).

From this basic formulation, note that GPML is nothing but an extension of a basic least squares linear regression problem:

$$g(X) \approx f(X)^T \beta$$

Except we additionally assume some spatial coherence (correlation) between the samples dictated by the correlation function. Indeed, ordinary least squares assumes the correlation model $R(|X - X'|)$ is one when $X = X'$ and zero otherwise : a *dirac* correlation model – sometimes referred to as a *nugget* correlation model in the kriging literature.

The best linear unbiased prediction (BLUP)

We now derive the *best linear unbiased prediction* of the sample path g conditioned on the observations:

$$\hat{G}(X) = G(X | y_1 = g(X_1), ..., y_{n_{\text{samples}}} = g(X_{n_{\text{samples}}}))$$

It is derived from its *given properties*:

- It is linear (a linear combination of the observations)

$$\hat{G}(X) \equiv a(X)^T y$$

- It is unbiased

$$\mathbb{E}[G(X) - \hat{G}(X)] = 0$$

- It is the best (in the Mean Squared Error sense)

$$\hat{G}(X)^* = \arg \min_{\hat{G}(X)} \; \mathbb{E}[(G(X) - \hat{G}(X))^2]$$

So that the optimal weight vector $a(X)$ is solution of the following equality constrained optimization problem:

$$a(X)^* = \arg \min_{a(X)} \; \mathbb{E}[(G(X) - a(X)^T y)^2]$$
$$\text{s.t.} \; \mathbb{E}[G(X) - a(X)^T y] = 0$$

Rewriting this constrained optimization problem in the form of a Lagrangian and looking further for the first order optimality conditions to be satisfied, one ends up with a closed form expression for the sought predictor – see references for the complete proof.

In the end, the BLUP is shown to be a Gaussian random variate with mean:

$$\mu_{\hat{Y}}(X) = f(X)^T \, \hat{\beta} + r(X)^T \, \gamma$$

and variance:

$$\sigma_{\hat{Y}}^2(X) = \sigma_Y^2 \, (1 - r(X)^T \, R^{-1} \, r(X) + u(X)^T \, (F^T \, R^{-1} \, F)^{-1} \, u(X))$$

where we have introduced:

- the correlation matrix whose terms are defined wrt the autocorrelation function and its built-in parameters θ:

$$R_{i\,j} = R(|X_i - X_j|, \theta), \; i, \, j = 1, ..., m$$

- the vector of cross-correlations between the point where the prediction is made and the points in the DOE:

$$r_i = R(|X - X_i|, \theta), \; i = 1, ..., m$$

- the regression matrix (eg the Vandermonde matrix if f is a polynomial basis):

$$F_{i\,j} = f_i(X_j), \; i = 1, ..., p, \, j = 1, ..., m$$

- the generalized least square regression weights:

$$\hat{\beta} = (F^T \, R^{-1} \, F)^{-1} \, F^T \, R^{-1} \, Y$$

- and the vectors:

$$\gamma = R^{-1}(Y - F \, \hat{\beta})$$
$$u(X) = F^T \, R^{-1} \, r(X) - f(X)$$

It is important to notice that the probabilistic response of a Gaussian Process predictor is fully analytic and mostly relies on basic linear algebra operations. More precisely the mean prediction is the sum of two simple linear combinations (dot products), and the variance requires two matrix inversions, but the correlation matrix can be decomposed only once using a Cholesky decomposition algorithm.

The empirical best linear unbiased predictor (EBLUP)

Until now, both the autocorrelation and regression models were assumed given. In practice however they are never known in advance so that one has to make (motivated) empirical choices for these models *Correlation Models*.

Provided these choices are made, one should estimate the remaining unknown parameters involved in the BLUP. To do so, one uses the set of provided observations in conjunction with some inference technique. The present implementation, which is based on the DACE's Matlab toolbox uses the *maximum likelihood estimation* technique – see DACE manual in references for the complete equations. This maximum likelihood estimation problem is turned into a global optimization problem onto the autocorrelation parameters. In the present implementation, this global optimization is solved by means of the fmin_cobyla optimization function from scipy.optimize. In the case of anisotropy however, we provide an implementation of Welch's componentwise optimization algorithm – see references.

For a more comprehensive description of the theoretical aspects of Gaussian Processes for Machine Learning, please refer to the references below:

References:

- DACE, A Matlab Kriging Toolbox S Lophaven, HB Nielsen, J Sondergaard 2002
- Screening, predicting, and computer experiments WJ Welch, RJ Buck, J Sacks, HP Wynn, TJ Mitchell, and MD Morris Technometrics 34(1) 15–25, 1992
- Gaussian Processes for Machine Learning CE Rasmussen, CKI Williams MIT Press, 2006 (Ed. T Diettrich)
- The design and analysis of computer experiments TJ Santner, BJ Williams, W Notz Springer, 2003

4.7.3 Correlation Models

Common correlation models matches some famous SVM's kernels because they are mostly built on equivalent assumptions. They must fulfill Mercer's conditions and should additionally remain stationary. Note however, that the choice of the correlation model should be made in agreement with the known properties of the original experiment from which the observations come. For instance:

- If the original experiment is known to be infinitely differentiable (smooth), then one should use the *squared-exponential correlation model*.
- If it's not, then one should rather use the *exponential correlation model*.
- Note also that there exists a correlation model that takes the degree of derivability as input: this is the Matern correlation model, but it's not implemented here (TODO).

For a more detailed discussion on the selection of appropriate correlation models, see the book by Rasmussen & Williams in references.

4.7.4 Regression Models

Common linear regression models involve zero- (constant), first- and second-order polynomials. But one may specify its own in the form of a Python function that takes the features X as input and that returns a vector containing the values of the functional set. The only constraint is that the number of functions must not exceed the number of available observations so that the underlying regression problem is not *underdetermined*.

4.7.5 Implementation details

The present implementation is based on a translation of the DACE Matlab toolbox.

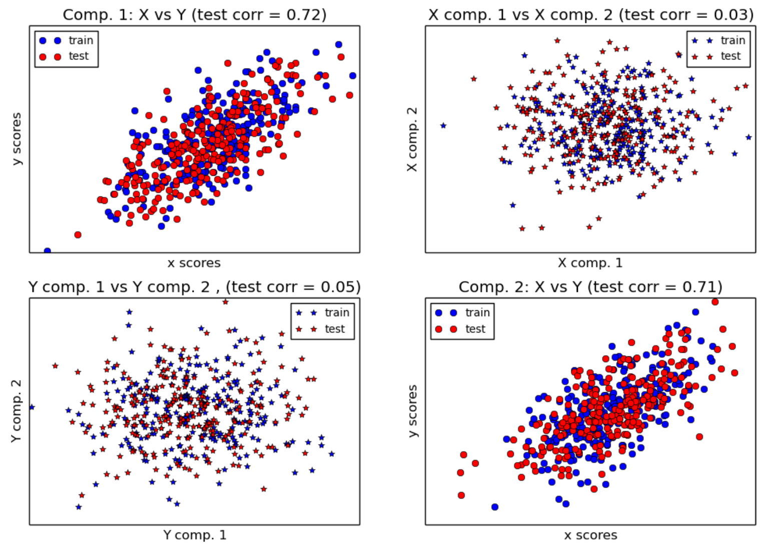
References:

- DACE, A Matlab Kriging Toolbox S Lophaven, HB Nielsen, J Sondergaard 2002,
- W.J. Welch, R.J. Buck, J. Sacks, H.P. Wynn, T.J. Mitchell, and M.D. Morris (1992). Screening, predicting, and computer experiments. Technometrics, 34(1) 15–25.

4.8 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*

4.9 Naive Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of independence between every pair of features. Given a class variable y and a dependent feature vector

x_1 through x_n, Bayes' theorem states the following relationship:

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) P(x_1, \dots x_n \mid y)}{P(x_1, \dots, x_n)}$$

Using the naive independence assumption that

$$P(x_i | y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i | y),$$

for all i, this relationship is simplified to

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \dots, x_n)}$$

Since $P(x_1, \dots, x_n)$ is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, \dots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

$$\Downarrow$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

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.

4.9.1 Gaussian Naive Bayes

`GaussianNB` implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma^2_y}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma^2_y}\right)$$

The parameters σ_y and μ_y are estimated using maximum likelihood.

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> from sklearn.naive_bayes import GaussianNB
>>> gnb = GaussianNB()
>>> y_pred = gnb.fit(iris.data, iris.target).predict(iris.data)
>>> print("Number of mislabeled points out of a total %d points : %d"
...       % (iris.data.shape[0],(iris.target != y_pred).sum()))
Number of mislabeled points out of a total 150 points : 6
```

4.9.2 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 $N_{y} = \sum_{i=1}^{|T|} N_{yi}$ is the total count of all features for class y.

The smoothing priors $\alpha \ge 0$ 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.

4.9.3 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).

4.9.4 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*. Both discrete classifiers support sample weighting; `GaussianNB` does not.

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.

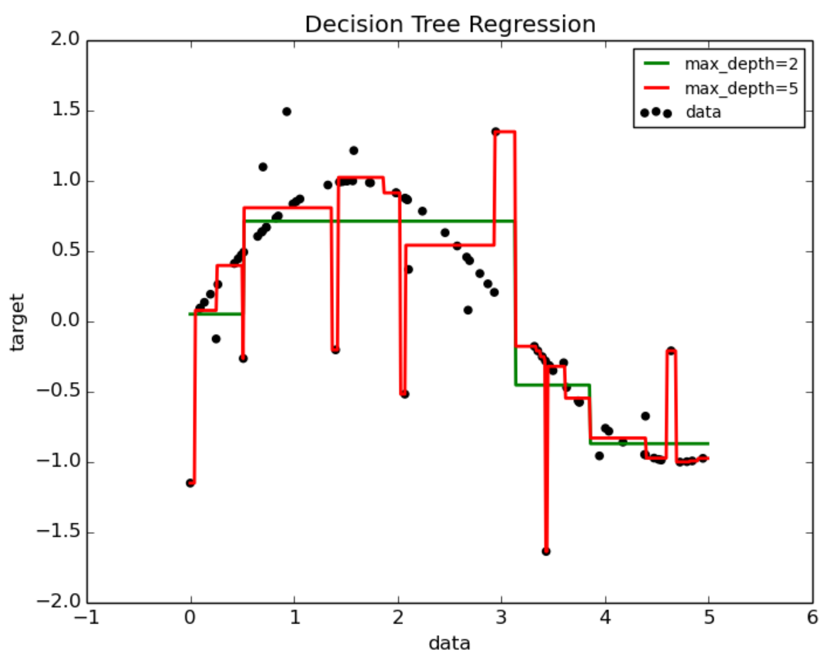
note:

```
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.
```

4.10 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.

4.10.1 Classification

`DecisionTreeClassifier` is a class capable of performing multi-class classification on a dataset.

As other classifiers, `DecisionTreeClassifier` take 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
>>> iris = load_iris()
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(iris.data, iris.target)
```

Once trained, we can export the tree in Graphviz format using the `export_graphviz` exporter. Below is an example export of a tree trained on the entire iris dataset:

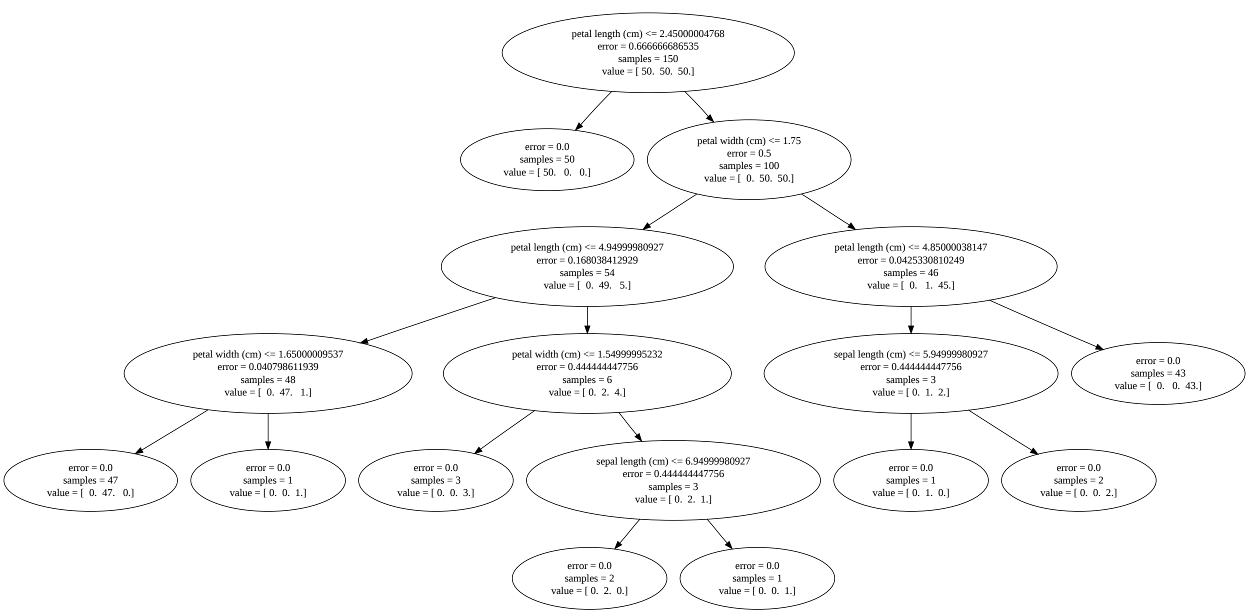
```
>>> from sklearn.externals.six import StringIO
>>> with open("iris.dot", 'w') as f:
...     f = tree.export_graphviz(clf, out_file=f)
```

Then we can use Graphviz's `dot` tool to create a PDF file (or any other supported file type): `dot -Tpdf iris.dot -o iris.pdf`.

```
>>> import os
>>> os.unlink('iris.dot')
```

Alternatively, if we have Python module `pydot` installed, we can generate a PDF file (or any other supported file type) directly in Python:

```
>>> from sklearn.externals.six import StringIO
>>> import pydot
>>> dot_data = StringIO()
>>> tree.export_graphviz(clf, out_file=dot_data)
>>> graph = pydot.graph_from_dot_data(dot_data.getvalue())
>>> graph.write_pdf("iris.pdf")
```

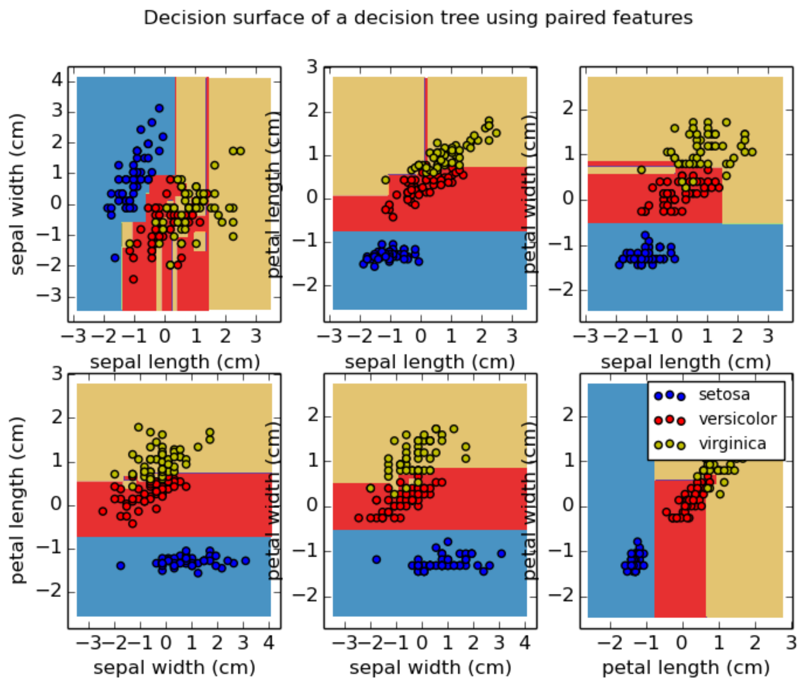


After being fitted, the model can then be used to predict the class of samples:

```
>>> clf.predict(iris.data[:1, :])
array([0])
```

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(iris.data[:1, :])
array([[ 1.,  0.,  0.]])
```



Examples:

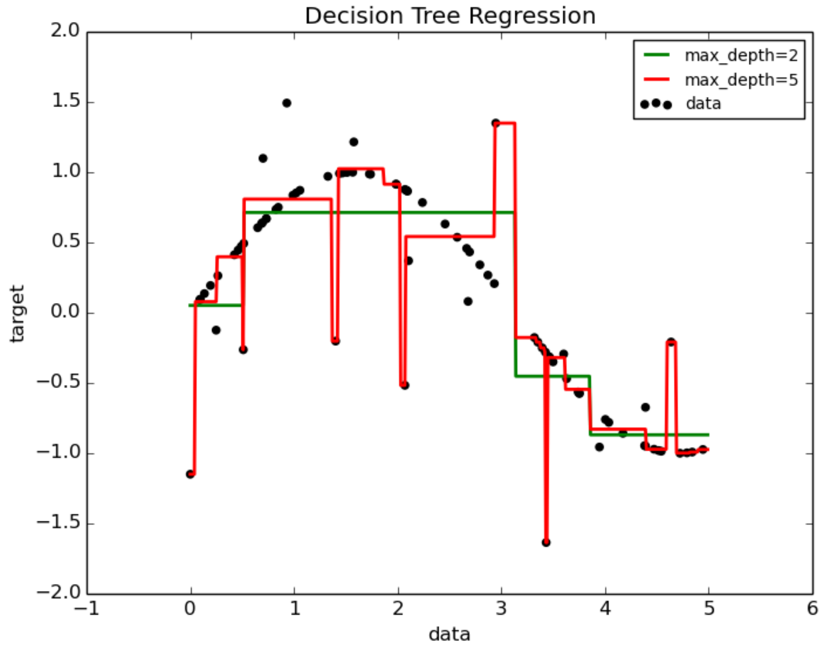
- *Plot the decision surface of a decision tree on the iris dataset*

4.10.2 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*

4.10.3 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.

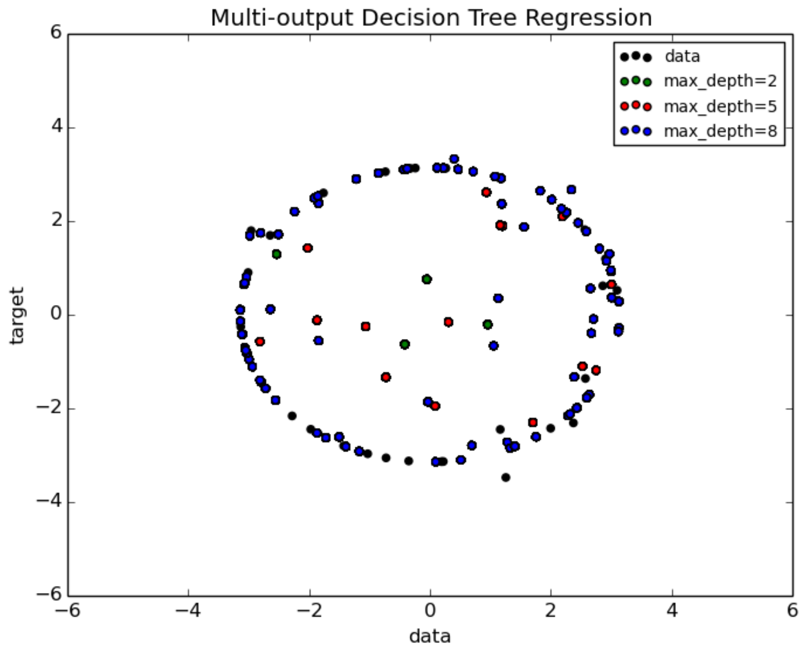
With regard to decision trees, this strategy can readily be used to support multi-output problems. This requires the following changes:

- 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

4.10.4 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 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 $O(n_{features}n_{samples}^2\log(n_{samples}))$.

Scikit-learn offers a more efficient implementation for the construction of decision trees. A naive implementation (as above) would recompute the class label histograms (for classification) or the means (for regression) at for each new split point along a given feature. By presorting the feature over all relevant samples, and retaining a running label count, we reduce the complexity at each node to $O(n_{features}\log(n_{samples}))$, which results in a total cost of $O(n_{features}n_{samples}\log(n_{samples}))$.


Face completion with multi-output estimators
true faces
Extra trees
K-nn
Linear regression
Ridge

4.10.5 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.
- 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 control the number of samples at a leaf node. 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. The main difference between the two is that `min_samples_leaf` guarantees a minimum number of samples in a leaf, while `min_samples_split` can create arbitrary small leaves, though `min_samples_split` is more common in the literature.
- 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.

4.10.6 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.

4.10.7 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

$$H(X_m) = \sum_k p_{mk}(1 - p_{mk})$$

Cross-Entropy

$$H(X_m) = \sum_k p_{mk} \log(p_{mk})$$

and Misclassification

$$H(X_m) = 1 - \max(p_{mk})$$

Regression criteria

If the target is a continuous value, then for node m, representing a region R_m with N_m observations, a common criterion to minimise is the Mean Squared Error

$$c_m = \frac{1}{N_m} \sum_{i \in N_m} y_i$$

$$H(X_m) = \frac{1}{N_m} \sum_{i \in N_m} (y_i - c_m)^2$$

References:

- http://en.wikipedia.org/wiki/Decision_tree_learning
- http://en.wikipedia.org/wiki/Predictive_analytics
- L. Breiman, J. Friedman, R. Olshen, and C. Stone. Classification and Regression Trees. Wadsworth, Belmont, CA, 1984.
- 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.

4.11 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*, ...

4.11.1 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. 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

4.11.2 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. In addition, when splitting a node during the construction of the tree, the split that is chosen is no longer the best split among all features. Instead, the split that is picked is the best split among a random subset of the features. As a result of this randomness, the bias of the forest usually slightly increases (with respect to the bias of a single non-random tree) but, due to averaging, its variance also decreases, usually more than compensating for the increase in bias, 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.

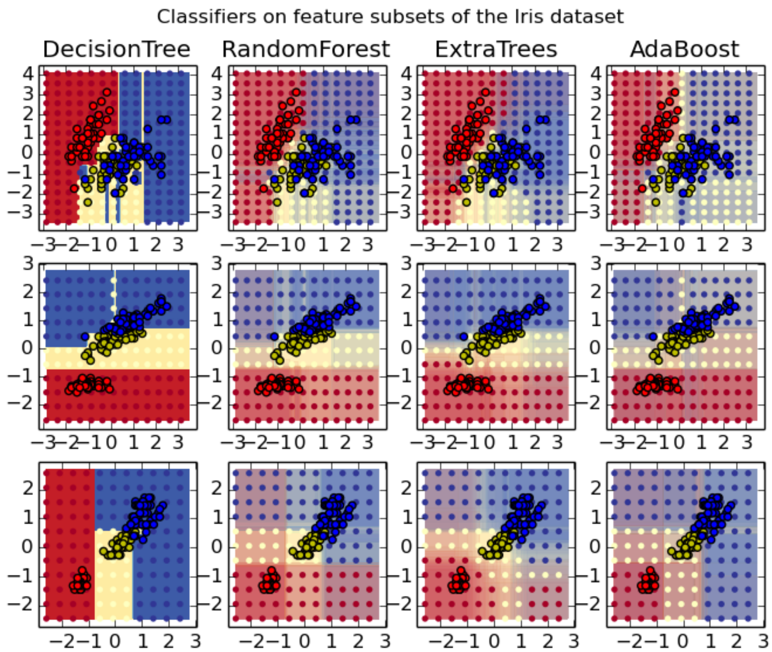
Extremely Randomized Trees

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.cross_validation 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=1,
...     random_state=0)
>>> scores = cross_val_score(clf, X, y)
>>> scores.mean()
0.97...

>>> clf = RandomForestClassifier(n_estimators=10, max_depth=None,
...     min_samples_split=1, random_state=0)
>>> scores = cross_val_score(clf, X, y)
>>> scores.mean()
0.999...

>>> clf = ExtraTreesClassifier(n_estimators=10, max_depth=None,
...     min_samples_split=1, random_state=0)
>>> scores = cross_val_score(clf, X, y)
>>> 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=n_features` for regression problems, and `max_features=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=1` (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`).



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*

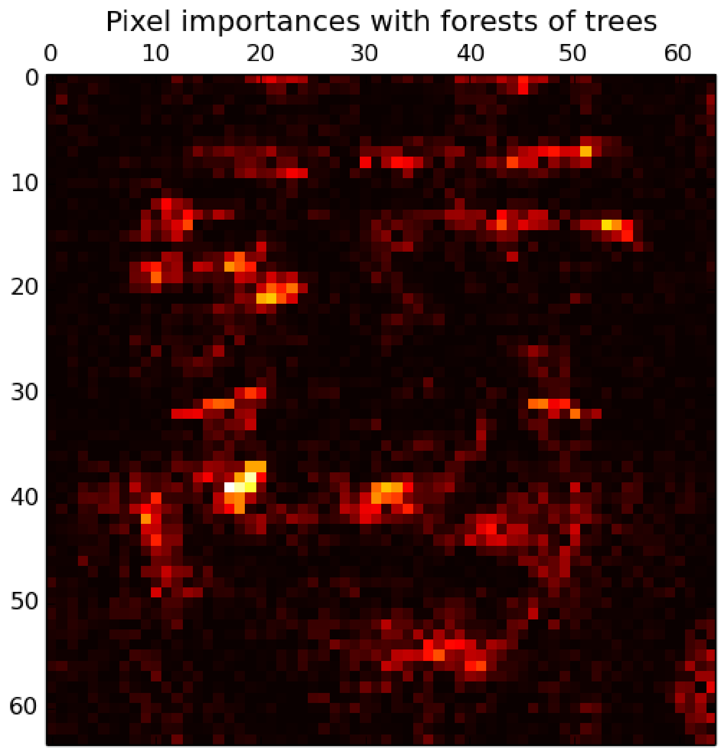
References

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 are used 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**.

By **averaging** those expected activity rates over several randomized trees one can **reduce the variance** of such an estimate and use it for feature selection.

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*

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.

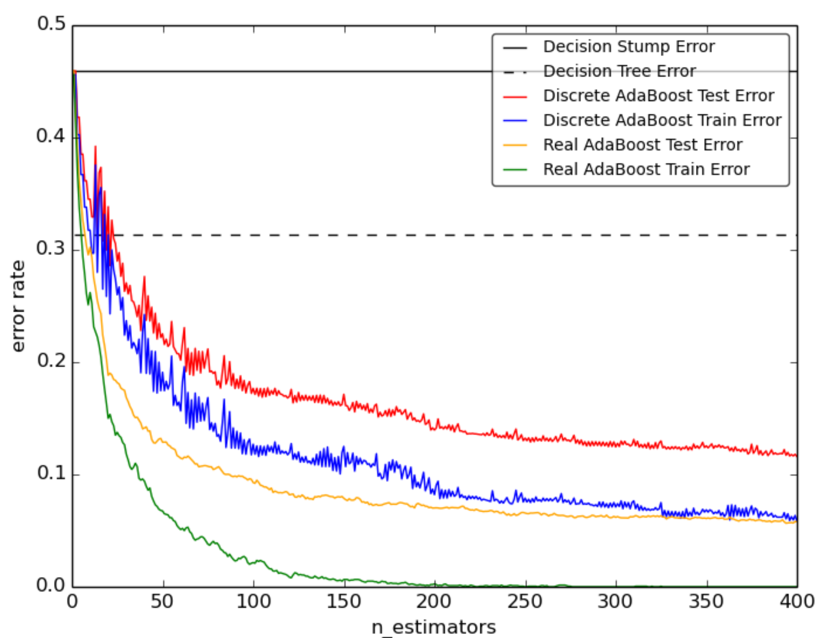
See also:

Manifold learning techniques can also be useful to derive non-linear representations of feature space, also these approaches focus also on dimensionality reduction.

4.11.3 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.cross_validation import cross_val_score
>>> from sklearn.datasets import load_iris
>>> from sklearn.ensemble import AdaBoostClassifier

>>> iris = load_iris()
>>> clf = AdaBoostClassifier(n_estimators=100)
>>> scores = cross_val_score(clf, iris.data, iris.target)
>>> 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 at a leaf `min_samples_leaf` in case of decision trees).

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

4.11.4 Gradient Tree Boosting

Gradient Tree Boosting or Gradient Boosted Regression Trees (GBRT) is a generalization of boosting to arbitrary differentiable loss functions. GBRT is an accurate and effective off-the-shelf procedure that can be used for both regression and classification problems. Gradient Tree Boosting models are used in a variety of areas including Web search ranking and ecology.

The advantages of GBRT are:

- Natural handling of data of mixed type (= heterogeneous features)
- Predictive power
- Robustness to outliers in output space (via robust loss functions)

The disadvantages of GBRT are:

- Scalability, due to the sequential nature of boosting it can hardly be parallelized.

The module `sklearn.ensemble` provides methods for both classification and regression via gradient boosted regression trees.

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* .

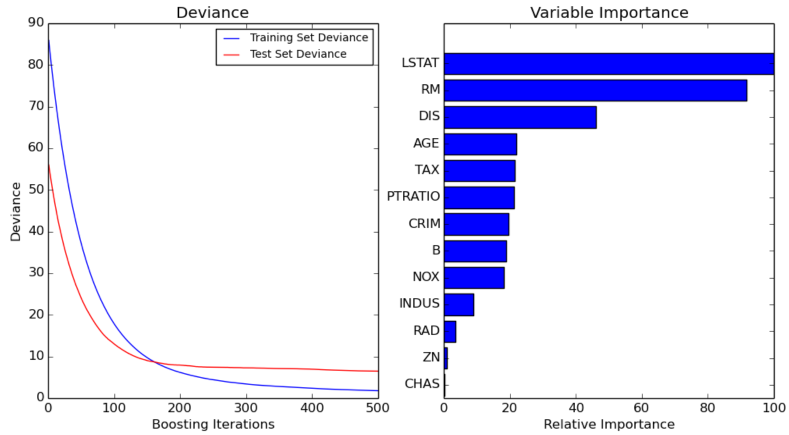
Note: Classification with more than 2 classes requires the induction of `n_classes` regression trees at each at each iteration, thus, the total number of induced trees equals `n_classes * n_estimators`. For datasets with a large number of classes we strongly recommend to use `RandomForestClassifier` 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 forward stagewise fashion:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

At each stage the decision tree $h_m(x)$ is chosen to minimize the loss function L given the current model F_{m-1} and its fit $F_{m-1}(x_i)$

$$F_m(x) = F_{m-1}(x) + \arg\min_{h} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) - h(x))$$

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 the gradient descent procedure; it can be set via the learning_rate parameter.

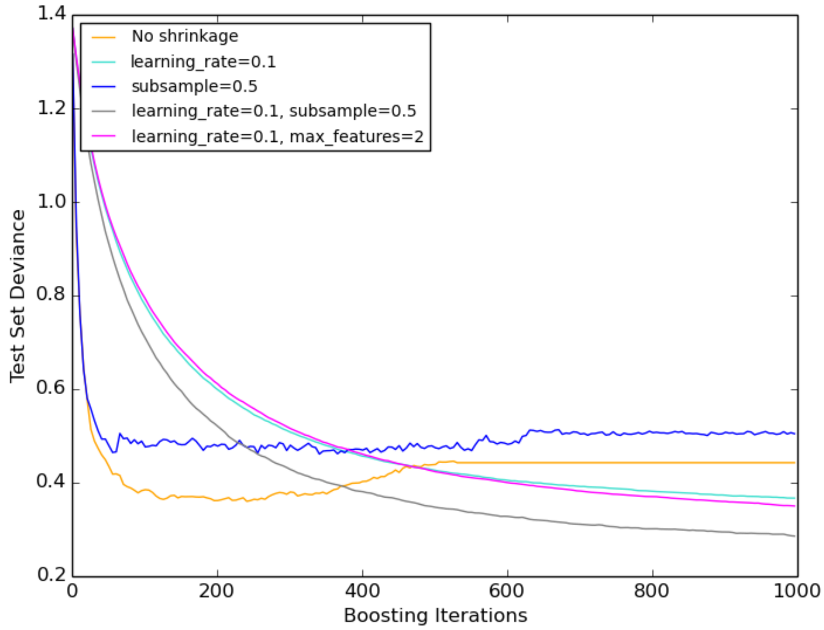
The parameter learning_rate strongly interacts with the parameter n_estimators, the number of weak learners to fit. Smaller values of learning_rate require larger numbers of weak learners to maintain a constant training error. Empirical evidence suggests that small values of learning_rate favor better test error. [HTF2009] 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*

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([ 0.11,  0.1 ,  0.11,  ...
```

Examples:

- *Gradient Boosting regression*

Partial dependence

Partial dependence plots (PDP) show the dependence between the target response 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 [5] as a function of the 'target' features [6].

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:

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.

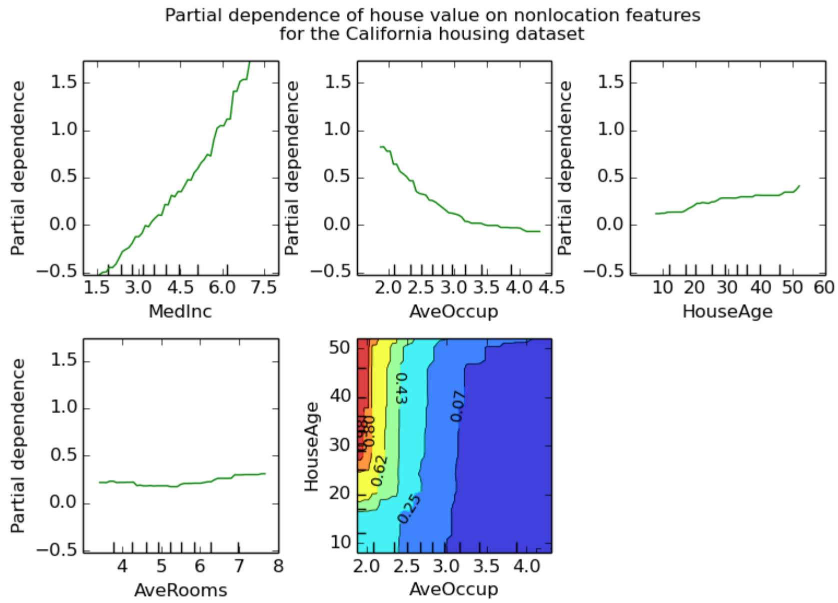
PDPs with two target features show the interactions among the two features. For example, the two-variable PDP in the above Figure shows the dependence of median house price on joint values of house age and avg. occupants per household. We can clearly see an interaction between the two features: For an avg. 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.

The module `partial_dependence` 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.ensemble.partial_dependence 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)
```

[5] For classification with `loss='deviance'` the target response is logit(p).

[6] More precisely its the expectation of the target response after accounting for the initial model; partial dependence plots do not include the `init` model.




```
>>> features = [0, 1, (0, 1)]
>>> fig, axs = plot_partial_dependence(clf, X, features)
```

For multi-class models, you need to set the class label for which the PDPs should be created via the `label` 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)]
>>> fig, axs = plot_partial_dependence(mc_clf, X, features, label=0)
```

If you need the raw values of the partial dependence function rather than the plots you can use the `partial_dependence` function:

```
>>> from sklearn.ensemble.partial_dependence import partial_dependence

>>> pdp, axes = partial_dependence(clf, [0], X=X)
>>> pdp
array([[ 2.46643157,  2.46643157, ...
>>> axes
[array([-1.62497054, -1.59201391, ...
```

The function requires either the argument `grid` which specifies the values of the target features on which the partial dependence function should be evaluated or the argument `X` which is a convenience mode for automatically creating `grid` from the training data. If `X` is given, the `axes` value returned by the function gives the axis for each target feature.

For each value of the 'target' features in the `grid` the partial dependence function need to marginalize the predictions of a tree over all possible values of the 'complement' features. In decision trees this function can be evaluated efficiently without reference to the training data. 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. For tree ensembles the results of each individual tree are again averaged.

Examples:

- *Partial Dependence Plots*

References

4.12 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.

- **Multiclass classification** means a classification task with more than two classes; e.g., classify a set of images of fruits which may be oranges, apples, or pears. Multiclass classification makes the assumption that each sample is assigned to one and only one label: a fruit can be either an apple or a pear but not both at the same time.
- **Multilabel classification** assigns to each sample a set of target labels. This can be thought as predicting properties of a data-point that are not mutually exclusive, such as topics that are relevant for a document. A text might be about any of religion, politics, finance or education at the same time or none of these.
- **Multioutput-multiclass classification** and **multi-task classification** means that a single estimator has to handle several joint classification tasks. This is a generalization of the multi-label classification task, where the set of classification problem is restricted to binary classification, and of the multi-class classification task. *The output format is a 2d numpy array or sparse matrix.*

  The set of labels can be different for each output variable. For instance a sample could be assigned "pear" for an output variable that takes possible values in a finite set of species such as "pear", "apple", "orange" and "green" for a second output variable that takes possible values in a finite set of colors such as "green", "red", "orange", "yellow"...

  This means that any classifiers handling multi-output multiclass or multi-task classification task supports the multi-label classification task as a special case. Multi-task classification is similar to the multi-output 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).

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: *Naive Bayes*, `sklearn.lda.LDA`, *Decision Trees*, *Random Forests*, *Nearest Neighbors*, setting "multi_class=multinomial" in `sklearn.linear_model.LogisticRegression`.
- One-Vs-One: `sklearn.svm.SVC`.
- One-Vs-All: all linear models except `sklearn.svm.SVC`.

Some estimators also support multioutput-multiclass classification tasks *Decision Trees*, *Random Forests*, *Nearest Neighbors*.

Warning: At present, no metric in `sklearn.metrics` supports the multioutput-multiclass classification task.

4.12.1 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 transformer `MultiLabelBinarizer` will convert between a collection of collections of labels and the indicator format.

```
>>> from sklearn.datasets import make_multilabel_classification
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> X, Y = make_multilabel_classification(n_samples=5, random_state=0,
...                                       return_indicator=False)
>>> 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]])
```

4.12.2 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 one classifier only, 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.

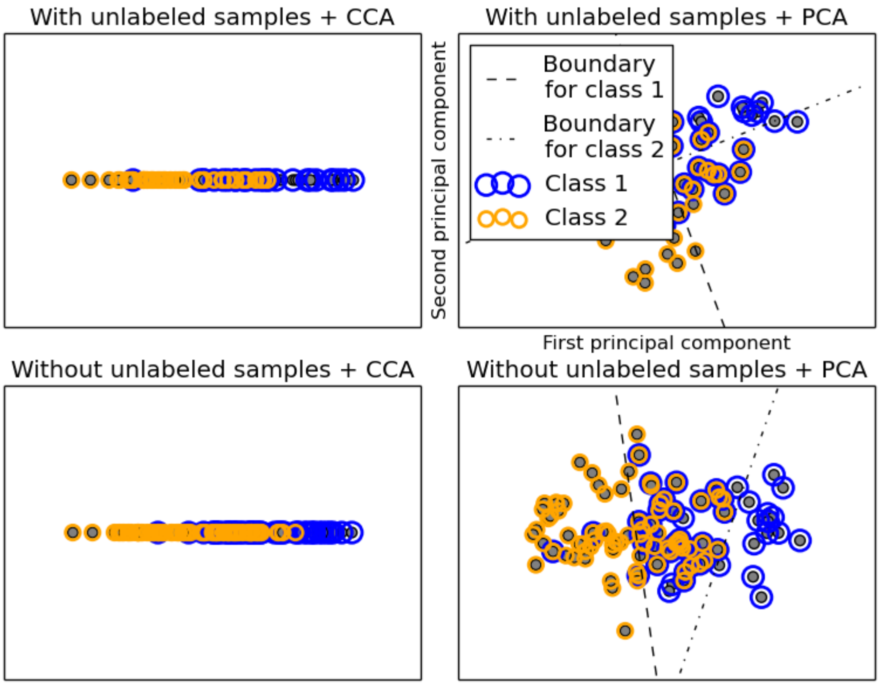
Multiclass learning

Below is an example of multiclass learning using OvR:

```
>>> from sklearn import datasets
>>> from sklearn.multiclass import OneVsRestClassifier
>>> from sklearn.svm import LinearSVC
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> 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, 0, 0, 1, 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, 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, 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*

4.12.3 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.

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
```

```
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> 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, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       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:

4.12.4 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 [7] 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 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
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> 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,
```

[7] "The error coding method and PICTs", James G., Hastie T., Journal of Computational and Graphical statistics 7, 1998.

```
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 1, 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,
       2, 2, 2, 2, 1, 2, 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:

4.13 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.

4.13.1 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.

4.13.2 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
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> 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 p-values:

- For regression: `f_regression`
- For classification: `chi2` or `f_classif`

Feature selection with sparse data

If you use sparse data (i.e. data represented as sparse matrices), only `chi2` 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

4.13.3 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 weights are assigned to each one of them. Then, features whose absolute weights are the smallest are pruned from the current set 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.

4.13.4 L1-based feature selection

Selecting non-zero coefficients

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 expose a `transform` method to select the non-zero coefficient. 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
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X.shape
(150, 4)
>>> X_new = LinearSVC(C=0.01, penalty="l1", dual=False).fit_transform(X, y)
>>> 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.

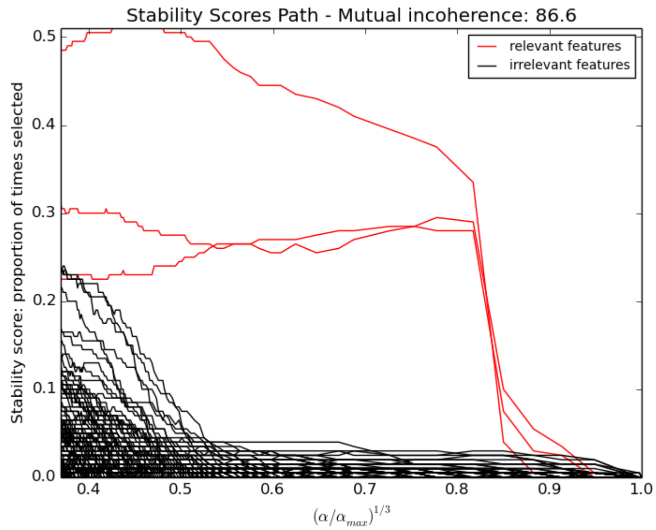
L1-recovery and compressive sensing

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://dsp.rice.edu/files/cs/baraniukCSlecture07.pdf

Randomized sparse models

The limitation of L1-based sparse models is that faced with a group of very correlated features, they will select only one. To mitigate this problem, it is possible to use randomization techniques, reestimating the sparse model many times perturbing the design matrix or sub-sampling data and counting how many times a given regressor is selected.

`RandomizedLasso` implements this strategy for regression settings, using the Lasso, while `RandomizedLogisticRegression` uses the logistic regression and is suitable for classification tasks. To get a full path of stability scores you can use `lasso_stability_path`.



Note that for randomized sparse models to be more powerful than standard F statistics at detecting non-zero features, the ground truth model should be sparse, in other words, there should be only a small fraction of features non zero.

Examples:

- *Sparse recovery: feature selection for sparse linear models*: An example comparing different feature selection approaches and discussing in which situation each approach is to be favored.

References:

- N. Meinshausen, P. Buhlmann, "Stability selection", Journal of the Royal Statistical Society, 72 (2010) http://arxiv.org/pdf/0809.2932
- F. Bach, "Model-Consistent Sparse Estimation through the Bootstrap" http://hal.inria.fr/hal-00354771/

4.13.5 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:

```
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X.shape
(150, 4)
>>> clf = ExtraTreesClassifier()
>>> X_new = clf.fit(X, y).transform(X)
>>> clf.feature_importances_
array([ 0.04...,  0.05...,  0.4...,  0.4...])
>>> 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.

4.13.6 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', LinearSVC(penalty="l1")),
  ('classification', RandomForestClassifier())
])
clf.fit(X, y)
```

In this snippet we make use of a `sklearn.svm.LinearSVC` 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.

4.14 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.

4.14.1 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.

`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

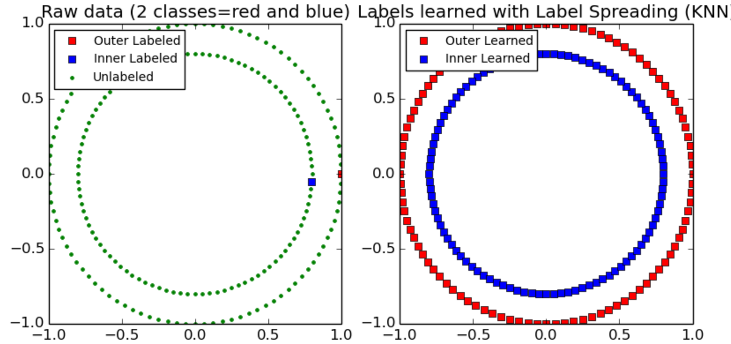


Figure 4.1: **An illustration of label-propagation:** *the structure of unlabeled observations is consistent with the class structure, and thus the class label can be propagated to the unlabeled observations of the training set.*

means $\alpha = 1$. This clamping factor can be relaxed, to say $\alpha = 0.8$, which means that we will always retain 80 percent of our original label distribution, but the algorithm gets to change it's 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 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 http://research.microsoft.com/en-us/people/nicolasl/efficient_ssl.pdf

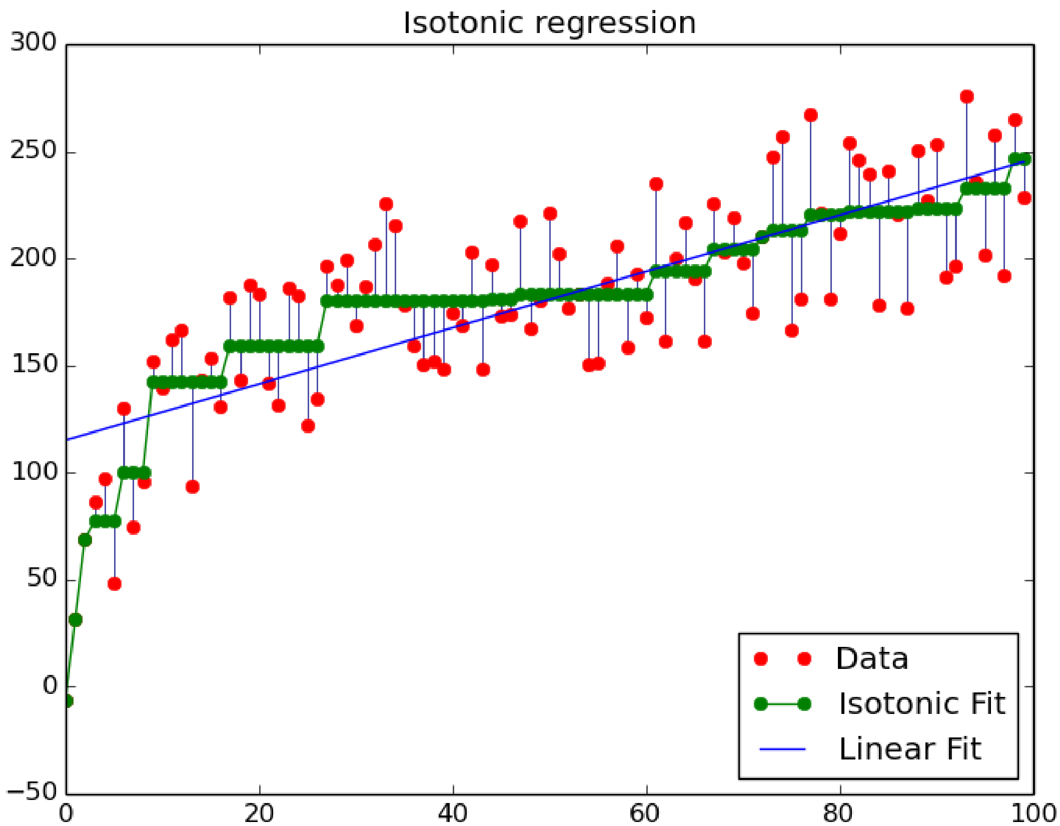
4.15 Isotonic regression

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.

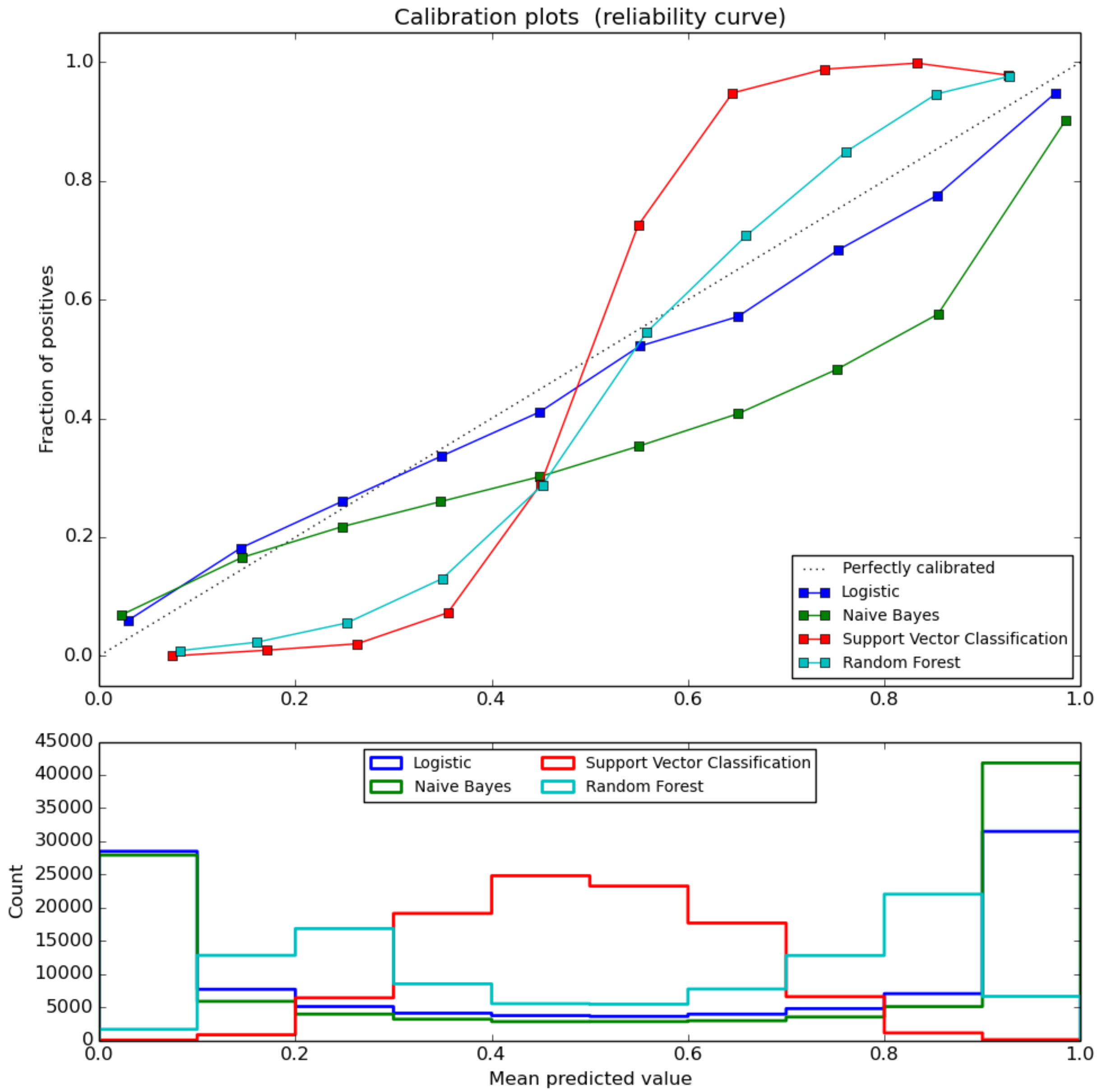


4.16 Probability calibration

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 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:


Calibration plots (reliability curve)
Fraction of positives
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
Perfectly calibrated
Logistic
Naive Bayes
Support Vector Classification
Random Forest
Count
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
Logistic
Naive Bayes
Support Vector Classification
Random Forest
Mean predicted value

- `GaussianNB` tends to push probabilties 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 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 subseting.” As a result, the calibration curve shows a characteristic sigmoid shape, indicating that the classifier could trust its “intuition” more and return probabilties closer to 0 or 1 typically.
- Linear Support Vector Classification (`LinearSVC`) shows an even more sigmoid curve as the RandomForest-Classifier, 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 paramter 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.

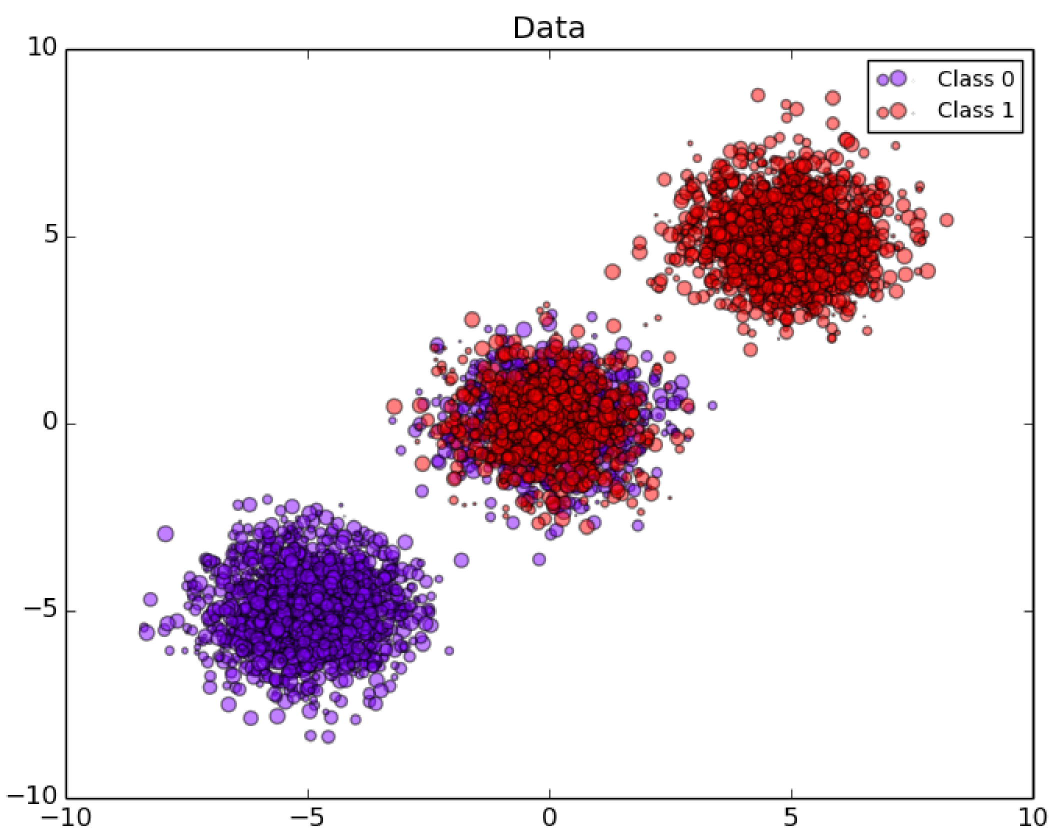
The following image shows on the data above the estimated probability using a Gaussian naive Bayes classifier without calibration, with a sigmoid calibration and with a non-parametric isotonic calibration. One can observe that the non-parametric model provides the most accurate probability estimates for samples in the middle, i.e., 0.5.

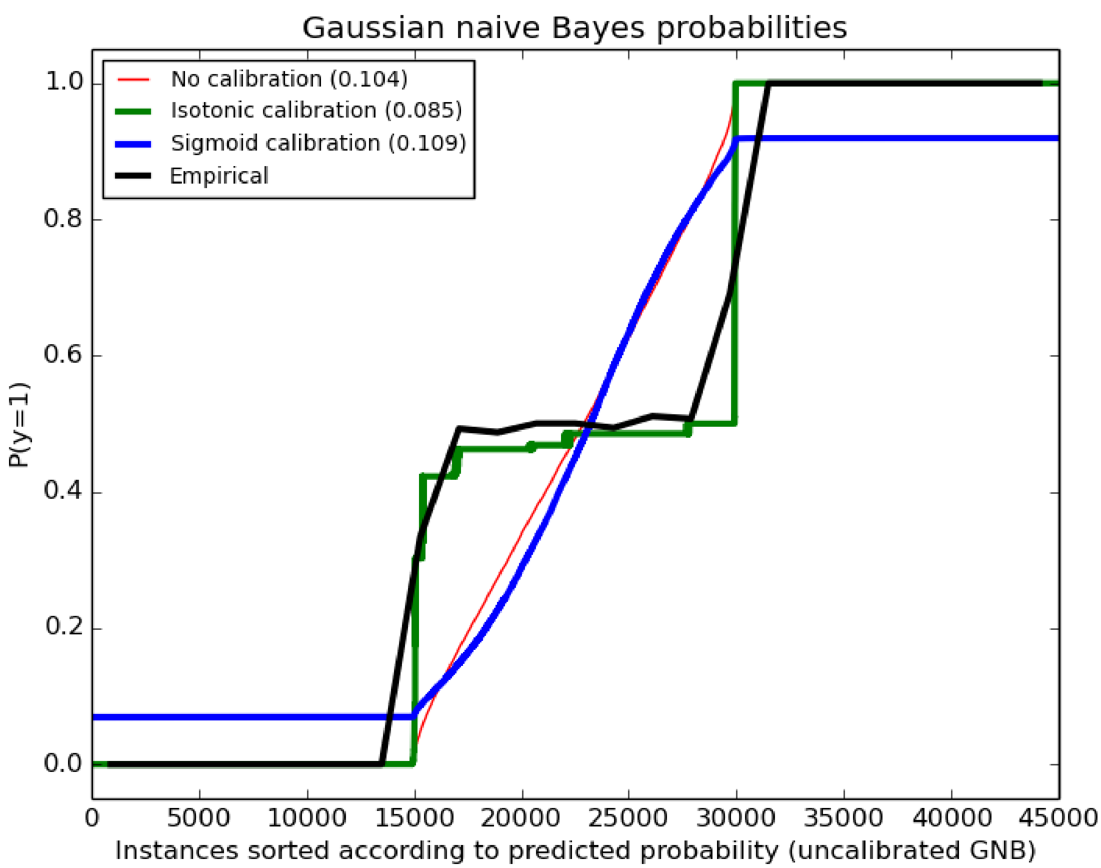
The following experiment is performed on an artificial dataset for binary classification with 100.000 samples (1.000 of them are used for model fitting) with 20 features. Of the 20 features, only 2 are informative and 10 are redundant. The figure shows the estimated probabilities obtained with logistic regression, a linear support-vector classifier (SVC), and linear SVC with both isotonic calibration and sigmoid calibration. The calibration performance is evaluated with Brier score `brier_score_loss`, reported in the legend (the smaller the better).

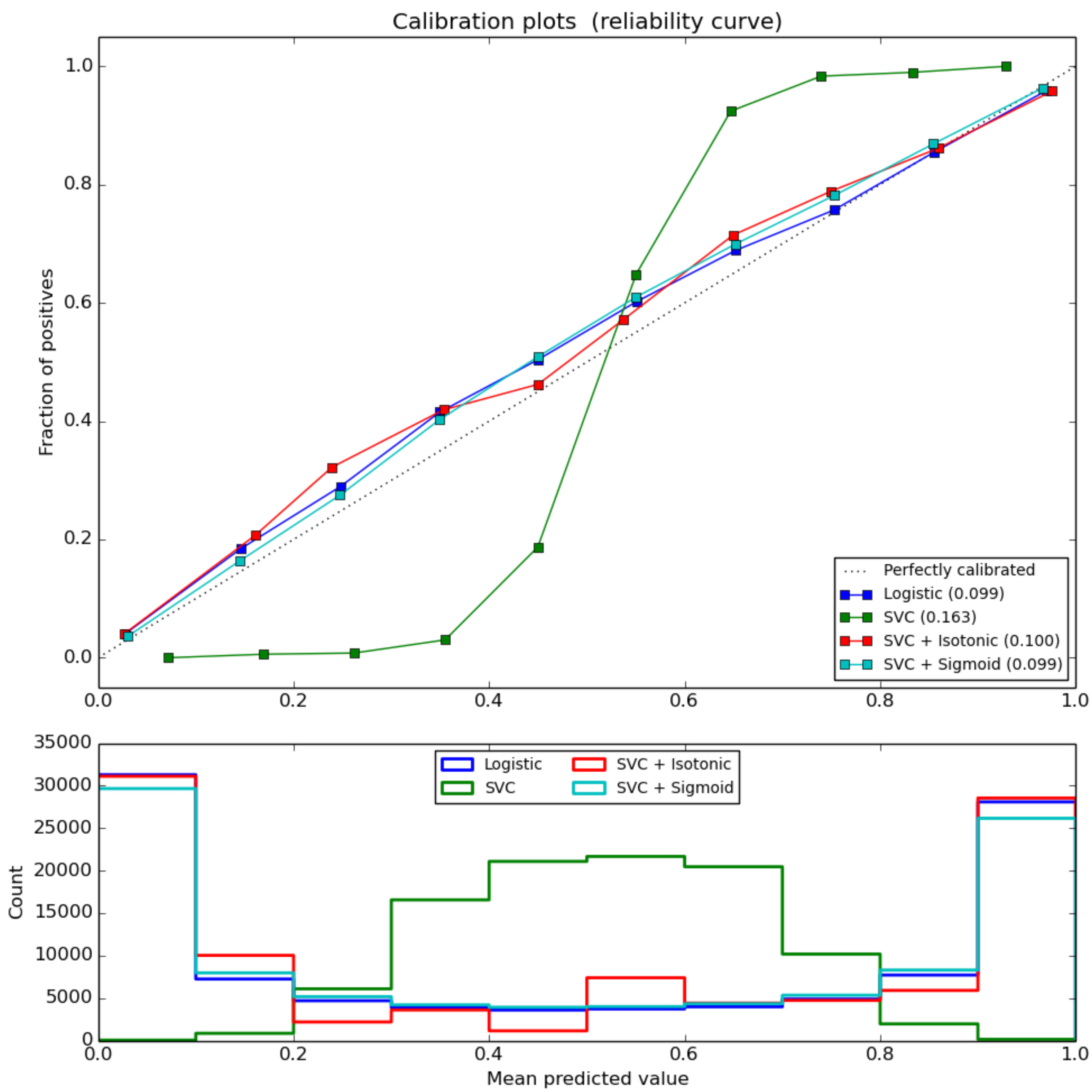
One can observe here that logistic regression is well calibrated as its curve is nearly diagonal. Linear SVC’s calibration curve has a sigmoid curve, which is typical for an under-confident classifier. In the case of LinearSVC, this is caused by the margin property of the hinge loss, which lets the model 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. The next figure shows the calibration curve of Gaussian naive Bayes on the same data, with both kinds of calibration and also without calibration.

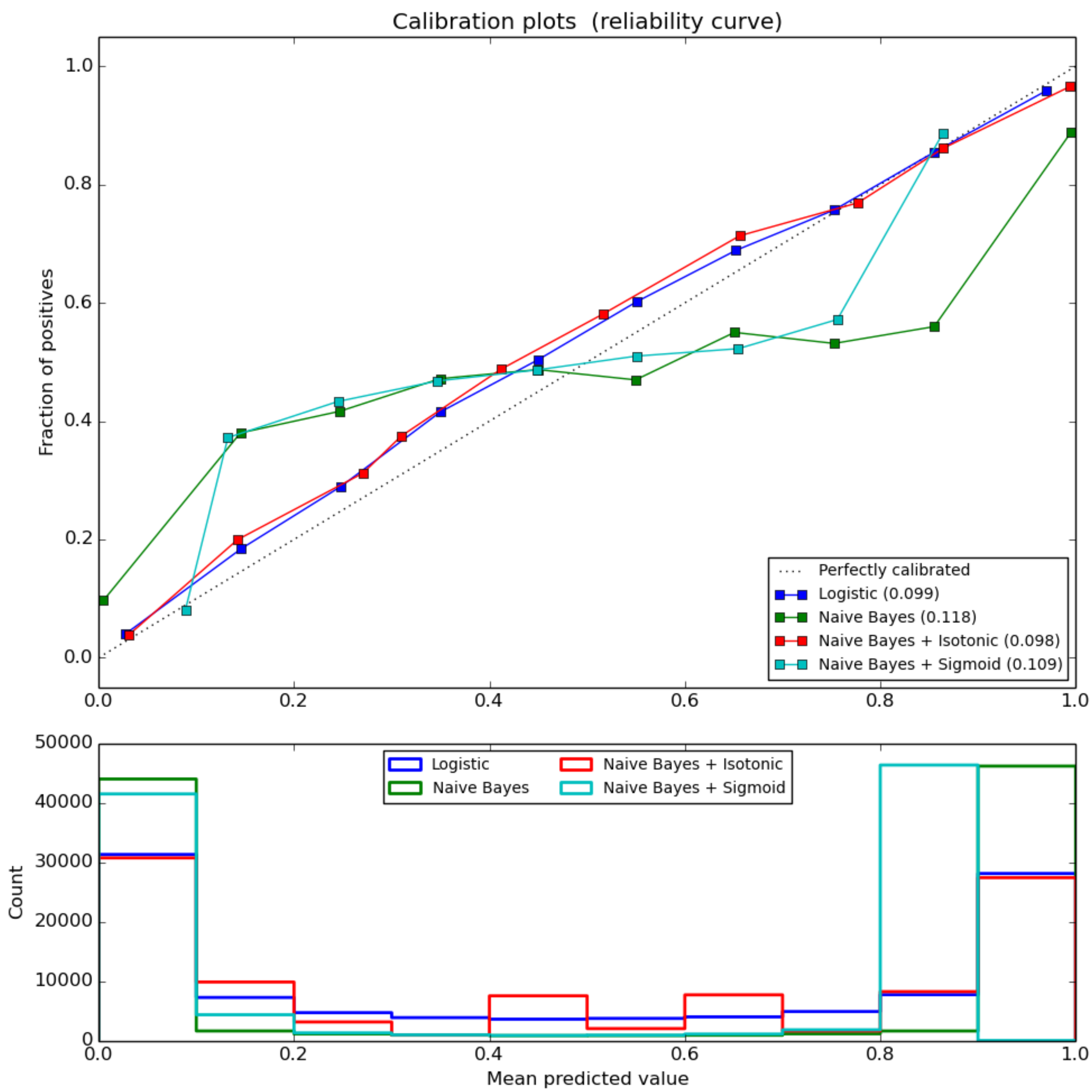
One can see that Gaussian naive Bayes performs very badly but does so in an other way than linear SVC: While linear SVC exhibited a sigmoid calibration curve, Gaussian naive Bayes’ calibration curve has a transposed-sigmoid shape. This is typical for an over-confident classifier. In this case, the classifier’s 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


Data
Class 0
Class 1
10
5
0
−5
−10
−10
−5
0
5
10


Gaussian naive Bayes probabilities
No calibration (0.104)
Isotonic calibration (0.085)
Sigmoid calibration (0.109)
Empirical
1.0
0.8
0.6
0.4
0.2
0.0
P(y=1)
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
Instances sorted according to predicted probability (uncalibrated GNB)

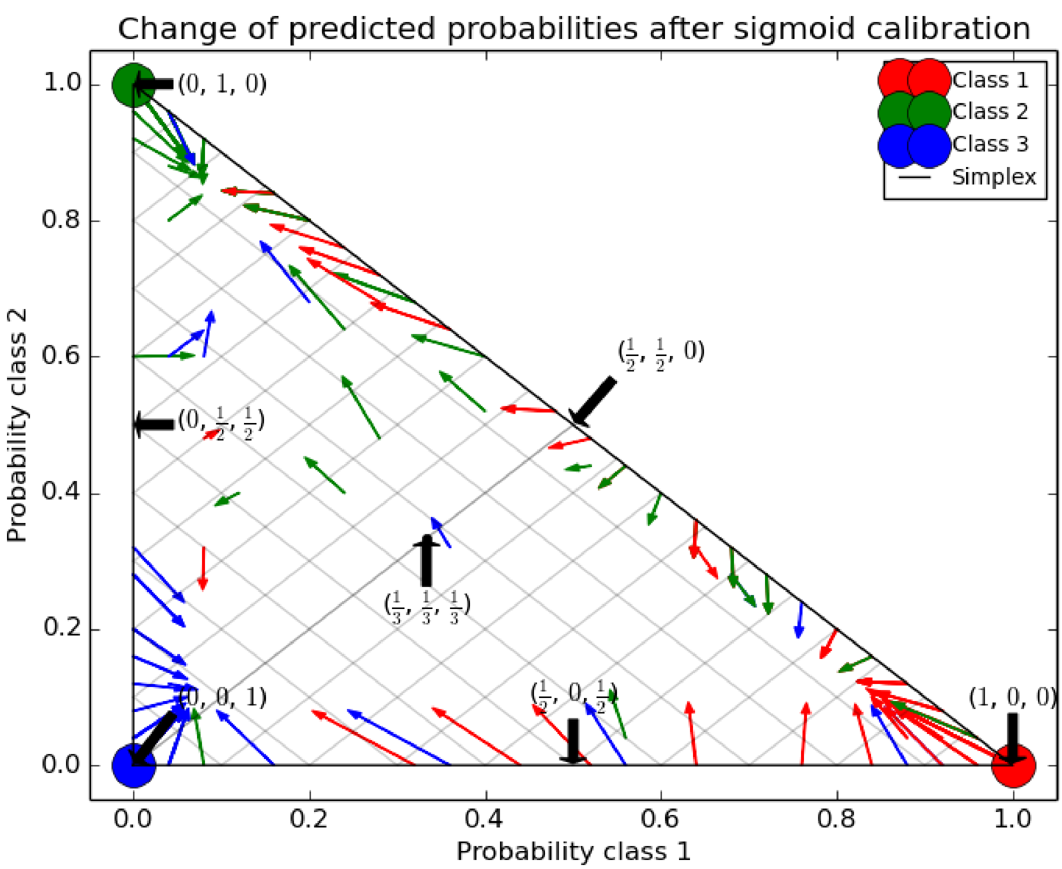

Calibration plots (reliability curve)
Fraction of positives
Perfectly calibrated
Logistic (0.099)
SVC (0.163)
SVC + Isotonic (0.100)
SVC + Sigmoid (0.099)
Count
Logistic
SVC
SVC + Isotonic
SVC + Sigmoid
Mean predicted value


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)
Count
Logistic
Naive Bayes
Naive Bayes + Isotonic
Naive Bayes + Sigmoid
Mean predicted value

as the non-parametric isotonic calibration. This is an intrinsic limitation of sigmoid calibration, whose parametric form 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 if the calibration curve is sigmoid and when there is few calibration data while isotonic calibration is preferable for non- sigmoid calibration curves and in situations where many additional data can be used 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.

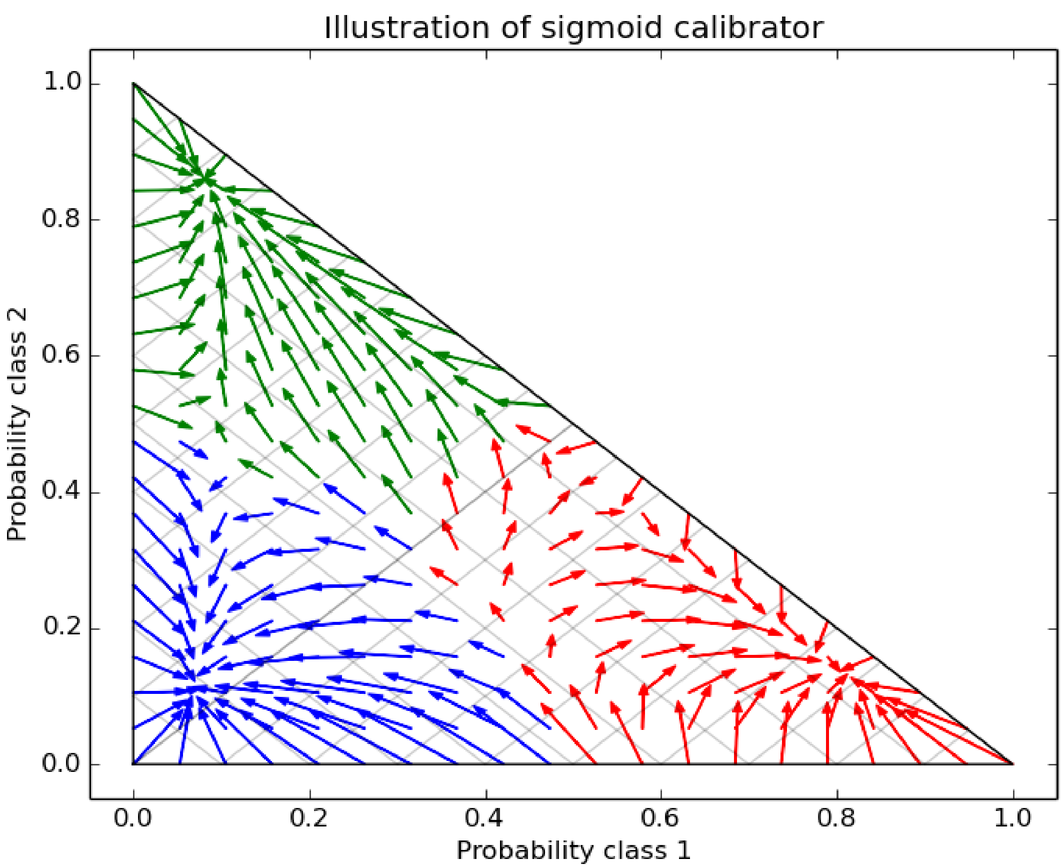


Illustration of sigmoid calibrator
Probability class 2
Probability class 1
0.0
0.2
0.4
0.6
0.8
1.0

References:

CHAPTER

FIVE

UNSUPERVISED LEARNING

5.1 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.

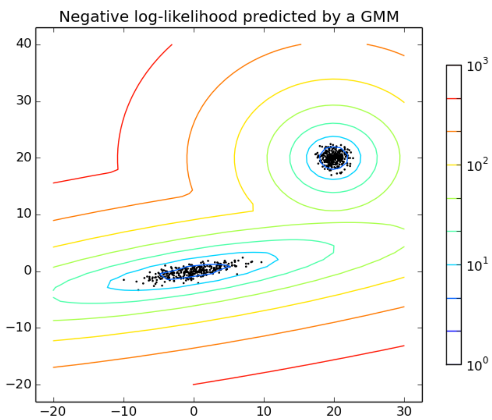


Figure 5.1: **Two-component Gaussian mixture model:** *data points, and equi-probability surfaces of the model.*

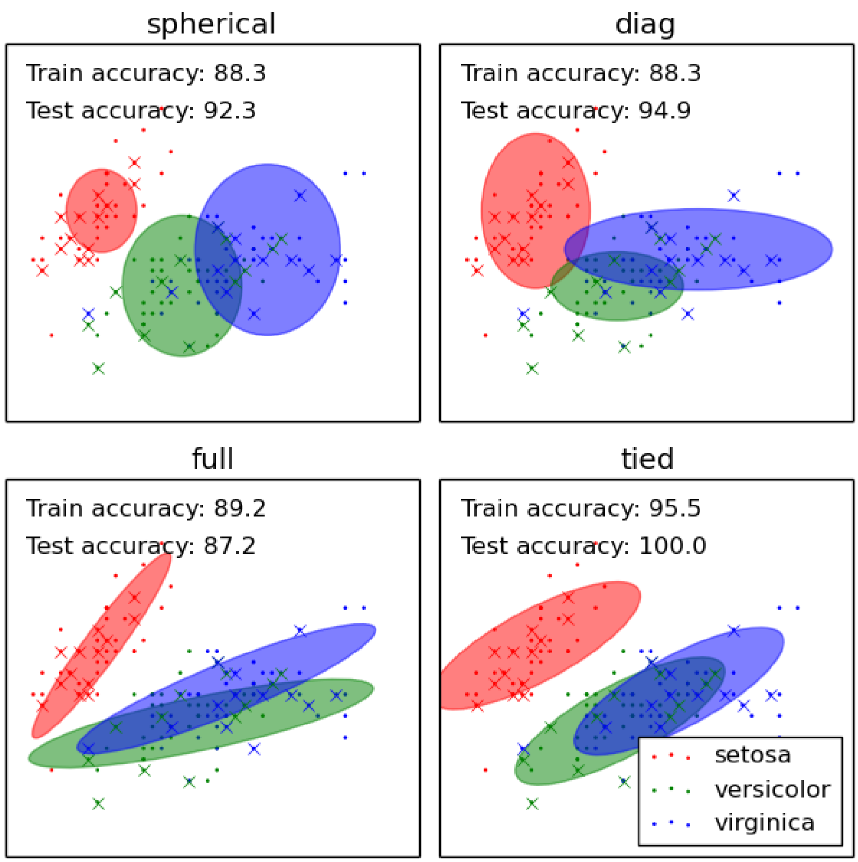
A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. One can think of mixture models as generalizing k-means clustering to incorporate information about the covariance structure 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.

5.1.1 GMM classifier

The `GMM` 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 `GMM.fit` method is provided that learns a Gaussian Mixture Model from train data. Given test data, it can assign to each sample the class of the Gaussian it mostly probably belong to using the `GMM.predict` method.

The GMM comes with different options to constrain the covariance of the difference classes estimated: spherical, diagonal, tied or full covariance.



Examples:

- See *GMM classification* for an example of using a GMM as a classifier on the iris dataset.
- See *Density Estimation for a mixture of Gaussians* for an example on plotting the density estimation.

Pros and cons of class `GMM`: expectation-maximization inference

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.

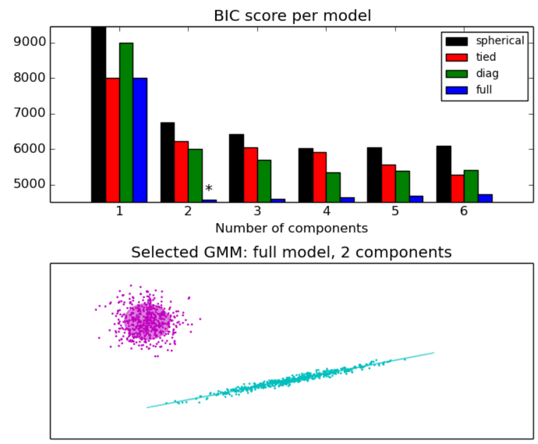
Cons

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 GMM

The BIC criterion can be used to select the number of components in a GMM 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). Note that using a *DPGMM* 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 GMM.

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.

5.1.2 VBGMM classifier: variational Gaussian mixtures

The `VBGMM` object implements a variant of the Gaussian mixture model with *variational inference* algorithms. The API is identical to `GMM`. It is essentially a middle-ground between `GMM` and `DPGMM`, as it has some of the properties of the Dirichlet process.

Pros and cons of class `VBGMM`: variational inference

Pros

Regularization due to the incorporation of prior information, variational solutions have less pathological special cases than expectation-maximization solutions. One can then use full covariance matrices in high dimensions or in cases where some components might be centered around a single point without risking divergence.

Cons

Bias to regularize a model one has to add biases. The variational algorithm will bias all the means towards the origin (part of the prior information adds a "ghost point" in the origin to every mixture component) and it will bias the covariances to be more spherical. It will also, depending on the concentration parameter, bias the cluster structure either towards uniformity or towards a rich-get-richer scenario.

Hyperparameters this algorithm needs an extra hyperparameter that might need experimental tuning via cross-validation.

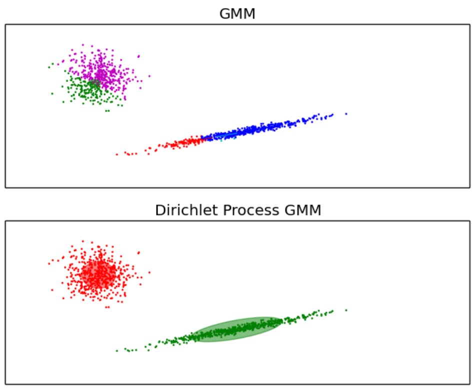
Estimation algorithm: variational inference

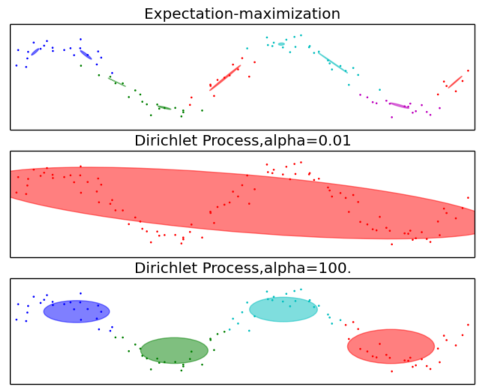
Variational inference is an extension of expectation-maximization that maximizes a lower bound on model evidence (including priors) instead of data likelihood. The principle behind variational methods is the same as expectation-maximization (that is both are iterative algorithms that alternate between finding the probabilities for each point to be generated by each mixture and fitting the mixtures 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 `alpha`. Specifying a high value of alpha leads more often to uniformly-sized mixture components, while specifying small (between 0 and 1) values will lead to some mixture components getting almost all the points while most mixture components will be centered on just a few of the remaining points.

5.1.3 DPGMM classifier: Infinite Gaussian mixtures

The `DPGMM` object implements a variant of the Gaussian mixture model with a variable (but bounded) number of components using the Dirichlet Process. The API is identical to `GMM`. This class doesn't require the user to choose the number of components, and at the expense of extra computational time the user only needs to specify a loose upper bound on this number and a concentration parameter.





The examples above compare Gaussian mixture models with fixed number of components, to DPGMM models. **On the left** the GMM is fitted with 5 components on a dataset composed of 2 clusters. We can see that the DPGMM is able to limit itself to only 2 components whereas the GMM fits the data fit too many components. Note that with very little observations, the DPGMM can take a conservative stand, and fit only one component. **On the right** we are fitting a dataset not well-depicted by a mixture of Gaussian. Adjusting the *alpha* parameter of the DPGMM controls the number of components used to fit this data.

Examples:

- See *Gaussian Mixture Model Ellipsoids* for an example on plotting the confidence ellipsoids for both `GMM` and `DPGMM`.
- *Gaussian Mixture Model Sine Curve* shows using `GMM` and `DPGMM` to fit a sine wave

Pros and cons of class `DPGMM`: Dirichlet process mixture model

Pros

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 Dirichlet process solution won't change much with changes to the parameters, leading to more stability and less tuning.

No need to specify the number of components only an upper bound of this number needs to be provided. Note however that the DPMM is not a formal model selection procedure, and thus provides no guarantee on the result.

Cons

Speed the extra parametrization necessary for variational inference and for the structure of the Dirichlet process can and will make inference slower, although not by much.

Bias as in variational techniques, but only more so, there are many implicit biases in the Dirichlet process and the inference algorithms, 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 mixtures. 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 chinese restaurant process analogy to help understanding it. The chinese restaurant process is a generative story for the Dirichlet process. Imagine a chinese restaurant with an infinite number of tables, at first all empty. When the first customer of the day arrives, he sits at the first table. Every following customer will then either sit on an occupied table with probability proportional to the number of customers in that table or sit in an entirely new table with probability proportional to the concentration parameter *alpha*. After a finite number of customers has sat, it is easy to see that only finitely many of the infinite tables will ever be used, and the higher the value of alpha the more total tables will be used. So the Dirichlet process does clustering with an unbounded number of mixture components by assuming a very asymmetrical prior structure over the assignments of points to components that is very concentrated (this property is known as rich-get-richer, as the full tables in the Chinese restaurant process only tend to get fuller as the simulation progresses).

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).

Derivation:

- See here the full derivation of this algorithm.

Variational Gaussian Mixture Models

The API is identical to that of the `GMM` class, the main difference being that it offers access to precision matrices as well as covariance matrices.

The inference algorithm is the one from the following paper:

- Variational Inference for Dirichlet Process Mixtures David Blei, Michael Jordan. Bayesian Analysis, 2006

While this paper presents the parts of the inference algorithm that are concerned with the structure of the dirichlet process, it does not go into detail in the mixture modeling part, which can be just as complex, or even more. For this reason we present here a full derivation of the inference algorithm and all the update and lower-bound equations. If you're not interested in learning how to derive similar algorithms yourself and you're not interested in changing/debugging the implementation in the scikit this document is not for you.

The complexity of this implementation is linear in the number of mixture components and data points. With regards to the dimensionality, it is linear when using `spherical` or `diag` and quadratic/cubic when using `tied` or `full`. For `spherical` or `diag` it is O(n_states * n_points * dimension) and for `tied` or `full` it is O(n_states * n_points * dimension^2 + n_states * dimension^3) (it is necessary to invert the covariance/precision matrices and compute its determinant, hence the cubic term).

This implementation is expected to scale at least as well as EM for the mixture of Gaussians.

Update rules for VB inference

Here the full mathematical derivation of the Variational Bayes update rules for Gaussian Mixture Models is given. The main parameters of the model, defined for any class $k \in [1..K]$ are the class proportion ϕ_k, the mean parameters μ_k, the covariance parameters Σ_k, which is characterized by variational Wishart density, $Wishart(a_k, \mathbf{B_k})$, where a is the degrees of freedom, and B is the scale matrix. Depending on the covariance parametrization, B_k can be a positive scalar, a positive vector or a Symmetric Positive Definite matrix.

The spherical model The model then is

$$\begin{array}{rcl} \phi_k & \sim & Beta(1, \alpha_1) \\ \mu_k & \sim & Normal(0, \mathbf{I}) \\ \sigma_k & \sim & Gamma(1, 1) \\ z_i & \sim & SBP(\phi) \\ X_t & \sim & Normal(\mu_{z_i}, \frac{1}{\sigma_{z_i}}\mathbf{I}) \end{array}$$

The variational distribution we'll use is

$$\begin{array}{rcl} \phi_k & \sim & Beta(\gamma_{k,1}, \gamma_{k,2}) \\ \mu_k & \sim & Normal(\nu_{\mu_k}, \mathbf{I}) \\ \sigma_k & \sim & Gamma(a_k, b_k) \\ z_i & \sim & Discrete(\nu_{z_i}) \end{array}$$

The bound The variational bound is

$$\begin{array}{rcl} \log P(X) & \geq & \sum_k (E_q[\log P(\phi_k)] - E_q[\log Q(\phi_k)]) \\ & & + \sum_k \left(E_q[\log P(\mu_k)] - E_q[\log Q(\mu_k)] \right) \\ & & + \sum_k \left(E_q[\log P(\sigma_k)] - E_q[\log Q(\sigma_k)] \right) \\ & & + \sum_i \left(E_q[\log P(z_i)] - E_q[\log Q(z_i)] \right) \\ & & + \sum_i E_q[\log P(X_t)] \end{array}$$

The bound for ϕ_k

$$\begin{array}{rcl} E_q[\log Beta(1, \alpha)] - E[\log Beta(\gamma_{k,1}, \gamma_{k,2})] & = & \log \Gamma(1+\alpha) - \log \Gamma(\alpha) \\ & & +(\alpha - 1)(\Psi(\gamma_{k,2}) - \Psi(\gamma_{k,1} + \gamma_{k,2})) \\ & & - \log \Gamma(\gamma_{k,1} + \gamma_{k,2}) + \log \Gamma(\gamma_{k,1}) + \log \Gamma(\gamma_{k,2}) \\ & & -(\gamma_{k,1} - 1)(\Psi(\gamma_{k,1}) - \Psi(\gamma_{k,1} + \gamma_{k,2})) \\ & & -(\gamma_{k,2} - 1)(\Psi(\gamma_{k,2}) - \Psi(\gamma_{k,1} + \gamma_{k,2})) \end{array}$$

The bound for μ_k

$$\begin{array}{rl} & E_q[\log P(\mu_k)] - E_q[\log Q(\mu_k)] \\ = & \int d\mu_f q(\mu_f) \log P(\mu_f) - \int d\mu_f q(\mu_f) \log Q(\mu_f) \\ = & -\frac{D}{2} \log 2\pi - \frac{1}{2} ||\nu_{\mu_k}||^2 - \frac{D}{2} + \frac{D}{2} \log 2\pi e \end{array}$$

The bound for σ_k

Here I'll use the inverse scale parametrization of the gamma distribution.

$$\begin{array}{rl} & E_q[\log P(\sigma_k)] - E_q[\log Q(\sigma_k)] \\ = & \log \Gamma(a_k) - (a_k - 1)\Psi(a_k) - \log b_k + a_k - \frac{a_k}{b_k} \end{array}$$

The bound for z

$$\begin{array}{rl} & E_q[\log P(z)] - E_q[\log Q(z)] \\ = & \sum_k \left(\left(\sum_{j=k+1}^K \nu_{z_{i,j}} \right) (\Psi(\gamma_{k,2}) - \Psi(\gamma_{k,1} + \gamma_{k,2})) + \nu_{z_{i,k}} (\Psi(\gamma_{k,1}) - \Psi(\gamma_{k,1} + \gamma_{k,2})) - \log \nu_{z_{i,k}} \right) \end{array}$$

The bound for X

Recall that there is no need for a $Q(X)$ so this bound is just

$$E_q[\log P(X_i)] \quad = \quad \sum_k \nu_{z_k} \left(-\frac{D}{2}\log 2\pi + \frac{D}{2}(\Psi(a_k) - \log(b_k)) - \frac{a_k}{2b_k}(||X_i - \nu_{\mu_k}||^2 + D) - \log 2\pi e \right)$$

For simplicity I'll later call the term inside the parenthesis $E_q[\log P(X_i|z_i = k)]$

The updates **Updating γ**

$$\begin{array}{rcl} \gamma_{k,1} & = & 1 + \sum_i \nu_{z_{i,k}} \\ \gamma_{k,2} & = & \alpha + \sum_i \sum_{j>k} \nu_{z_{i,j}}. \end{array}$$

Updating μ

The updates for mu essentially are just weighted expectations of X regularized by the prior. We can see this by taking the gradient of the bound with regards to ν_μ and setting it to zero. The gradient is

$$\nabla L = -\nu_{\mu_k} + \sum_i \frac{\nu_{z_{i,k}} b_k}{a_k}(X_i + -\nu_\mu)$$

so the update is

$$\nu_{\mu_k} = \frac{\sum_i \frac{\nu_{z_{i,k}} b_k}{a_k} X_i}{1 + \sum_i \frac{\nu_{z_{i,k}} b_k}{a_k}}$$

Updating a and b

For some odd reason it doesn't really work when you derive the updates for a and b using the gradients of the lower bound (terms involving the Ψ' function show up and a is hard to isolate). However, we can use the other formula,

$$\log Q(\sigma_k) = E_{v \neq \sigma_k}[\log P] + const$$

All the terms not involving σ_k get folded over into the constant and we get two terms: the prior and the probability of X. This gives us

$$\log Q(\sigma_k) = -\sigma_k + \frac{D}{2}\sum_i \nu_{z_{i,k}} \log \sigma_k - \frac{\sigma_k}{2}\sum_i \nu_{z_{i,k}}(||X_i - \mu_k||^2 + D)$$

This is the log of a gamma distribution, with $a_k = 1 + \frac{D}{2}\sum_i \nu_{z_{i,k}}$ and

$$b_k = 1 + \frac{1}{2}\sum_i \nu_{z_{i,k}}(||X_i - \mu_k||^2 + D).$$

You can verify this by normalizing the previous term.

Updating z

$$\log \nu_{z_{i,k}} \propto \Psi(\gamma_{k,1}) - \Psi(\gamma_{k,1} + \gamma_{k,2}) + E_Q[\log P(X_i|z_i = k)] + \sum_{j<k} (\Psi(\gamma_{j,2}) - \Psi(\gamma_{j,1} + \gamma_{j,2})).$$

The diagonal model The model then is

$$\begin{array}{rcl} \phi_k & \sim & Beta(1, \alpha_1) \\ \mu_k & \sim & Normal(0, \mathbf{I}) \\ \sigma_{k,d} & \sim & Gamma(1, 1) \\ z_i & \sim & SBP(\phi) \\ X_t & \sim & Normal(\mu_{z_i}, \boldsymbol{\sigma}_{\boldsymbol{z_i}}^{-1}) \end{array}$$

Tha variational distribution we'll use is

$$\begin{array}{rcl} \phi_k & \sim & Beta(\gamma_{k,1}, \gamma_{k,2}) \\ \mu_k & \sim & Normal(\nu_{\mu_k}, \mathbf{I}) \\ \sigma_{k,d} & \sim & Gamma(a_{k,d}, b_{k,d}) \\ z_i & \sim & Discrete(\nu_{z_i}) \end{array}$$

The lower bound The changes in this lower bound from the previous model are in the distributions of σ (as there are a lot more σ s now) and X.

The bound for $\sigma_{k,d}$ is the same bound for σ_k and can be safely omitted.

The bound for X :

The main difference here is that the precision matrix $\boldsymbol{\sigma_k}$ scales the norm, so we have an extra term after computing the expectation of $\mu_k^T \boldsymbol{\sigma_k} \mu_k$, which is $\nu_{\mu_k}^T \boldsymbol{\sigma_k} \nu_{\mu_k} + \sum_d \sigma_{k,d}$. We then have

$$\begin{array}{rcl} E_q[\log P(X_i)] & = & \sum_k \nu_{z_k} \Big(-\frac{D}{2} \log 2\pi + \frac{1}{2} \sum_d (\Psi(a_{k,d}) - \log(b_{k,d})) \\ & & -\frac{1}{2}((X_i - \nu_{\mu_k})^T \frac{\boldsymbol{a_k}}{\boldsymbol{b_k}} (X_i - \nu_{\mu_k}) + \sum_d \sigma_{k,d}) - \log 2\pi e \Big) \end{array}$$

The updates The updates only chance for μ (to weight them with the new σ), z (but the change is all folded into the $E_q[P(X_i|z_i = k)]$ term), and the a and b variables themselves.

The update for μ

$$\nu_{\mu_k} = \left(\mathbf{I} + \sum_i \frac{\nu_{z_{i,k}} \mathbf{b_k}}{\mathbf{a_k}} \right)^{-1} \left(\sum_i \frac{\nu_{z_{i,k}} b_k}{a_k} X_i \right)$$

The updates for a and b

Here we'll do something very similar to the spheric model. The main difference is that now each $\sigma_{k,d}$ controls only one dimension of the bound:

$$\log Q(\sigma_{k,d}) = -\sigma_{k,d} + \sum_i \nu_{z_{i,k}} \frac{1}{2} \log \sigma_{k,d} - \frac{\sigma_{k,d}}{2} \sum_i \nu_{z_{i,k}} ((X_{i,d} - \mu_{k,d})^2 + 1)$$

Hence

$$a_{k,d} = 1 + \frac{1}{2} \sum_i \nu_{z_{i,k}}$$

$$b_{k,d} = 1 + \frac{1}{2} \sum_i \nu_{z_{i,k}} ((X_{i,d} - \mu_{k,d})^2 + 1)$$

The tied model The model then is

$$\begin{array}{rcl} \phi_k & \sim & Beta(1, \alpha_1) \\ \mu_k & \sim & Normal(0, \mathbf{I}) \\ \Sigma & \sim & Wishart(D, \mathbf{I}) \\ z_i & \sim & SBP(\phi) \\ X_t & \sim & Normal(\mu_{z_i}, \Sigma^{-1}) \end{array}$$

Tha variational distribution we'll use is

$$\begin{array}{rcl} \phi_k & \sim & Beta(\gamma_{k,1}, \gamma_{k,2}) \\ \mu_k & \sim & Normal(\nu_{\mu_k}, \mathbf{I}) \\ \Sigma & \sim & Wishart(a, \mathbf{B}) \\ z_i & \sim & Discrete(\nu_{z_i}) \end{array}$$

The lower bound There are two changes in the lower-bound: for Σ and for X.

The bound for Σ

$$\begin{array}{r}\frac{D^2}{2}\log 2+\sum_d \log\Gamma(\frac{D+1-d}{2})\\ -\frac{aD}{2}\log 2+\frac{a}{2}\log|\mathbf{B}|+\sum_d\log\Gamma(\frac{a+1-d}{2})\\ +\frac{a-D}{2}\left(\sum_d \Psi\left(\frac{a+1-d}{2}\right)+D\log 2+\log|\mathbf{B}|\right)\\ +\frac{1}{2}a\mathbf{tr}[\mathbf{B}-\mathbf{I}]\end{array}$$

The bound for X

$$\begin{array}{rcl}E_q[\log P(X_i)] & = & \sum_k \nu_{z_k}\Big(-\frac{D}{2}\log 2\pi+\frac{1}{2}\left(\sum_d\Psi\left(\frac{a+1-d}{2}\right)+D\log 2+\log|\mathbf{B}|\right)\\ & & -\frac{1}{2}((X_i-\nu_{\mu_k})a\mathbf{B}(X_i-\nu_{\mu_k})+a\mathbf{tr}(\mathbf{B}))-\log 2\pi e\Big)\end{array}$$

The updates As in the last setting, what changes are the trivial update for z, the update for μ and the update for a and $\mathbf{B}$.

The update for μ

$$\nu_{\mu_k}=\left(\mathbf{I}+a\mathbf{B}\sum_i \nu_{z_{i,k}}\right)^{-1}\left(a\mathbf{B}\sum_i \nu_{z_{i,k}}X_i\right)$$

The update for a **and** B

As this distribution is far too complicated I'm not even going to try going at it the gradient way.

$$\log Q(\Sigma)=+\frac{1}{2}\log|\Sigma|-\frac{1}{2}\mathbf{tr}[\Sigma]+\sum_i\sum_k \nu_{z_{i,k}}\left(+\frac{1}{2}\log|\Sigma|-\frac{1}{2}((X_i-\nu_{\mu_k})^T\Sigma(X_i-\nu_{\mu_k})+\mathbf{tr}[\Sigma])\right)$$

which non-trivially (seeing that the quadratic form with Σ in the middle can be expressed as the trace of something) reduces to

$$\log Q(\Sigma)=+\frac{1}{2}\log|\Sigma|-\frac{1}{2}\mathbf{tr}[\Sigma]+\sum_i\sum_k \nu_{z_{i,k}}\left(+\frac{1}{2}\log|\Sigma|-\frac{1}{2}(\mathbf{tr}[(X_i-\nu_{\mu_k})(X_i-\nu_{\mu_k})^T\Sigma]+\mathbf{tr}[I\Sigma])\right)$$

hence this (with a bit of squinting) looks like a wishart with parameters

$$a=2+D+T$$

and

$$\mathbf{B}=\left(\mathbf{I}+\sum_i\sum_k \nu_{z_{i,k}}(X_i-\nu_{\mu_k})(X_i-\nu_{\mu_k})^T\right)^{-1}$$

The full model

The model then is

$$\begin{array}{rcl}\phi_k & \sim & Beta(1,\alpha_1)\\ \mu_k & \sim & Normal(0,\mathbf{I})\\ \Sigma_k & \sim & Wishart(D,\mathbf{I})\\ z_i & \sim & SBP(\phi)\\ X_t & \sim & Normal(\mu_{z_i},\Sigma_{z,i}^{-1})\end{array}$$

The variational distribution we'll use is

$$\begin{array}{rcl}\phi_k & \sim & Beta(\gamma_{k,1},\gamma_{k,2})\\ \mu_k & \sim & Normal(\nu_{\mu_k},\mathbf{I})\\ \Sigma_k & \sim & Wishart(a_k,\mathbf{B_k})\\ z_i & \sim & Discrete(\nu_{z_i})\end{array}$$

The lower bound All that changes in this lower bound in comparison to the previous one is that there are K priors on different Σ precision matrices and there are the correct indices on the bound for X.

The updates All that changes in the updates is that the update for mu uses only the proper sigma and the updates for a and B don't have a sum over K, so

$$\nu_{\mu_k} = \left(\mathbf{I} + a_k \mathbf{B_k} \sum_i \nu_{z_{i,k}}\right)^{-1} \left(a_k \mathbf{B_k} \sum_i \nu_{z_{i,k}} X_i\right)$$

$$a_k = 2 + D + \sum_i \nu_{z_{i,k}}$$

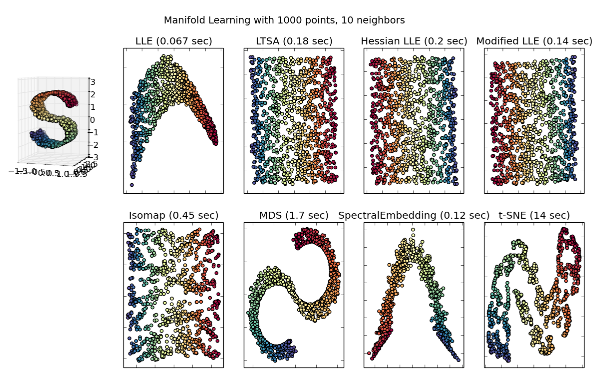
and

$$\mathbf{B} = \left(\left(\sum_i \nu_{z_{i,k}} + 1\right) \mathbf{I} + \sum_i \nu_{z_{i,k}} (X_i - \nu_{\mu_k})(X_i - \nu_{\mu_k})^T\right)^{-1}$$

5.2 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]

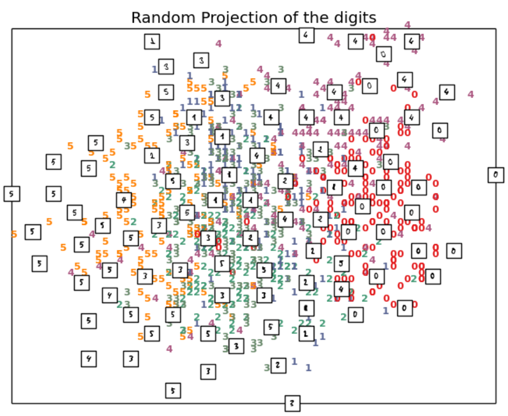


Manifold learning is an approach to non-linear dimensionality reduction. Algorithms for this task are based on the idea that the dimensionality of many data sets is only artificially high.

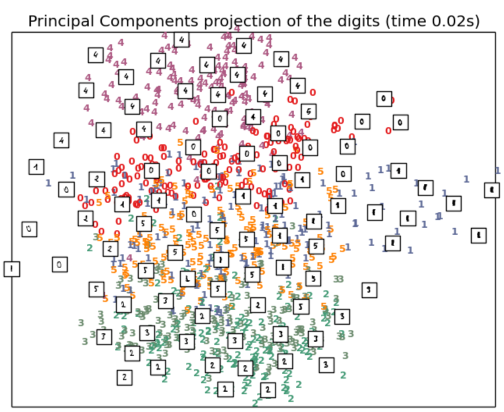
5.2.1 Introduction

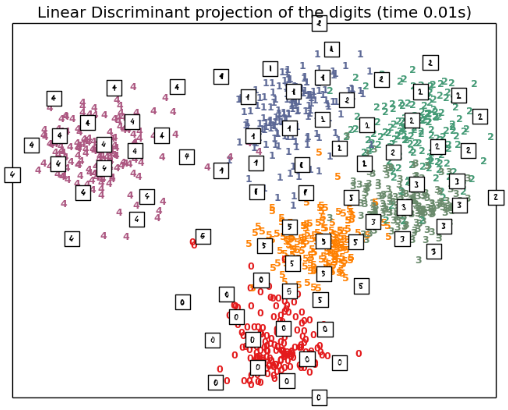
High-dimensional datasets can be very difficult to visualize. While data in two or three dimensions can be plotted to show the inherent structure of the data, equivalent high-dimensional plots are much less intuitive. To aid visualization of the structure of a dataset, the dimension must be reduced in 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.

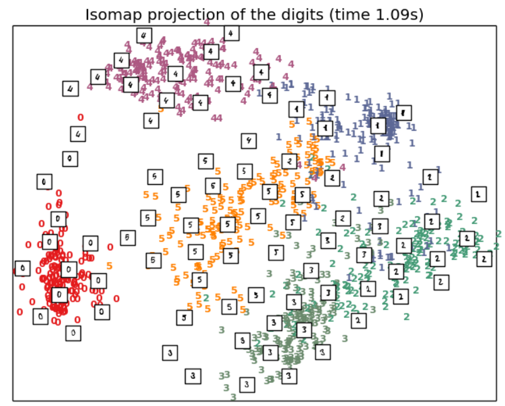
Examples:

- 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 sklearn are summarized below

5.2.2 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

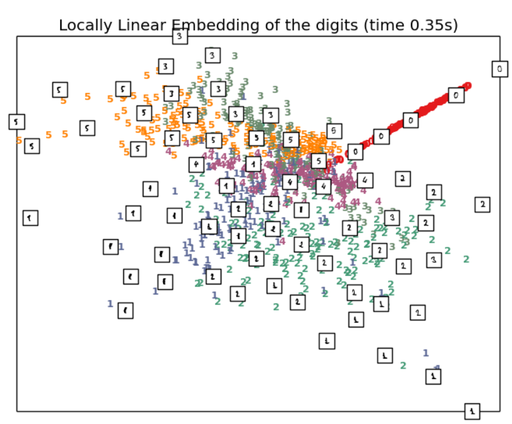
References:

- "A global geometric framework for nonlinear dimensionality reduction" Tenenbaum, J.B.; De Silva, V.; & Langford, J.C. Science 290 (5500)

5.2.3 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

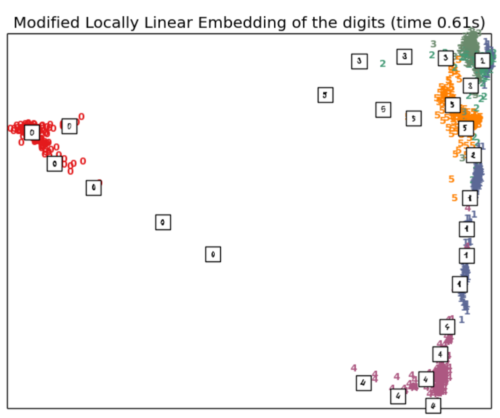
References:

- "Nonlinear dimensionality reduction by locally linear embedding" Roweis, S. & Saul, L. Science 290:2323 (2000)

5.2.4 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 steps 1 and 3.
3. **Partial Eigenvalue Decomposition**. Same as standard LLE

The overall complexity of MLLE is $O[D\log(k)N\log(N)] + O[DNk^3] + O[N(k-D)k^2] + O[dN^2]$.

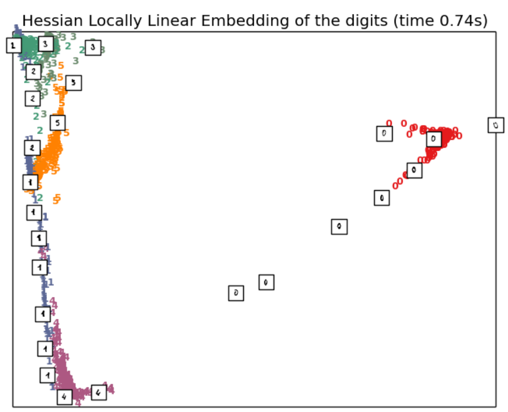
- N : number of training data points
- D : input dimension
- k : number of nearest neighbors
- d : output dimension

References:

- "MLLE: Modified Locally Linear Embedding Using Multiple Weights" Zhang, Z. & Wang, J.

5.2.5 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)

5.2.6 Spectral Embedding

Spectral Embedding (also known as Laplacian Eigenmaps) is one method to calculate non-linear embedding. It 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 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 $O[D\log(k)N\log(N)] + O[DNk^3] + O[dN^2]$.

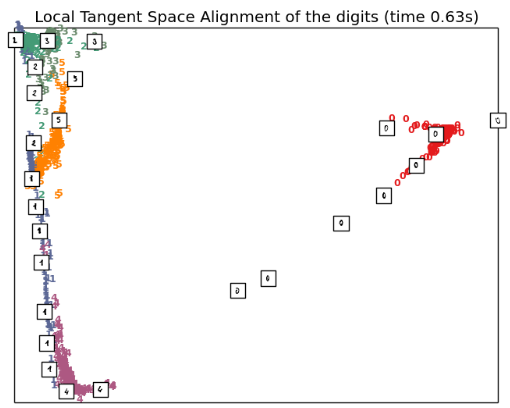
- N : number of training data points
- D : input dimension
- k : number of nearest neighbors
- d : output dimension

References:

- "Laplacian Eigenmaps for Dimensionality Reduction and Data Representation" M. Belkin, P. Niyogi, Neural Computation, June 2003; 15 (6):1373-1396

5.2.7 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 $O[DNk^3] + O[k^2 d]$. 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 $O[D \log(k) N \log(N)] + O[DNk^3] + O[k^2 d] + O[dN^2]$.

- N : number of training data points
- D : input dimension
- k : number of nearest neighbors
- d : output dimension

References:

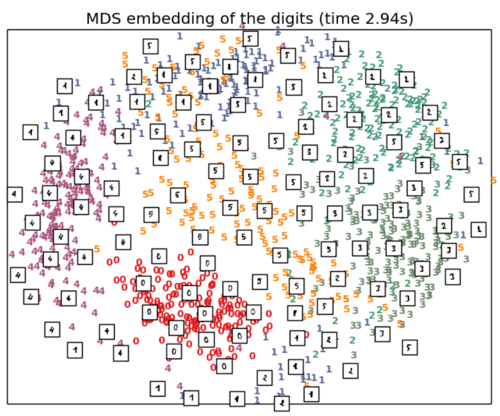
- "Principal manifolds and nonlinear dimensionality reduction via tangent space alignment" Zhang, Z. & Zha, H. Journal of Shanghai Univ. 8:406 (2004)

5.2.8 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, is a technique used for analyzing similarity or dissimilarity data. `MDS` 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

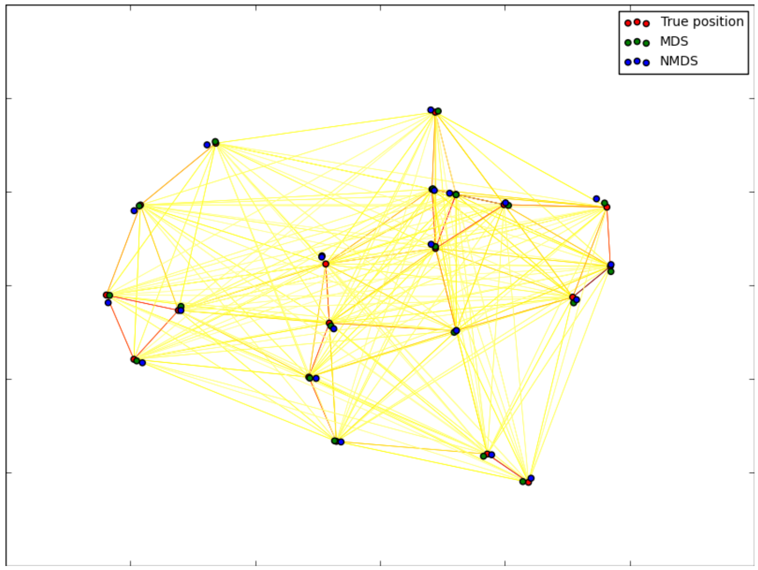
The simplest metric `MDS` model, called *absolute MDS*, disparities are defined by $\hat{d}_{ij} = S_{ij}$. With absolute MDS, the value S_{ij} should then correspond exactly to the distance between point i and j in the embedding point.

Most commonly, disparities are set to $\hat{d}_{ij} = bS_{ij}$.

Nonmetric MDS

Non metric `MDS` focuses on the ordination of the data. If $S_{ij} < S_{kl}$, 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)

5.2.9 t-distributed Stochastic Neighbor Embedding (t-SNE)

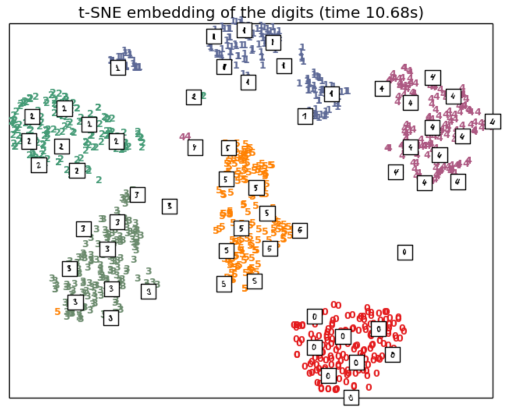
t-SNE (`TSNE`) converts affinities of data points to probabilities. The affinities in the original space are represented by Gaussian joint probabilities and the affinities in the embedded space are represented by Student's t-distributions. The Kullback-Leibler (KL) divergence of the joint probabilities 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 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 three parameters that control the optimization of t-SNE:

- early exaggeration factor
- learning rate
- maximum number of iterations

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).

Standard 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 on $O[dN^2]$, where d is the number of output dimensions and N is the number of samples.

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.

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 implie that the data cannot be correctly classified by a supervised model. It might be the case that 2 dimensions are not enough low 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.

5.2.10 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.

5.3 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 take different kinds of matrix as input. On one hand, `MeanShift` and `KMeans` take data matrices of shape [n_samples, n_features]. These can be obtained from the classes in the `sklearn.feature_extraction` module. On the other hand, `AffinityPropagation` and `SpectralClustering` take similarity matrices of shape [n_samples, n_samples]. These can be obtained from the functions in the `sklearn.metrics.pairwise` module. In other words, `MeanShift` and `KMeans` work with points in a vector space, whereas `AffinityPropagation` and `SpectralClustering` can work with arbitrary objects, as long as a similarity measure exists for such objects.

5.3.1 Overview of clustering methods

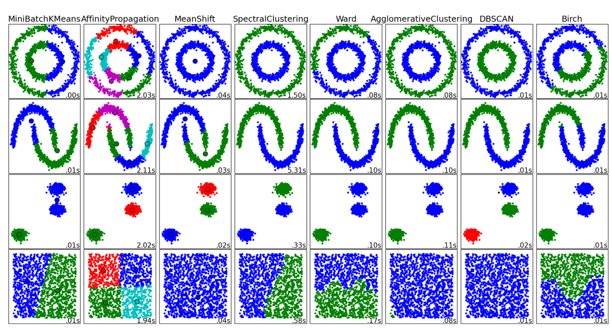


Figure 5.2: A comparison of the clustering algorithms in scikit-learn

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	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters, 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
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers
Birch	branching factor, threshold, optional global clusterer.	Large `n_clusters` and `n_samples`	Large dataset, outlier removal, data reduction.	Euclidean distance between points

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.

5.3.2 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 <inertia>* or within-cluster sum-of-squares. 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.

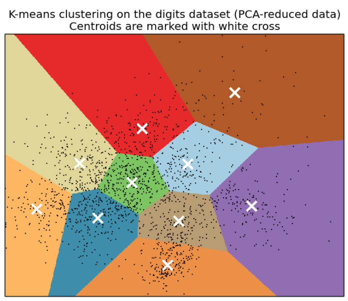
The k-means algorithm divides a set of N samples X into K disjoint clusters C, each described by the mean μ_j of the samples in the cluster. The means are commonly called the cluster "centroids"; note that they are not, in general, points from X, although they live in the same space. The K-means algorithm aims to choose centroids that minimise the *inertia*, or within-cluster sum of squared criterion:

$$\sum_{i=0}^{n}\min_{\mu_j \in C}(||x_j - \mu_i||^2)$$

Inertia, or the within-cluster sum of squares criterion, 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 *PCA <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='kmeans++'` 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.

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:

- *A demo of K-Means clustering on the handwritten digits data*: Clustering handwritten digits

References:

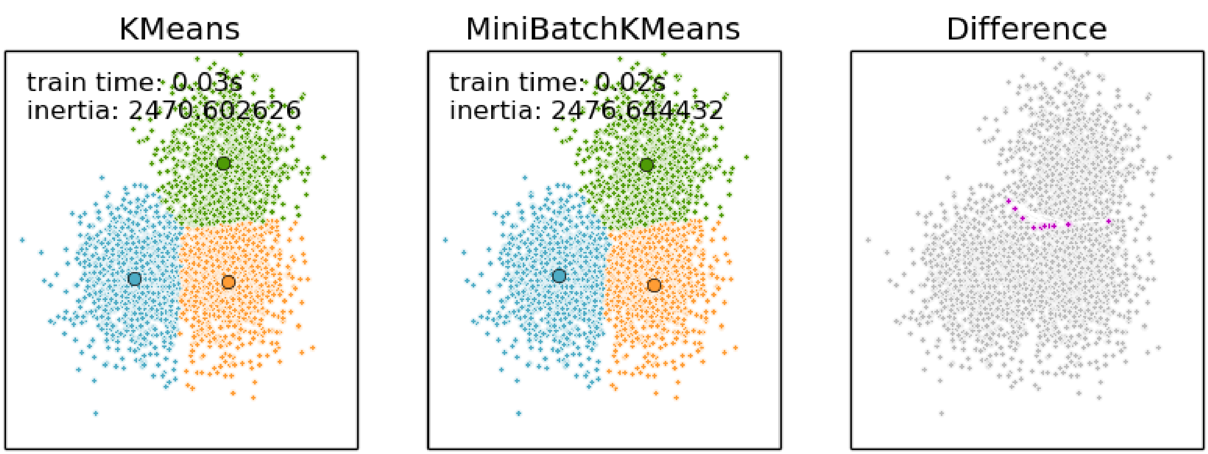
- "k-means++: The advantages of careful seeding" Arthur, David, and Sergei Vassilvitskii, *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics (2007)

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:

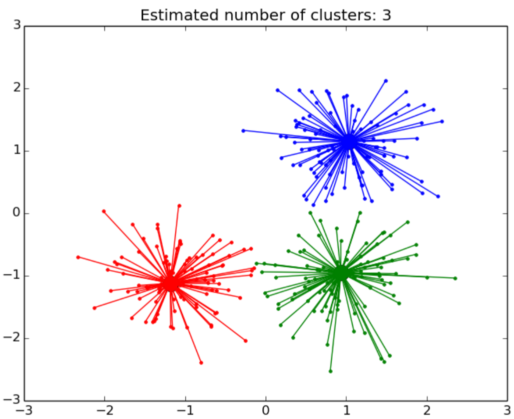
- *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*: Comparison of KMeans and MiniBatchKMeans
- *Clustering text documents using k-means*: Document clustering using sparse MiniBatchKMeans
- *Online learning of a dictionary of parts of faces*

References:

- "Web Scale K-Means clustering" D. Sculley, *Proceedings of the 19th international conference on World wide web* (2010)

5.3.3 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*.

The main drawback of Affinity Propagation is its complexity. The algorithm has a time complexity of the order $O(N^2 T)$, 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:

$$r(i,k) \leftarrow s(i,k) - max[a(i,\acute{k}) + s(i,\acute{k}) \forall \acute{k} \neq k]$$

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:

$$a(i,k) \leftarrow min[0, r(k,k) + \sum_{\acute{i}~s.t.~\acute{i} \notin \{i,k\}} r(\acute{i},k)]$$

To begin with, all values for r and a are set to zero, and the calculation of each iterates until convergence.

5.3.4 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:

$$x_i^{t+1} = x_i^t + m(x_i^t)$$

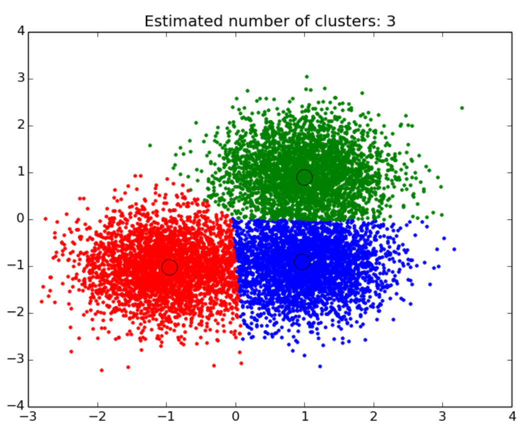
Where $N(x_i)$ is the neighborhood of samples within a given distance around x_i and m is the *mean shift* vector that is computed for each centroid that points towards a region of the maximum increase in the density of points. This is computed using the following equation, effectively updating a centroid 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.

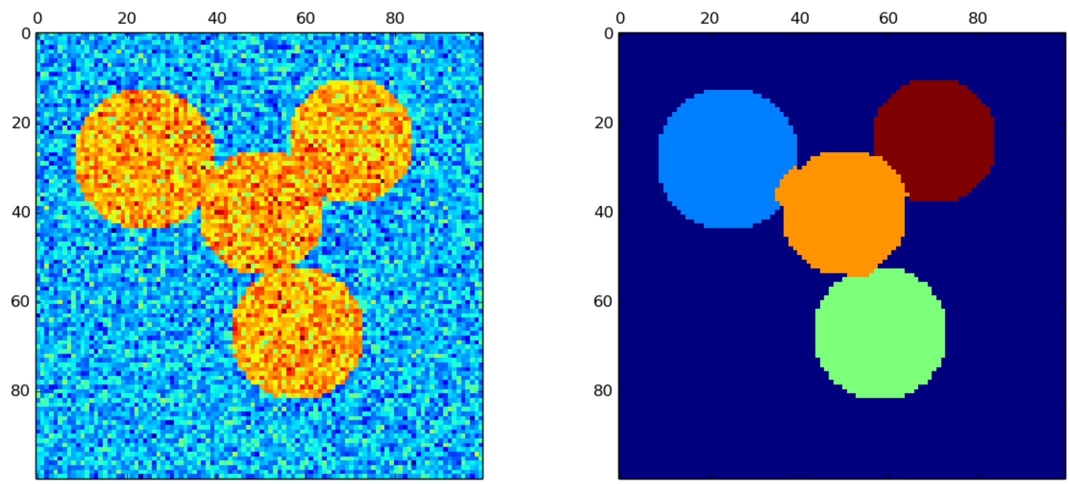
References:

- "Mean shift: A robust approach toward feature space analysis." D. Comaniciu, & P. Meer *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2002)

5.3.5 Spectral clustering

`SpectralClustering` does a low-dimension embedding of the affinity matrix between samples, followed by a KMeans in the low dimensional space. It is especially efficient if the affinity matrix is sparse and the pyamg module is installed. SpectralClustering requires the number of clusters to be specified. It works well for a small number of clusters but is not advised when using many clusters.

For two clusters, it 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: graph vertices are pixels, and edges of the similarity graph are a function of the 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 Lena in regions*: Spectral clustering to split the image of lena in regions.

Different label assignment strategies

Different label assignment strategies can be used, corresponding to the `assign_labels` parameter of `SpectralClustering`. The `"kmeans"` strategy can match finer details of the data, but it can be more unstable. In particular, unless you control the `random_state`, it may not be reproducible from run-to-run, as it depends on a random initialization. On the other hand, the `"discretize"` strategy is 100% reproducible, but it tends to create parcels of fairly even and geometrical shape.

`assign_labels="kmeans"`	`assign_labels="discretize"`
	

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

5.3.6 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:

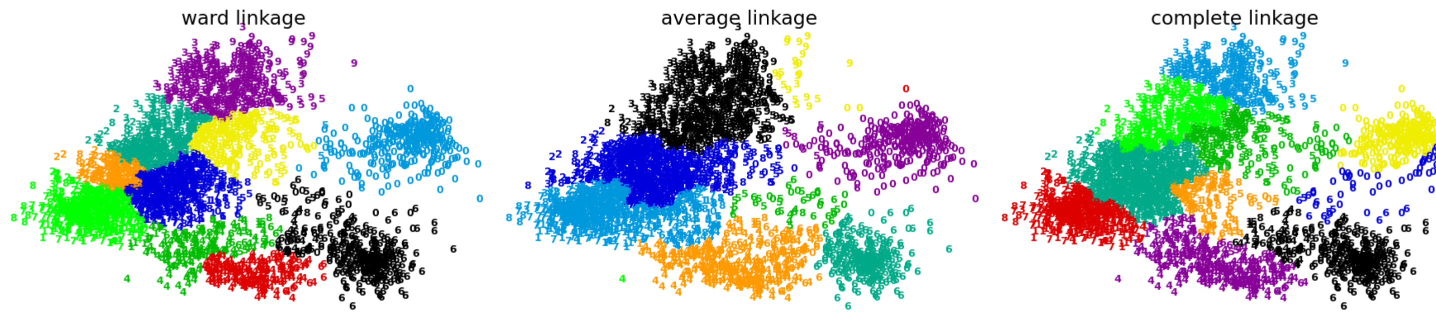
- **Ward** minimizes the sum of squared differences within all clusters. It is a variance-minimizing approach and in this sense is similar to the k-means objective function but tackled with an agglomerative hierarchical approach.
- **Maximum** or **complete linkage** minimizes the maximum distance between observations of pairs of clusters.
- **Average linkage** minimizes the average of the distances between all observations of pairs of clusters.

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 and average linkage

AgglomerativeClustering supports Ward, average, and complete linkage strategies.

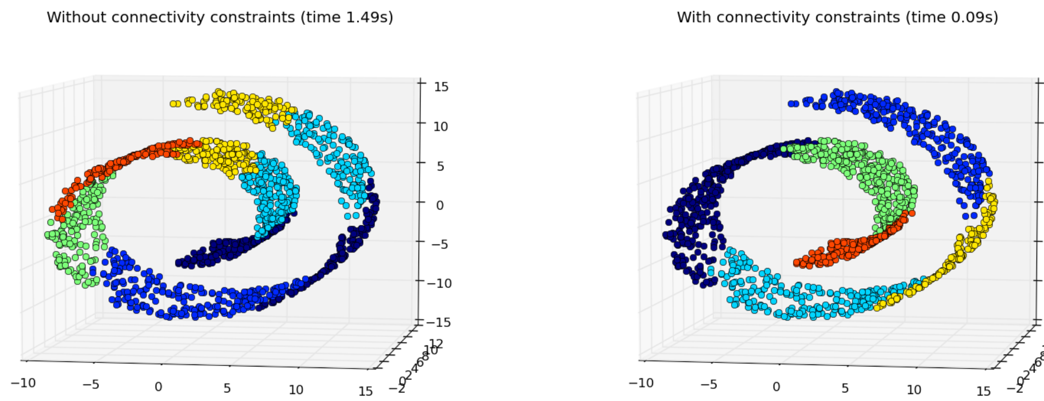


Agglomerative cluster has a "rich get richer" behavior that leads to uneven cluster sizes. In this regard, complete linkage is the worst strategy, and Ward gives the most regular sizes. However, the affinity (or distance used in clustering) cannot be varied with Ward, thus for non Euclidean metrics, average linkage is a good alternative.

> **Examples:**
>
> - *Various Agglomerative Clustering on a 2D embedding of digits*: exploration of the different linkage strategies in a real dataset.

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.

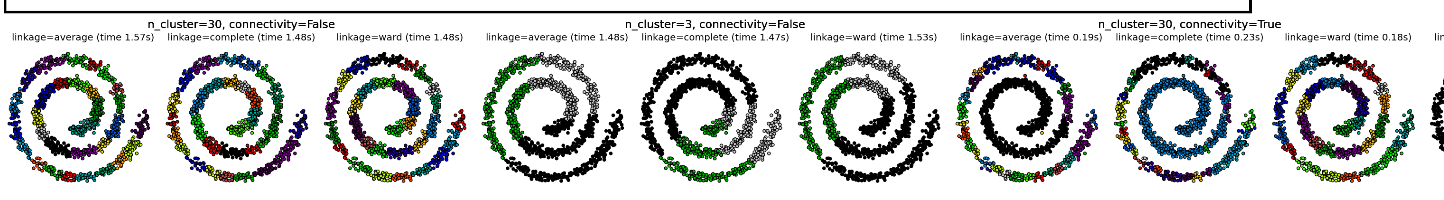
The connectivity constraints are imposed via an connectivity matrix: a scipy sparse matrix that has elements only at the intersection of a row and a column with indices of the dataset that should be connected. This matrix can be constructed from a-priori information: for instance, you may wish to cluster 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 *Lena* example.

Examples:

- *A demo of structured Ward hierarchical clustering on Lena image*: Ward clustering to split the image of lena 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 average and complete linkage

Connectivity constraints and 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*).

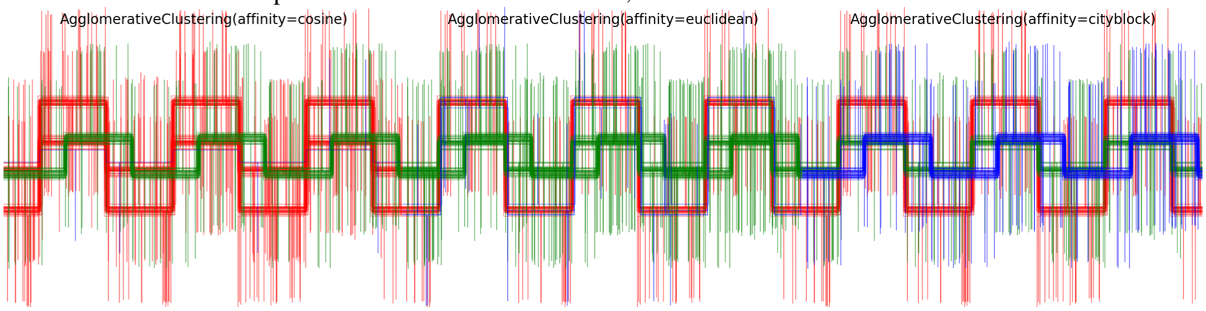


Varying the metric

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: ie many of the features are zero, as in text mining using occurences 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*

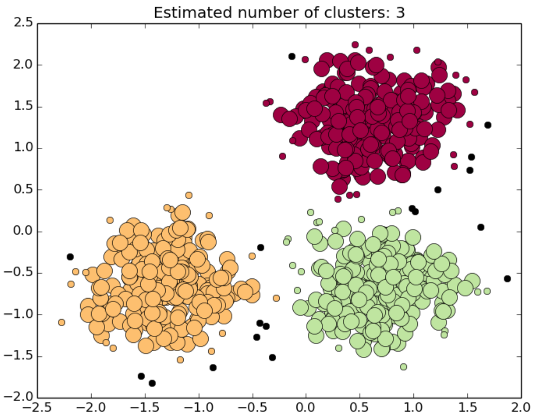
5.3.7 DBSCAN

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 by 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. Further, any cluster has at least `min_samples` points in it, following the definition of a core sample. For any sample that is not a core sample, and does have a distance higher than `eps` to any core sample, it is considered an outlier by the algorithm.

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 algorithm is non-deterministic, but the core samples will always belong to the same clusters (although the labels may be different). The non-determinism comes from deciding to which cluster a non-core sample belongs. A non-core sample can have 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, where the order is determined randomly. Other than the ordering of the dataset, the algorithm is deterministic, making the results relatively stable between runs on the same data.
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`.

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

5.3.8 Birch

The `Birch` builds a tree called the Characteristic Feature Tree (CFT) for the given data. The data is essentially lossy compressed to a set of Characteristic Feature nodes (CF Nodes). The CF Nodes have a number of subclusters called Characteristic 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.

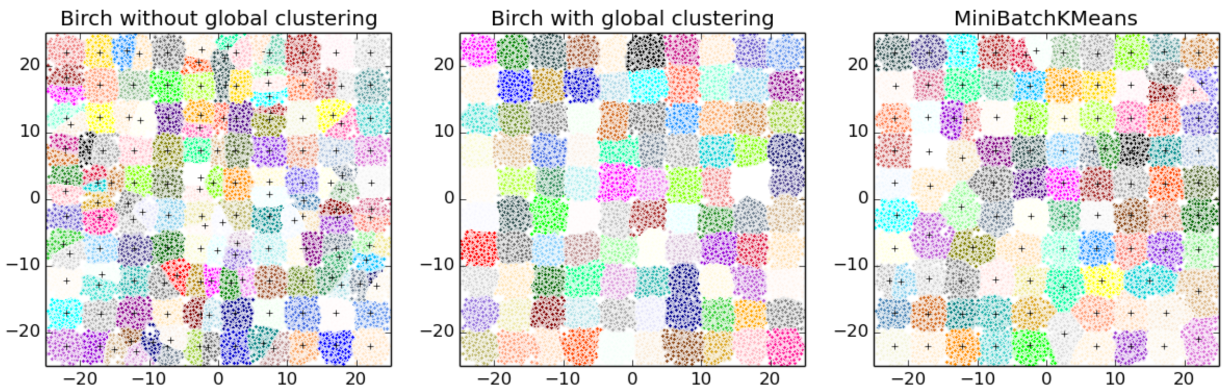
Birch or MiniBatchKMeans?

- 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. http://www.cs.sfu.ca/CourseCentral/459/han/papers/zhang96.pdf
- Roberto Perdisci JBirch - Java implementation of BIRCH clustering algorithm https://code.google.com/p/jbirch/

5.3.9 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 that members of different classes according to some similarity metric.

Adjusted Rand index

Presentation and usage

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

Presentation and usage

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).
- **Bounded range [0, 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 AMI 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, **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.

The normalized mutual information is defined as

$$\text{NMI}(U,V) = \frac{\text{MI}(U,V)}{\sqrt{H(U)H(V)}}$$

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, from Vinh, Epps, and Bailey, (2009). 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:

$$\text{AMI} = \frac{\text{MI} - E[\text{MI}]}{\max(H(U), H(V)) - E[\text{MI}]}$$

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
- 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.
- 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
- Wikipedia entry for the (normalized) Mutual Information
- Wikipedia entry for the Adjusted Mutual Information

Homogeneity, completeness and V-measure

Presentation and usage

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...
```

The V-measure is actually equivalent to the mutual information (NMI) discussed above normalized by the sum of the label entropies [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)
```

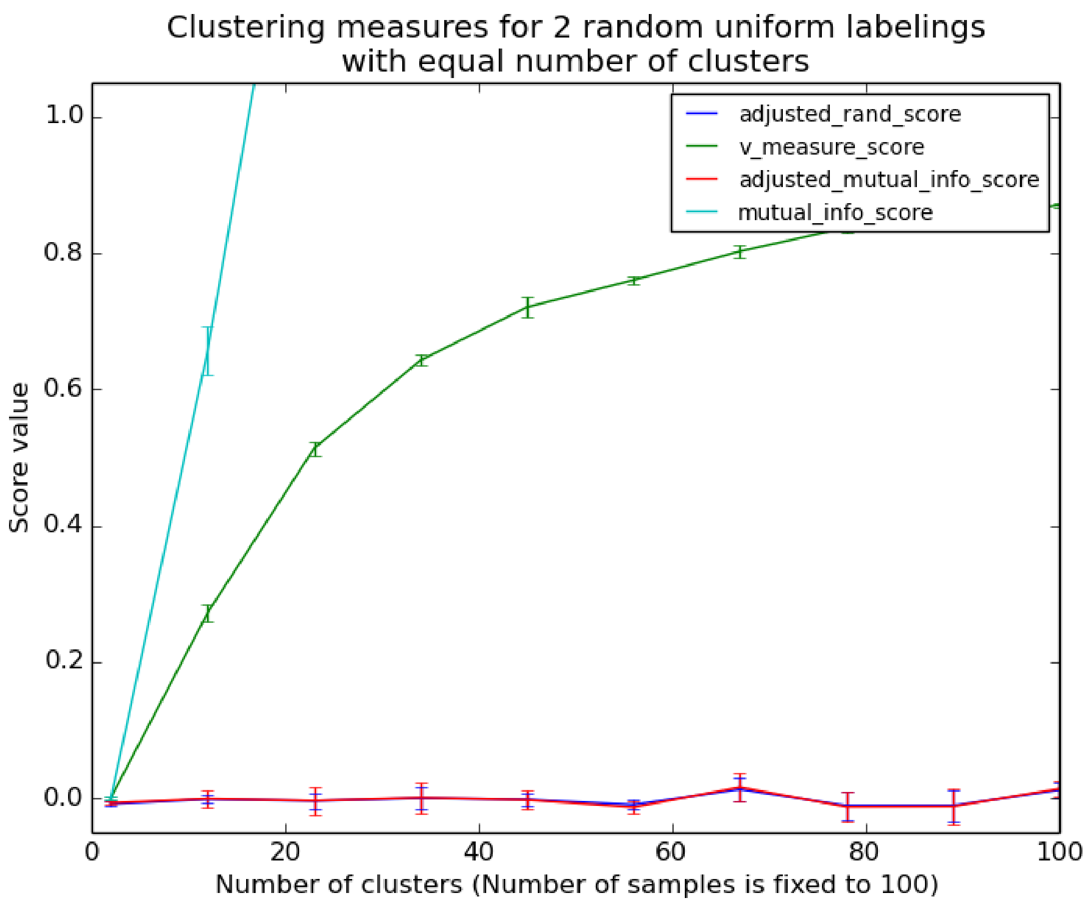
Advantages

- **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)**.



- 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

Silhouette Coefficient

Presentation and usage

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
>>> dataset = datasets.load_iris()
>>> X = dataset.data
>>> y = dataset.target
```

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.

5.4 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:

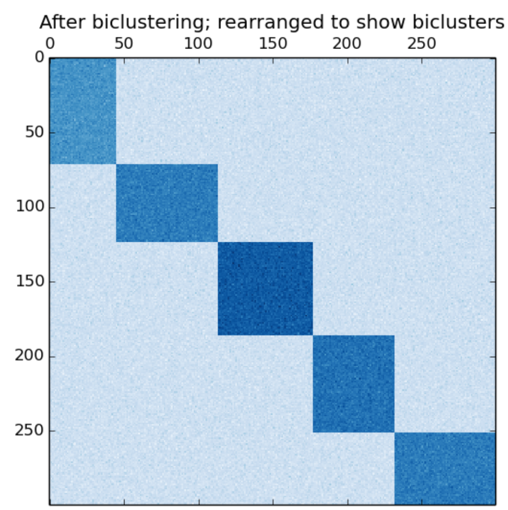


Figure 5.3: An example of biclusters formed by partitioning rows and columns.

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:

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.

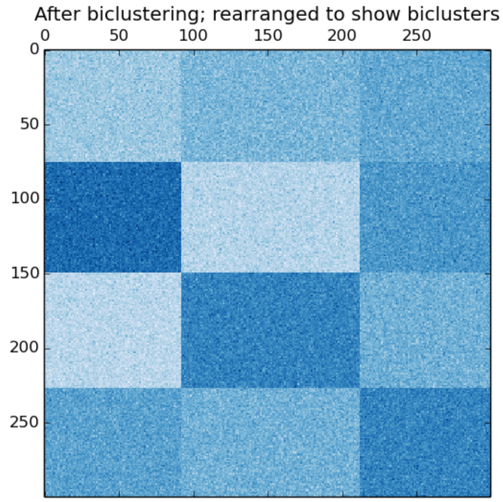


Figure 5.4: An example of checkerboard biclusters.

5.4.1 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 the columns of U are $u_2, \dots, u_{\ell+1}$, 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.

5.4.2 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 is 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.

5.4.3 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.

5.5 Decomposing signals in components (matrix factorization problems)

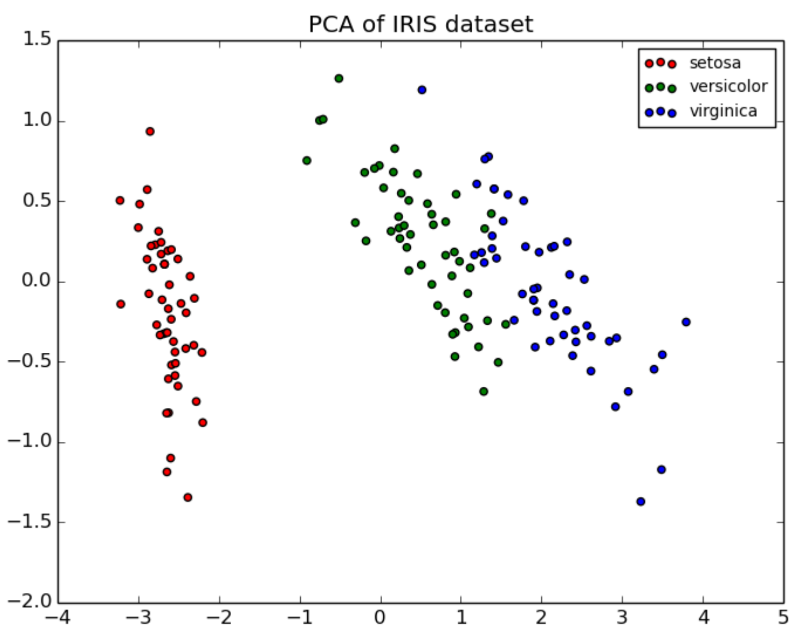
5.5.1 Principal component analysis (PCA)

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.

The optional parameter `whiten=True` parameter make 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.

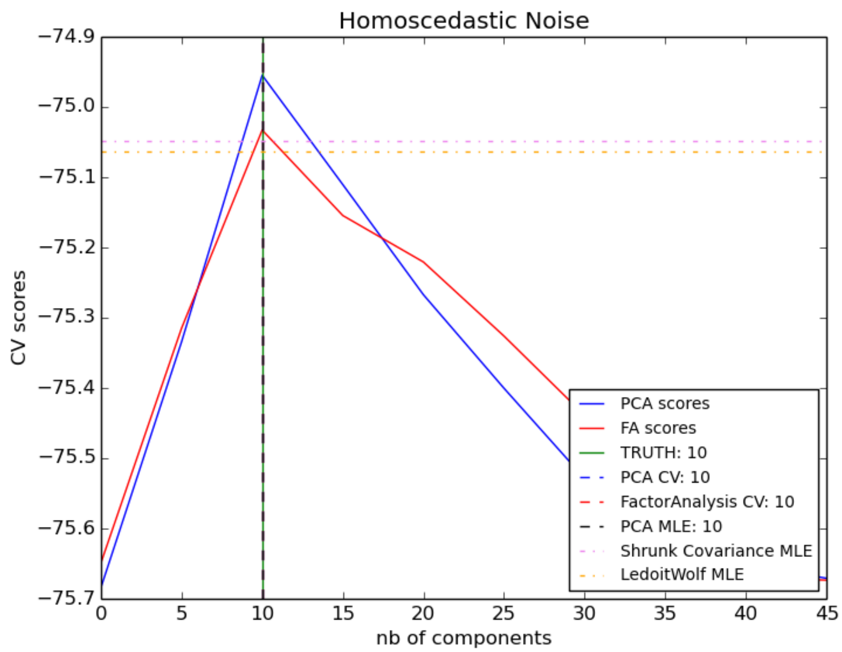
Below is an example of the iris dataset, which is comprised of 4 features, projected on the 2 dimensions that explain most variance:



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 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.

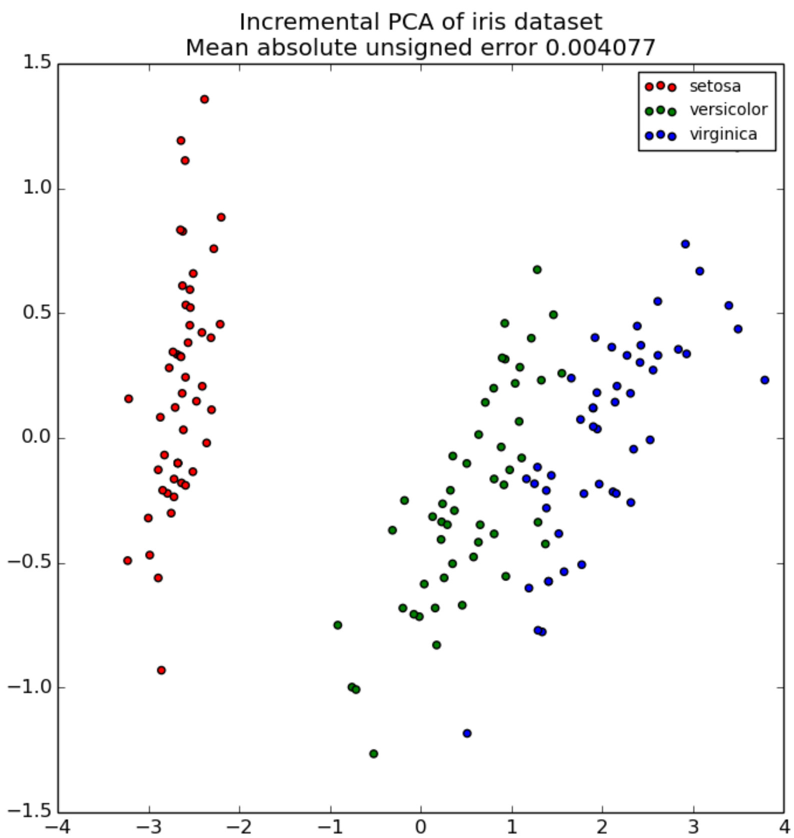
Examples:

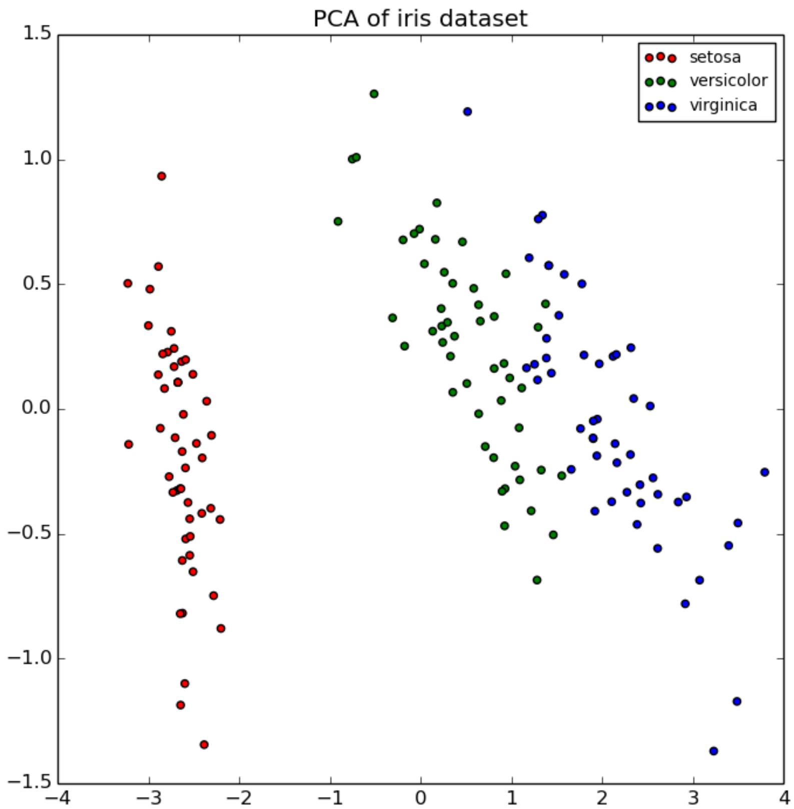
- *Incremental PCA*

Approximate PCA

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

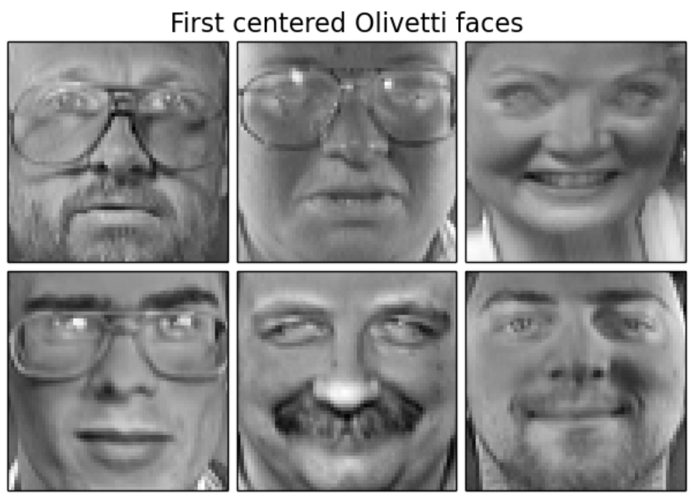

Incremental PCA of iris dataset
Mean absolute unsigned error 0.004077
setosa
versicolor
virginica
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−4
−3
−2
−1
0
1
2
3
4

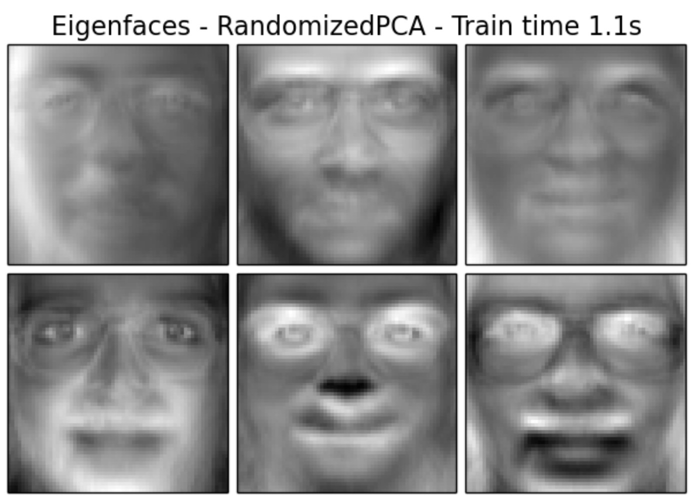

PCA of iris dataset
setosa
versicolor
virginica
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−4
−3
−2
−1
0
1
2
3
4

linearly transform the data while both reducing the dimensionality and preserve most of the explained variance at the same time.

The class RandomizedPCA 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.

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 it less than 1s:





RandomizedPCA can hence be used as a drop in replacement for PCA with the exception that we need to give it the size of the lower-dimensional space `n_components` as a mandatory input parameter.

If we note $n_{max} = max(n_{samples}, n_{features})$ and $n_{min} = min(n_{samples}, n_{features})$, the time complexity of RandomizedPCA is $O(n_{max}^2 \cdot n_{components})$ instead of $O(n_{max}^2 \cdot n_{min})$ for the exact method implemented in PCA.

The memory footprint of RandomizedPCA is also proportional to $2 \cdot n_{max} \cdot n_{components}$ instead of $n_{max} \cdot n_{min}$ for the exact method.

Note: the implementation of `inverse_transform` in RandomizedPCA is not the exact inverse transform of `transform` even when `whiten=False` (default).

Examples:

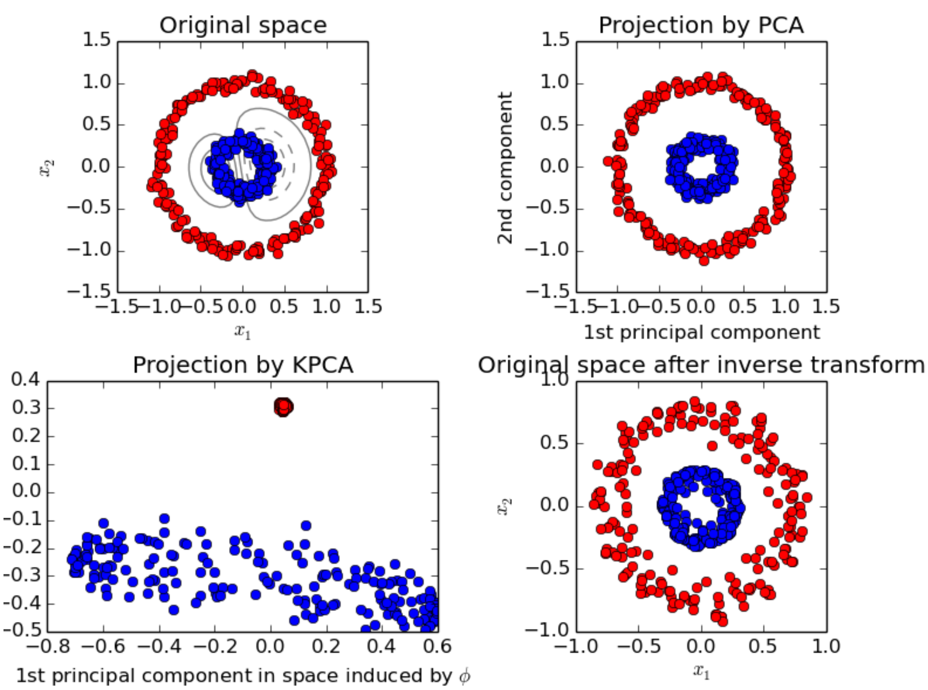
- *Faces recognition example using eigenfaces and SVMs*
- *Faces dataset decompositions*

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)

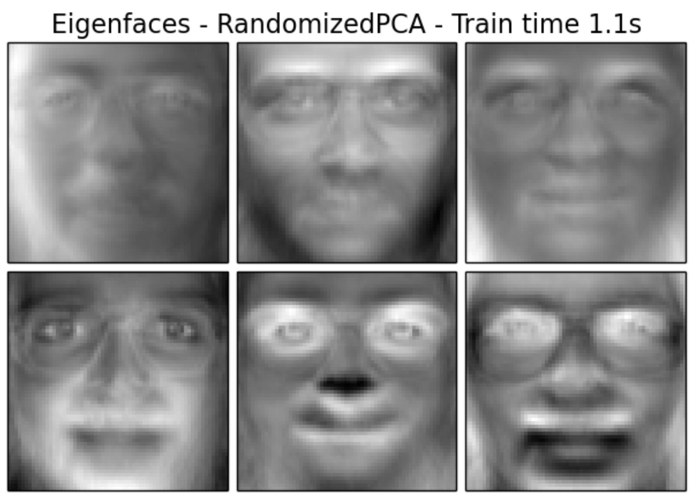
`SparsePCA` is a variant of PCA, with the goal of extracting the set of sparse components that best reconstruct the data.

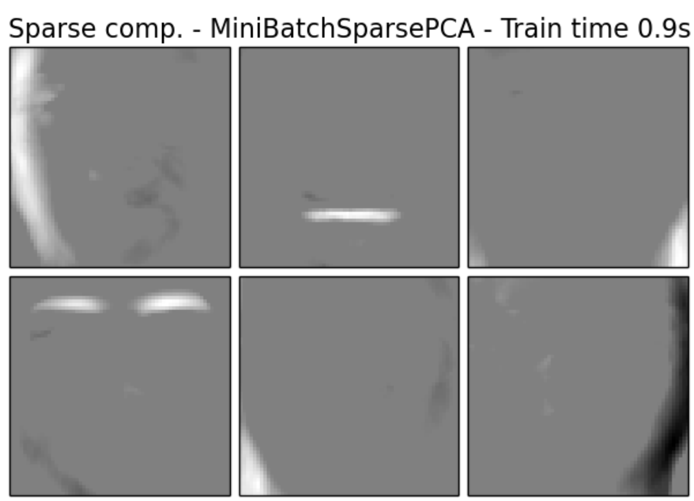
Mini-batch sparse PCA (`MiniBatchSparsePCA`) is a variant of `SparsePCA` that is faster but less accurate. The increased speed is reached by iterating over small chunks of the set of features, for a given number of iterations.

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:

$$(U^*, V^*) = \underset{U,V}{\arg\min}\, \frac{1}{2}||X - UV||_2^2 + \alpha||V||_1$$
$$\text{subject to } ||U_k||_2 = 1 \text{ for all } 0 \leq k < n_{components}$$

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:

5.5.2 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:

$$X' = X V_k^\top$$

Note: Most treatments of LSA in the natural language processing (NLP) and information retrieval (IR) literature swap the axis 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

5.5.3 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*

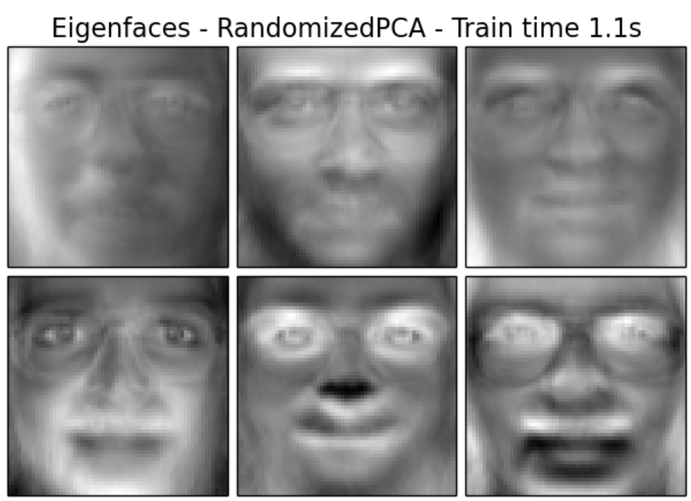
Generic dictionary learning

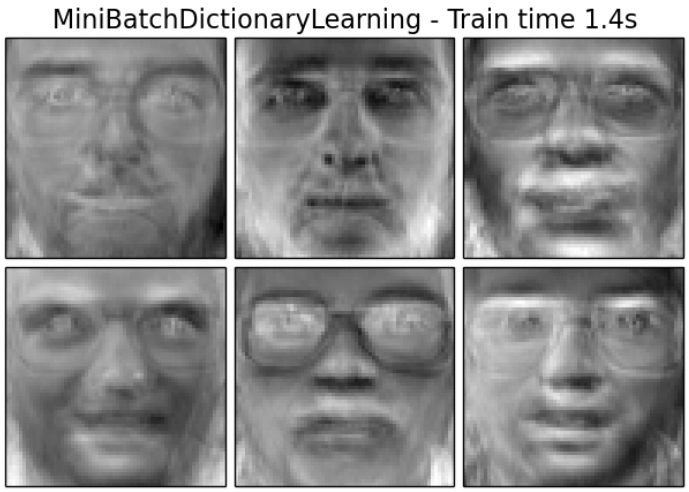
Dictionary learning (`DictionaryLearning`) is a matrix factorization problem that amounts to finding a (usually overcomplete) dictionary that will perform good at sparsely encoding the fitted data.

Representing data as sparse combinations of atoms from an overcomplete dictionary is suggested to be the way the mammal 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.

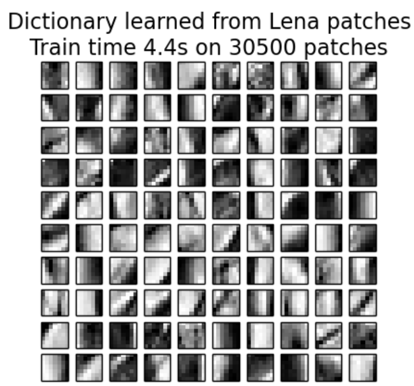
$$(U^*, V^*) = \underset{U, V}{\arg\min} \frac{1}{2}||X - UV||_2^2 + \alpha||U||_1$$
$$\text{subject to } ||V_k||_2 = 1 \text{ for all } 0 \leq k < n_{atoms}$$





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*).

The following image shows how a dictionary learned from 4x4 pixel image patches extracted from part of the image of Lena looks like.



Examples:

- *Image denoising using dictionary learning*

References:

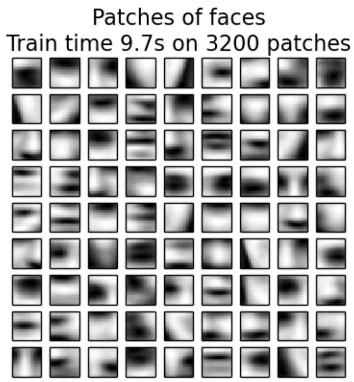
- "Online dictionary learning for sparse coding" J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009

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*

5.5.4 Factor Analysis

In unsupervised learning we only have a dataset $X = \{x_1, x_2, \dots, x_n\}$. How can this dataset be described mathematically? A very simple *continuous latent variabel* 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:

$$p(x_i|h_i) = \mathcal{N}(Wh_i + \mu, \Psi)$$

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:

$$p(x) = \mathcal{N}(\mu, WW^T + \Psi)$$

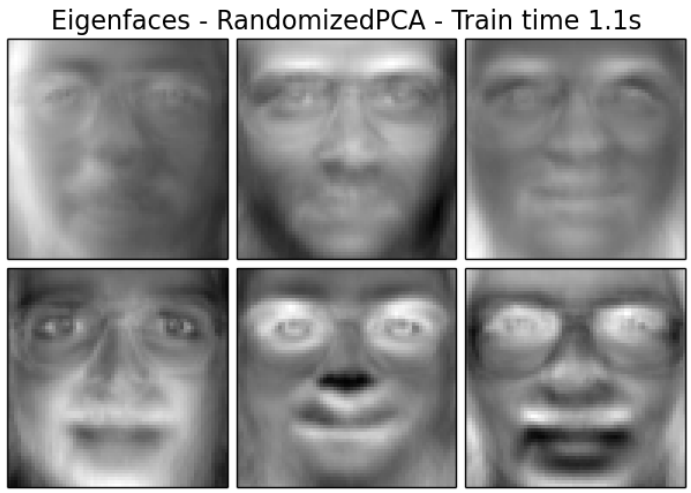
Now, without any further assumptions the idea of having a latent variable h would be superfluous – x can be completely modelled with a mean and a covariance. We need to impose some more specific structure on one of these two parameters. A simple additional assumption regards the structure of the error covariance Ψ:

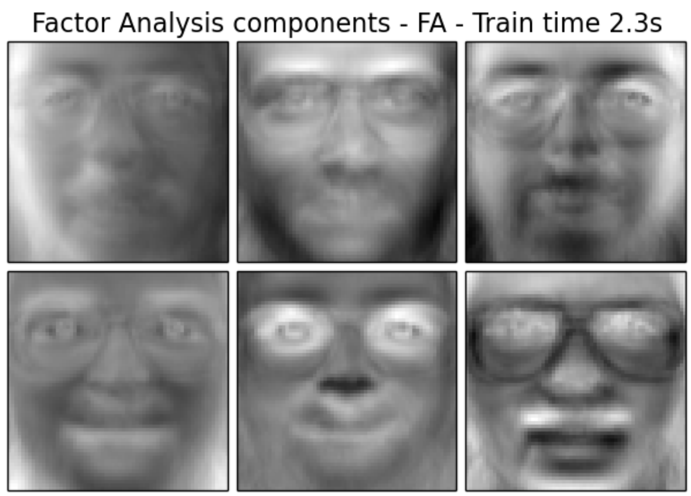
- $\Psi = \sigma^2 \mathbf{I}$: This assumption leads to the probabilistic model of `PCA`.

- $\Psi = diag(\psi_1, \psi_2, \dots, \psi_n)$: This model is called Factor Analysis, a classical statistical model. The matrix W is sometimes called the "factor loading matrix".

Both model 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):





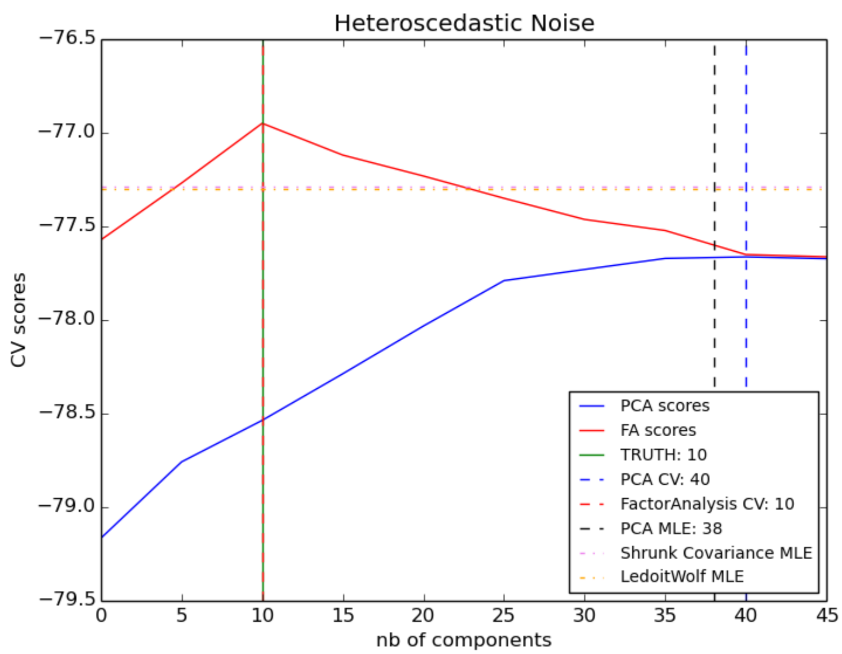
The main advantage for Factor Analysis (over `PCA` is that it can model the variance in every direction of the input space independently (heteroscedastic noise):

This allows better model selection than probabilistic PCA in the presence of heteroscedastic noise:

Examples:

- *Model selection with Probabilistic (PCA) and Factor Analysis (FA)*

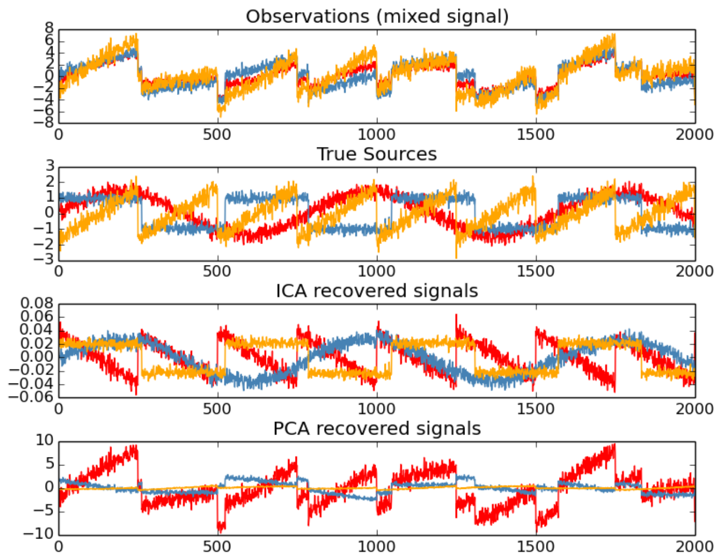

Pixelwise variance


Heteroscedastic Noise
CV scores
nb of components
−76.5
−77.0
−77.5
−78.0
−78.5
−79.0
−79.5
0
5
10
15
20
25
30
35
40
45
PCA scores
FA scores
TRUTH: 10
PCA CV: 40
FactorAnalysis CV: 10
PCA MLE: 38
Shrunk Covariance MLE
LedoitWolf MLE

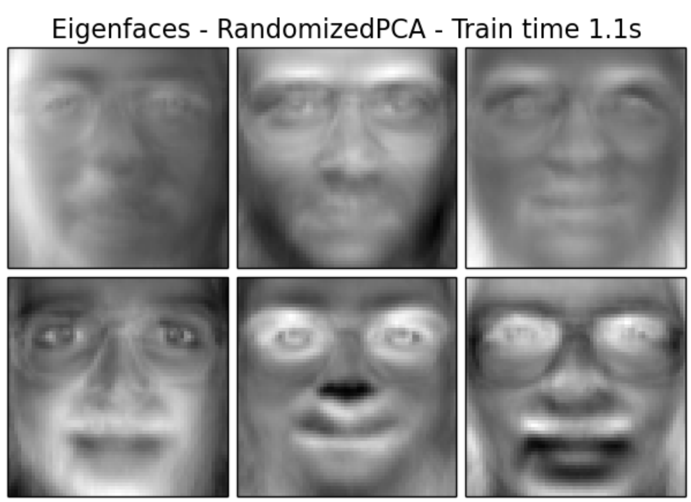
5.5.5 Independent component analysis (ICA)

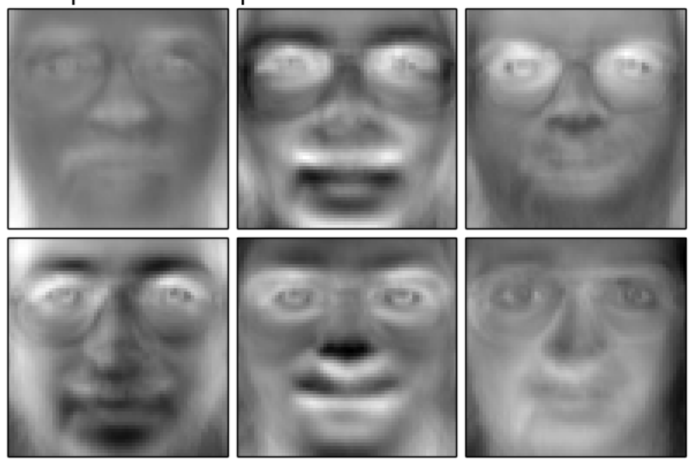
Independent component analysis separates a multivariate signal into additive subcomponents that are maximally independent. It is implemented in scikit-learn using the `Fast ICA` algorithm. Typically, ICA is not used for reducing dimensionality but for separating superimposed signals. Since the ICA model 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*

5.5.6 Non-negative matrix factorization (NMF or NNMF)

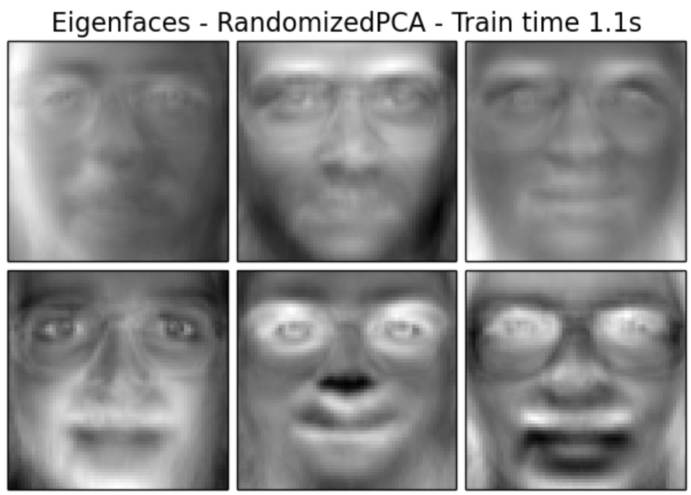
`NMF` is an alternative approach to decomposition that assumes that the data and the components are non-negative. `NMF` can be plugged in instead of `PCA` or its variants, in the cases where the data matrix does not contain negative values. It finds a decomposition of samples X into two matrices V and H of non-negative elements, by optimizing for the squared Frobenius norm:

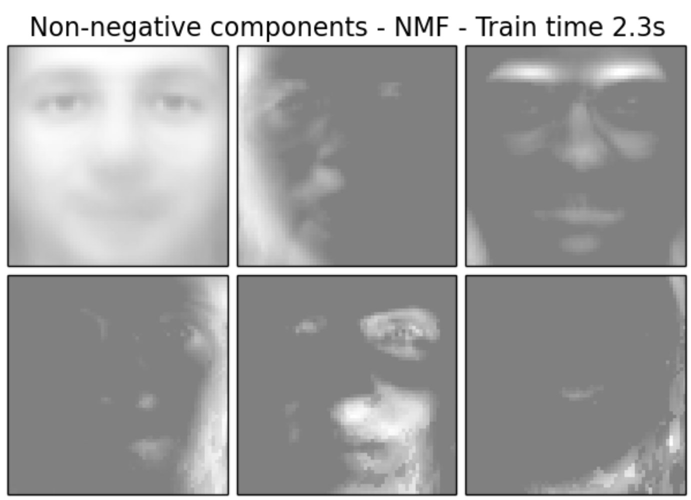
$$\arg\min_{W,H} ||X - WH||^2 = \sum_{i,j} X_{ij} - WH_{ij}$$

This norm is an obvious extension of the Euclidean norm to matrices. (Other optimization objectives have been suggested in the NMF literature, in particular Kullback-Leibler divergence, but these are not currently implemented.)

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, 04] 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.





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 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.

`NMF` can also be initialized with random non-negative matrices, by passing an integer seed or a `RandomState` to `init`.

In `NMF`, sparseness can be enforced by setting the attribute `sparseness` to `"data"` or `"components"`. Sparse components lead to localized features, and sparse data leads to a more efficient representation of the data.

Examples:

- *Faces dataset decompositions*
- *Topics extraction with Non-Negative Matrix Factorization*

References:

- "Learning the parts of objects by non-negative matrix factorization" D. Lee, S. Seung, 1999
- "Non-negative Matrix Factorization with Sparseness Constraints" P. Hoyer, 2004
- "Projected gradient methods for non-negative matrix factorization" C.-J. Lin, 2007
- "SVD based initialization: A head start for nonnegative matrix factorization" C. Boutsidis, E. Gallopoulos, 2008

5.6 Covariance estimation

Many statistical problems require at some point 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) has a large influence on the estimation's quality. The *sklearn.covariance* package aims at providing tools affording an accurate estimation of a population's covariance matrix under various settings.

We assume that the observations are independent and identically distributed (i.i.d.).

5.6.1 Empirical covariance

The covariance matrix of a data set is known to be well approximated with 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 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 depending whether the data are centered or not, the result will be different, so one may want to use the `assume_centered` parameter accurately. More precisely if one uses `assume_centered=False`, then the test set is supposed to have the same mean vector as the training set. If not so, 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.

5.6.2 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 the 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, depending whether the data are centered or not, the result will be different, 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 eigenvalue 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 : $\Sigma_{\rm shrunk} = (1-\alpha)\hat{\Sigma} + \alpha\frac{{\rm Tr}\hat{\Sigma}}{p}{\rm Id}$.

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 so as 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.

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.

[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.

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.

[2] Chen et al., "Shrinkage Algorithms for MMSE Covariance Estimation", IEEE Trans. on Sign. Proc., Volume 58, Issue 10, October 2010.

Examples:

- See *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood* for an example on how to fit an `OAS` object to data.
- See *Ledoit-Wolf vs OAS estimation* to visualize the Mean Squared Error difference between a `LedoitWolf` and an `OAS` estimator of the covariance.

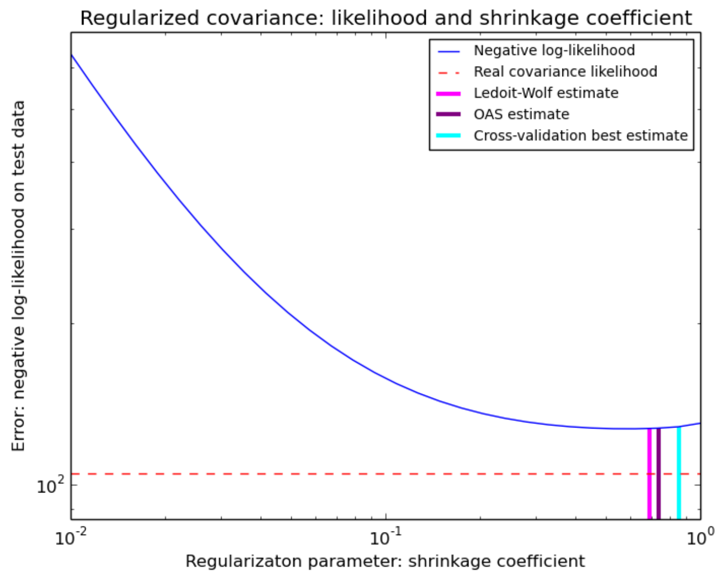
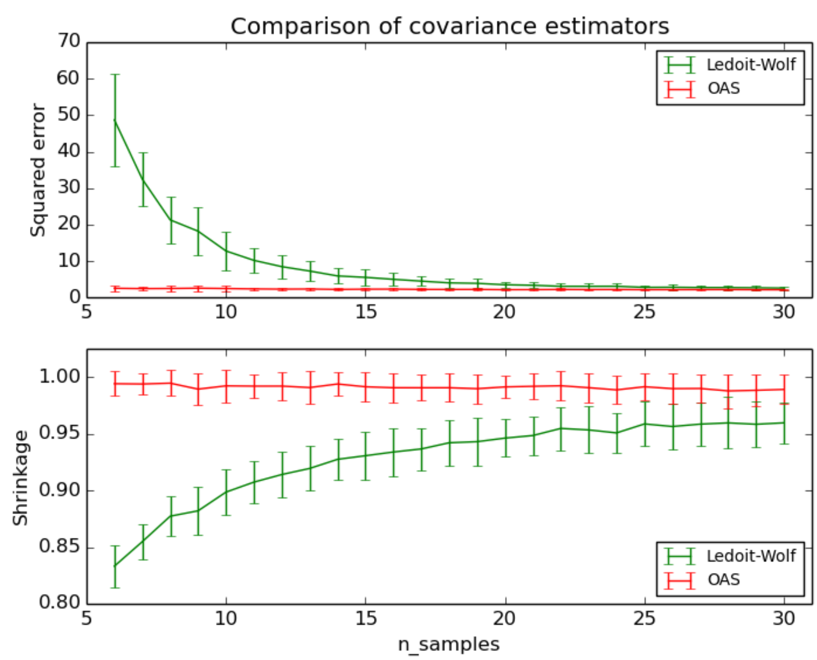


Figure 5.5: Bias-variance trade-off when setting the shrinkage: comparing the choices of Ledoit-Wolf and OAS estimators



5.6.3 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: by learning independence relations from the data, the estimation of the covariance matrix is better conditioned. 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 `GraphLasso` 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 `GraphLassoCV` object uses cross-validation to automatically set the `alpha` parameter.

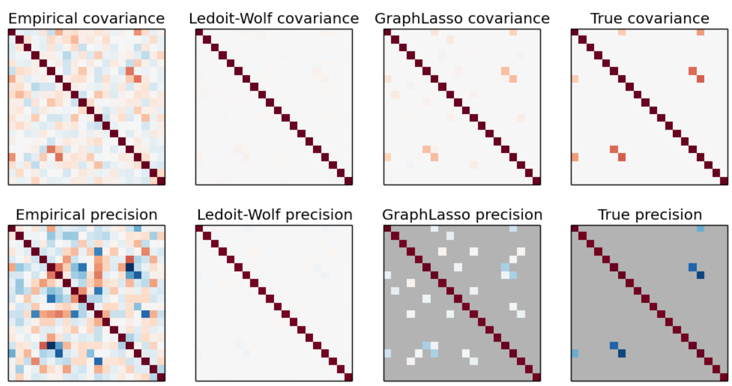


Figure 5.6: *A comparison of maximum likelihood, shrinkage and sparse estimates of the covariance and precision matrix in the very small samples settings.*

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 `GraphLasso`
- If the underlying graph has nodes with much more connections than the average node, the algorithm will miss some of these connections.
- 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 `GraphLassoCV` 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:

$$\hat{K} = \mathrm{argmin}_K \big(\mathrm{tr} S K - \mathrm{log} \mathrm{det} K + \alpha \|K\|_1 \big)$$

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

5.6.4 Robust Covariance Estimation

Real data set are often subjects to measurement or recording errors. Regular but uncommon observations may also appear for a variety of reason. Every observation which is very uncommon is called an outlier. The empirical covariance estimator and the shrunk covariance estimators presented above are very sensitive to the presence of outlying observations 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 the 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.

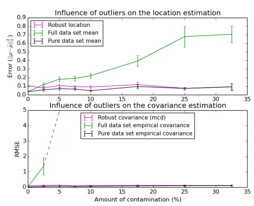
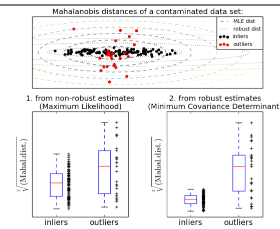
Raw estimates can be accessed as `raw_location_` and `raw_covariance_` attributes of a `MinCovDet` robust covariance estimator object.

[3] P. J. Rousseeuw. Least median of squares regression.

10. 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.

> **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).

Influence of outliers on location and covariance estimates	Separating inliers from outliers using a Mahalanobis distance
	

5.7 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 distinction must be made:

novelty detection The training data is not polluted by outliers, and we are interested in detecting anomalies in new observations.

outlier detection The training data contains outliers, and we need to fit the central mode of the training data, ignoring the deviant observations.

The scikit-learn project provides a set of machine learning tools that can be used both for novelty or outliers 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.

5.7.1 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, 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.

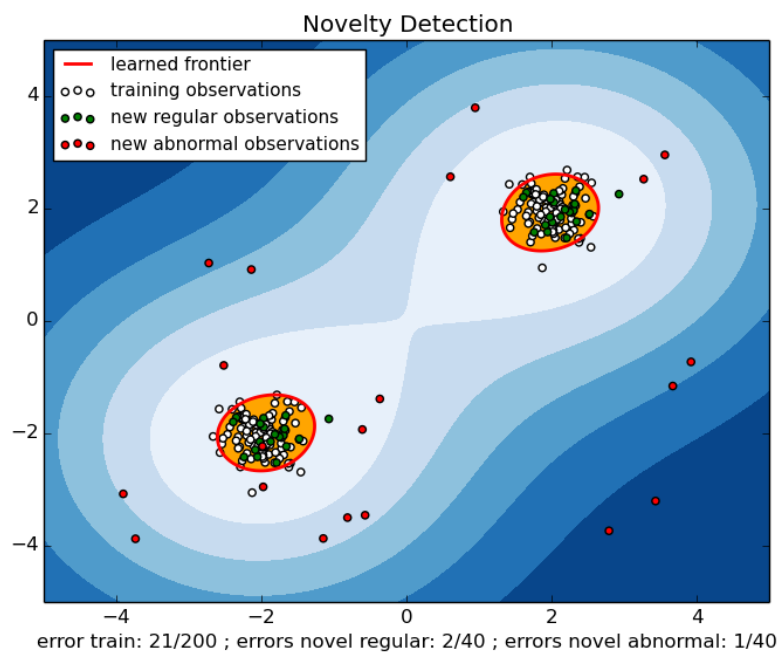
The One-Class SVM has been introduced by Schölkopf et al. for that purpose and implemented in the *Support Vector Machines* module in the `svm.OneClassSVM` object. It requires the choice of a kernel and a scalar parameter to define a frontier. The RBF kernel is usually chosen although there exists no 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.



5.7.2 Outlier Detection

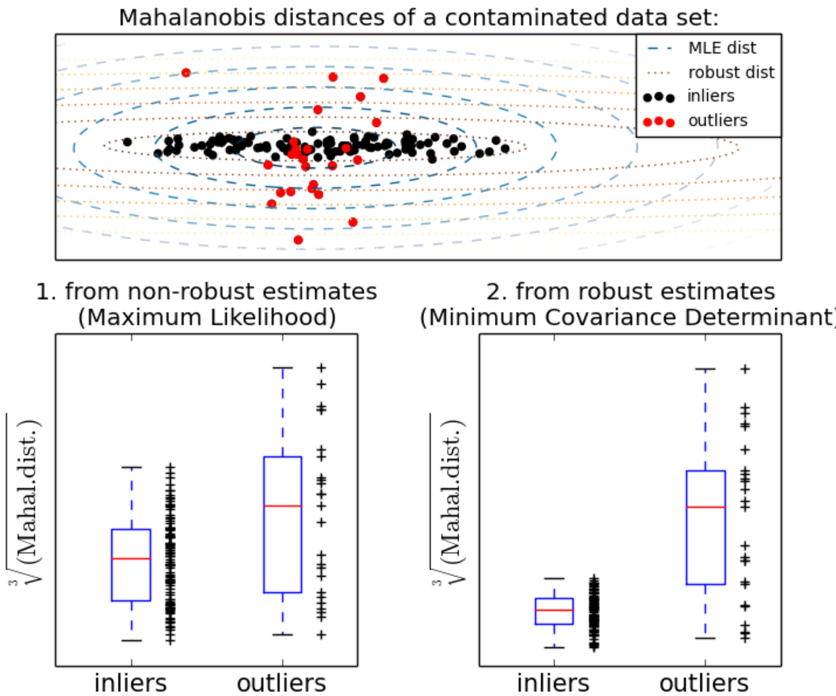
Outlier detection is similar to novelty detection in the sense that the goal is to separate a core of regular observations from some polluting ones, called "outliers". Yet, in the case of outlier detection, we don't have a clean data set representing the population of regular observations that can be 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. whithout 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.

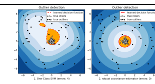
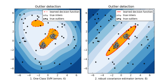
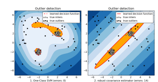
References:

One-class SVM versus elliptic envelope

Strictly-speaking, the One-class SVM is not an outlier-detection method, but a novelty-detection method: its training set should not be contaminated by outliers as it may fit them. 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 gives useful results in these situations.

The examples below illustrate how the performance of the `covariance.EllipticEnvelope` degrades as the data is less and less unimodal. `svm.OneClassSVM` works better on data with multiple modes.

Table 5.1: Comparing One-class SVM approach, and elliptic envelope

For a inlier mode well-centered and elliptic, the `svm.OneClassSVM` is not able to benefit from the rotational symmetry of the inlier population. In addition, it fits a bit the outliers present in the training set. On the opposite, the decision rule based on fitting an `covariance.EllipticEnvelope` learns an ellipse, which fits well the inlier distribution.	
As the inlier distribution becomes bimodal, the `covariance.EllipticEnvelope` does not fit well the inliers. However, we can see that the `svm.OneClassSVM` tends to overfit: because it has not model of inliers, it interprets a region where, by chance some outliers are clustered, as inliers.	
If the inlier distribution is strongly non Gaussian, the `svm.OneClassSVM` is able to recover a reasonable approximation, whereas the `covariance.EllipticEnvelope` completely fails.	

Examples:

- See *Outlier detection with several methods.* for a comparison of the `svm.OneClassSVM` (tuned to perform like an outlier detection method) and a covariance-based outlier detection with `covariance.MinCovDet`.

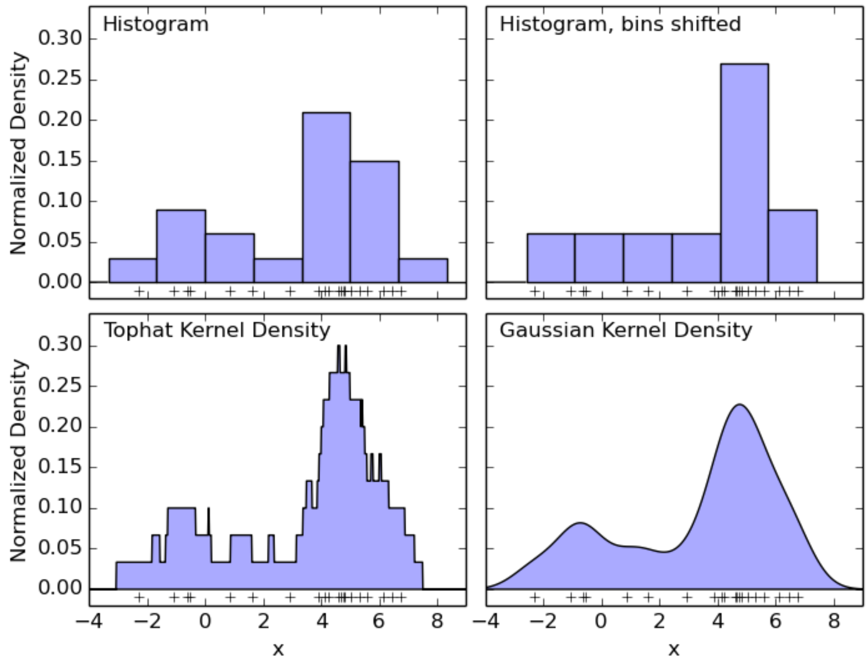
5.8 Density Estimation

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.GMM`), 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.

5.8.1 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.

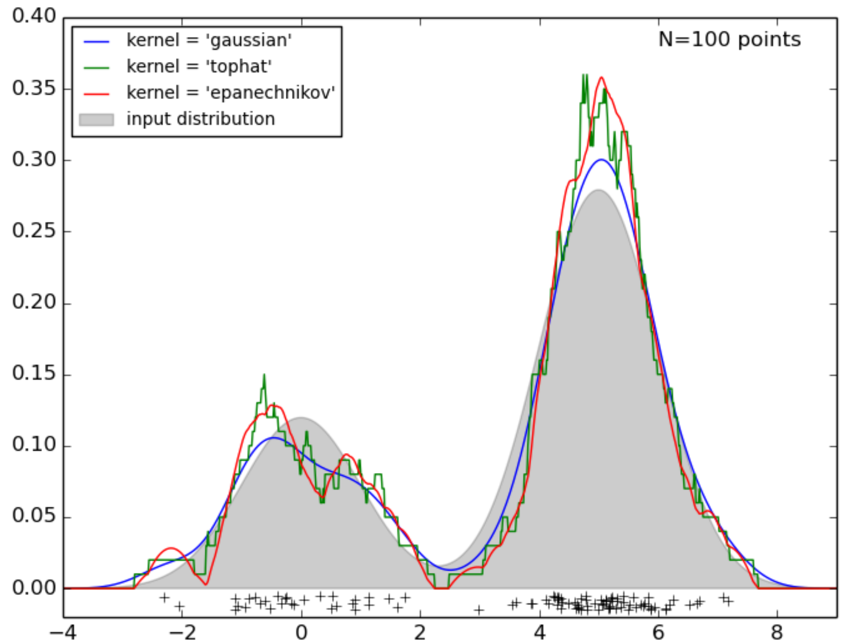
Intuitively, one can also think of a histogram as a stack of blocks, one block per point. By stacking the blocks in the appropriate grid space, we recover the histogram. But what if, instead of stacking the blocks on a regular grid, we center each block on the point it represents, and sum the total height 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.

5.8.2 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:

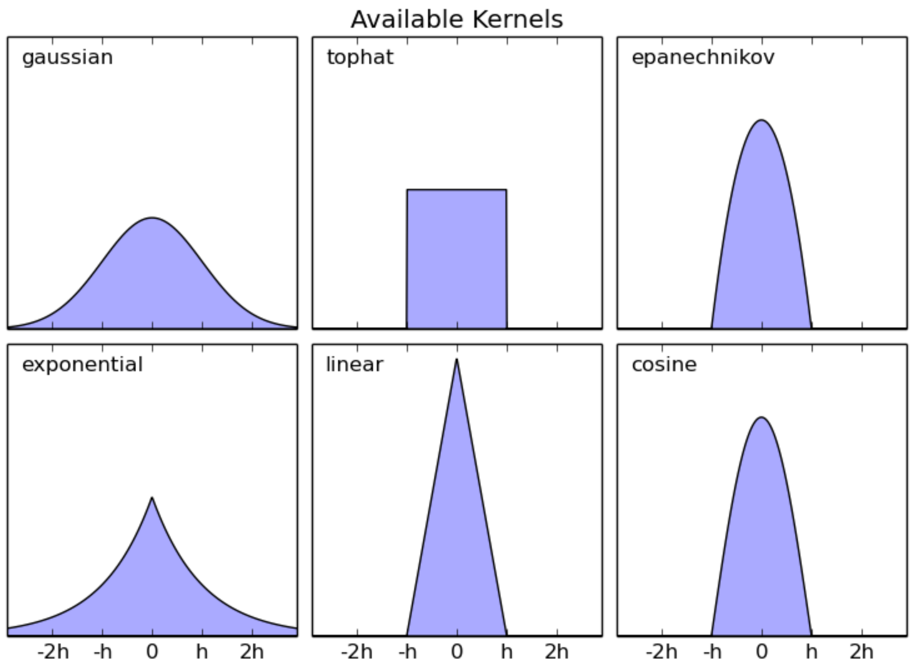
```
>>> from sklearn.neighbors.kde import KernelDensity
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> kde = KernelDensity(kernel='gaussian', bandwidth=0.2).fit(X)
>>> kde.score_samples(X)
array([-0.41075698, -0.41075698, -0.41076071, -0.41075698, -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'`)

  $K(x; h) \propto \exp(-x/h)$

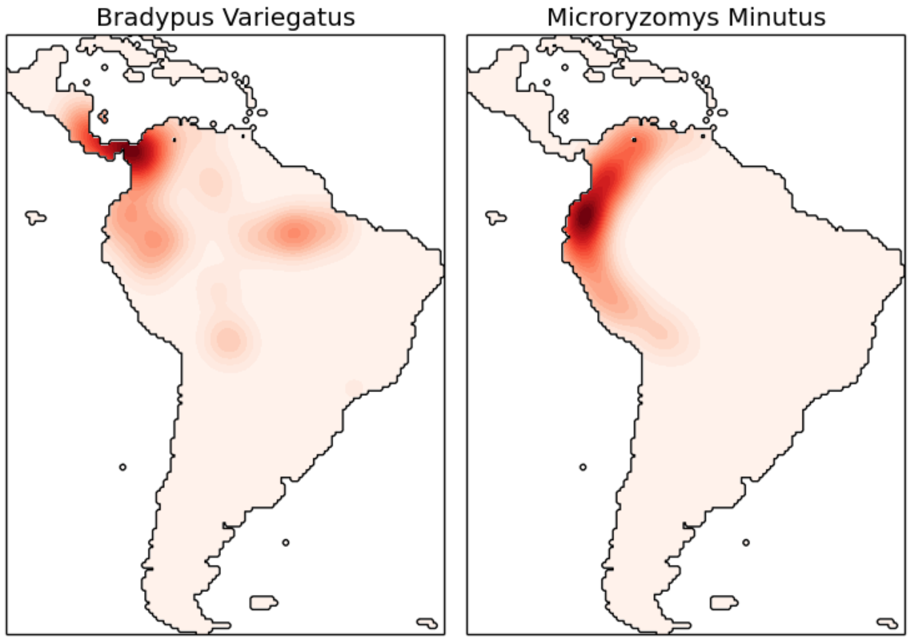
- Linear kernel (`kernel = 'linear'`)

  $K(x; h) \propto 1 - x/h$ if $x < h$

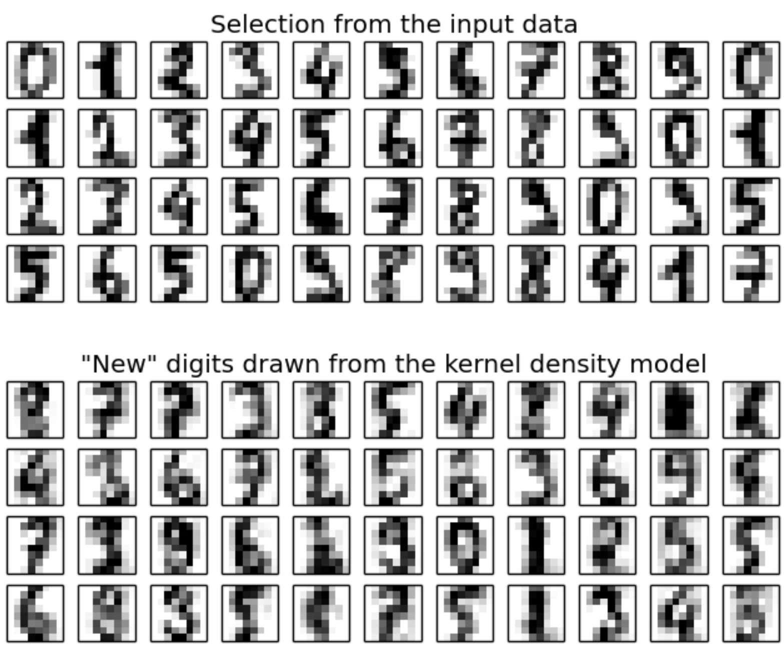
- Cosine kernel (`kernel = 'cosine'`)

  $K(x; h) \propto \cos(\frac{\pi x}{2h})$ 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

5.9 Neural network models (unsupervised)

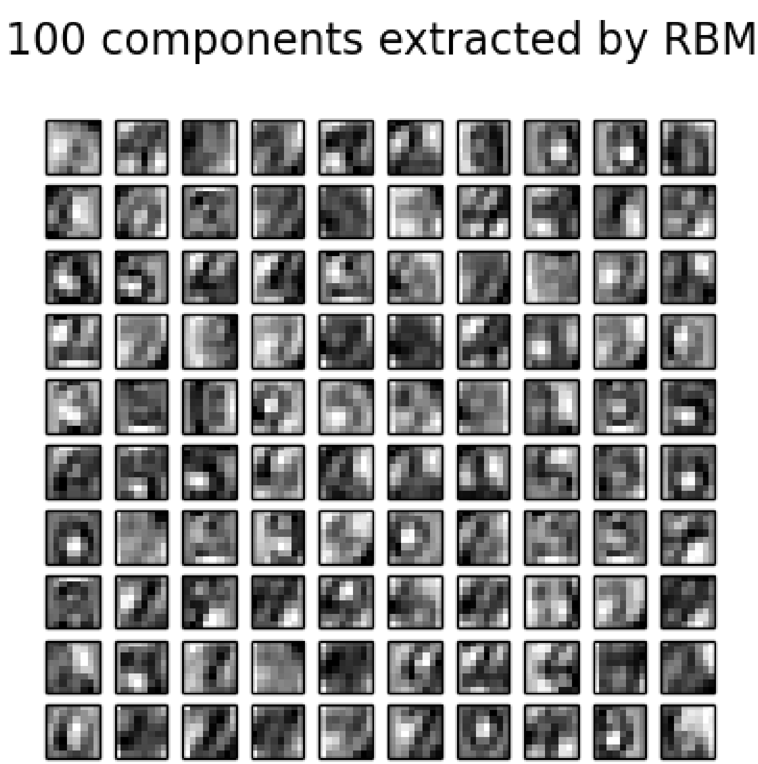
5.9.1 Restricted Boltzmann machines

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.

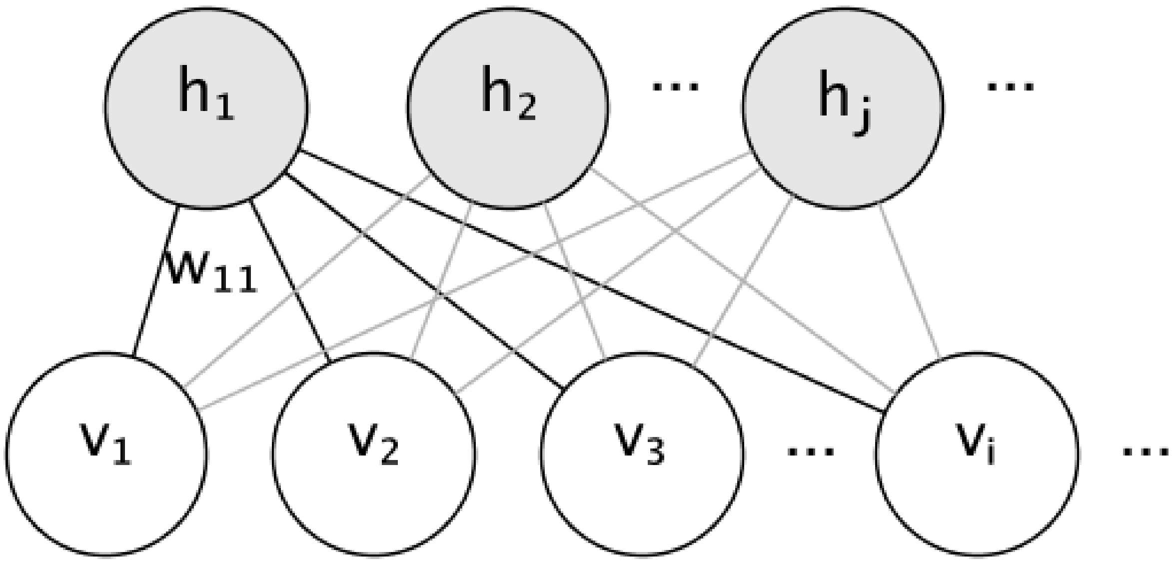


Examples:

- *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, ommited 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:

$$h_i \bot h_j | \mathbf{v}$$
$$v_i \bot v_j | \mathbf{h}$$

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 refered 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

CHAPTER

SIX

MODEL SELECTION AND EVALUATION

6.1 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.

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 import cross_validation
>>> from sklearn import datasets
>>> from sklearn import svm

>>> iris = datasets.load_iris()
>>> iris.data.shape, iris.target.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 = cross_validation.train_test_split(
...     iris.data, iris.target, 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 it is the case when fixing an arbitrary test set), which is a major advantage in problem such as inverse inference where the number of samples is very small.

6.1.1 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):

```
>>> clf = svm.SVC(kernel='linear', C=1)
>>> scores = cross_validation.cross_val_score(
...    clf, iris.data, iris.target, 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_validation.cross_val_score(clf, iris.data, iris.target,
...     cv=5, scoring='f1_weighted')
>>> 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:

```
>>> n_samples = iris.data.shape[0]
>>> cv = cross_validation.ShuffleSplit(n_samples, n_iter=3,
...     test_size=0.3, random_state=0)

>>> cross_validation.cross_val_score(clf, iris.data, iris.target, cv=cv)
```

```
...
array([ 0.97...,  0.97...,  1.        ])
```

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 = cross_validation.train_test_split(
...     iris.data, iris.target, 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_validation.cross_val_score(clf, iris.data, iris.target, cv=cv)
...
array([ 0.97...,  0.93...,  0.95...])
```

See *Pipeline and FeatureUnion: combining estimators*.

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).

These prediction can then be used to evaluate the classifier:

```
>>> predicted = cross_validation.cross_val_predict(clf, iris.data,
...                                                iris.target, cv=10)
>>> metrics.accuracy_score(iris.target, predicted)
0.966...
```

Note that the result of this computation may be slightly different from those obtained using `cross_val_score` as the elements are grouped in different ways.

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*,

6.1.2 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.

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.cross_validation import KFold

>>> kf = KFold(4, n_folds=2)
>>> for train, test in kf:
...     print("%s %s" % (train, test))
[2 3] [0 1]
[0 1] [2 3]
```

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]
```

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.

Example of stratified 2-fold cross-validation on a dataset with 10 samples from two slightly unbalanced classes:

```
>>> from sklearn.cross_validation import StratifiedKFold

>>> labels = [0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
>>> skf = StratifiedKFold(labels, 3)
>>> for train, test in skf:
...     print("%s %s" % (train, test))
[2 3 6 7 8 9] [0 1 4 5]
[0 1 3 4 5 8 9] [2 6 7]
[0 1 2 4 5 6 7] [3 8 9]
```

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.cross_validation import LeaveOneOut

>>> loo = LeaveOneOut(4)
```

```
>>> for train, test in loo:
...     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.cross_validation import LeavePOut

>>> lpo = LeavePOut(4, p=2)
>>> for train, test in lpo:
...     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]
```

Leave-One-Label-Out - LOLO

`LeaveOneLabelOut` (LOLO) is a cross-validation scheme which holds out the samples according to a third-party provided array of integer labels. This label 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 label.

For example, in the cases of multiple experiments, *LOLO* 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.cross_validation import LeaveOneLabelOut

>>> labels = [1, 1, 2, 2]
>>> lolo = LeaveOneLabelOut(labels)
>>> for train, test in lolo:
...     print("%s %s" % (train, test))
[2 3] [0 1]
[0 1] [2 3]
```

Another common application is to use time information: for instance the labels could be the year of collection of the samples and thus allow for cross-validation against time-based splits.

> **Warning:** Contrary to `StratifiedKFold`, **the ‘‘labels‘‘ of :class:‘LeaveOneLabelOut‘ should not encode the target class to predict**: the goal of `StratifiedKFold` is to rebalance dataset classes across the train / test split to ensure that the train and test folds have approximately the same percentage of samples of each class while `LeaveOneLabelOut` will do the opposite by ensuring that the samples of the train and test fold will not share the same label value.

Leave-P-Label-Out

`LeavePLabelOut` is similar as *Leave-One-Label-Out*, but removes samples related to P labels for each training/test set.

Example of Leave-2-Label Out:

```
>>> from sklearn.cross_validation import LeavePLabelOut

>>> labels = [1, 1, 2, 2, 3, 3]
>>> lplo = LeavePLabelOut(labels, p=2)
>>> for train, test in lplo:
...     print("%s %s" % (train, test))
[4 5] [0 1 2 3]
[2 3] [0 1 4 5]
[0 1] [2 3 4 5]
```

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:

```
>>> ss = cross_validation.ShuffleSplit(5, n_iter=3, test_size=0.25,
...     random_state=0)
>>> for train_index, test_index in ss:
...     print("%s %s" % (train_index, test_index))
...
[1 3 4] [2 0]
[1 4 3] [0 2]
[4 0 2] [1 3]
```

`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 in on each side of the train / test split.

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.

See also

`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.

6.1.3 A note on shuffling

If the data ordering is not arbitrary (e.g. samples with the same 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 ensure results are repeatable (*on the same platform*), use a fixed value for `random_state`.

6.1.4 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 if the next section: *Grid Search: Searching for estimator parameters*.

6.2 Grid Search: Searching for estimator parameters

Parameters that are not directly learnt within estimators can be set by searching a parameter space for the best *Cross-validation: evaluating estimator performance* score. Typical examples include `C`, `kernel` and `gamma` for Support Vector Classifier, `alpha` for Lasso, etc.

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()
```

Such parameters are often referred to as *hyperparameters* (particularly in Bayesian learning), distinguishing them from the parameters optimised in a machine learning procedure.

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.

6.2.1 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.

6.2.2 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':['auto', 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` 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`.

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.

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)

6.2.3 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.

Composite estimators and parameter spaces

Pipeline: chaining estimators describes building composite estimators whose parameter space can be searched with these tools.

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 `cross_validation.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.

6.2.4 Alternatives to brute force parameter search

Model specific cross-validation

Some models can fit data for a range of value 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 L1/L2 Lasso with built-in cross-validation.
`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=1*, *positive=False*, *random_state=None*, *selection='cyclic'*)

Elastic Net model with iterative fitting along a regularization path

The best model is selected by cross-validation.

Parameters **l1_ratio** : float, 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

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 : integer or cross-validation generator, optional

If an integer is passed, it is the number of fold (default 3). Specific cross-validation objects can be passed, see the `sklearn.cross_validation` module for the list of possible objects.

verbose : bool or integer

Amount of verbosity.

n_jobs : integer, optional

Number of CPUs to use during the cross validation. If `-1`, use all the CPUs.

positive : bool, optional

When set to `True`, forces the coefficients to be positive.

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.

random_state : int, RandomState instance, or None (default)

The seed of the pseudo random number generator that selects a random feature to update. Useful only when selection is set to 'random'.

fit_intercept : boolean

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 : boolean, optional, default False

If `True`, the regressors X will be normalized before regression.

copy_X : boolean, optional, default True

If `True`, X will be copied; else, it may be overwritten.

Attributesalpha_ : 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

See examples/linear_model/lasso_path_with_crossvalidation.py for an example.

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).
```

Methods

`decision_function`(X)	Decision function of the linear model.
`fit`(X, y)	Fit linear model with coordinate descent
`get_params`([deep])	Get parameters for this estimator.
`path`	
`predict`(X)	Predict using the linear model
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

static __init__(*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=1*, *positive=False*, *random_state=None*, *selection='cyclic'*)

decision_function(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

fit(*X*, *y*)

Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

ParametersX : {array-like}, shape (n_samples, n_features)

Training data. Pass directly as float64, 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(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

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=1*, *eps=2.2204460492503131e-16*, *copy_X=True*)

Cross-validated Least Angle Regression model

Parametersfit_intercept : boolean

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).

verbose : boolean or integer, optional

Sets the verbosity amount

normalize : boolean, optional, default False

If True, the regressors X will be normalized before regression.

copy_X : boolean, optional, default True

If `True`, X will be copied; else, it may be overwritten.

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: integer, optional :

Maximum number of iterations to perform.

cv : cross-validation generator, optional

see `sklearn.cross_validation`. If `None` is passed, default to a 5-fold strategy

max_n_alphas : integer, optional

The maximum number of points on the path used to compute the residuals in the cross-validation

n_jobs : integer, optional

Number of CPUs to use during the cross validation. If `-1`, use all the CPUs

eps : float, optional

The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.

Attributescoef_ : array, shape (n_features,)

parameter vector (w in the formulation formula)

intercept_ : float

independent term in decision function

coef_path_ : array, shape (n_features, n_alphas)

the varying values of the coefficients along the path

alpha_ : float

the estimated regularization parameter alpha

alphas_ : array, shape (n_alphas,)

the different values of alpha along the path

cv_alphas_ : array, shape (n_cv_alphas,)

all the values of alpha along the path for the different folds

cv_mse_path_ : array, 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`

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y)	Fit the model using X, y as training data.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__ (*fit_intercept=True, verbose=False, max_iter=500, normalize=True, precompute='auto', cv=None, max_n_alphas=1000, n_jobs=1, eps=2.2204460492503131e-16, copy_X=True*)

decision_function (*X*)
Decision function of the linear model.

Parameters X : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returns C : array, shape = (n_samples,)

Returns predicted values.

static fit (*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.

Returns self : object

returns an instance of self.

get_params (*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.

predict (*X*)
Predict using the linear model

Parameters X : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returns C : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

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=1*, *positive=False*, *random_state=None*, *selection='cyclic'*)

Lasso linear model with iterative fitting along a regularization path

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
```

Parameterseps : 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

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 : integer or cross-validation generator, optional

If an integer is passed, it is the number of fold (default 3). Specific cross-validation objects can be passed, see the `sklearn.cross_validation` module for the list of possible objects.

verbose : bool or integer

Amount of verbosity.

n_jobs : integer, optional

Number of CPUs to use during the cross validation. If `-1`, use all the CPUs.

positive : bool, optional

If positive, restrict regression coefficients to be positive

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.

random_state : int, RandomState instance, or None (default)

The seed of the pseudo random number generator that selects a random feature to update. Useful only when selection is set to 'random'.

fit_intercept : boolean, 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 : boolean, optional, default False

If `True`, the regressors X will be normalized before regression.

copy_X : boolean, optional, default True

If `True`, X will be copied; else, it may be overwritten.

Attributesalpha_ : 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

See examples/linear_model/lasso_path_with_crossvalidation.py for an example.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Methods

`decision_function`(X)	Decision function of the linear model.
`fit`(X, y)	Fit linear model with coordinate descent
`get_params`([deep])	Get parameters for this estimator.
`path`	
`predict`(X)	Predict using the linear model
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*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=1, positive=False, random_state=None, selection='cyclic'*)

`decision_function`(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

`fit`(*X, y*)

Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

ParametersX : {array-like}, shape (n_samples, n_features)

Training data. Pass directly as float64, 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`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.LassoCV`

`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=1*, *eps=2.2204460492503131e-16*, *copy_X=True*)

Cross-validated Lasso, using the LARS algorithm

The optimization objective for Lasso is:

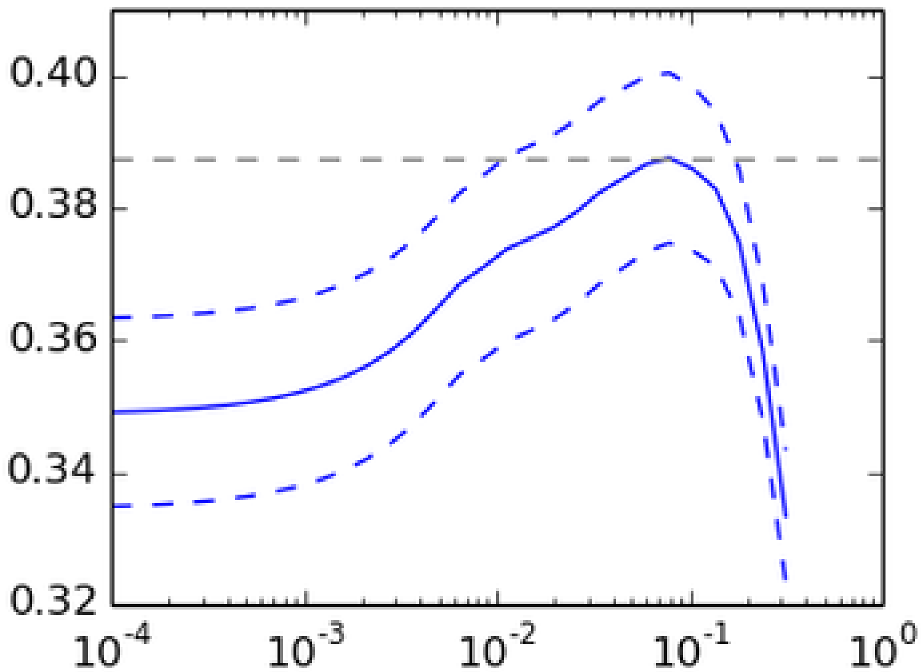


Figure 6.1: *Cross-validation on diabetes Dataset Exercise*

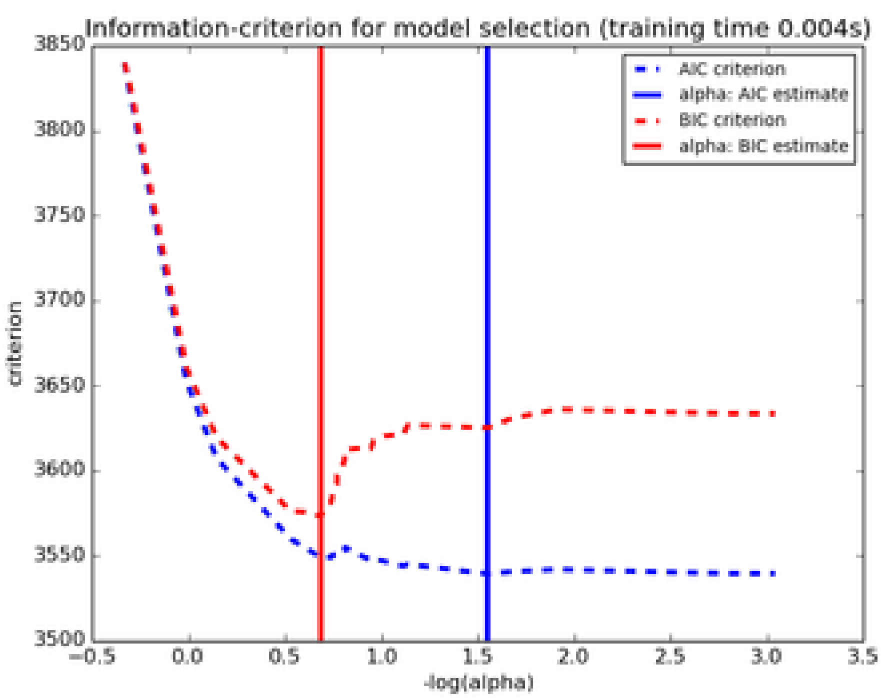


Figure 6.2: *Lasso model selection: Cross-Validation / AIC / BIC*

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Parametersfit_intercept : boolean

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).

verbose : boolean or integer, optional

Sets the verbosity amount

normalize : boolean, optional, default False

If True, the regressors X will be normalized before regression.

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 : integer, optional

Maximum number of iterations to perform.

cv : cross-validation generator, optional

see sklearn.cross_validation module. If None is passed, default to a 5-fold strategy

max_n_alphas : integer, optional

The maximum number of points on the path used to compute the residuals in the cross-validation

n_jobs : integer, optional

Number of CPUs to use during the cross validation. If `-1`, use all the CPUs

eps : float, optional

The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.

copy_X : boolean, optional, default True

If True, X will be copied; else, it may be overwritten.

Attributescoef_ : array, shape (n_features,)

parameter vector (w in the formulation formula)

intercept_ : float

independent term in decision function.

coef_path_ : array, shape (n_features, n_alphas)

the varying values of the coefficients along the path

alpha_ : float

the estimated regularization parameter alpha

alphas_ : array, shape (n_alphas,)

the different values of alpha along the path

cv_alphas_ : array, shape (n_cv_alphas,)

all the values of alpha along the path for the different folds

cv_mse_path_ : array, 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.

Methods

`decision_function`(X)	Decision function of the linear model.
`fit`(X, y)	Fit the model using X, y as training data.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict using the linear model
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

`__init__`(*fit_intercept=True*, *verbose=False*, *max_iter=500*, *normalize=True*, *precompute='auto'*, *cv=None*, *max_n_alphas=1000*, *n_jobs=1*, *eps=2.2204460492503131e-16*, *copy_X=True*)

`decision_function`(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

`fit`(*X*, *y*)

Fit the model using X, y as training data.

ParametersX : array-like, shape (n_samples, n_features)

Training data.

y : array-like, shape (n_samples,)

Target values.

Returnsself : object

returns an instance of self.

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)
Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.LassoLarsCV`

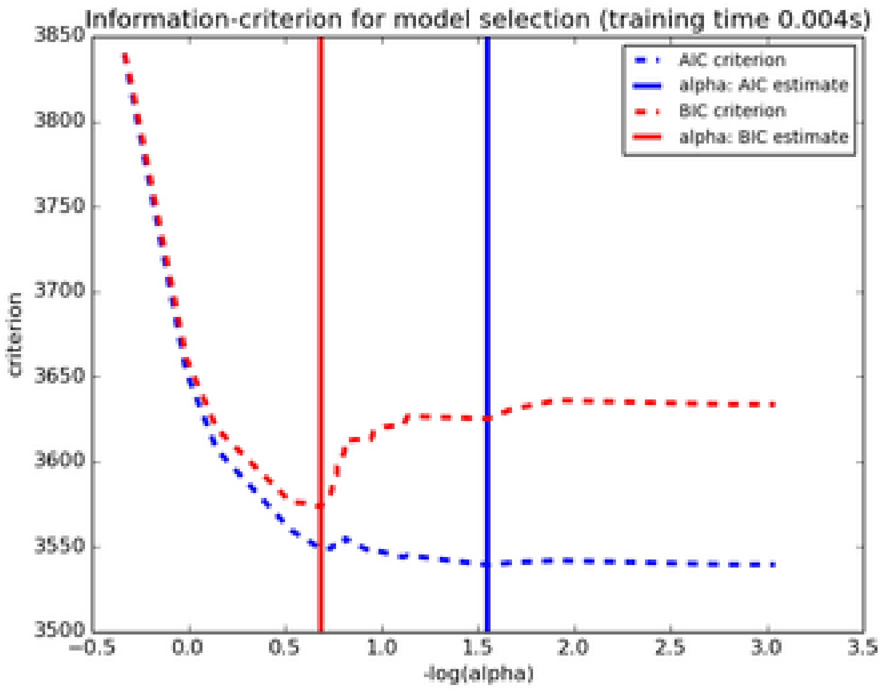


Figure 6.3: *Lasso model selection: Cross-Validation / AIC / BIC*

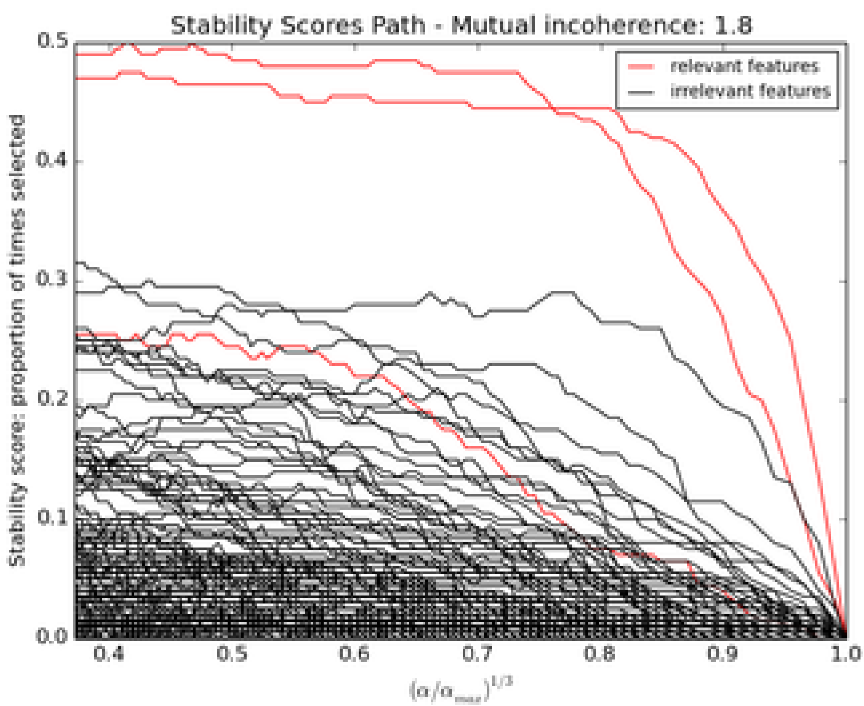


Figure 6.4: *Sparse recovery: feature selection for sparse linear models*

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=1*, *verbose=0*, *refit=True*, *intercept_scaling=1.0*, *multi_class='ovr'*)

Logistic Regression CV (aka logit, MaxEnt) classifier.

This class implements logistic regression using liblinear, newton-cg or LBFGS optimizer. The newton-cg 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.

For the grid of Cs values (that are set by default to be ten values in a logarithmic scale between 1e-4 and 1e4), the best hyperparameter is selected by the cross-validator StratifiedKFold, but it can be changed using the cv parameter. In the case of newton-cg and lbfgs solvers, we warm start along the path i.e guess the initial coefficients of the present fit to be the coefficients got after convergence in the previous fit, so in general it is supposed to be faster.

For a multiclass problem, the hyperparameters for each class are computed using the best scores got by doing a one-vs-rest in parallel across all folds and classes. Hence this is not the true multinomial loss.

ParametersCs : list of floats | int

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 the decision function.

class_weight : {dict, 'auto'}, optional

Over-/undersamples the samples of each class according to the given weights. If not given, all classes are supposed to have weight one. The 'auto' mode selects weights inversely proportional to class frequencies in the training set.

cv : integer or cross-validation generator

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.cross_validation module for the list of possible cross-validation objects.

penalty : str, 'l1' or 'l2'

Used to specify the norm used in the penalization. The newton-cg and lbfgs solvers support only l2 penalties.

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.

scoring : callabale

Scoring function to use as cross-validation criteria. For a list of scoring functions that can be used, look at sklearn.metrics. The default scoring option used is accuracy_score.

solver : {'newton-cg', 'lbfgs', 'liblinear'}

Algorithm to use in the optimization problem.

tol : float, optional

Tolerance for stopping criteria.

max_iter : int, optional

Maximum number of iterations of the optimization algorithm.

class_weight : {dict, 'auto'}, optional

Over-/undersamples the samples of each class according to the given weights. If not given, all classes are supposed to have weight one. The 'auto' mode selects weights inversely proportional to class frequencies in the training set.

n_jobs : int, optional

Number of CPU cores used during the cross-validation loop. If given a value of -1, all cores are used.

verbose : int

For the liblinear and lbfgs solvers set verbose to any positive number for verbosity.

refit : bool

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.

multi_class : str, {'ovr', 'multinomial'}

Multiclass option can be either 'ovr' or 'multinomial'. If the option chosen is 'ovr', then a binary problem is fit for each label. Else the loss minimised is the multinomial loss fit across the entire probability distribution. Works only for the 'lbfgs' solver.

intercept_scaling : float, default 1.

Useful only if solver is liblinear. This parameter is 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 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.

Attributescoef_ : array, 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. *coef_* is readonly property derived from *raw_coef_* that follows the internal memory layout of liblinear.

intercept_ : array, shape (1,) or (n_classes,)

Intercept (a.k.a. bias) added to the decision function. It is available only when parameter intercept is set to True and is of shape(1,) when the problem is binary.

Cs_ : array

Array of C i.e. inverse of regularization parameter values used for cross-validation.

coefs_paths_ : array, shape `(n_folds, len(Cs_), n_features)` or `(n_folds, len(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, len(Cs_), n_features)` or `(n_folds, len(Cs_), n_features + 1)` depending on whether the intercept is fit or not.

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, len(Cs))

C_ : array, 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.

See also:

`LogisticRegression`

Methods

`decision_function`(X)	Predict confidence scores for samples.
`densify`()	Convert coefficient matrix to dense array format.
`fit`(X, y)	Fit the model according to the given training data.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict class labels for samples in X.
`predict_log_proba`(X)	Log of probability estimates.
`predict_proba`(X)	Probability estimates.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.
`sparsify`()	Convert coefficient matrix to sparse format.
`transform`(X[, threshold])	Reduce X to its most important features.

static `__init__`(*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=1*, *verbose=0*, *refit=True*, *intercept_scaling=1.0*, *multi_class='ovr'*)

`decision_function`(*X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returnsarray, 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()

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.

Returnsself: estimator :

static **fit** (*X*, *y*)

Fit the model according to the given training data.

ParametersX : {array-like, sparse matrix}, 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.

Returnsself : object

Returns self.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*X*)

Predict class labels for samples in X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Samples.

ReturnsC : array, shape = [n_samples]

Predicted class label per sample.

predict_log_proba(*X*)
Log of probability estimates.

The returned estimates for all classes are ordered by the label of classes.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsT : 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_`.

predict_proba(*X*)
Probability estimates.

The returned estimates for all classes are ordered by the label of classes.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsT : array-like, 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(*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

sparsify()
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.

Returnsself: 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.

transform(*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

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=1*, *random_state=None*, *selection='cyclic'*)

Multi-task L1/L2 ElasticNet with built-in cross-validation.

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.

Parameterseps : float, optional

Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

alphas : array-like, optional

List of alphas where to compute the models. If not provided, set automatically.

n_alphas : int, optional

Number of alphas along the regularization path

l1_ratio : float or array of floats

The ElasticNet mixing parameter, with 0 < l1_ratio <= 1. For l1_ratio = 0 the penalty is an L1/L2 penalty. For l1_ratio = 1 it is an L1 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 (e.g. data is expected to be already centered).

normalize : boolean, optional, default False

If `True`, the regressors X will be normalized before regression.

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`.

cv : integer or cross-validation generator, optional

If an integer is passed, it is the number of fold (default 3). Specific cross-validation objects can be passed, see the `sklearn.cross_validation` module for the list of possible objects.

verbose : bool or integer

Amount of verbosity.

n_jobs : integer, optional

Number of CPUs to use during the cross validation. If `-1`, use all the CPUs. Note that this is used only if multiple values for l1_ratio are given.

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.

random_state : int, RandomState instance, or None (default)

The seed of the pseudo random number generator that selects a random feature to update. Useful only when selection is set to 'random'.

Attributesintercept_ : array, shape (n_tasks,)

Independent term in decision function.

coef_ : array, shape (n_tasks, n_features)

Parameter vector (W in the cost function formula).

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()
>>> clf.fit([[0,0], [1, 1], [2, 2]],
...         [[0, 0], [1, 1], [2, 2]])
...
MultiTaskElasticNetCV(alphas=None, copy_X=True, cv=None, eps=0.001,
       fit_intercept=True, l1_ratio=0.5, max_iter=1000, n_alphas=100,
       n_jobs=1, normalize=False, random_state=None, selection='cyclic',
       tol=0.0001, verbose=0)
>>> print(clf.coef_)
[[ 0.52875032  0.46958558]
 [ 0.52875032  0.46958558]]
>>> print(clf.intercept_)
[ 0.00166409  0.00166409]
```

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y)	Fit linear model with coordinate descent
get_params([deep])	Get parameters for this estimator.
path	
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static `__init__`(*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=1*, *random_state=None*, *selection='cyclic'*)

`decision_function`(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

`fit`(*X*, *y*)

Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

ParametersX : {array-like}, shape (n_samples, n_features)

Training data. Pass directly as float64, 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`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

`predict`(*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

`score`(*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

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=1*, *random_state=None*, *selection='cyclic'*)

Multi-task L1/L2 Lasso with built-in cross-validation.

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.

Parameterseps : float, optional

Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

alphas : array-like, optional

List of alphas where to compute the models. If not provided, set automaticlly.

n_alphas : int, optional

Number of alphas along the regularization path

fit_intercept : boolean

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 : boolean, optional, default False

If `True`, the regressors X will be normalized before regression.

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`.

cv : integer or cross-validation generator, optional

If an integer is passed, it is the number of fold (default 3). Specific cross-validation objects can be passed, see the `sklearn.cross_validation` module for the list of possible objects.

verbose : bool or integer

Amount of verbosity.

n_jobs : integer, optional

Number of CPUs to use during the cross validation. If `-1`, use all the CPUs. Note that this is used only if multiple values for l1_ratio are given.

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.

random_state : int, RandomState instance, or None (default)

The seed of the pseudo random number generator that selects a random feature to update. Useful only when selection is set to 'random'.

Attributesintercept_ : array, shape (n_tasks,)

Independent term in decision function.

coef_ : array, shape (n_tasks, n_features)

Parameter vector (W in the cost function formula).

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.

Methods

`decision_function`(X)	Decision function of the linear model.
`fit`(X, y)	Fit linear model with coordinate descent
`get_params`([deep])	Get parameters for this estimator.
`path`	
`predict`(X)	Predict using the linear model
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

static __init__(*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=1*, *random_state=None*, *selection='cyclic'*)

decision_function(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

fit(*X*, *y*)

Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

ParametersX : {array-like}, shape (n_samples, n_features)

Training data. Pass directly as float64, 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(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

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.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returns self :

sklearn.linear_model.OrthogonalMatchingPursuitCV

class sklearn.linear_model.**OrthogonalMatchingPursuitCV** (*copy=True*, *fit_intercept=True*, *normalize=True*, *max_iter=None*, *cv=None*, *n_jobs=1*, *verbose=False*)

Cross-validated Orthogonal Matching Pursuit model (OMP)

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 (e.g. data is expected to be already centered).

normalize : boolean, optional

If False, the regressors X are assumed to be already normalized.

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 : cross-validation generator, optional

see `sklearn.cross_validation`. If `None` is passed, default to a 5-fold strategy

n_jobs : integer, optional

Number of CPUs to use during the cross validation. If `-1`, use all the CPUs

verbose : boolean or integer, optional

Sets the verbosity amount

Attributesintercept_ : float or array, shape (n_targets,)

Independent term in decision function.

coef_ : array, shape (n_features,) or (n_features, n_targets)

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`

Methods

`decision_function`(X)	Decision function of the linear model.
`fit`(X, y)	Fit the model using X, y as training data.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict using the linear model
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*copy=True*, *fit_intercept=True*, *normalize=True*, *max_iter=None*, *cv=None*, *n_jobs=1*, *verbose=False*)

`decision_function`(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

static `fit`(*X*, *y*)

Fit the model using X, y as training data.

ParametersX : array-like, shape [n_samples, n_features]

Training data.

y : array-like, shape [n_samples]

Target values.

Returnsself : object

returns an instance of self.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.OrthogonalMatchingPursuitCV`

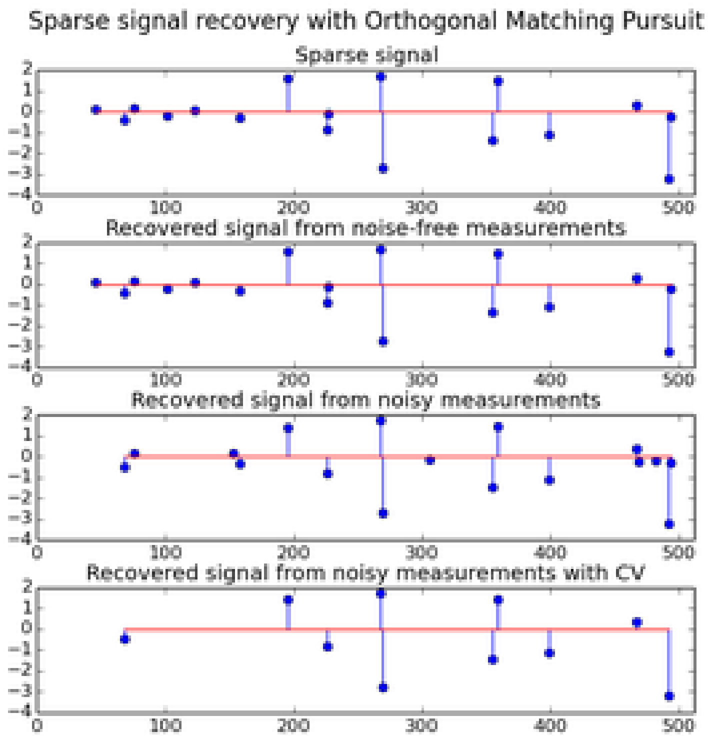


Figure 6.5: *Orthogonal Matching Pursuit*

`sklearn.linear_model`.RidgeCV

class `sklearn.linear_model.`**`RidgeCV`**(*alphas=array([0.1, 1., 10.]), fit_intercept=True, normalize=False, scoring=None, cv=None, gcv_mode=None, store_cv_values=False*)

Ridge regression with built-in cross-validation.

By default, it performs Generalized Cross-Validation, which is a form of efficient Leave-One-Out cross-validation.

Parametersalphas : numpy array of shape [n_alphas]

Array of alpha values to try. 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.

fit_intercept : boolean

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 : boolean, optional, default False

If True, the regressors X will be normalized before regression.

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)`.

cv : integer or cross-validation generator, optional

If None, Generalized Cross-Validation (efficient Leave-One-Out) will be used. If an integer is passed, it is the number of folds for KFold cross validation. Specific cross-validation objects can be passed, see sklearn.cross_validation module for the list of possible objects

gcv_mode : {None, 'auto', 'svd', eigen'}, optional

Flag indicating which strategy to use when performing Generalized Cross-Validation. Options are:

```
'auto' : use svd if n_samples > n_features or when X is a sparse
         matrix, otherwise use eigen
'svd' : force computation via singular value decomposition of X
        (does not work for sparse matrices)
'eigen' : force computation via eigendecomposition of X^T X
```

The 'auto' mode is the default and is intended to pick the cheaper option of the two depending upon the shape and format of the training data.

store_cv_values : boolean, 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).

Attributescv_values_ : array, 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_ : array, shape = [n_features] or [n_targets, n_features]

Weight vector(s).

alpha_ : float

Estimated regularization parameter.

intercept_ : float | array, shape = (n_targets,)

Independent term in decision function. Set to 0.0 if `fit_intercept = False`.

See also:

`Ridge`Ridge regression

`RidgeClassifier`Ridge classifier

`RidgeClassifierCV`Ridge classifier with built-in cross validation

Methods

`decision_function`(X)	Decision function of the linear model.
`fit`(X, y[, sample_weight])	Fit Ridge regression model
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict using the linear model
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.

Continued on next page

Table 6.10 – continued from previous page

set_params(**params)	Set the parameters of this estimator.

__init__(*alphas=array([0.1, 1., 10.]), fit_intercept=True, normalize=False, scoring=None, cv=None, gcv_mode=None, store_cv_values=False*)

decision_function(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

fit(*X, y, sample_weight=None*)

Fit Ridge regression model

ParametersX : array-like, shape = [n_samples, n_features]

Training data

y : array-like, shape = [n_samples] or [n_samples, n_targets]

Target values

sample_weight : float or array-like of shape [n_samples]

Sample weight

Returnsself : Returns self.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score(*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Figure 6.6: *Face completion with a multi-output estimators*

Examples using `sklearn.linear_model.RidgeCV`

`sklearn.linear_model`.RidgeClassifierCV

class `sklearn.linear_model.`**`RidgeClassifierCV`**(*alphas=array([0.1, 1., 10.]), fit_intercept=True, normalize=False, scoring=None, cv=None, class_weight=None*)

Ridge classifier with built-in cross-validation.

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.

Parametersalphas : numpy array of shape [n_alphas]

Array of alpha values to try. 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.

fit_intercept : boolean

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 : boolean, optional, default False

If True, the regressors X will be normalized before regression.

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)`.

cv : cross-validation generator, optional

If None, Generalized Cross-Validation (efficient Leave-One-Out) will be used.

class_weight : dict, optional

Weights associated with classes in the form `{class_label : weight}`. If not given, all classes are supposed to have weight one.

Attributescv_values_ : array, shape = [n_samples, n_alphas] or shape = [n_samples, n_responses, 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_ : array, shape = [n_features] or [n_targets, n_features]

Weight vector(s).

alpha_ : float

Estimated regularization parameter

See also:

RidgeRidge regression

RidgeClassifierRidge classifier

RidgeCVRidge 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.

Methods

`decision_function`(X)	Predict confidence scores for samples.
`fit`(X, y[, sample_weight])	Fit the ridge classifier.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict class labels for samples in X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.

static __init__(*alphas=array([0.1, 1., 10.]), fit_intercept=True, normalize=False, scoring=None, cv=None, class_weight=None*)

decision_function(*X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returnsarray, 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.

static fit(*X, y, sample_weight=None*)

Fit the ridge classifier.

ParametersX : 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 : float or numpy array of shape (n_samples,)

Sample weight.

Returnsself : object

Returns self.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*X*)

Predict class labels for samples in X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Samples.

ReturnsC : array, shape = [n_samples]

Predicted class label per sample.

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

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 benefitting from the Aikike 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=2.2204460492503131e-16*, *copy_X=True*)

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.

Parameterscriterion : 'bic' | 'aic'

The type of criterion to use.

fit_intercept : boolean

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).

verbose : boolean or integer, optional

Sets the verbosity amount

normalize : boolean, optional, default False

If True, the regressors X will be normalized before regression.

copy_X : boolean, optional, default True

If True, X will be copied; else, it may be overwritten.

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 : integer, optional

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.

Attributescoef_ : array, 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, shape (n_alphas,)

The value of the information criteria ('aic', 'bic') across all alphas. The alpha which has the smallest information criteria is chosen.

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.

http://en.wikipedia.org/wiki/Akaike_information_criterion http://en.wikipedia.org/wiki/Bayesian_information_criterion

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.LassoLarsIC(criterion='bic')
>>> clf.fit([[-1, 1], [0, 0], [1, 1]], [-1.1111, 0, -1.1111])
...
LassoLarsIC(copy_X=True, criterion='bic', eps=..., fit_intercept=True,
      max_iter=500, normalize=True, precompute='auto',
      verbose=False)
>>> print(clf.coef_)
[ 0.  -1.11...]
```

Methods

`decision_function`(X)	Decision function of the linear model.
`fit`(X, y[, copy_X])	Fit the model using X, y as training data.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict using the linear model
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

static __init__(*criterion='aic', fit_intercept=True, verbose=False, normalize=True, precompute='auto', max_iter=500, eps=2.2204460492503131e-16, copy_X=True*)

decision_function(*X*)
Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

static fit(*X, y, copy_X=True*)
Fit the model using X, y as training data.

ParametersX : array-like, shape (n_samples, n_features)

training data.

y : array-like, shape (n_samples,)

target values.

copy_X : boolean, optional, default True

If `True`, X will be copied; else, it may be overwritten.

Returnsself : object

returns an instance of self.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.LassoLarsIC`

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:

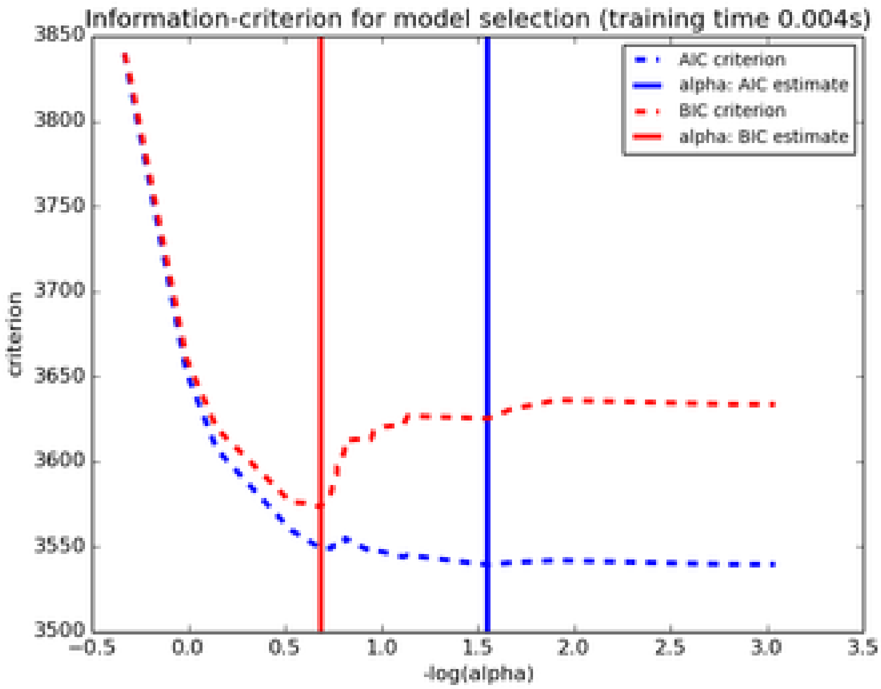


Figure 6.7: *Lasso model selection: Cross-Validation / AIC / BIC*

`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=10, 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, bootstrap=True, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False, class_weight=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 use averaging to improve the predictive accuracy and control over-fitting.

Parameters **n_estimators** : integer, optional (default=10)

The number of trees in the forest.

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_features : int, float, string or None, optional (default="auto")

The number of features to consider when looking for the best split:

The minimum number of samples required to split an internal node. Note: this parameter is tree-specific.

min_samples_leaf : integer, optional (default=1)

The minimum number of samples in newly created leaves. A split is discarded if after the split, one of the leaves would contain less then `min_samples_leaf` samples. Note: this parameter is tree-specific.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node. Note: this parameter is tree-specific.

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. If not None then `max_depth` will be ignored. Note: this parameter is tree-specific.

bootstrap : boolean, optional (default=True)

Whether bootstrap samples are used when building trees.

oob_score : bool

Whether to use out-of-bag samples to estimate the generalization error.

n_jobs : integer, optional (default=1)

The number of jobs to run in parallel for both *fit* and *predict*. If -1, then the number of jobs is set to the number of cores.

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.

class_weight : dict, list of dicts, “auto”, “subsample” 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.

The “auto” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data.

The “subsample” mode is the same as “auto” 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.

Attributesestimators_ : 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_ : array of shape = [n_features]

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.

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.

See also:

`DecisionTreeClassifier`, `ExtraTreesClassifier`

References

[R134]

Methods

`apply`(X)	Apply trees in the forest to X, return leaf indices.
`fit`(X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict class for X.
`predict_log_proba`(X)	Predict class log-probabilities for X.
`predict_proba`(X)	Predict class probabilities for X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, threshold])	Reduce X to its most important features.

static **`__init__`**(*n_estimators=10*, *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*, *bootstrap=True*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*, *warm_start=False*, *class_weight=None*)

`apply`(*X*)

Apply trees in the forest to X, return leaf indices.

ParametersX : 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`.

ReturnsX_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.

fit (*X*, *y*, *sample_weight=None*)

Build a forest of trees from the training set (X, y).

ParametersX : array-like or 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, shape = [n_samples] or [n_samples, n_outputs]

The target values (class labels in classification, real numbers in regression).

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.

Returnsself : object

Returns self.

fit_transform (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict class for X.

The predicted class of an input sample is computed as the majority prediction of the trees in the forest.

ParametersX : 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`.

Returnsy : array of shape = [n_samples] or [n_samples, n_outputs]

The predicted classes.

predict_log_proba(*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.

ParametersX : 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`.

Returnsp : 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(*X*)

Predict class probabilities for X.

The predicted class probabilities of an input sample is 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.

ParametersX : 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`.

Returnsp : 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(*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*, *threshold=None*)
Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

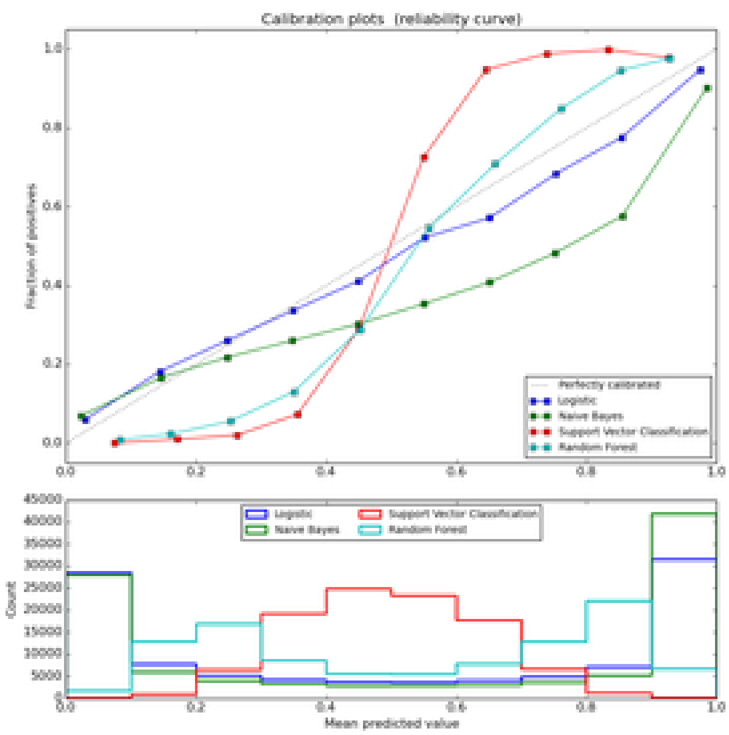


Figure 6.8: *Comparison of Calibration of Classifiers*

Examples using `sklearn.ensemble.RandomForestClassifier`

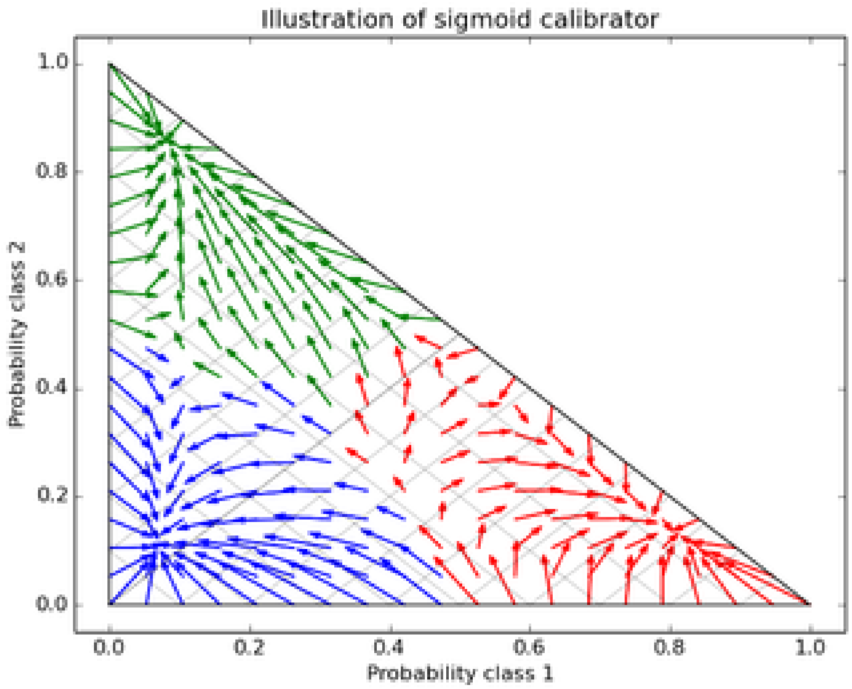


Figure 6.9: *Probability Calibration for 3-class classification*

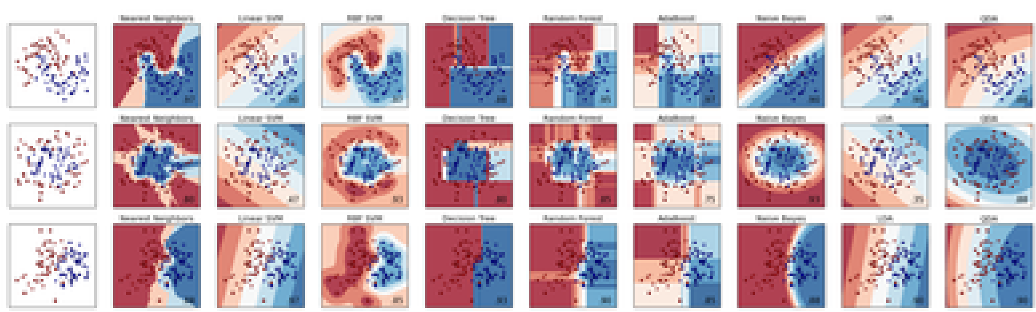


Figure 6.10: *Classifier comparison*

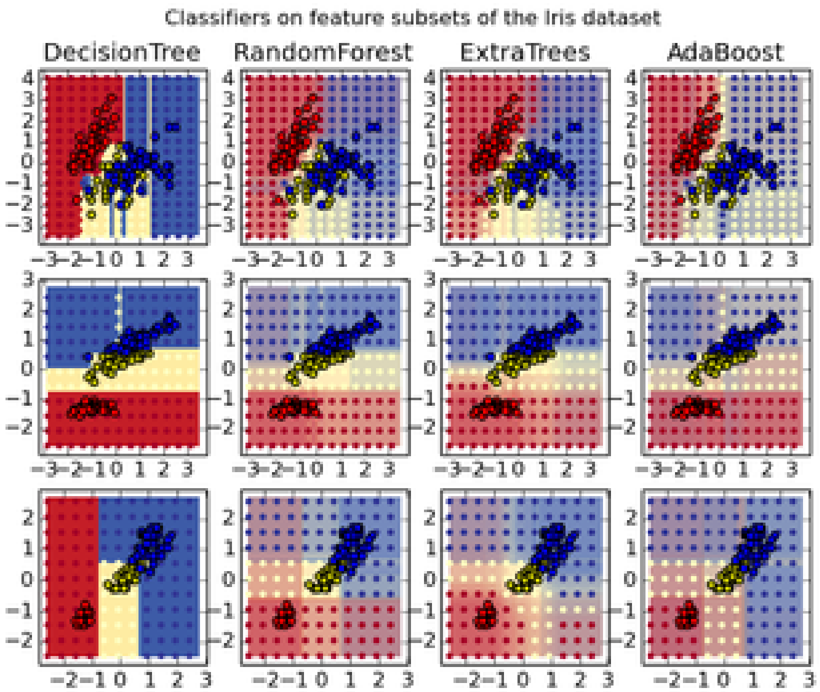


Figure 6.11: *Plot the decision surfaces of ensembles of trees on the iris dataset*

Figure 6.12: *Comparing randomized search and grid search for hyperparameter estimation*

Figure 6.13: *Classification of text documents using sparse features*

sklearn.ensemble.RandomForestRegressor

class sklearn.ensemble.**RandomForestRegressor**(*n_estimators=10*, *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*, *bootstrap=True*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*, *warm_start=False*)

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 use averaging to improve the predictive accuracy and control over-fitting.

Parametersn_estimators : integer, optional (default=10)

The number of trees in the forest.

criterion : string, optional (default="mse")

The function to measure the quality of a split. The only supported criterion is "mse" for the mean squared error. Note: this parameter is tree-specific.

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 percentage 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. 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. Ignored if `max_leaf_nodes` is not None. Note: this parameter is tree-specific.

min_samples_split : integer, optional (default=2)

The minimum number of samples required to split an internal node. Note: this parameter is tree-specific.

min_samples_leaf : integer, optional (default=1)

The minimum number of samples in newly created leaves. A split is discarded if after the split, one of the leaves would contain less then `min_samples_leaf` samples. Note: this parameter is tree-specific.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node. Note: this parameter is tree-specific.

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. If not None then `max_depth` will be ignored. Note: this parameter is tree-specific.

bootstrap : boolean, optional (default=True)

Whether bootstrap samples are used when building trees.

oob_score : bool

whether to use out-of-bag samples to estimate the generalization error.

n_jobs : integer, optional (default=1)

The number of jobs to run in parallel for both *fit* and *predict*. If -1, then the number of jobs is set to the number of cores.

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.

Attributesestimators_ : list of DecisionTreeRegressor

The collection of fitted sub-estimators.

feature_importances_ : array of shape = [n_features]

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.

oob_prediction_ : array of shape = [n_samples]

Prediction computed with out-of-bag estimate on the training set.

See also:

DecisionTreeRegressor, ExtraTreesRegressor

References

[R135]

Methods

apply(X)	Apply trees in the forest to X, return leaf indices.
fit(X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict regression target for X.
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.
transform(X[, threshold])	Reduce X to its most important features.

static `__init__`(*n_estimators=10, 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, bootstrap=True, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False*)

`apply`(*X*)

Apply trees in the forest to X, return leaf indices.

ParametersX : 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`.

ReturnsX_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.

`fit`(*X, y, sample_weight=None*)

Build a forest of trees from the training set (X, y).

ParametersX : array-like or 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, shape = [n_samples] or [n_samples, n_outputs]

The target values (class labels in classification, real numbers in regression).

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.

Returnsself : object

Returns self.

fit_transform (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*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.

ParametersX : 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`.

Returnsy : array of shape = [n_samples] or [n_samples, n_outputs]

The predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Figure 6.14: *Imputing missing values before building an estimator*

Examples using `sklearn.ensemble.RandomForestRegressor`

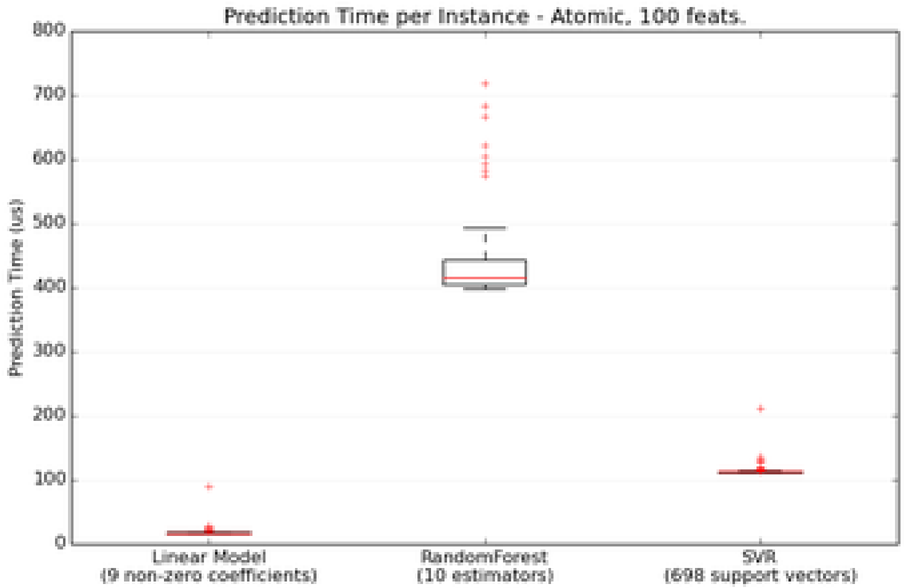


Figure 6.15: *Prediction Latency*

sklearn.ensemble.ExtraTreesClassifier

class sklearn.ensemble.**ExtraTreesClassifier**(*n_estimators=10*, *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*, *bootstrap=False*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*, *warm_start=False*, *class_weight=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 use averaging to improve the predictive accuracy and control over-fitting.

Parametersn_estimators : integer, optional (default=10)

The number of trees in the forest.

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_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 percentage 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. 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. Ignored if `max_leaf_nodes` is not None. Note: this parameter is tree-specific.

min_samples_split : integer, optional (default=2)

The minimum number of samples required to split an internal node. Note: this parameter is tree-specific.

min_samples_leaf : integer, optional (default=1)

The minimum number of samples in newly created leaves. A split is discarded if after the split, one of the leaves would contain less then `min_samples_leaf` samples. Note: this parameter is tree-specific.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node. Note: this parameter is tree-specific.

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. If not None then `max_depth` will be ignored. Note: this parameter is tree-specific.

bootstrap : boolean, optional (default=False)

Whether bootstrap samples are used when building trees.

oob_score : bool

Whether to use out-of-bag samples to estimate the generalization error.

n_jobs : integer, optional (default=1)

The number of jobs to run in parallel for both *fit* and *predict*. If -1, then the number of jobs is set to the number of cores.

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.

class_weight : dict, list of dicts, "auto", "subsample" 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.

The "auto" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data.

The "subsample" mode is the same as "auto" 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.

Attributesestimators_ : 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_ : array of shape = [n_features]

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.

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.

See also:

`sklearn.tree.ExtraTreeClassifier`Base classifier for this ensemble.

`RandomForestClassifier`Ensemble Classifier based on trees with optimal splits.

References

[R132]

Methods

`apply`(X)	Apply trees in the forest to X, return leaf indices.

Continued on next page

Table 6.17 – continued from previous page

`fit`(X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict class for X.
`predict_log_proba`(X)	Predict class log-probabilities for X.
`predict_proba`(X)	Predict class probabilities for X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, threshold])	Reduce X to its most important features.

static `__init__`(*n_estimators=10*, *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*, *bootstrap=False*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*, *warm_start=False*, *class_weight=None*)

`apply`(*X*)

Apply trees in the forest to X, return leaf indices.

ParametersX : 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`.

ReturnsX_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.

`fit`(*X*, *y*, *sample_weight=None*)

Build a forest of trees from the training set (X, y).

ParametersX : array-like or 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, shape = [n_samples] or [n_samples, n_outputs]

The target values (class labels in classification, real numbers in regression).

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.

Returnsself : object

Returns self.

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*X*)

Predict class for X.

The predicted class of an input sample is computed as the majority prediction of the trees in the forest.

ParametersX : 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`.

Returnsy : array of shape = [n_samples] or [n_samples, n_outputs]

The predicted classes.

predict_log_proba(*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.

ParametersX : 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`.

Returnsp : 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(*X*)

Predict class probabilities for X.

The predicted class probabilities of an input sample is 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.

ParametersX : 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`.

Returnsp : 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 (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*, *threshold=None*)
Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Examples using `sklearn.ensemble.ExtraTreesClassifier`

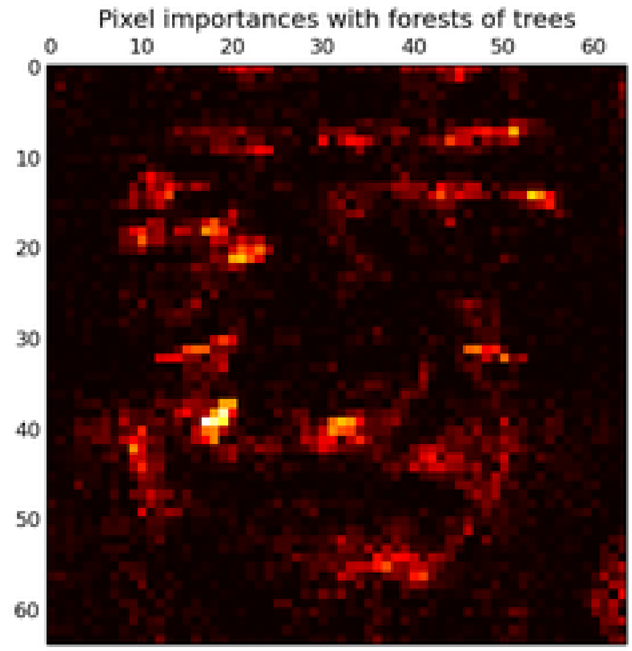


Figure 6.16: *Pixel importances with a parallel forest of trees*

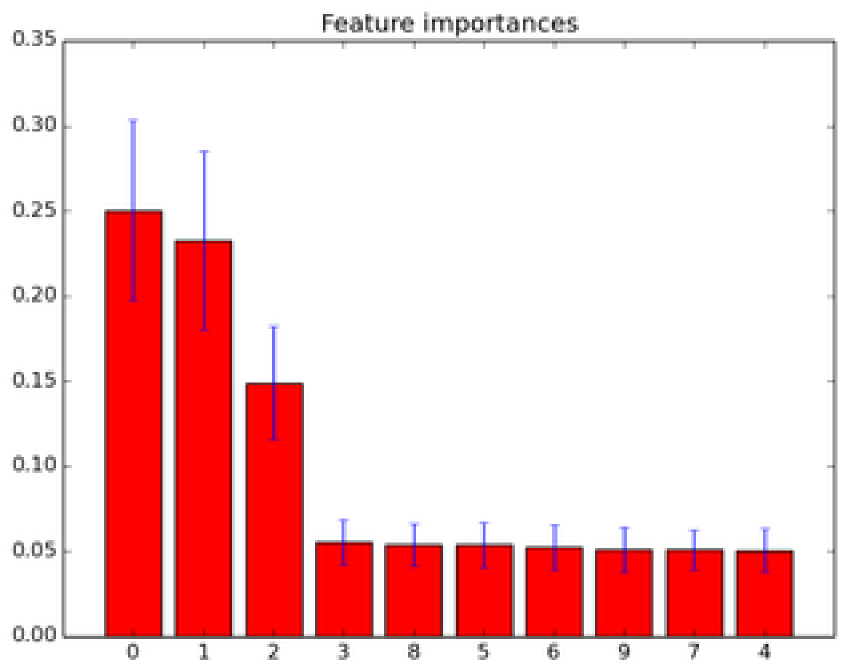


Figure 6.17: *Feature importances with forests of trees*

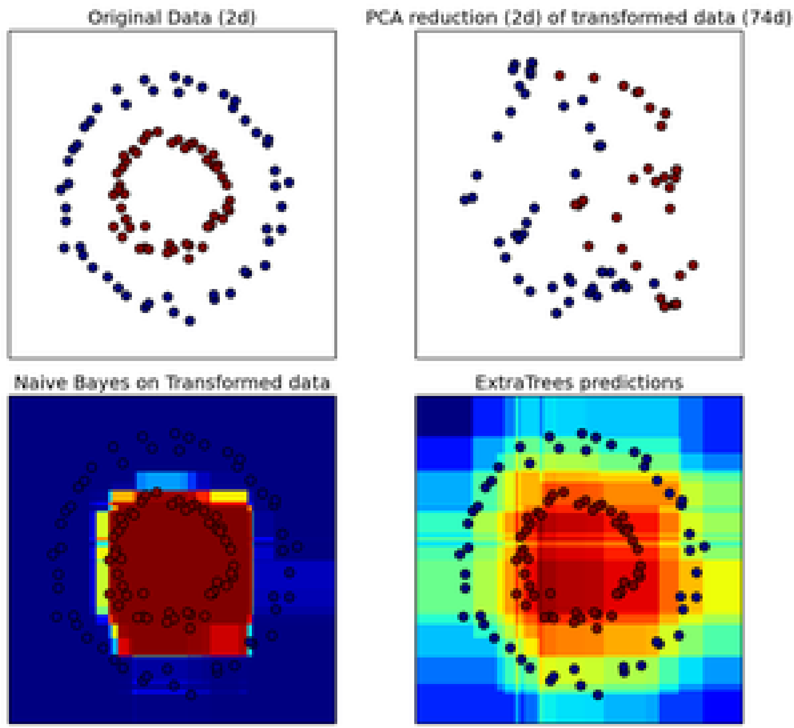


Figure 6.18: *Hashing feature transformation using Totally Random Trees*

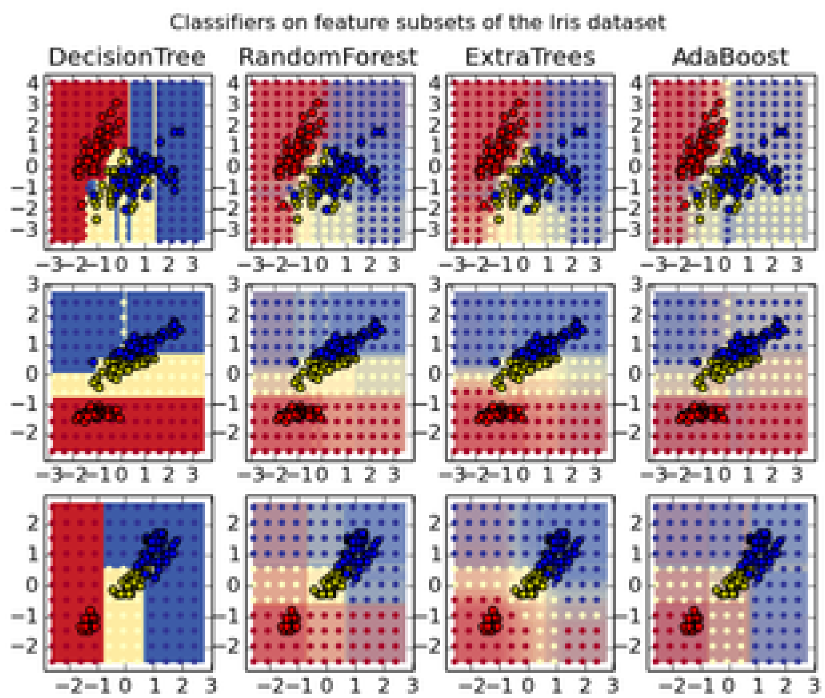


Figure 6.19: *Plot the decision surfaces of ensembles of trees on the iris dataset*

sklearn.ensemble.ExtraTreesRegressor

class sklearn.ensemble.**ExtraTreesRegressor**(*n_estimators=10, 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, bootstrap=False, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False*)

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 use averaging to improve the predictive accuracy and control over-fitting.

Parametersn_estimators : integer, optional (default=10)

The number of trees in the forest.

criterion : string, optional (default="mse")

The function to measure the quality of a split. The only supported criterion is "mse" for the mean squared error. Note: this parameter is tree-specific.

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 percentage 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. 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. Ignored if `max_leaf_nodes` is not None. Note: this parameter is tree-specific.

min_samples_split : integer, optional (default=2)

The minimum number of samples required to split an internal node. Note: this parameter is tree-specific.

min_samples_leaf : integer, optional (default=1)

The minimum number of samples in newly created leaves. A split is discarded if after the split, one of the leaves would contain less then `min_samples_leaf` samples. Note: this parameter is tree-specific.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node. Note: this parameter is tree-specific.

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. If not None then `max_depth` will be ignored. Note: this parameter is tree-specific.

bootstrap : boolean, optional (default=False)

Whether bootstrap samples are used when building trees. Note: this parameter is tree-specific.

oob_score : bool

Whether to use out-of-bag samples to estimate the generalization error.

n_jobs : integer, optional (default=1)

The number of jobs to run in parallel for both *fit* and *predict*. If -1, then the number of jobs is set to the number of cores.

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.

Attributesestimators_ : list of DecisionTreeRegressor

The collection of fitted sub-estimators.

feature_importances_ : array of shape = [n_features]

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.

oob_prediction_ : array of shape = [n_samples]

Prediction computed with out-of-bag estimate on the training set.

See also:

`sklearn.tree.ExtraTreeRegressor`Base estimator for this ensemble.

`RandomForestRegressor`Ensemble regressor using trees with optimal splits.

References

[R133]

Methods

`apply`(X)	Apply trees in the forest to X, return leaf indices.
`fit`(X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict regression target for X.
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, threshold])	Reduce X to its most important features.

static __init__(*n_estimators=10*, *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*, *bootstrap=False*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*, *warm_start=False*)

apply(*X*)

Apply trees in the forest to X, return leaf indices.

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 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.

fit(*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, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y : array-like, shape = [n_samples] or [n_samples, n_outputs]

The target values (class labels in classification, real numbers in regression).

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.

Returns self : object

Returns self.

fit_transform(*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.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*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.

ParametersX : 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`.

Returnsy : array of shape = [n_samples] or [n_samples, n_outputs]

The predicted values.

score(*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*, *threshold=None*)
Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Figure 6.20: *Face completion with a multi-output estimators*

Examples using `sklearn.ensemble.ExtraTreesRegressor`

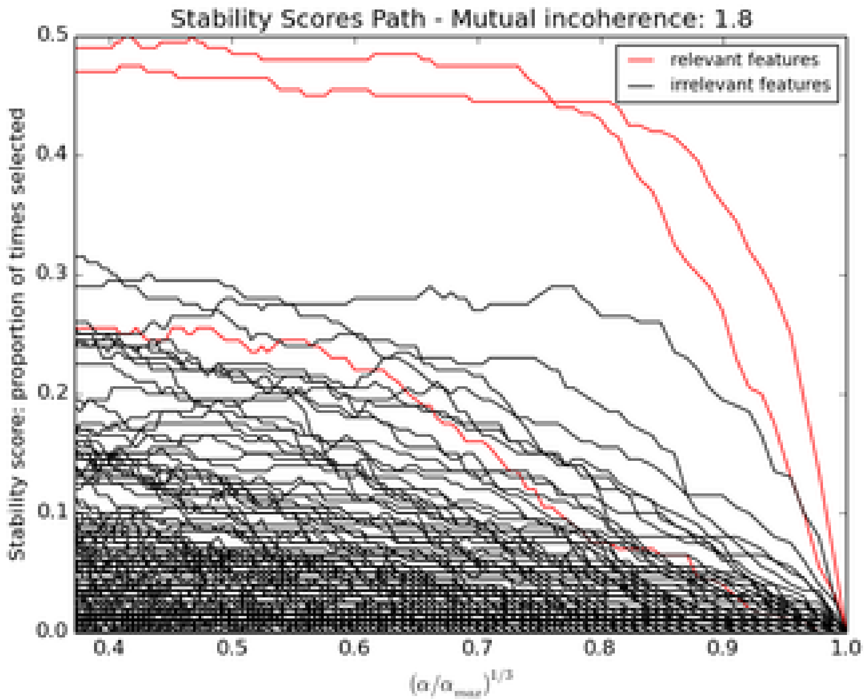


Figure 6.21: *Sparse recovery: feature selection for sparse linear models*

`sklearn.ensemble`.GradientBoostingClassifier

class `sklearn.ensemble.`**`GradientBoostingClassifier`**(*loss='deviance'*, *learning_rate=0.1*, *n_estimators=100*, *subsample=1.0*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_depth=3*, *init=None*, *random_state=None*, *max_features=None*, *verbose=0*, *max_leaf_nodes=None*, *warm_start=False*)

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.

Parametersloss : {'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.

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. Ignored if `max_leaf_nodes` is not None.

min_samples_split : integer, optional (default=2)

The minimum number of samples required to split an internal node.

min_samples_leaf : integer, optional (default=1)

The minimum number of samples required to be at a leaf node.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node.

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.

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 percentage 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.

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. If not None then `max_depth` will be ignored.

init : BaseEstimator, None, optional (default=None)

An estimator object that is used to compute the initial predictions. `init` has to provide `fit` and `predict`. If None it uses `loss.init_estimator`.

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.

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.

Attributesfeature_importances_ : array, shape = [n_features]

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.

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 : BaseEstimator

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.

See also:

`sklearn.tree.DecisionTreeClassifier`, `RandomForestClassifier`, `AdaBoostClassifier`

References

J. Friedman, Greedy Function Approximation: A Gradient Boosting Machine, The Annals of Statistics, Vol. 29, No. 5, 2001.

10.Friedman, Stochastic Gradient Boosting, 1999

T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning Ed. 2, Springer, 2009.

Methods

`decision_function`(X)	Compute the decision function of `X`.
`fit`(X, y[, sample_weight, monitor])	Fit the gradient boosting model.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict class for X.
`predict_log_proba`(X)	Predict class log-probabilities for X.

Continued on next page

Table 6.19 – continued from previous page

predict_proba(X)	Predict class probabilities for X.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.
staged_decision_function(X)	Compute decision function of X for each iteration.
staged_predict(X)	Predict class at each stage for X.
staged_predict_proba(X)	Predict class probabilities at each stage for X.
transform(X[, threshold])	Reduce X to its most important features.

static __init__(*loss='deviance'*, *learning_rate=0.1*, *n_estimators=100*, *subsample=1.0*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_depth=3*, *init=None*, *random_state=None*, *max_features=None*, *verbose=0*, *max_leaf_nodes=None*, *warm_start=False*)

decision_function(*X*)

Compute the decision function of X.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsscore : 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(*X*, *y*, *sample_weight=None*, *monitor=None*)

Fit the gradient boosting model.

ParametersX : 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 (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.

Returnsself : object

Returns self.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **predict**(*X*)

Predict class for X.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsy: array of shape = ["n_samples] :

The predicted values.

static **predict_log_proba**(*X*)

Predict class log-probabilities for X.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsp : array of shape = [n_samples]

The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

RaisesAttributeError :

If the `loss` does not support probabilities.

static **predict_proba**(*X*)

Predict class probabilities for X.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsp : array of shape = [n_samples]

The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

RaisesAttributeError :

If the `loss` does not support probabilities.

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

staged_decision_function (*X*)
Compute decision function of `X` for each iteration.

This method allows monitoring (i.e. determine error on testing set) after each stage.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsscore : generator of array, shape = [n_samples, k]

The decision function of the input samples. The order of the classes corresponds to that in the attribute *classes_*. Regression and binary classification are special cases with `k == 1`, otherwise `k==n_classes`.

static **staged_predict** (*X*)
Predict class at each stage for X.

This method allows monitoring (i.e. determine error on testing set) after each stage.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsy : generator of array of shape = [n_samples]

The predicted value of the input samples.

static **staged_predict_proba** (*X*)
Predict class probabilities at each stage for X.

This method allows monitoring (i.e. determine error on testing set) after each stage.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsy : generator of array of shape = [n_samples]

The predicted value of the input samples.

transform (*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

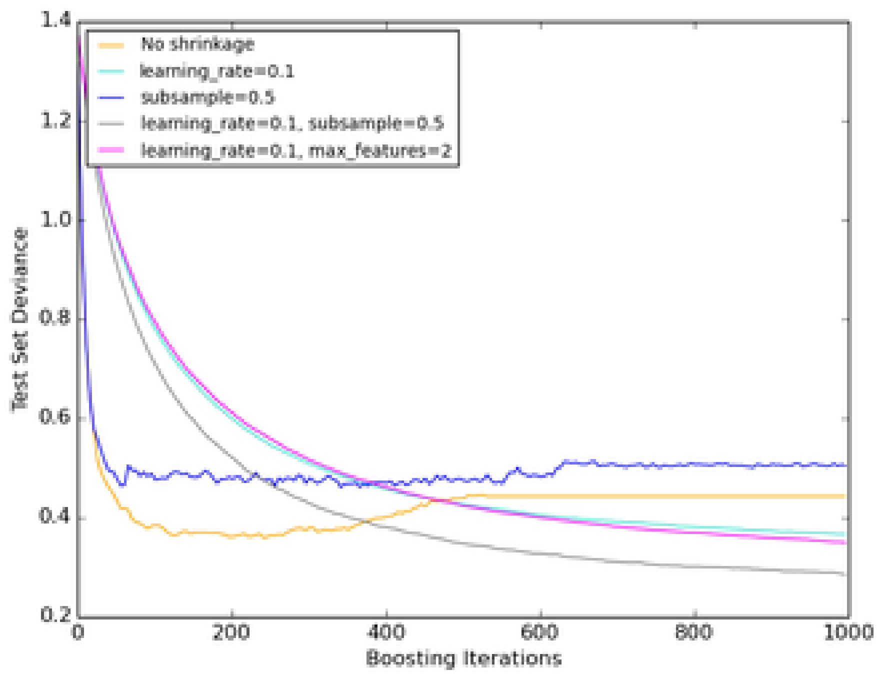


Figure 6.22: *Gradient Boosting regularization*

Examples using `sklearn.ensemble.GradientBoostingClassifier`

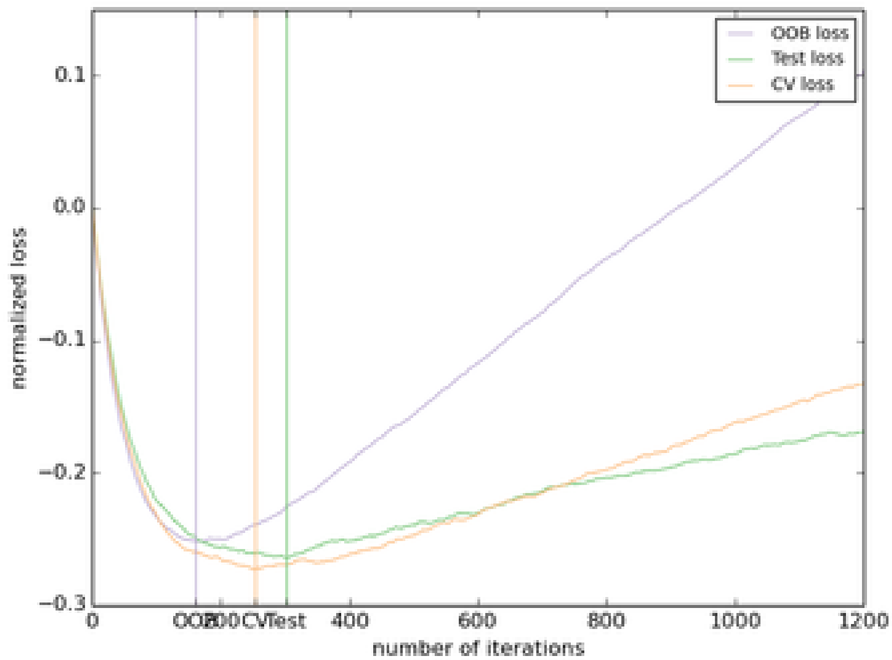


Figure 6.23: *Gradient Boosting Out-of-Bag estimates*

sklearn.ensemble.GradientBoostingRegressor

class sklearn.ensemble.**GradientBoostingRegressor**(*loss='ls'*, *learning_rate=0.1*, *n_estimators=100*, *subsample=1.0*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_depth=3*, *init=None*, *random_state=None*, *max_features=None*, *alpha=0.9*, *verbose=0*, *max_leaf_nodes=None*, *warm_start=False*)

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.

Parametersloss : {'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.

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. Ignored if `max_leaf_nodes` is not None.

min_samples_split : integer, optional (default=2)

The minimum number of samples required to split an internal node.

min_samples_leaf : integer, optional (default=1)

The minimum number of samples required to be at a leaf node.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node.

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.

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 percentage 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.

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.

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'`.

init : BaseEstimator, None, optional (default=None)

An estimator object that is used to compute the initial predictions. `init` has to provide `fit` and `predict`. If None it uses `loss.init_estimator`.

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.

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.

Attributesfeature_importances_ : array, shape = [n_features]

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.

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' : BaseEstimator

The estimator that provides the initial predictions. Set via the `init` argument or `loss.init_estimator`.

estimators_ : ndarray of DecisionTreeRegressor, shape = [n_estimators, 1]

The collection of fitted sub-estimators.

See also:

`DecisionTreeRegressor`, `RandomForestRegressor`

References

J. Friedman, Greedy Function Approximation: A Gradient Boosting Machine, The Annals of Statistics, Vol. 29, No. 5, 2001.

10.Friedman, Stochastic Gradient Boosting, 1999

T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning Ed. 2, Springer, 2009.

Methods

`decision_function`(X)	Compute the decision function of `X`.
`fit`(X, y[, sample_weight, monitor])	Fit the gradient boosting model.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict regression target for X.
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.

Continued on next page

Table 6.20 – continued from previous page

set_params(**params)	Set the parameters of this estimator.
staged_decision_function(X)	Compute decision function of X for each iteration.
staged_predict(X)	Predict regression target at each stage for X.
transform(X[, threshold])	Reduce X to its most important features.

static __init__ (*loss='ls'*, *learning_rate=0.1*, *n_estimators=100*, *subsample=1.0*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_depth=3*, *init=None*, *random_state=None*, *max_features=None*, *alpha=0.9*, *verbose=0*, *max_leaf_nodes=None*, *warm_start=False*)

decision_function (*X*)

Compute the decision function of X.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsscore : 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 (*X*, *y*, *sample_weight=None*, *monitor=None*)

Fit the gradient boosting model.

ParametersX : 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 (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.

Returnsself : object

Returns self.

fit_transform (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static predict (*X*)

Predict regression target for X.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsy : array of shape = [n_samples]

The predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

staged_decision_function (*X*)

Compute decision function of `X` for each iteration.

This method allows monitoring (i.e. determine error on testing set) after each stage.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsscore : generator of array, shape = [n_samples, k]

The decision function of the input samples. The order of the classes corresponds to that in the attribute *classes_*. Regression and binary classification are special cases with `k == 1`, otherwise `k==n_classes`.

static staged_predict (*X*)

Predict regression target at each stage for X.

This method allows monitoring (i.e. determine error on testing set) after each stage.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsy : generator of array of shape = [n_samples]

The predicted value of the input samples.

transform (*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Examples using sklearn.ensemble.GradientBoostingRegressor

6.3 Model evaluation: quantifying the quality of predictions

There are 3 different approaches to evaluate the quality of predictions of a model:

- **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 `cross_validation.cross_val_score` and `grid_search.GridSearchCV`) rely on an internal *scoring* strategy. This is discussed in the section *The scoring parameter: defining model evaluation rules*.

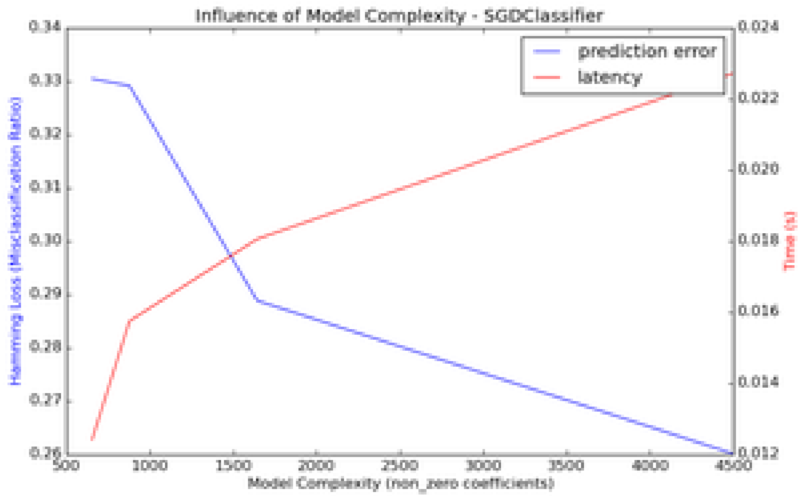


Figure 6.24: *Model Complexity Influence*

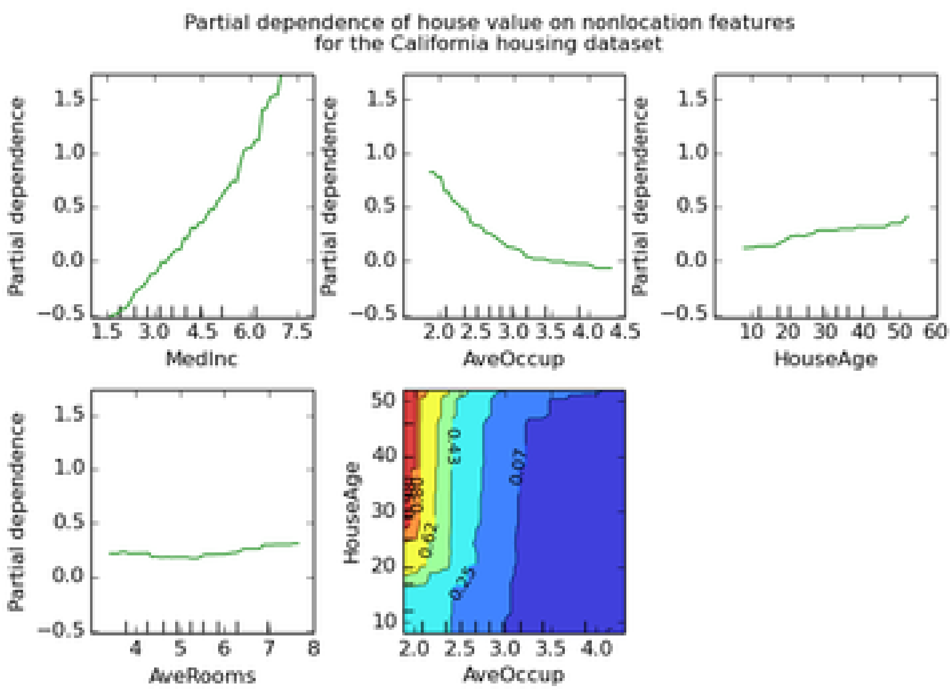


Figure 6.25: *Partial Dependence Plots*

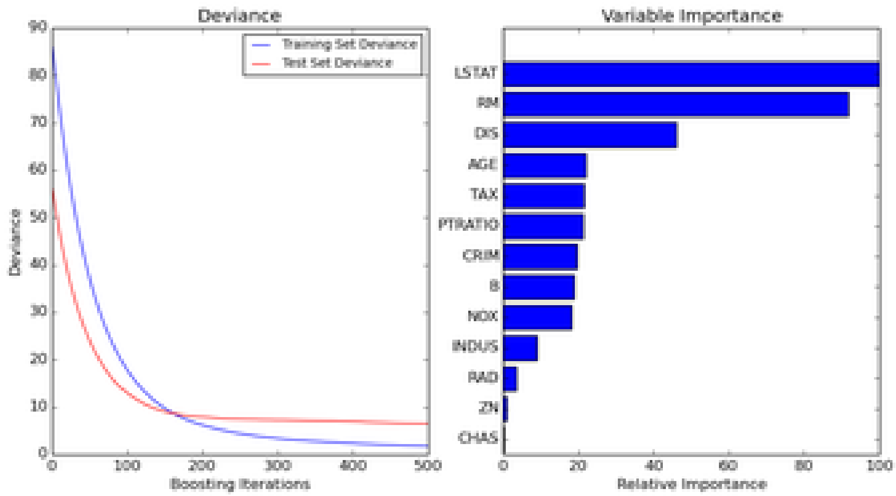


Figure 6.26: *Gradient Boosting regression*

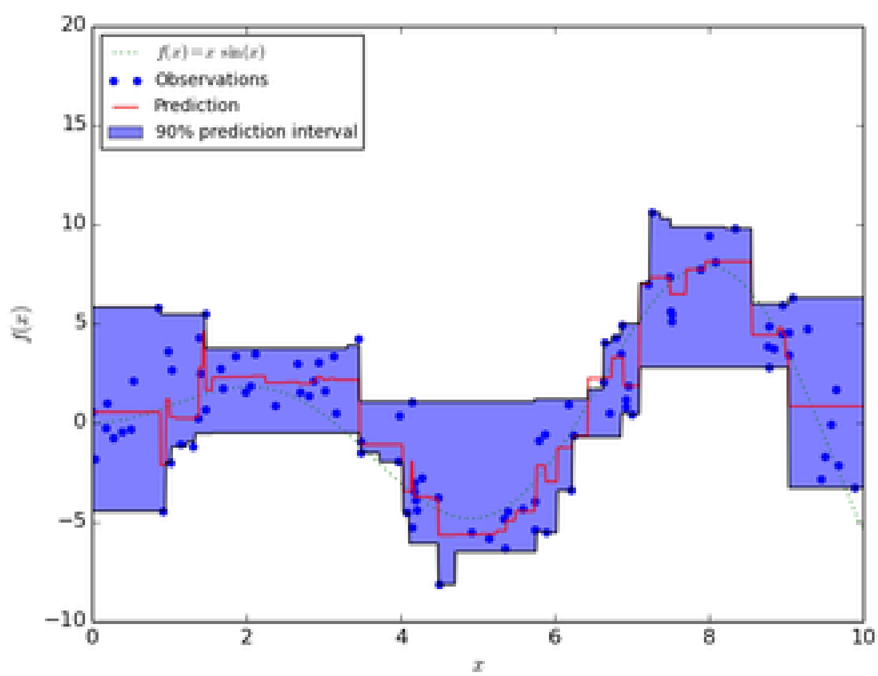


Figure 6.27: *Prediction Intervals for Gradient Boosting Regression*

- **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.

6.3.1 The `scoring` parameter: defining model evaluation rules

Model selection and evaluation using tools, such as `grid_search.GridSearchCV` and `cross_validation.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 ojects follow the convention that higher return values are better than lower return values. Thus the returns from mean_absolute_error and mean_squared_error, which measure the distance between the model and the data, are negated.

Scoring	Function	Comment
Classification		
'accuracy'	`metrics.accuracy_score`	
'average_precision'	`metrics.average_precision_score`	
'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
'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'
'roc_auc'	`metrics.roc_auc_score`	
Clustering		
'adjusted_rand_score'	`metrics.adjusted_rand_score`	
Regression		
'mean_absolute_error'	`metrics.mean_absolute_error`	
'mean_squared_error'	`metrics.mean_squared_error`	
'median_absolute_error'	`metrics.median_absolute_error`	
'r2'	`metrics.r2_score`	

Usage examples:

```
>>> from sklearn import svm, cross_validation, datasets
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> model = svm.SVC()
>>> cross_validation.cross_val_score(model, X, y, scoring='wrong_choice')
Traceback (most recent call last):
ValueError: 'wrong_choice' is not a valid scoring value. Valid options are ['accuracy', 'adjusted_ra
>>> clf = svm.SVC(probability=True, random_state=0)
>>> cross_validation.cross_val_score(clf, X, y, scoring='log_loss')
```

```
array([-0.07..., -0.16..., -0.06...])
Traceback (most recent call last):
ValueError: 'wrong_choice' is not a valid scoring value. Valid options are ['accuracy', 'adjusted_ran
```

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.metric` 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.grid_search import GridSearchCV
>>> from sklearn.svm import LinearSVC
>>> grid = GridSearchCV(LinearSVC(), param_grid={'C': [1, 10]}, scoring=ftwo_scorer)
```

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` in an `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(ground_truth, predictions):
...     diff = np.abs(ground_truth - predictions).max()
...     return np.log(1 + diff)
...
>>> # loss_func will negate the return value of my_custom_loss_func,
>>> #  which will be np.log(2), 0.693, given the values for ground_truth
>>> #  and predictions defined below.
```

```
>>> loss  = make_scorer(my_custom_loss_func, greater_is_better=False)
>>> score = make_scorer(my_custom_loss_func, greater_is_better=True)
>>> ground_truth = [1, 1]
>>> predictions  = [0, 1]
>>> from sklearn.dummy import DummyClassifier
>>> clf = DummyClassifier(strategy='most_frequent', random_state=0)
>>> clf = clf.fit(ground_truth, predictions)
>>> loss(clf,ground_truth, predictions)
-0.69...
>>> score(clf,ground_truth, predictions)
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.

6.3.2 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:

`matthews_corrcoef`(y_true, y_pred)	Compute the Matthews correlation coefficient (MCC) for binary classes
`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:

`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)

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[, classes])	Compute the average Hamming loss.
`jaccard_similarity_score`(y_true, y_pred[, ...])	Jaccard similarity coefficient score
`log_loss`(y_true, y_pred[, eps, normalize, ...])	Log loss, aka logistic loss or cross-entropy loss.

Continued on next page

Table 6.23 – continued from previous page

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
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
roc_auc_score(y_true, y_score[, average, ...])	Compute Area Under the Curve (AUC) 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 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.

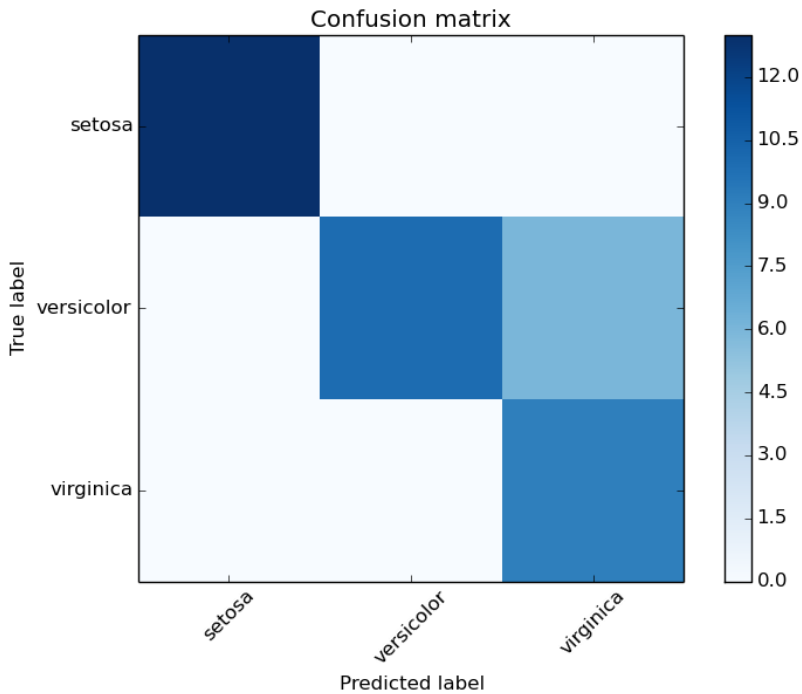
Confusion matrix

The `confusion_matrix` function evaluates classification accuracy by computing the confusion matrix.

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]])
```

Here is a visual representation of such a confusion matrix (this figure comes from the *Confusion matrix* example):



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, 2, 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      1.00      1.00         2

avg / total       0.67      0.80      0.72         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:

$$L_{Hamming}(y, \hat{y}) = \frac{1}{n_\text{labels}} \sum_{j=0}^{n_\text{labels} - 1} 1(\hat{y}_j \not= y_j)$$

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.

Jaccard similarity coefficient score

The `jaccard_similarity_score` function computes the average (default) or sum 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

$$J(y_i, \hat{y}_i) = \frac{|y_i \cap \hat{y}_i|}{|y_i \cup \hat{y}_i|}.$$

In binary and multiclass classification, the Jaccard similarity coefficient score is equal to the classification accuracy.

```
>>> import numpy as np
>>> from sklearn.metrics import jaccard_similarity_score
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]
>>> jaccard_similarity_score(y_true, y_pred)
```

```
0.5
>>> jaccard_similarity_score(y_true, y_pred, normalize=False)
2
```

In the multilabel case with binary label indicators:

```
>>> jaccard_similarity_score(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.75
```

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. This score corresponds to the area under the precision-recall curve.

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.

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.
- See *Sparse recovery: feature selection for sparse linear models* for an example of `precision_recall_curve` usage to select features for sparse linear models.

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:

Predicted class (expectation)	Actual class (observation) tp (true positive) Correct result fn (false negative) Missing result	 fp (false positive) Unexpected 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. ])
>>> threshold
array([ 0.35,  0.4 ,  0.8 ])
>>> average_precision_score(y_true, y_scores)
0.79...
```

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 for "micro"-averaging in a multiclass setting will produce equal precision, recall and F, while "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 $\hat{y}$
- $P(A, B) := \frac{|A \cap B|}{|A|}$
- $R(A, B) := \frac{|A \cap B|}{|B|}$ (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"	$\frac{1}{\|S\|} \sum_{s \in S} P(y_s, \hat{y}_s)$	$\frac{1}{\|S\|} \sum_{s \in S} R(y_s, \hat{y}_s)$	$\frac{1}{\|S\|} \sum_{s \in S} F_\beta(y_s, \hat{y}_s)$
"macro"	$\frac{1}{\|L\|} \sum_{l \in L} P(y_l, \hat{y}_l)$	$\frac{1}{\|L\|} \sum_{l \in L} R(y_l, \hat{y}_l)$	$\frac{1}{\|L\|} \sum_{l \in L} F_\beta(y_l, \hat{y}_l)$
"weighted"	$\frac{1}{\sum_{l \in L} \|\hat{y}_l\|} \sum_{l \in L} \|\hat{y}_l\| P(y_l, \hat{y}_l)$	$\frac{1}{\sum_{l \in L} \|\hat{y}_l\|} \sum_{l \in L} \|\hat{y}_l\| R(y_l, \hat{y}_l)$	$\frac{1}{\sum_{l \in L} \|\hat{y}_l\|} \sum_{l \in L} \|\hat{y}_l\| F_\beta(y_l, \hat{y}_l)$
None	$\langle P(y_l, \hat{y}_l) \| l \in L \rangle$	$\langle R(y_l, \hat{y}_l) \| l \in L \rangle$	$\langle F_\beta(y_l, \hat{y}_l) \| l \in L \rangle$

```
>>> 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')
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.        ,
```

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:

$$L_{\text{Hinge}}(y, w) = \max\left\{1 - wy, 0\right\} = \left|1 - wy\right|_+$$

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:

$$L_{\text{Hinge}}(y_w, y_t) = \max\left\{1 + y_t - y_w, 0\right\}$$

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(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=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(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
>>> 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 $p = \operatorname{Pr}(y = 1)$, the log loss per sample is the negative log-likelihood of the classifier given the true label:

$$L_{\log}(y, p) = -\log \operatorname{Pr}(y|p) = -(y \log p) + (1 - y) \log(1 - p))$$

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."

If tp, tn, fp and fn are respectively the number of true positives, true negatives, false positives and false negatives, the MCC coefficient is defined as

$$MCC = \frac{tp \times tn - fp \times fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}}.$$

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...
```

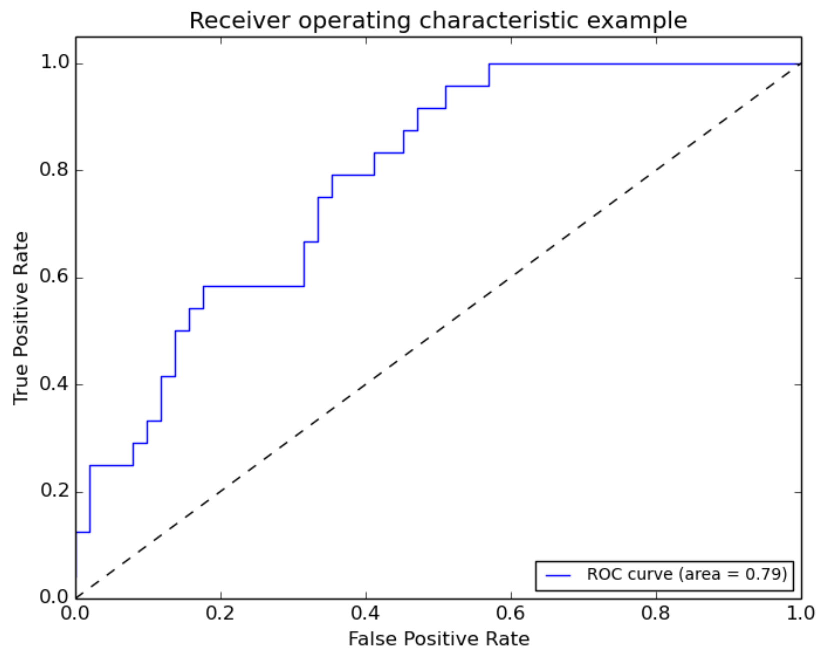
Receiver operating characteristic (ROC)

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. ,  0.5,  0.5,  1. ])
>>> tpr
array([ 0.5,  0.5,  1. ,  1. ])
>>> thresholds
array([ 0.8 ,  0.4 ,  0.35,  0.1 ])
```



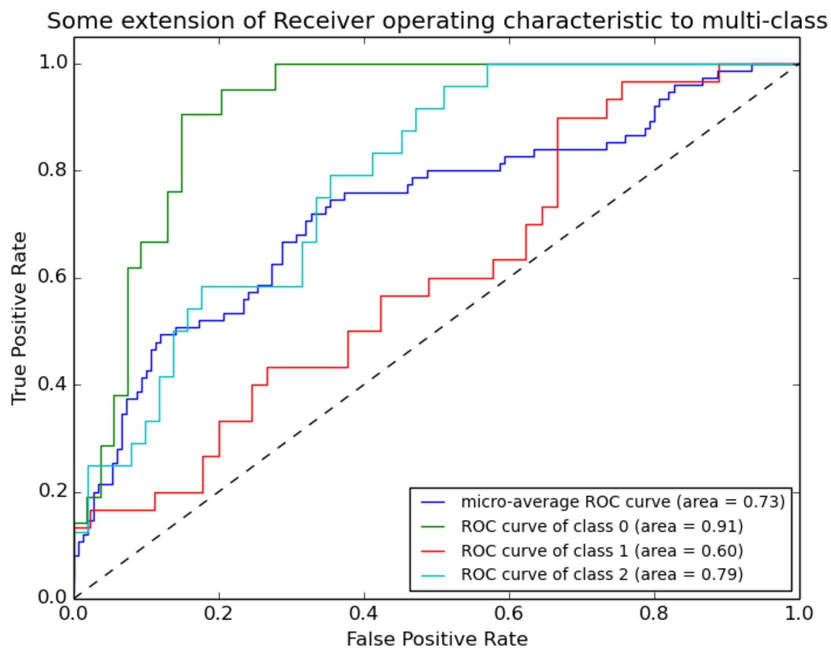
This figure shows an example of such an ROC curve:

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, if the predicted



outputs have been binarized.

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.

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.

6.3.3 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.

Formally, given a binary indicator matrix of the ground truth labels $y \in \left\{0, 1\right\}^{n_\text{samples} \times n_\text{labels}}$ 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) 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 will yield better scores 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 $y \in \mathcal{R}^{n_\text{samples} \times n_\text{labels}}$ and the score associated with each label $\hat{f} \in \mathcal{R}^{n_\text{samples} \times n_\text{labels}}$, the average precision is defined as

$$LRAP(y, \hat{f}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}} - 1} \frac{1}{|y_i|} \sum_{j:y_{ij} = 1} \frac{|\mathcal{L}_{ij}|}{\text{rank}_{ij}}$$

with $\mathcal{L}_{ij} = \left\{k: y_{ik} = 1, \hat{f}_{ik} \geq \hat{f}_{ij} \right\}$, $\text{rank}_{ij} = \left|\left\{k: \hat{f}_{ik} \geq \hat{f}_{ij} \right\}\right|$ and $|\cdot|$ is the l0 norm or the cardinality of the set.

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...
```

6.3.4 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_absolute_error`, `mean_squared_error`, `median_absolute_error` and `r2_score`.

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:

$$\texttt{explained\_variance}(y, \hat{y}) = 1 - \frac{Var\{ y - \hat{y}\}}{Var\{y\}}$$

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...
```

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

$$\text{MAE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \left| y_i - \hat{y}_i \right|.$$

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 squared error

The mean_squared_error function computes mean square error, a risk metric corresponding to the expected value of the squared (quadratic) error loss 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.

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
```

R² score, the coefficient of determination

The r2_score function computes R², the coefficient of determination. It provides a measure of how well future samples are likely to be predicted by the model.

If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value, then the score R² estimated over n_{samples} is defined as

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=0}^{n_{\text{samples}}-1}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{n_{\text{samples}}-1}(y_i - \bar{y})^2}$$

where $\bar{y} = \frac{1}{n_{\text{samples}}}\sum_{i=0}^{n_{\text{samples}}-1} y_i$.

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)
0.938...
```

Example:

- See *Lasso and Elastic Net for Sparse Signals* for an example of R² score usage to evaluate Lasso and Elastic Net on sparse signals.

6.3.5 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.

6.3.6 Dummy estimators

When doing supervised learning, a simple sanity check consists of comparing one's estimator against simple rules of thumb. DummyClassifier implements three 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.
- 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.cross_validation import train_test_split
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> 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(constant=None, 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)
0.97...
```

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 *Grid Search: Searching for estimator parameters* 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.

6.4 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.

6.4.1 Persistence example

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

```
>>> from sklearn import svm
>>> from sklearn import datasets
>>> clf = svm.SVC()
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> clf.fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3, gamma=0.0,
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)

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

In the specific case of the scikit, it may be more interesting to use joblib's replacement of pickle (`joblib.dump` & `joblib.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 sklearn.externals import joblib
>>> joblib.dump(clf, 'filename.pkl')
```

Later you can load back the pickled model (possibly in another Python process) with:

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

Note: joblib.dump returns a list of filenames. Each individual numpy array contained in the `clf` object is serialized as a separate file on the filesystem. All files are required in the same folder when reloading the model with joblib.load.

6.4.2 Security & maintainability limitations

pickle (and joblib by extension), has some issues regarding maintainability and security. Because of this,

- Never unpickle untrusted data
- Models saved in one version of scikit-learn might not load in another version.

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 a 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

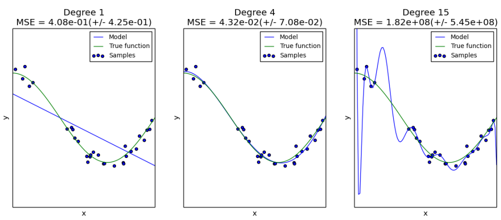
This should make it possible to check that the cross-validation score is in the same range as before.

If you want to know more about these issues and explore other possible serialization methods, please refer to this talk by Alex Gaynor.

6.5 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 hyper-parameters 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*

6.5.1 Validation curve

To validate a model we need a scoring function (see *Model evaluation: 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 *Grid Search: Searching for estimator parameters*) 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.

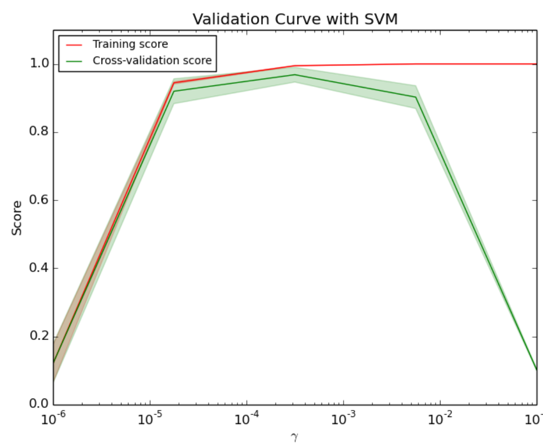
However, it is sometimes helpful to plot the influence of a single hyperparameter on the training score and the validation score to find out whether the estimator is overfitting or underfitting for some hyperparameter values.

The function `validation_curve` can help in this case:

```
>>> import numpy as np
>>> from sklearn.learning_curve import validation_curve
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import Ridge
```

```
>>> np.random.seed(0)
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> 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))
>>> train_scores
array([[ 0.94...,  0.92...,  0.92...],
       [ 0.94...,  0.92...,  0.92...],
       [ 0.47...,  0.45...,  0.42...]])
>>> valid_scores
array([[ 0.90...,  0.92...,  0.94...],
       [ 0.90...,  0.92...,  0.94...],
       [ 0.44...,  0.39...,  0.45...]])
```

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.

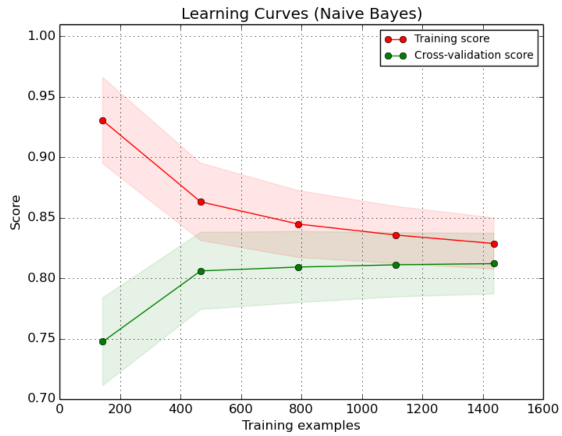


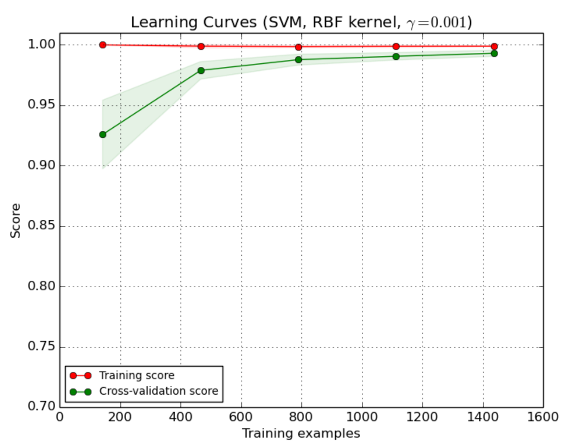
6.5.2 Learning curve

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. If both the validation score and the training score converge to a value that is too low with increasing size of the training set, we will not benefit much from more training data. In the following plot you can see an example: naive Bayes roughly converges to a low score.

We will probably have to use an estimator or a parametrization of the current estimator that can learn more complex concepts (i.e. has a lower bias). If the training score is much greater than the validation score for the maximum number of training samples, adding more training samples will most likely increase generalization. In the following plot you can see that the SVM could benefit from more training examples.

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


Learning Curves (Naive Bayes)
Training score
Cross-validation score
Score
Training examples
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0
200
400
600
800
1000
1200
1400
1600


Learning Curves (SVM, RBF kernel, $\gamma = 0.001$)
Training score
Cross-validation score
Score
Training examples
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0
200
400
600
800
1000
1200
1400
1600

sets):

```
>>> from sklearn.learning_curve 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...]])
```

CHAPTER

SEVEN

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 `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 *Pipeline and FeatureUnion: combining 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.

7.1 Pipeline and FeatureUnion: combining estimators

7.1.1 Pipeline: chaining estimators

`Pipeline` can be used to chain multiple estimators into one. This is useful as there is often a fixed sequence of steps in processing the data, for example feature selection, normalization and classification. `Pipeline` serves two purposes here:

> **Convenience**: You only have to call `fit` and `predict` once on your data to fit a whole sequence of estimators.
>
> **Joint parameter selection**: You can *grid search* over parameters of all estimators in the pipeline at once.

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

The `Pipeline` is build using a list of `(key, value)` pairs, where the `key` 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()), ('svm', SVC())]
>>> clf = Pipeline(estimators)
>>> clf
Pipeline(steps=[('reduce_dim', PCA(copy=True, n_components=None,
```

```
    whiten=False)), ('svm', SVC(C=1.0, cache_size=200, class_weight=None,
    coef0=0.0, degree=3, gamma=0.0, kernel='rbf', max_iter=-1,
    probability=False, random_state=None, shrinking=True, tol=0.001,
    verbose=False))])
```

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(copy=True, threshold=0.0)),
                ('multinomialnb', MultinomialNB(alpha=1.0,
                                                class_prior=None,
                                                fit_prior=True))])
```

The estimators of a pipeline are stored as a list in the `steps` attribute:

```
>>> clf.steps[0]
('reduce_dim', PCA(copy=True, n_components=None, whiten=False))
```

and as a `dict` in `named_steps`:

```
>>> clf.named_steps['reduce_dim']
PCA(copy=True, n_components=None, whiten=False)
```

Parameters of the estimators in the pipeline can be accessed using the `<estimator>__<parameter>` syntax:

```
>>> clf.set_params(svm__C=10)
Pipeline(steps=[('reduce_dim', PCA(copy=True, n_components=None,
    whiten=False)), ('svm', SVC(C=10, cache_size=200, class_weight=None,
    coef0=0.0, degree=3, gamma=0.0, kernel='rbf', max_iter=-1,
    probability=False, random_state=None, shrinking=True, tol=0.001,
    verbose=False))])
```

This is particularly important for doing grid searches:

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = dict(reduce_dim__n_components=[2, 5, 10],
...               svm__C=[0.1, 10, 100])
>>> grid_search = GridSearchCV(clf, param_grid=params)
```

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*

See also:

- *Grid Search: Searching for estimator parameters*

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.

7.1.2 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. For transforming data, the transformers are applied in parallel, and the sample vectors they output are concatenated end-to-end into larger vectors.

`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 is 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(n_jobs=1, transformer_list=[('linear_pca', PCA(copy=True,
    n_components=None, whiten=False)), ('kernel_pca', KernelPCA(alpha=1.0,
    coef0=1, degree=3, eigen_solver='auto', fit_inverse_transform=False,
    gamma=None, kernel='linear', kernel_params=None, max_iter=None,
    n_components=None, remove_zero_eig=False, tol=0))],
    transformer_weights=None)
```

Like pipelines, feature unions have a shorthand constructor called `make_union` that does require manual naming of the components.

Examples:

- *Concatenating multiple feature extraction methods*
- *Feature Union with Heterogeneous Data Sources*

7.2 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.

7.2.1 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 Fransisco', '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 Fransisco', '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`.

7.2.2 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.

If `non_negative=True` is passed to the constructor, the absolute value is taken. This undoes some of the collision handling, but allows 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.

7.2.3 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(min_df=1)
>>> vectorizer
CountVectorizer(analyzer=...'word', binary=False, decode_error=...'strict',
        dtype=<... 'numpy.int64'>, encoding=...'utf-8', input=...'content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
        strip_accents=None, token_pattern=...'(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

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, 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]...)
```

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**. This is a 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.

This normalization is implemented by the `text.TfidfTransformer` class:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> transformer = TfidfTransformer()
>>> transformer
TfidfTransformer(norm=...'l2', smooth_idf=True, sublinear_tf=False,
                 use_idf=True)
```

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([[ 0.85...,  0.  ...,  0.52...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 0.55...,  0.83...,  0.  ...],
       [ 0.63...,  0.  ...,  0.77...]])
```

Each row is normalized to have unit euclidean norm. 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(min_df=1)
>>> 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
>>> 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
>>> decoded = [x.decode(chardet.detect(x)['encoding'])
...            for x in (text1, text2, text3)]
```

```
>>> v = CountVectorizer().fit(decoded).vocabulary_
>>> for term in v: print(v)
```

(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)*:

- *Topics extraction with Non-Negative Matrix Factorization*

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), min_df=1)
>>> 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), min_df=1)
>>> 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), min_df=1)
>>> 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(object):
  ...     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.)

Customizing the vectorizer can also be useful when handling Asian languages that do not use an explicit word separator such as whitespace.

7.2.4 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],
       [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:

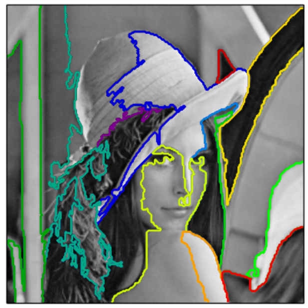
For this purpose, the estimators use a 'connectivity' matrix, giving which samples are connected.

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 Lena image*
- *Spectral clustering for image segmentation*



- *Feature agglomeration vs. univariate selection*

7.3 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.

7.3.1 Standardization, or mean removal and variance scaling

Standardization of datasets is a **common requirement for many machine learning estimators** implemented in the scikit: they might behave badly if the individual feature 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 that 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 = np.array([[ 1., -1.,  2.],
...               [ 2.,  0.,  0.],
...               [ 0.,  1., -1.]])
>>> X_scaled = preprocessing.scale(X)

>>> 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)
>>> scaler
StandardScaler(copy=True, with_mean=True, with_std=True)

>>> scaler.mean_
array([ 1. ...,  0. ...,  0.33...])

>>> scaler.std_
array([ 0.81...,  0.81...,  1.24...])

>>> scaler.transform(X)
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:

```
>>> scaler.transform([[-1.,  1., 0.]])
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. This can be achieved using `MinMaxScaler`.

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)
>>> 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
```

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` or `sklearn.decomposition.RandomizedPCA` with `whiten=True` to further remove the linear correlation across features.

Sparse input

`scale` and `StandardScaler` accept `scipy.sparse` matrices as input **only when 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.
If the centered data is expected to be small enough, explicitly convert the input to an array using the `toarray` method of sparse matrices instead.
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.

Scaling target variables in regression

`scale` and `StandardScaler` work out-of-the-box with 1d arrays. This is very useful for scaling the target / response variables used for regression.

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 phi, a `KernelCenterer` can transform the kernel matrix so that it contains inner products in the feature space defined by *phi* followed by removal of the mean in that space.

7.3.2 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:

```
>>> 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(copy=True, norm='l2')
```

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.  ...]])
```

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.

7.3.3 Binarization

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(copy=True, threshold=0.0)

>>> 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.

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.

7.3.4 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]`.

Such integer representation can not be used directly with 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).

One possibility to convert categorical features to features that can be used with scikit-learn estimators is to use a one-of-K or one-hot encoding, which is implemented in `OneHotEncoder`. This estimator transforms each categorical feature with `m` possible values into `m` binary features, with only one active.

Continuing the example above:

```
>>> enc = preprocessing.OneHotEncoder()
>>> enc.fit([[0, 0, 3], [1, 1, 0], [0, 2, 1], [1, 0, 2]])
OneHotEncoder(categorical_features='all', dtype=<... 'float'>,
       handle_unknown='error', n_values='auto', sparse=True)
>>> enc.transform([[0, 1, 3]]).toarray()
array([[ 1.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  1.]])
```

By default, how many values each feature can take is inferred automatically from the dataset. It is possible to specify this explicitly using the parameter `n_values`. There are two genders, three possible continents and four web browsers in our dataset. Then we fit the estimator, and transform a data point. In the result, the first two numbers encode the gender, the next set of three numbers the continent and the last four the web browser.

See *Loading features from dicts* for categorical features that are represented as a dict, not as integers.

7.3.5 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.

The `Imputer` class provides basic strategies for imputing missing values, either using the mean, the median or the most frequent value of the row or 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.preprocessing import Imputer
>>> imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
>>> imp.fit([[1, 2], [np.nan, 3], [7, 6]])
Imputer(axis=0, copy=True, missing_values='NaN', strategy='mean', verbose=0)
>>> X = [[np.nan, 2], [6, np.nan], [7, 6]]
>>> print(imp.transform(X))
[[ 4.          2.        ]
 [ 6.          3.666...]
 [ 7.          6.        ]]
```

The `Imputer` class also supports sparse matrices:

```
>>> import scipy.sparse as sp
>>> X = sp.csc_matrix([[1, 2], [0, 3], [7, 6]])
>>> imp = Imputer(missing_values=0, strategy='mean', axis=0)
>>> imp.fit(X)
Imputer(axis=0, copy=True, missing_values=0, strategy='mean', verbose=0)
>>> X_test = sp.csc_matrix([[0, 2], [6, 0], [7, 6]])
```

```
>>> print(imp.transform(X_test))
[[ 4.          2.        ]
 [ 6.          3.666...]
 [ 7.          6.        ]]
```

Note that, here, missing values are encoded by 0 and are thus implicitly stored in the matrix. This format is thus suitable when there are many more missing values than observed values.

`Imputer` 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*

7.4 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*.

7.4.1 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*

7.4.2 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*

7.4.3 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.

7.5 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.

7.5.1 The Johnson-Lindenstrauss lemma

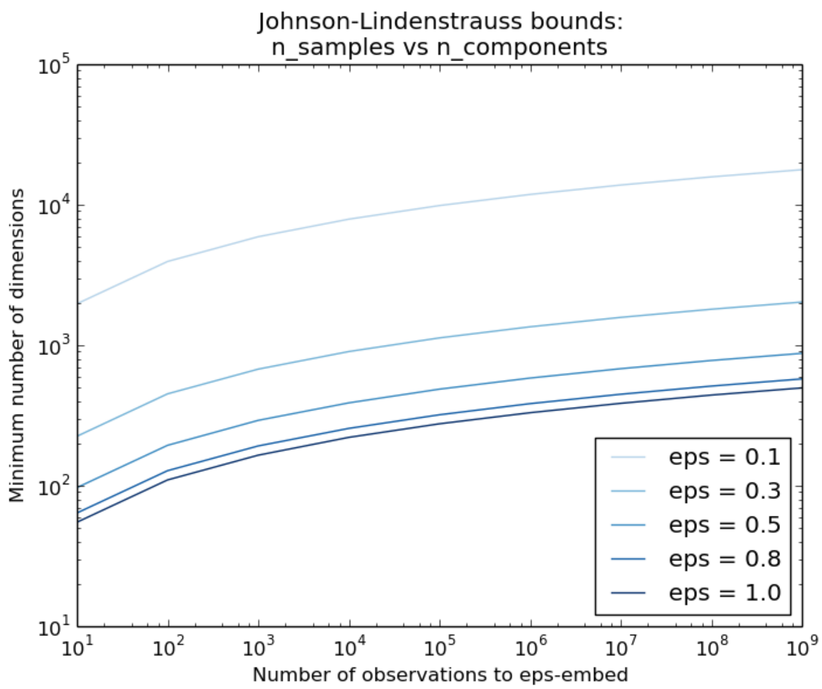
The main theoretical result behind the efficiency of random projection is the Johnson-Lindenstrauss lemma (quoting Wikipedia):

> 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:

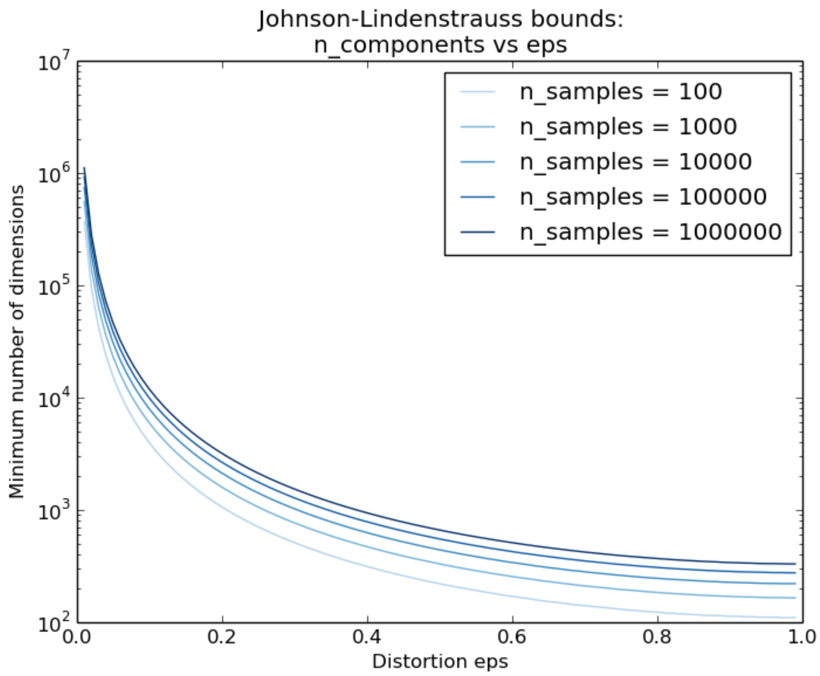
- Sanjoy Dasgupta and Anupam Gupta, 1999. An elementary proof of the Johnson-Lindenstrauss Lemma.

7.5.2 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)
```



7.5.3 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

$$\left\{\begin{array}{c c l} -\sqrt{\frac{s}{n_{\text{components}}}} & & 1/2s \\ 0 & \text{with probability} & 1-1/s \\ +\sqrt{\frac{s}{n_{\text{components}}}} & & 1/2s \end{array}\right.$$

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.

7.6 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.

7.6.1 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`.

7.6.2 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()
>>> clf.fit(X_features, y)
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', n_iter=5, n_jobs=1,
       penalty='l2', power_t=0.5, random_state=None, shuffle=True,
       verbose=0, warm_start=False)
```

```
>>> 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.

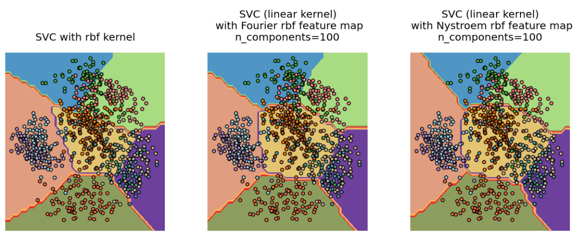


Figure 7.1: Comparing an exact RBF kernel (left) with the approximation (right)

Examples:

- *Explicit feature map approximation for RBF kernels*

7.6.3 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`.

7.6.4 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].

7.6.5 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:

7.7 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))`

7.7.1 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:

$$k(x, y) = \frac{x y^\top}{\|x\| \|y\|}$$

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. http://nlp.stanford.edu/IR-book/html/htmledition/the-vector-space-model-for-scoring-1.html

7.7.2 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:

$$k(x, y) = x^\top y$$

7.7.3 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.

7.7.4 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

7.7.5 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.

7.7.6 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.        ,  0.36...,  0.89...,  0.58...],
       [ 0.36...,  1.        ,  0.51...,  0.83...],
       [ 0.89...,  0.51...,  1.        ,  0.77... ],
       [ 0.58...,  0.83...,  0.77... ,  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 http://eprints.pascal-network.org/archive/00002309/01/Zhang06-IJCV.pdf

7.8 Transforming the prediction target (y)

7.8.1 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(neg_label=0, pos_label=1, sparse_output=False)
>>> 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])
```

7.8.2 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']
```

CHAPTER

EIGHT

DATASET LOADING UTILITIES

The `sklearn.datasets` package embeds some small toy datasets as introduced in the *Getting Started* section.

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.

This package also features helpers to fetch larger datasets commonly used by the machine learning community to benchmark algorithm on data that comes from the 'real world'.

8.1 General dataset API

There are three distinct kinds of dataset interfaces for different types of datasets. The simplest one is the interface for sample images, which is described below in the *Sample images* section.

The dataset generation functions and the svmlight loader share a simplistic interface, returning 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`.

The toy datasets as well as the 'real world' datasets and the datasets fetched from mldata.org have more sophisticated structure. These 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`.

The datasets also contain a description in `DESCR` and some contain `feature_names` and `target_names`. See the dataset descriptions below for details.

8.2 Toy datasets

scikit-learn comes with a few small standard datasets that do not require to download any file from some external website.

`load_boston`()	Load and return the boston house-prices dataset (regression).
`load_iris`()	Load and return the iris dataset (classification).
`load_diabetes`()	Load and return the diabetes dataset (regression).
`load_digits`([n_class])	Load and return the digits dataset (classification).
`load_linnerud`()	Load and return the linnerud dataset (multivariate regression).

These datasets are useful to quickly illustrate the behavior of the various algorithms implemented in the scikit. They are however often too small to be representative of real world machine learning tasks.

8.3 Sample images

The scikit also embed a couple of sample JPEG images published under Creative Commons license by their authors. Those image 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. Ofte algorithms work best if the input is converted to a floating point representation first. Also, `pylab.imshow` don't forget to scale to the range 0 - 1 as done in the following example.

Examples:

- *Color Quantization using K-Means*

8.4 Sample generators

In addition, scikit-learn includes various random sample generators that can be used to build artificial datasets of controlled size and complexity.

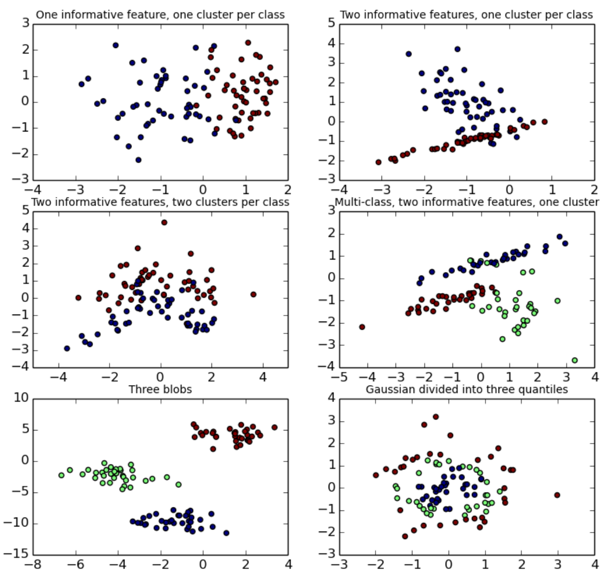
8.4.1 Generators for classification and clustering

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.

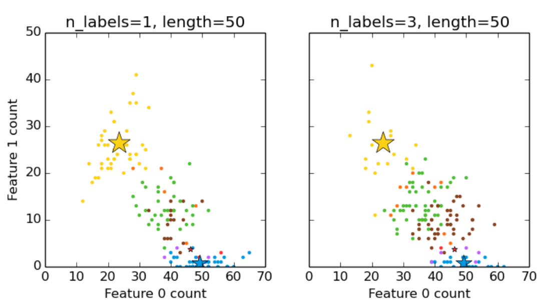


make_circles and make_moons generate 2d binary classification datasets that are challenging to certain algorithms (e.g. centroid-based clustering or linear classification), including optional Gaussian noise. They are useful for visualisation. produces Gaussian data with a spherical decision boundary for binary classification.

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.

Continued on next page

Table 8.3 – continued from previous page

make_checkerboard(shape, n_clusters[, ...])	Generate an array with block checkerboard structure for biclustering.

8.4.2 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.

8.4.3 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.

8.4.4 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.

8.5 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: http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
Faster API-compatible implementation: https://github.com/mblondel/svmlight-loader

8.6 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 website describing the original dataset is now defunct, but archived copies can be accessed through the Internet Archive's Wayback Machine. 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).

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.

8.7 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.

8.7.1 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))
['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([12,  6,  9,  8,  6,  7,  9,  2, 13, 19])
```

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([1, 1, 1, 0, 1, 0, 0, 1, 1, 1])
```

8.7.2 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',
...               '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])
159.01327433628319
```

`sklearn.datasets.fetch_20newsgroups_vectorized` is a function which returns ready-to-use tfidf features instead of file names.

8.7.3 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)
>>> pred = clf.predict(vectors_test)
>>> metrics.f1_score(newsgroups_test.target, pred, average='weighted')
0.88251152461278892
```

(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: sgi livesey atheists writes people caltech com god keith edu
comp.graphics: organization thanks files subject com image lines university edu graphics
sci.space: toronto moon gov com alaska access henry nasa edu space
talk.religion.misc: article writes kent people christian jesus sandvik edu com god
```

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='weighted')
0.78409163025839435
```

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 = BernoulliNB(alpha=.01)
>>> clf.fit(vectors, newsgroups_train.target)
>>> vectors_test = vectorizer.transform(newsgroups_test.data)
>>> pred = clf.predict(vectors_test)
>>> metrics.f1_score(newsgroups_test.target, pred, average='weighted')
0.73160869205141166
```

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*

8.8 Downloading datasets from the mldata.org repository

mldata.org is a public repository for machine learning data, supported by the PASCAL network .

The `sklearn.datasets` package is able to directly download data sets from the repository using the function `sklearn.datasets.fetch_mldata`.

For example, to download the MNIST digit recognition database:

```
>>> from sklearn.datasets import fetch_mldata
>>> mnist = fetch_mldata('MNIST original', data_home=custom_data_home)
```

The MNIST database contains a total of 70000 examples of handwritten digits of size 28x28 pixels, labeled from 0 to 9:

```
>>> mnist.data.shape
(70000, 784)
>>> mnist.target.shape
(70000,)
>>> np.unique(mnist.target)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

After the first download, the dataset is cached locally in the path specified by the `data_home` keyword argument, which defaults to `~/scikit_learn_data/`:

```
>>> os.listdir(os.path.join(custom_data_home, 'mldata'))
['mnist-original.mat']
```

Data sets in mldata.org do not adhere to a strict naming or formatting convention. `sklearn.datasets.fetch_mldata` is able to make sense of the most common cases, but allows to tailor the defaults to individual datasets:

- The data arrays in mldata.org are most often shaped as `(n_features, n_samples)`. This is the opposite of the `scikit-learn` convention, so `sklearn.datasets.fetch_mldata` transposes the matrix by default. The `transpose_data` keyword controls this behavior:

```
>>> iris = fetch_mldata('iris', data_home=custom_data_home)
>>> iris.data.shape
(150, 4)
>>> iris = fetch_mldata('iris', transpose_data=False,
...                     data_home=custom_data_home)
>>> iris.data.shape
(4, 150)
```

- For datasets with multiple columns, `sklearn.datasets.fetch_mldata` tries to identify the target and data columns and rename them to `target` and `data`. This is done by looking for arrays named `label` and `data` in the dataset, and failing that by choosing the first array to be `target` and the second to be `data`. This behavior can be changed with the `target_name` and `data_name` keywords, setting them to a specific name or index number (the name and order of the columns in the datasets can be found at its mldata.org under the tab "Data":

```
>>> iris2 = fetch_mldata('datasets-UCI iris', target_name=1, data_name=0,
...                      data_home=custom_data_home)
>>> iris3 = fetch_mldata('datasets-UCI iris', target_name='class',
...                      data_name='double0', data_home=custom_data_home)
```

8.9 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.

8.9.1 Usage

`scikit-learn` provides two loaders that will automatically download, cache, parse the metadata files, decode the jpeg and convert the interesting slices into memmaped 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 memmaped 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.

8.9.2 Examples

Faces recognition example using eigenfaces and SVMs

8.10 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.

`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.

CHAPTER

NINE

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.

9.1 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 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

9.1.1 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.

9.1.2 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 statefull 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.

9.1.3 Incremental learning

Finally, for 3. we have a number of options inside scikit-learn. Although all algorithms cannot 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`
- **Regression**
  - `sklearn.linear_model.SGDRegressor`
  - `sklearn.linear_model.PassiveAggressiveRegressor`
- **Clustering**
  - `sklearn.cluster.MiniBatchKMeans`
- **Decomposition / feature Extraction**
  - `sklearn.decomposition.MiniBatchDictionaryLearning`
  - `sklearn.decomposition.IncrementalPCA`
  - `sklearn.cluster.MiniBatchKMeans`

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.

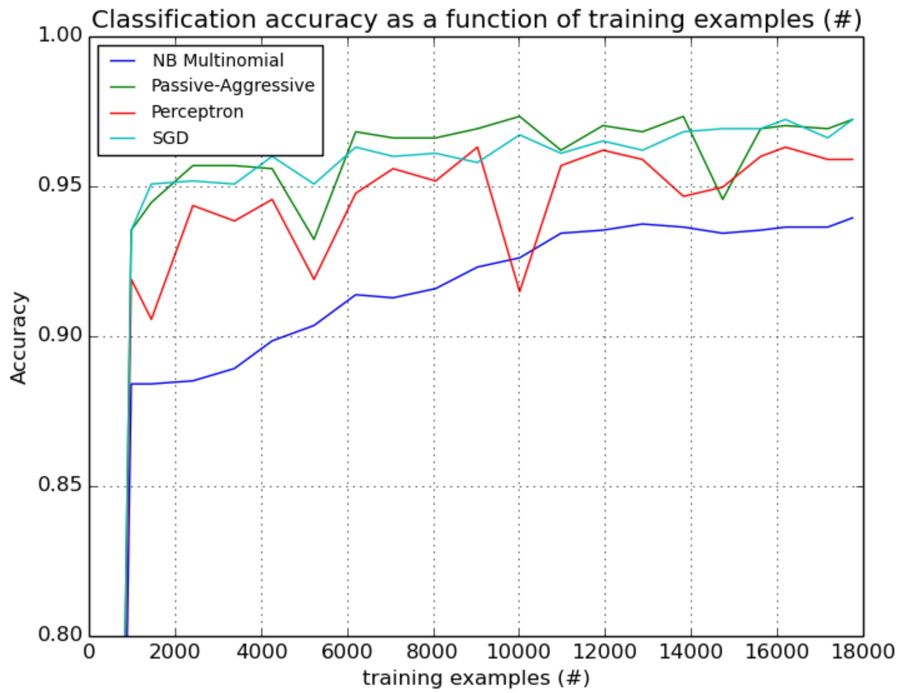
Another aspect to consider when choosing a proper algorithm is that all of them don't put the same importance on each example over time. Namely, the `Perceptron` is still sensitive to badly labeled examples even after many examples whereas the `SGD*` and `PassiveAggressive*` families are more robust to this kind of artifacts. Conversely, the later 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.

9.1.4 Examples

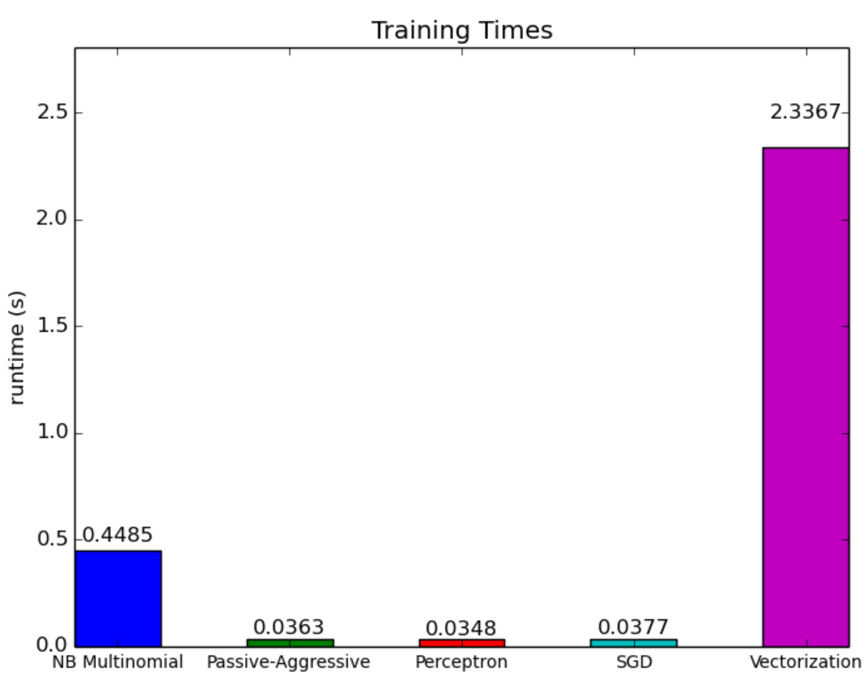
Finally, we have a full-fledged example of *Out-of-core classification of text documents*. It is aimed at providing a starting point for people wanting to build out-of-core learning systems and demonstrates most of the notions discussed above.

Furthermore, it also shows the evolution of the performance of different algorithms with the number of processed examples.

[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.



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).



9.1.5 Notes

CHAPTER

TEN

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.

10.1 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.

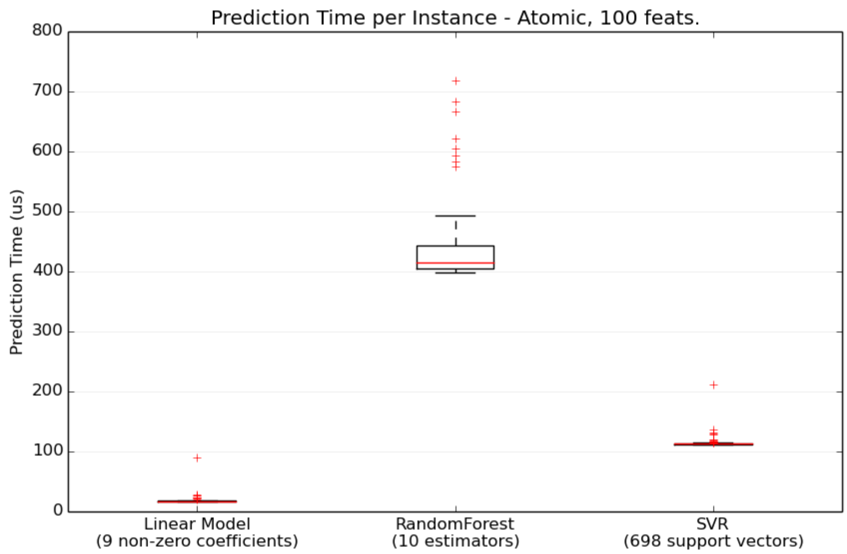
The main factors that influence the prediction latency are

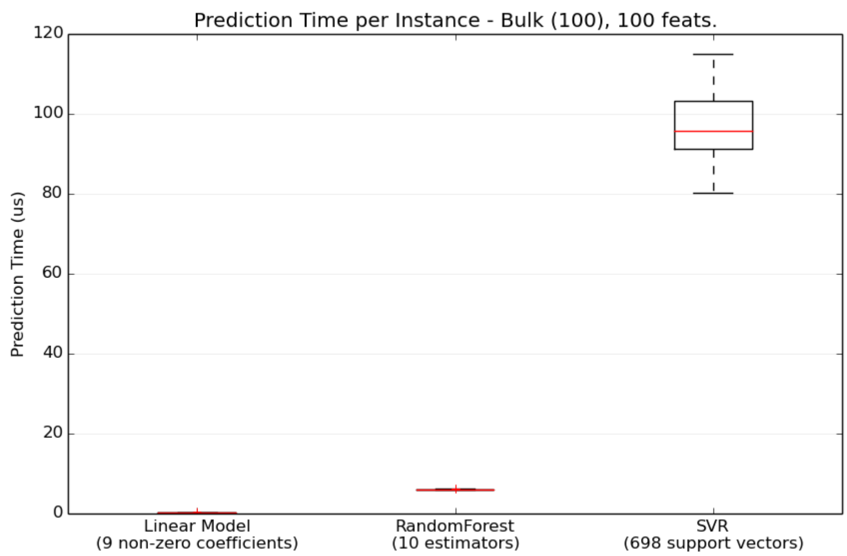
1. Number of features
2. Input data representation and sparsity
3. Model complexity
4. Feature extraction

A last major parameter is also the possibility to do predictions in bulk or one-at-a-time mode.

10.1.1 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:

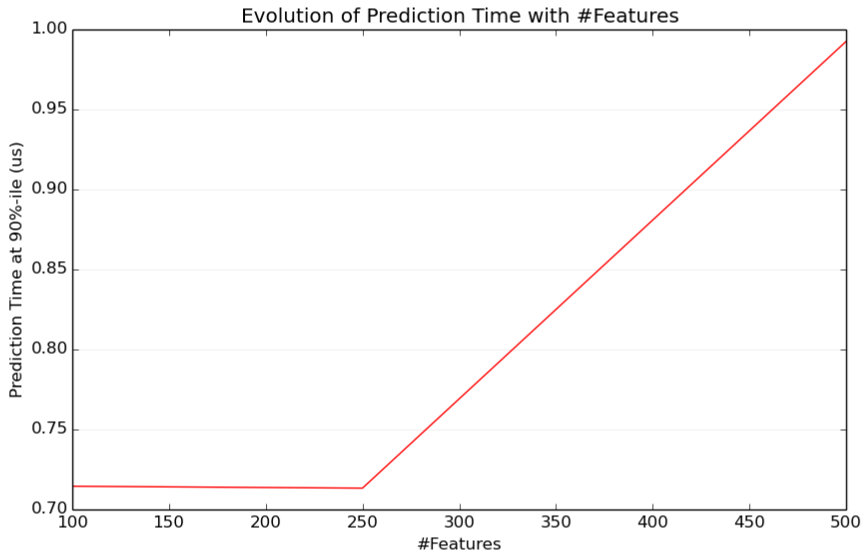




To benchmark different estimators for your case you can simply change the `n_features` parameter in this example: *Prediction Latency*. This should give you an estimate of the order of magnitude of the prediction latency.

10.1.2 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).

10.1.3 Influence of the Input Data Representation

Scipy provides sparse matrix datastructures 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))
```

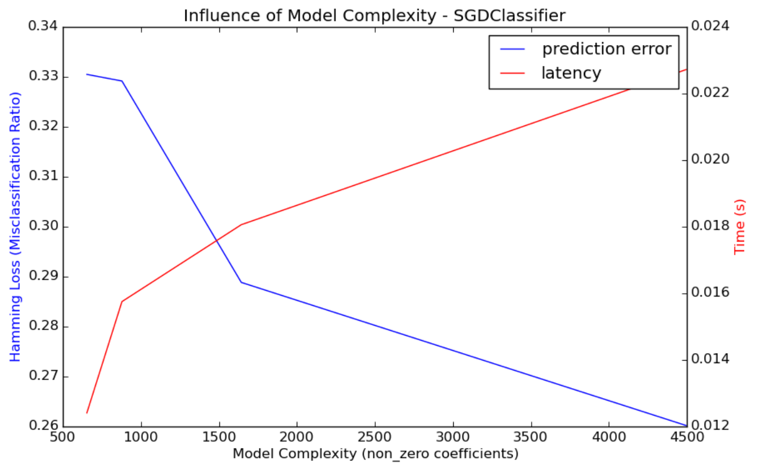
As a rule of thumb you can consider that if the sparsity ratio is greater than 90% you can probably benefit from sparse formats. Check Scipy's sparse matrix formats documentation for more information on how to build (or convert your data to) sparse matrix formats. Most of the time the `CSR` and `CSC` formats work best.

10.1.4 Influence of the Model Complexity

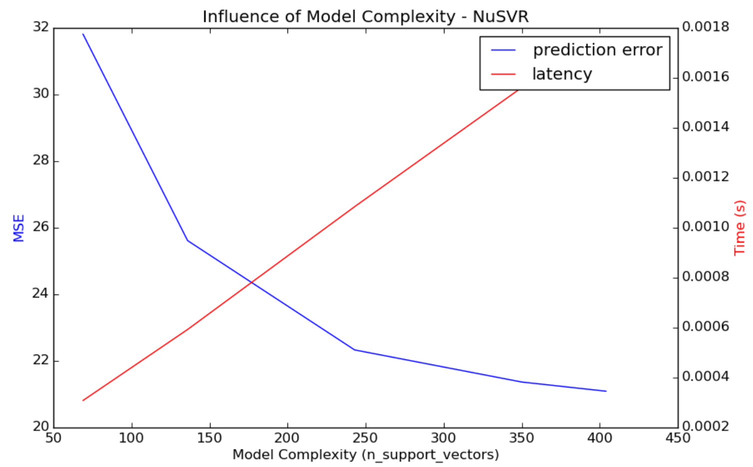
Generally speaking, when model complexity increases, predictive power and latency are supposed to increase. Increasing predictive power is usually interesting, but for many applications we would better not increase prediction latency too much. We will now review this idea for different families of supervised models.

For `sklearn.linear_model` (e.g. Lasso, ElasticNet, SGDClassifier/Regressor, Ridge & RidgeClassifier, PassiveAgressiveClassifier/Regressor, LinearSVC, LogisticRegression...) the decision function that is applied at prediction time is the same (a dot product) , so latency should be equivalent.

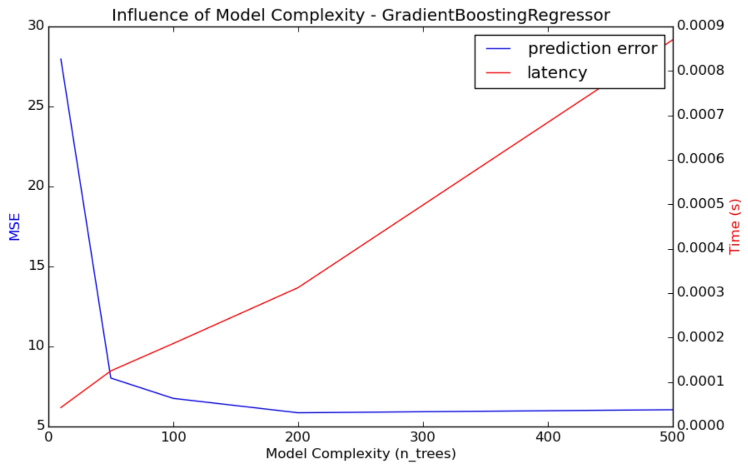
Here is an example using `sklearn.linear_model.stochastic_gradient.SGDClassifier` with the `elasticnet` penalty. The regularization strength is globally controlled by the `alpha` parameter. With a sufficiently high `alpha`, one can then increase the `l1_ratio` parameter of `elasticnet` to enforce various 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.classes.NuSVR` was used to influence the number of support vectors.



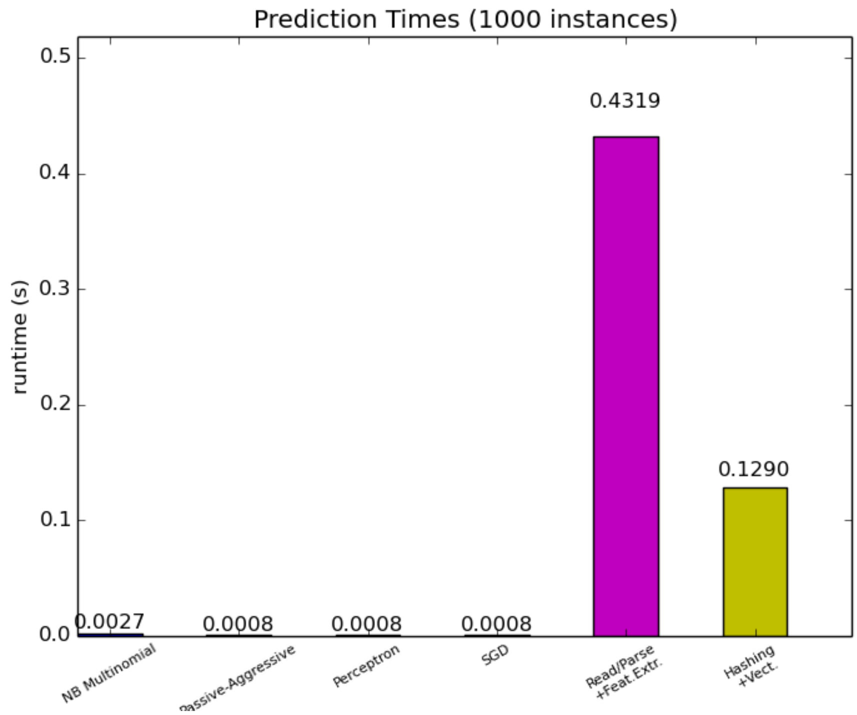
For `sklearn.ensemble` of trees (e.g. RandomForest, GBT, ExtraTrees etc) the number of trees and their depth play the most important role. Latency and throughput should scale linearly with the number of trees. In this case we used directly the `n_estimators` parameter of `sklearn.ensemble.gradient_boosting.GradientBoostingRegressor`.



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.

10.1.5 Feature Extraction Latency

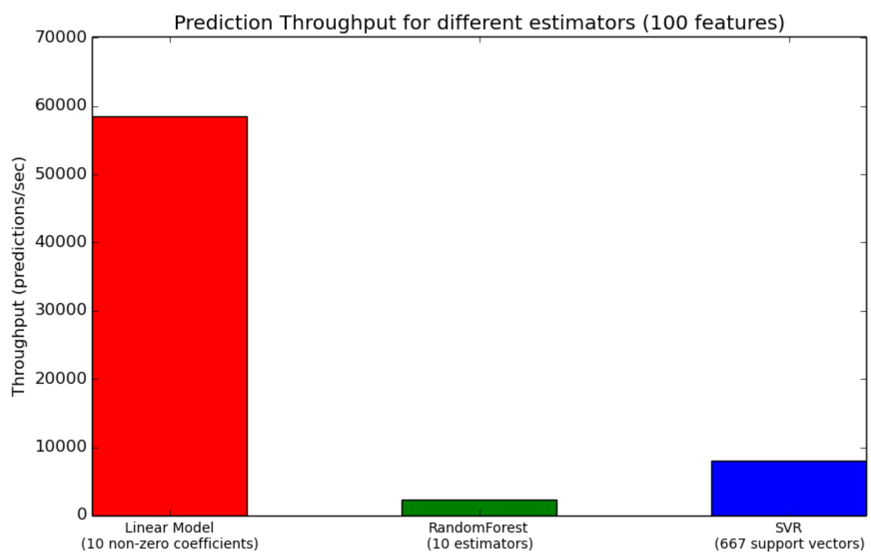
Most scikit-learn models are usually pretty fast as they are implemented either with compiled Cython extensions or optimized computing libraries. On the other hand, in many real world applications the feature extraction process (i.e. turning raw data like database rows or network packets into numpy arrays) 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.

10.2 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.

10.3 Tips and Tricks

10.3.1 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.

> **Warning:** Multithreaded BLAS libraries sometimes conflict with Python's `multiprocessing` module, which is used by e.g. `GridSearchCV` and most other estimators that take an `n_jobs` argument (with the exception of `SGDClassifier`, `SGDRegressor`, `Perceptron`, `PassiveAggressiveClassifier` and tree-based methods such as random forests). This is true of Apple's Accelerate and OpenBLAS when built with OpenMP support.
> Besides scikit-learn, NumPy and SciPy also use BLAS internally, as explained earlier.
> If you experience hanging subprocesses with `n_jobs>1` or `n_jobs=-1`, make sure you have a single-threaded BLAS library, or set `n_jobs=1`, or upgrade to Python 3.4 which has a new version of `multiprocessing` that should be immune to this problem.

10.3.2 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.

10.3.3 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.

10.3.4 Links

- scikit-learn developer performance documentation
- Scipy sparse matrix formats documentation

CHAPTER

ELEVEN

EXAMPLES

11.1 General examples

General-purpose and introductory examples for the scikit.

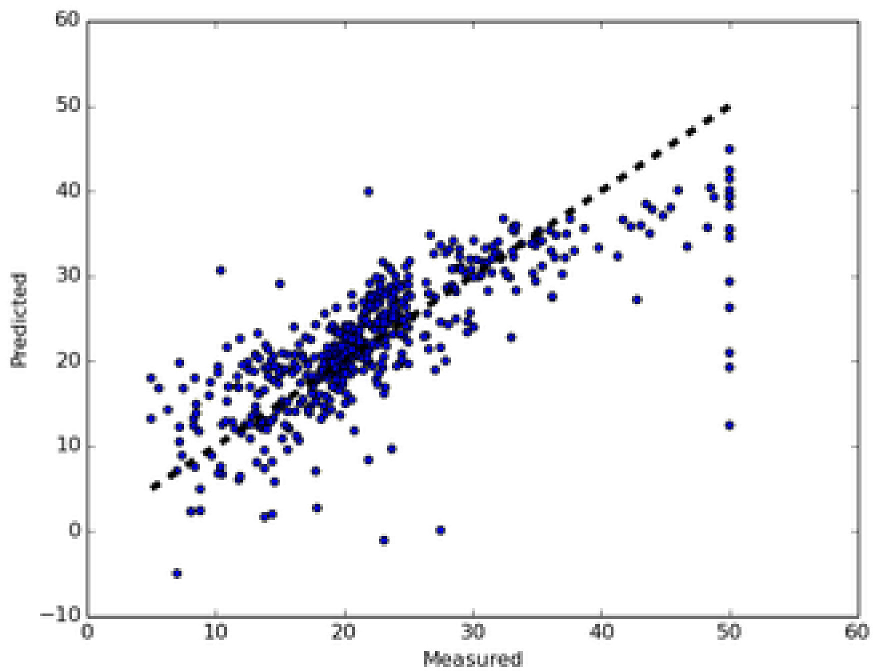
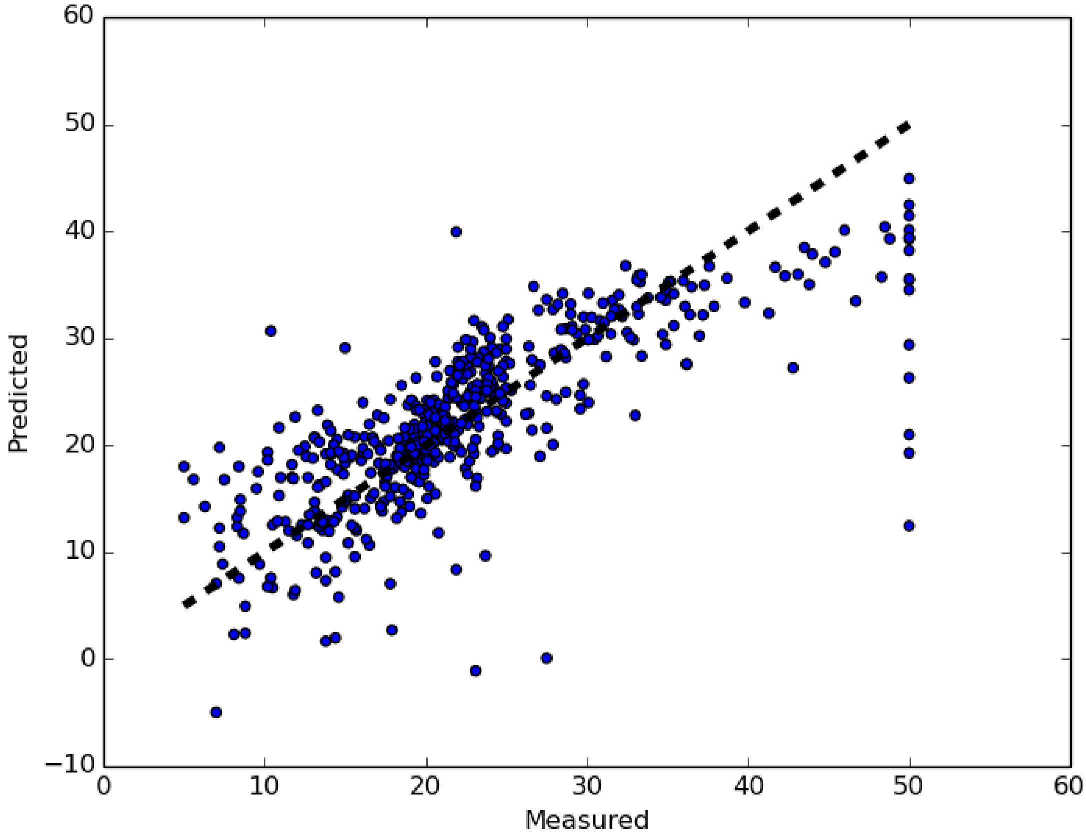


Figure 11.1: *Plotting Cross-Validated Predictions*

11.1.1 Plotting Cross-Validated Predictions

This example shows how to use *cross_val_predict* to visualize prediction errors.



Python source code: `plot_cv_predict.py`

```
from sklearn import datasets
from sklearn.cross_validation import cross_val_predict
from sklearn import linear_model
import matplotlib.pyplot as plt

lr = linear_model.LinearRegression()
boston = datasets.load_boston()
y = boston.target

# cross_val_predict returns an array of the same size as `y` where each entry
# is a prediction obtained by cross validated:
predicted = cross_val_predict(lr, boston.data, y, cv=10)

fig,ax = plt.subplots()
ax.scatter(y, predicted)
ax.plot([y.min(), y.max()], [y.min(), y.max()], 'k--', lw=4)
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
fig.show()
```

Total running time of the example: 0.42 seconds (0 minutes 0.42 seconds)

Figure 11.2: *Concatenating multiple feature extraction methods*

11.1.2 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.

Python source code: `feature_stacker.py`

```
# Author: Andreas Mueller <amueller@ais.uni-bonn.de>
#
# License: BSD 3 clause

from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.grid_search import GridSearchCV
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 to 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)

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_)
```

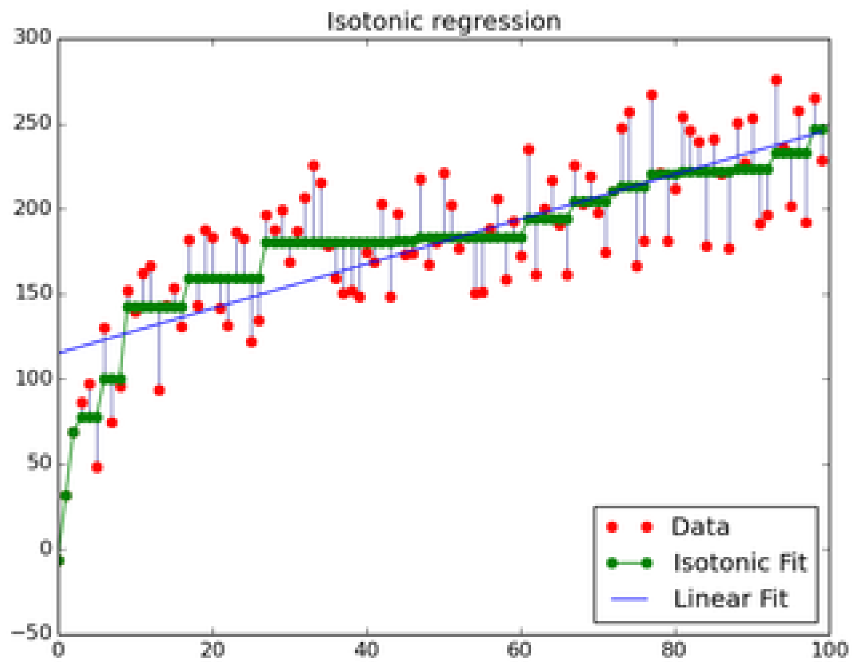
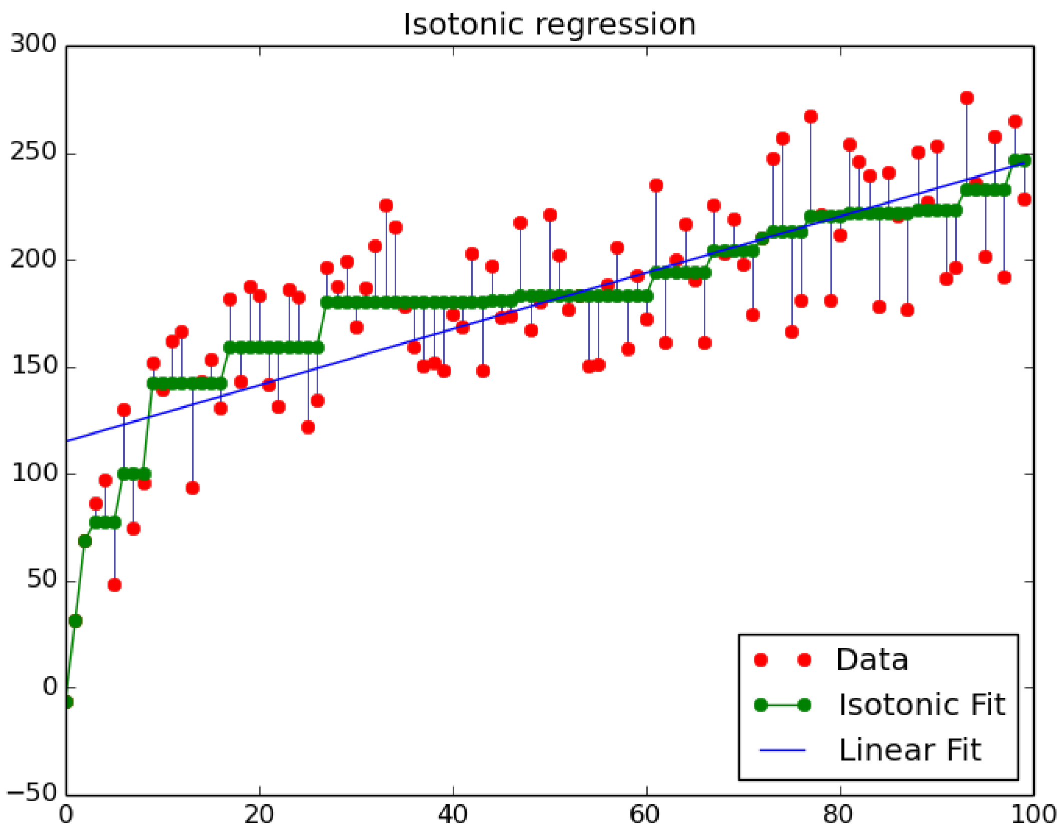


Figure 11.3: *Isotonic Regression*

11.1.3 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.



Python source code: `plot_isotonic_regression.py`

```
print(__doc__)

# Author: Nelle Varoquaux <nelle.varoquaux@gmail.com>
#         Alexandre Gramfort <alexandre.gramfort@inria.fr>
# Licence: 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.log(1 + np.arange(n))

###############################################################################
# Fit IsotonicRegression and LinearRegression models

ir = IsotonicRegression()
```

```
y_ = ir.fit_transform(x, y)

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(0.5 * np.ones(n))

fig = plt.figure()
plt.plot(x, y, 'r.', markersize=12)
plt.plot(x, y_, 'g.-', 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 example: 0.08 seconds (0 minutes 0.08 seconds)

Figure 11.4: *Imputing missing values before building an estimator*

11.1.4 Imputing missing values before building an estimator

This example shows that imputing the missing values can give better results than discarding the samples containing any missing value. Imputing does not always improve the predictions, so please check via cross-validation. Sometimes dropping rows or using marker values is more effective.

Missing values can be replaced by the mean, the median or the most frequent value using the `strategy` hyper-parameter. The median is a more robust estimator for data with high magnitude variables which could dominate results (otherwise known as a 'long tail').

Script output:

```
Score with the entire dataset = 0.56
Score without the samples containing missing values = 0.48
Score after imputation of the missing values = 0.55
```

In this case, imputing helps the classifier get close to the original score.

Python source code: `missing_values.py`

```
import numpy as np

from sklearn.datasets import load_boston
from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import Imputer
from sklearn.cross_validation import cross_val_score

rng = np.random.RandomState(0)

dataset = load_boston()
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
estimator = RandomForestRegressor(random_state=0, n_estimators=100)
score = cross_val_score(estimator, X_full, y_full).mean()
print("Score with the entire dataset = %.2f" % score)

# Add missing values in 75% of the lines
missing_rate = 0.75
n_missing_samples = 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)

# Estimate the score without the lines containing missing values
X_filtered = X_full[~missing_samples, :]
y_filtered = y_full[~missing_samples]
estimator = RandomForestRegressor(random_state=0, n_estimators=100)
score = cross_val_score(estimator, X_filtered, y_filtered).mean()
print("Score without the samples containing missing values = %.2f" % score)

# Estimate the score after imputation of the missing values
X_missing = X_full.copy()
X_missing[np.where(missing_samples)[0], missing_features] = 0
y_missing = y_full.copy()
estimator = Pipeline([("imputer", Imputer(missing_values=0,
                                          strategy="mean",
                                          axis=0)),
                      ("forest", RandomForestRegressor(random_state=0,
                                                       n_estimators=100))])
score = cross_val_score(estimator, X_missing, y_missing).mean()
print("Score after imputation of the missing values = %.2f" % score)
```

11.1.5 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

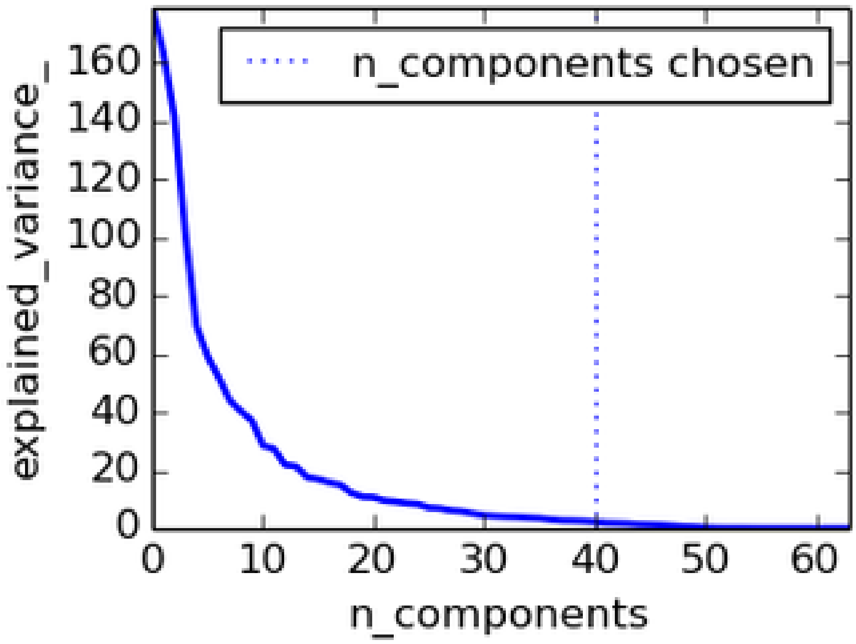
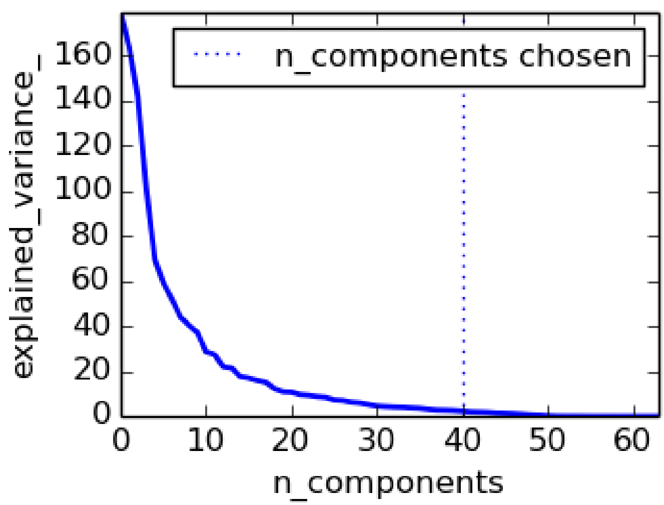


Figure 11.5: *Pipelining: chaining a PCA and a logistic regression*



Python source code: `plot_digits_pipe.py`

```
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, decomposition, datasets
from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV

logistic = linear_model.LogisticRegression()

pca = decomposition.PCA()
pipe = Pipeline(steps=[('pca', pca), ('logistic', logistic)])

digits = datasets.load_digits()
X_digits = digits.data
y_digits = digits.target

###############################################################################
# Plot the PCA spectrum
pca.fit(X_digits)

plt.figure(1, figsize=(4, 3))
plt.clf()
plt.axes([.2, .2, .7, .7])
plt.plot(pca.explained_variance_, linewidth=2)
plt.axis('tight')
plt.xlabel('n_components')
plt.ylabel('explained_variance_')

###############################################################################
# Prediction

n_components = [20, 40, 64]
Cs = np.logspace(-4, 4, 3)

#Parameters of pipelines can be set using '__' separated parameter names:

estimator = GridSearchCV(pipe,
                         dict(pca__n_components=n_components,
                              logistic__C=Cs))
estimator.fit(X_digits, y_digits)

plt.axvline(estimator.best_estimator_.named_steps['pca'].n_components,
            linestyle=':', label='n_components chosen')
plt.legend(prop=dict(size=12))
plt.show()
```

Total running time of the example: 6.09 seconds (0 minutes 6.09 seconds)

11.1.6 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)

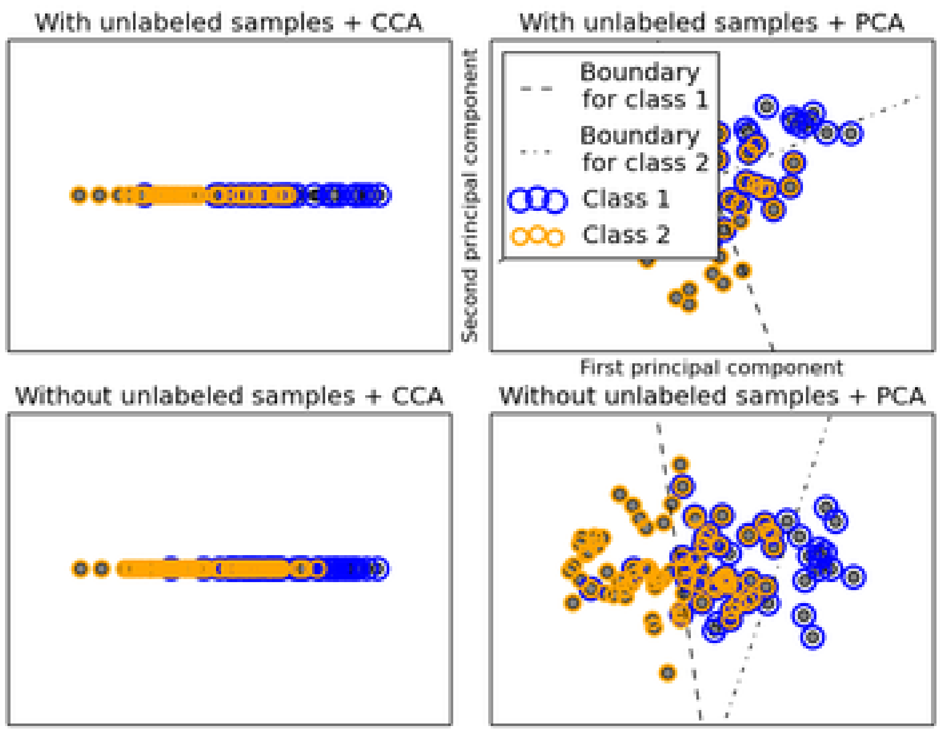


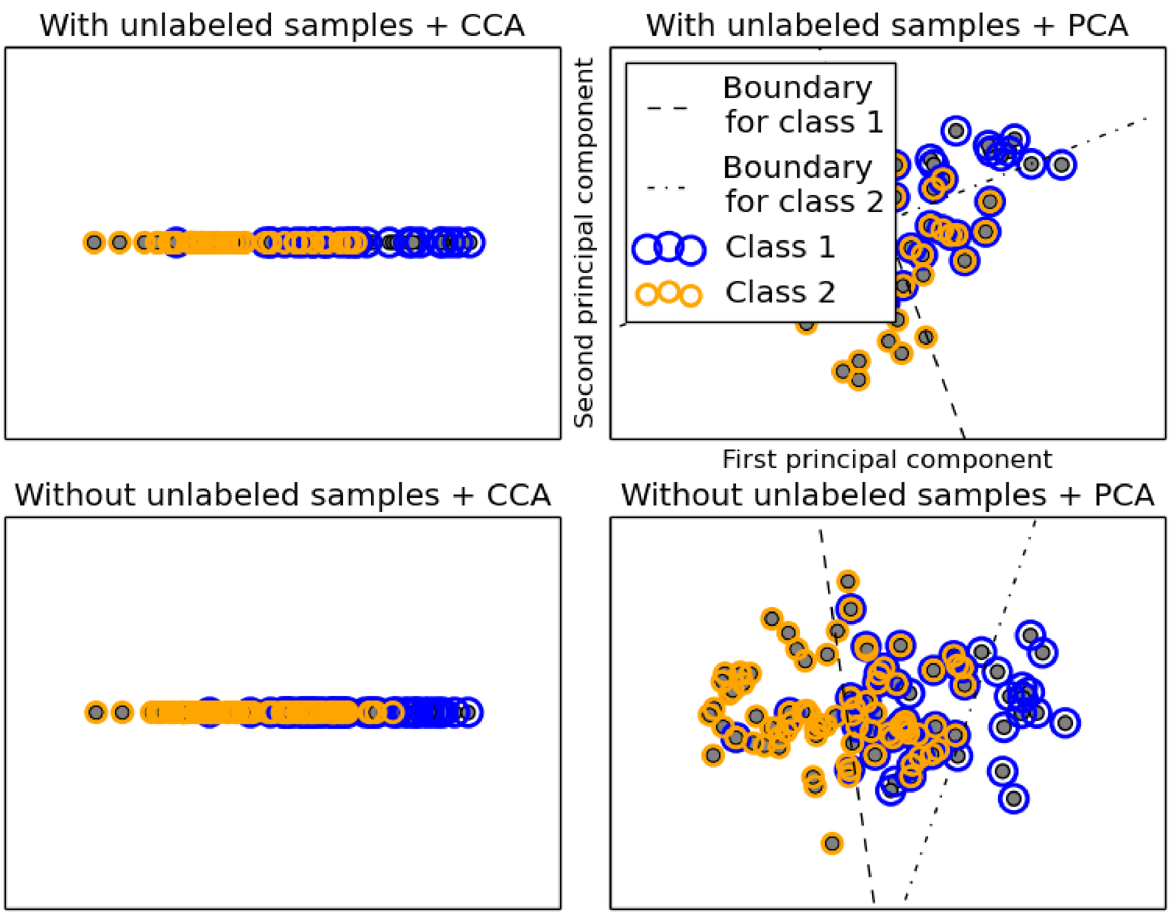
Figure 11.6: *Multilabel classification*

- 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.



Python source code: `plot_multilabel.py`

```
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.preprocessing import LabelBinarizer
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":
        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')
    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,
                                      return_indicator=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,
                                      return_indicator=True,
                                      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 example: 0.21 seconds (0 minutes 0.21 seconds)

Figure 11.7: *Face completion with a multi-output estimators*

11.1.7 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.

Python source code: `plot_multioutput_face_completion.py`

```
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
```

```python
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import RidgeCV

# Load the faces datasets
data = fetch_olivetti_faces()
targets = data.target

data = data.images.reshape((len(data.images), -1))
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]
X_train = train[:, :np.ceil(0.5 * n_pixels)]  # Upper half of the faces
y_train = train[:, np.floor(0.5 * n_pixels):]  # Lower half of the faces
X_test = test[:, :np.ceil(0.5 * n_pixels)]
y_test = test[:, np.floor(0.5 * n_pixels):]

# 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,
                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 example: 18.02 seconds (0 minutes 18.02 seconds)

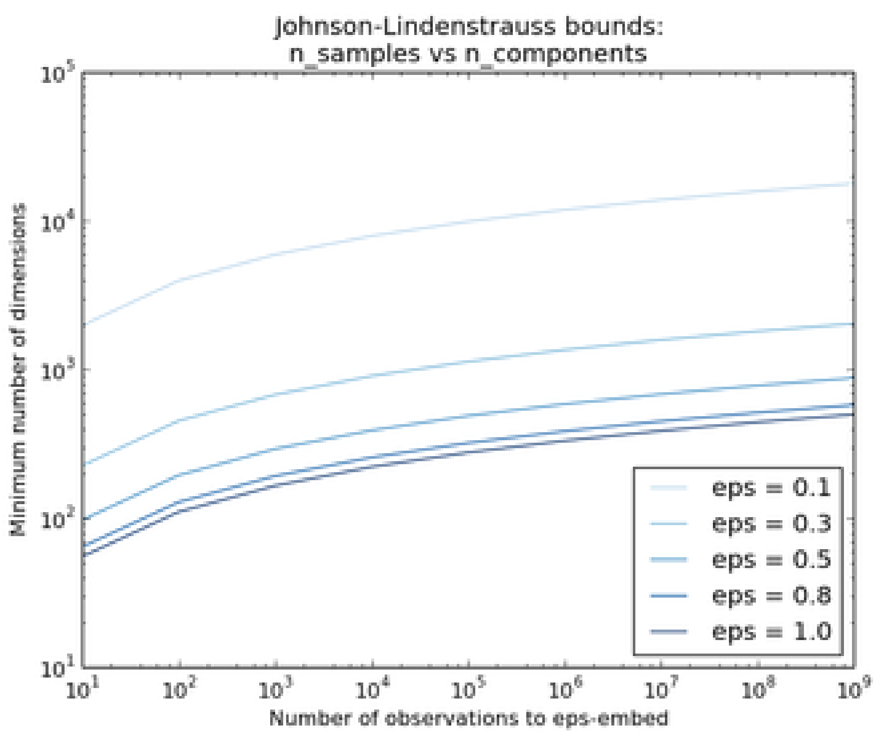


Figure 11.8: *The Johnson-Lindenstrauss bound for embedding with random projections*

11.1.8 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.

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 >= 4 log(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.

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`

Empirical validation

We validate the above bounds on the the digits dataset or on the 20 newsgroups text document (TF-IDF word frequencies) dataset:

- 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`.
- 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`.

The default dataset is the digits dataset. To run the example on the twenty newsgroups dataset, pass the –twenty-newsgroups command line argument to this script.

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).

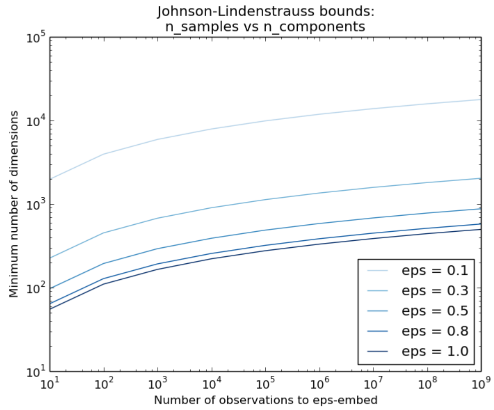
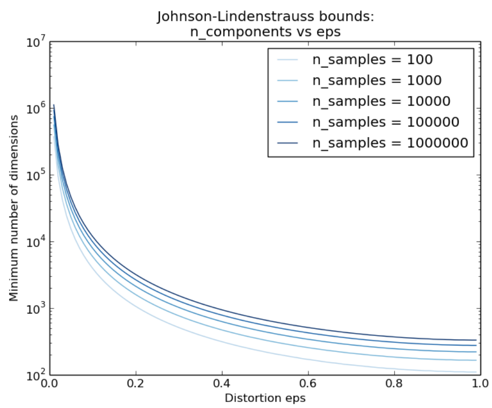
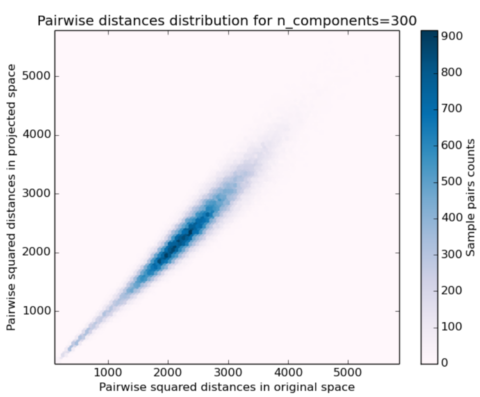
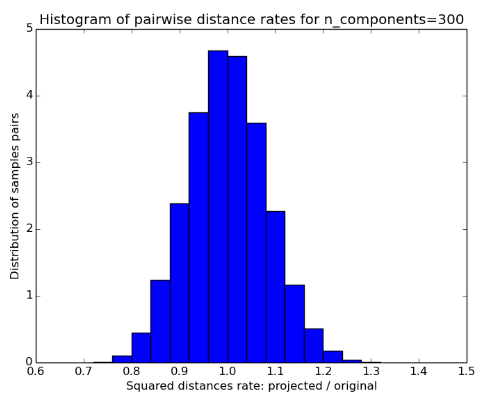
We can see that for low values of `n_components` the distribution is wide with many distorted pairs and a skewed distribution (due to the hard limit of zero ratio on the left as distances are always positives) while for larger values of n_components the distortion is controlled and the distances are well preserved by the random projection.

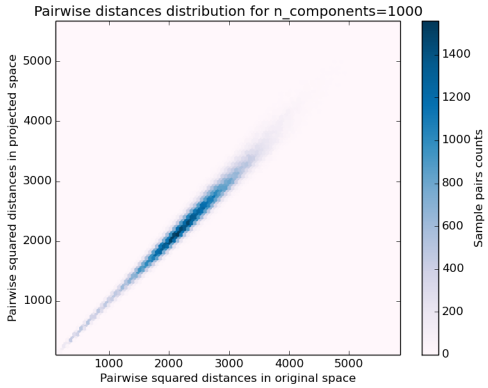
Remarks

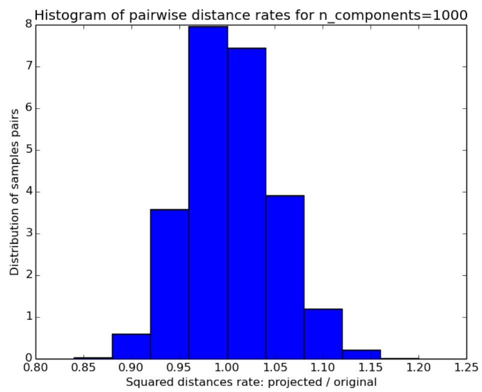
According to the JL lemma, projecting 500 samples without too much distortion will require at least several thousands dimensions, irrespective of the number of features of the original dataset.

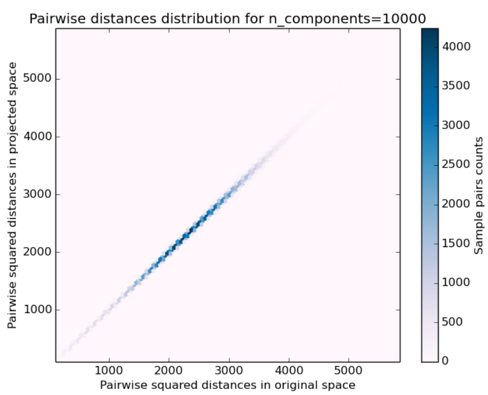
Hence using random projections on the digits dataset which only has 64 features in the input space does not make sense: it does not allow for dimensionality reduction in this case.

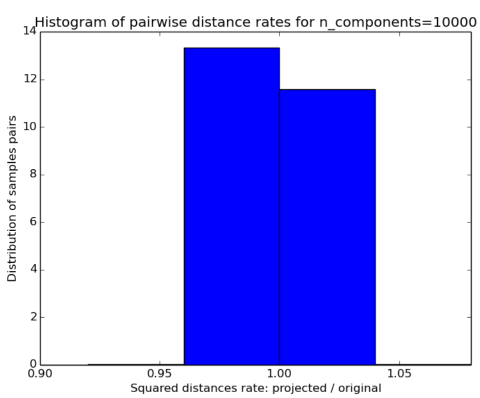
On the twenty newsgroups on the other hand the dimensionality can be decreased from 56436 down to 10000 while reasonably preserving pairwise distances.

- 
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
- 
Pairwise distances distribution for n_components=300
Pairwise squared distances in projected space
Pairwise squared distances in original space
Sample pairs counts
- 
Histogram of pairwise distance rates for n_components=300
Distribution of samples pairs
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

- 
Histogram of pairwise distance rates for n_components=1000
Distribution of samples pairs
Squared distances rate: projected / original

- 
Pairwise distances distribution for n_components=10000
Pairwise squared distances in projected space
Pairwise squared distances in original space
Sample pairs counts

- 
Histogram of pairwise distance rates for n_components=10000
Distribution of samples pairs
Squared distances rate: projected / original

Script output:

```
Embedding 500 samples with dim 64 using various random projections
Projected 500 samples from 64 to 300 in 0.007s
Random matrix with size: 0.028MB
Mean distances rate: 1.00 (0.08)
Projected 500 samples from 64 to 1000 in 0.010s
Random matrix with size: 0.096MB
Mean distances rate: 1.00 (0.05)
Projected 500 samples from 64 to 10000 in 0.103s
Random matrix with size: 0.957MB
Mean distances rate: 1.00 (0.01)
```

Python source code: plot_johnson_lindenstrauss_bound.py

```
print(__doc__)

import sys
from time import time
import numpy as np
import matplotlib.pyplot as plt
from sklearn.random_projection import johnson_lindenstrauss_min_dim
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

# Part 1: plot the theoretical dependency between n_components_min and
# n_samples

# 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")

# 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")

# Part 2: perform sparse random projection of some digits images which are
# quite low dimensional and dense or documents of the 20 newsgroups dataset
# which is both high dimensional and sparse

if '--twenty-newsgroups' in sys.argv:
    # Need an internet connection hence not enabled by default
    data = fetch_20newsgroups_vectorized().data[:500]
else:
    data = load_digits().data[:500]

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()
    plt.hexbin(dists, projected_dists, gridsize=100, cmap=plt.cm.PuBu)
    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, normed=True, range=(0., 2.))
    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()
```

Total running time of the example: 11.56 seconds (0 minutes 11.56 seconds)

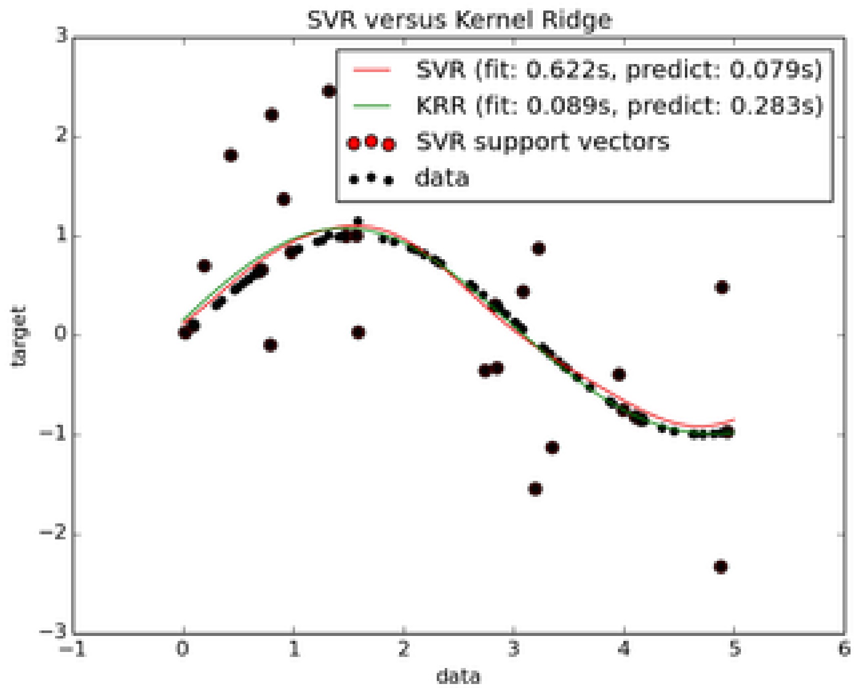


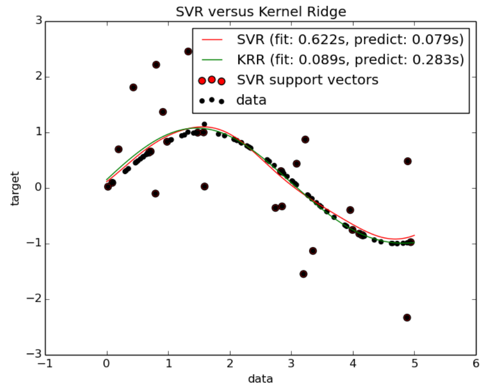
Figure 11.9: *Comparison of kernel ridge regression and SVR*

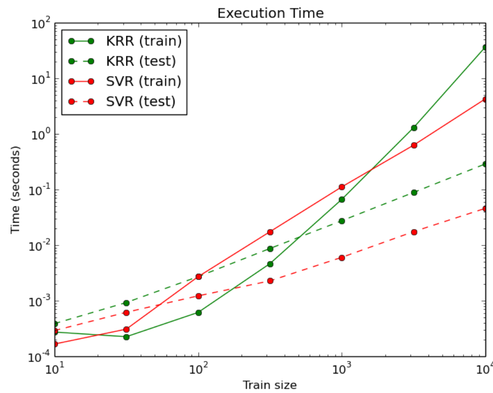
11.1.9 Comparison of kernel ridge regression and SVR

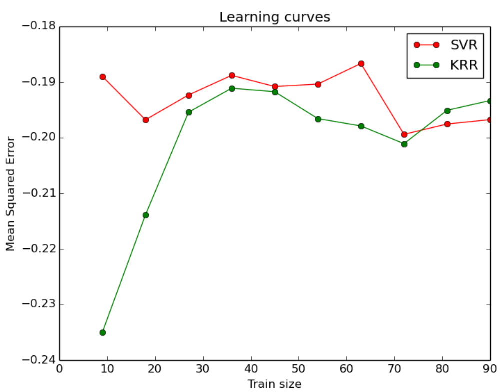
Both kernel ridge regression (KRR) and SVR learn a non-linear function by employing the kernel trick, i.e., they learn a linear function in the space induced by the respective kernel which corresponds to a non-linear function in the original space. They differ in the loss functions (ridge versus epsilon-insensitive 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.

- 

- 

- 

Script output:

```
SVR complexity and bandwidth selected and model fitted in 0.622 s
KRR complexity and bandwidth selected and model fitted in 0.089 s
Support vector ratio: 0.320
SVR prediction for 100000 inputs in 0.079 s
KRR prediction for 100000 inputs in 0.283 s
```

Python source code: `plot_kernel_ridge_regression.py`

```
# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD 3 clause


from __future__ import division
import time

import numpy as np

from sklearn.svm import SVR
from sklearn.grid_search import GridSearchCV
from sklearn.learning_curve import learning_curve
from sklearn.kernel_ridge import KernelRidge
import matplotlib.pyplot as plt

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), cv=5,
                   param_grid={"C": [1e0, 1e1, 1e2, 1e3],
                               "gamma": np.logspace(-2, 2, 5)})

kr = GridSearchCV(KernelRidge(kernel='rbf', gamma=0.1), cv=5,
                  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')
plt.scatter(X[:100], y[:100], c='k', label='data')
plt.hold('on')
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)
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="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="mean_squared_error", cv=10)

plt.plot(train_sizes, test_scores_svr.mean(1), 'o-', color="r",
         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 example: 44.97 seconds (0 minutes 44.97 seconds)

Figure 11.10: *Feature Union with Heterogeneous Data Sources*

11.1.10 Feature Union 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.feature_extraction.FeatureUnion` 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 FeatureUnion 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.

Python source code: `hetero_feature_union.py`

```python
# Author: Matt Terry <matt.terry@gmail.com>
#
# License: BSD 3 clause
from __future__ import print_function

import numpy as np

from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.datasets import fetch_20newsgroups
from sklearn.datasets.twenty_newsgroups import strip_newsgroup_footer
from sklearn.datasets.twenty_newsgroups import strip_newsgroup_quoting
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 FeatureUnion
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC


class ItemSelector(BaseEstimator, TransformerMixin):
    """For data grouped by feature, select subset of data at a provided key.

    The data is expected to be stored in a 2D data structure, where the first
    index is over features and the second is over samples.  i.e.

    >> len(data[key]) == n_samples

    Please note that this is the opposite convention to sklearn feature
    matrixes (where the first index corresponds to sample).

    ItemSelector only requires that the collection implement getitem
    (data[key]).  Examples include: a dict of lists, 2D numpy array, Pandas
    DataFrame, numpy record array, etc.

    >> data = {'a': [1, 5, 2, 5, 2, 8],
               'b': [9, 4, 1, 4, 1, 3]}
    >> ds = ItemSelector(key='a')
    >> data['a'] == ds.transform(data)

    ItemSelector is not designed to handle data grouped by sample.  (e.g. a
    list of dicts).  If your data is structured this way, consider a
    transformer along the lines of `sklearn.feature_extraction.DictVectorizer`.

    Parameters
    ----------
    key : hashable, required
        The key corresponding to the desired value in a mappable.
    """
    def __init__(self, key):
        self.key = key

    def fit(self, x, y=None):
        return self

    def transform(self, data_dict):
        return data_dict[self.key]
```

```python
class TextStats(BaseEstimator, TransformerMixin):
    """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(BaseEstimator, TransformerMixin):
    """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):
        features = np.recarray(shape=(len(posts),),
                               dtype=[('subject', object), ('body', object)])
        for i, text in enumerate(posts):
            headers, _, bod = text.partition('\n\n')
            bod = strip_newsgroup_footer(bod)
            bod = strip_newsgroup_quoting(bod)
            features['body'][i] = bod

            prefix = 'Subject:'
            sub = ''
            for line in headers.split('\n'):
                if line.startswith(prefix):
                    sub = line[len(prefix):]
                    break
            features['subject'][i] = sub

        return features


pipeline = Pipeline([
    # Extract the subject & body
    ('subjectbody', SubjectBodyExtractor()),

    # Use FeatureUnion to combine the features from subject and body
    ('union', FeatureUnion(
        transformer_list=[

            # Pipeline for pulling features from the post's subject line
            ('subject', Pipeline([
                ('selector', ItemSelector(key='subject')),
                ('tfidf', TfidfVectorizer(min_df=50)),
            ])),

            # Pipeline for standard bag-of-words model for body
            ('body_bow', Pipeline([
                ('selector', ItemSelector(key='body')),
```

```
                ('tfidf', TfidfVectorizer()),
                ('best', TruncatedSVD(n_components=50)),
            ])),

            # Pipeline for pulling ad hoc features from post's body
            ('body_stats', Pipeline([
                ('selector', ItemSelector(key='body')),
                ('stats', TextStats()),  # returns a list of dicts
                ('vect', DictVectorizer()),  # list of dicts -> feature matrix
            ])),

        ],

        # weight components in FeatureUnion
        transformer_weights={
            'subject': 0.8,
            'body_bow': 0.5,
            'body_stats': 1.0,
        },
    )),

    # Use a SVC classifier on the combined features
    ('svc', SVC(kernel='linear')),
])

# limit the list of categories to make running this exmaple faster.
categories = ['alt.atheism', 'talk.religion.misc']
train = fetch_20newsgroups(random_state=1,
                           subset='train',
                           categories=categories,
                           )
test = fetch_20newsgroups(random_state=1,
                          subset='test',
                          categories=categories,
                          )

pipeline.fit(train.data, train.target)
y = pipeline.predict(test.data)
print(classification_report(y, test.target))
```

11.1.11 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.

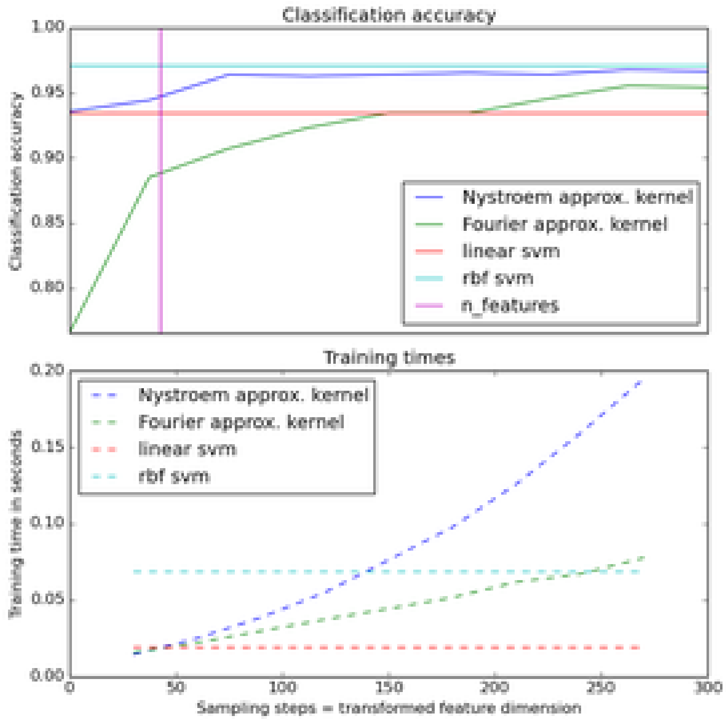
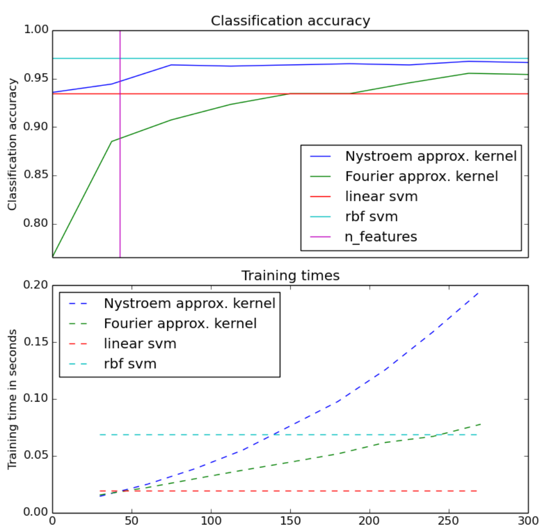
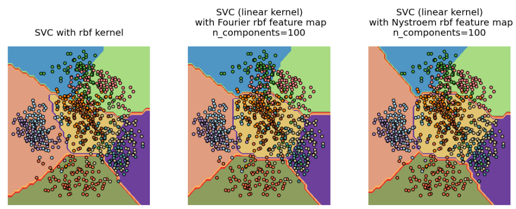


Figure 11.11: *Explicit feature map approximation for RBF kernels*

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*.

- 

- 

Python source code: `plot_kernel_approximation.py`

```
print(__doc__)

# Author: Gael Varoquaux <gael dot varoquaux at normalesup dot org>
#         Andreas Mueller <amueller@ais.uni-bonn.de>
# License: BSD 3 clause

# 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

# The digits dataset
digits = datasets.load_digits(n_class=9)

# 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())])
```

```python
# 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)
    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=(8, 8))
accuracy = plt.subplot(211)
# second y axis for timeings
timescale = plt.subplot(212)

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')
```

```python
# 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')

# 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)

# Gemerate 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.tight_layout()
plt.figure(figsize=(12, 5))

# 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, m_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)
    plt.axis('off')

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=targets_train, cmap=plt.cm.Paired)

    plt.title(titles[i])
```

```
plt.tight_layout()
plt.show()
```

Total running time of the example: 2.30 seconds (0 minutes 2.30 seconds)

11.2 Examples based on real world datasets

Applications to real world problems with some medium sized datasets or interactive user interface.

Figure 11.12: *Topics extraction with Non-Negative Matrix Factorization*

11.2.1 Topics extraction with Non-Negative Matrix Factorization

This is a proof of concept application of Non Negative Matrix Factorization of the term frequency matrix of a corpus of documents so as to extract an additive model 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).

The default parameters (n_samples / n_features / n_topics) 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 than the time complexity is polynomial.

Python source code: `topics_extraction_with_nmf.py`

```
# Author: Olivier Grisel <olivier.grisel@ensta.org>
#         Lars Buitinck <L.J.Buitinck@uva.nl>
# License: BSD 3 clause

from __future__ import print_function
from time import time

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import NMF
from sklearn.datasets import fetch_20newsgroups

n_samples = 2000
n_features = 1000
n_topics = 10
n_top_words = 20

# 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.

t0 = time()
print("Loading dataset and extracting TF-IDF features...")
dataset = fetch_20newsgroups(shuffle=True, random_state=1,
                             remove=('headers', 'footers', 'quotes'))

vectorizer = TfidfVectorizer(max_df=0.95, min_df=2, max_features=n_features,
                             stop_words='english')
tfidf = vectorizer.fit_transform(dataset.data[:n_samples])
print("done in %0.3fs." % (time() - t0))

# Fit the NMF model
print("Fitting the NMF model with n_samples=%d and n_features=%d..."
      % (n_samples, n_features))
nmf = NMF(n_components=n_topics, random_state=1).fit(tfidf)
print("done in %0.3fs." % (time() - t0))

feature_names = vectorizer.get_feature_names()

for topic_idx, topic in enumerate(nmf.components_):
    print("Topic #%d:" % topic_idx)
    print(" ".join([feature_names[i]
                    for i in topic.argsort()[:-n_top_words - 1:-1]]))
    print()
```

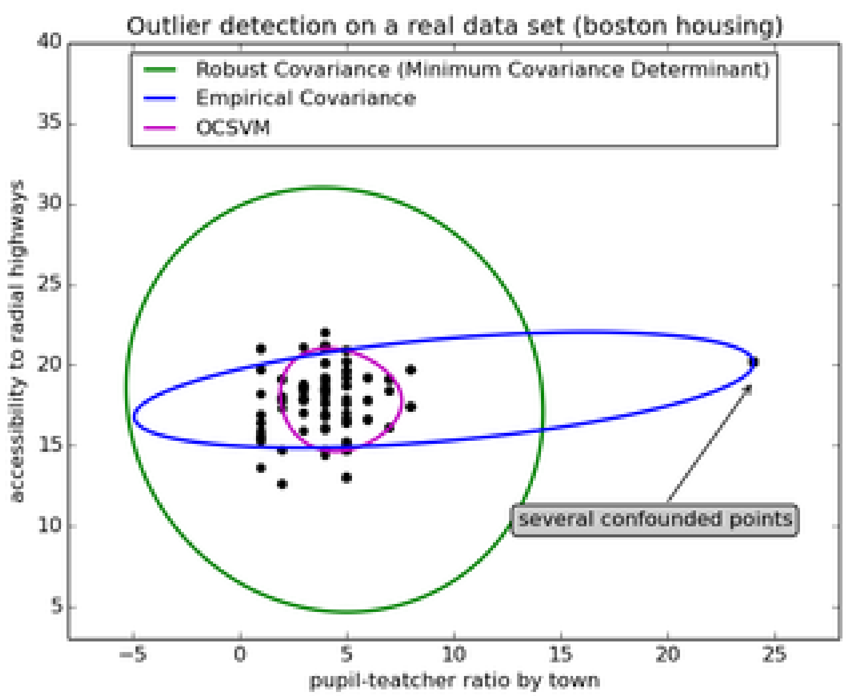


Figure 11.13: *Outlier detection on a real data set*

11.2.2 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.

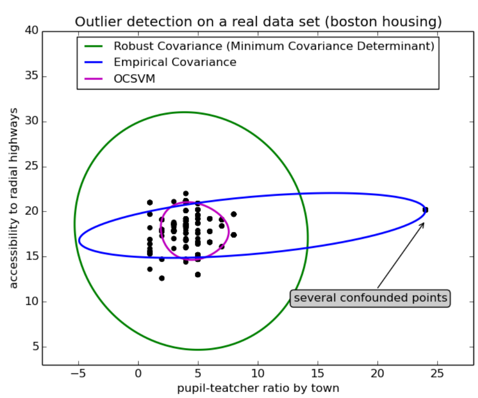
In both examples below, the main result is that the empirical covariance estimate, as a non-robust one, is highly influenced by the heterogeneous structure of the observations. Although the robust covariance estimate is able to focus on the main mode of the data distribution, it sticks to the assumption 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 algorithm

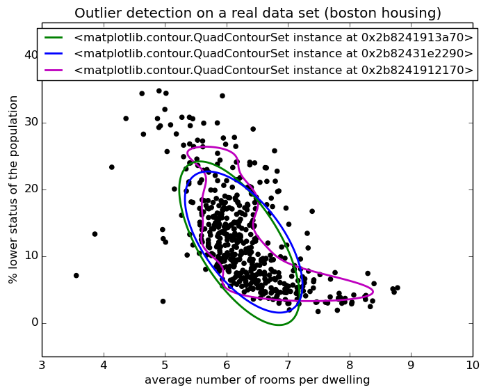
First example

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.

- 



•

Python source code: `plot_outlier_detection_housing.py`

```
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 = {}

# 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-teatcher 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_values_list[0], legend2_values_list[1], legend2_values_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")

plt.show()
```

Total running time of the example: 3.27 seconds (0 minutes 3.27 seconds)

11.2.3 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**.

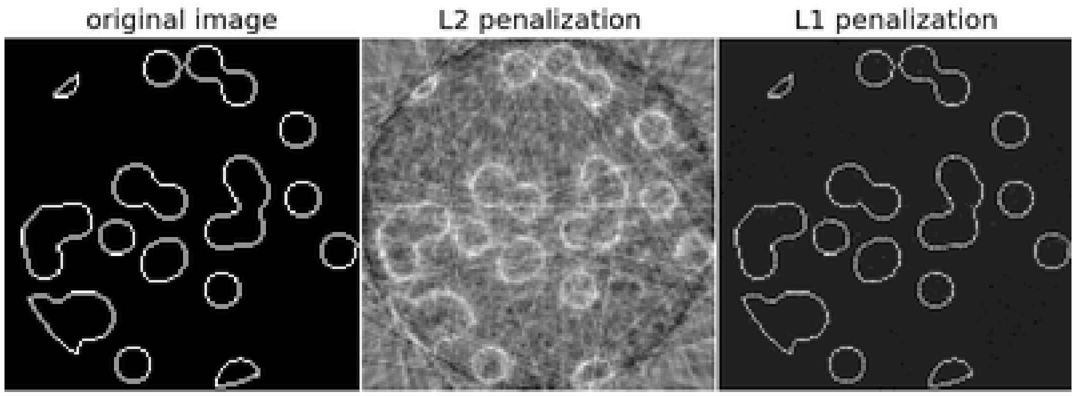
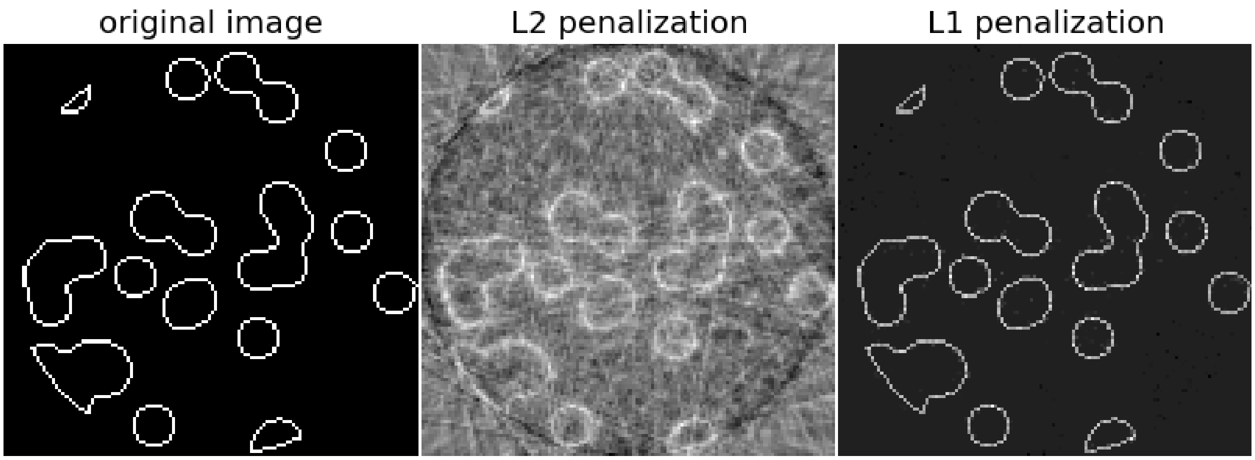


Figure 11.14: *Compressive sensing: tomography reconstruction with L1 prior (Lasso)*

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.



Python source code: `plot_tomography_l1_reconstruction.py`

```
print(__doc__)

# Author: Emmanuelle Gouillart <emmanuelle.gouillart@nsup.org>
# 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)
    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):
    l_x = float(l_x)
    X, Y = np.mgrid[:l_x, :l_x]
    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


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 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 example: 7.30 seconds (0 minutes 7.30 seconds)

Figure 11.15: *Faces recognition example using eigenfaces and SVMs*

11.2.4 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:

```
                   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
```

Python source code: `face_recognition.py`

```
from __future__ import print_function

from time import time
import logging
import matplotlib.pyplot as plt

from sklearn.cross_validation import train_test_split
from sklearn.datasets import fetch_lfw_people
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.decomposition import RandomizedPCA
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)


###############################################################################
# Compute a PCA (eigenfaces) on the face dataset (treated as unlabeled
# dataset): unsupervised feature extraction / dimensionality reduction
n_components = 150
```

```python
print("Extracting the top %d eigenfaces from %d faces"
      % (n_components, X_train.shape[0]))
t0 = time()
pca = RandomizedPCA(n_components=n_components, 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='auto'), 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)
                     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()
```

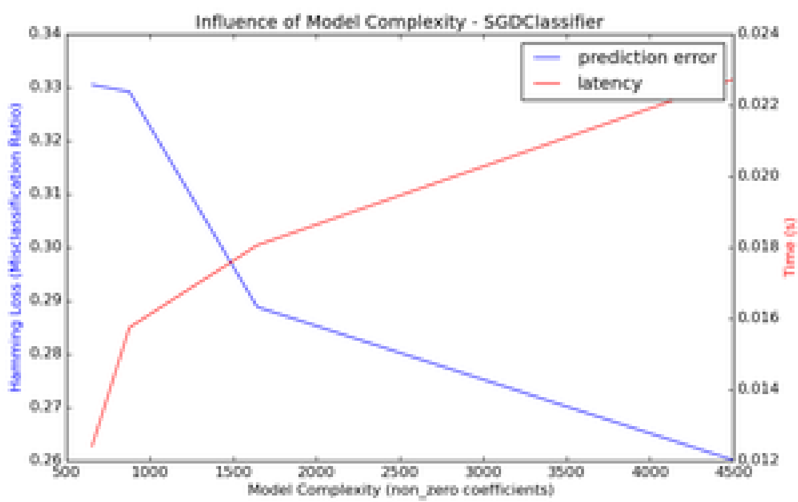


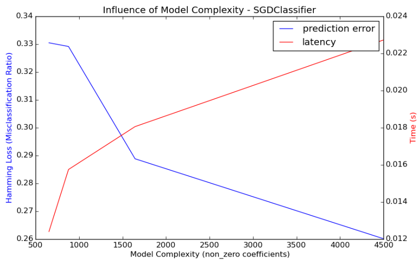
Figure 11.16: *Model Complexity Influence*

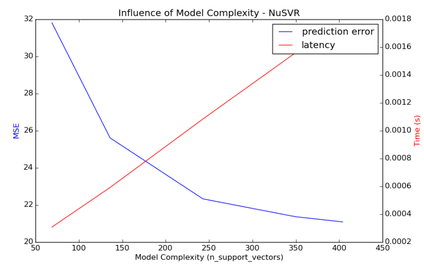
11.2.5 Model Complexity Influence

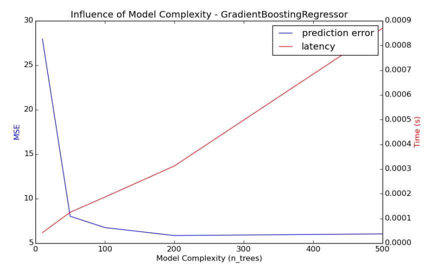
Demonstrate how model complexity influences both prediction accuracy and computational performance.

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).

- 

- 

- 

Script output:

```
Benchmarking SGDClassifier(alpha=0.001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.25,
       learning_rate='optimal', loss='modified_huber', n_iter=5, n_jobs=1,
       penalty='elasticnet', power_t=0.5, random_state=None, shuffle=True,
       verbose=0, warm_start=False)
Complexity: 4495 | Hamming Loss (Misclassification Ratio): 0.2602 | Pred. Time: 0.022719s

Benchmarking SGDClassifier(alpha=0.001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.5, learning_rate='optimal',
       loss='modified_huber', n_iter=5, n_jobs=1, penalty='elasticnet',
       power_t=0.5, random_state=None, shuffle=True, verbose=0,
       warm_start=False)
Complexity: 1644 | Hamming Loss (Misclassification Ratio): 0.2889 | Pred. Time: 0.018062s

Benchmarking SGDClassifier(alpha=0.001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.75,
       learning_rate='optimal', loss='modified_huber', n_iter=5, n_jobs=1,
       penalty='elasticnet', power_t=0.5, random_state=None, shuffle=True,
       verbose=0, warm_start=False)
Complexity: 879 | Hamming Loss (Misclassification Ratio): 0.3292 | Pred. Time: 0.015752s

Benchmarking SGDClassifier(alpha=0.001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.9, learning_rate='optimal',
       loss='modified_huber', n_iter=5, n_jobs=1, penalty='elasticnet',
       power_t=0.5, random_state=None, shuffle=True, verbose=0,
       warm_start=False)
```

```
Complexity: 654 | Hamming Loss (Misclassification Ratio): 0.3305 | Pred. Time: 0.012409s

Benchmarking NuSVR(C=1000.0, cache_size=200, coef0=0.0, degree=3, gamma=3.0517578125e-05,
   kernel='rbf', max_iter=-1, nu=0.1, shrinking=True, tol=0.001,
   verbose=False)
Complexity: 69 | MSE: 31.8133 | Pred. Time: 0.000308s

Benchmarking NuSVR(C=1000.0, cache_size=200, coef0=0.0, degree=3, gamma=3.0517578125e-05,
   kernel='rbf', max_iter=-1, nu=0.25, shrinking=True, tol=0.001,
   verbose=False)
Complexity: 136 | MSE: 25.6140 | Pred. Time: 0.000593s

Benchmarking NuSVR(C=1000.0, cache_size=200, coef0=0.0, degree=3, gamma=3.0517578125e-05,
   kernel='rbf', max_iter=-1, nu=0.5, shrinking=True, tol=0.001,
   verbose=False)
Complexity: 243 | MSE: 22.3315 | Pred. Time: 0.001084s

Benchmarking NuSVR(C=1000.0, cache_size=200, coef0=0.0, degree=3, gamma=3.0517578125e-05,
   kernel='rbf', max_iter=-1, nu=0.75, shrinking=True, tol=0.001,
   verbose=False)
Complexity: 350 | MSE: 21.3679 | Pred. Time: 0.001561s

Benchmarking NuSVR(C=1000.0, cache_size=200, coef0=0.0, degree=3, gamma=3.0517578125e-05,
   kernel='rbf', max_iter=-1, nu=0.9, shrinking=True, tol=0.001,
   verbose=False)
Complexity: 404 | MSE: 21.0915 | Pred. Time: 0.001658s

Benchmarking GradientBoostingRegressor(alpha=0.9, init=None, learning_rate=0.1, loss='ls',
             max_depth=3, max_features=None, max_leaf_nodes=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=10,
             random_state=None, subsample=1.0, verbose=0, warm_start=False)
Complexity: 10 | MSE: 27.9672 | Pred. Time: 0.000043s

Benchmarking GradientBoostingRegressor(alpha=0.9, init=None, learning_rate=0.1, loss='ls',
             max_depth=3, max_features=None, max_leaf_nodes=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=50,
             random_state=None, subsample=1.0, verbose=0, warm_start=False)
Complexity: 50 | MSE: 8.0288 | Pred. Time: 0.000125s

Benchmarking GradientBoostingRegressor(alpha=0.9, init=None, learning_rate=0.1, loss='ls',
             max_depth=3, max_features=None, max_leaf_nodes=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=100,
             random_state=None, subsample=1.0, verbose=0, warm_start=False)
Complexity: 100 | MSE: 6.7578 | Pred. Time: 0.000187s

Benchmarking GradientBoostingRegressor(alpha=0.9, init=None, learning_rate=0.1, loss='ls',
             max_depth=3, max_features=None, max_leaf_nodes=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=200,
             random_state=None, subsample=1.0, verbose=0, warm_start=False)
Complexity: 200 | MSE: 5.8592 | Pred. Time: 0.000312s

Benchmarking GradientBoostingRegressor(alpha=0.9, init=None, learning_rate=0.1, loss='ls',
             max_depth=3, max_features=None, max_leaf_nodes=None,
             min_samples_leaf=1, min_samples_split=2,
```

```
             min_weight_fraction_leaf=0.0, n_estimators=500,
             random_state=None, subsample=1.0, verbose=0, warm_start=False)
Complexity: 500 | MSE: 6.0492 | Pred. Time: 0.000870s
```

Python source code: `plot_model_complexity_influence.py`

```
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.classes import NuSVR
from sklearn.ensemble.gradient_boosting import GradientBoostingRegressor
from sklearn.linear_model.stochastic_gradient 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."""
    bunch = None
    if case == 'regression':
        bunch = datasets.load_boston()
    elif case == 'classification':
        bunch = datasets.fetch_20newsgroups_vectorized(subset='all')
    X, y = shuffle(bunch.data, bunch.target)
    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)
    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},
     '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 = \
        benchmark_influence(conf)
    plot_influence(conf, prediction_performances, prediction_times,
                   complexities)
```

Total running time of the example: 19.42 seconds (0 minutes 19.42 seconds)

11.2.6 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 *OneClassSVM* provided by the package *sklearn.svm* 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:

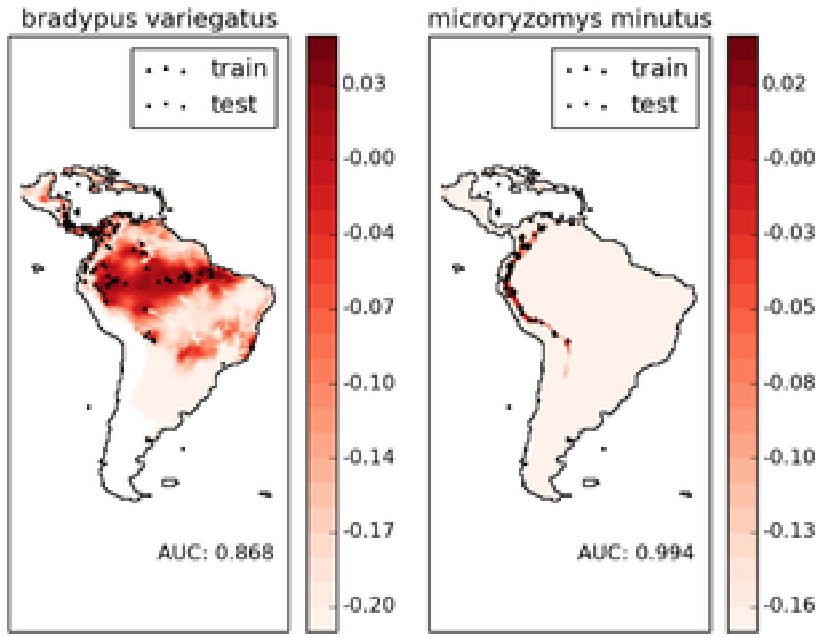
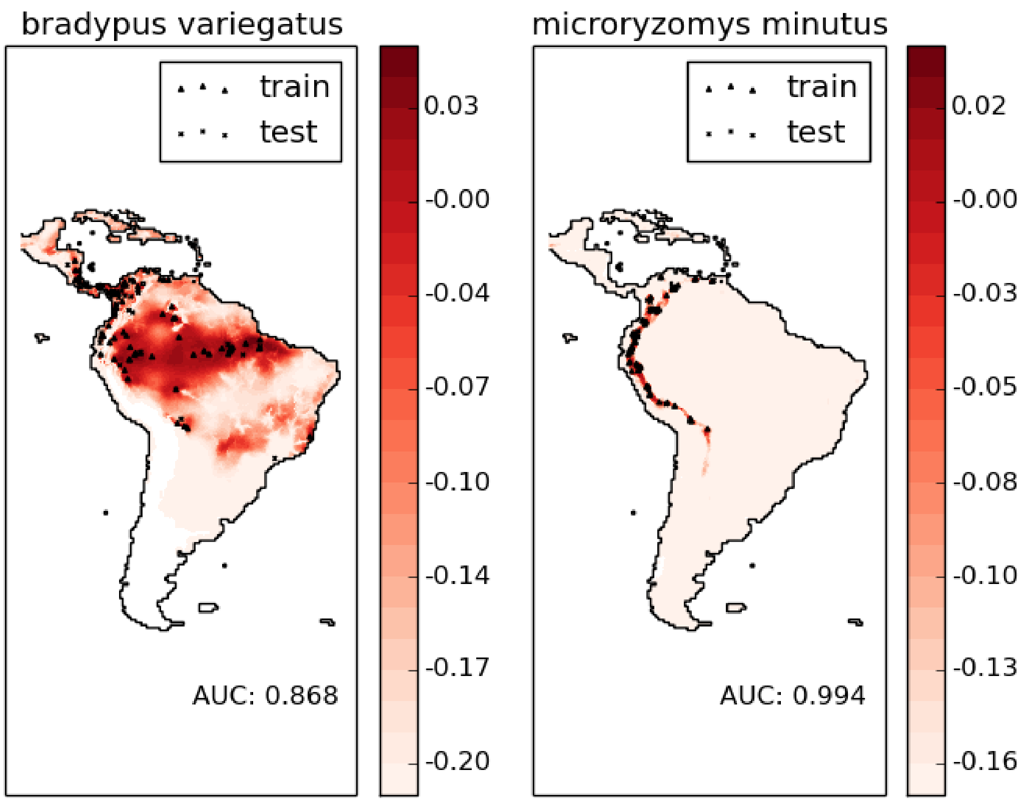


Figure 11.17: *Species distribution modeling*

- "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.



Script output:

```
Modeling distribution of species 'bradypus variegatus'
 - fit OneClassSVM ... done.
 - plot coastlines from coverage
 - predict species distribution

 Area under the ROC curve : 0.868380
```

```
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: 4.44s
```

Python source code: `plot_species_distribution_modeling.py`

```python
# Authors: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#          Jake Vanderplas <vanderplas@astro.washington.edu>
#
# License: BSD 3 clause

from __future__ import print_function
```

```
from time import time

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets.base import Bunch
from sklearn.datasets import fetch_species_distributions
from sklearn.datasets.species_distributions import construct_grids
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 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.
    """
    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.
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=[-9999], 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)[:, 0]
        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'],
                    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)[:, 0]
        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 example: 4.51 seconds (0 minutes 4.51 seconds)

11.2.7 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.

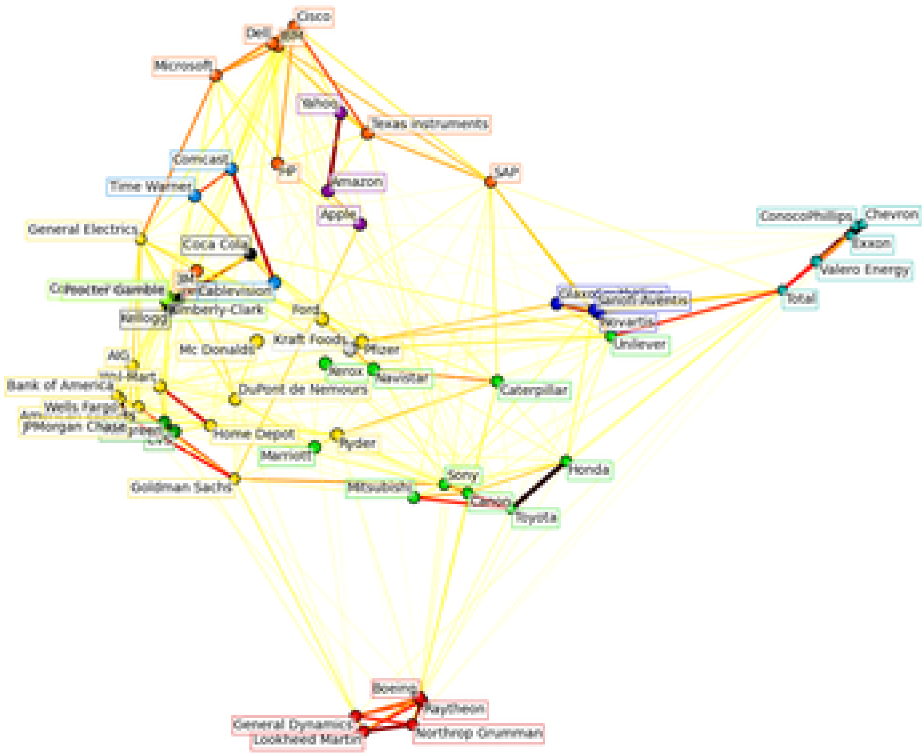


Figure 11.18: *Visualizing the stock market structure*

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.

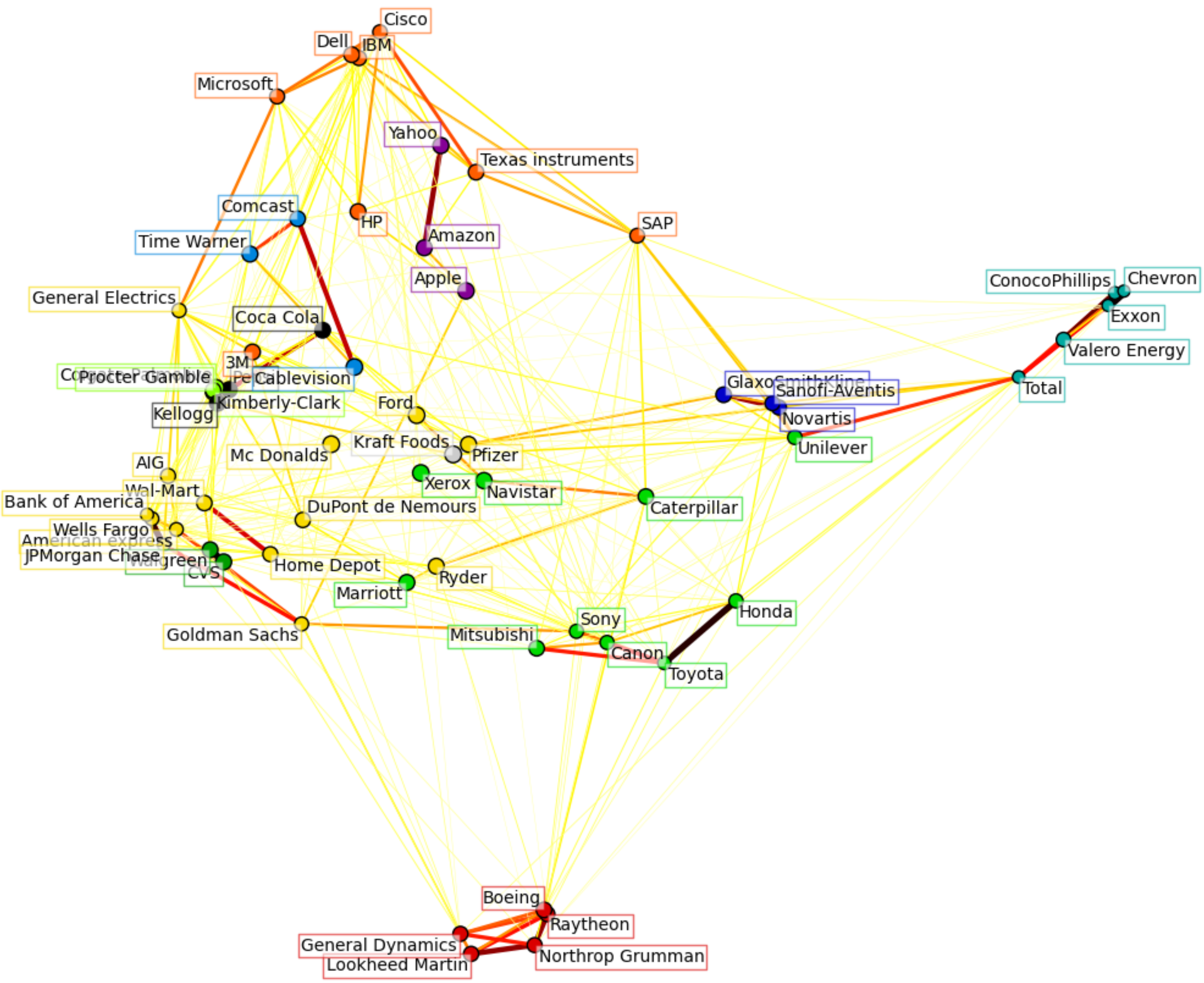
Visualization

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.



Script output:

```
Cluster 1: Pepsi, Coca Cola, Kellogg
Cluster 2: Apple, Amazon, Yahoo
Cluster 3: GlaxoSmithKline, Novartis, Sanofi-Aventis
Cluster 4: Comcast, Time Warner, Cablevision
Cluster 5: ConocoPhillips, Chevron, Total, Valero Energy, Exxon
Cluster 6: CVS, Walgreen
Cluster 7: Navistar, Sony, Marriott, Caterpillar, Canon, Toyota, Honda, Mitsubishi, Xerox, Unilever
Cluster 8: Kimberly-Clark, Colgate-Palmolive, Procter Gamble
Cluster 9: American express, Ryder, Goldman Sachs, Wal-Mart, General Electrics, Pfizer, Wells Fargo,
Cluster 10: Microsoft, SAP, 3M, IBM, Texas instruments, HP, Dell, Cisco
Cluster 11: Raytheon, Boeing, Lookheed Martin, General Dynamics, Northrop Grumman
Cluster 12: Kraft Foods
```

Python source code: `plot_stock_market.py`

```
print(__doc__)

# Author: Gael Varoquaux gael.varoquaux@normalesup.org
# License: BSD 3 clause

import datetime

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import finance
from matplotlib.collections import LineCollection

from sklearn import cluster, covariance, manifold

###############################################################################
# Retrieve the data from Internet

# Choose a time period reasonably calm (not too long ago so that we get
# high-tech firms, and before the 2008 crash)
d1 = datetime.datetime(2003, 1, 1)
d2 = datetime.datetime(2008, 1, 1)

# kraft symbol has now changed from KFT to MDLZ in yahoo
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',
    'MTU': 'Mitsubishi',
    'SNE': 'Sony',
    'F': 'Ford',
    'HMC': 'Honda',
    'NAV': 'Navistar',
    'NOC': 'Northrop Grumman',
    'BA': 'Boeing',
    'KO': 'Coca Cola',
    'MMM': '3M',
    'MCD': 'Mc Donalds',
    'PEP': 'Pepsi',
    'MDLZ': 'Kraft Foods',
    'K': 'Kellogg',
    'UN': 'Unilever',
    'MAR': 'Marriott',
    'PG': 'Procter Gamble',
    'CL': 'Colgate-Palmolive',
    'GE': 'General Electrics',
```

```
    'WFC': 'Wells Fargo',
    'JPM': 'JPMorgan Chase',
    'AIG': 'AIG',
    'AXP': 'American express',
    'BAC': 'Bank of America',
    'GS': 'Goldman Sachs',
    'AAPL': 'Apple',
    'SAP': 'SAP',
    'CSCO': 'Cisco',
    'TXN': 'Texas instruments',
    'XRX': 'Xerox',
    'LMT': 'Lookheed Martin',
    'WMT': 'Wal-Mart',
    'WBA': 'Walgreen',
    '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(list(symbol_dict.items())).T

quotes = [finance.quotes_historical_yahoo(symbol, d1, d2, asobject=True)
          for symbol in symbols]

open = np.array([q.open for q in quotes]).astype(np.float)
close = np.array([q.close for q in quotes]).astype(np.float)

# The daily variations of the quotes are what carry most information
variation = close - open

###############################################################################
# Learn a graphical structure from the correlations
edge_model = covariance.GraphLassoCV()

# 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()

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.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))]
    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.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 example: 2.39 seconds (0 minutes 2.39 seconds)

Figure 11.19: *Wikipedia principal eigenvector*

11.2.8 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:

> http://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:

> http://en.wikipedia.org/wiki/Power_iteration

Here the computation is achieved thanks to Martinsson's Randomized SVD algorithm implemented in the scikit.

The graph data is fetched from the DBpedia dumps. DBpedia is an extraction of the latent structured data of the Wikipedia content.

Python source code: `wikipedia_principal_eigenvector.py`

```
# Author: Olivier Grisel <olivier.grisel@ensta.org>
# License: BSD 3 clause

from __future__ import print_function

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 sklearn.decomposition import randomized_svd
from sklearn.externals.joblib import Memory


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):
        import urllib
        print("Downloading data from '%s', please wait..." % url)
        opener = urllib.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)


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 = set([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)):
        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 = dict((i, name) for name, i in index_map.iteritems())

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 compenents of the 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")
    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(X.sum(axis=1) == 0, 1.0 / n, 0)).ravel()

    scores = np.ones(n, dtype=np.float32) / n  # 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, tol=1e-10)
print("done in %0.3fs" % (time() - t0))
pprint([names[i] for i in np.abs(scores).argsort()[-10:]])
```

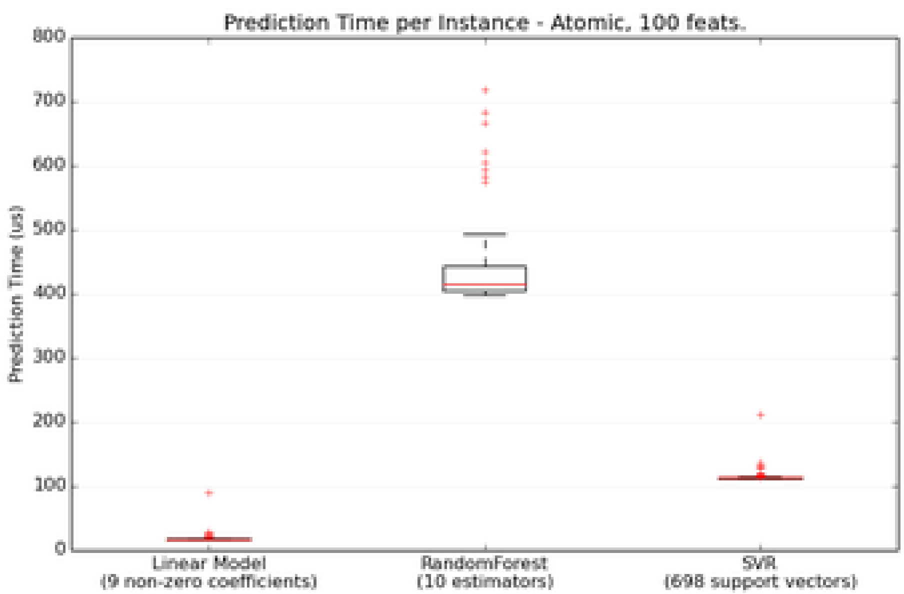


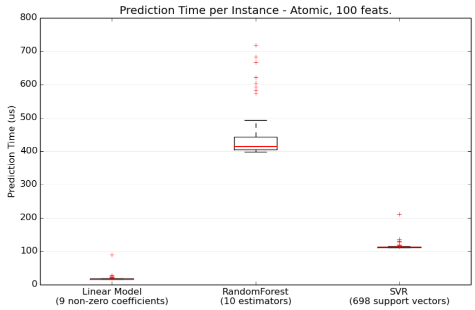
Figure 11.20: *Prediction Latency*

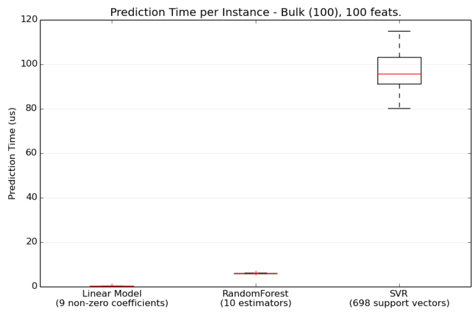
11.2.9 Prediction Latency

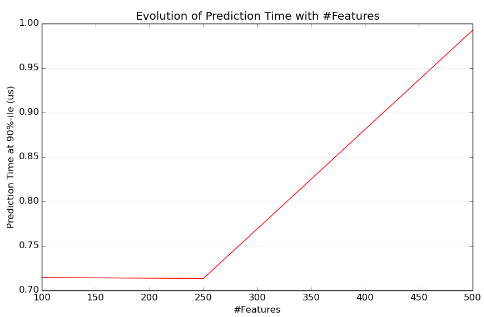
This is an example showing the prediction latency of various scikit-learn estimators.

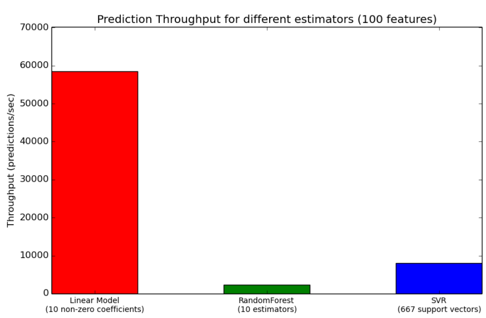
The goal is to measure the latency one can expect when doing predictions either in bulk or atomic (i.e. one by one) mode.

The plots represent the distribution of the prediction latency as a boxplot.

- 

- 

- 

- 

Script output:

```
Benchmarking SGDRegressor(alpha=0.01, average=False, epsilon=0.1, eta0=0.01,
       fit_intercept=True, l1_ratio=0.25, learning_rate='invscaling',
       loss='squared_loss', n_iter=5, penalty='elasticnet', power_t=0.25,
       random_state=None, shuffle=True, verbose=0, warm_start=False)
Benchmarking RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features='auto', max_leaf_nodes=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
           verbose=0, warm_start=False)
Benchmarking SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma=0.0,
  kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
benchmarking with 100 features
benchmarking with 250 features
benchmarking with 500 features
example run in 3.76s
```

Python source code: `plot_prediction_latency.py`

```python
# Authors: Eustache Diemert <eustache@diemert.fr>
# License: BSD 3 clause

from __future__ import print_function
from collections import defaultdict

import time
import gc
import numpy as np
import matplotlib.pyplot as plt

from scipy.stats import scoreatpercentile
from sklearn.datasets.samples_generator import make_regression
from sklearn.ensemble.forest import RandomForestRegressor
from sklearn.linear_model.ridge import Ridge
from sklearn.linear_model.stochastic_gradient import SGDRegressor
from sklearn.svm.classes import SVR


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), scoreatpercentile(
            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), scoreatpercentile(
            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


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)
    X_train = X[:n_train]
    y_train = y[:n_train]
    X_test = X[n_train:]
    y_test = y[n_train:]
    idx = np.arange(n_train)
    np.random.seed(13)
    np.random.shuffle(idx)
    X_train = X_train[idx]
    y_train = y_train[idx]

    std = X_train.std(axis=0)
    mean = X_train.mean(axis=0)
    X_train = (X_train - mean) / std
    X_test = (X_test - mean) / std

    std = y_train.std(axis=0)
```

```
    mean = y_train.mean(axis=0)
    y_train = (y_train - mean) / std
    y_test = (y_test - mean) / std

    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)

    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():
            estimator.fit(X_train, y_train)
            gc.collect()
            runtimes = bulk_benchmark_estimator(estimator, X_test, 30, False)
            percentiles[cls_name][n] = 1e6 * scoreatpercentile(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)')
    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, fit_intercept=True),
         '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 example: 3.83 seconds (0 minutes 3.83 seconds)

Figure 11.21: *Libsvm GUI*

11.2.10 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.

Python source code: `svm_gui.py`

```
from __future__ import division, print_function
```

```python
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 Tkinter as Tk
import sys
import numpy as np

from sklearn import svm
from sklearn.datasets import dump_svmlight_file
from sklearn.externals.six.moves import xrange

y_min, y_max = -50, 50
x_min, x_max = -50, 50


class Model(object):
    """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)
```

```python
class Controller(object):
    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())
        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(object):
    """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()
        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, "RBF: $\exp (-\gamma \| u-v \|^2)$")
        self.ax.text(10, -60, "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 xrange(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()

    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(object):
    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)
        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)
    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)
```

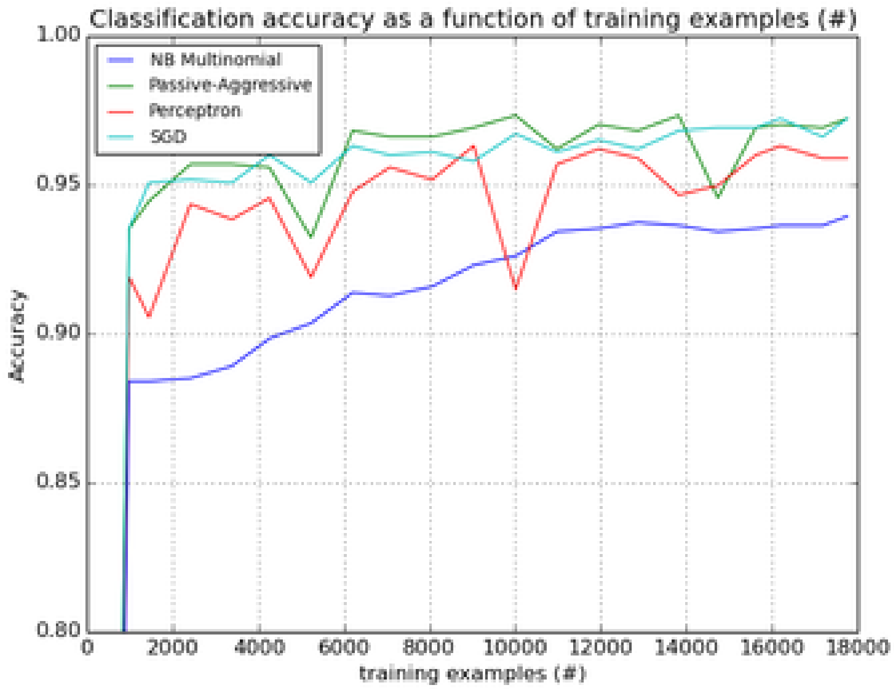


Figure 11.22: *Out-of-core classification of text documents*

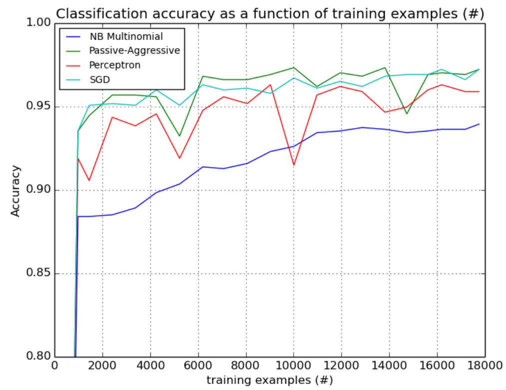
11.2.11 Out-of-core classification of text documents

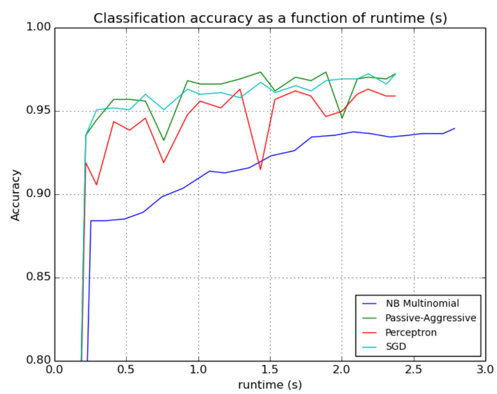
This is an example showing how scikit-learn can be used for classification using an out-of-core approach: learning from data that doesn't fit into main memory. We make use of an online classifier, i.e., one that supports the partial_fit method, that will be fed with batches of examples. To guarantee 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.

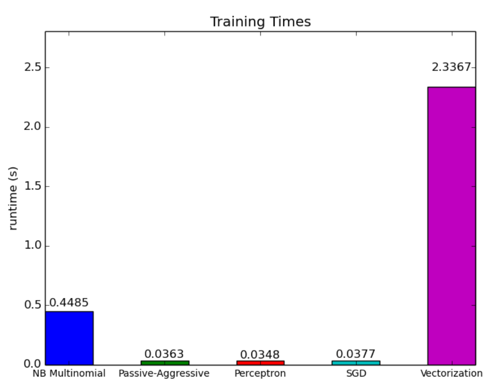
The dataset used in this example is Reuters-21578 as provided by the UCI ML repository. It will be automatically downloaded and uncompressed on first run.

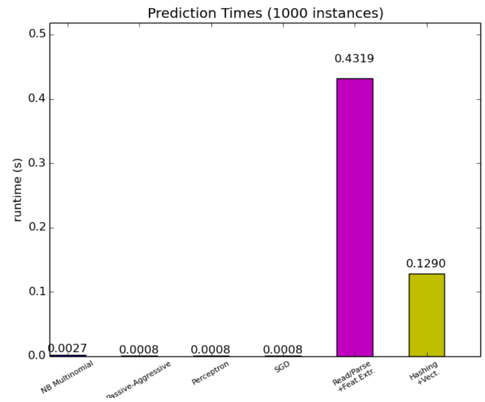
The plot represents the learning curve of the classifier: the evolution of classification accuracy over the course of the mini-batches. Accuracy is measured on the first 1000 samples, held out as a validation set.

To limit the memory consumption, we queue examples up to a fixed amount before feeding them to the learner.

- 

- 

- 

• 

Script output:

```
Test set is 975 documents (114 positive)
  Passive-Aggressive classifier :          985 train docs (   132 positive)    975 test docs (   114
          Perceptron classifier :          985 train docs (   132 positive)    975 test docs (   114
                 SGD classifier :          985 train docs (   132 positive)    975 test docs (   114
      NB Multinomial classifier :          985 train docs (   132 positive)    975 test docs (   114


  Passive-Aggressive classifier :         3383 train docs (   396 positive)    975 test docs (   114
          Perceptron classifier :         3383 train docs (   396 positive)    975 test docs (   114
                 SGD classifier :         3383 train docs (   396 positive)    975 test docs (   114
      NB Multinomial classifier :         3383 train docs (   396 positive)    975 test docs (   114


  Passive-Aggressive classifier :         6203 train docs (   784 positive)    975 test docs (   114
          Perceptron classifier :         6203 train docs (   784 positive)    975 test docs (   114
                 SGD classifier :         6203 train docs (   784 positive)    975 test docs (   114
      NB Multinomial classifier :         6203 train docs (   784 positive)    975 test docs (   114


  Passive-Aggressive classifier :         9033 train docs (  1079 positive)    975 test docs (   114
          Perceptron classifier :         9033 train docs (  1079 positive)    975 test docs (   114
                 SGD classifier :         9033 train docs (  1079 positive)    975 test docs (   114
      NB Multinomial classifier :         9033 train docs (  1079 positive)    975 test docs (   114


  Passive-Aggressive classifier :        11951 train docs (  1440 positive)    975 test docs (   114
          Perceptron classifier :        11951 train docs (  1440 positive)    975 test docs (   114
                 SGD classifier :        11951 train docs (  1440 positive)    975 test docs (   114
      NB Multinomial classifier :        11951 train docs (  1440 positive)    975 test docs (   114


  Passive-Aggressive classifier :        14736 train docs (  1810 positive)    975 test docs (   114
          Perceptron classifier :        14736 train docs (  1810 positive)    975 test docs (   114
                 SGD classifier :        14736 train docs (  1810 positive)    975 test docs (   114
      NB Multinomial classifier :        14736 train docs (  1810 positive)    975 test docs (   114


  Passive-Aggressive classifier :        17179 train docs (  2101 positive)    975 test docs (   114
          Perceptron classifier :        17179 train docs (  2101 positive)    975 test docs (   114
                 SGD classifier :        17179 train docs (  2101 positive)    975 test docs (   114
      NB Multinomial classifier :        17179 train docs (  2101 positive)    975 test docs (   114
```

Python source code: `plot_out_of_core_classification.py`

```python
# Authors: Eustache Diemert <eustache@diemert.fr>
#          @FedericoV <https://github.com/FedericoV/>
# License: BSD 3 clause

from __future__ import print_function

from glob import glob
import itertools
import os.path
import re
import tarfile
import time

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams

from sklearn.externals.six.moves import html_parser
from sklearn.externals.six.moves import urllib
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
###############################################################################


class ReutersParser(html_parser.HTMLParser):
    """Utility class to parse a SGML file and yield documents one at a time."""

    def __init__(self, encoding='latin-1'):
        html_parser.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):
        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

    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():
                print('\rdownloaded %s / %s' % (current_sz_mb, total_sz_mb),
                      end='')

        archive_path = os.path.join(data_path, ARCHIVE_FILENAME)
        urllib.request.urlretrieve(DOWNLOAD_URL, filename=archive_path,
                                   reporthook=progress)
        if _not_in_sphinx():
            print('\r', end='')
        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,
                               non_negative=True)


# 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(),
    '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 = [(u'{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
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-batchs 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()
        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')


###############################################################################
# Plot results
###############################################################################


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'])
    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 = []
for cls_name, stats in sorted(cls_stats.items()):
    cls_runtime.append(stats['total_fit_time'])

cls_runtime.append(total_vect_time)
cls_names.append('Vectorization')
bar_colors = rcParams['axes.color_cycle'][:len(cls_names)]

ax = plt.subplot(111)
rectangles = plt.bar(range(len(cls_names)), cls_runtime, width=0.5,
                     color=bar_colors)

ax.set_xticks(np.linspace(0.25, len(cls_names) - 0.75, 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')

autolabel(rectangles)
plt.show()

# Plot prediction times
plt.figure()
#fig = plt.gcf()
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)
cls_names.append('Read/Parse\n+Feat.Extr.')
cls_runtime.append(vectorizing_time)
cls_names.append('Hashing\n+Vect.')
bar_colors = rcParams['axes.color_cycle'][:len(cls_names)]

ax = plt.subplot(111)
rectangles = plt.bar(range(len(cls_names)), cls_runtime, width=0.5,
                     color=bar_colors)

ax.set_xticks(np.linspace(0.25, len(cls_names) - 0.75, 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.show()
```

Total running time of the example: 13.54 seconds (0 minutes 13.54 seconds)

11.3 Biclustering

Examples concerning the `sklearn.cluster.bicluster` module.

11.3.1 A demo of the Spectral Co-Clustering algorithm

This example demonstrates how to generate a dataset and bicluster it using the 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

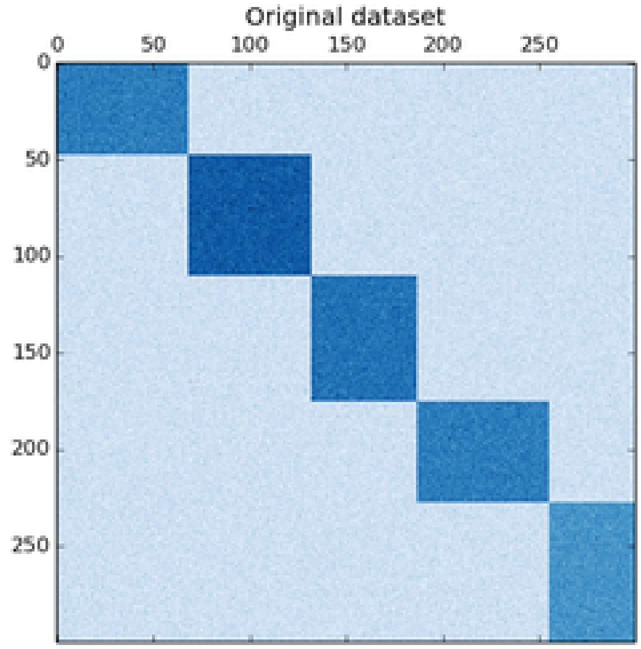
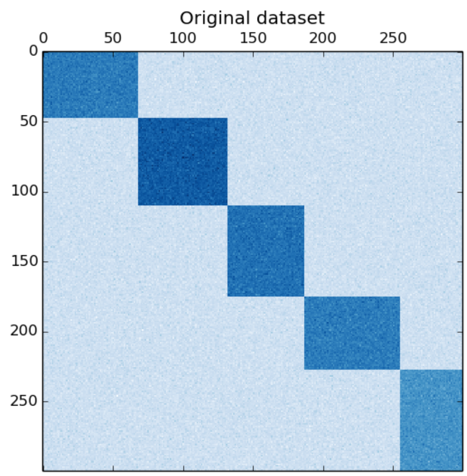
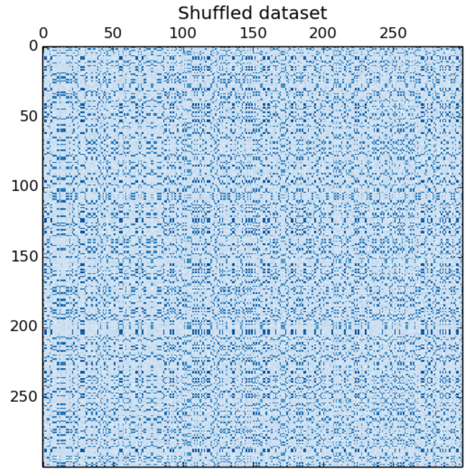


Figure 11.23: *A demo of the Spectral Co-Clustering algorithm*

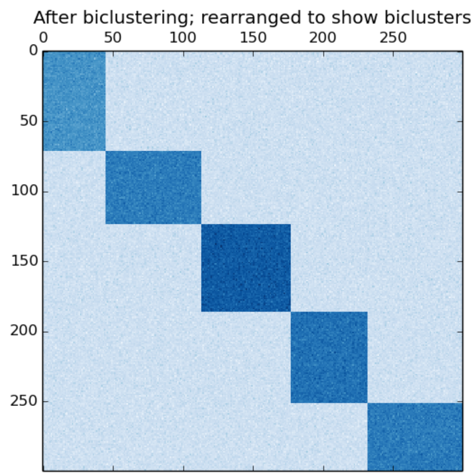
algorithm. Rearranging the shuffled matrix to make biclusters contiguous shows how accurately the algorithm found the biclusters.



•



•



•

Script output:

```
consensus score: 1.000
```

Python source code: `plot_spectral_coclustering.py`

```
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.datasets import samples_generator as sg
from sklearn.cluster.bicluster 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)

plt.matshow(data, cmap=plt.cm.Blues)
plt.title("Original dataset")

data, row_idx, col_idx = sg._shuffle(data, random_state=0)
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 example: 0.18 seconds (0 minutes 0.18 seconds)

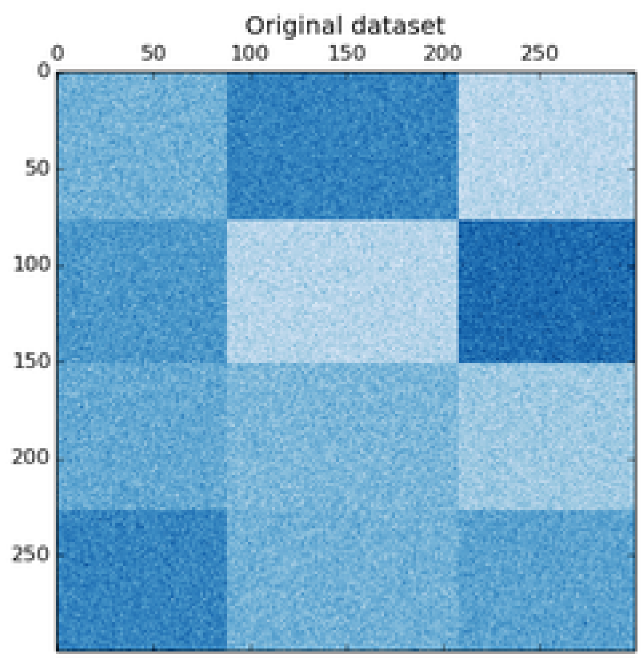


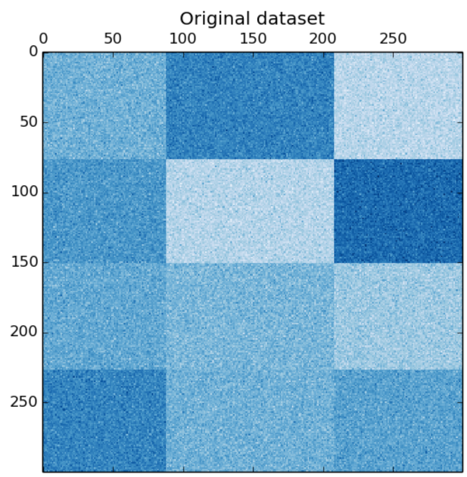
Figure 11.24: *A demo of the Spectral Biclustering algorithm*

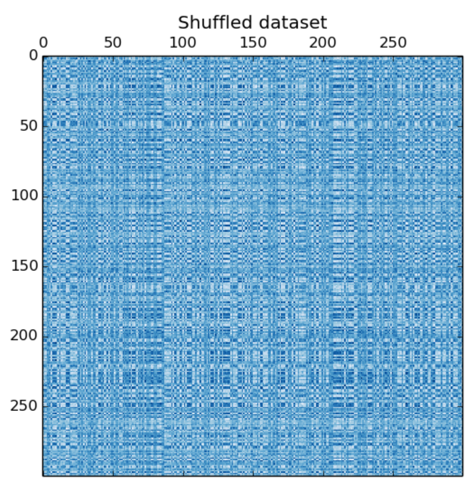
11.3.2 A demo of the Spectral Biclustering algorithm

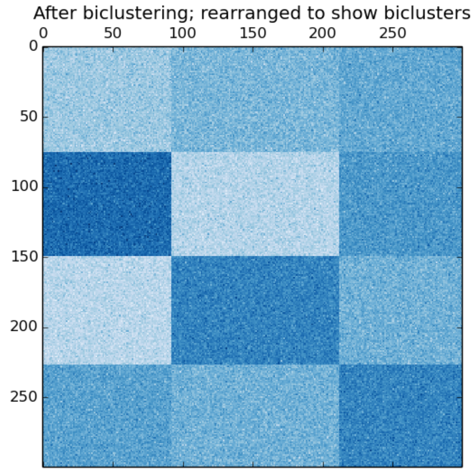
This example demonstrates how to generate a checkerboard dataset and bicluster it using the Spectral Biclustering algorithm.

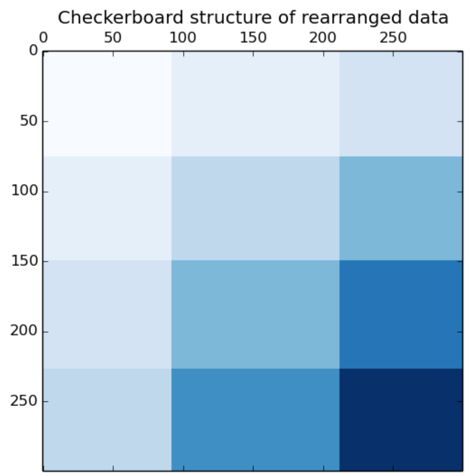
The data is generated with the `make_checkerboard` function, then shuffled and passed to the Spectral Biclustering algorithm. The rows and columns of the shuffled matrix are rearranged to show the biclusters found by the algorithm.

The outer product of the row and column label vectors shows a representation of the checkerboard structure.

- 

- 

- 

- 

Script output:

```
consensus score: 1.0
```

Python source code: `plot_spectral_biclustering.py`

```
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.datasets import samples_generator as sg
from sklearn.cluster.bicluster 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,
    shuffle=False, random_state=0)

plt.matshow(data, cmap=plt.cm.Blues)
plt.title("Original dataset")

data, row_idx, col_idx = sg._shuffle(data, random_state=0)
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 example: 0.39 seconds (0 minutes 0.39 seconds)

Figure 11.25: *Biclustering documents with the Spectral Co-clustering algorithm*

11.3.3 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.

Output:

```
Vectorizing...
Coclustering...
Done in 9.53s. V-measure: 0.4455
MiniBatchKMeans...
Done in 12.00s. V-measure: 0.3309

Best biclusters:
----------------
bicluster 0 : 1951 documents, 4373 words
categories   : 23% talk.politics.guns, 19% talk.politics.misc, 14% sci.med
words        : gun, guns, geb, banks, firearms, drugs, gordon, clinton, cdt, amendment

bicluster 1 : 1165 documents, 3304 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 2 : 2219 documents, 2830 words
categories   : 18% comp.sys.mac.hardware, 16% comp.sys.ibm.pc.hardware, 16% comp.graphics
words        : voltage, dsp, board, receiver, circuit, shipping, packages, stereo, compression, packa

bicluster 3 : 1860 documents, 2745 words
categories   : 26% rec.motorcycles, 23% rec.autos, 13% misc.forsale
words        : bike, car, dod, engine, motorcycle, ride, honda, cars, bmw, bikes

bicluster 4 : 12 documents, 155 words
categories   : 100% rec.sport.hockey
words        : scorer, unassisted, reichel, semak, sweeney, kovalenko, ricci, audette, momesso, nedve
```

Python source code: `bicluster_newsgroups.py`

```python
from __future__ import print_function

print(__doc__)

from collections import defaultdict
import operator
import re
from time import time

import numpy as np

from sklearn.cluster.bicluster import SpectralCoclustering
from sklearn.cluster import MiniBatchKMeans
from sklearn.externals import six
from sklearn.datasets.twenty_newsgroups import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.cluster import v_measure_score
```

```python
def number_aware_tokenizer(doc):
    """ Tokenizer that maps all numeric tokens to a placeholder.

    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.
    """
    token_pattern = re.compile(u'(?u)\\b\\w\\w+\\b')
    tokens = token_pattern.findall(doc)
    tokens = ["#NUMBER" if token[0] in "0123456789_" else token
              for token in tokens]
    return tokens

# 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 = TfidfVectorizer(stop_words='english', min_df=5,
                             tokenizer=number_aware_tokenizer)
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)):
        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]
    weight = X[rows[:, np.newaxis], cols].sum()
    cut = (X[row_complement[:, np.newaxis], cols].sum() +
           X[rows[:, np.newaxis], 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(six.iteritems(d), key=operator.itemgetter(1), reverse=True)


bicluster_ncuts = list(bicluster_ncut(i)
                       for i in xrange(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)))
```

11.4 Calibration

Examples illustrating the calibration of predicted probabilities of classifiers.

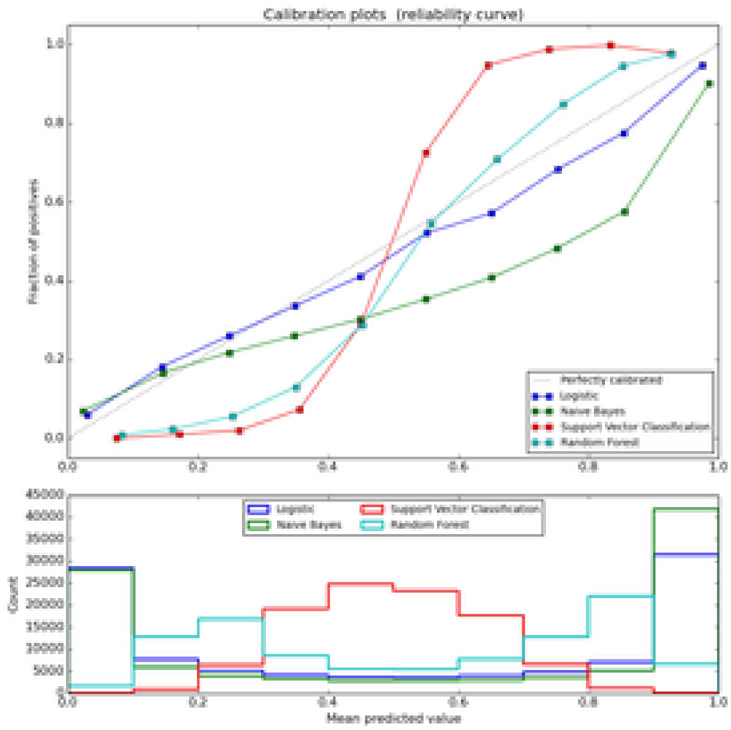


Figure 11.26: *Comparison of Calibration of Classifiers*

11.4.1 Comparison of Calibration of Classifiers

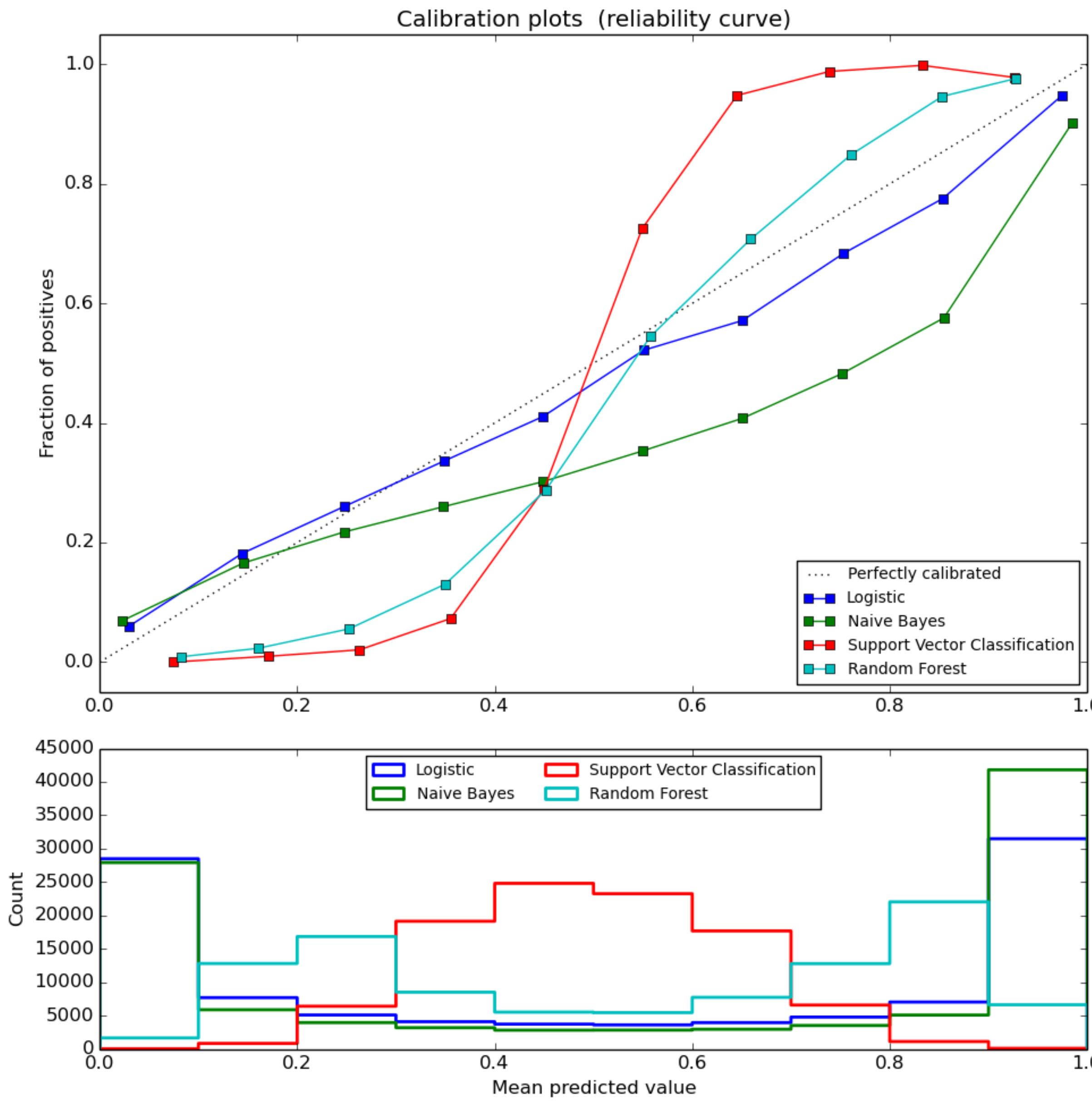
Well calibrated classifiers are probabilistic classifiers for which the output of the predict_proba method can be directly interpreted as a confidence level. For instance a well calibrated (binary) classifier should classify the samples such that among the samples to which it gave a predict_proba value 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 probilities, with different biases per method:

- GaussianNaiveBayes tends to push probabilties 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 subseting." As a result, the calibration curve shows a characteristic sigmoid shape, indicating that the classifier could trust its "intuition" more and return probabilties 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:



Python source code: `plot_compare_calibration.py`

```
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
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(n_estimators=100)


###############################################################################
# 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)

plt.tight_layout()
plt.show()
```

Total running time of the example: 1.13 seconds (0 minutes 1.13 seconds)

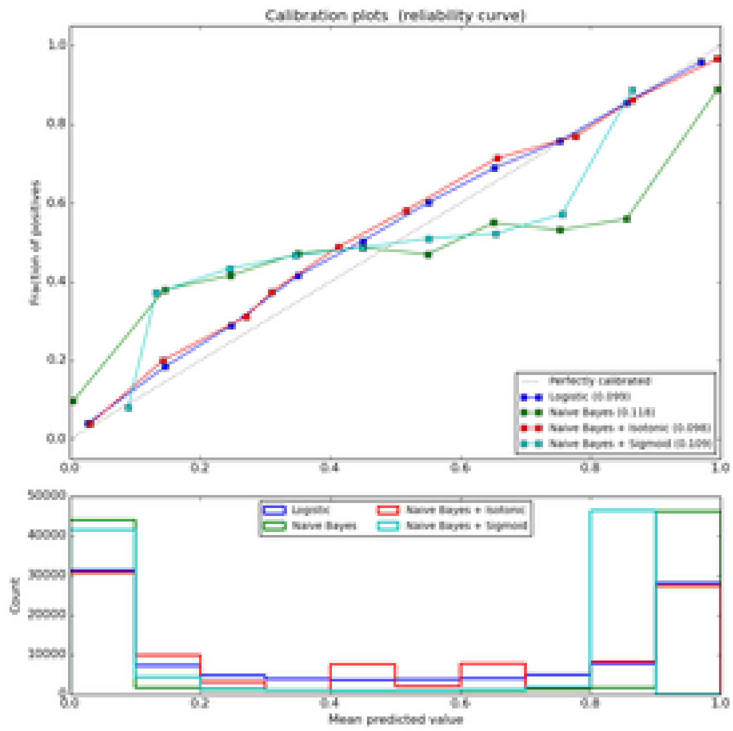


Figure 11.27: *Probability Calibration curves*

11.4.2 Probability Calibration curves

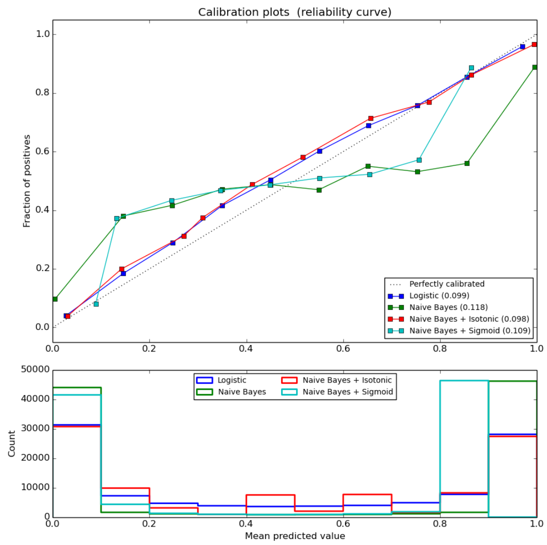
When performing classification one often wants to predict not only the class label, but also the associated probability. This probability gives some kind of confidence on the prediction. This example demonstrates how to display how well calibrated the predicted probabilities are and how to calibrate an uncalibrated classifier.

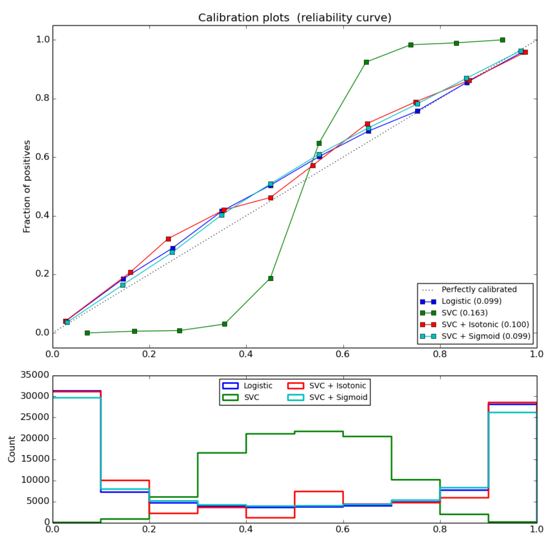
The experiment is performed on an artificial dataset for binary classification with 100.000 samples (1.000 of them are used for model fitting) with 20 features. Of the 20 features, only 2 are informative and 10 are redundant. The first figure shows the estimated probabilities obtained with logistic regression, 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).

- 

- 

Script output:

```
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
        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
```

Python source code: `plot_calibration_curve.py`

```
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
```

```python
from sklearn.cross_validation import train_test_split


# 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., solver='lbfgs')

    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])
    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 cuve for Gaussian Naive Bayes
plot_calibration_curve(GaussianNB(), "Naive Bayes", 1)

# Plot calibration cuve for Linear SVC
plot_calibration_curve(LinearSVC(), "SVC", 2)

plt.show()
```

Total running time of the example: 1.88 seconds (0 minutes 1.88 seconds)

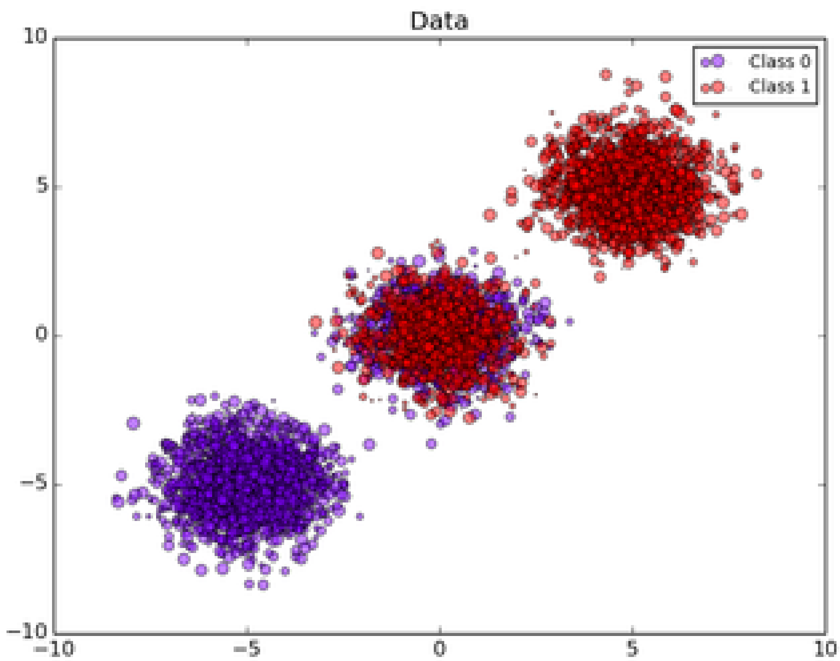
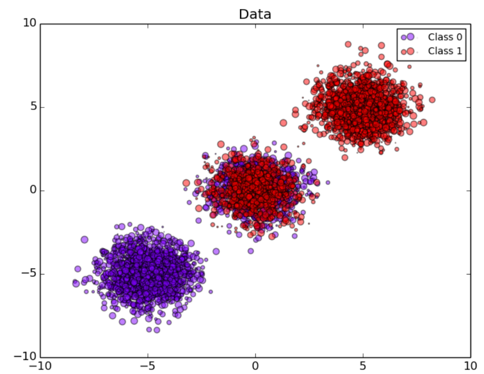


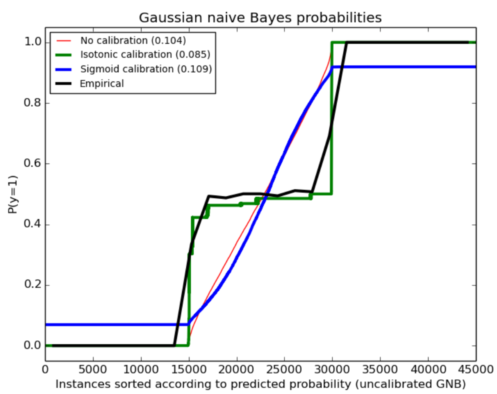
Figure 11.28: *Probability calibration of classifiers*

11.4.3 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 http://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.

- 

- 

Script output:

```
Brier scores: (the smaller the better)
No calibration: 0.104
With isotonic calibration: 0.085
With sigmoid calibration: 0.109
```

Python source code: `plot_calibration.py`

```python
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
from sklearn.calibration import CalibratedClassifierCV
from sklearn.cross_validation 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, n_features=2, cluster_std=1.0,
                  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, c=color, alpha=0.5,
                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 example: 0.43 seconds (0 minutes 0.43 seconds)

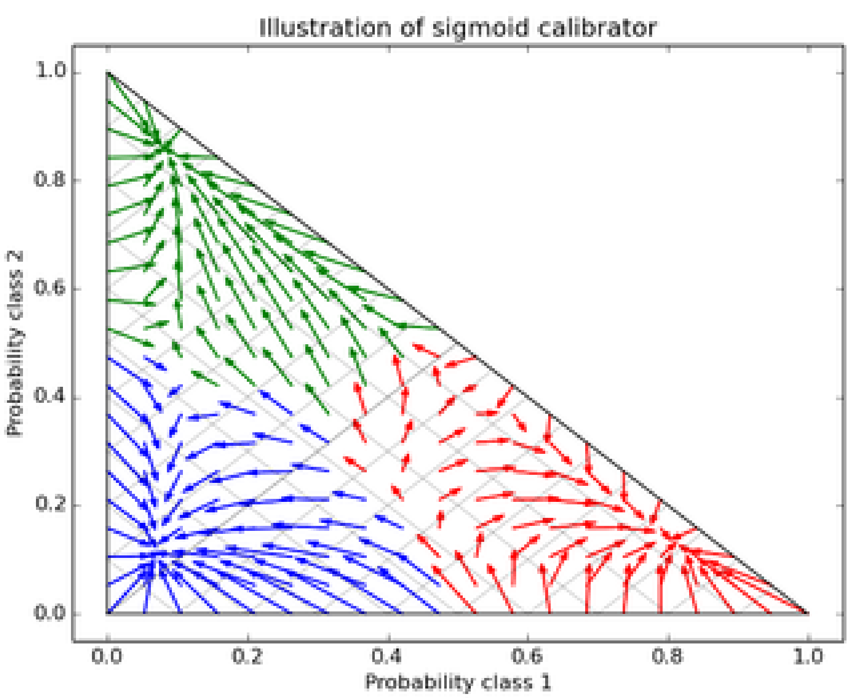
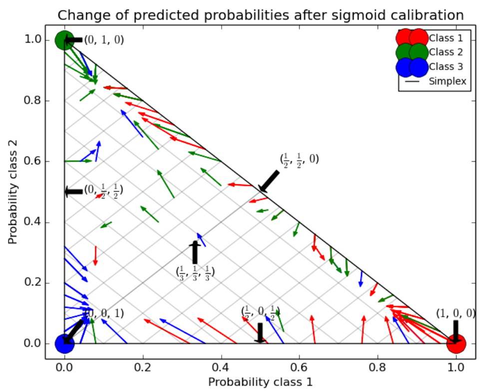


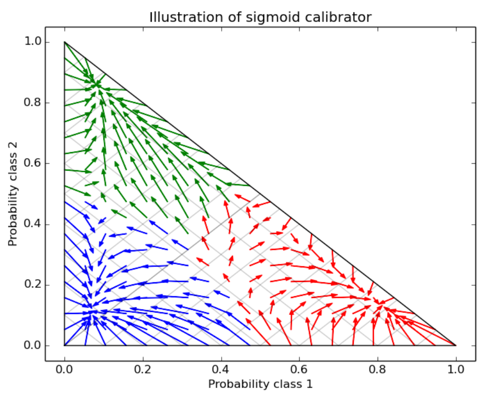
Figure 11.29: *Probability Calibration for 3-class classification*

11.4.4 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.

- 

- 

Script output:

```
Log-loss of
 * uncalibrated classifier trained on 800 datapoints: 1.280
 * classifier trained on 600 datapoints and calibrated on 200 datapoint: 0.536
```

Python source code: `plot_calibration_multiclass.py`

```python
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)

# Generate data
X, y = make_blobs(n_samples=1000, n_features=2, 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(0)
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'($\frac{1}{2}$, $\frac{1}{2}$, $0$)',
             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("off")
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(1)
# 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]

# Ploit 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")

plt.grid("off")
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 example: 0.51 seconds (0 minutes 0.51 seconds)

11.5 Classification

General examples about classification algorithms.

11.5.1 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*.

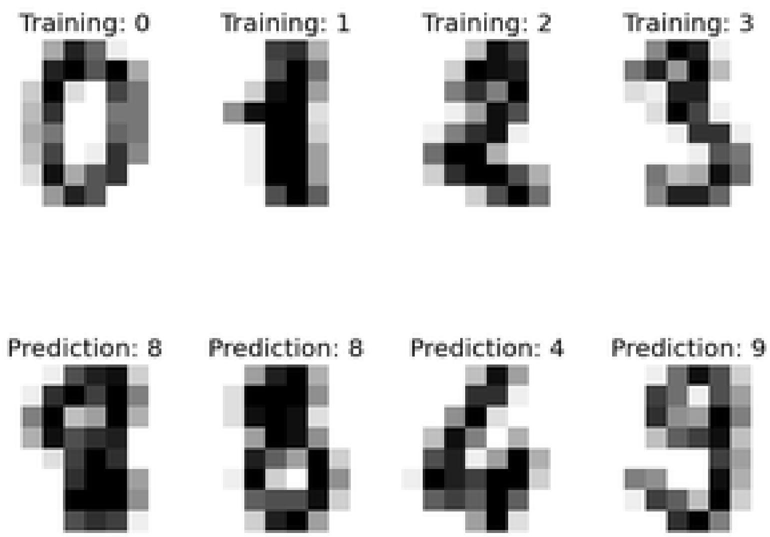
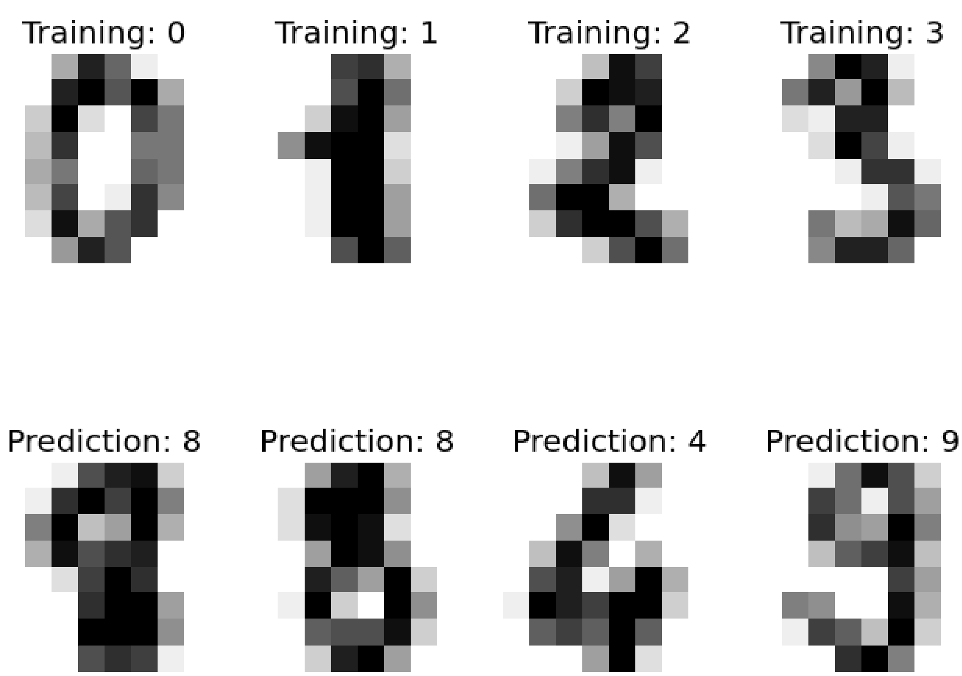


Figure 11.30: *Recognizing hand-written digits*



Script output:

```
Classification report for classifier SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=
  gamma=0.001, kernel='rbf', max_iter=-1, probability=False,
  random_state=None, shrinking=True, tol=0.001, verbose=False):
             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

avg / total       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]
 [ 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]]
```

Python source code: `plot_digits_classification.py`

```python
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

# 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 3 images, stored in the `images` attribute of the
# dataset.  If we were working from image files, we could load them using
# pylab.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.
images_and_labels = list(zip(digits.images, digits.target))
for index, (image, label) in enumerate(images_and_labels[:4]):
    plt.subplot(2, 4, index + 1)
    plt.axis('off')
```

```
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.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)

# We learn the digits on the first half of the digits
classifier.fit(data[:n_samples / 2], digits.target[:n_samples / 2])

# Now predict the value of the digit on the second half:
expected = digits.target[n_samples / 2:]
predicted = classifier.predict(data[n_samples / 2:])

print("Classification report for classifier %s:\n%s\n"
      % (classifier, metrics.classification_report(expected, predicted)))
print("Confusion matrix:\n%s" % metrics.confusion_matrix(expected, predicted))

images_and_predictions = list(zip(digits.images[n_samples / 2:], predicted))
for index, (image, prediction) in enumerate(images_and_predictions[:4]):
    plt.subplot(2, 4, index + 5)
    plt.axis('off')
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.title('Prediction: %i' % prediction)

plt.show()
```

Total running time of the example: 0.42 seconds (0 minutes 0.42 seconds)

11.5.2 Normal and Shrinkage Linear Discriminant Analysis for classification

Shows how shrinkage improves classification.

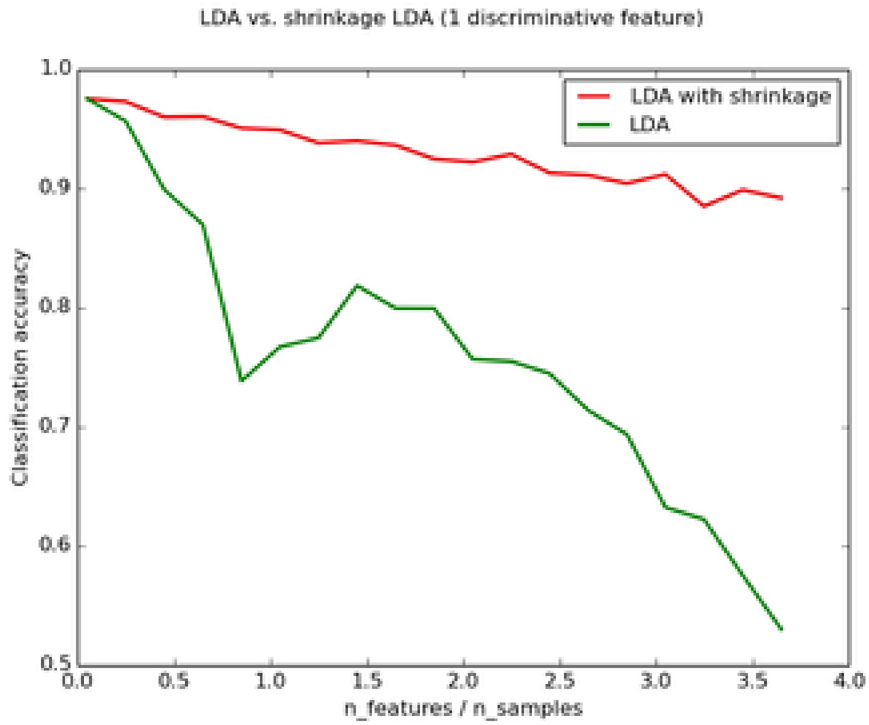
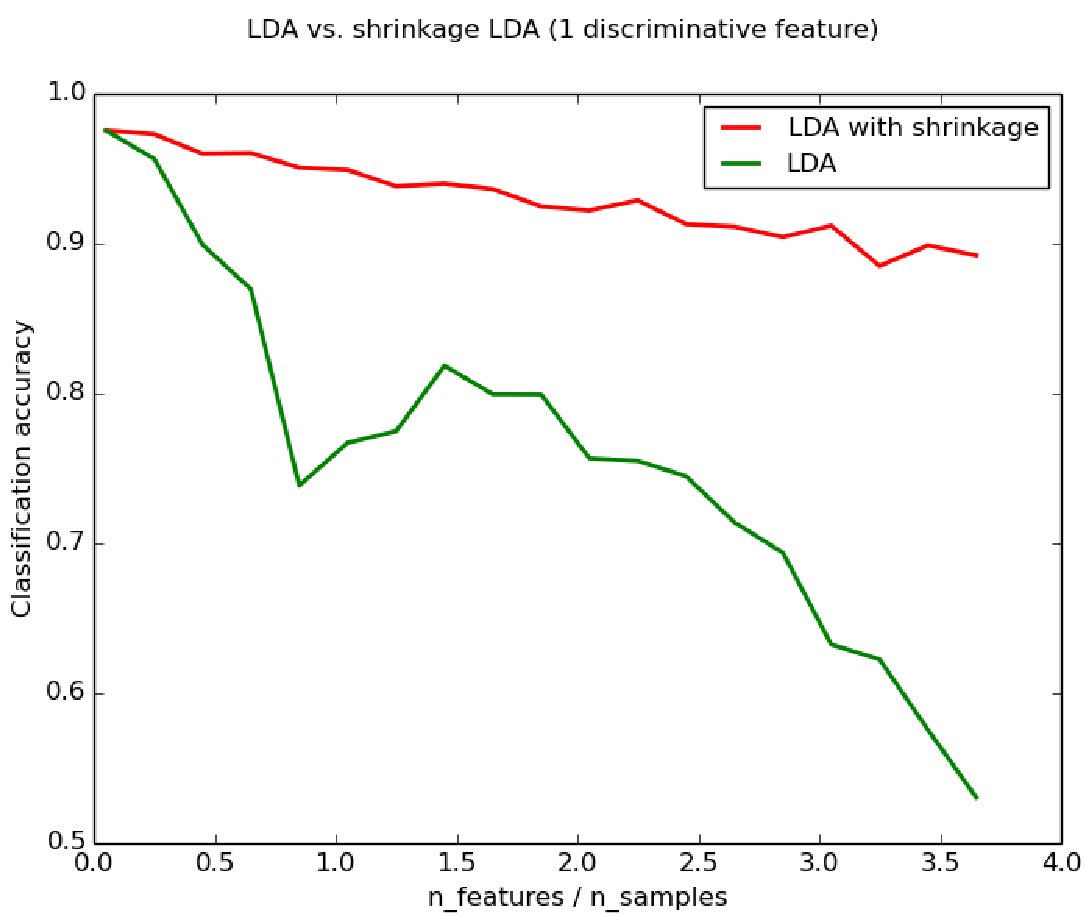


Figure 11.31: *Normal and Shrinkage Linear Discriminant Analysis for classification*



Python source code: plot_lda.py

```
from __future__ import division

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs
from sklearn.lda import LDA


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]])

    # 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 = LDA(solver='lsqr', shrinkage='auto').fit(X, y)
        clf2 = LDA(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="LDA with shrinkage", color='r')
plt.plot(features_samples_ratio, acc_clf2, linewidth=2,
         label="LDA", color='g')

plt.xlabel('n_features / n_samples')
```

```
plt.ylabel('Classification accuracy')

plt.legend(loc=1, prop={'size': 12})
plt.suptitle('LDA vs. shrinkage LDA (1 discriminative feature)')
plt.show()
```

Total running time of the example: 4.58 seconds (0 minutes 4.58 seconds)

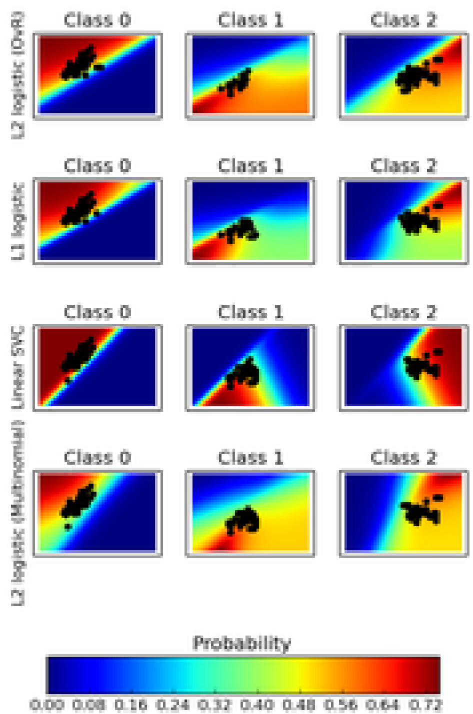
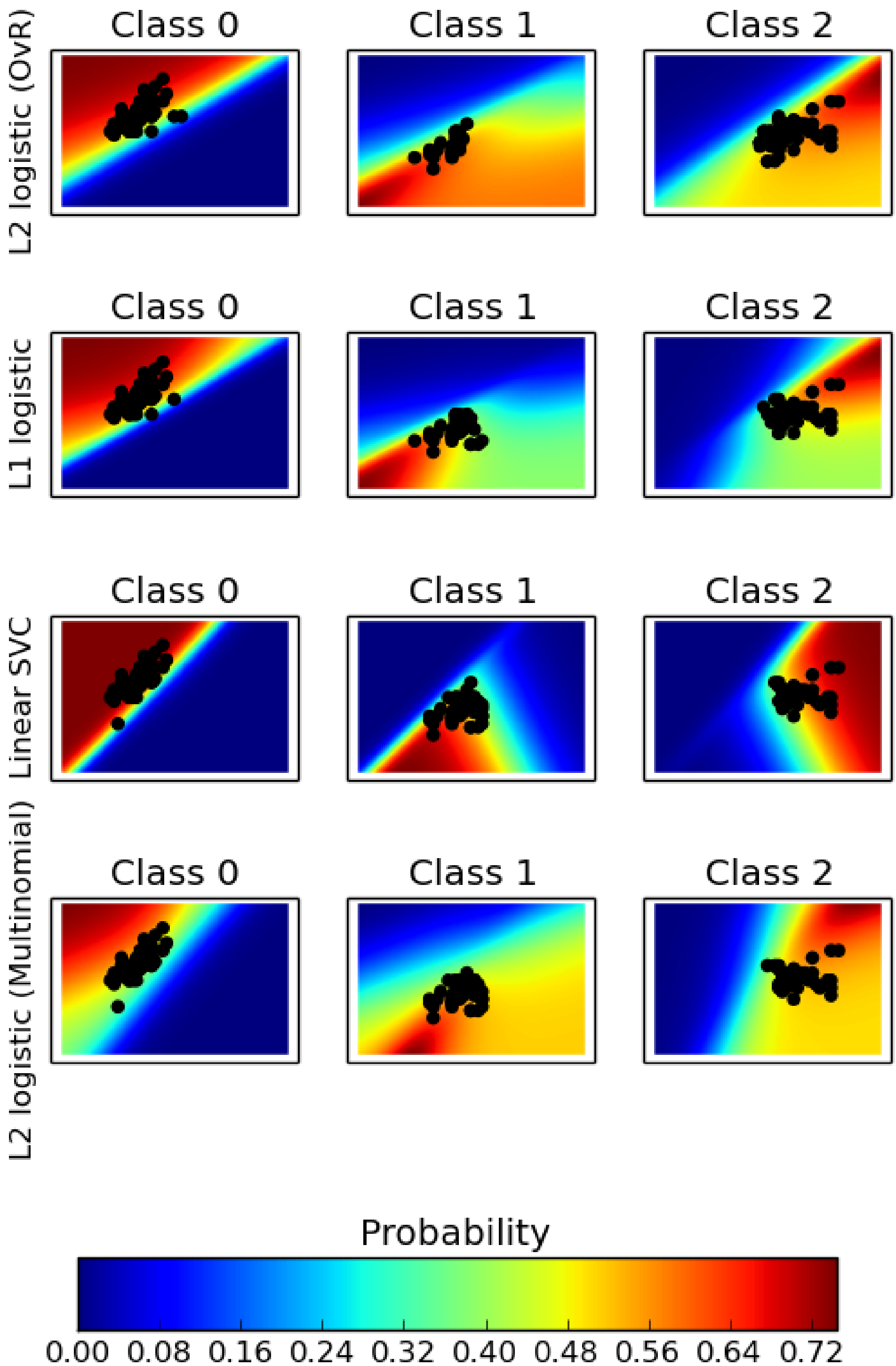


Figure 11.32: *Plot classification probability*

11.5.3 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.

The logistic regression is not a multiclass classifier out of the box. As a result it can identify only the first class.



Script output:

```
classif_rate for L2 logistic (OvR) : 76.666667
classif_rate for L1 logistic : 79.333333
classif_rate for Linear SVC : 82.000000
classif_rate for L2 logistic (Multinomial) : 82.000000
```

Python source code: plot_classification_probability.py

```
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 LogisticRegression
from sklearn.svm import SVC
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 = 1.0

# Create different classifiers. The logistic regression cannot do
# multiclass out of the box.
classifiers = {'L1 logistic': LogisticRegression(C=C, penalty='l1'),
               'L2 logistic (OvR)': LogisticRegression(C=C, penalty='l2'),
               'Linear SVC': SVC(kernel='linear', C=C, probability=True,
                                 random_state=0),
               'L2 logistic (Multinomial)': LogisticRegression(
                C=C, solver='lbfgs', multi_class='multinomial'
                )}

n_classifiers = len(classifiers)

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)
    classif_rate = np.mean(y_pred.ravel() == y.ravel()) * 100
    print("classif_rate for %s : %f " % (name, classif_rate))

    # 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='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 example: 0.44 seconds (0 minutes 0.44 seconds)

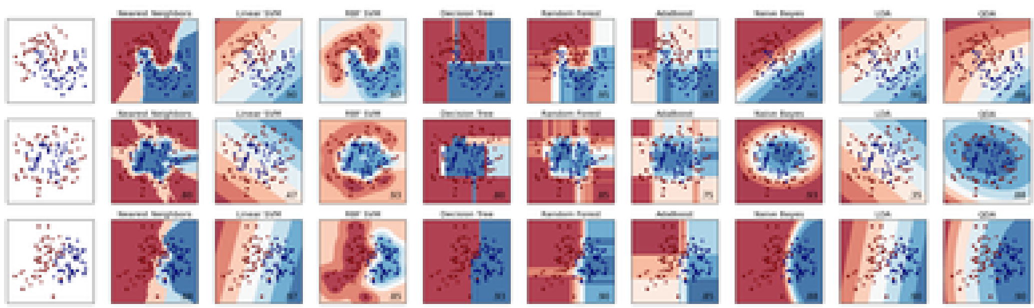


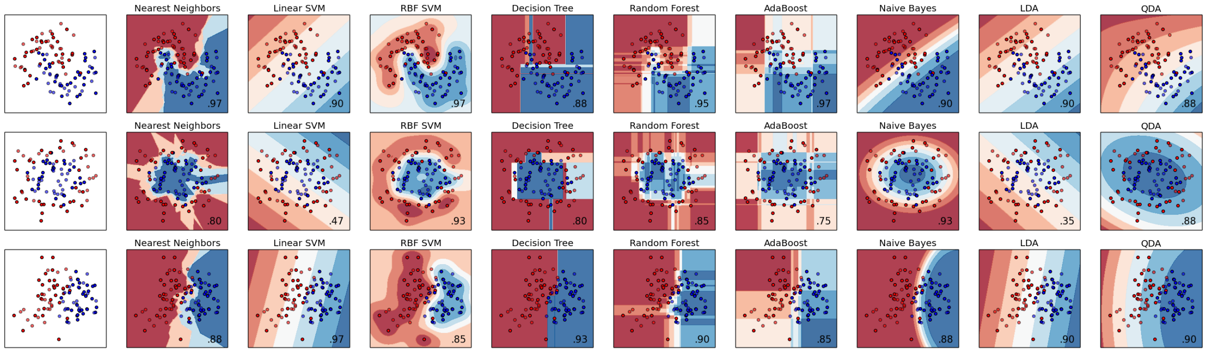
Figure 11.33: *Classifier comparison*

11.5.4 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.



Python source code: `plot_classifier_comparison.py`

```
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.cross_validation import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.lda import LDA
from sklearn.qda import QDA

h = .02  # step size in the mesh

names = ["Nearest Neighbors", "Linear SVM", "RBF SVM", "Decision Tree",
         "Random Forest", "AdaBoost", "Naive Bayes", "LDA", "QDA"]
classifiers = [
    KNeighborsClassifier(3),
    SVC(kernel="linear", C=0.025),
    SVC(gamma=2, C=1),
    DecisionTreeClassifier(max_depth=5),
    RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1),
    AdaBoostClassifier(),
    GaussianNB(),
    LDA(),
    QDA()]

X, y = make_classification(n_features=2, n_redundant=0, n_informative=2,
                           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 in 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)

    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, m_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)
        # 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(())
        ax.set_title(name)
        ax.text(xx.max() - .3, yy.min() + .3, ('%.2f' % score).lstrip('0'),
                size=15, horizontalalignment='right')
        i += 1

figure.subplots_adjust(left=.02, right=.98)
plt.show()
```

Total running time of the example: 3.44 seconds (0 minutes 3.44 seconds)

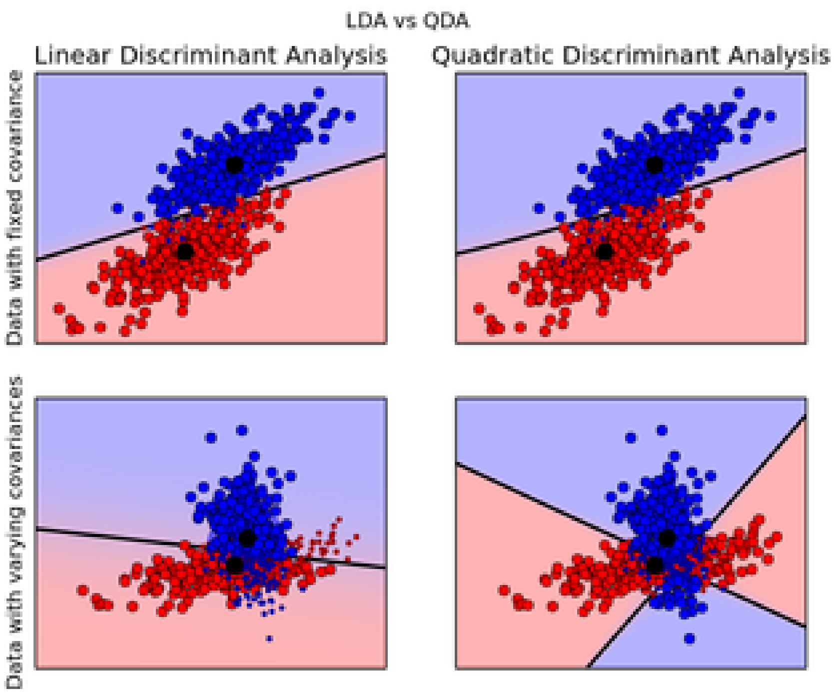
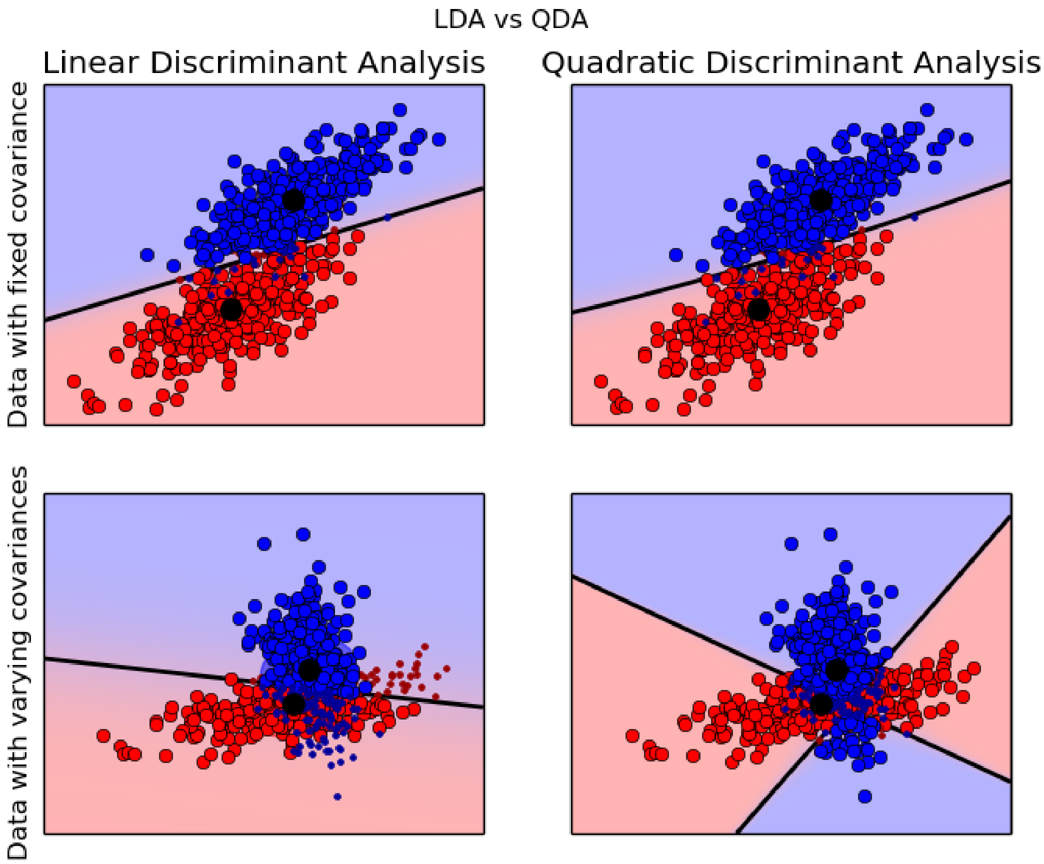


Figure 11.34: *Linear and Quadratic Discriminant Analysis with confidence ellipsoid*

11.5.5 Linear and Quadratic Discriminant Analysis with confidence ellipsoid

Plot the confidence ellipsoids of each class and decision boundary



Python source code: plot_lda_qda.py

```
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.lda import LDA
from sklearn.qda import QDA

###############################################################################
# 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
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 fixed covariance')
    elif fig_index == 2:
        plt.title('Quadratic Discriminant Analysis')
    elif fig_index == 3:
        plt.ylabel('Data with 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]
    xmin, xmax = X[:, 0].min(), X[:, 0].max()
    ymin, ymax = X[:, 1].min(), X[:, 1].max()

    # class 0: dots
    plt.plot(X0_tp[:, 0], X0_tp[:, 1], 'o', color='red')
    plt.plot(X0_fp[:, 0], X0_fp[:, 1], '.', color='#990000')  # dark red

    # class 1: dots
    plt.plot(X1_tp[:, 0], X1_tp[:, 1], 'o', color='blue')
    plt.plot(X1_fp[:, 0], X1_fp[:, 1], '.', 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()])
    Z = Z[:, 1].reshape(xx.shape)
    plt.pcolormesh(xx, yy, Z, cmap='red_blue_classes',
```

```
                   norm=colors.Normalize(0., 1.))
    plt.contour(xx, yy, Z, [0.5], linewidths=2., colors='k')

    # means
    plt.plot(lda.means_[0][0], lda.means_[0][1],
             'o', color='black', markersize=10)
    plt.plot(lda.means_[1][0], lda.means_[1][1],
             'o', color='black', markersize=10)

    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, color=color)
    ell.set_clip_box(splot.bbox)
    ell.set_alpha(0.5)
    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.covariances_[0], 'red')
    plot_ellipse(splot, qda.means_[1], qda.covariances_[1], 'blue')

###############################################################################
for i, (X, y) in enumerate([dataset_fixed_cov(), dataset_cov()]):
    # LDA
    lda = LDA(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')

    # QDA
    qda = QDA()
    y_pred = qda.fit(X, y, store_covariances=True).predict(X)
    splot = plot_data(qda, X, y, y_pred, fig_index=2 * i + 2)
    plot_qda_cov(qda, splot)
    plt.axis('tight')
plt.suptitle('LDA vs QDA')
plt.show()
```

Total running time of the example: 0.26 seconds (0 minutes 0.26 seconds)

11.6 Clustering

Examples concerning the `sklearn.cluster` module.

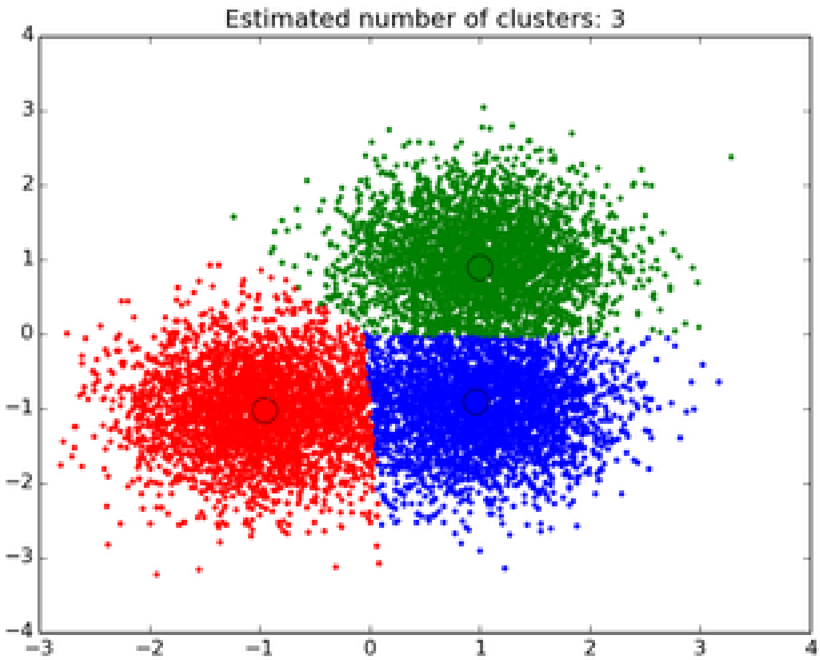
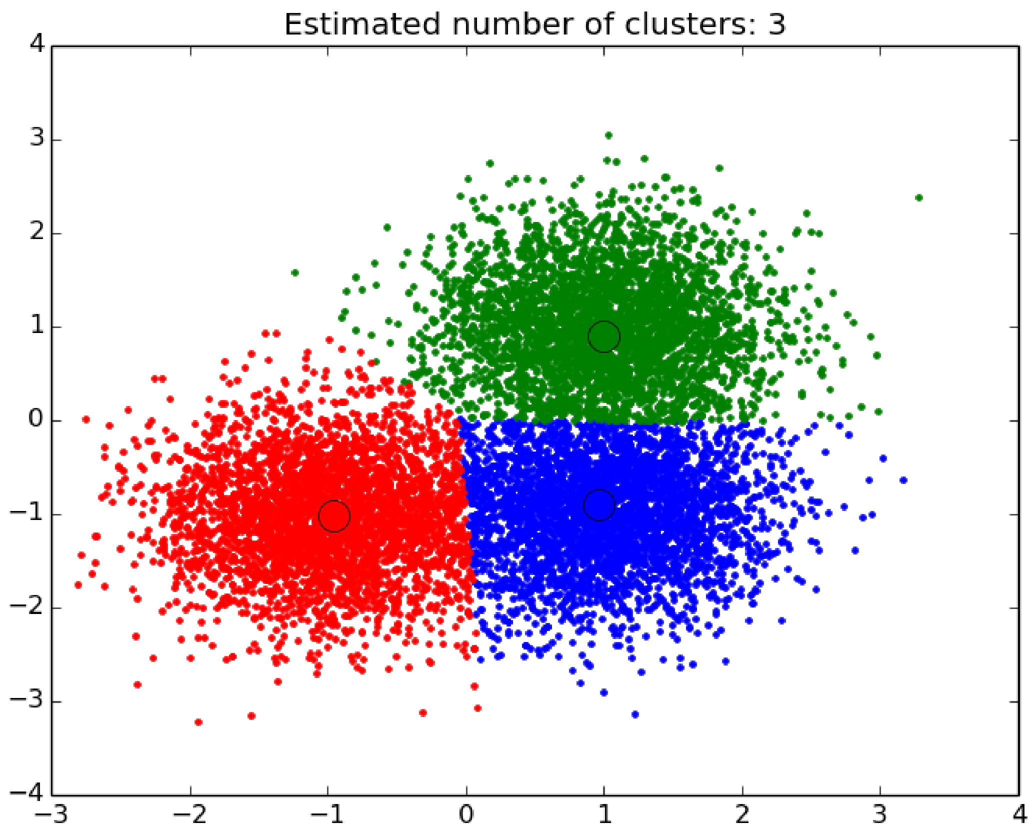


Figure 11.35: *A demo of the mean-shift clustering algorithm*

11.6.1 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.



Script output:

```
number of estimated clusters : 3
```

Python source code: `plot_mean_shift.py`

```
print(__doc__)

import numpy as np
from sklearn.cluster import MeanShift, estimate_bandwidth
from sklearn.datasets.samples_generator 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)

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 example: 0.35 seconds (0 minutes 0.35 seconds)

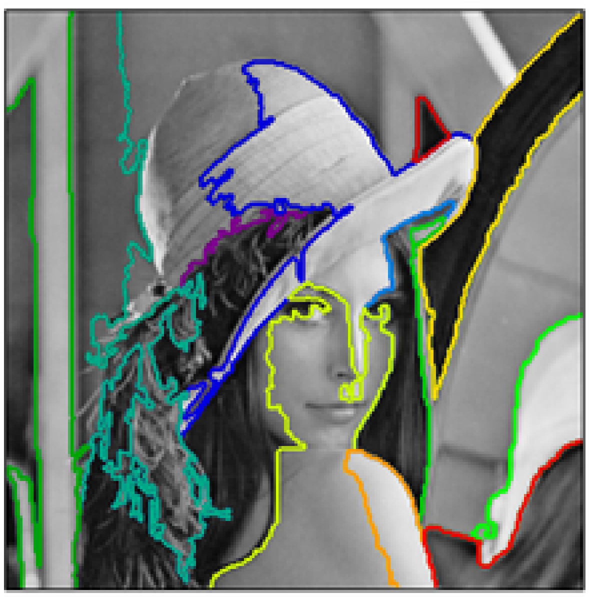
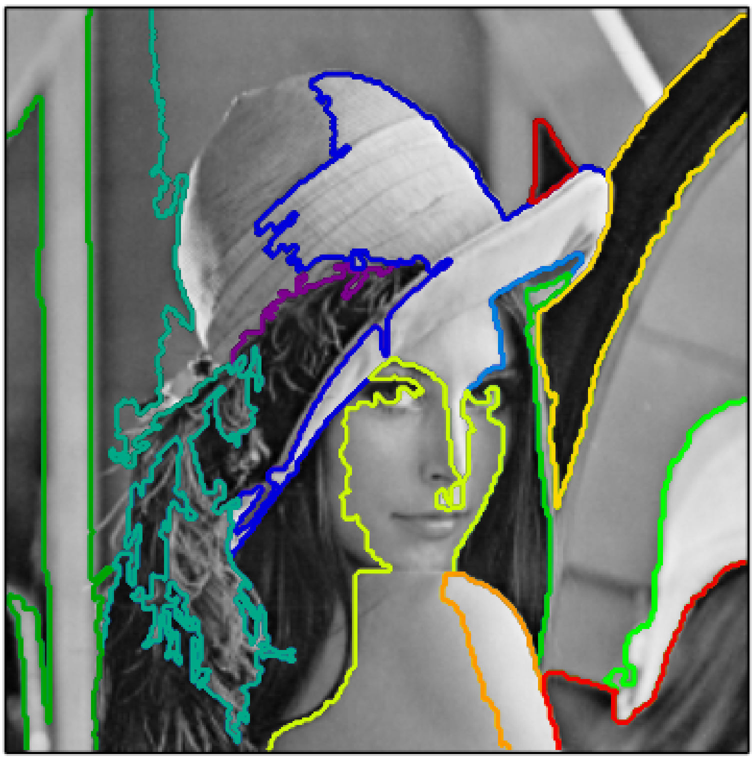


Figure 11.36: *A demo of structured Ward hierarchical clustering on Lena image*

11.6.2 A demo of structured Ward hierarchical clustering on Lena image

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.



Script output:

```
Compute structured hierarchical clustering...
Elapsed time:  4.84303092957
Number of pixels:  65536
Number of clusters:  15
```

Python source code: `plot_lena_ward_segmentation.py`

```python
# Author : Vincent Michel, 2010
#          Alexandre Gramfort, 2011
# License: BSD 3 clause

print(__doc__)

import time as time
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from sklearn.feature_extraction.image import grid_to_graph
from sklearn.cluster import AgglomerativeClustering
```

```
###############################################################################
# Generate data
lena = sp.misc.lena()
# Downsample the image by a factor of 4
lena = lena[::2, ::2] + lena[1::2, ::2] + lena[::2, 1::2] + lena[1::2, 1::2]
X = np.reshape(lena, (-1, 1))

###############################################################################
# Define the structure A of the data. Pixels connected to their neighbors.
connectivity = grid_to_graph(*lena.shape)

###############################################################################
# Compute clustering
print("Compute structured hierarchical clustering...")
st = time.time()
n_clusters = 15  # number of regions
ward = AgglomerativeClustering(n_clusters=n_clusters,
        linkage='ward', connectivity=connectivity).fit(X)
label = np.reshape(ward.labels_, lena.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(lena, cmap=plt.cm.gray)
for l in range(n_clusters):
    plt.contour(label == l, contours=1,
                colors=[plt.cm.spectral(l / float(n_clusters)), ])
plt.xticks(())
plt.yticks(())
plt.show()
```

Total running time of the example: 5.39 seconds (0 minutes 5.39 seconds)

11.6.3 Feature agglomeration

These images how similar features are merged together using feature agglomeration.

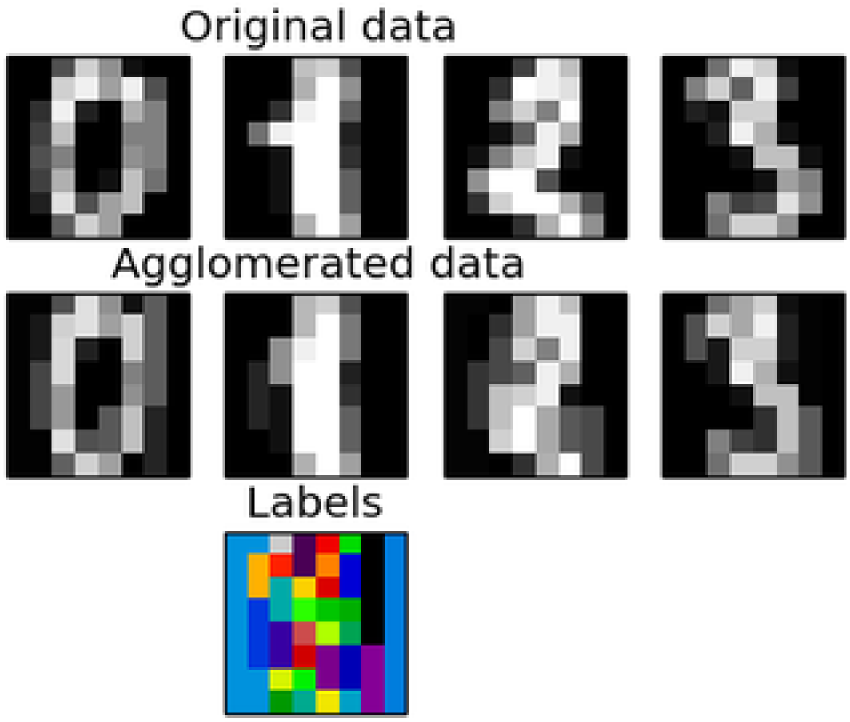
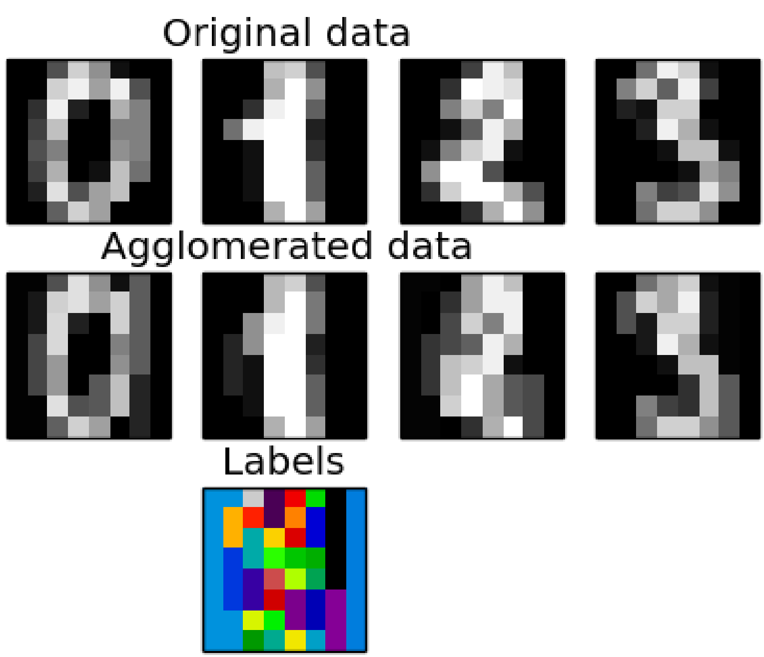


Figure 11.37: *Feature agglomeration*



Python source code: `plot_digits_agglomeration.py`

```
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:
        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.spectral)
plt.xticks(())
plt.yticks(())
plt.title('Labels')
plt.show()
```

Total running time of the example: 0.33 seconds (0 minutes 0.33 seconds)

11.6.4 Demo of affinity propagation clustering algorithm

Reference: Brendan J. Frey and Delbert Dueck, "Clustering by Passing Messages Between Data Points", Science Feb. 2007

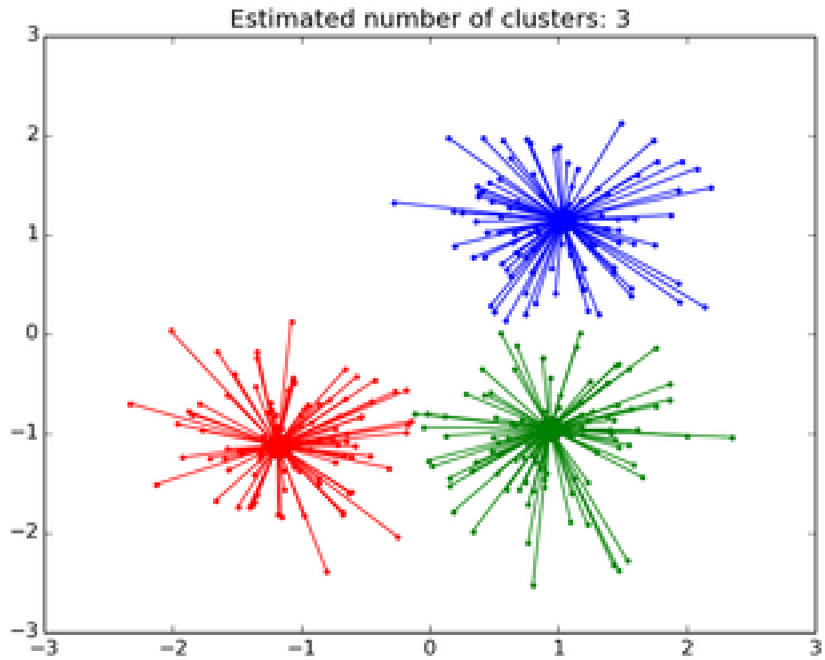
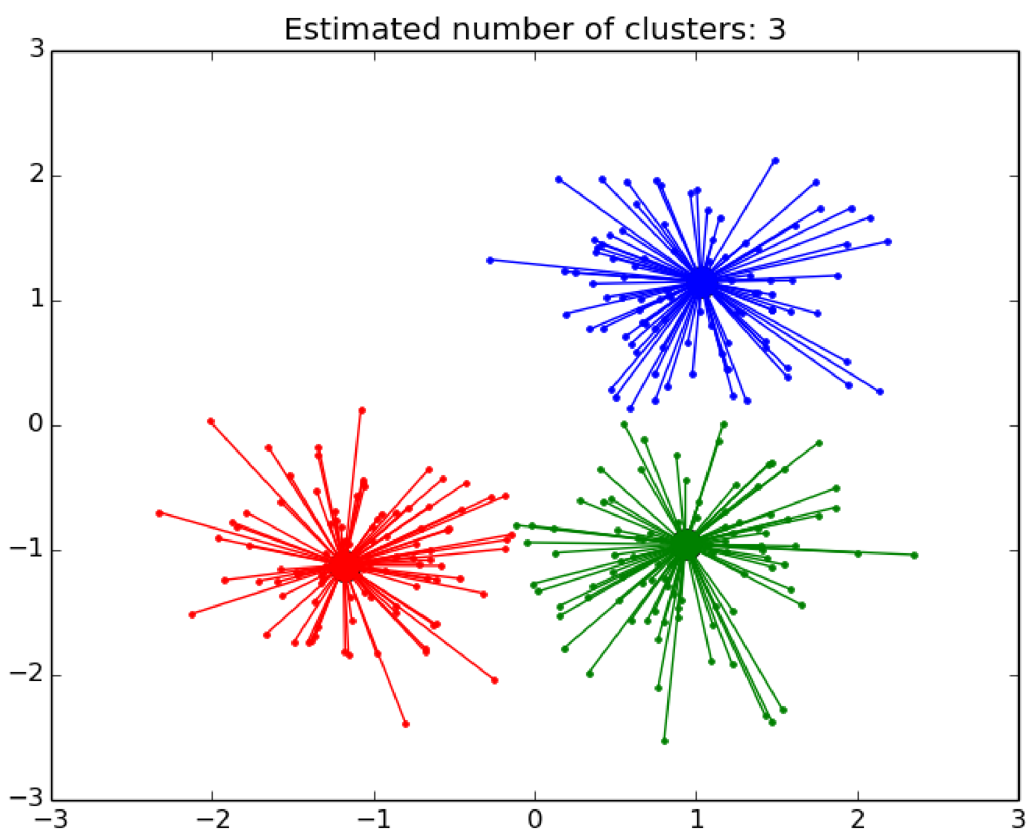


Figure 11.38: *Demo of affinity propagation clustering algorithm*



Script output:

```
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
```

Python source code: `plot_affinity_propagation.py`

```python
print(__doc__)

from sklearn.cluster import AffinityPropagation
from sklearn import metrics
from sklearn.datasets.samples_generator 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 example: 0.62 seconds (0 minutes 0.62 seconds)

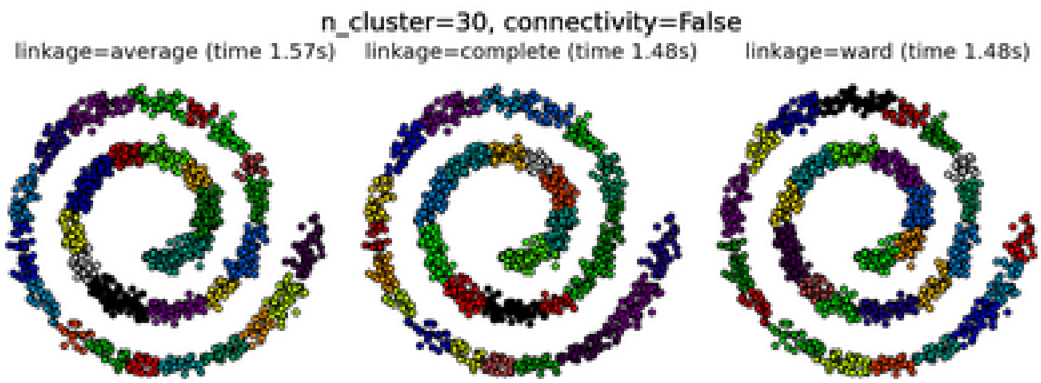


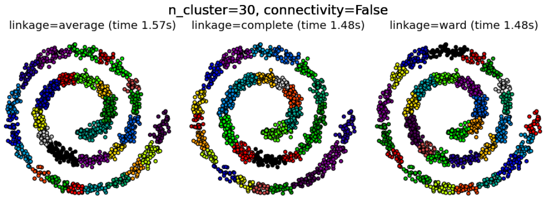
Figure 11.39: *Agglomerative clustering with and without structure*

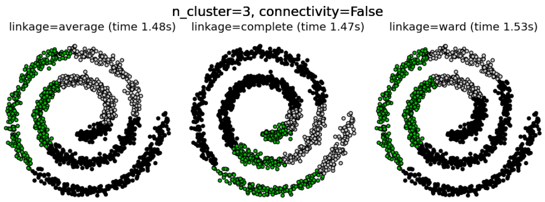
11.6.5 Agglomerative clustering with and without structure

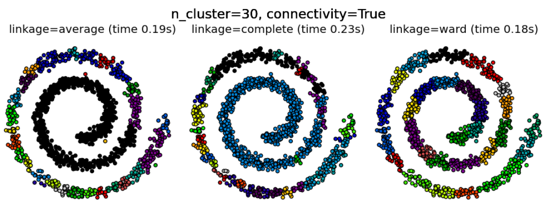
This example shows the effect of imposing a connectivity graph to capture local structure in the data. The graph is simply the graph of 20 nearest neighbors.

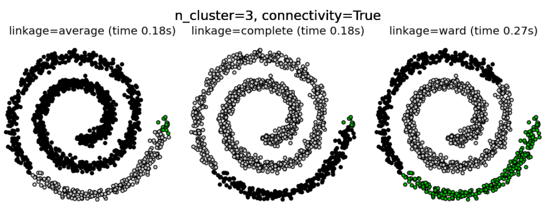
Two consequences of imposing a connectivity can be seen. First clustering with a connectivity matrix is much faster.

Second, when using a connectivity matrix, 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. The connectivity graph breaks this mechanism. 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.

- 

- 

- 

- 

Python source code: `plot_agglomerative_clustering.py`

```python
# 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
```

```python
# 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')):
            plt.subplot(1, 3, 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.spectral)
            plt.title('linkage=%s (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 example: 10.71 seconds (0 minutes 10.71 seconds)

11.6.6 Segmenting the picture of Lena 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.

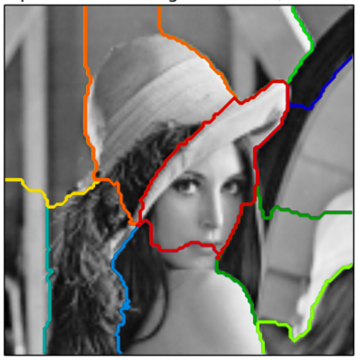
This procedure (spectral clustering on an image) is an efficient approximate solution for finding normalized graph cuts.

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.

Figure 11.40: *Segmenting the picture of Lena in regions*

•



•

Python source code: `plot_lena_segmentation.py`

```
print(__doc__)

# Author: Gael Varoquaux <gael.varoquaux@normalesup.org>, Brian Cheung
# License: BSD 3 clause

import time

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

from sklearn.feature_extraction import image
from sklearn.cluster import spectral_clustering

lena = sp.misc.lena()
# Downsample the image by a factor of 4
lena = lena[::2, ::2] + lena[1::2, ::2] + lena[::2, 1::2] + lena[1::2, 1::2]
lena = lena[::2, ::2] + lena[1::2, ::2] + lena[::2, 1::2] + lena[1::2, 1::2]

# Convert the image into a graph with the value of the gradient on the
# edges.
graph = image.img_to_graph(lena)

# 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 = 5
eps = 1e-6
graph.data = np.exp(-beta * graph.data / lena.std()) + eps

# Apply spectral clustering (this step goes much faster if you have pyamg
# installed)
N_REGIONS = 11

###############################################################################
# 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=1)
    t1 = time.time()
    labels = labels.reshape(lena.shape)

    plt.figure(figsize=(5, 5))
    plt.imshow(lena,   cmap=plt.cm.gray)
    for l in range(N_REGIONS):
        plt.contour(labels == l, contours=1,
                    colors=[plt.cm.spectral(l / float(N_REGIONS)), ])
    plt.xticks(())
    plt.yticks(())
    plt.title('Spectral clustering: %s, %.2fs' % (assign_labels, (t1 - t0)))

plt.show()
```

Total running time of the example: 51.05 seconds (0 minutes 51.05 seconds)

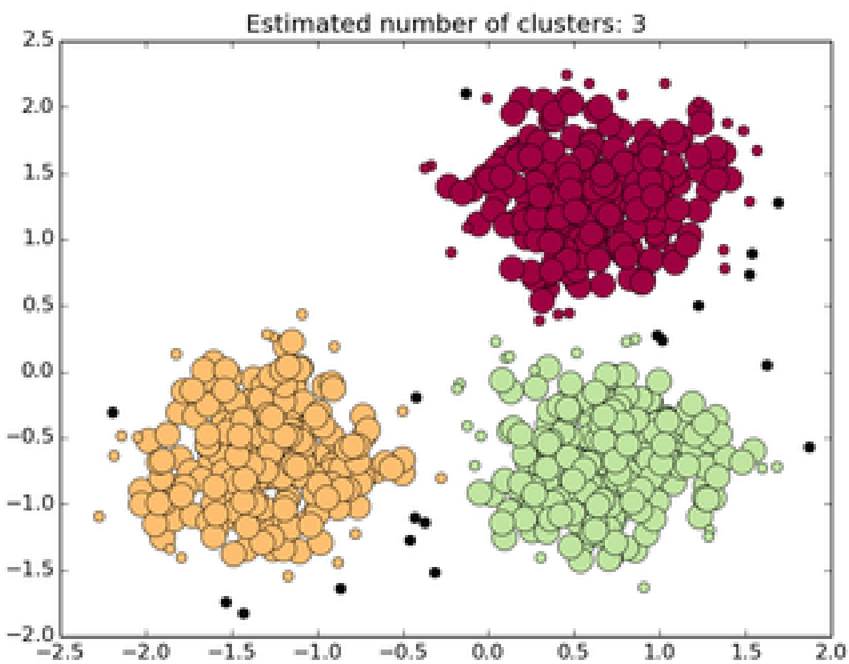
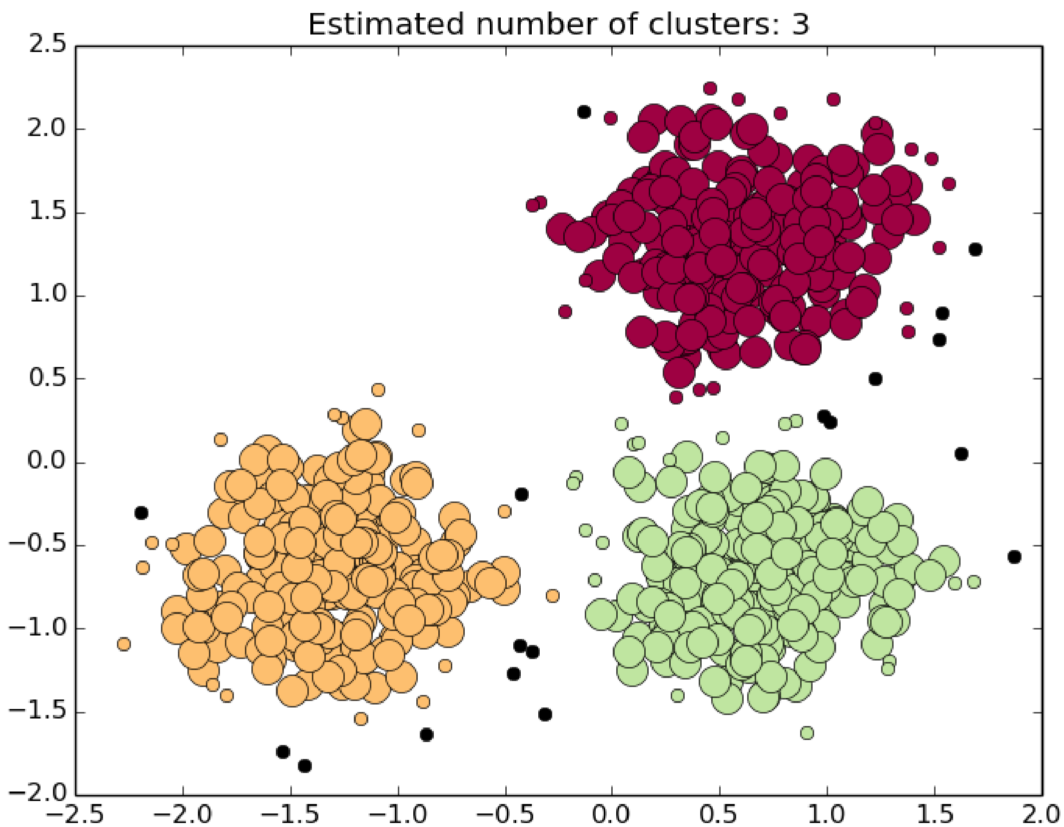


Figure 11.41: *Demo of DBSCAN clustering algorithm*

11.6.7 Demo of DBSCAN clustering algorithm

Finds core samples of high density and expands clusters from them.



Script output:

```
Estimated number of clusters: 3
Homogeneity: 0.953
Completeness: 0.883
V-measure: 0.917
Adjusted Rand Index: 0.952
Adjusted Mutual Information: 0.883
Silhouette Coefficient: 0.626
```

Python source code: `plot_dbscan.py`

```python
print(__doc__)

import numpy as np

from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.datasets.samples_generator 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)
```

```python
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)

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))

##############################################################################
# Plot result
import matplotlib.pyplot as plt

# Black removed and is used for noise instead.
unique_labels = set(labels)
colors = plt.cm.Spectral(np.linspace(0, 1, len(unique_labels)))
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = 'k'

    class_member_mask = (labels == k)

    xy = X[class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=col,
             markeredgecolor='k', markersize=14)

    xy = X[class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=col,
             markeredgecolor='k', markersize=6)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

Total running time of the example: 0.27 seconds (0 minutes 0.27 seconds)

11.6.8 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

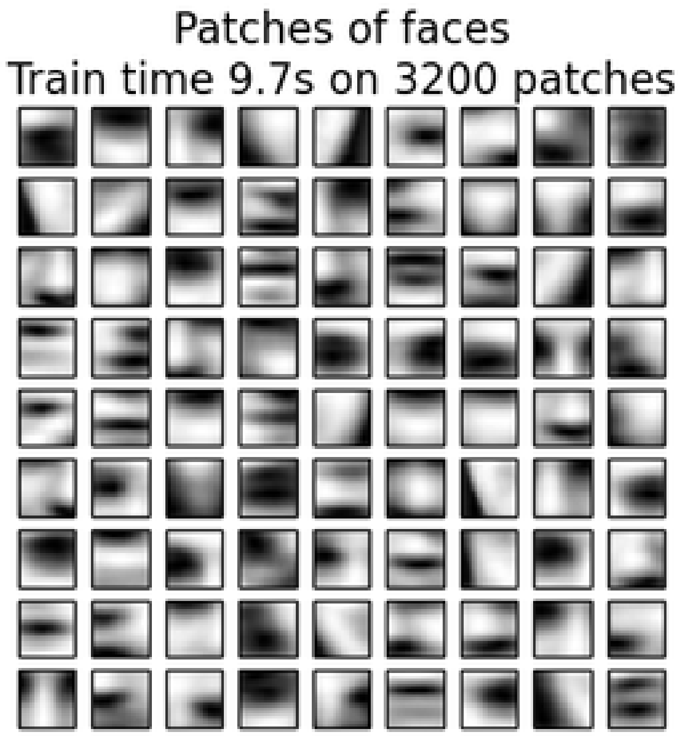
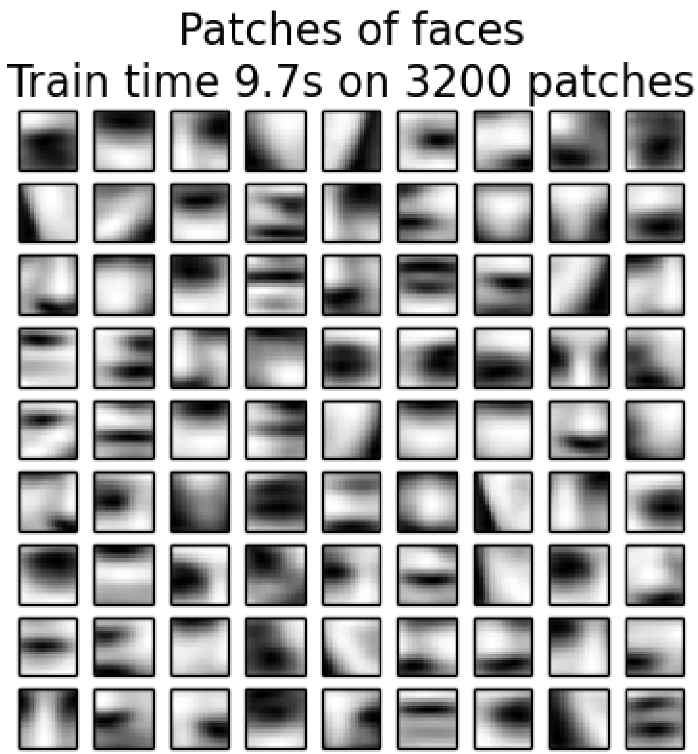


Figure 11.42: *Online learning of a dictionary of parts of faces*

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.



Script output:

```
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
Partial fit of 2400 out of 2400
done in 9.68s.
```

Python source code: `plot_dict_face_patches.py`

```
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 = []
index = 1
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)

###############################################################################
# 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 example: 12.63 seconds (0 minutes 12.63 seconds)

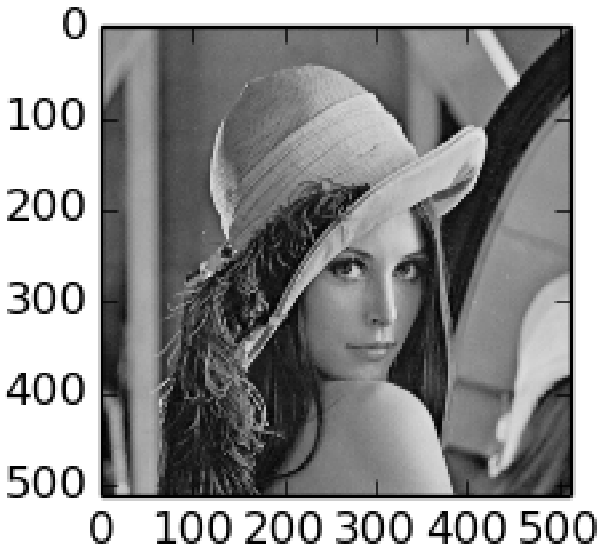
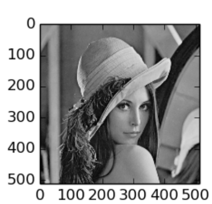
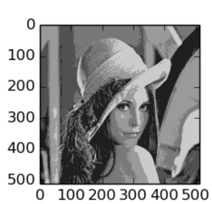


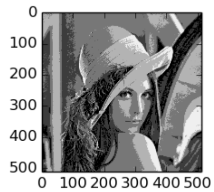
Figure 11.43: *Vector Quantization Example*

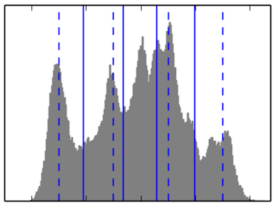
11.6.9 Vector Quantization Example

The classic image processing example, Lena, an 8-bit grayscale bit-depth, 512 x 512 sized image, is used here to illustrate how *k*-means is used for vector quantization.

- 

- 

- 

- 

Python source code: `plot_lena_compress.py`

```
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

n_clusters = 5
np.random.seed(0)

try:
    lena = sp.lena()
except AttributeError:
    # Newer versions of scipy have lena in misc
    from scipy import misc
    lena = misc.lena()
X = lena.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
lena_compressed = np.choose(labels, values)
lena_compressed.shape = lena.shape

vmin = lena.min()
vmax = lena.max()

# original lena
plt.figure(1, figsize=(3, 2.2))
plt.imshow(lena, cmap=plt.cm.gray, vmin=vmin, vmax=256)

# compressed lena
plt.figure(2, figsize=(3, 2.2))
plt.imshow(lena_compressed, cmap=plt.cm.gray, vmin=vmin, vmax=vmax)
```

```
# equal bins lena
regular_values = np.linspace(0, 256, n_clusters + 1)
regular_labels = np.searchsorted(regular_values, lena) - 1
regular_values = .5 * (regular_values[1:] + regular_values[:-1])  # mean
regular_lena = np.choose(regular_labels.ravel(), regular_values)
regular_lena.shape = lena.shape
plt.figure(3, figsize=(3, 2.2))
plt.imshow(regular_lena, 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 example: 1.51 seconds (0 minutes 1.51 seconds)

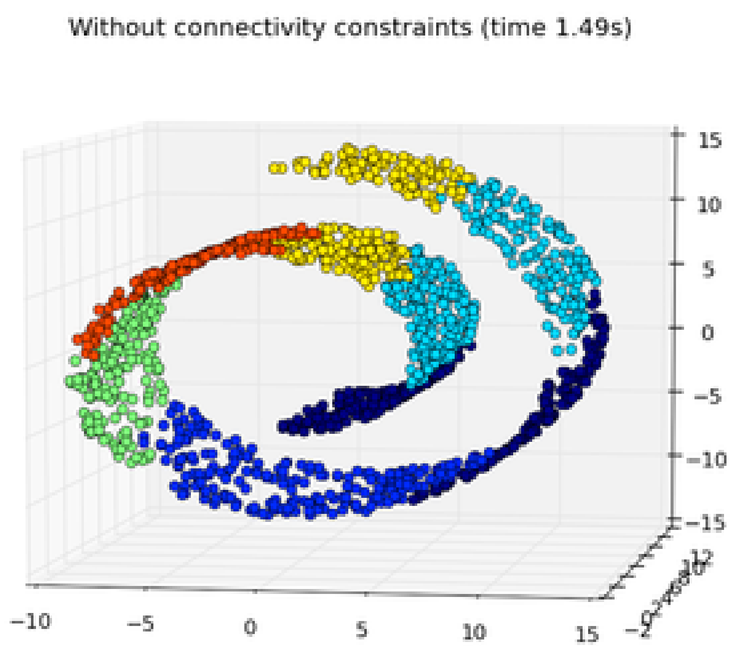


Figure 11.44: *Hierarchical clustering: structured vs unstructured ward*

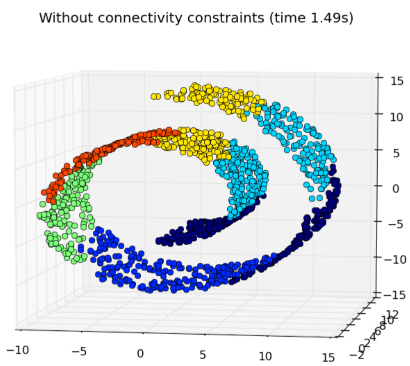
11.6.10 Hierarchical clustering: structured vs unstructured ward

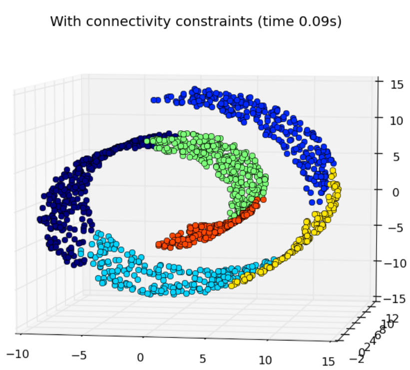
Example builds a swiss roll dataset and runs hierarchical clustering on their position.

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.

- 

- 

Script output:

```
Compute unstructured hierarchical clustering...
Elapsed time: 1.49s
Number of points: 1500
Compute structured hierarchical clustering...
Elapsed time: 0.09s
Number of points: 1500
```

Python source code: `plot_ward_structured_vs_unstructured.py`

```
# 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.samples_generator import make_swiss_roll

###############################################################################
# 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.plot3D(X[label == l, 0], X[label == l, 1], X[label == l, 2],
              'o', color=plt.cm.jet(np.float(l) / np.max(label + 1)))
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.plot3D(X[label == l, 0], X[label == l, 1], X[label == l, 2],
              'o', color=plt.cm.jet(float(l) / np.max(label + 1)))
plt.title('With connectivity constraints (time %.2fs)' % elapsed_time)

plt.show()
```

Total running time of the example: 1.87 seconds (0 minutes 1.87 seconds)

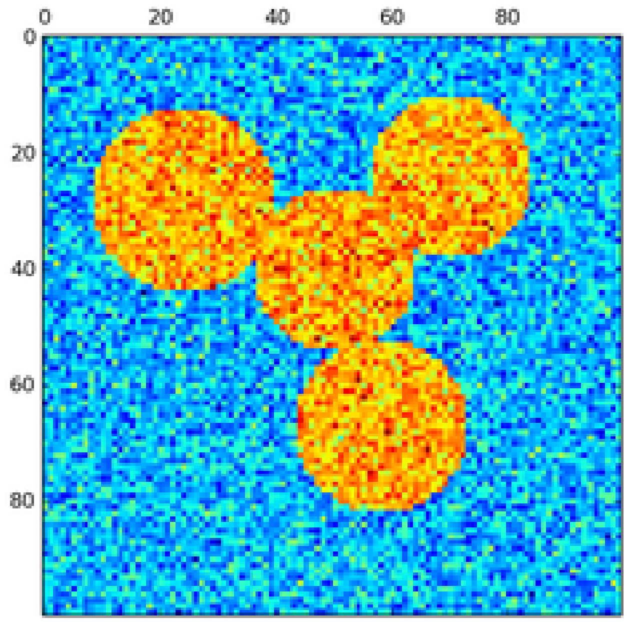


Figure 11.45: *Spectral clustering for image segmentation*

11.6.11 Spectral clustering for image segmentation

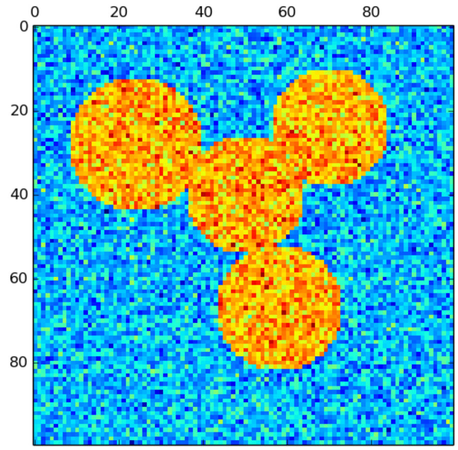
In this example, an image with connected circles is generated and spectral clustering is used to separate the circles.

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.

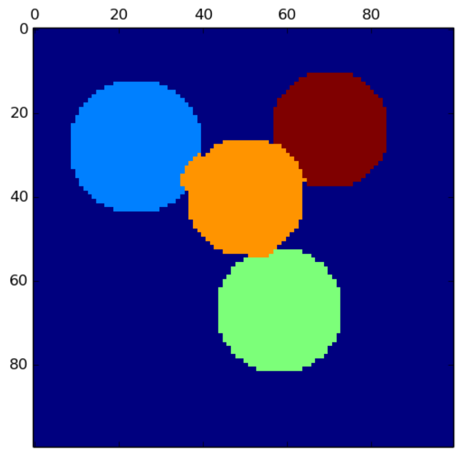
As the algorithm tries to balance the volume (ie balance the region sizes), if we take circles with different sizes, the segmentation fails.

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.

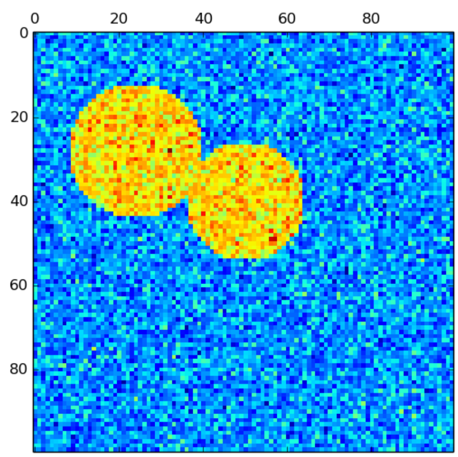
In addition, we use the mask of the objects to restrict the graph to the outline of the objects. In this example, we are interested in separating the objects one from the other, and not from the background.


0
20
40
60
80

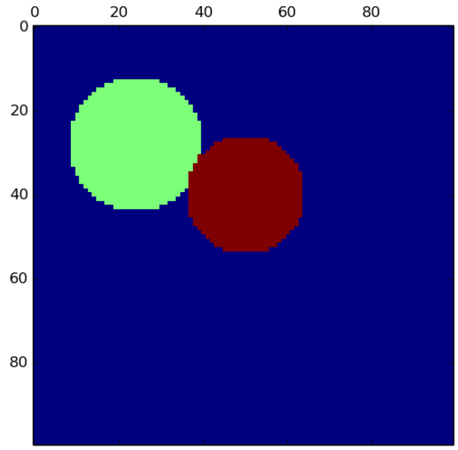
-


0
20
40
60
80

-


0
20
40
60
80

-



•

Python source code: `plot_segmentation_toy.py`

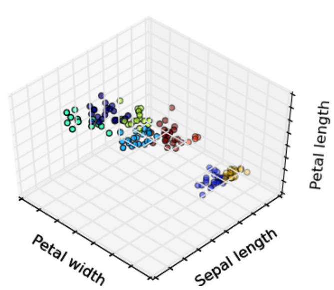
```
print(__doc__)

# 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
circle4 = (x - center4[0]) ** 2 + (y - center4[1]) ** 2 < radius4 ** 2

###############################################################################
# 4 circles
img = circle1 + circle2 + circle3 + circle4
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.ones(mask.shape)
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.ones(mask.shape)
label_im[mask] = labels

plt.matshow(img)
plt.matshow(label_im)

plt.show()
```

Total running time of the example: 0.81 seconds (0 minutes 0.81 seconds)

11.6.12 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.

- 

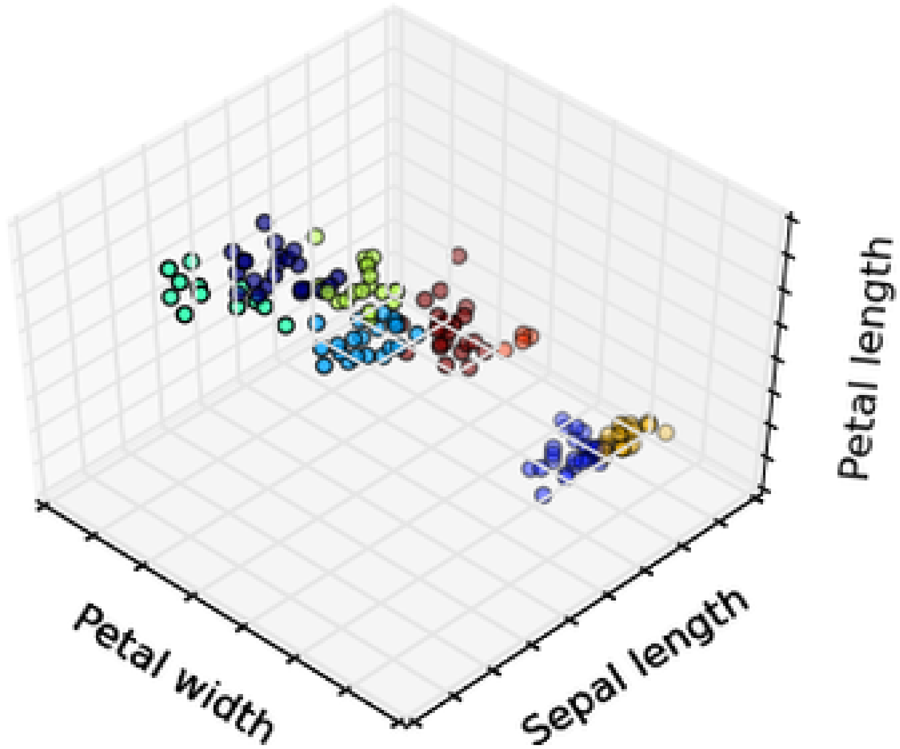
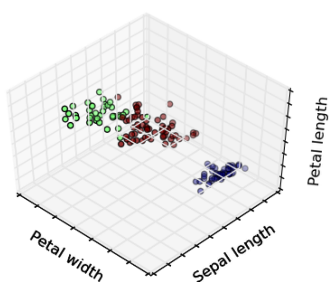
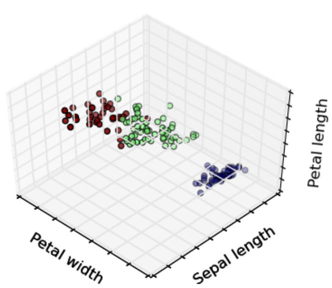
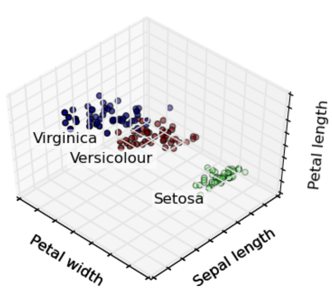


Figure 11.46: *K-means Clustering*

- 

- 

- 

Python source code: `plot_cluster_iris.py`

```
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 mpl_toolkits.mplot3d import Axes3D


from sklearn.cluster import KMeans
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

estimators = {'k_means_iris_3': KMeans(n_clusters=3),
              'k_means_iris_8': KMeans(n_clusters=8),
              'k_means_iris_bad_init': KMeans(n_clusters=3, n_init=1,
                                              init='random')}


fignum = 1
for name, est in estimators.items():
    fig = plt.figure(fignum, figsize=(4, 3))
    plt.clf()
    ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

    plt.cla()
    est.fit(X)
    labels = est.labels_

    ax.scatter(X[:, 3], X[:, 0], X[:, 2], c=labels.astype(np.float))

    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')
    fignum = fignum + 1

# Plot the ground truth
fig = plt.figure(fignum, figsize=(4, 3))
plt.clf()
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

plt.cla()

for name, label in [('Setosa', 0),
                    ('Versicolour', 1),
```

```
                    ('Virginica', 2)]:
    ax.text3D(X[y == label, 3].mean(),
              X[y == label, 0].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[:, 3], X[:, 0], X[:, 2], c=y)

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')
plt.show()
```

Total running time of the example: 0.25 seconds (0 minutes 0.25 seconds)

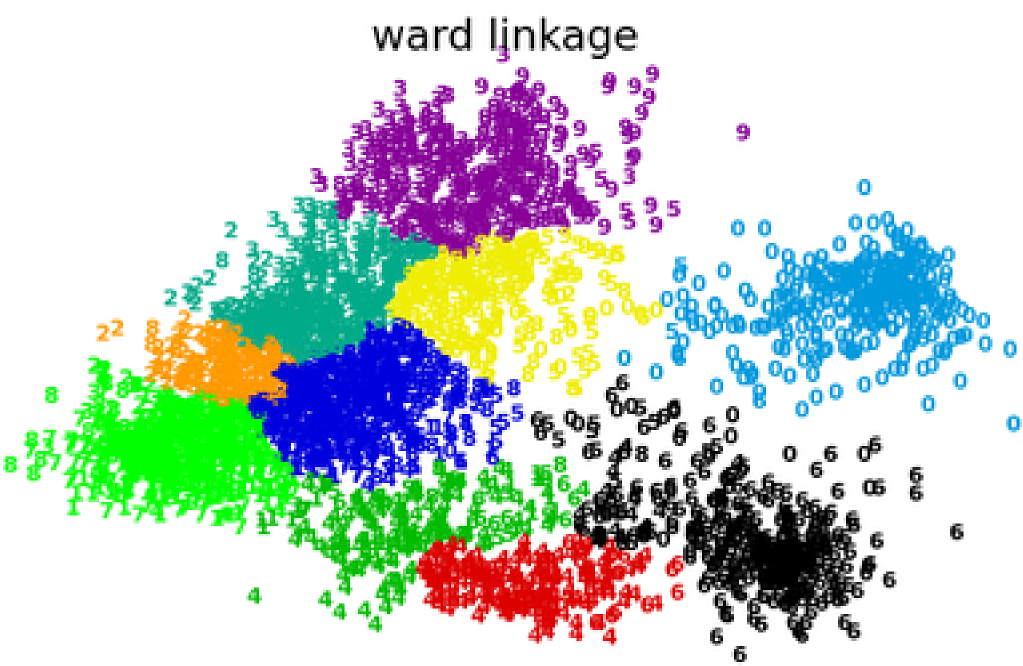


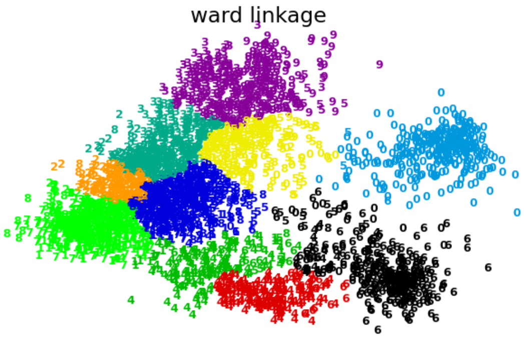
Figure 11.47: *Various Agglomerative Clustering on a 2D embedding of digits*

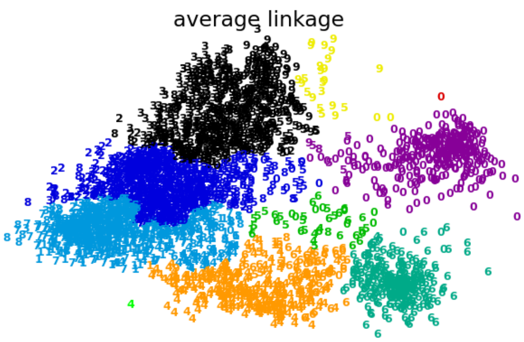
11.6.13 Various Agglomerative Clustering on a 2D embedding of digits

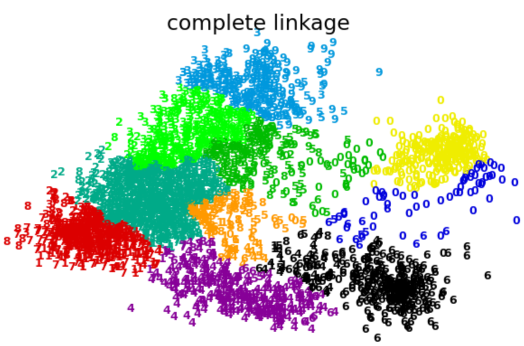
An illustration of various linkage option for agglomerative clustering on a 2D embedding of the digits dataset.

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 especially pronounced for the average linkage strategy, that ends up with a couple of singleton clusters.

- 

- 

- 

Script output:

```
Computing embedding
Done.
ward : 22.36s
average : 23.63s
complete : 22.78s
```

Python source code: `plot_digits_linkage.py`

```
# 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

digits = datasets.load_digits(n_class=10)
X = digits.data
y = digits.target
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, X, 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.spectral(labels[i] / 10.),
                 fontdict={'weight': 'bold', 'size': 9})

    plt.xticks([])
    plt.yticks([])
    if title is not None:
        plt.title(title, size=17)
    plt.axis('off')
    plt.tight_layout()

#----------------------------------------------------------------------
# 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'):
    clustering = AgglomerativeClustering(linkage=linkage, n_clusters=10)
```

```
    t0 = time()
    clustering.fit(X_red)
    print("%s : %.2fs" % (linkage, time() - t0))

    plot_clustering(X_red, X, clustering.labels_, "%s linkage" % linkage)


plt.show()
```

Total running time of the example: 84.48 seconds (1 minutes 24.48 seconds)

Figure 11.48: *Color Quantization using K-Means*

11.6.14 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.

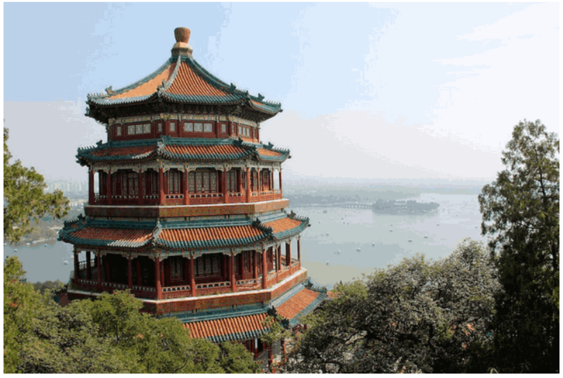
In this example, pixels are represented in a 3D-space and K-means is used to find 64 color clusters. In the image processing literature, the codebook obtained from K-means (the cluster centers) is called the color palette. Using a single byte, up to 256 colors can be addressed, whereas an RGB encoding 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)

- Quantized image (64 colors, Random)

Script output:

```
Fitting model on a small sub-sample of the data
done in 0.268s.
Predicting color indices on the full image (k-means)
done in 0.295s.
Predicting color indices on the full image (random)
done in 0.263s.
```

Python source code: `plot_color_quantization.py`

```
# Authors: Robert Layton <robertlayton@gmail.com>
#          Olivier Grisel <olivier.grisel@ensta.org>
#          Mathieu Blondel <mathieu@mblondel.org>
#
```

```python
# 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
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 + 1]
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()
ax = plt.axes([0, 0, 1, 1])
plt.axis('off')
plt.title('Original image (96,615 colors)')
plt.imshow(china)

plt.figure(2)
plt.clf()
ax = plt.axes([0, 0, 1, 1])
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()
ax = plt.axes([0, 0, 1, 1])
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 example: 1.90 seconds (0 minutes 1.90 seconds)

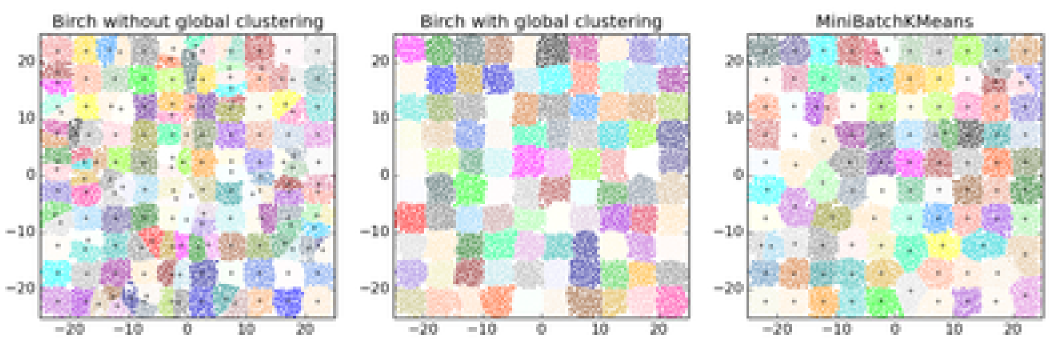
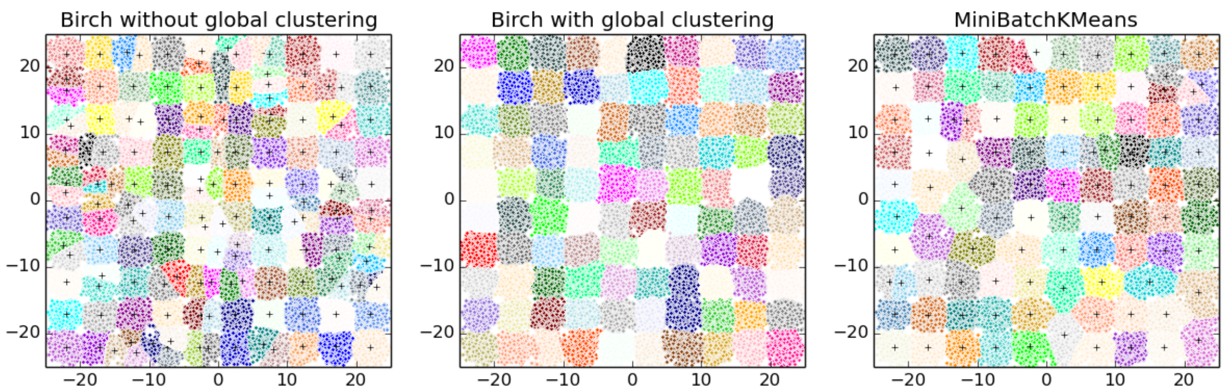


Figure 11.49: *Compare BIRCH and MiniBatchKMeans*

11.6.15 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.



Script output:

```
Birch without global clustering as the final step took 2.01 seconds
n_clusters : 158
Birch with global clustering as the final step took 2.05 seconds
n_clusters : 100
Time taken to run MiniBatchKMeans 2.83 seconds
```

Python source code: `plot_birch_vs_minibatchkmeans.py`

```
# 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.preprocessing import StandardScaler
from sklearn.cluster import Birch, MiniBatchKMeans
from sklearn.datasets.samples_generator 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)
```

```python
# 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.plot(X[mask, 0], X[mask, 1], 'w',
                markerfacecolor=col, marker='.')
        if birch_model.n_clusters is None:
            ax.plot(this_centroid[0], this_centroid[1], '+', markerfacecolor=col,
                    markeredgecolor='k', markersize=5)
    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)
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.plot(X[mask, 0], X[mask, 1], 'w', markerfacecolor=col, marker='.')
    ax.plot(this_centroid[0], this_centroid[1], '+', markeredgecolor='k',
            markersize=5)
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 example: 7.90 seconds (0 minutes 7.90 seconds)

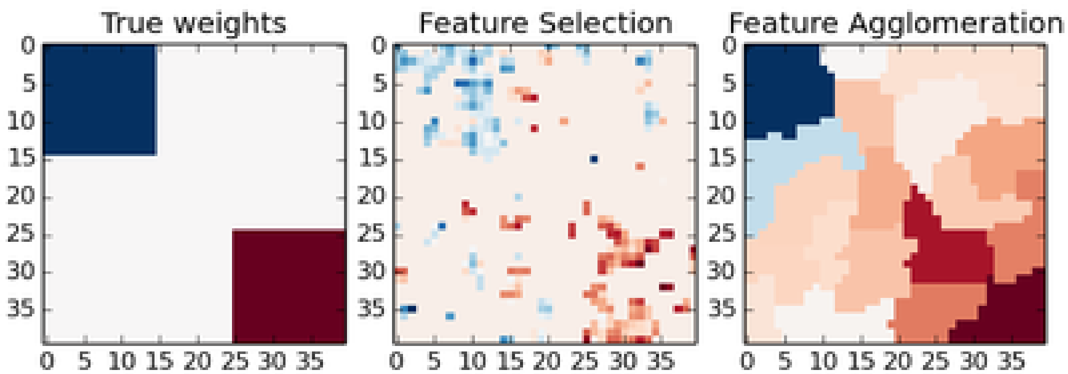


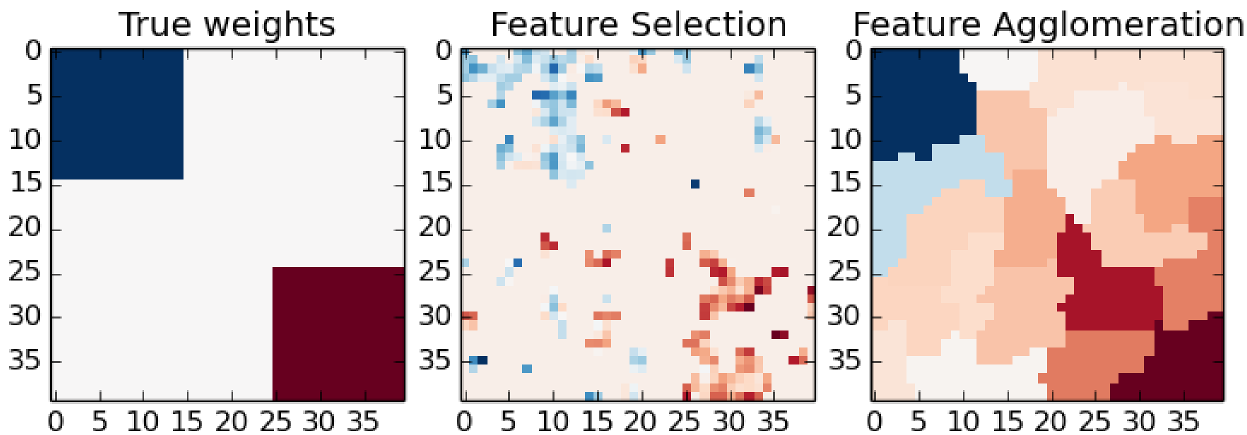
Figure 11.50: *Feature agglomeration vs. univariate selection*

11.6.16 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.



Script output:

```
________________________________________________________________________________
[Memory] Calling sklearn.cluster.hierarchical.ward_tree...
ward_tree(array([[-0.451933, ..., -0.675318],
       ...,
       [ 0.275706, ..., -1.085711]]),
<1600x1600 sparse matrix of type '<type 'numpy.int64'>'
        with 7840 stored elements in COOrdinate format>, n_components=None, n_clusters=None)
________________________________________________________ward_tree - 0.1s, 0.0min
________________________________________________________________________________
[Memory] Calling sklearn.cluster.hierarchical.ward_tree...
ward_tree(array([[ 0.905206, ...,  0.161245],
       ...,
       [-0.849835, ..., -1.091621]]),
<1600x1600 sparse matrix of type '<type 'numpy.int64'>'
        with 7840 stored elements in COOrdinate format>, n_components=None, n_clusters=None)
________________________________________________________ward_tree - 0.1s, 0.0min
________________________________________________________________________________
[Memory] Calling sklearn.cluster.hierarchical.ward_tree...
ward_tree(array([[ 0.905206, ..., -0.675318],
       ...,
       [-0.849835, ..., -1.085711]]),
<1600x1600 sparse matrix of type '<type 'numpy.int64'>'
        with 7840 stored elements in COOrdinate format>, n_components=None, n_clusters=None)
________________________________________________________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
```

Python source code: `plot_feature_agglomeration_vs_univariate_selection.py`

```
# 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 sklearn.feature_extraction.image import grid_to_graph
from sklearn import feature_selection
from sklearn.cluster import FeatureAgglomeration
from sklearn.linear_model import BayesianRidge
from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV
from sklearn.externals.joblib import Memory
from sklearn.cross_validation 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(len(y), 2)  # cross-validation generator for model selection
ridge = BayesianRidge()
cachedir = tempfile.mkdtemp()
mem = Memory(cachedir=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_
coef_ = clf.best_estimator_.steps[0][1].inverse_transform(coef_)
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 example: 1.72 seconds (0 minutes 1.72 seconds)

11.6.17 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"

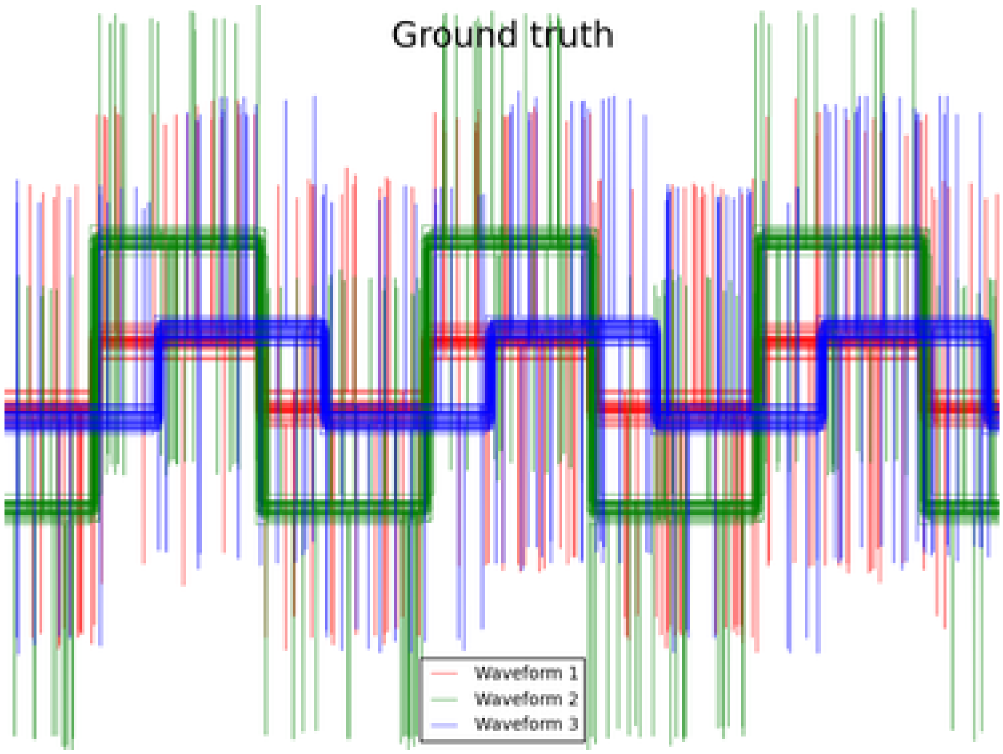
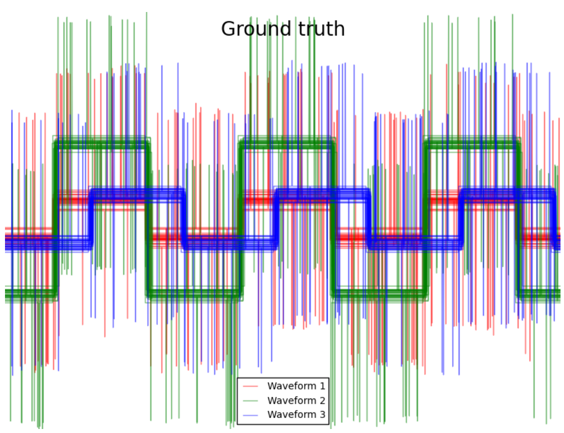
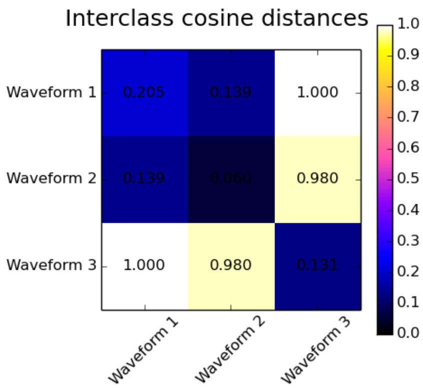
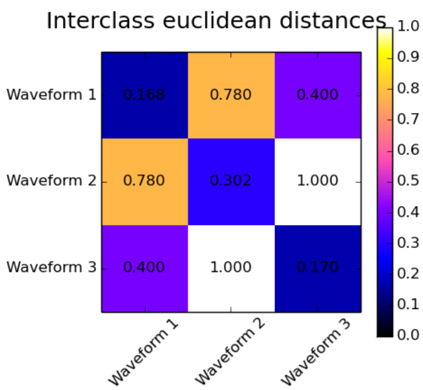
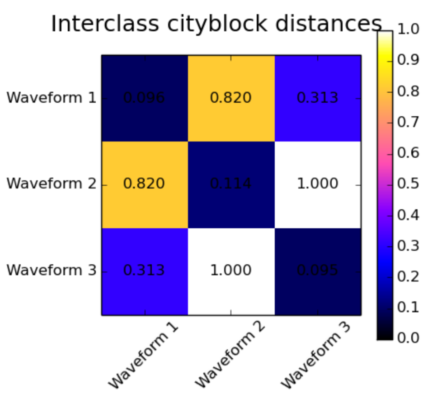


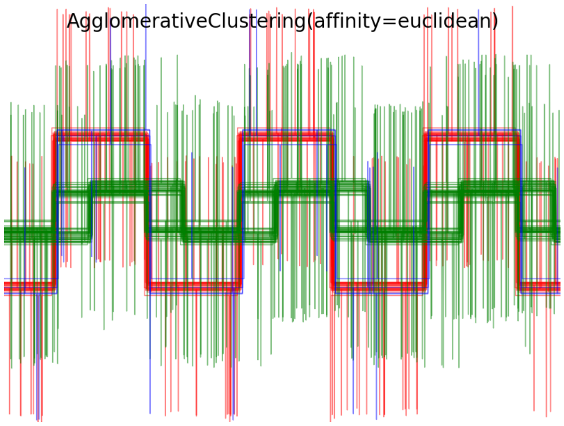
Figure 11.51: *Agglomerative clustering with different metrics*

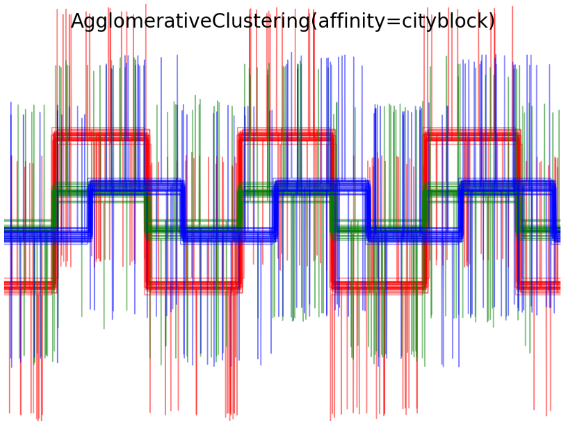
distance). This can be seen on the inter-class distance matrices: the values on the diagonal, that characterize the spread of the class, are much bigger for the Euclidean distance than for the cityblock distance.

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.

- 

- 
Interclass cosine distances
Waveform 1
Waveform 2
Waveform 3
0.205
0.139
1.000
0.139
0.000
0.980
1.000
0.980
0.131
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
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
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
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
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
- 
AgglomerativeClustering(affinity=cosine)

- 

- 

Python source code: `plot_agglomerative_clustering_metrics.py`

```
# 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
```

```
        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)
    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 example: 0.67 seconds (0 minutes 0.67 seconds)

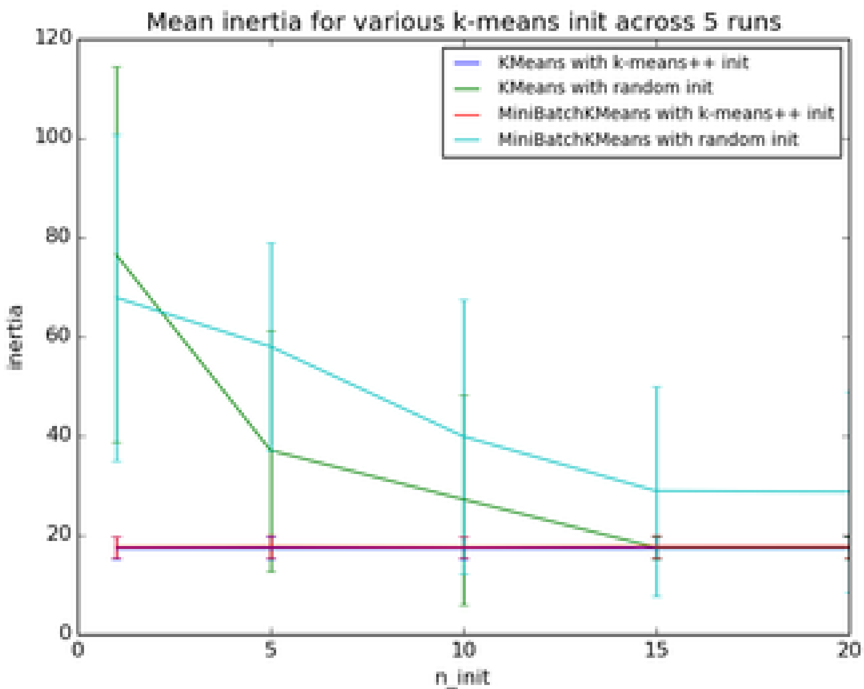


Figure 11.52: *Empirical evaluation of the impact of k-means initialization*

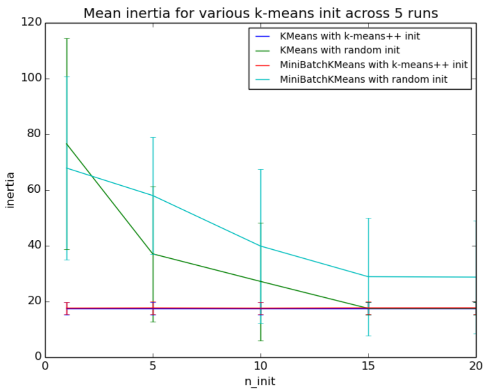
11.6.18 Empirical evaluation of the impact of k-means initialization

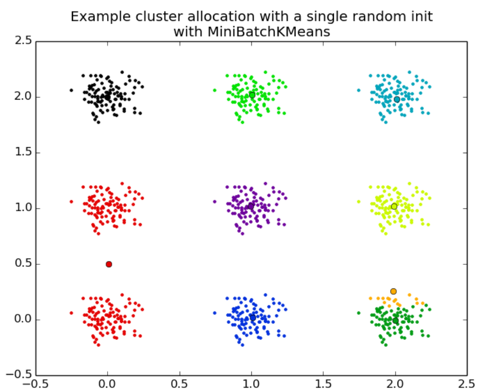
Evaluate the ability of k-means initializations strategies to make the algorithm convergence robust as measured by the relative standard deviation of the inertia of the clustering (i.e. the sum of distances to the nearest cluster center).

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.

- 

- 

Script output:

```
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
```

Python source code: `plot_kmeans_stability_low_dim_dense.py`

```
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)

# 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

fig = 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))

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)

fig = plt.figure()
for k in range(n_clusters):
    my_members = km.labels_ == k
    color = cm.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 example: 2.16 seconds (0 minutes 2.16 seconds)

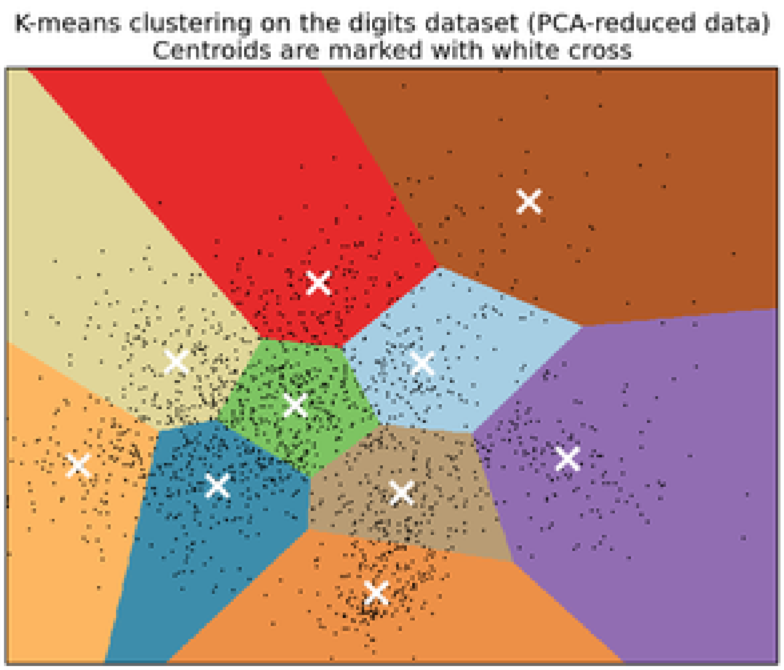


Figure 11.53: *A demo of K-Means clustering on the handwritten digits data*

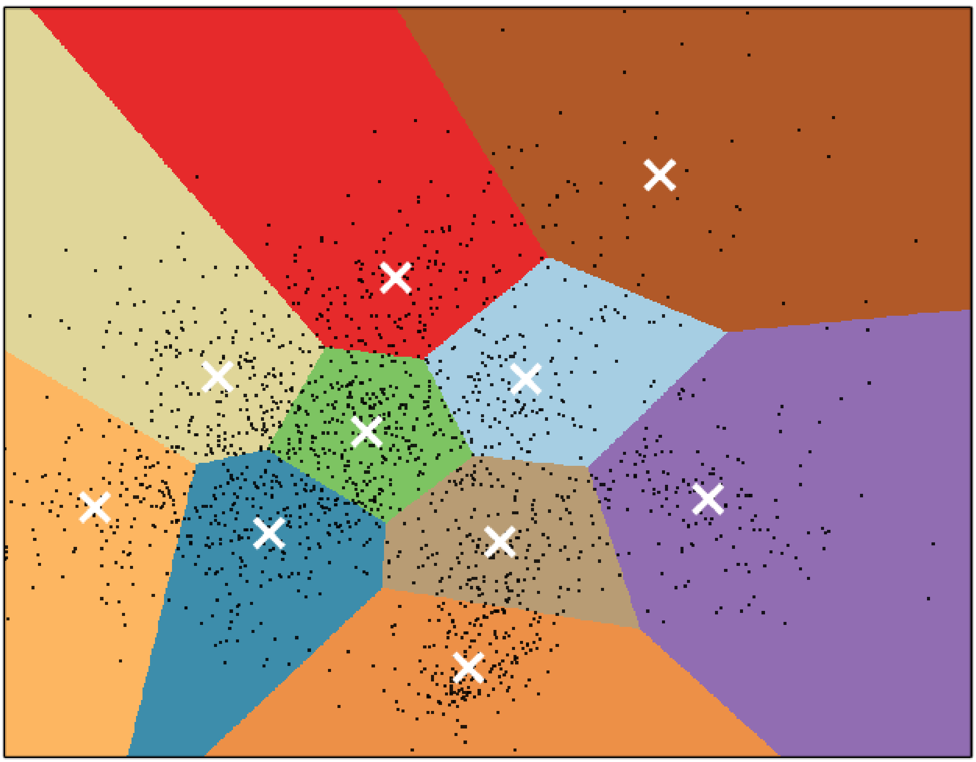
11.6.19 A demo of K-Means clustering on the handwritten digits data

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



Script output:

```
n_digits: 10,    n_samples 1797,         n_features 64
_______________________________________________________________________________
init     time  inertia    homo   compl  v-meas     ARI AMI  silhouette
k-means++   0.44s    69432   0.602   0.650   0.625   0.465   0.598    0.146
   random   0.41s    69694   0.669   0.710   0.689   0.553   0.666    0.147
PCA-based   0.03s    71820   0.673   0.715   0.693   0.567   0.670    0.150
_______________________________________________________________________________
```

Python source code: `plot_kmeans_digits.py`

```python
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
from sklearn.preprocessing import scale

np.random.seed(42)

digits = load_digits()
data = scale(digits.data)

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target

sample_size = 300

print("n_digits: %d, \t n_samples %d, \t n_features %d"
      % (n_digits, n_samples, n_features))


print(79 * '_')
print('% 9s' % 'init'
      '    time  inertia    homo   compl  v-meas     ARI AMI  silhouette')


def bench_k_means(estimator, name, data):
    t0 = time()
    estimator.fit(data)
    print('% 9s   %.2fs    %i   %.3f   %.3f   %.3f   %.3f   %.3f    %.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(79 * '_')

###############################################################################
```

```
# 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.
h = .02     # point in the mesh [x_min, m_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 example: 1.38 seconds (0 minutes 1.38 seconds)

11.6.20 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.

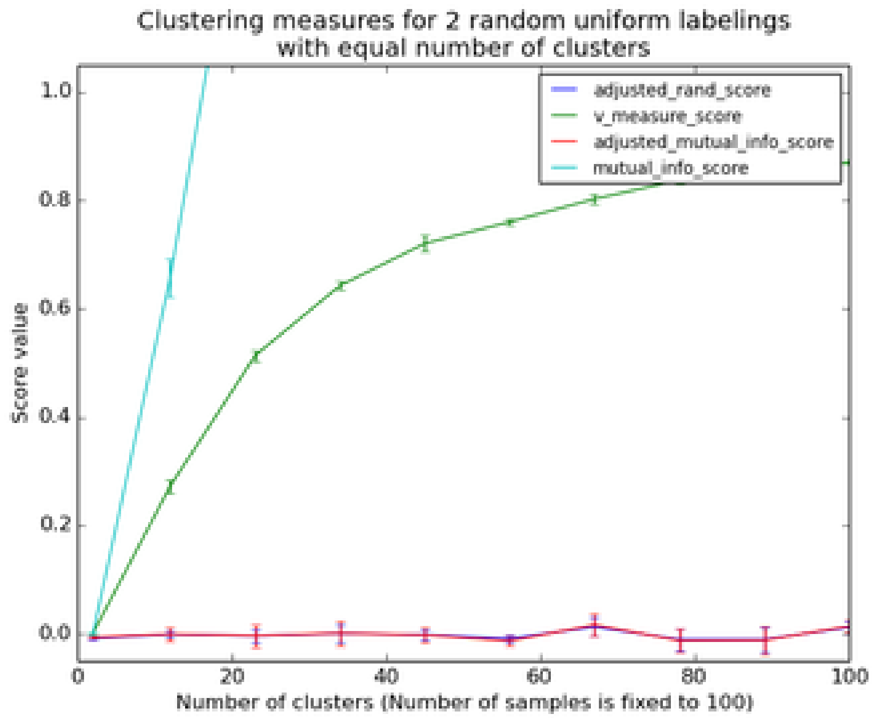
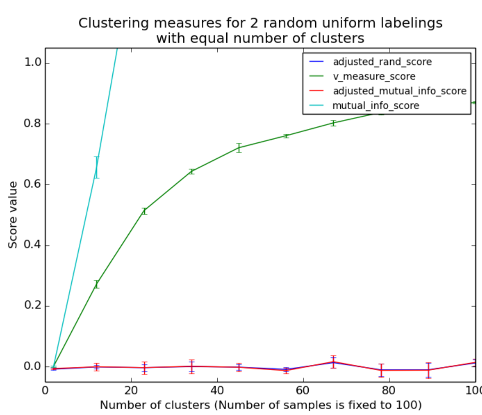
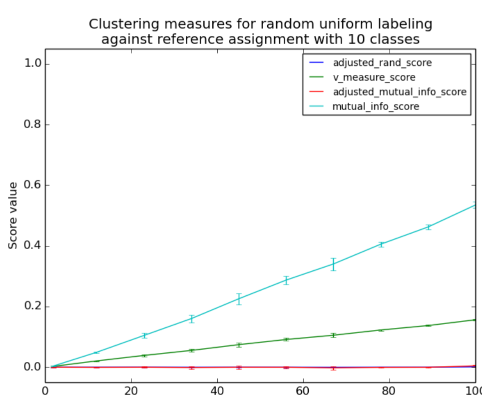


Figure 11.54: *Adjustment for chance in clustering performance evaluation*

- 

- 

Script output:

```
Computing adjusted_rand_score for 10 values of n_clusters and n_samples=100
done in 0.046s
Computing v_measure_score for 10 values of n_clusters and n_samples=100
done in 0.014s
Computing adjusted_mutual_info_score for 10 values of n_clusters and n_samples=100
done in 0.245s
Computing mutual_info_score for 10 values of n_clusters and n_samples=100
done in 0.010s
Computing adjusted_rand_score for 10 values of n_clusters and n_samples=1000
done in 0.048s
Computing v_measure_score for 10 values of n_clusters and n_samples=1000
done in 0.034s
Computing adjusted_mutual_info_score for 10 values of n_clusters and n_samples=1000
done in 0.200s
Computing mutual_info_score for 10 values of n_clusters and n_samples=1000
done in 0.021s
```

Python source code: `plot_adjusted_for_chance_measures.py`

```python
print(__doc__)

# Author: Olivier Grisel <olivier.grisel@ensta.org>
# 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).random_integers
    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 - 1,
                                 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 - 1, size=n_samples)
            labels_b = random_labels(low=0, high=k - 1, size=n_samples)
            scores[i, j] = score_func(labels_a, labels_b)
    return scores

score_funcs = [
    metrics.adjusted_rand_score,
```

```
    metrics.v_measure_score,
    metrics.adjusted_mutual_info_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)
    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(ymin=-0.05, ymax=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(ymin=-0.05, ymax=1.05)
plt.legend(plots, names)
plt.show()
```

Total running time of the example: 0.77 seconds (0 minutes 0.77 seconds)

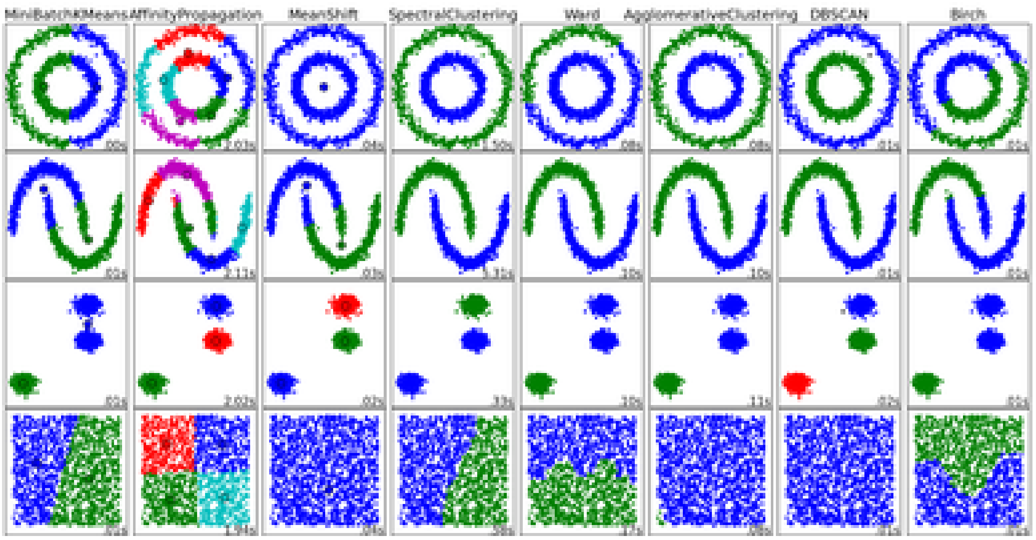


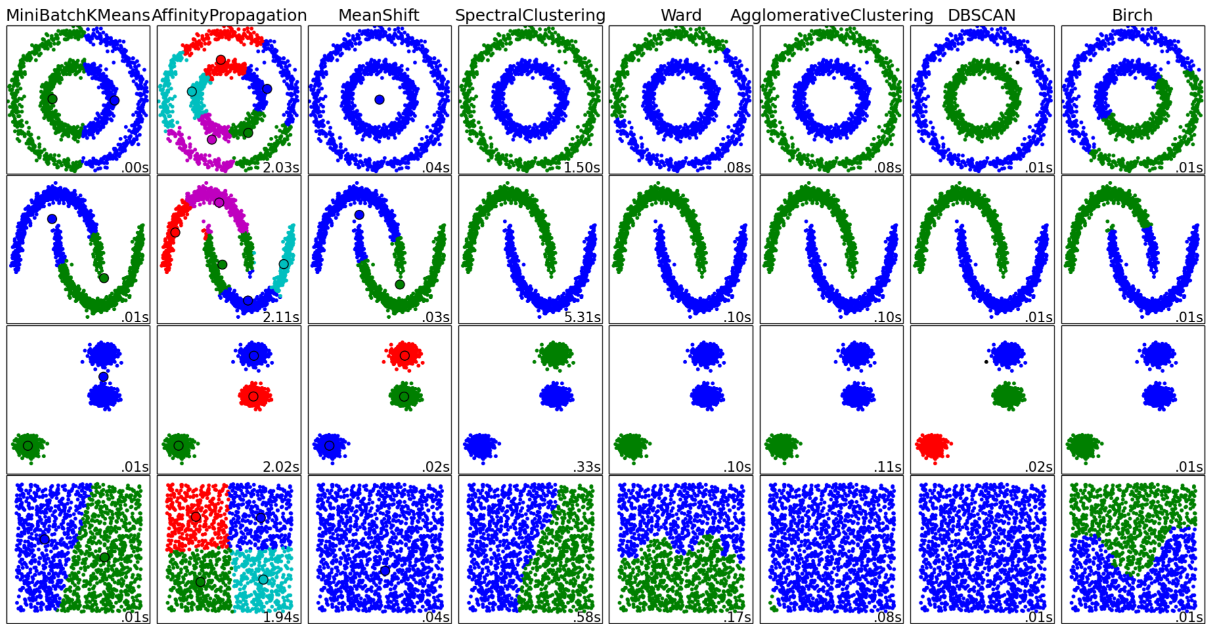
Figure 11.55: *Comparing different clustering algorithms on toy datasets*

11.6.21 Comparing different clustering algorithms on toy datasets

This example aims at showing characteristics of different clustering algorithms on datasets that are "interesting" but still in 2D. The last dataset is an example of a 'null' situation for clustering: the data is homogeneous, and there is no good clustering.

While these examples give some intuition about the algorithms, this intuition might not apply to very high dimensional data.

The results could be improved by tweaking the parameters for each clustering strategy, for instance setting the number of clusters for the methods that needs this parameter specified. Note that affinity propagation has a tendency to create many clusters. Thus in this example its two parameters (damping and per-point preference) were set to to mitigate this behavior.



Python source code: plot_cluster_comparison.py

```
print(__doc__)

import time

import numpy as np
import matplotlib.pyplot as plt

from sklearn import cluster, datasets
from sklearn.neighbors import kneighbors_graph
from sklearn.preprocessing import StandardScaler

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

colors = np.array([x for x in 'bgrcmykbgrcmykbgrcmykbgrcmyk'])
colors = np.hstack([colors] * 20)

clustering_names = [
    'MiniBatchKMeans', 'AffinityPropagation', 'MeanShift',
    'SpectralClustering', 'Ward', 'AgglomerativeClustering',
    'DBSCAN', 'Birch']

plt.figure(figsize=(len(clustering_names) * 2 + 3, 9.5))
plt.subplots_adjust(left=.02, right=.98, bottom=.001, top=.96, wspace=.05,
                    hspace=.01)

plot_num = 1
```

```
datasets = [noisy_circles, noisy_moons, blobs, no_structure]
for i_dataset, dataset in enumerate(datasets):
    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=0.3)

    # connectivity matrix for structured Ward
    connectivity = kneighbors_graph(X, n_neighbors=10, include_self=False)
    # make connectivity symmetric
    connectivity = 0.5 * (connectivity + connectivity.T)

    # create clustering estimators
    ms = cluster.MeanShift(bandwidth=bandwidth, bin_seeding=True)
    two_means = cluster.MiniBatchKMeans(n_clusters=2)
    ward = cluster.AgglomerativeClustering(n_clusters=2, linkage='ward',
                                           connectivity=connectivity)
    spectral = cluster.SpectralClustering(n_clusters=2,
                                          eigen_solver='arpack',
                                          affinity="nearest_neighbors")
    dbscan = cluster.DBSCAN(eps=.2)
    affinity_propagation = cluster.AffinityPropagation(damping=.9,
                                                       preference=-200)

    average_linkage = cluster.AgglomerativeClustering(
        linkage="average", affinity="cityblock", n_clusters=2,
        connectivity=connectivity)

    birch = cluster.Birch(n_clusters=2)
    clustering_algorithms = [
        two_means, affinity_propagation, ms, spectral, ward, average_linkage,
        dbscan, birch]

    for name, algorithm in zip(clustering_names, clustering_algorithms):
        # predict cluster memberships
        t0 = time.time()
        algorithm.fit(X)
        t1 = time.time()
        if hasattr(algorithm, 'labels_'):
            y_pred = algorithm.labels_.astype(np.int)
        else:
            y_pred = algorithm.predict(X)

        # plot
        plt.subplot(4, len(clustering_algorithms), plot_num)
        if i_dataset == 0:
            plt.title(name, size=18)
        plt.scatter(X[:, 0], X[:, 1], color=colors[y_pred].tolist(), s=10)

        if hasattr(algorithm, 'cluster_centers_'):
            centers = algorithm.cluster_centers_
            center_colors = colors[:len(centers)]
            plt.scatter(centers[:, 0], centers[:, 1], s=100, c=center_colors)
        plt.xlim(-2, 2)
        plt.ylim(-2, 2)
        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 example: 18.49 seconds (0 minutes 18.49 seconds)

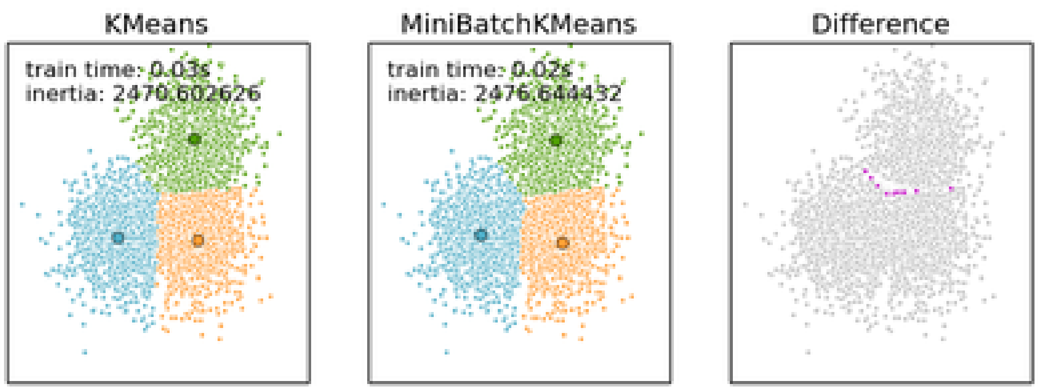
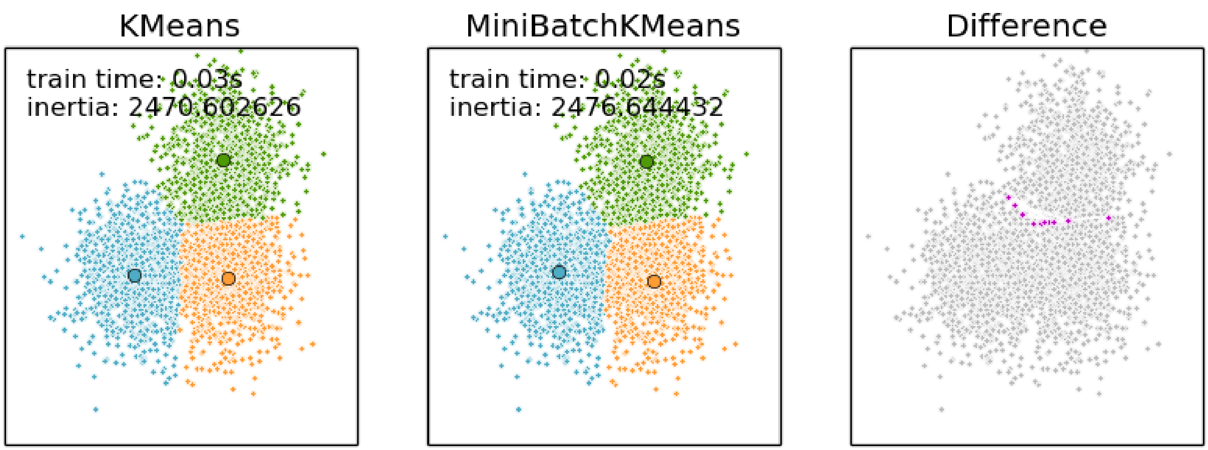


Figure 11.56: *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*

11.6.22 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.



Python source code: `plot_mini_batch_kmeans.py`

```
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.samples_generator 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
k_means_labels = k_means.labels_
k_means_cluster_centers = k_means.cluster_centers_
k_means_labels_unique = np.unique(k_means_labels)

##############################################################################
# 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
mbk_means_labels = mbk.labels_
```

```
mbk_means_cluster_centers = mbk.cluster_centers_
mbk_means_labels_unique = np.unique(mbk_means_labels)

##############################################################################
# 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.

order = pairwise_distances_argmin(k_means_cluster_centers,
                                  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 == order[k]
    cluster_center = mbk_means_cluster_centers[order[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 l in range(n_clusters):
    different += ((k_means_labels == k) != (mbk_means_labels == order[k]))

identic = np.logical_not(different)
ax.plot(X[identic, 0], X[identic, 1], 'w',
        markerfacecolor='#bbbbbb', marker='.')
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 example: 0.23 seconds (0 minutes 0.23 seconds)

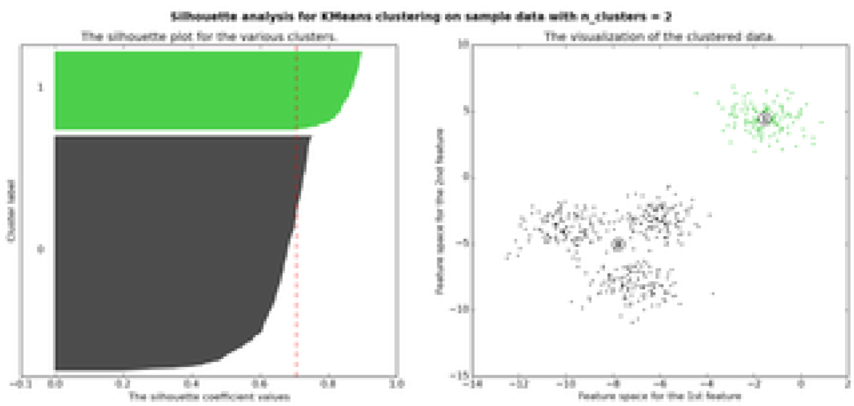


Figure 11.57: *Selecting the number of clusters with silhouette analysis on KMeans clustering*

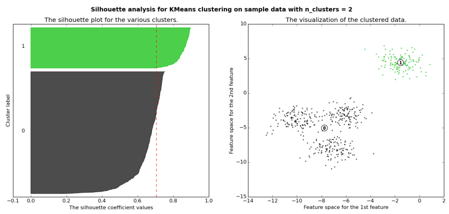
11.6.23 Selecting the number of clusters with silhouette analysis on KMeans clustering

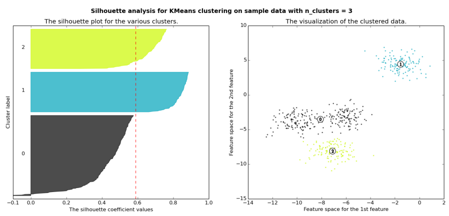
Silhouette analysis can be used to study the separation distance between the resulting clusters. The silhouette plot displays a measure of how close each point in one cluster is to points in the neighboring clusters and thus provides a way to assess parameters like number of clusters visually. This measure has a range of [-1, 1].

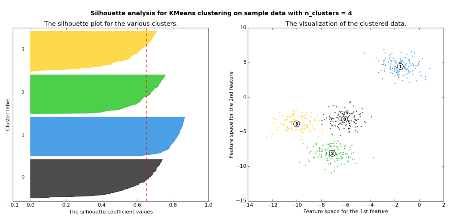
Silhoette coefficients (as these values are referred to as) near +1 indicate that the sample is far away from the neighboring clusters. A value of 0 indicates that the sample is on or very close to the decision boundary between two neighboring clusters and negative values indicate that those samples might have been assigned to the wrong cluster.

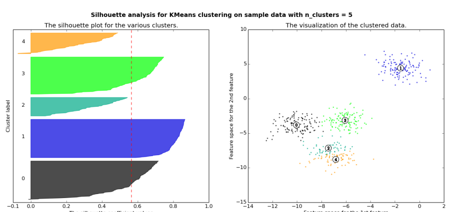
In this example the silhouette analysis is used to choose an optimal value for `n_clusters`. The silhouette plot shows that the `n_clusters` value of 3, 5 and 6 are a bad pick for the given data due to the presence of clusters with below average silhouette scores and also due to wide fluctuations in the size of the silhouette plots. Silhouette analysis is more ambivalent in deciding between 2 and 4.

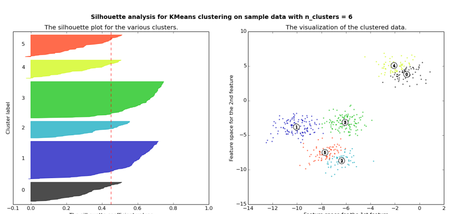
Also from the thickness of the silhouette plot the cluster size can be visualized. The silhouette plot for cluster 0 when `n_clusters` is equal to 2, is bigger in size owing to the grouping of the 3 sub clusters into one big cluster. However when the n_clusters is equal to 4, all the plots are more or less of similar thickness and hence are of similar sizes as can be also verified from the labelled scatter plot on the right.

- 

- 

- 

- 

- 

Script output:

```
For n_clusters = 2 The average silhouette_score is : 0.704978749608
For n_clusters = 3 The average silhouette_score is : 0.588200401213
For n_clusters = 4 The average silhouette_score is : 0.650518663273
For n_clusters = 5 The average silhouette_score is : 0.563764690262
For n_clusters = 6 The average silhouette_score is : 0.450301208266
```

Python source code: plot_kmeans_silhouette_analysis.py

```
from __future__ import print_function

from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score
```

```python
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 distict 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)

    # 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.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 silhoutte 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
    colors = cm.spectral(cluster_labels.astype(float) / n_clusters)
    ax2.scatter(X[:, 0], X[:, 1], marker='.', s=30, lw=0, alpha=0.7,
                c=colors)

    # 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)

    for i, c in enumerate(centers):
        ax2.scatter(c[0], c[1], marker='$%d$' % i, alpha=1, s=50)

    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 example: 1.09 seconds (0 minutes 1.09 seconds)

11.7 Covariance estimation

Examples concerning the `sklearn.covariance` module.

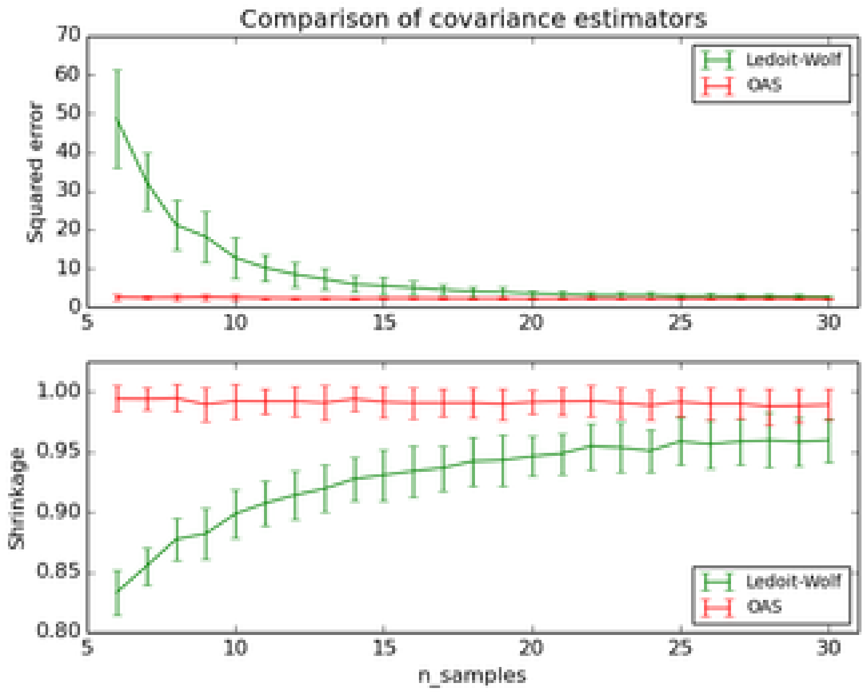


Figure 11.58: *Ledoit-Wolf vs OAS estimation*

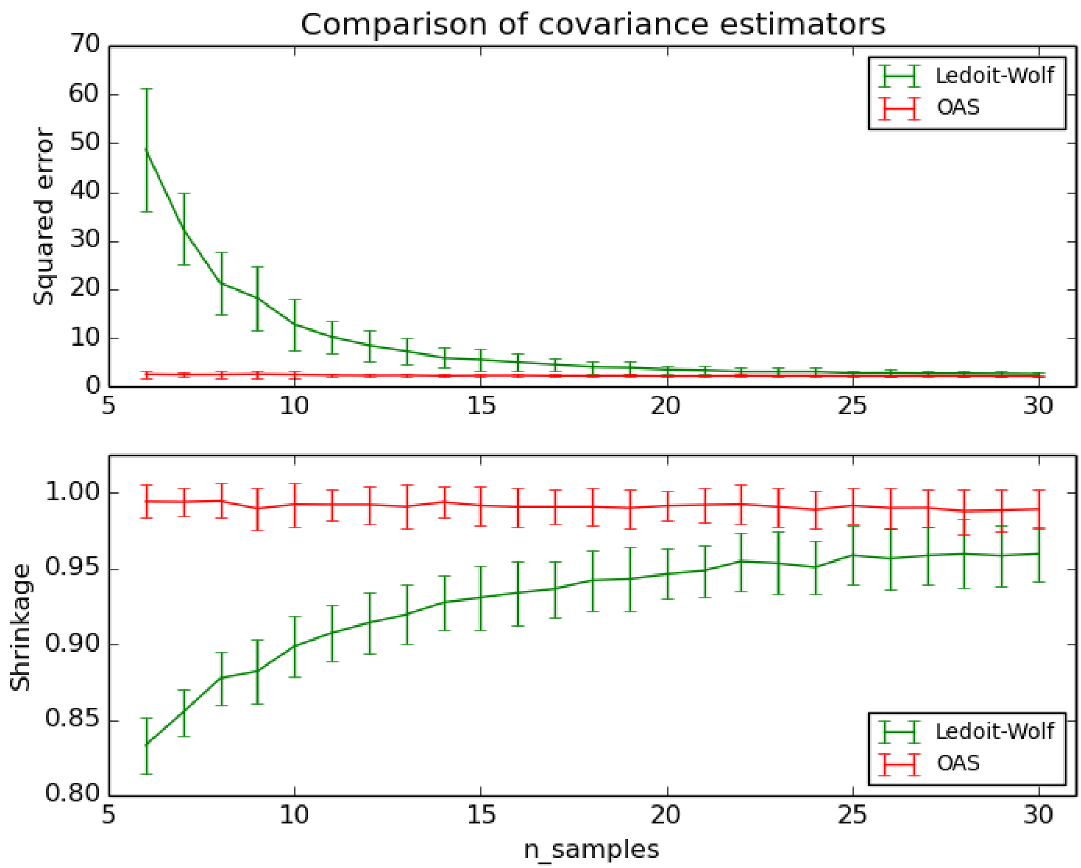
11.7.1 Ledoit-Wolf vs OAS estimation

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.



Python source code: `plot_lw_vs_oas.py`

```
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='g')
plt.errorbar(n_samples_range, oa_mse.mean(1), yerr=oa_mse.std(1),
             label='OAS', color='r')
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='g')
plt.errorbar(n_samples_range, oa_shrinkage.mean(1), yerr=oa_shrinkage.std(1),
             label='OAS', color='r')
plt.xlabel("n_samples")
plt.ylabel("Shrinkage")
plt.legend(loc="lower right")
plt.ylim(plt.ylim()[0], 1. + (plt.ylim()[1] - plt.ylim()[0]) / 10.)
plt.xlim(5, 31)

plt.show()
```

Total running time of the example: 3.48 seconds (0 minutes 3.48 seconds)

11.7.2 Outlier detection with several methods.

When the amount of contamination is known, this example illustrates two different ways of performing *Novelty and Outlier Detection*:

- based on a robust estimator of covariance, which is assuming that the data are Gaussian distributed and performs better than the One-Class SVM in that case.
- using the One-Class SVM and its ability to capture the shape of the data set, hence performing better when the data is strongly non-Gaussian, i.e. with two well-separated clusters;

The ground truth about inliers and outliers is given by the points colors while the orange-filled area indicates which points are reported as inliers by each method.

Here, we assume that we know the fraction of outliers in the datasets. Thus rather than using the 'predict' method of the objects, we set the threshold on the decision_function to separate out the corresponding fraction.

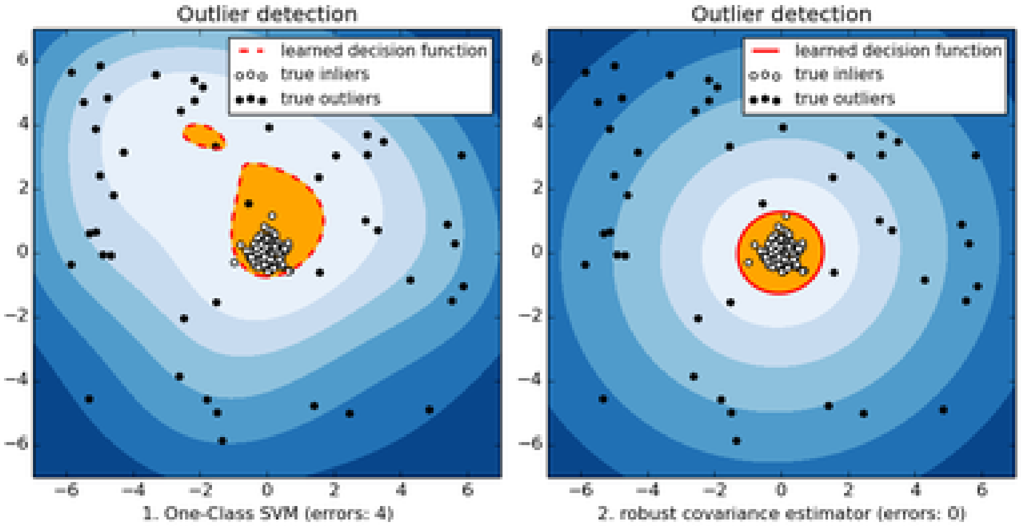
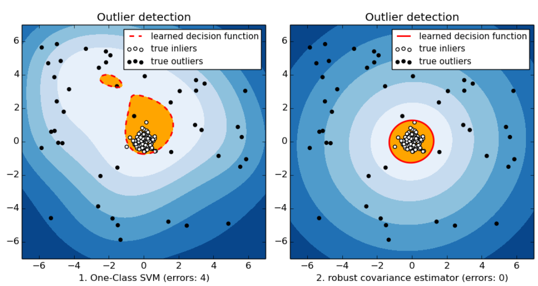
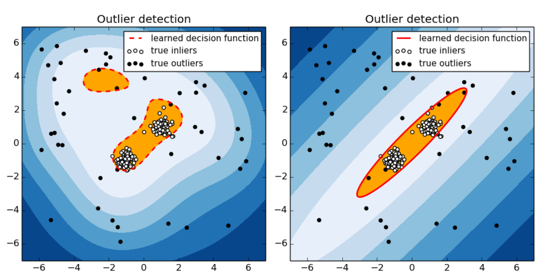
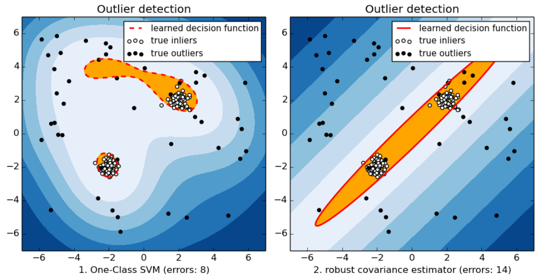


Figure 11.59: *Outlier detection with several methods.*

- 

- 

- 

Python source code: `plot_outlier_detection.py`

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.font_manager
from scipy import stats

from sklearn import svm
from sklearn.covariance import EllipticEnvelope

# Example settings
n_samples = 200
outliers_fraction = 0.25
clusters_separation = [0, 1, 2]

# define two outlier detection tools to be compared
classifiers = {
    "One-Class SVM": svm.OneClassSVM(nu=0.95 * outliers_fraction + 0.05,
                                     kernel="rbf", gamma=0.1),
    "robust covariance estimator": EllipticEnvelope(contamination=.1)}

# Compare given classifiers under given settings
xx, yy = np.meshgrid(np.linspace(-7, 7, 500), np.linspace(-7, 7, 500))
n_inliers = int((1. - outliers_fraction) * n_samples)
n_outliers = int(outliers_fraction * n_samples)
ground_truth = np.ones(n_samples, dtype=int)
ground_truth[-n_outliers:] = 0

# Fit the problem with varying cluster separation
for i, offset in enumerate(clusters_separation):
    np.random.seed(42)
    # Data generation
    X1 = 0.3 * np.random.randn(0.5 * n_inliers, 2) - offset
    X2 = 0.3 * np.random.randn(0.5 * n_inliers, 2) + offset
    X = np.r_[X1, X2]
    # Add outliers
    X = np.r_[X, np.random.uniform(low=-6, high=6, size=(n_outliers, 2))]

    # Fit the model with the One-Class SVM
    plt.figure(figsize=(10, 5))
    for i, (clf_name, clf) in enumerate(classifiers.items()):
        # fit the data and tag outliers
        clf.fit(X)
        y_pred = clf.decision_function(X).ravel()
        threshold = stats.scoreatpercentile(y_pred,
                                            100 * outliers_fraction)
        y_pred = y_pred > threshold
```

```
        n_errors = (y_pred != ground_truth).sum()
        # plot the levels lines and the points
        Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)
        subplot = plt.subplot(1, 2, i + 1)
        subplot.set_title("Outlier detection")
        subplot.contourf(xx, yy, Z, levels=np.linspace(Z.min(), threshold, 7),
                         cmap=plt.cm.Blues_r)
        a = subplot.contour(xx, yy, Z, levels=[threshold],
                            linewidths=2, colors='red')
        subplot.contourf(xx, yy, Z, levels=[threshold, Z.max()],
                         colors='orange')
        b = subplot.scatter(X[:-n_outliers, 0], X[:-n_outliers, 1], c='white')
        c = subplot.scatter(X[-n_outliers:, 0], X[-n_outliers:, 1], c='black')
        subplot.axis('tight')
        subplot.legend(
            [a.collections[0], b, c],
            ['learned decision function', 'true inliers', 'true outliers'],
            prop=matplotlib.font_manager.FontProperties(size=11))
        subplot.set_xlabel("%d. %s (errors: %d)" % (i + 1, clf_name, n_errors))
        subplot.set_xlim((-7, 7))
        subplot.set_ylim((-7, 7))
    plt.subplots_adjust(0.04, 0.1, 0.96, 0.94, 0.1, 0.26)

plt.show()
```

Total running time of the example: 1.65 seconds (0 minutes 1.65 seconds)

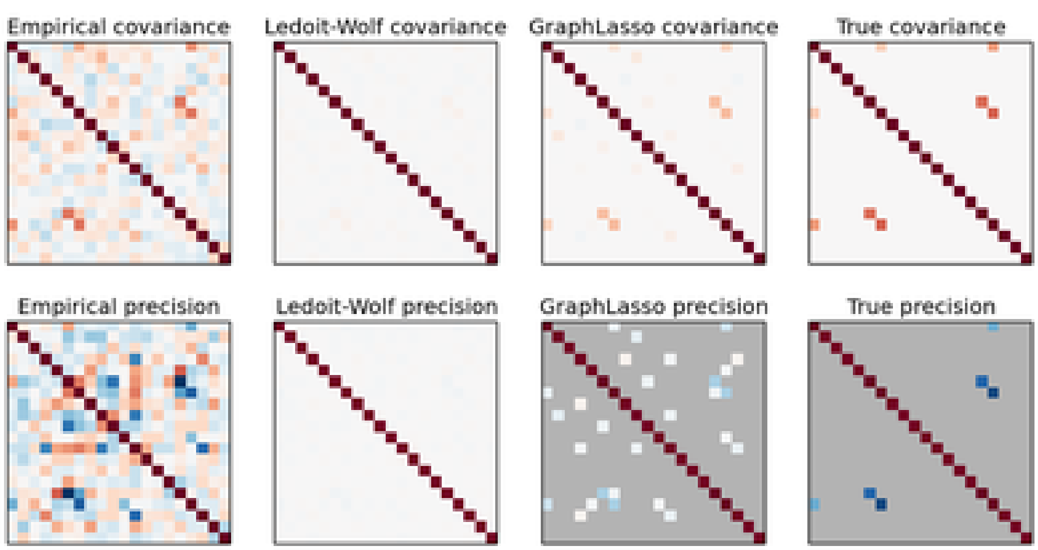


Figure 11.60: *Sparse inverse covariance estimation*

11.7.3 Sparse inverse covariance estimation

Using the GraphLasso 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.

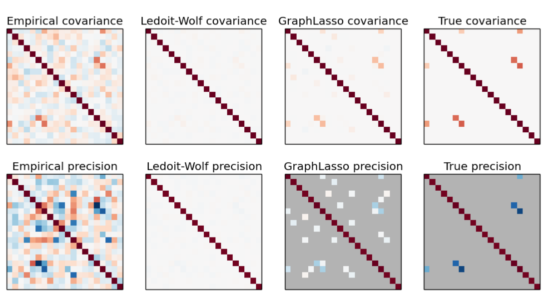
Here, the number of samples is slightly larger than the number of dimensions, thus the empirical covariance is still invertible. However, as the observations are strongly correlated, the empirical covariance matrix is ill-conditioned and as a result its inverse –the empirical precision matrix– is very far from the ground truth.

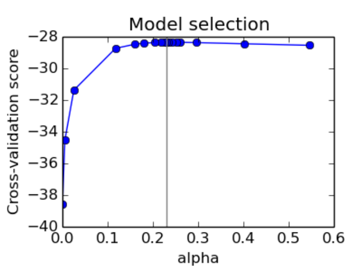
If we use l2 shrinkage, as with the Ledoit-Wolf estimator, as the number of samples is small, we need to shrink a lot. As a result, the Ledoit-Wolf precision is fairly close to the ground truth precision, that is not far from being diagonal, but the off-diagonal structure is lost.

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 GraphLasso setting the sparsity of the model is set by internal cross-validation in the GraphLassoCV. As can be seen on figure 2, the grid to compute the cross-validation score is iteratively refined in the neighborhood of the maximum.

- 

- 

Python source code: `plot_sparse_cov.py`

```python
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 GraphLassoCV, 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 = GraphLassoCV()
model.fit(X)
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_),
        ('GraphLasso', 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_),
         ('GraphLasso', 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)
    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 example: 0.45 seconds (0 minutes 0.45 seconds)

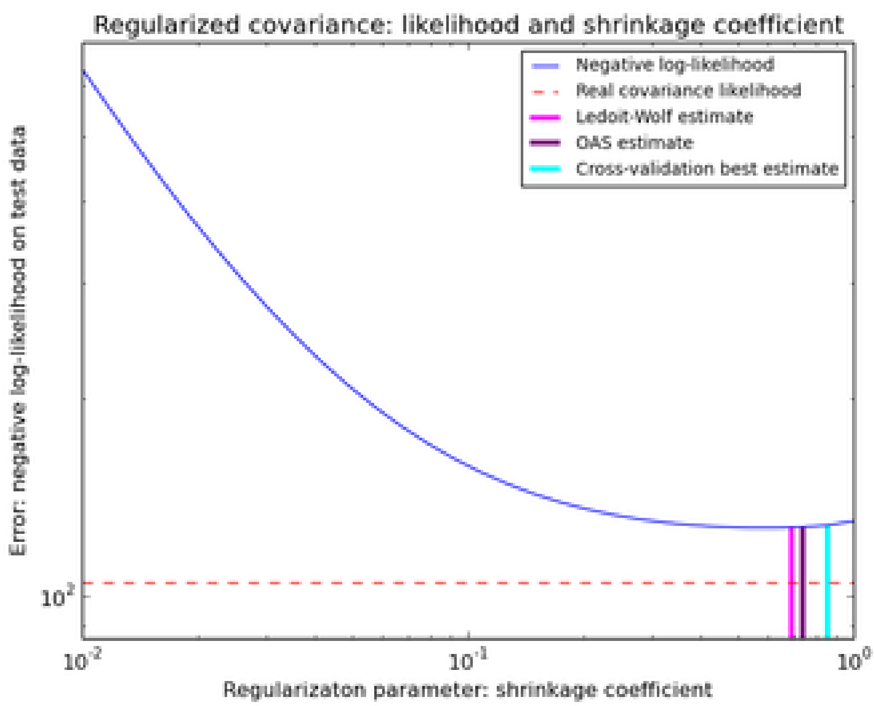


Figure 11.61: *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*

11.7.4 Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood

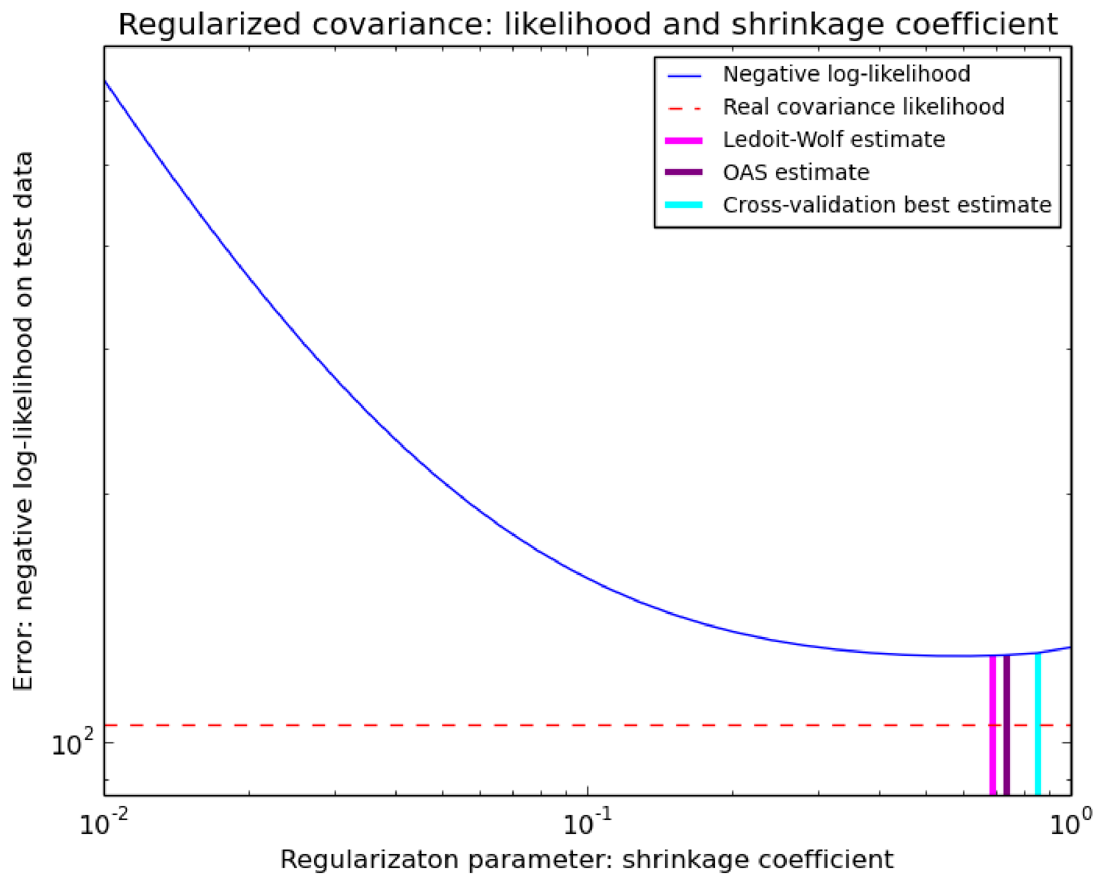
When working with covariance estimation, the usual approach is to use a maximum likelihood estimator, such as the `sklearn.covariance.EmpiricalCovariance`. It is unbiased, i.e. it converges to the true (population) covariance when given many observations. However, it can also be beneficial to regularize 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.



Python source code: `plot_covariance_estimation.py`

```
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.grid_search 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

# 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('Regularizaton 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')
```

```
plt.ylim(ymin, ymax)
plt.xlim(xmin, xmax)
plt.legend()

plt.show()
```

Total running time of the example: 0.20 seconds (0 minutes 0.20 seconds)

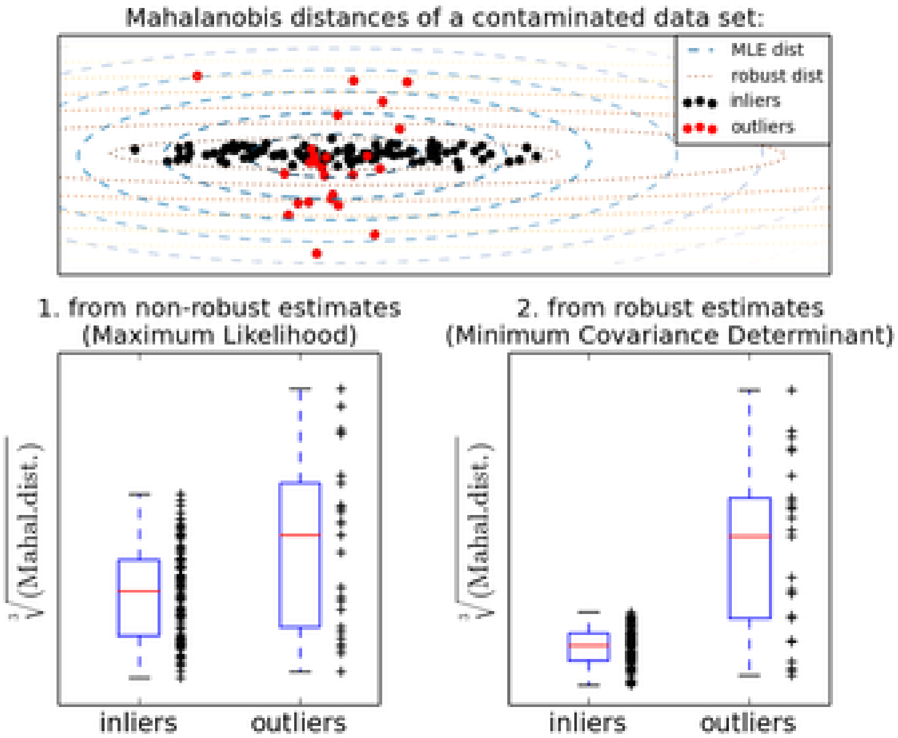


Figure 11.62: *Robust covariance estimation and Mahalanobis distances relevance*

11.7.5 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.

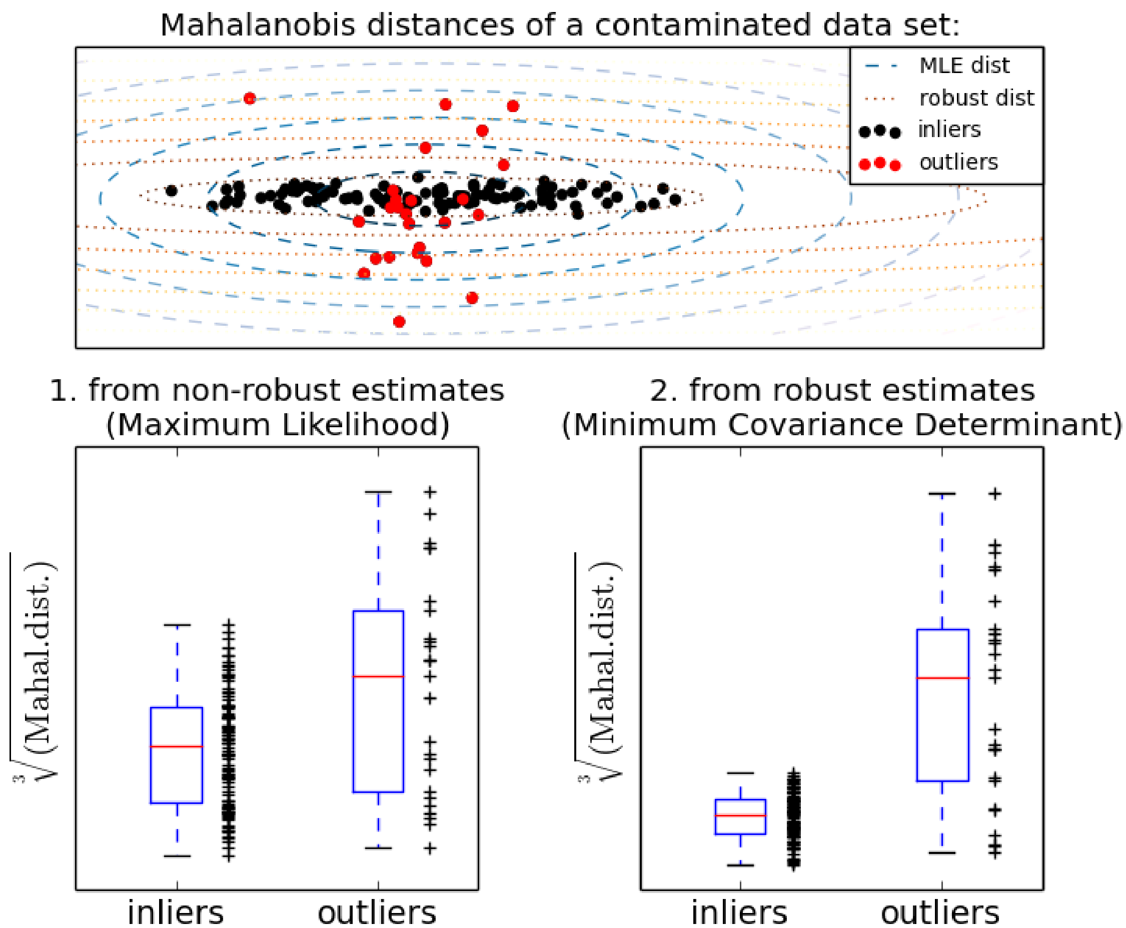
The Minimum Covariance Determinant estimator is a robust, high-breakdown point (i.e. it can be used to estimate the covariance matrix of highly contaminated datasets, up to $\frac{n_{\text{samples}} - n_{\text{features}} - 1}{2}$ outliers) estimator of covariance. The idea is to find $\frac{n_{\text{samples}} + n_{\text{features}} + 1}{2}$ 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.



Python source code: `plot_mahalanobis_distances.py`

```
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)

###############################################################################
# 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(1.26 * np.ones(n_samples - n_outliers),
             emp_mahal[:-n_outliers], '+k', markeredgewidth=1)
subfig2.plot(2.26 * np.ones(n_outliers),
```

```
             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(1.26 * np.ones(n_samples - n_outliers),
             robust_mahal[:-n_outliers], '+k', markeredgewidth=1)
subfig3.plot(2.26 * np.ones(n_outliers),
             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 example: 0.19 seconds (0 minutes 0.19 seconds)

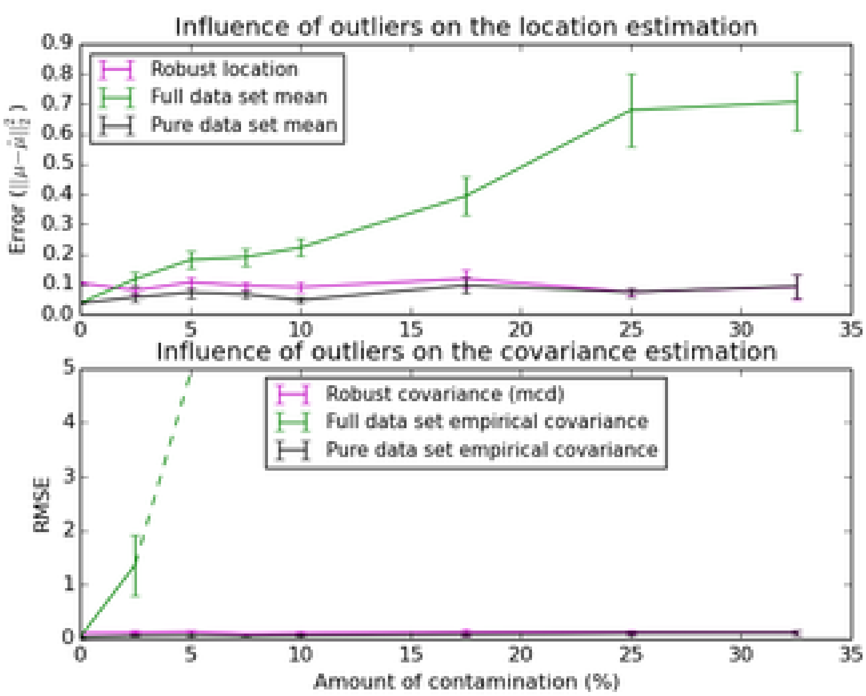


Figure 11.63: *Robust vs Empirical covariance estimate*

11.7.6 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.

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 [1].

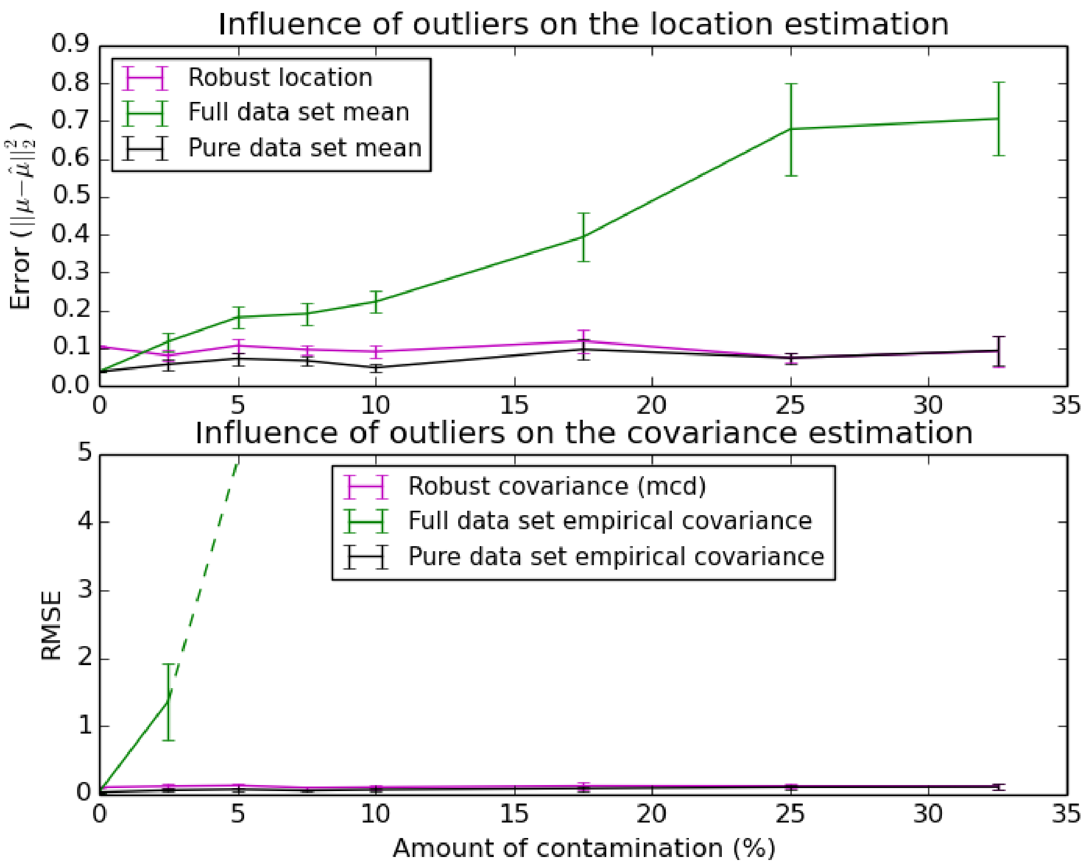
Evaluation

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] P. J. Rousseeuw. Least median of squares regression. J. Am Stat Ass, 79:871, 1984.

References



Python source code: `plot_robust_vs_empirical_covariance.py`

```
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]))

# 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))
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)
plt.errorbar(range_n_outliers, err_loc_mcd.mean(1),
             yerr=err_loc_mcd.std(1) / np.sqrt(repeat),
             label="Robust location", 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", 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", 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
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 example: 1.70 seconds (0 minutes 1.70 seconds)

11.8 Cross decomposition

Examples concerning the `sklearn.cross_decomposition` module.

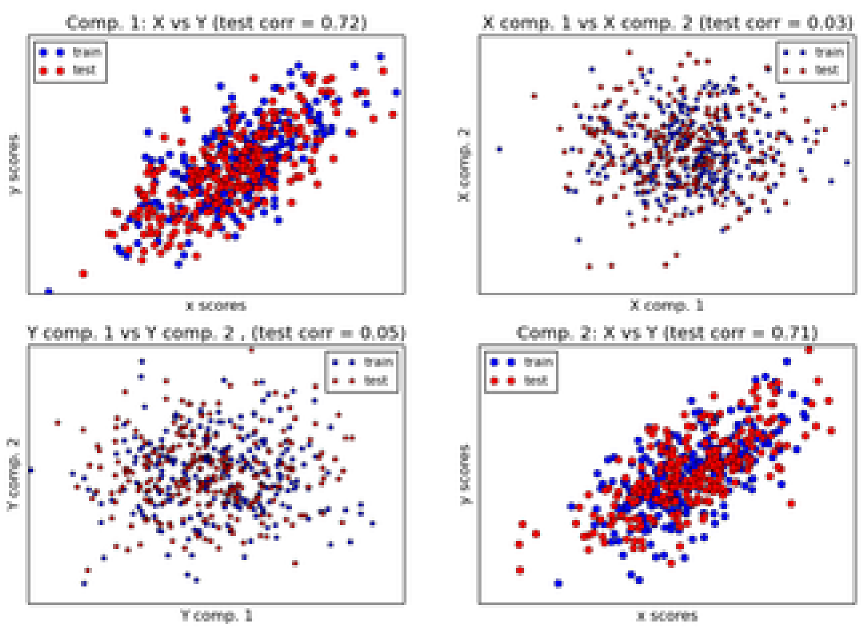
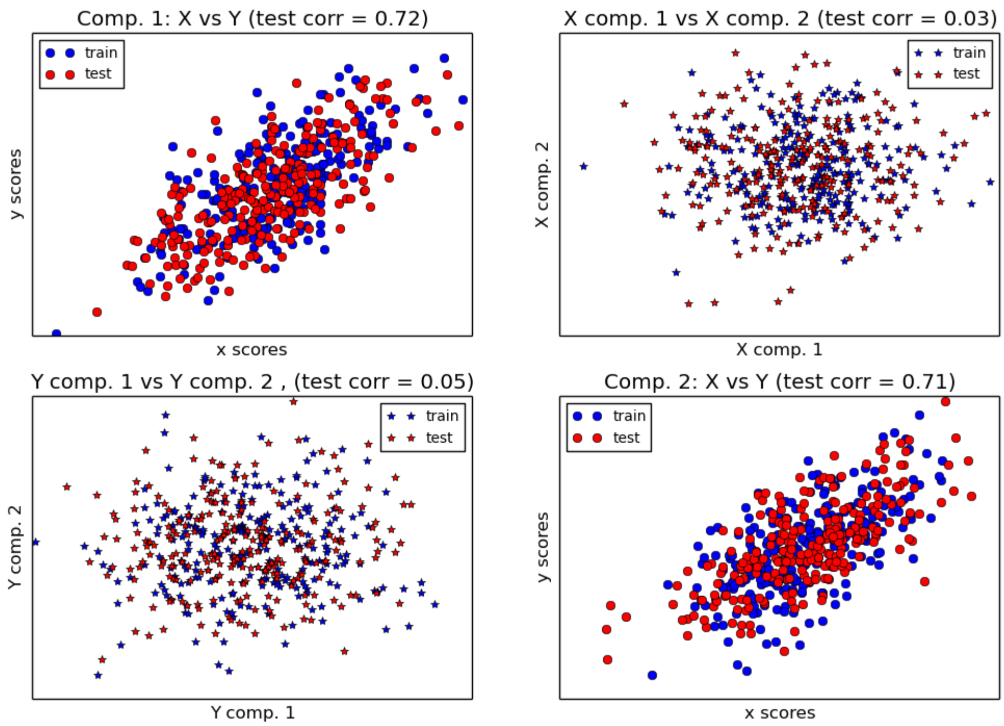


Figure 11.64: *Compare cross decomposition methods*

11.8.1 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.



Script output:

```
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. ]
 [ 1.9   2.    2. ]
 [-0.   -0.    0. ]
 [ 0.    0.    0. ]
 [ 0.    0.    0. ]
 [ 0.    0.   -0. ]
 [-0.   -0.   -0.1]
 [-0.   -0.    0. ]
 [ 0.    0.    0.1]
 [ 0.    0.   -0. ]]
Estimated betas
[[ 1.]
 [ 2.]
 [ 0.]
 [ 0.]
 [ 0.]
 [-0.]
 [-0.]
 [ 0.]
 [-0.]
 [-0.]]
```

Python source code: `plot_compare_cross_decomposition.py`

```python
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)
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.plot(X_train_r[:, 0], Y_train_r[:, 0], "ob", label="train")
plt.plot(X_test_r[:, 0], Y_test_r[:, 0], "or", label="test")
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.plot(X_train_r[:, 1], Y_train_r[:, 1], "ob", label="train")
plt.plot(X_test_r[:, 1], Y_test_r[:, 1], "or", label="test")
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.plot(X_train_r[:, 0], X_train_r[:, 1], "*b", label="train")
plt.plot(X_test_r[:, 0], X_test_r[:, 1], "*r", label="test")
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.plot(Y_train_r[:, 0], Y_train_r[:, 1], "*b", label="train")
plt.plot(Y_test_r[:, 0], Y_test_r[:, 1], "*r", label="test")
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])
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.coefs with B
print("Estimated B")
print(np.round(pls2.coefs, 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 compements exceeds 1 (the dimension of y)
print("Estimated betas")
print(np.round(pls1.coefs, 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 = plsca.transform(X_train, Y_train)
X_test_r, Y_test_r = plsca.transform(X_test, Y_test)
```

Total running time of the example: 0.19 seconds (0 minutes 0.19 seconds)

11.9 Dataset examples

Examples concerning the `sklearn.datasets` module.

11.9.1 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.

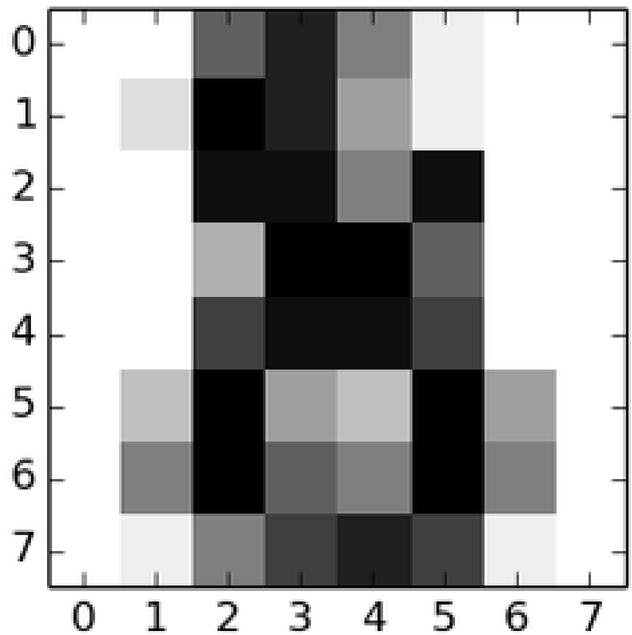
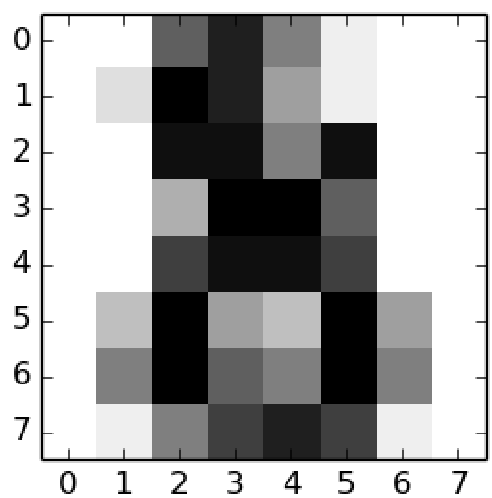


Figure 11.65: *The Digit Dataset*



Python source code: `plot_digits_last_image.py`

```
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 example: 0.12 seconds (0 minutes 0.12 seconds)

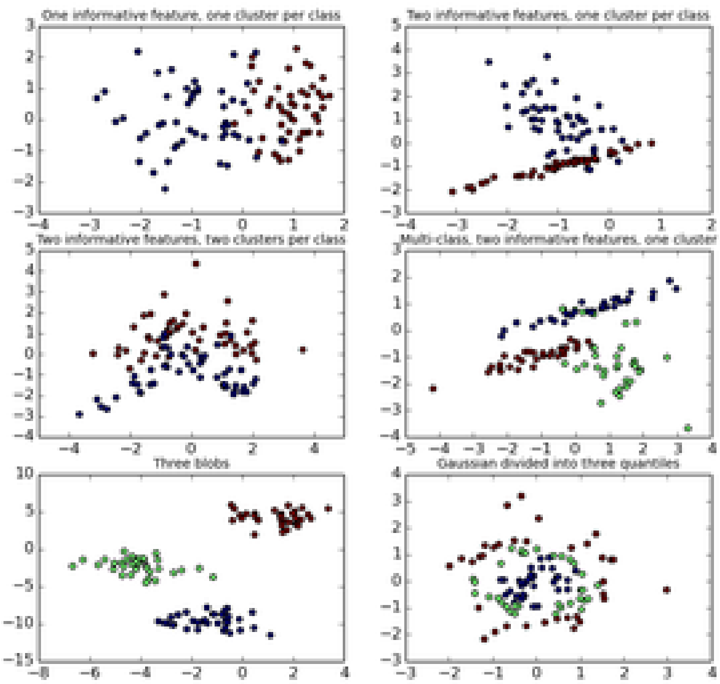
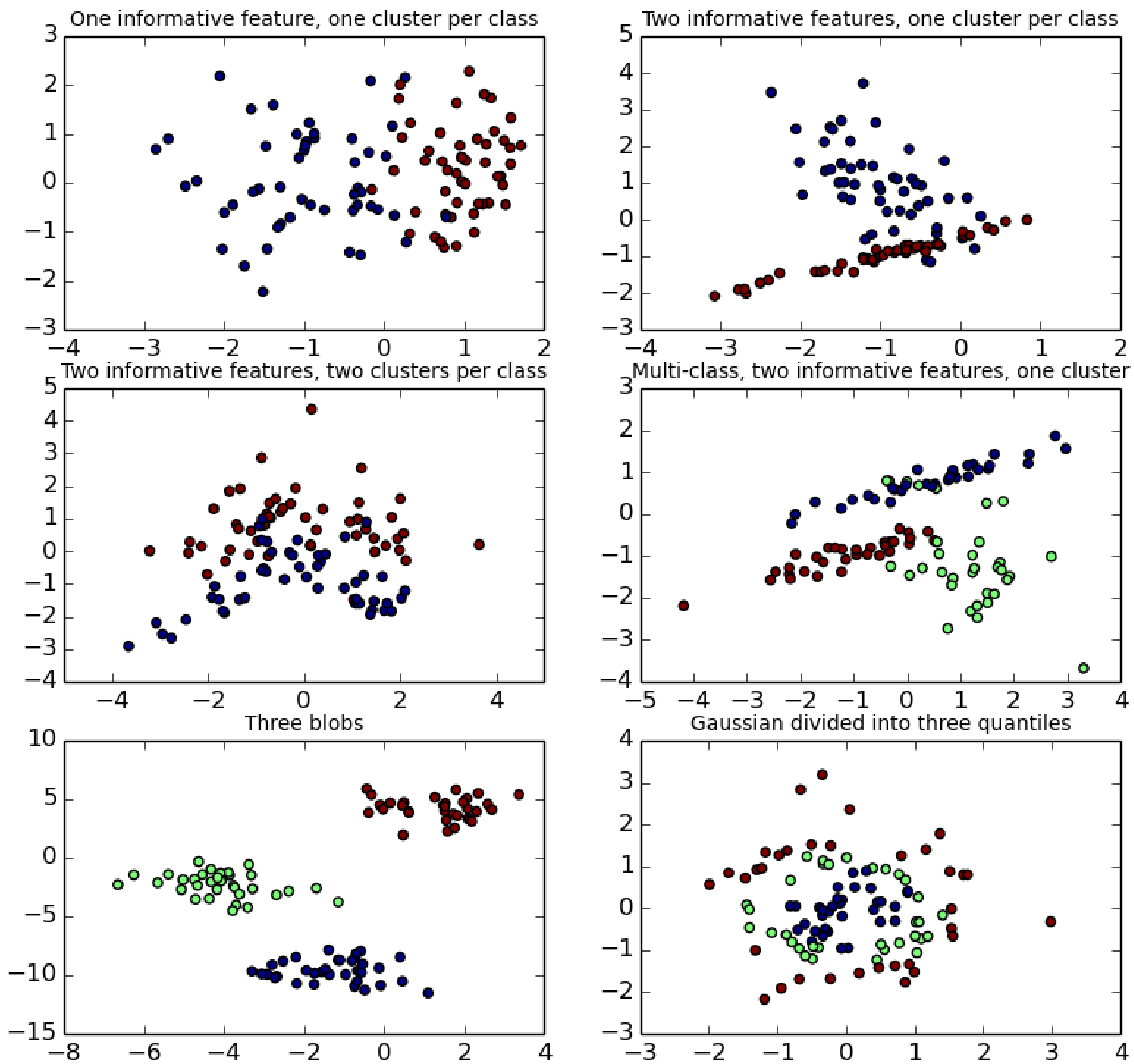


Figure 11.66: *Plot randomly generated classification dataset*

11.9.2 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.



Python source code: `plot_random_dataset.py`

```
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,
                             n_clusters_per_class=1)
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1)

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)

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)


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)

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)

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)

plt.show()
```

Total running time of the example: 0.21 seconds (0 minutes 0.21 seconds)

11.9.3 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.

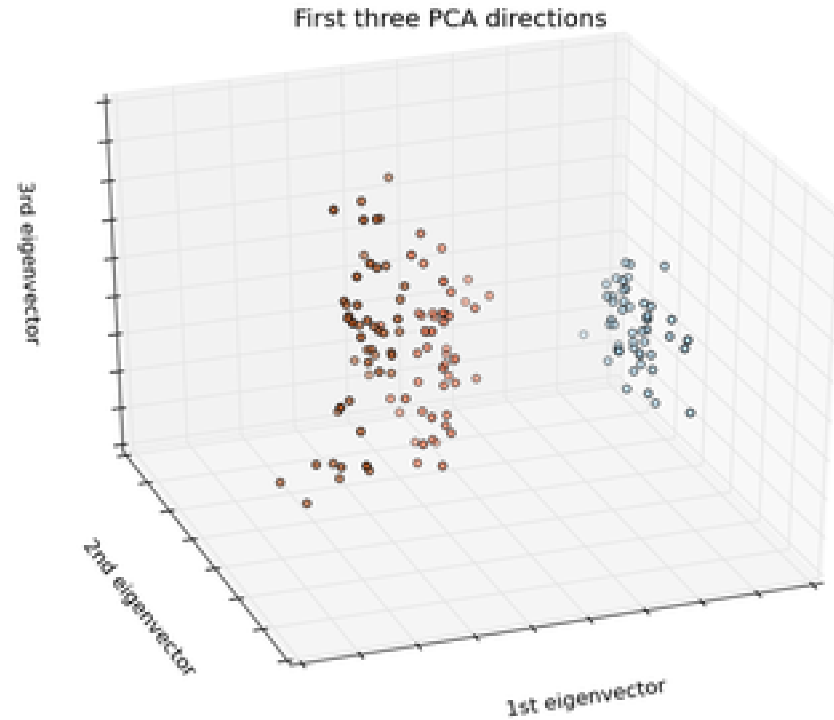
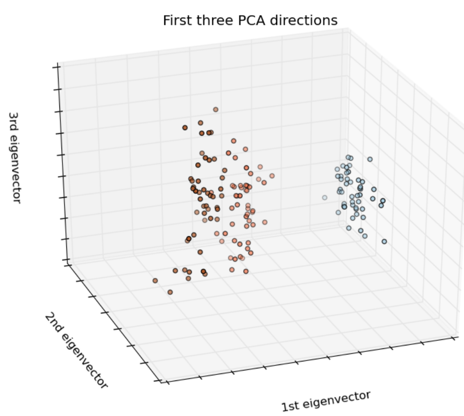
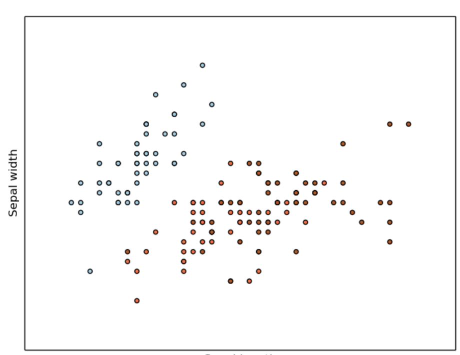


Figure 11.67: *The Iris Dataset*

- 

- 

Python source code: plot_iris_dataset.py

```
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.Paired)
plt.xlabel('Sepal length')
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.Paired)
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 example: 0.13 seconds (0 minutes 0.13 seconds)

11.9.4 Plot randomly generated multilabel dataset

This illustrates the *datasets.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.

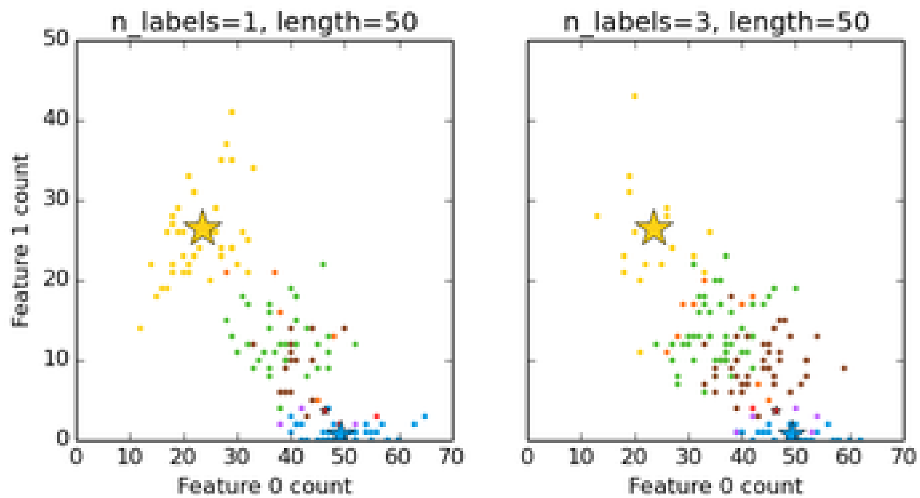


Figure 11.68: *Plot randomly generated multilabel dataset*

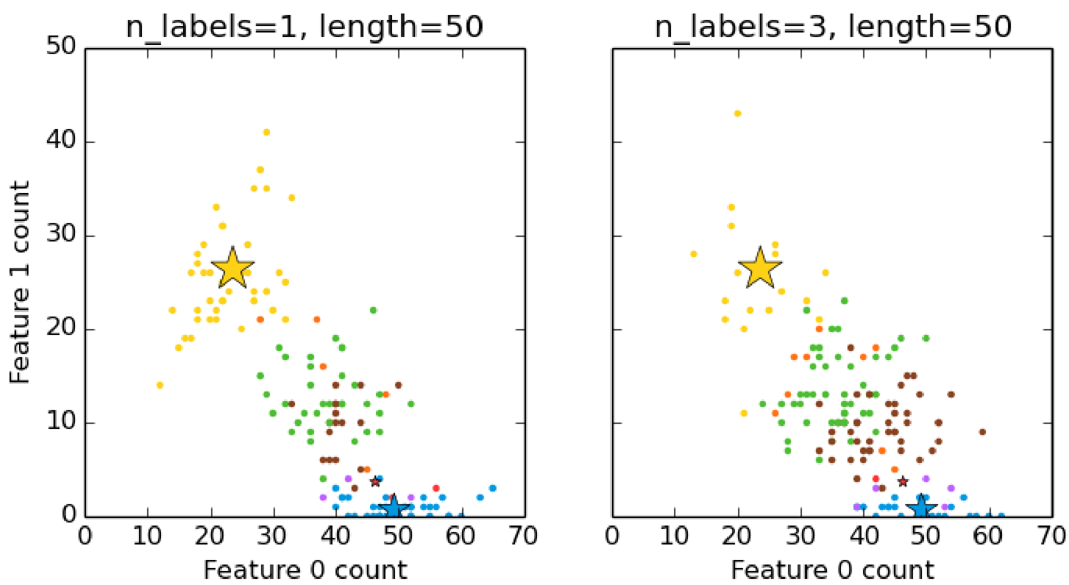
Points are labeled as follows, where Y means the class is present:

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.



Script output:

```
The data was generated from (random_state=268):
Class    P(C)     P(w0|C)  P(w1|C)
red      0.08     0.93     0.07
blue     0.38     0.99     0.01
yellow   0.54     0.47     0.53
```

Python source code: `plot_random_multilabel_dataset.py`

```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt

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_indicator=True,
                                   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 example: 0.14 seconds (0 minutes 0.14 seconds)

11.10 Decomposition

Examples concerning the `sklearn.decomposition` module.

11.10.1 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.

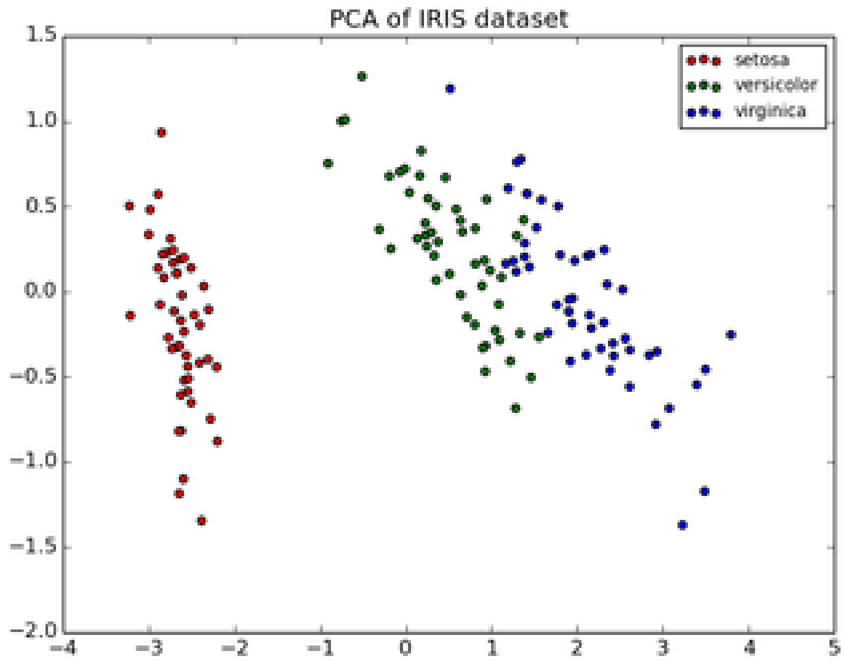
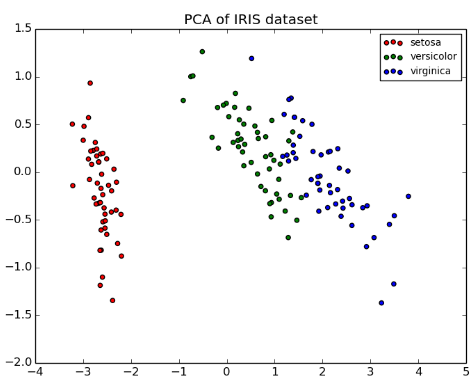
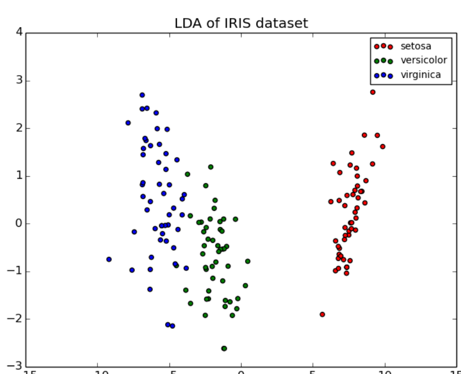


Figure 11.69: *Comparison of LDA and PCA 2D projection of Iris dataset*

- 

- 

Script output:

```
explained variance ratio (first two components): [ 0.92461621  0.05301557]
```

Python source code: `plot_pca_vs_lda.py`

```python
print(__doc__)

import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.lda import LDA

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 = LDA(n_components=2)
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()
for c, i, target_name in zip("rgb", [0, 1, 2], target_names):
    plt.scatter(X_r[y == i, 0], X_r[y == i, 1], c=c, label=target_name)
plt.legend()
plt.title('PCA of IRIS dataset')

plt.figure()
for c, i, target_name in zip("rgb", [0, 1, 2], target_names):
    plt.scatter(X_r2[y == i, 0], X_r2[y == i, 1], c=c, label=target_name)
plt.legend()
plt.title('LDA of IRIS dataset')

plt.show()
```

Total running time of the example: 0.12 seconds (0 minutes 0.12 seconds)

11.10.2 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.

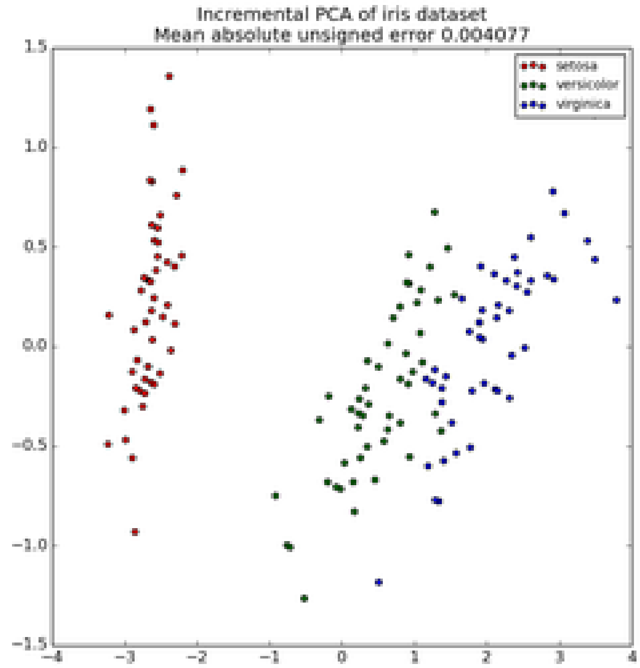
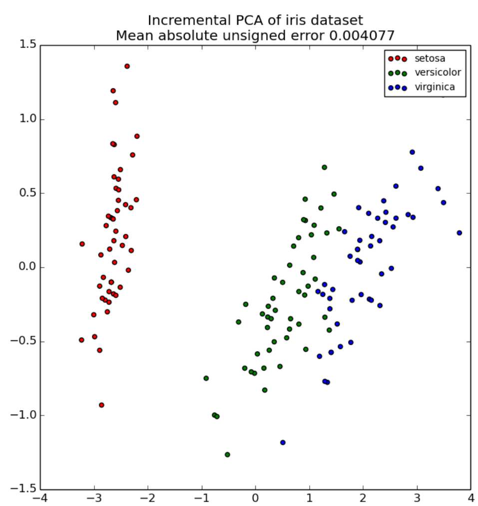
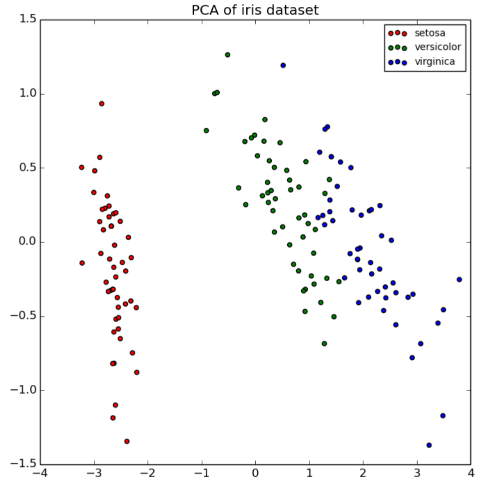


Figure 11.70: *Incremental PCA*

- 



•

Python source code: `plot_incremental_pca.py`

```
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
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)

for X_transformed, title in [(X_ipca, "Incremental PCA"), (X_pca, "PCA")]:
    plt.figure(figsize=(8, 8))
    for c, i, target_name in zip("rgb", [0, 1, 2], iris.target_names):
        plt.scatter(X_transformed[y == i, 0], X_transformed[y == i, 1],
                    c=c, 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")
    plt.axis([-4, 4, -1.5, 1.5])
```

```
plt.show()
```

Total running time of the example: 0.13 seconds (0 minutes 0.13 seconds)

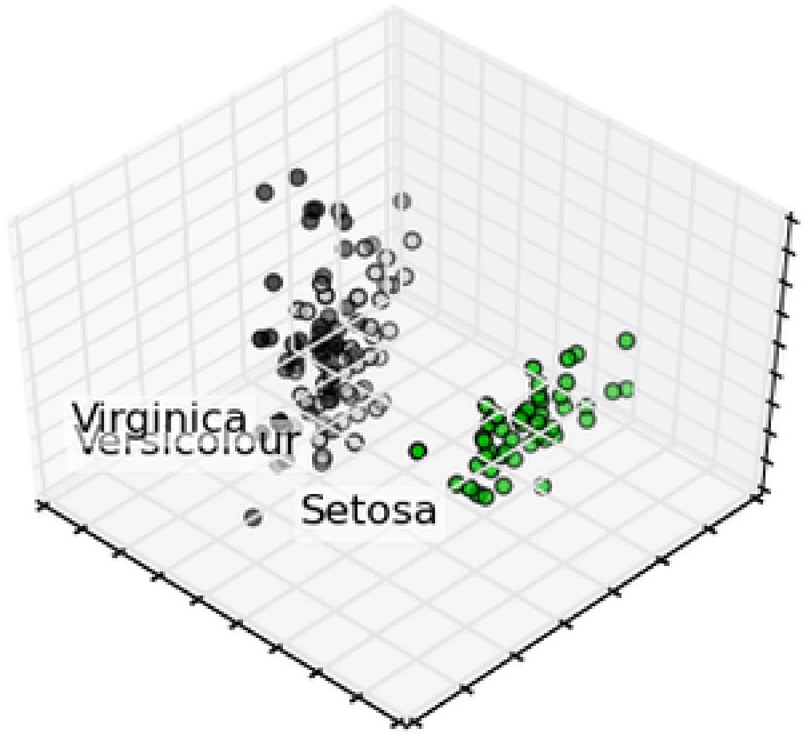
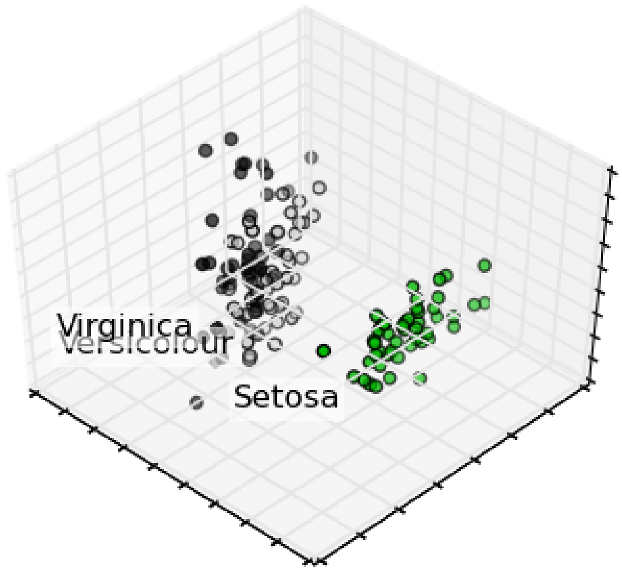


Figure 11.71: *PCA example with Iris Data-set*

11.10.3 PCA example with Iris Data-set

Principal Component Analysis applied to the Iris dataset.

See here for more information on this dataset.



Python source code: `plot_pca_iris.py`

```
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)
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.spectral)

x_surf = [X[:, 0].min(), X[:, 0].max(),
          X[:, 0].min(), X[:, 0].max()]
y_surf = [X[:, 0].max(), X[:, 0].max(),
          X[:, 0].min(), X[:, 0].min()]
x_surf = np.array(x_surf)
y_surf = np.array(y_surf)
v0 = pca.transform(pca.components_[0])
v0 /= v0[-1]
v1 = pca.transform(pca.components_[1])
v1 /= v1[-1]

ax.w_xaxis.set_ticklabels([])
ax.w_yaxis.set_ticklabels([])
ax.w_zaxis.set_ticklabels([])

plt.show()
```

Total running time of the example: 0.10 seconds (0 minutes 0.10 seconds)

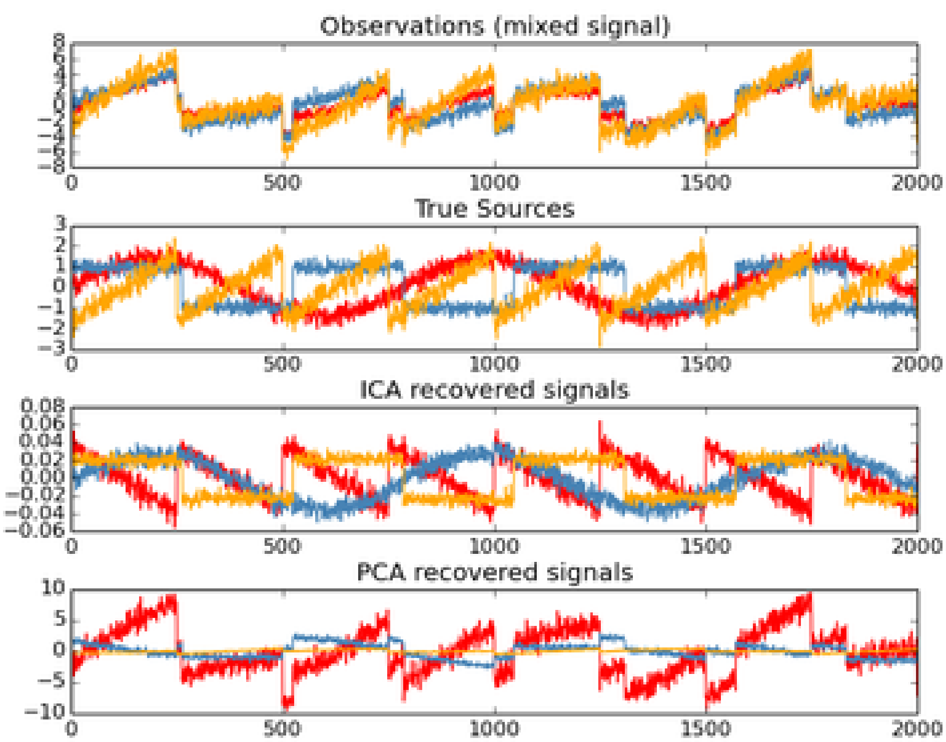
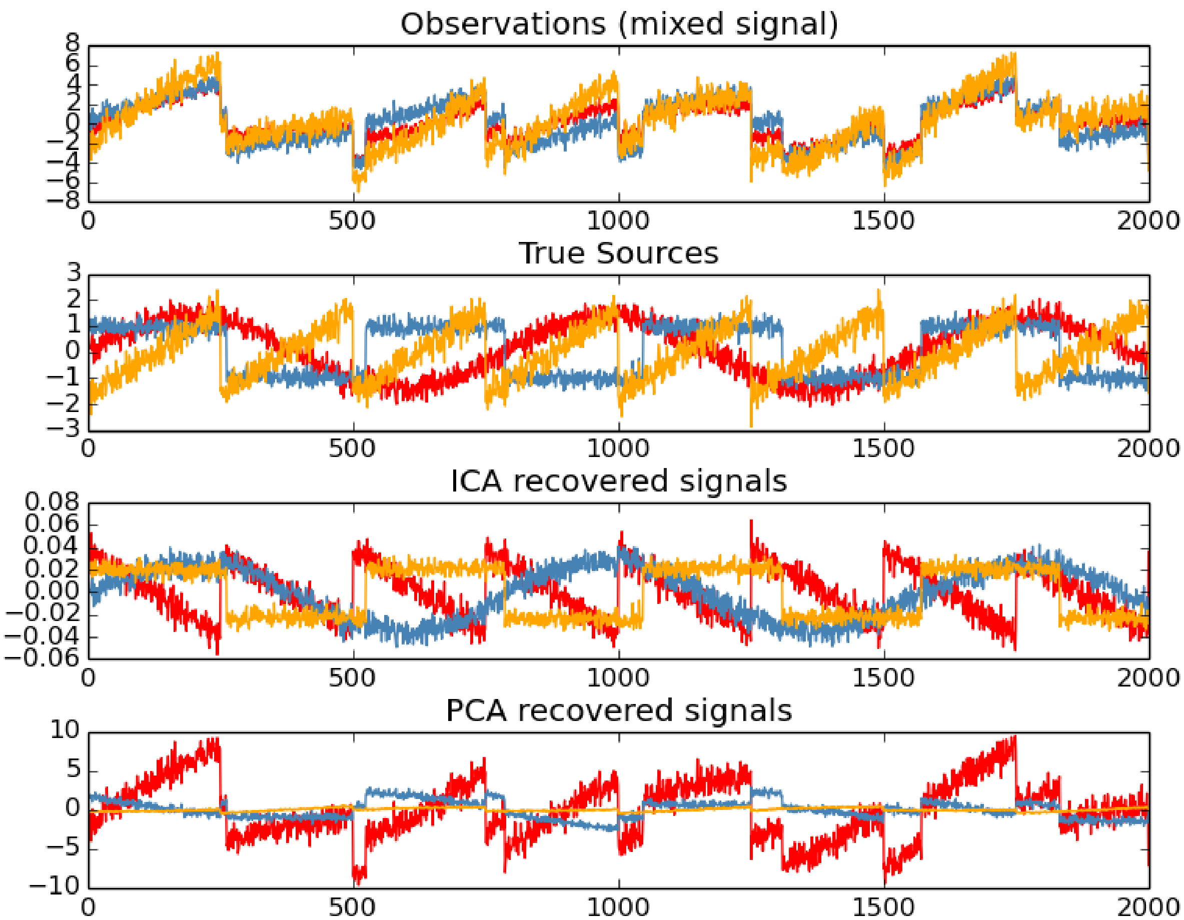


Figure 11.72: *Blind source separation using FastICA*

11.10.4 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.



Python source code: `plot_ica_blind_source_separation.py`

```
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]
```

```python
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

# 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.subplots_adjust(0.09, 0.04, 0.94, 0.94, 0.26, 0.46)
plt.show()
```

Total running time of the example: 0.17 seconds (0 minutes 0.17 seconds)

11.10.5 Kernel PCA

This example shows that Kernel PCA is able to find a projection of the data that makes data linearly separable.

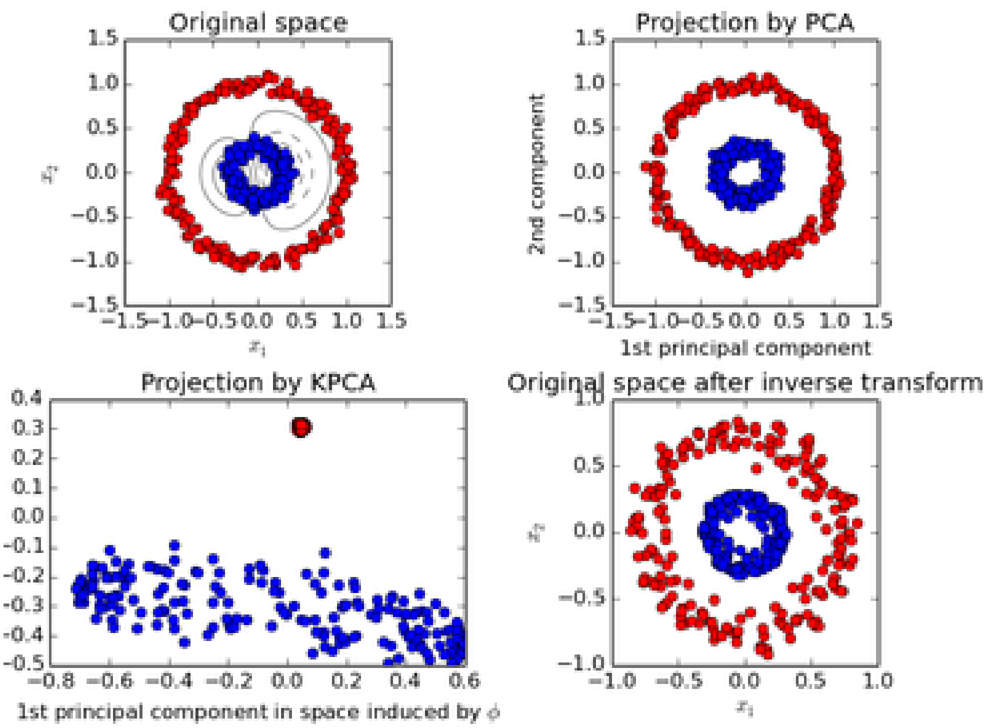
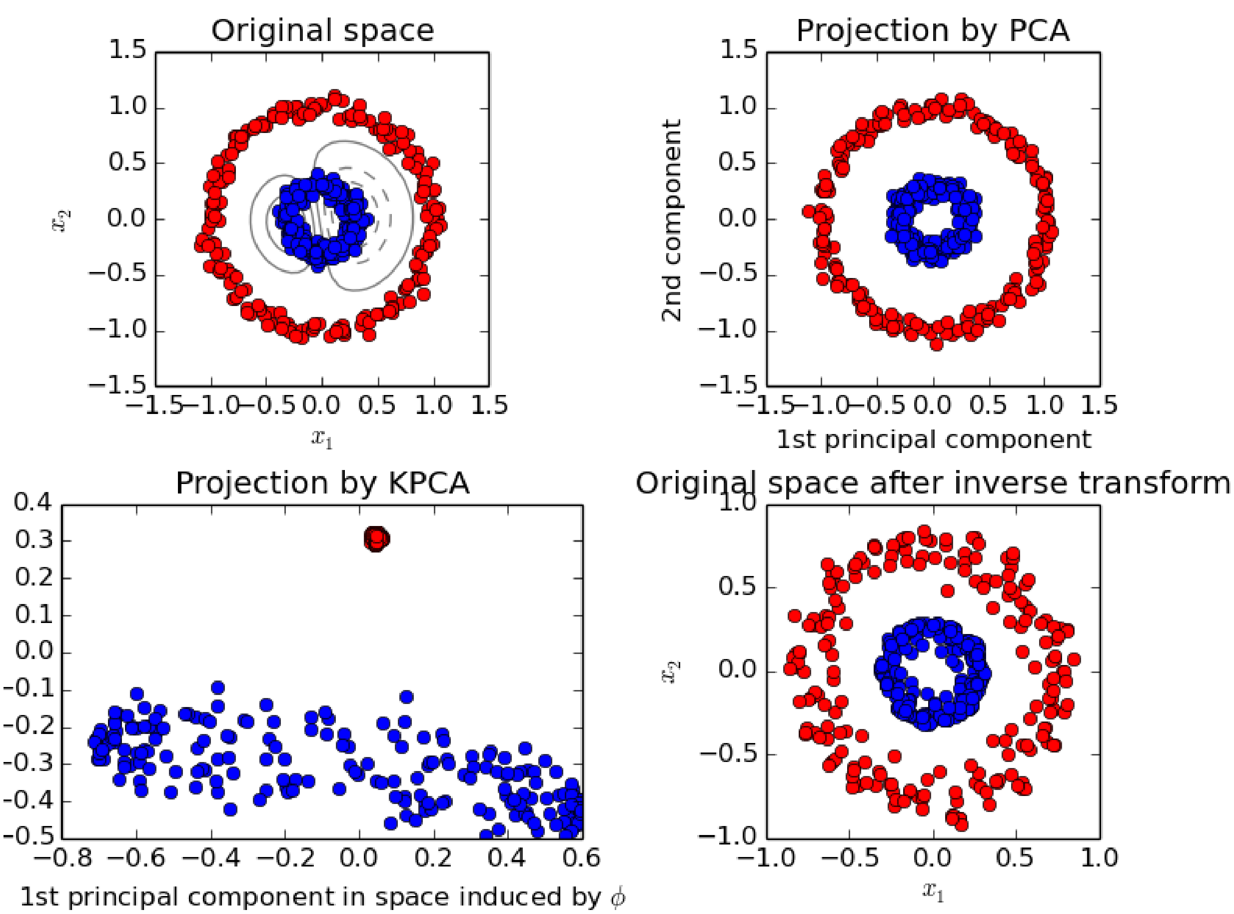


Figure 11.73: *Kernel PCA*



Python source code: `plot_kernel_pca.py`

```
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)

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.plot(X[reds, 0], X[reds, 1], "ro")
plt.plot(X[blues, 0], X[blues, 1], "bo")
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.plot(X_pca[reds, 0], X_pca[reds, 1], "ro")
plt.plot(X_pca[blues, 0], X_pca[blues, 1], "bo")
plt.title("Projection by PCA")
plt.xlabel("1st principal component")
plt.ylabel("2nd component")

plt.subplot(2, 2, 3, aspect='equal')
plt.plot(X_kpca[reds, 0], X_kpca[reds, 1], "ro")
plt.plot(X_kpca[blues, 0], X_kpca[blues, 1], "bo")
plt.title("Projection by KPCA")
plt.xlabel("1st principal component in space induced by $\phi$")
plt.ylabel("2nd component")

plt.subplot(2, 2, 4, aspect='equal')
plt.plot(X_back[reds, 0], X_back[reds, 1], "ro")
```

```
plt.plot(X_back[blues, 0], X_back[blues, 1], "bo")
plt.title("Original space after inverse transform")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")

plt.subplots_adjust(0.02, 0.10, 0.98, 0.94, 0.04, 0.35)

plt.show()
```

Total running time of the example: 1.01 seconds (0 minutes 1.01 seconds)

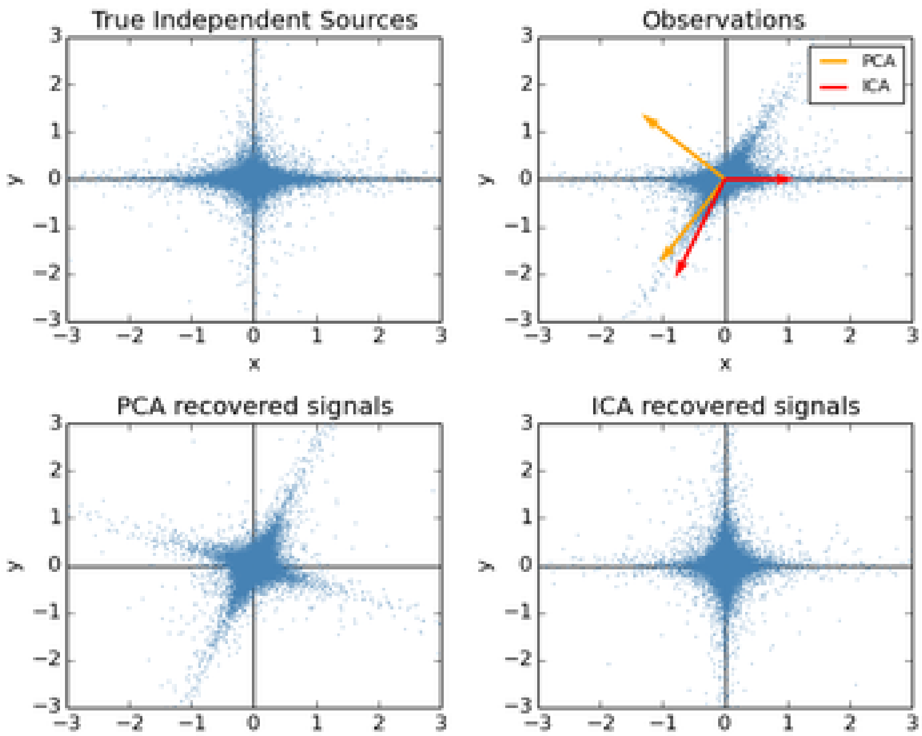


Figure 11.74: *FastICA on 2D point clouds*

11.10.6 FastICA on 2D point clouds

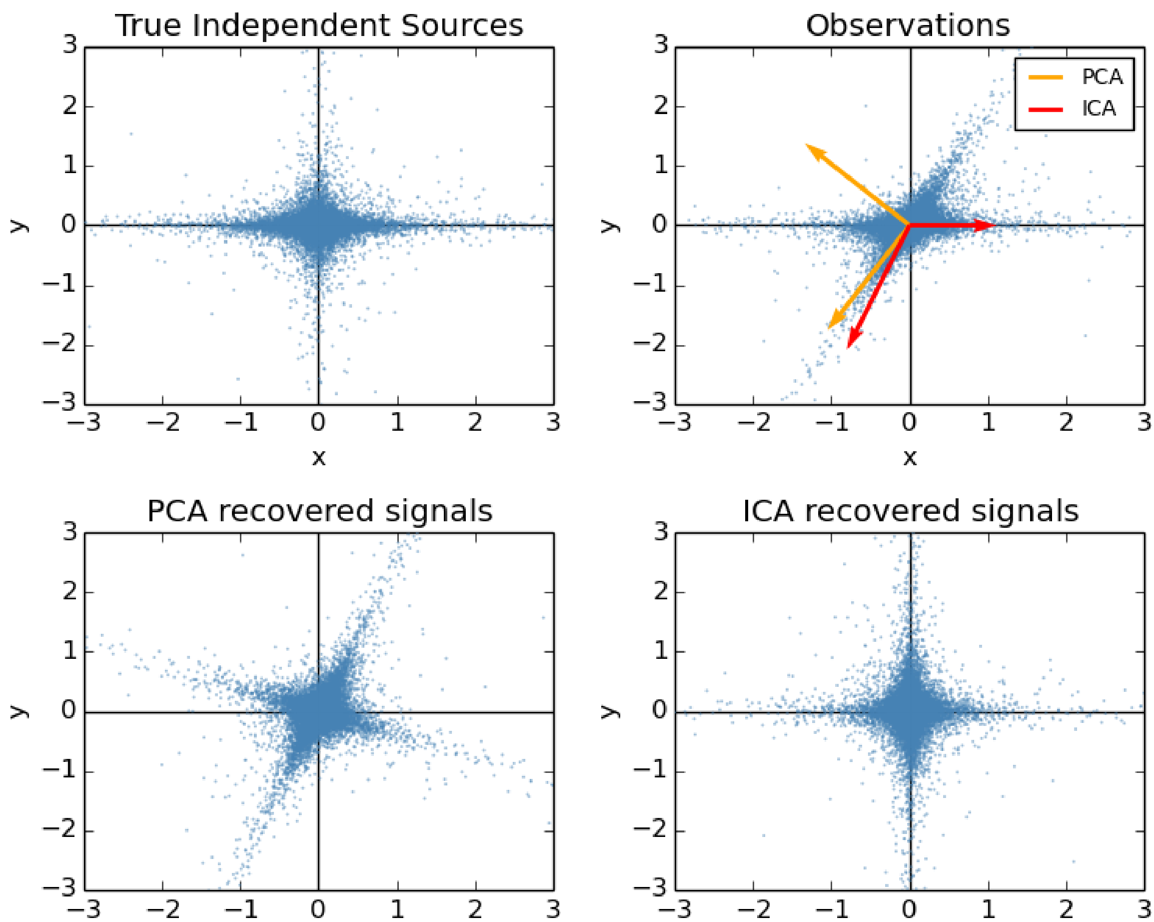
This example illustrates visually in the feature space a comparison by results using two different component analysis techniques.

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).



Python source code: `plot_ica_vs_pca.py`

```
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)
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', linewidths=0, 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()
```

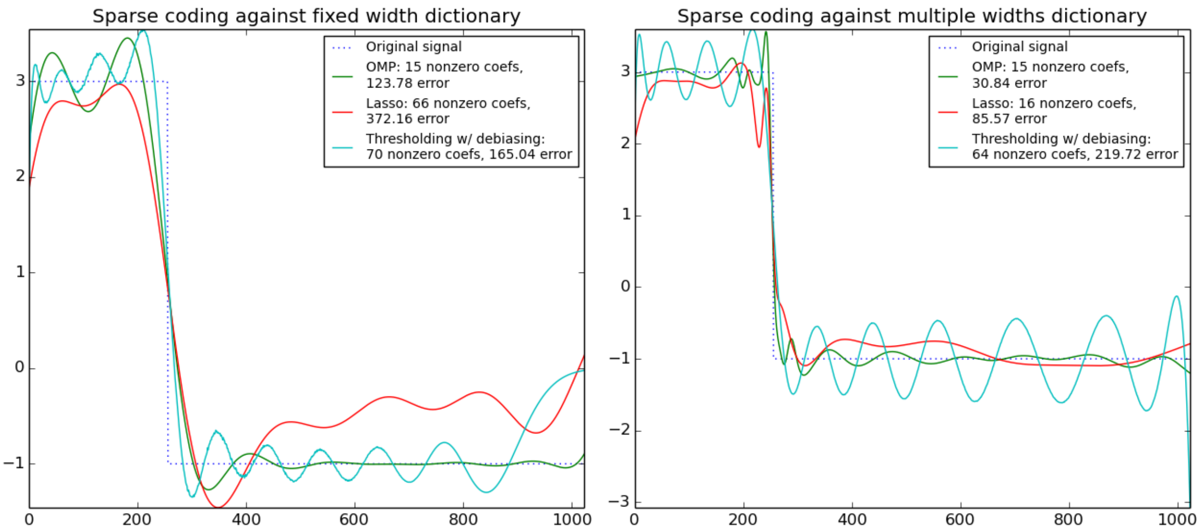
Total running time of the example: 0.45 seconds (0 minutes 0.45 seconds)

Figure 11.75: *Sparse coding with a precomputed dictionary*

11.10.7 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.



Python source code: `plot_sparse_coding.py`

```
print(__doc__)

import numpy as np
import matplotlib.pylab as pl

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 ** 1 / 4)))
         * (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)
D_multi = np.r_[tuple(ricker_matrix(width=w, resolution=resolution,
```

```
                                          n_components=np.floor(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)
estimators = [('OMP', 'omp', None, 15), ('Lasso', 'lasso_cd', 2, None), ]

pl.figure(figsize=(13, 6))
for subplot, (D, title) in enumerate(zip((D_fixed, D_multi),
                                         ('fixed width', 'multiple widths'))):
    pl.subplot(1, 2, subplot + 1)
    pl.title('Sparse coding against %s dictionary' % title)
    pl.plot(y, ls='dotted', label='Original signal')
    # Do a wavelet approximation
    for title, algo, alpha, n_nonzero in estimators:
        coder = SparseCoder(dictionary=D, transform_n_nonzero_coefs=n_nonzero,
                            transform_alpha=alpha, transform_algorithm=algo)
        x = coder.transform(y)
        density = len(np.flatnonzero(x))
        x = np.ravel(np.dot(x, D))
        squared_error = np.sum((y - x) ** 2)
        pl.plot(x, 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)
    _, idx = np.where(x != 0)
    x[0, idx], _, _, _ = np.linalg.lstsq(D[idx, :].T, y)
    x = np.ravel(np.dot(x, D))
    squared_error = np.sum((y - x) ** 2)
    pl.plot(x,
            label='Thresholding w/ debiasing:\n%d nonzero coefs, %.2f error' %
            (len(idx), squared_error))
    pl.axis('tight')
    pl.legend()
pl.subplots_adjust(.04, .07, .97, .90, .09, .2)
pl.show()
```

Total running time of the example: 0.77 seconds (0 minutes 0.77 seconds)

11.10.8 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.

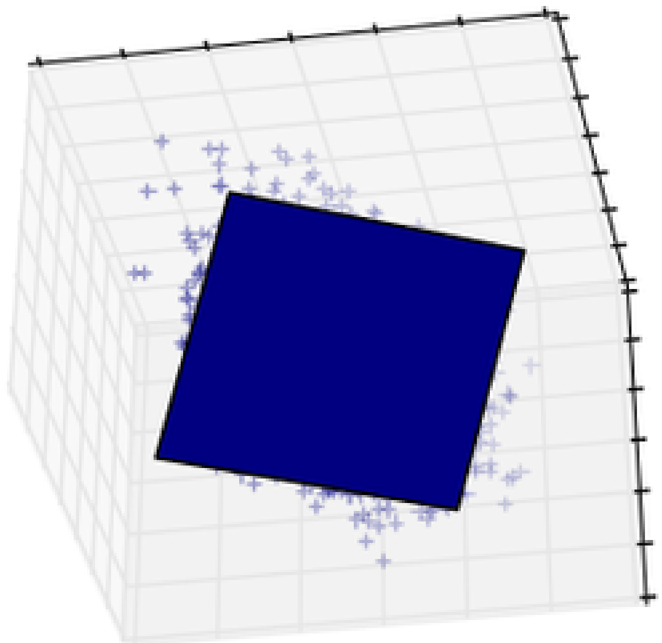
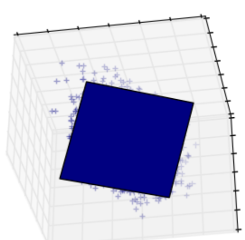
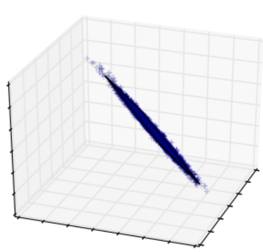


Figure 11.76: *Principal components analysis (PCA)*

- 
- 

Python source code: `plot_pca_3d.py`

```
print(__doc__)

# Authors: Gael Varoquaux
#          Jaques Grobler
#          Kevin Hughes
# License: BSD 3 clause
```

```python
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

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 = V.T * pca_score / pca_score.min()

    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([])


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 example: 0.16 seconds (0 minutes 0.16 seconds)

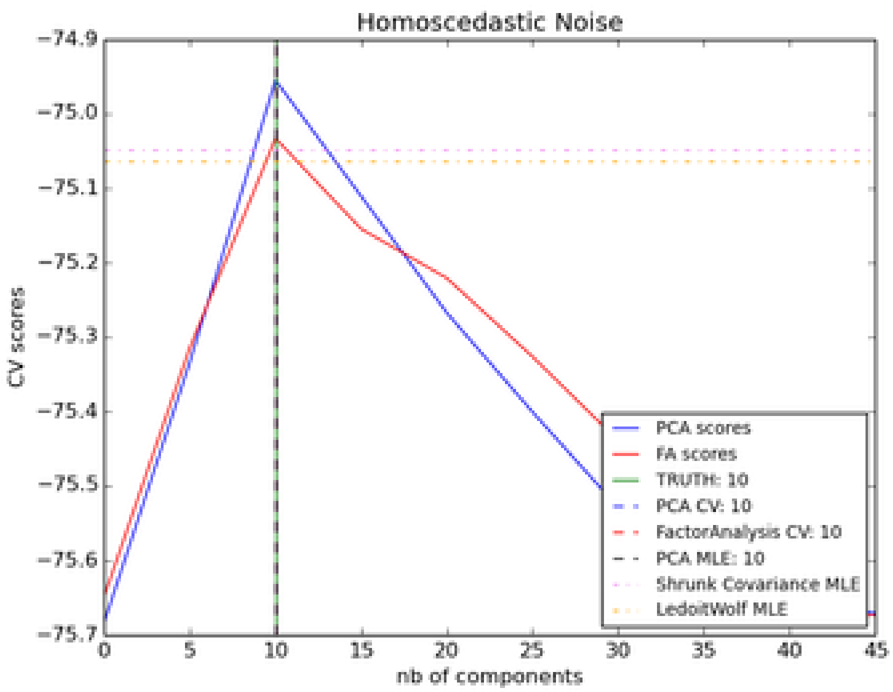


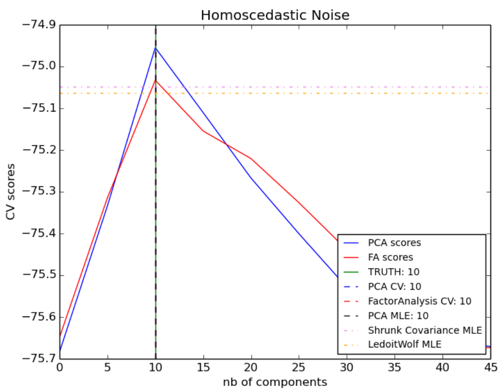
Figure 11.77: *Model selection with Probabilistic (PCA) and Factor Analysis (FA)*

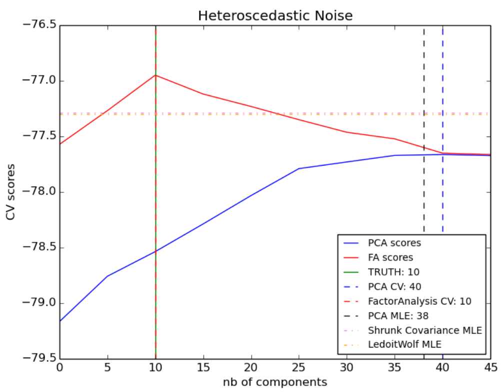
11.10.9 Model selection with Probabilistic (PCA) and Factor Analysis (FA)

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.

- 

- 

Script output:

```
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 = 40
best n_components by FactorAnalysis CV = 10
best n_components by PCA MLE = 38
```

Python source code: `plot_pca_vs_fa_model_selection.py`

```
print(__doc__)
```

```python
# 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.cross_validation import cross_val_score
from sklearn.grid_search import GridSearchCV

###############################################################################
# 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()
    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(n_components='mle')
    pca.fit(X)
    n_components_pca_mle = pca.n_components_

    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 example: 32.53 seconds (0 minutes 32.53 seconds)

11.10.10 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)*) .

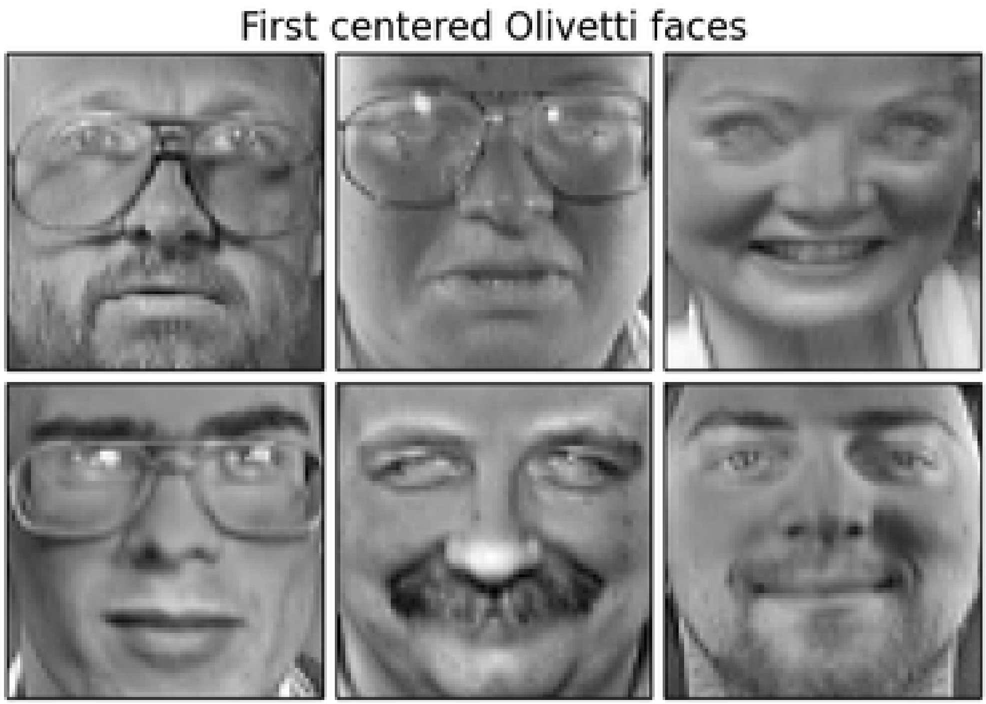
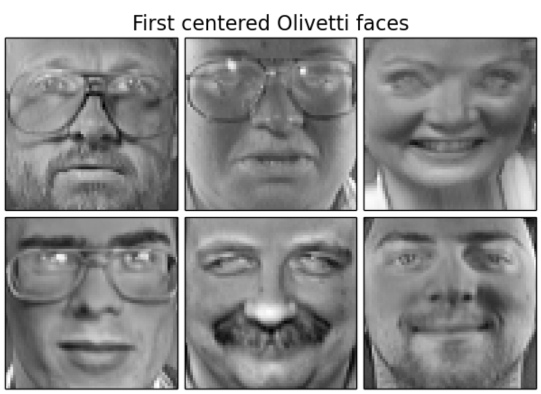
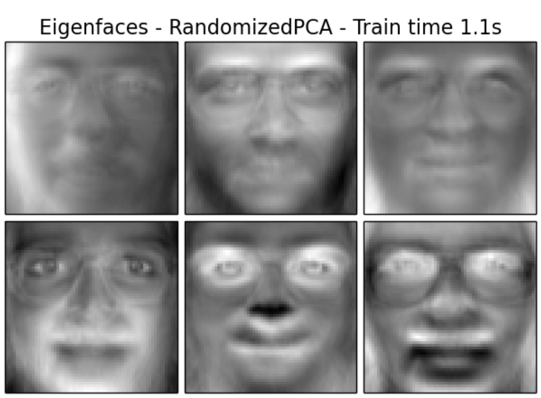
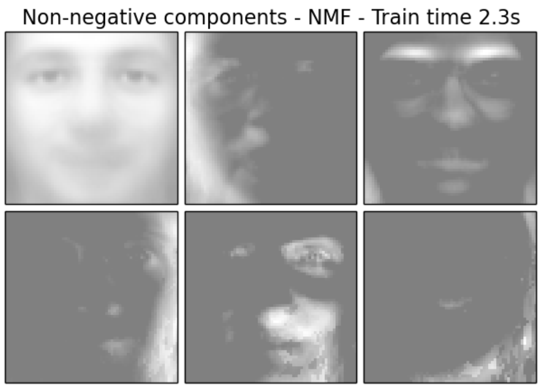
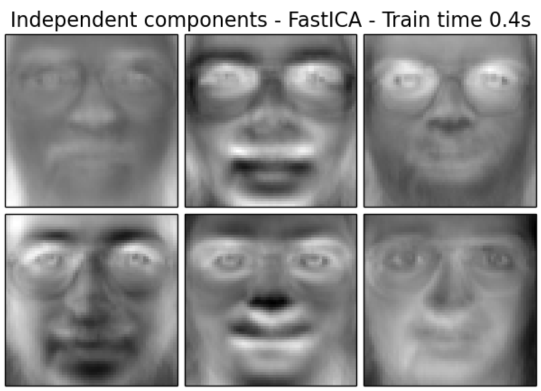


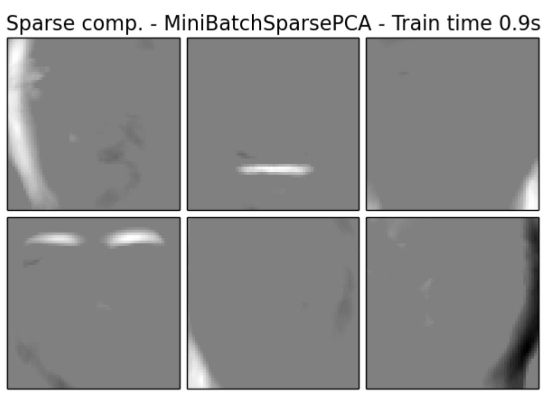
Figure 11.78: *Faces dataset decompositions*

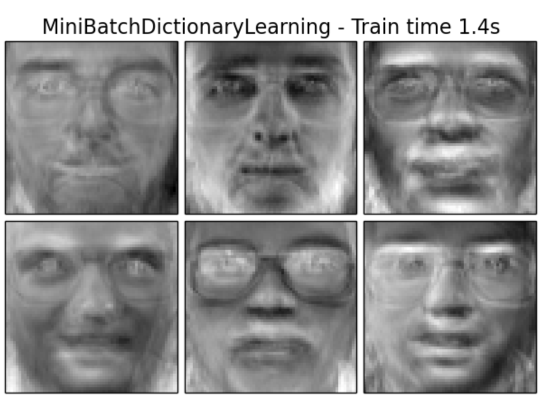
- 

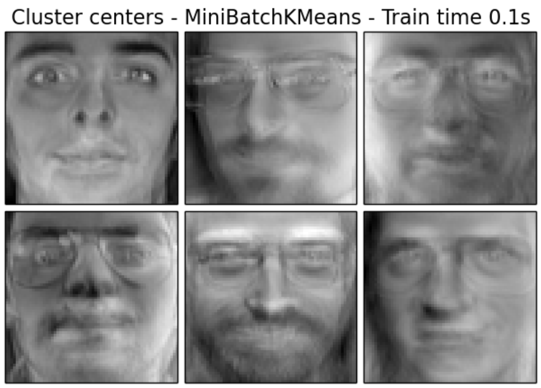
- 

- 
Non-negative components - NMF - Train time 2.3s

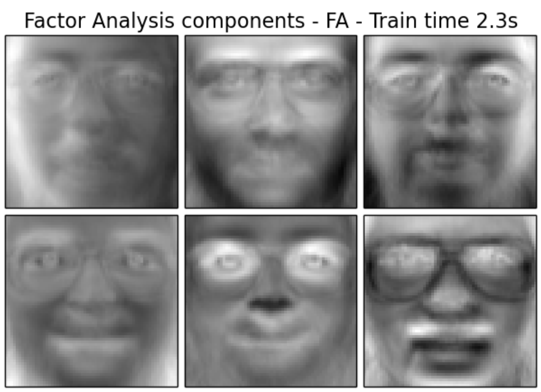
- 
Independent components - FastICA - Train time 0.4s

- 
Sparse comp. - MiniBatchSparsePCA - Train time 0.9s

- 
MiniBatchDictionaryLearning - Train time 1.4s

- 

- 

- 

Script output:

```
Dataset consists of 400 faces
Extracting the top 6 Eigenfaces - RandomizedPCA...
done in 1.104s
Extracting the top 6 Non-negative components - NMF...
done in 2.278s
Extracting the top 6 Independent components - FastICA...
done in 0.419s
Extracting the top 6 Sparse comp. - MiniBatchSparsePCA...
done in 0.905s
Extracting the top 6 MiniBatchDictionaryLearning...
done in 1.377s
Extracting the top 6 Cluster centers - MiniBatchKMeans...
done in 0.070s
Extracting the top 6 Factor Analysis components - FA...
done in 2.255s
```

Python source code: `plot_faces_decomposition.py`

```
print(__doc__)
```

```python
# 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
dataset = fetch_olivetti_faces(shuffle=True, random_state=rng)
faces = dataset.data

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):
    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=plt.cm.gray,
                   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 - RandomizedPCA',
     decomposition.RandomizedPCA(n_components=n_components, whiten=True),
     True),
```

```
    ('Non-negative components - NMF',
     decomposition.NMF(n_components=n_components, init='nndsvda', beta=5.0,
                       tol=5e-3, sparseness='components'),
     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=2),
     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_
    if hasattr(estimator, 'noise_variance_'):
        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()
```

Total running time of the example: 9.73 seconds (0 minutes 9.73 seconds)

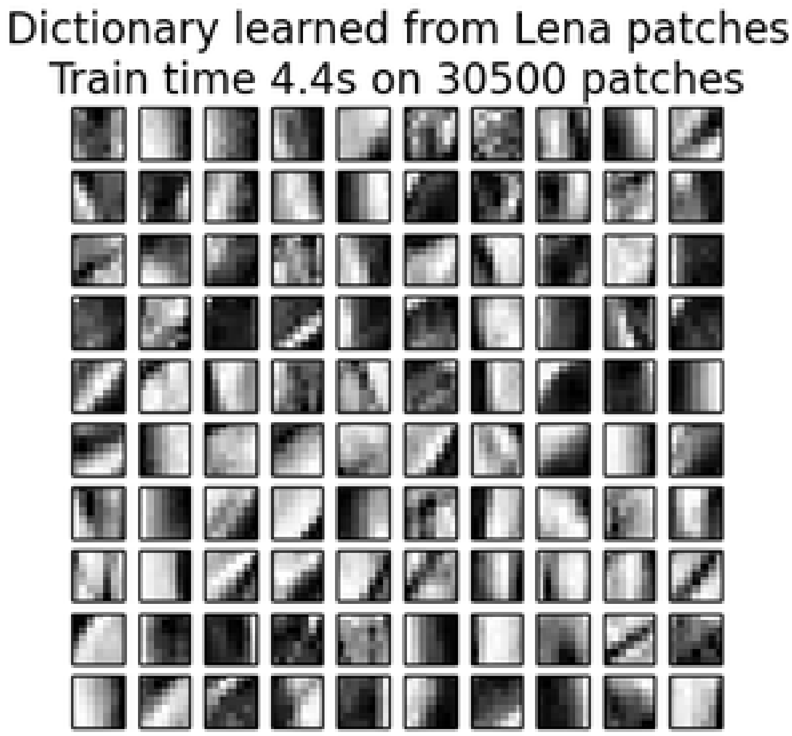


Figure 11.79: *Image denoising using dictionary learning*

11.10.11 Image denoising using dictionary learning

An example comparing the effect of reconstructing noisy fragments of the Lena image using firstly online *Dictionary Learning* and various transform methods.

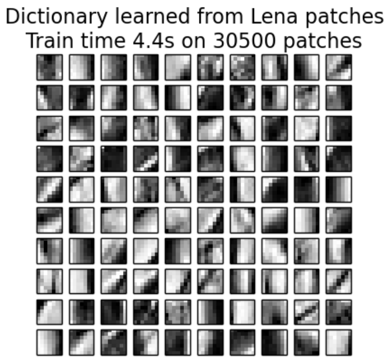
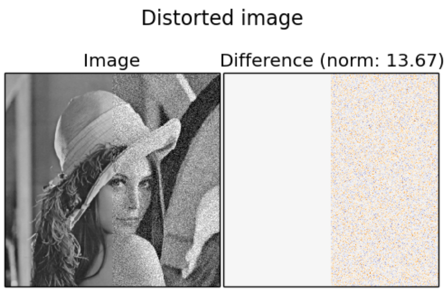
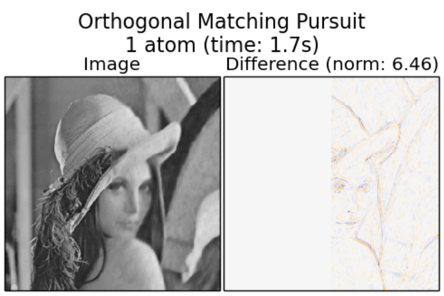
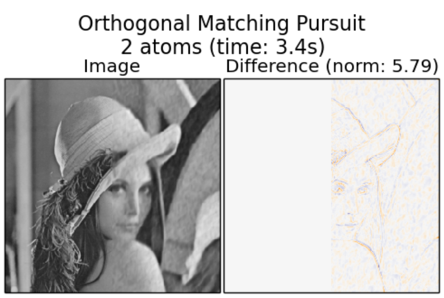
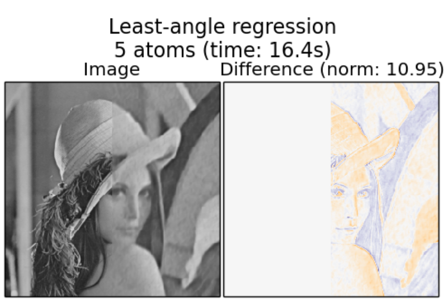
The dictionary is fitted on the distorted left half of the image, and subsequently used to reconstruct the right half. Note that even better performance could be achieved by fitting to an undistorted (i.e. noiseless) image, but here we start from the assumption that it is not available.

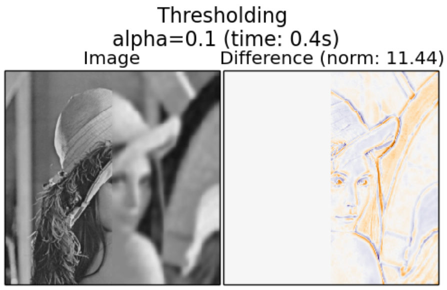
A common practice for evaluating the results of image denoising is by looking at the difference between the reconstruction and the original image. If the reconstruction is perfect this will look like Gaussian noise.

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.

- 
Dictionary learned from Lena patches
Train time 4.4s on 30500 patches
- 
Distorted image
Image
Difference (norm: 13.67)
- 
Orthogonal Matching Pursuit
1 atom (time: 1.7s)
Image
Difference (norm: 6.46)
- 
Orthogonal Matching Pursuit
2 atoms (time: 3.4s)
Image
Difference (norm: 5.79)
- 
Least-angle regression
5 atoms (time: 16.4s)
Image
Difference (norm: 10.95)

- 

Script output:

```
Distorting image...
Extracting reference patches...
done in 0.02s.
Learning the dictionary...
done in 4.41s.
Extracting noisy patches...
done in 0.00s.
Orthogonal Matching Pursuit
1 atom...
done in 1.75s.
Orthogonal Matching Pursuit
2 atoms...
done in 3.43s.
Least-angle regression
5 atoms...
done in 16.40s.
Thresholding
 alpha=0.1...
done in 0.44s.
```

Python source code: `plot_image_denoising.py`

```python
print(__doc__)

from time import time

import matplotlib.pyplot as plt
import numpy as np

from scipy.misc import lena

from sklearn.decomposition import MiniBatchDictionaryLearning
from sklearn.feature_extraction.image import extract_patches_2d
from sklearn.feature_extraction.image import reconstruct_from_patches_2d

###############################################################################
# Load Lena image and extract patches

lena = lena() / 256.0

# downsample for higher speed
lena = lena[::2, ::2] + lena[1::2, ::2] + lena[::2, 1::2] + lena[1::2, 1::2]
lena /= 4.0
height, width = lena.shape

# Distort the right half of the image
```

```
print('Distorting image...')
distorted = lena.copy()
distorted[:, height // 2:] += 0.075 * np.random.randn(width, height // 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[:, :height // 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 Lena 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)


###############################################################################
# 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, lena, 'Distorted image')

###############################################################################
# Extract noisy patches and reconstruct them using the dictionary

print('Extracting noisy patches... ')
t0 = time()
data = extract_patches_2d(distorted[:, height // 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] = lena.copy()
    t0 = time()
    dico.set_params(transform_algorithm=transform_algorithm, **kwargs)
    code = dico.transform(data)
    patches = np.dot(code, V)

    if transform_algorithm == 'threshold':
        patches -= patches.min()
        patches /= patches.max()

    patches += intercept
    patches = patches.reshape(len(data), *patch_size)
    if transform_algorithm == 'threshold':
        patches -= patches.min()
        patches /= patches.max()
    reconstructions[title][:, height // 2:] = reconstruct_from_patches_2d(
        patches, (width, height // 2))
    dt = time() - t0
    print('done in %.2fs.' % dt)
    show_with_diff(reconstructions[title], lena,
                   title + ' (time: %.1fs)' % dt)

plt.show()
```

Total running time of the example: 29.70 seconds (0 minutes 29.70 seconds)

11.11 Ensemble methods

Examples concerning the `sklearn.ensemble` module.

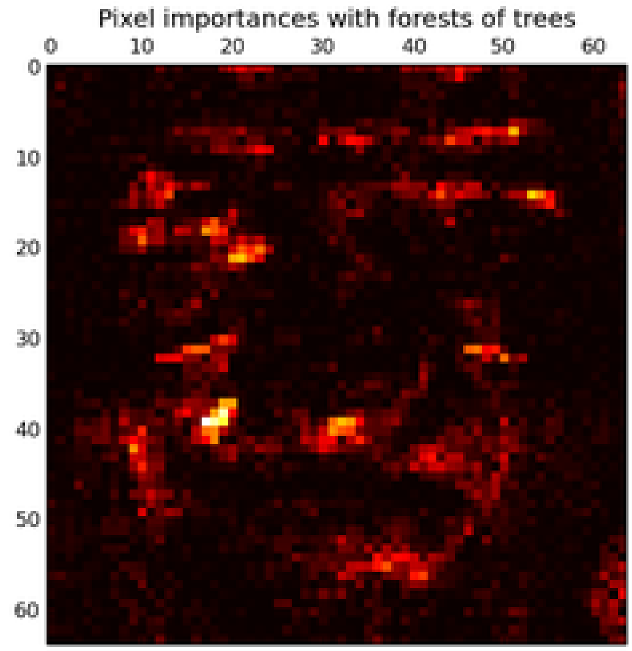
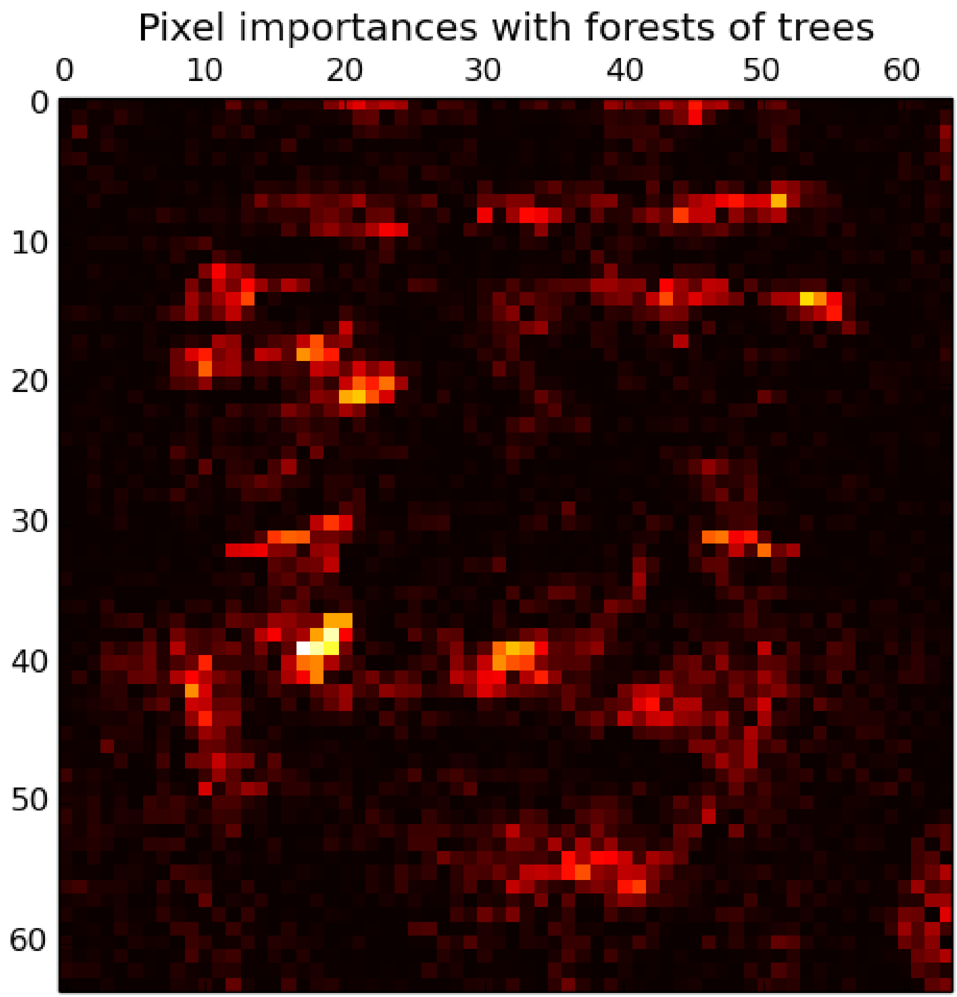


Figure 11.80: *Pixel importances with a parallel forest of trees*

11.11.1 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.



Script output:

```
Fitting ExtraTreesClassifier on faces data with 1 cores...
done in 0.620s
```

Python source code: `plot_forest_importances_faces.py`

```python
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 = data.images.reshape((len(data.images), -1))
y = data.target

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 example: 0.96 seconds (0 minutes 0.96 seconds)

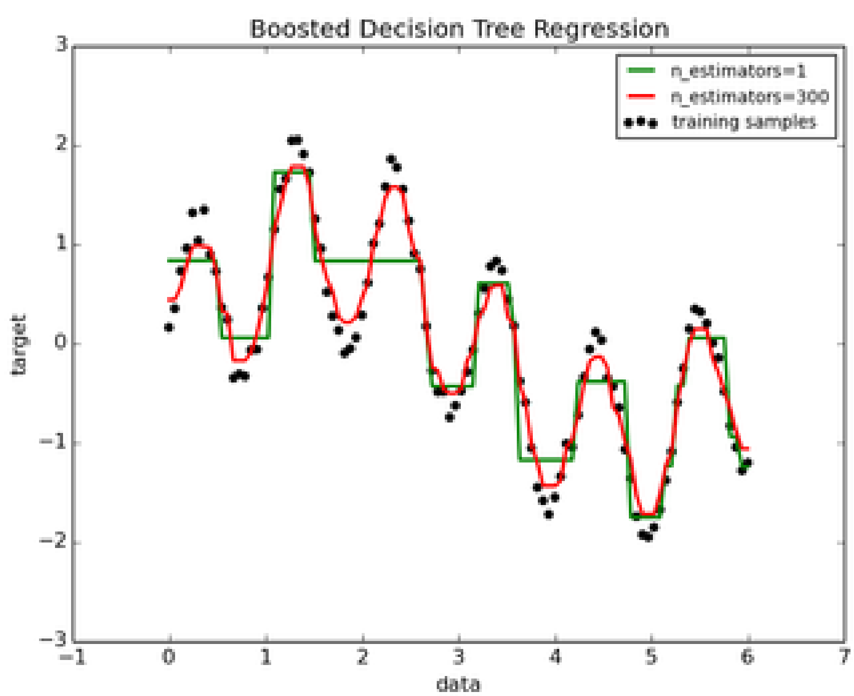
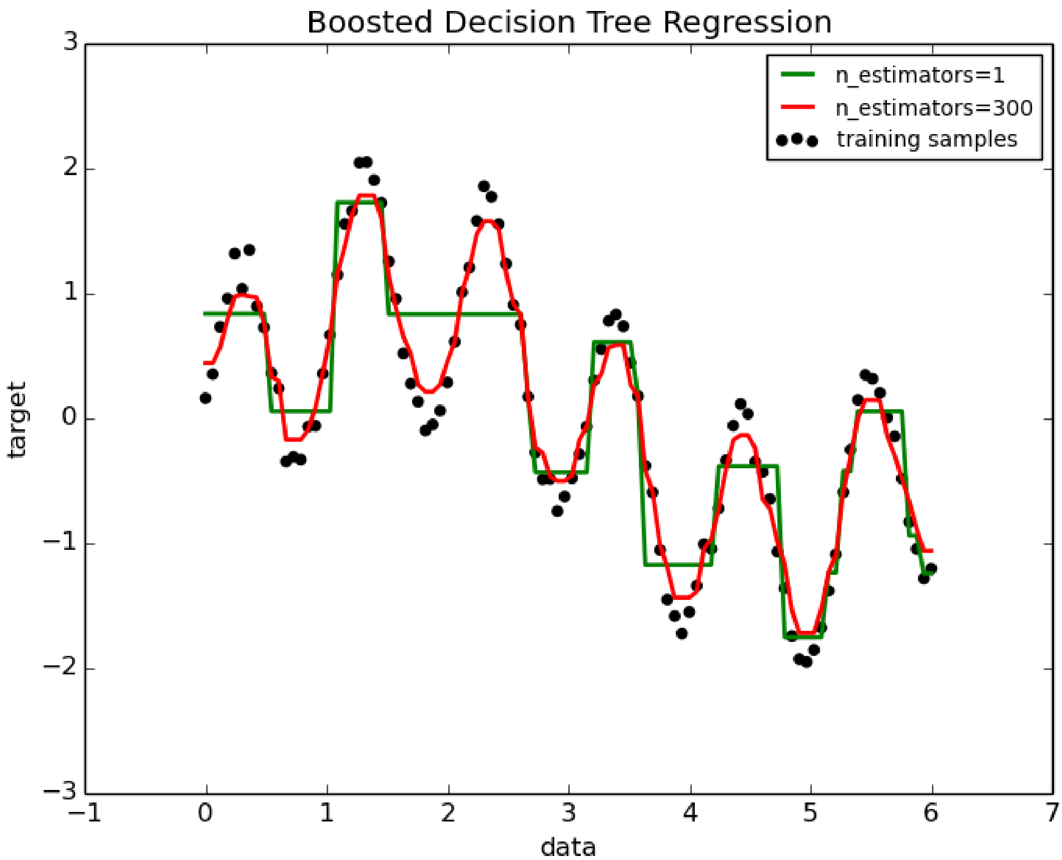


Figure 11.81: *Decision Tree Regression with AdaBoost*

11.11.2 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.



Python source code: `plot_adaboost_regression.py`

```
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
```

```
clf_1 = DecisionTreeRegressor(max_depth=4)

clf_2 = AdaBoostRegressor(DecisionTreeRegressor(max_depth=4),
                          n_estimators=300, random_state=rng)

clf_1.fit(X, y)
clf_2.fit(X, y)

# Predict
y_1 = clf_1.predict(X)
y_2 = clf_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 example: 0.20 seconds (0 minutes 0.20 seconds)

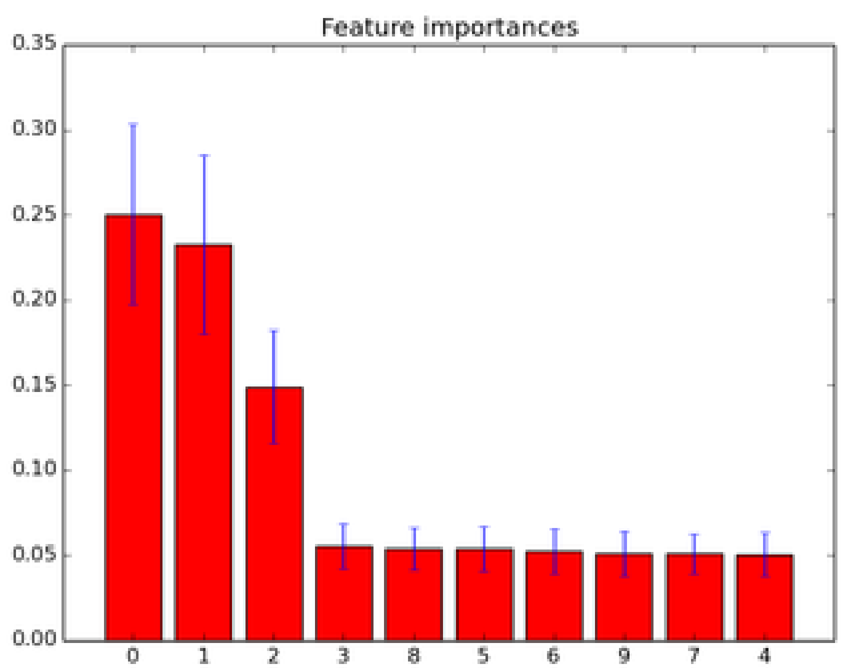
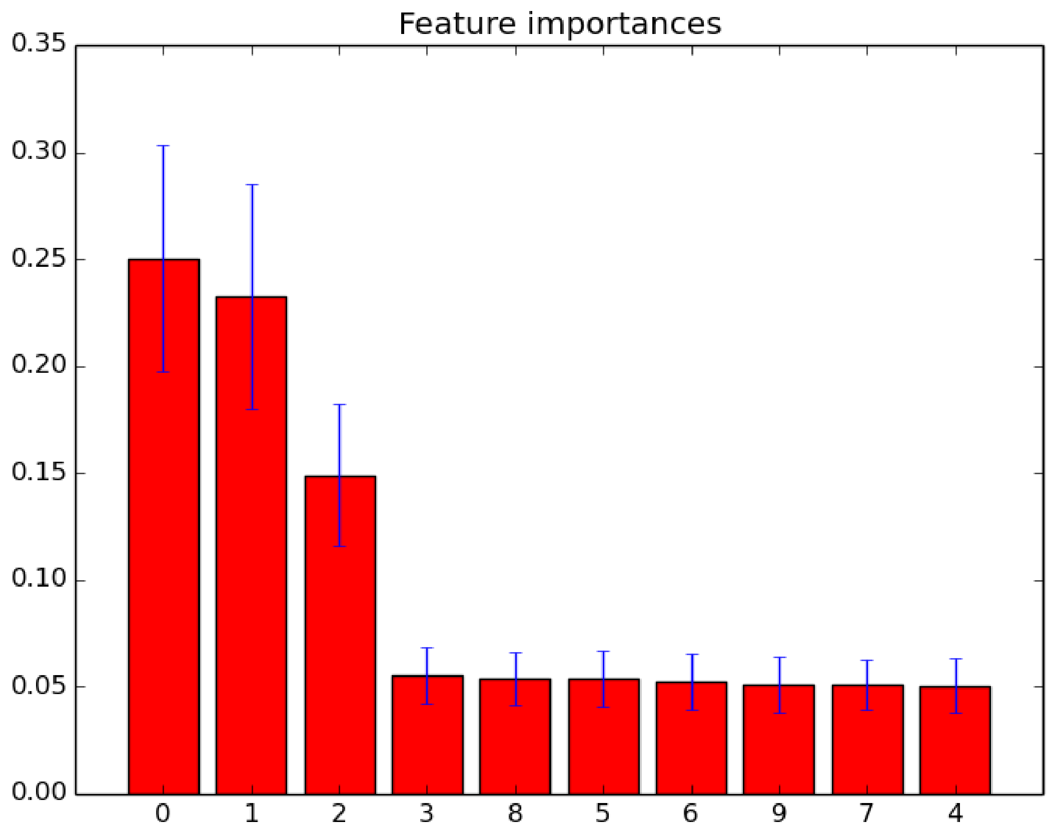


Figure 11.82: *Feature importances with forests of trees*

11.11.3 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.



Script output:

```
Feature ranking:
1. feature 0 (0.250398)
2. feature 1 (0.232397)
3. feature 2 (0.148898)
4. feature 3 (0.055363)
5. feature 8 (0.054010)
6. feature 5 (0.053878)
7. feature 6 (0.052583)
8. feature 9 (0.051020)
9. feature 7 (0.050963)
10. feature 4 (0.050489)
```

Python source code: `plot_forest_importances.py`

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
```

```
from sklearn.datasets import make_classification
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(10):
    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(10), importances[indices],
       color="r", yerr=std[indices], align="center")
plt.xticks(range(10), indices)
plt.xlim([-1, 10])
plt.show()
```

Total running time of the example: 0.33 seconds (0 minutes 0.33 seconds)

11.11.4 Gradient Boosting regularization

Illustration of the effect of different regularization strategies for Gradient Boosting. The example is taken from Hastie et al 2009.

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).

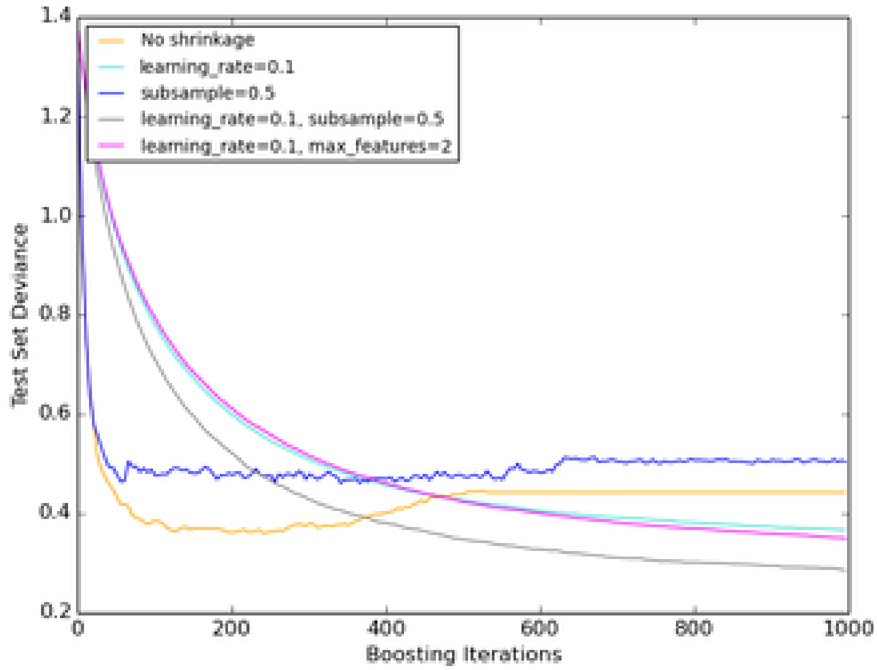
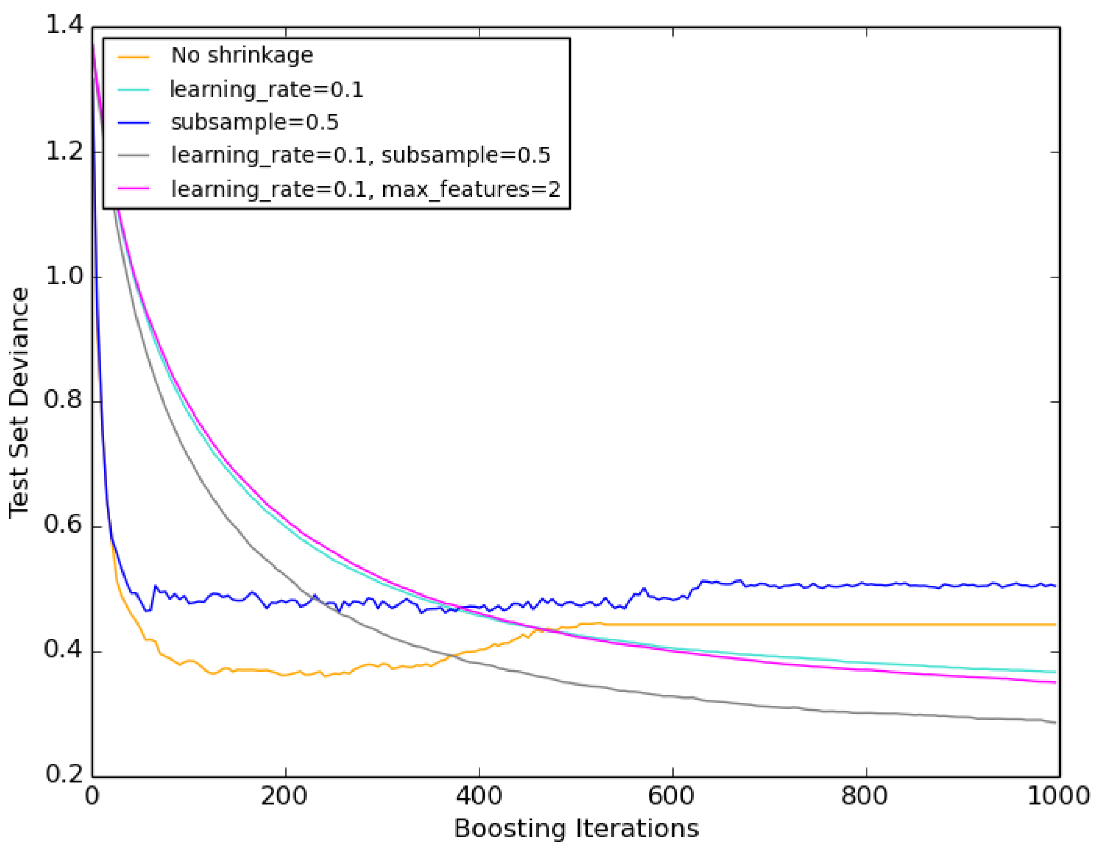


Figure 11.83: *Gradient Boosting regularization*



Python source code: `plot_gradient_boosting_regularization.py`

```
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


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 example: 15.18 seconds (0 minutes 15.18 seconds)

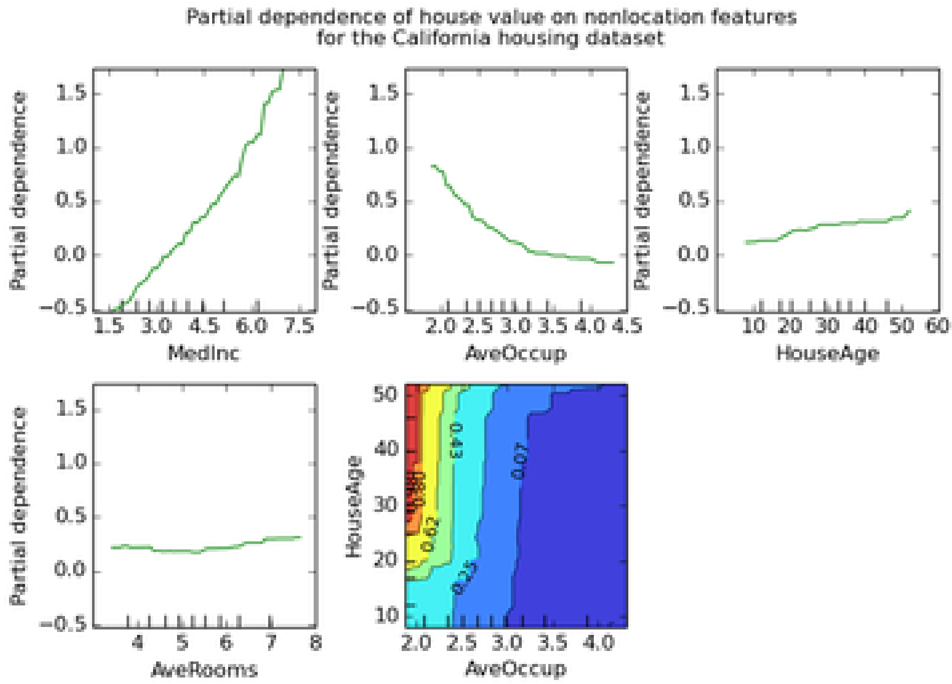


Figure 11.84: *Partial Dependence Plots*

11.11.5 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 (see `feature_importances_`).

This example shows how to obtain partial dependence plots from a `GradientBoostingRegressor` trained on the California housing dataset. The example is taken from [HTF2009].

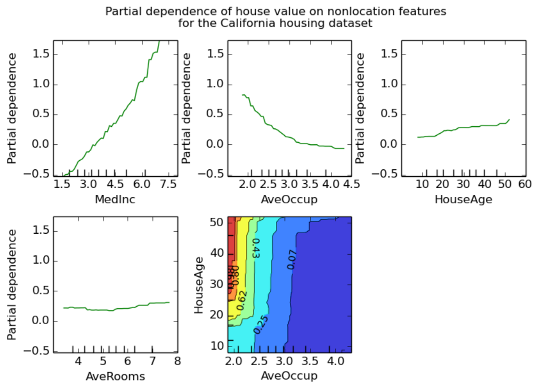
The plot shows four one-way and one two-way partial dependence plots. The target variables for the one-way PDP are: median income (*MedInc*), avg. occupants per household (*AvgOccup*), median house age (*HouseAge*), and avg. rooms per household (*AveRooms*).

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 avg. 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.

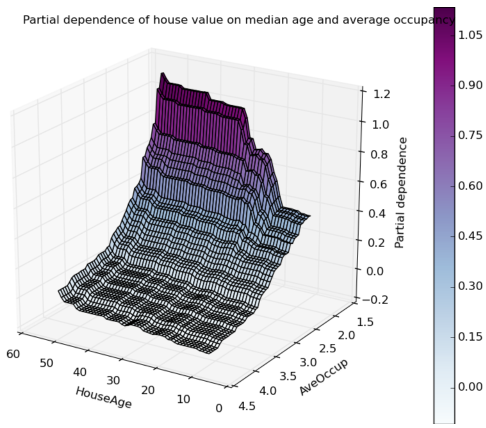
Partial dependence plots with two target features enable us to visualize interactions among them. The two-way partial dependence plot shows the dependence of median house price on joint values of house age and avg. occupants per household. We can clearly see an interaction between the two features: For an avg. occupancy greater than two, the

[2] For classification you can think of it as the regression score before the link function.

house price is nearly independent of the house age, whereas for values less than two there is a strong dependence on age.



•



•

Script output:

```
Training GBRT...
done.
```

```
Convenience plot with ``partial_dependence_plots``
```

```
Custom 3d plot via ``partial_dependence``
```

Python source code: `plot_partial_dependence.py`

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import Axes3D

from sklearn.cross_validation import train_test_split
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble.partial_dependence import plot_partial_dependence
from sklearn.ensemble.partial_dependence import partial_dependence
from sklearn.datasets.california_housing import fetch_california_housing

# fetch California housing dataset
cal_housing = fetch_california_housing()
```

```python
# split 80/20 train-test
X_train, X_test, y_train, y_test = train_test_split(cal_housing.data,
                                                    cal_housing.target,
                                                    test_size=0.2,
                                                    random_state=1)
names = cal_housing.feature_names

print('_' * 80)
print("Training GBRT...")
clf = GradientBoostingRegressor(n_estimators=100, max_depth=4,
                                learning_rate=0.1, loss='huber',
                                random_state=1)
clf.fit(X_train, y_train)
print("done.")

print('_' * 80)
print('Convenience plot with ``partial_dependence_plots``')
print

features = [0, 5, 1, 2, (5, 1)]
fig, axs = plot_partial_dependence(clf, X_train, features, feature_names=names,
                                   n_jobs=3, grid_resolution=50)
fig.suptitle('Partial dependence of house value on nonlocation features\n'
             'for the California housing dataset')
plt.subplots_adjust(top=0.9)  # tight_layout causes overlap with suptitle

print('_' * 80)
print('Custom 3d plot via ``partial_dependence``')
print
fig = plt.figure()

target_feature = (1, 5)
pdp, (x_axis, y_axis) = partial_dependence(clf, target_feature,
                                           X=X_train, grid_resolution=50)
XX, YY = np.meshgrid(x_axis, y_axis)
Z = pdp.T.reshape(XX.shape).T
ax = Axes3D(fig)
surf = ax.plot_surface(XX, YY, Z, rstride=1, cstride=1, cmap=plt.cm.BuPu)
ax.set_xlabel(names[target_feature[0]])
ax.set_ylabel(names[target_feature[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 age and '
            'average occupancy')
plt.subplots_adjust(top=0.9)

plt.show()
```

Total running time of the example: 3.26 seconds (0 minutes 3.26 seconds)

11.11.6 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.

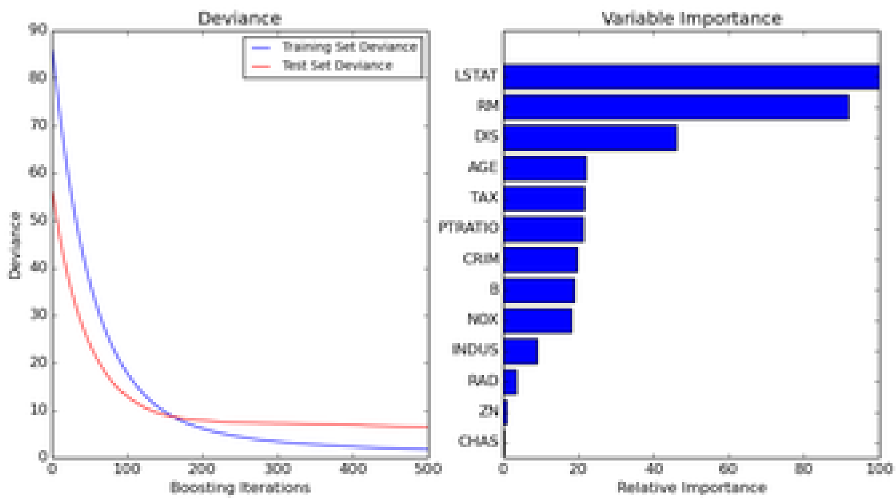
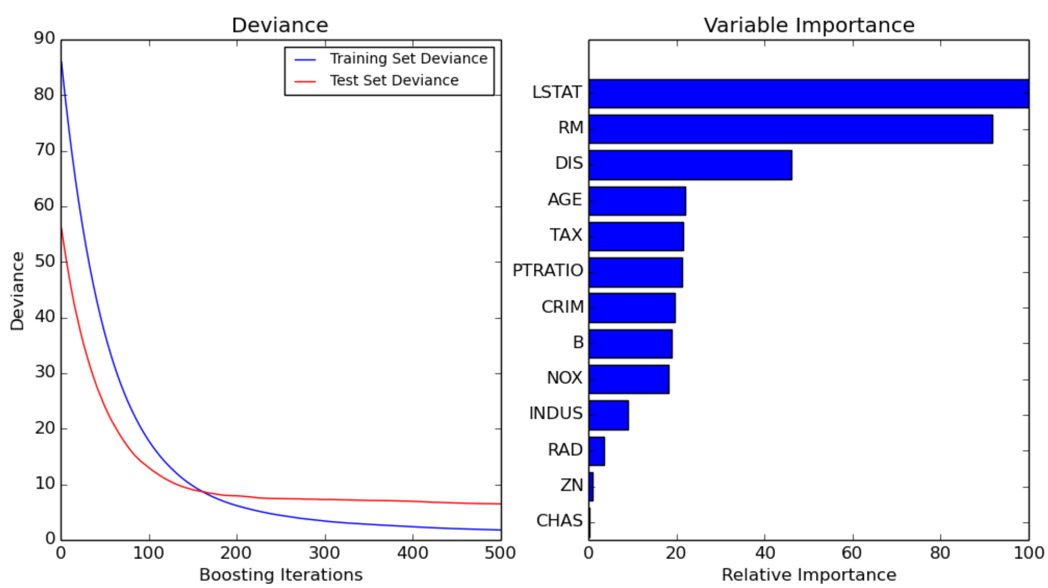


Figure 11.85: *Gradient Boosting regression*



Script output:

```
MSE: 6.4852
```

Python source code: `plot_gradient_boosting_regression.py`

```
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]
X_test, y_test = X[offset:], y[offset:]

###############################################################################
# Fit regression model
params = {'n_estimators': 500, 'max_depth': 4, 'min_samples_split': 1,
          '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_decision_function(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 example: 0.67 seconds (0 minutes 0.67 seconds)

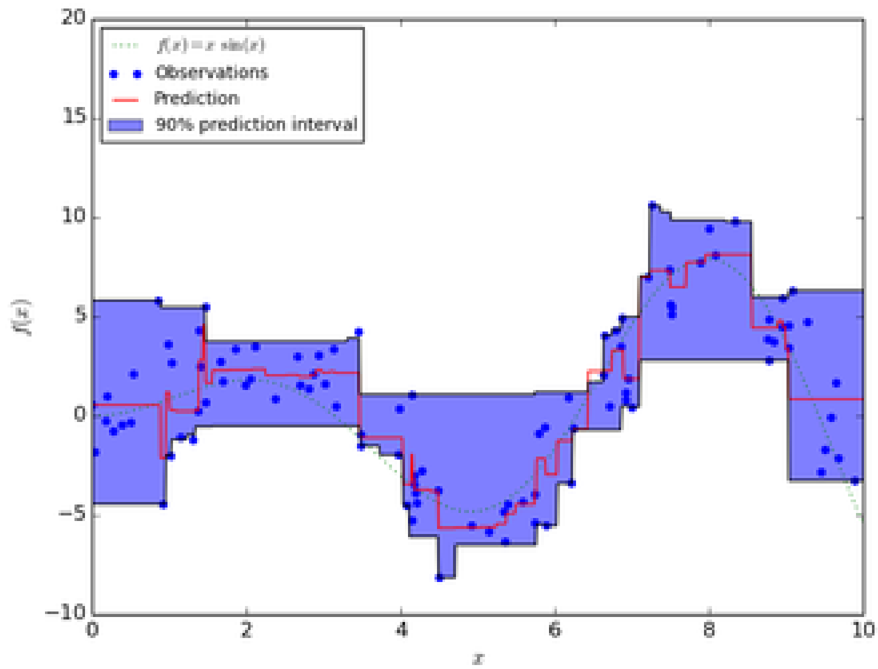
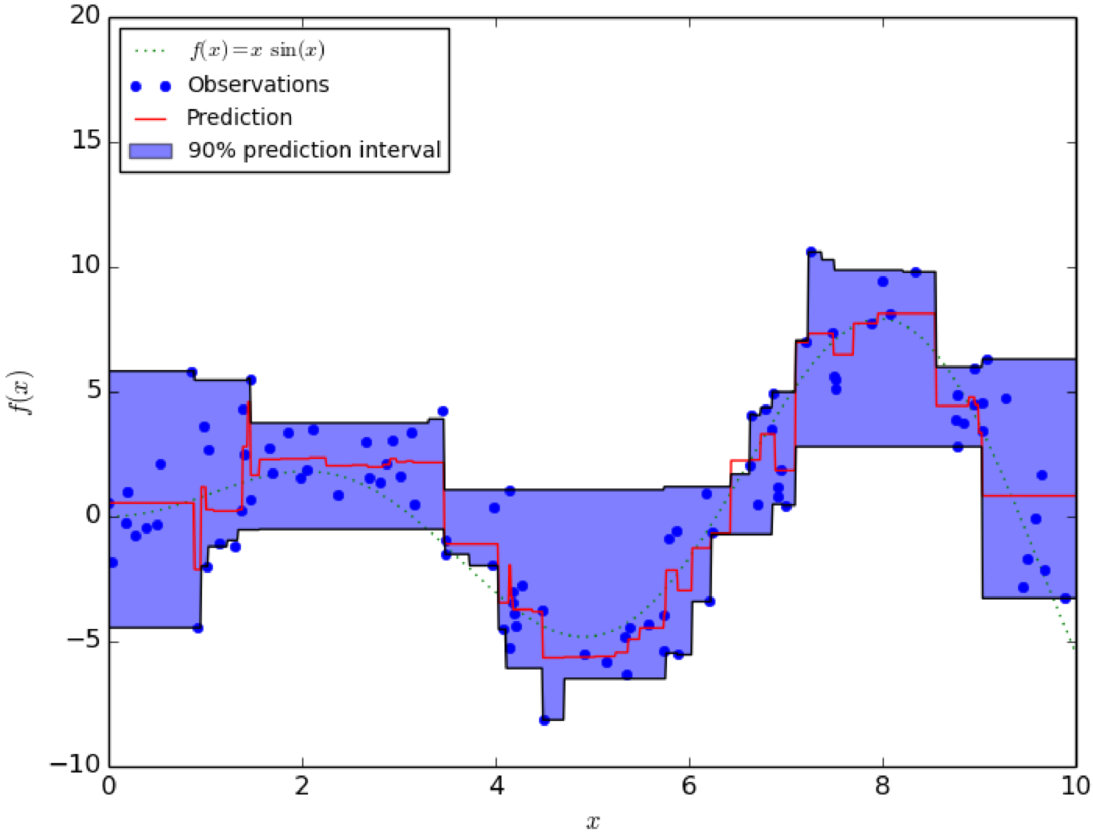


Figure 11.86: *Prediction Intervals for Gradient Boosting Regression*

11.11.7 Prediction Intervals for Gradient Boosting Regression

This example shows how quantile regression can be used to create prediction intervals.



Python source code: `plot_gradient_boosting_quantile.py`

```
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)
```

```
# 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=u'$f(x) = x\,\sin(x)$')
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 example: 0.21 seconds (0 minutes 0.21 seconds)

11.11.8 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 repre-

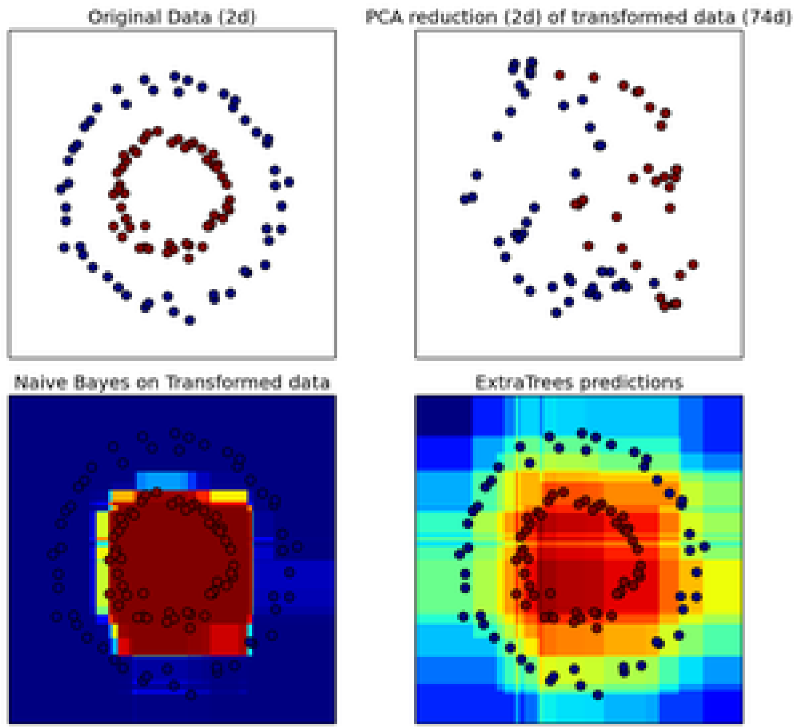
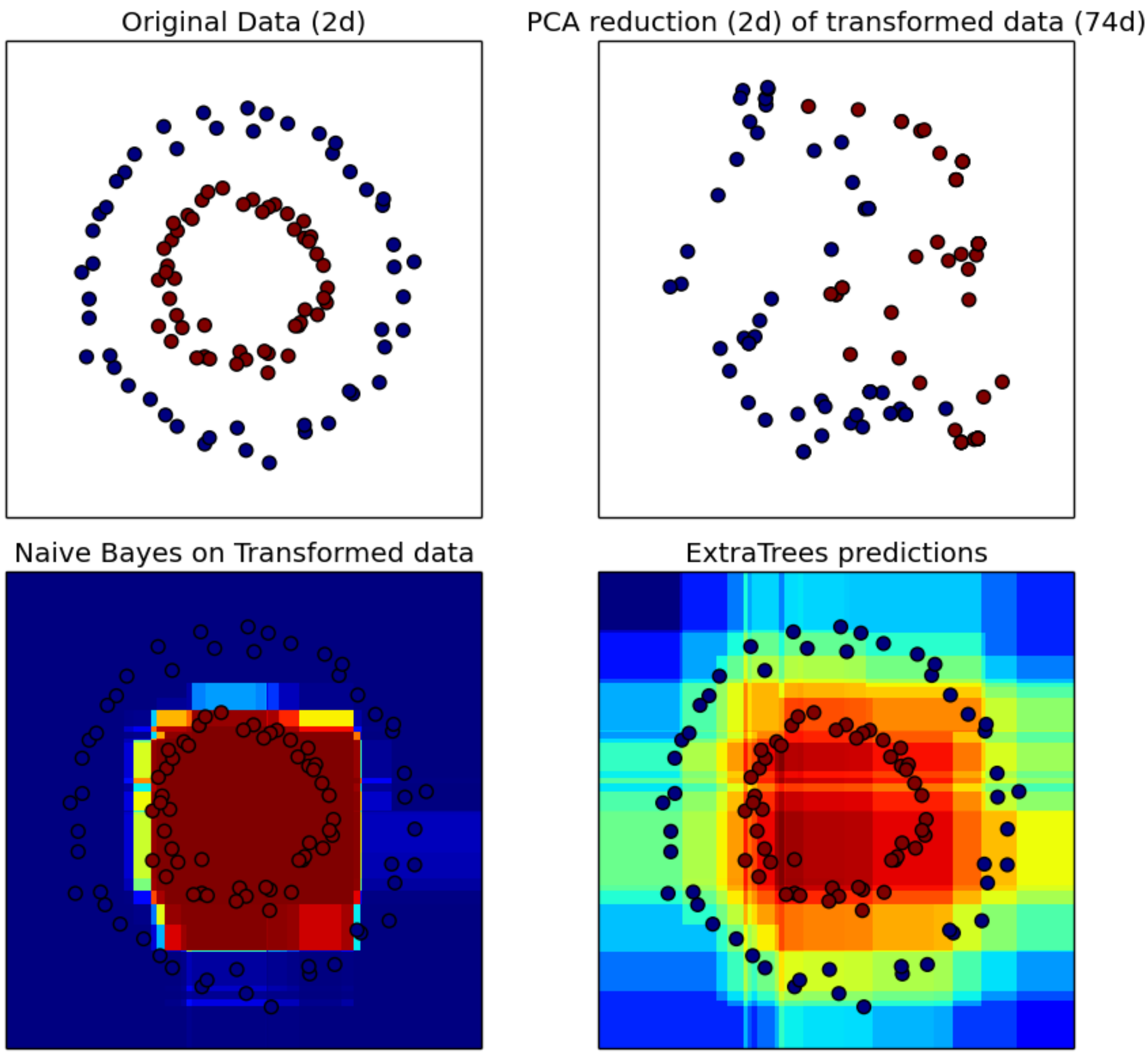


Figure 11.87: *Hashing feature transformation using Totally Random Trees*

sentation. This allows to separate two concentric circles simply based on the principal components of the transformed data.

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.



Python source code: `plot_random_forest_embedding.py`

```
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 using PCA
pca = TruncatedSVD(n_components=2)
X_reduced = pca.fit_transform(X_transformed)
```

```
# 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)
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)
ax.set_title("PCA 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, m_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)
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)
ax.set_ylim(-1.4, 1.4)
ax.set_xlim(-1.4, 1.4)
ax.set_xticks(())
ax.set_yticks(())
```

```
plt.tight_layout()
plt.show()
```

Total running time of the example: 0.30 seconds (0 minutes 0.30 seconds)

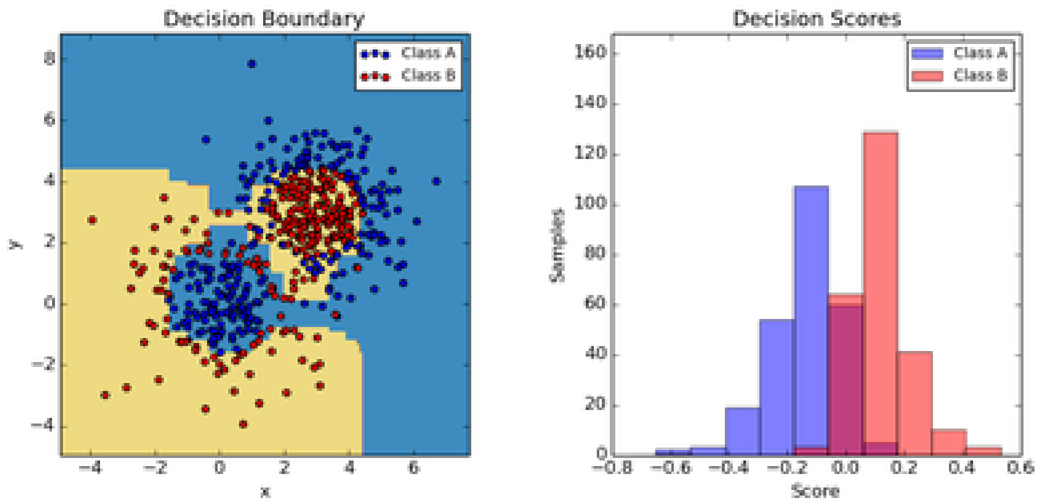
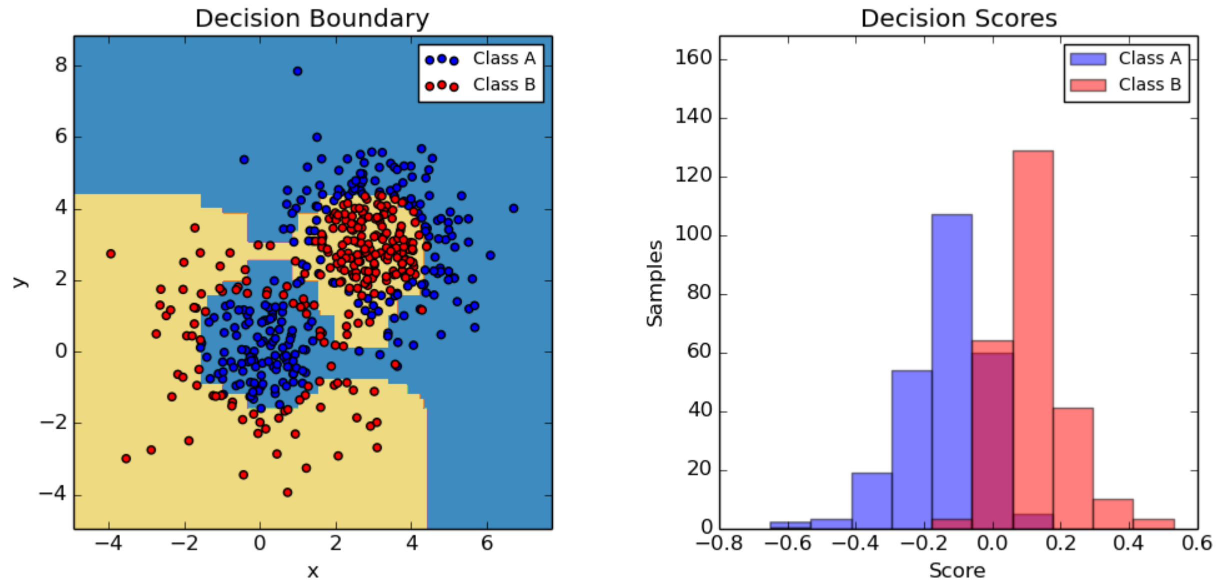


Figure 11.88: *Two-class AdaBoost*

11.11.9 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.



Python source code: `plot_adaboost_twoclass.py`

```
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,
                                 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,
                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)
x1, x2, y1, y2 = plt.axis()
plt.axis((x1, x2, y1, y2 * 1.2))
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 example: 2.16 seconds (0 minutes 2.16 seconds)

11.11.10 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.

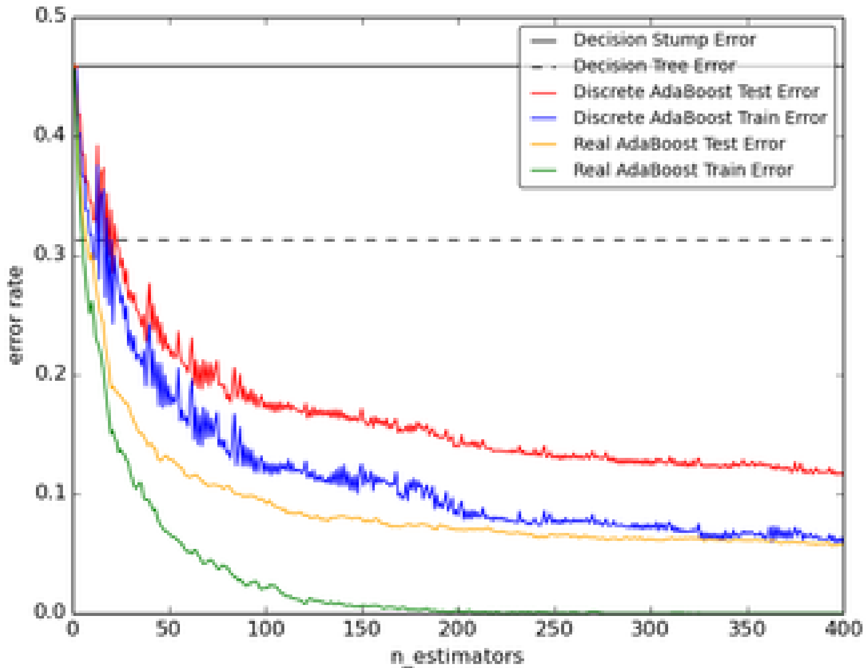
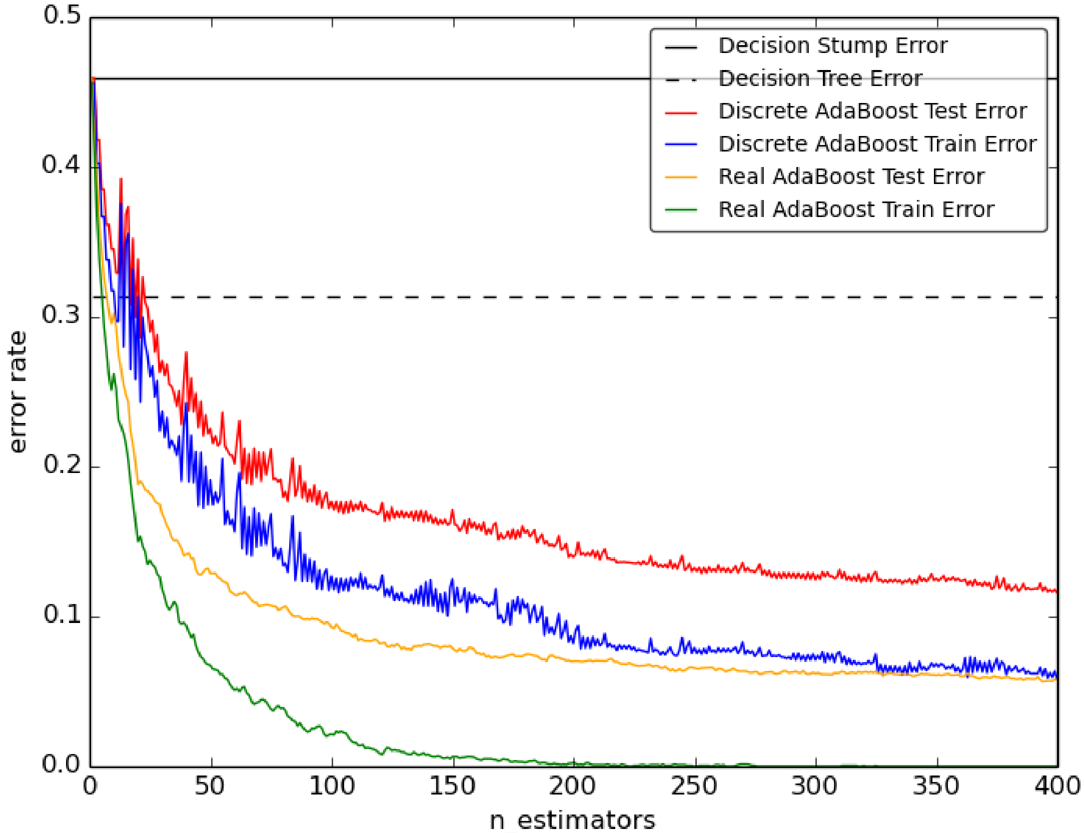


Figure 11.89: *Discrete versus Real AdaBoost*

Discrete SAMME AdaBoost adapts based on errors in predicted class labels whereas real SAMME.R uses the predicted class probabilities.



Python source code: plot_adaboost_hastie_10_2.py

```
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]
```

```
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',
        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 example: 3.83 seconds (0 minutes 3.83 seconds)

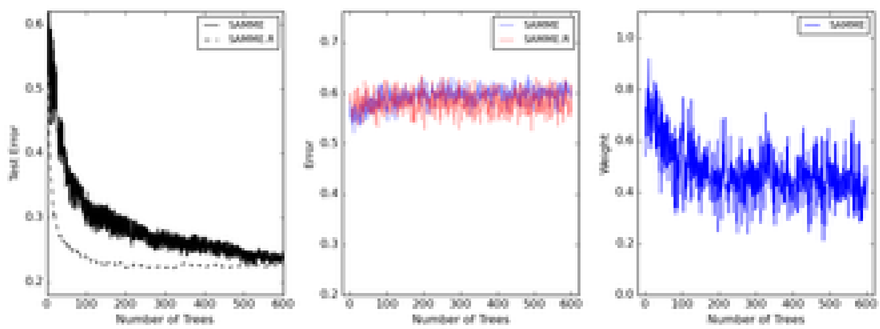
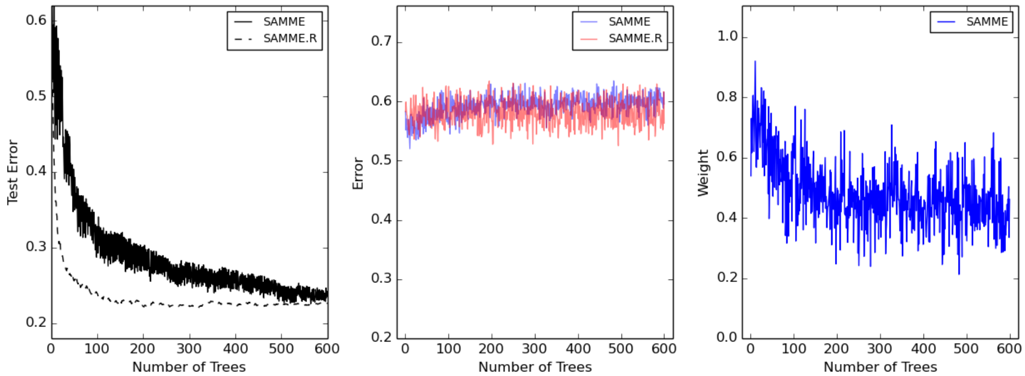


Figure 11.90: *Multi-class AdaBoosted Decision Trees*

11.11.11 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.



Python source code: `plot_adaboost_multiclass.py`

```
print(__doc__)

# Author: Noel Dawe <noel.dawe@gmail.com>
#
# License: BSD 3 clause

from sklearn.externals.six.moves import zip

import matplotlib.pyplot as plt

from sklearn.datasets import make_gaussian_quantiles
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score
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')

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 example: 11.08 seconds (0 minutes 11.08 seconds)

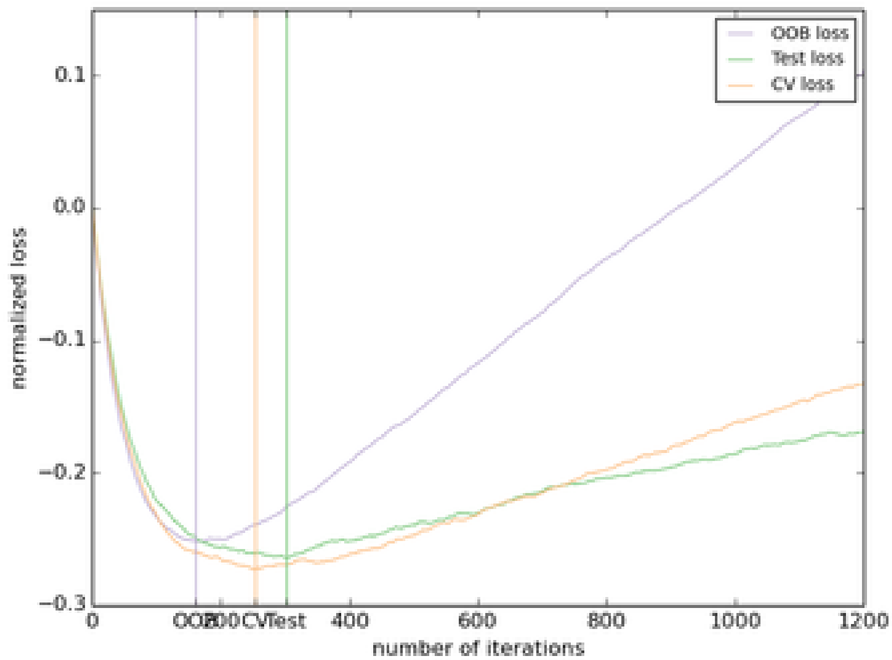
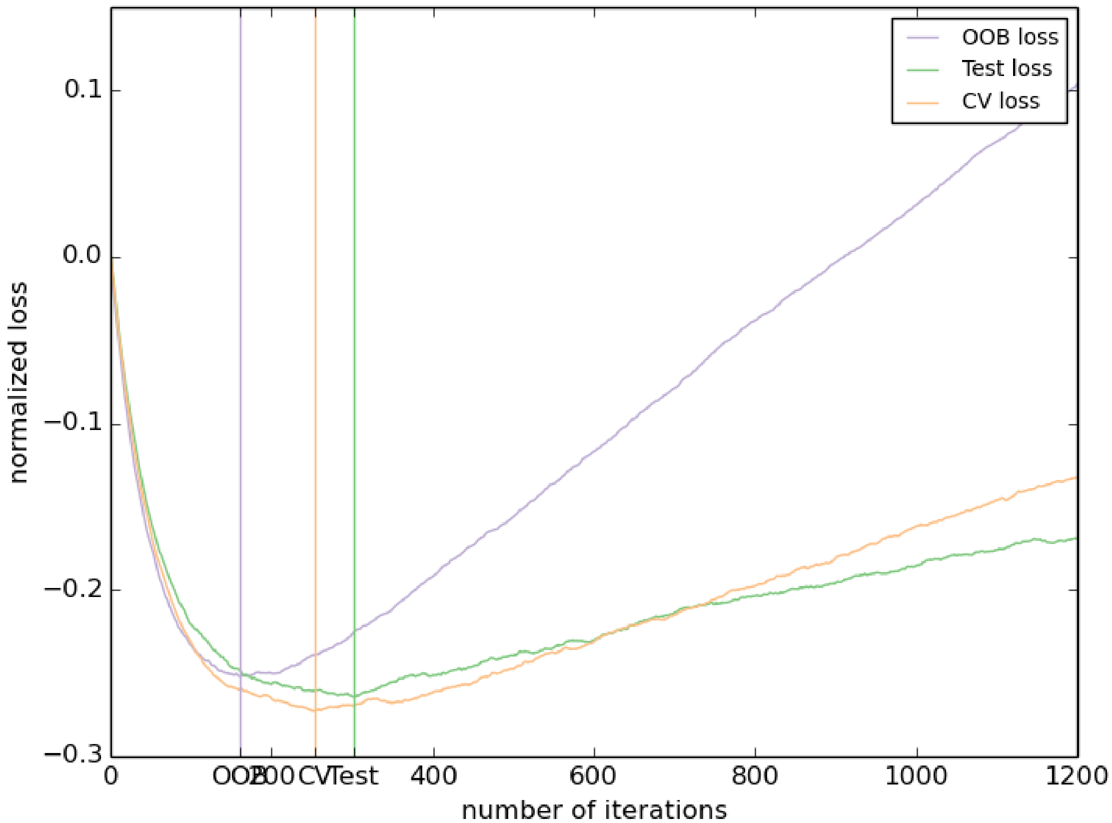


Figure 11.91: *Gradient Boosting Out-of-Bag estimates*

11.11.12 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.



Script output:

```
Accuracy: 0.6840
```

Python source code: `plot_gradient_boosting_oob.py`

```
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.cross_validation import KFold
from sklearn.cross_validation import train_test_split


# Generate data (adapted from G. Ridgeway's gbm example)
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 = 1 / (1.0 + np.exp(-(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_folds=3):
    cv = KFold(n=X_train.shape[0], n_folds=n_folds)
    cv_clf = ensemble.GradientBoostingClassifier(**params)
    val_scores = np.zeros((n_estimators,), dtype=np.float64)
    for train, test in cv:
        cv_clf.fit(X_train[train], y_train[train])
        val_scores += heldout_score(cv_clf, X_train[test], y_train[test])
    val_scores /= n_folds
    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
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 example: 2.57 seconds (0 minutes 2.57 seconds)

11.11.13 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:

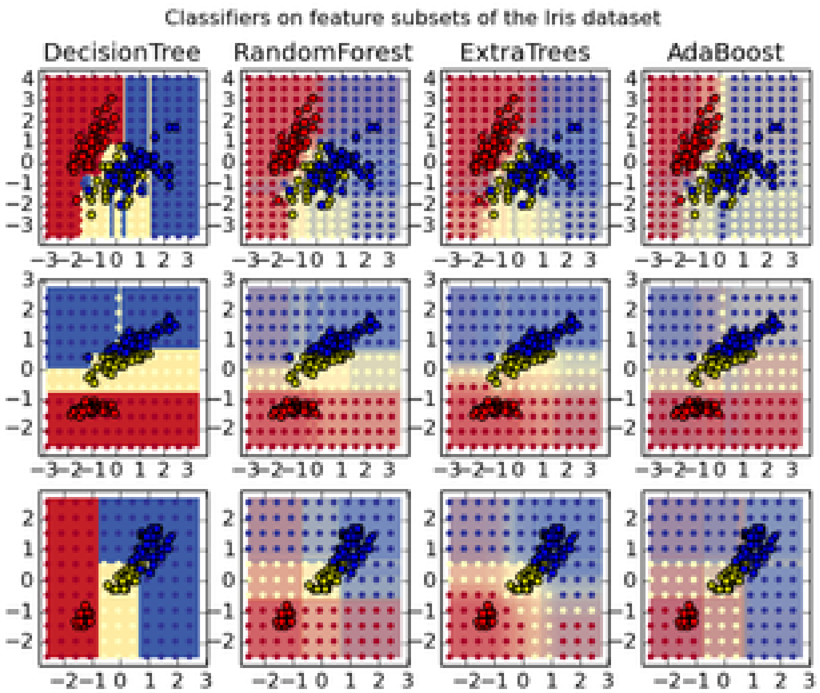
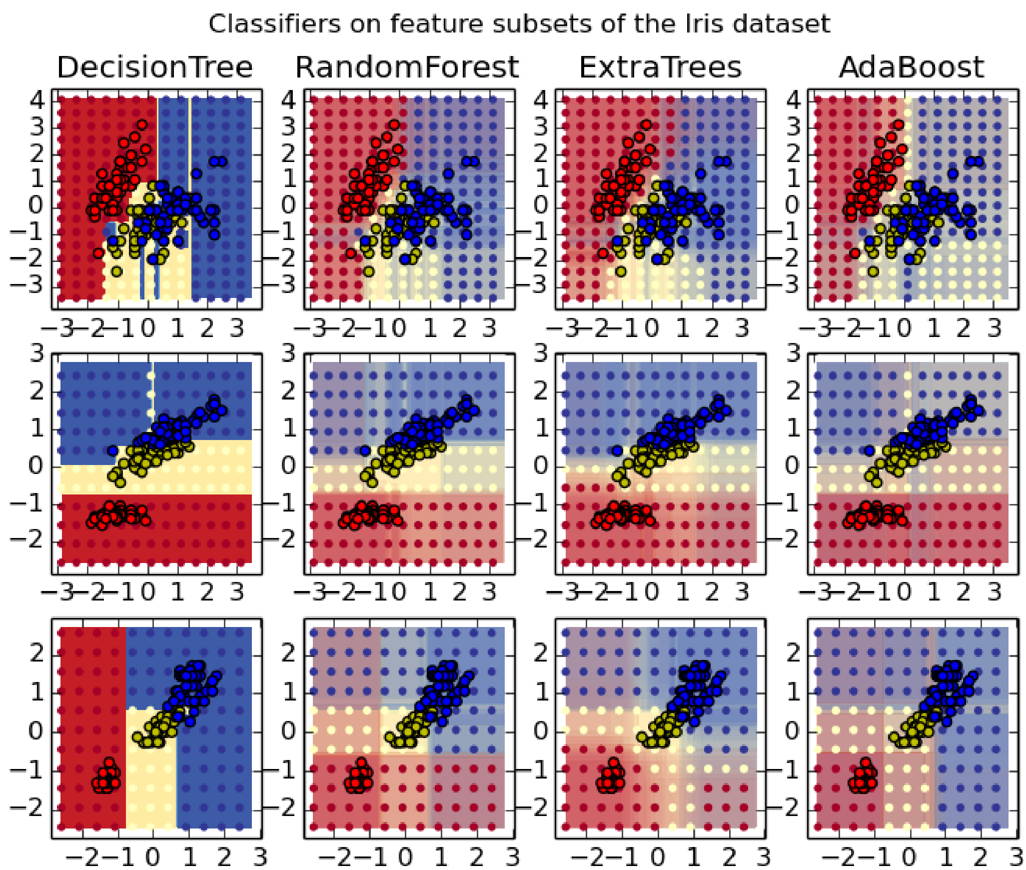


Figure 11.92: *Plot the decision surfaces of ensembles of trees on the iris dataset*

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.



Script output:

```
DecisionTree with features [0, 1] has a score of 0.926666666667
RandomForest with 30 estimators with features [0, 1] has a score of 0.926666666667
ExtraTrees with 30 estimators with features [0, 1] has a score of 0.926666666667
AdaBoost with 30 estimators with features [0, 1] has a score of 0.86
DecisionTree with features [0, 2] has a score of 0.993333333333
RandomForest with 30 estimators with features [0, 2] has a score of 0.993333333333
ExtraTrees with 30 estimators with features [0, 2] has a score of 0.993333333333
AdaBoost with 30 estimators with features [0, 2] has a score of 0.993333333333
DecisionTree with features [2, 3] has a score of 0.993333333333
RandomForest with 30 estimators with features [2, 3] has a score of 0.993333333333
ExtraTrees with 30 estimators with features [2, 3] has a score of 0.993333333333
AdaBoost with 30 estimators with features [2, 3] has a score of 0.993333333333
```

Python source code: `plot_forest_iris.py`

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn import clone
from sklearn.datasets import load_iris
from sklearn.ensemble import (RandomForestClassifier, ExtraTreesClassifier,
                              AdaBoostClassifier)
```

```python
from sklearn.externals.six.moves import xrange
from sklearn.tree import DecisionTreeClassifier

# Parameters
n_classes = 3
n_estimators = 30
plot_colors = "ryb"
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
        clf = clone(model)
        clf = model.fit(X, y)

        scores = clf.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, "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)
```

```
        # 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_co
        cs_points = plt.scatter(xx_coarser, yy_coarser, s=15, c=Z_points_coarser, cmap=cmap, edgecolo

        # Plot the training points, these are clustered together and have a
        # black outline
        for i, c in zip(xrange(n_classes), plot_colors):
            idx = np.where(y == i)
            plt.scatter(X[idx, 0], X[idx, 1], c=c, label=iris.target_names[i],
                        cmap=cmap)

        plot_idx += 1  # move on to the next plot in sequence

plt.suptitle("Classifiers on feature subsets of the Iris dataset")
plt.axis("tight")

plt.show()
```

Total running time of the example: 4.48 seconds (0 minutes 4.48 seconds)

11.11.14 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.

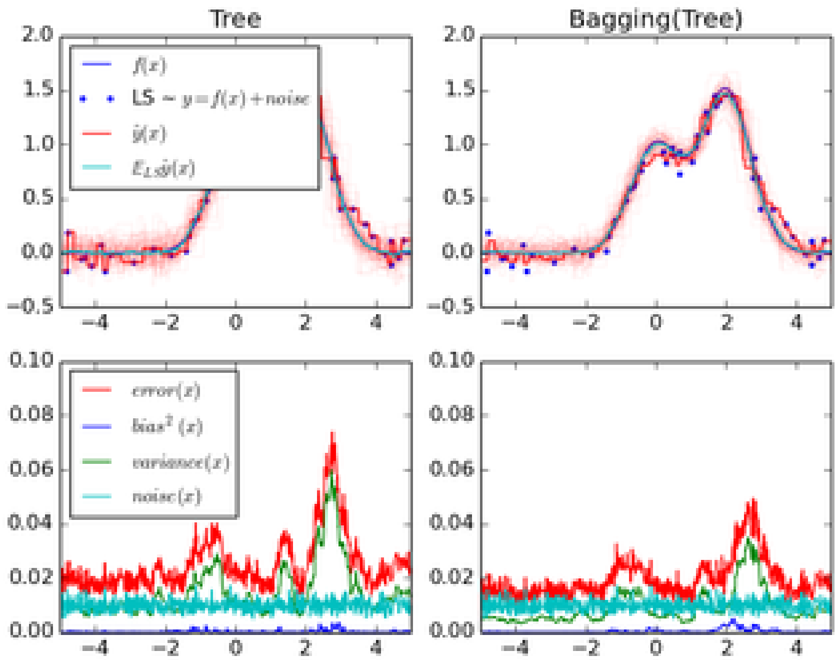


Figure 11.93: *Single estimator versus bagging: bias-variance decomposition*

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).

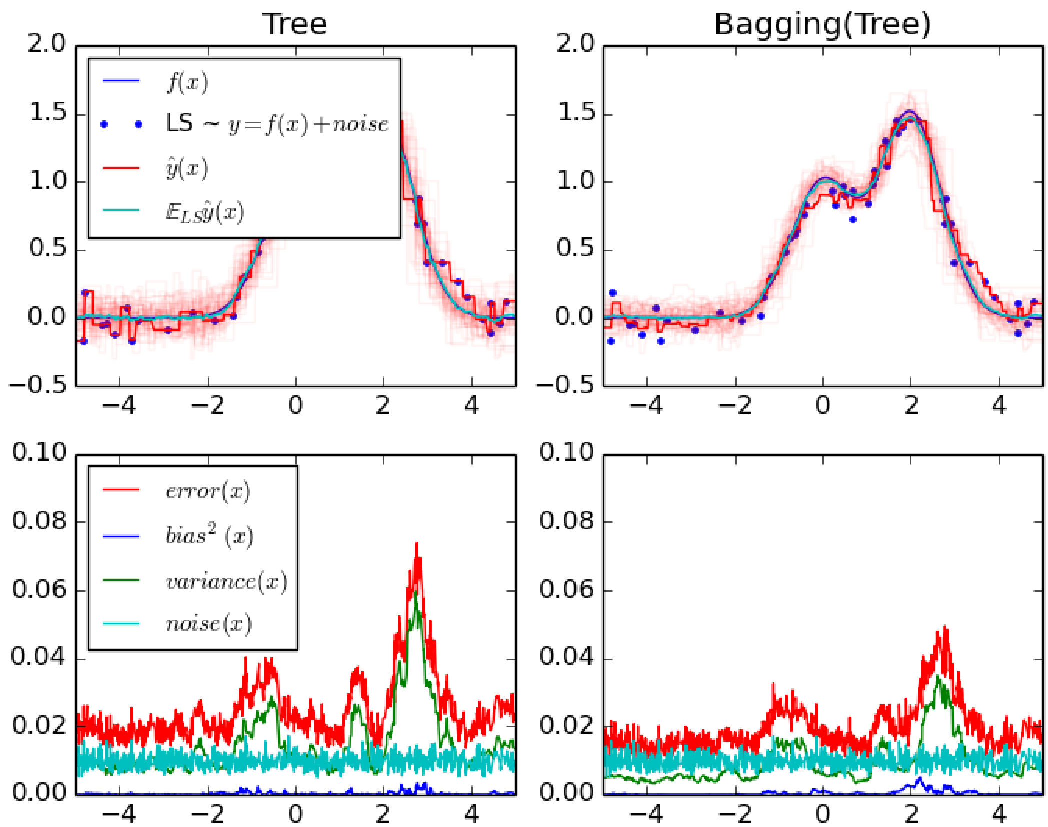
The lower left figure plots the pointwise decomposition of the expected mean squared error of a single decision tree. It confirms that the bias term (in blue) is low while the variance is large (in green). It also illustrates the noise part of the error which, as expected, appears to be constant and 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 [3].

[3] T. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning", Springer, 2009.

References



Script output:

```
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)
```

Python source code: `plot_bias_variance.py`

```
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

# 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)

# Loop over estimators to compare
for n, (name, estimator) in enumerate(estimators):
    # Compute predictions
    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="$\^y(x)$")
        else:
            plt.plot(X_test, y_predict[:, i], "r", alpha=0.05)

    plt.plot(X_test, np.mean(y_predict, axis=1), "c",
             label="$\mathbb{E}_{LS} \^y(x)$")

    plt.xlim([-5, 5])
    plt.title(name)

    if n == 0:
        plt.legend(loc="upper left", prop={"size": 11})

    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 == 0:
        plt.legend(loc="upper left", prop={"size": 11})

plt.show()
```

Total running time of the example: 0.60 seconds (0 minutes 0.60 seconds)

11.12 Tutorial exercises

Exercises for the tutorials

11.12.1 Digits Classification Exercise

A tutorial exercise regarding the use of classification techniques on the Digits dataset.

Figure 11.94: *Digits Classification Exercise*

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*.

Python source code: digits_classification_exercise.py

```
print(__doc__)

from sklearn import datasets, neighbors, linear_model

digits = datasets.load_digits()
X_digits = digits.data
y_digits = digits.target

n_samples = len(X_digits)

X_train = X_digits[:.9 * n_samples]
y_train = y_digits[:.9 * n_samples]
X_test = X_digits[.9 * n_samples:]
y_test = y_digits[.9 * n_samples:]

knn = neighbors.KNeighborsClassifier()
logistic = linear_model.LogisticRegression()

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))
```

11.12.2 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*.

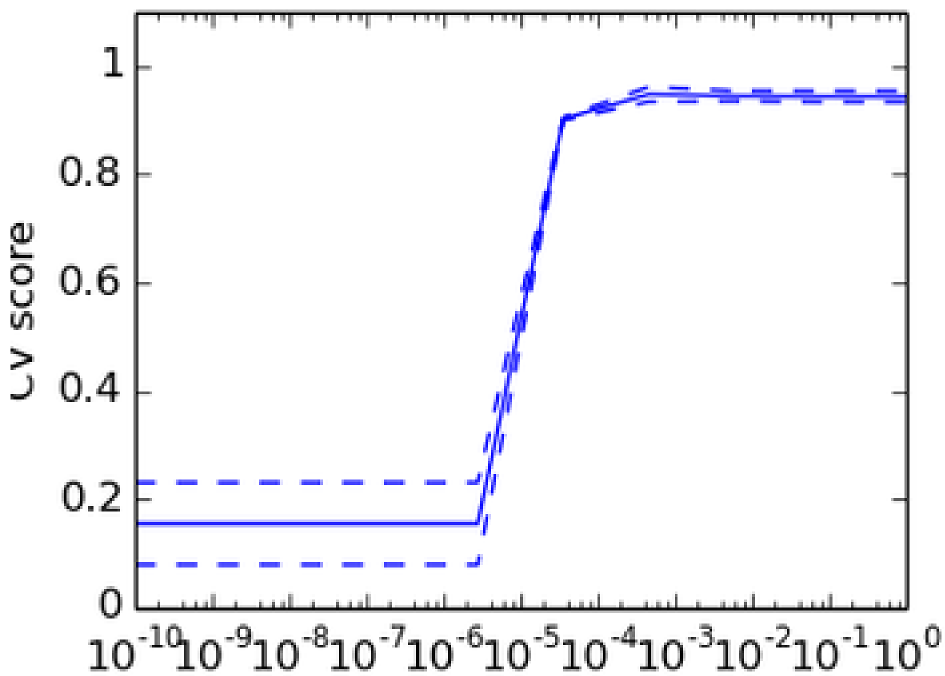
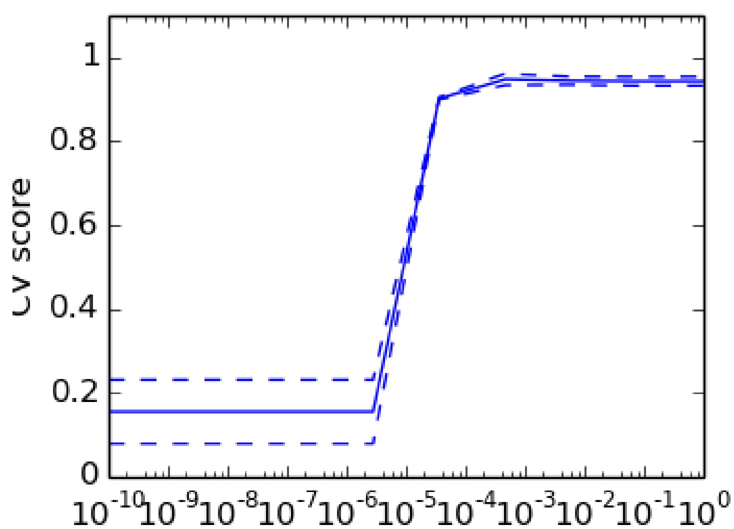


Figure 11.95: *Cross-validation on Digits Dataset Exercise*



Python source code: `plot_cv_digits.py`

```
print(__doc__)


import numpy as np
from sklearn import cross_validation, datasets, svm

digits = datasets.load_digits()
```

```
X = digits.data
y = digits.target

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_validation.cross_val_score(svc, X, y, n_jobs=1)
    scores.append(np.mean(this_scores))
    scores_std.append(np.std(this_scores))

# Do the plotting
import matplotlib.pyplot as plt
plt.figure(1, figsize=(4, 3))
plt.clf()
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 example: 4.47 seconds (0 minutes 4.47 seconds)

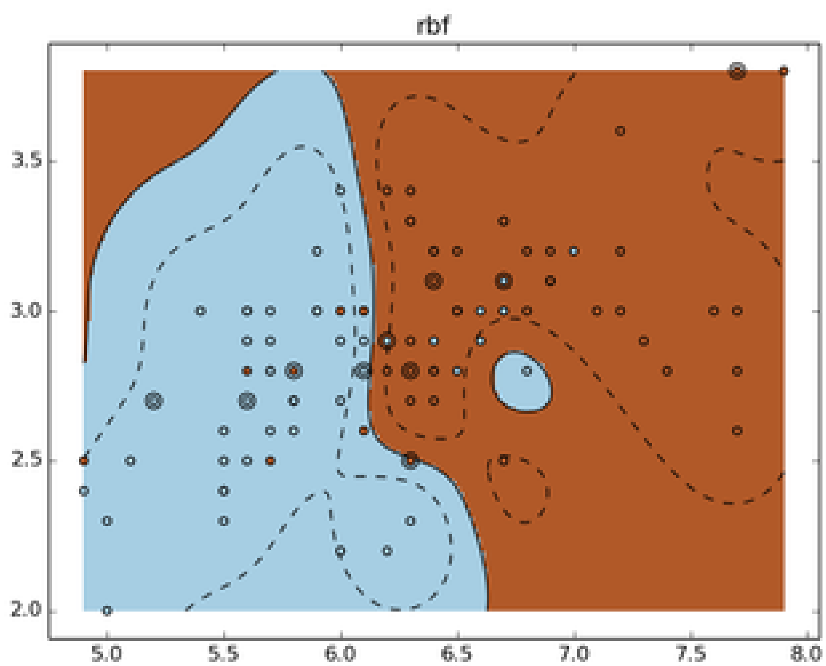
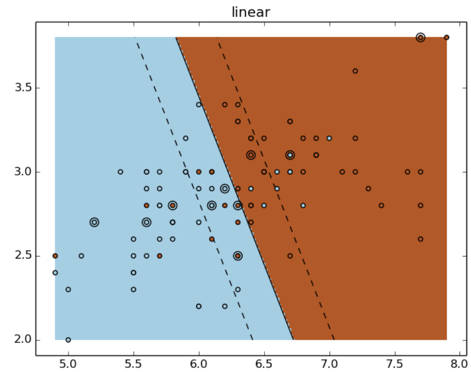


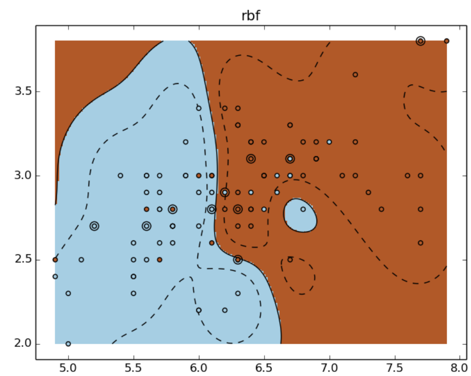
Figure 11.96: *SVM Exercise*

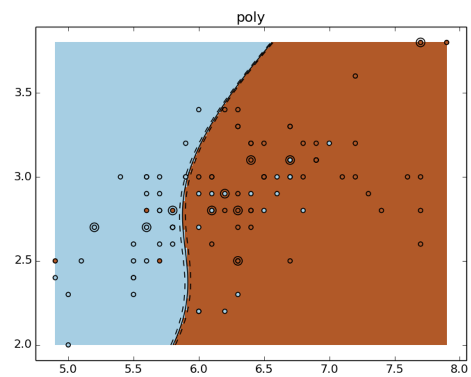
11.12.3 SVM Exercise

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*.

- 

- 

- 

Python source code: `plot_iris_exercise.py`

```
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[:.9 * n_sample]
y_train = y[:.9 * n_sample]
X_test = X[.9 * n_sample:]
y_test = y[.9 * n_sample:]

# fit the model
for fig_num, kernel in enumerate(('linear', 'rbf', 'poly')):
    clf = svm.SVC(kernel=kernel, gamma=10)
    clf.fit(X_train, y_train)

    plt.figure(fig_num)
    plt.clf()
    plt.scatter(X[:, 0], X[:, 1], c=y, zorder=10, cmap=plt.cm.Paired)

    # Circle out the test data
    plt.scatter(X_test[:, 0], X_test[:, 1], s=80, facecolors='none', zorder=10)

    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 example: 6.43 seconds (0 minutes 6.43 seconds)

11.12.4 Cross-validation on diabetes Dataset Exercise

A tutorial exercise which uses cross-validation with linear models.

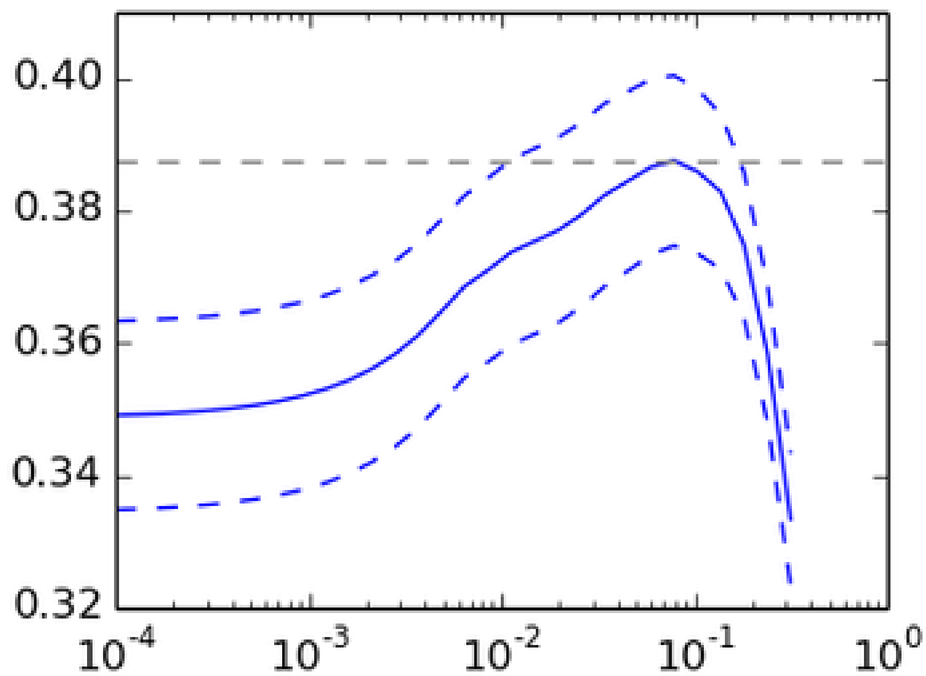
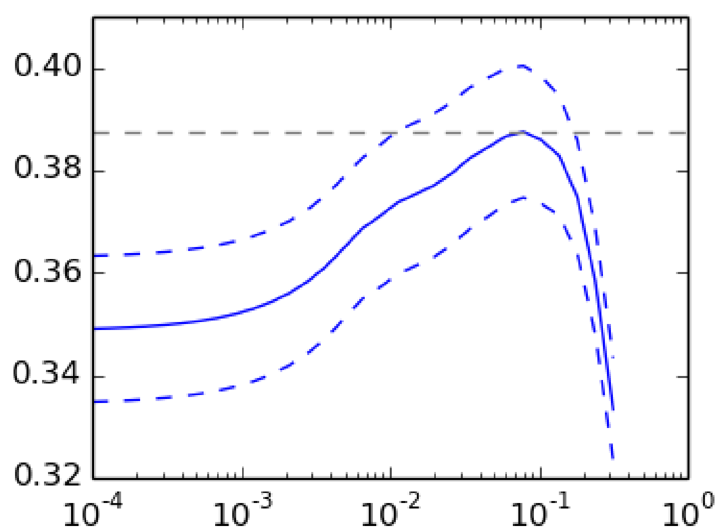


Figure 11.97: *Cross-validation on diabetes Dataset Exercise*

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*.



Script output:

```
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: 0.10405, score: 0.53573
[fold 1] alpha: 0.05968, score: 0.16278
[fold 2] alpha: 0.10405, score: 0.44437

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.
```

Python source code: `plot_cv_diabetes.py`

```python
from __future__ import print_function
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn import cross_validation, datasets, linear_model

diabetes = datasets.load_diabetes()
X = diabetes.data[:150]
y = diabetes.target[:150]

lasso = linear_model.Lasso()
alphas = np.logspace(-4, -.5, 30)

scores = list()
scores_std = list()

for alpha in alphas:
    lasso.alpha = alpha
    this_scores = cross_validation.cross_val_score(lasso, X, y, n_jobs=1)
    scores.append(np.mean(this_scores))
    scores_std.append(np.std(this_scores))

plt.figure(figsize=(4, 3))
plt.semilogx(alphas, scores)
# plot error lines showing +/- std. errors of the scores
plt.semilogx(alphas, np.array(scores) + np.array(scores_std) / np.sqrt(len(X)),
             'b--')
plt.semilogx(alphas, np.array(scores) - np.array(scores_std) / np.sqrt(len(X)),
             'b--')
plt.ylabel('CV score')
plt.xlabel('alpha')
plt.axhline(np.max(scores), linestyle='--', color='.5')

##############################################################################
# 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 = linear_model.LassoCV(alphas=alphas)
k_fold = cross_validation.KFold(len(X), 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):
    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()
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 example: 0.26 seconds (0 minutes 0.26 seconds)

11.13 Feature Selection

Examples concerning the `sklearn.feature_selection` module.

Figure 11.98: *Pipeline Anova SVM*

11.13.1 Pipeline Anova SVM

Simple usage of Pipeline that runs successively a univariate feature selection with anova and then a C-SVM of the selected features.

Python source code: `feature_selection_pipeline.py`

```
print(__doc__)

from sklearn import svm
from sklearn.datasets import samples_generator
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.pipeline import make_pipeline

# import some data to play with
X, y = samples_generator.make_classification(
    n_features=20, n_informative=3, n_redundant=0, n_classes=4,
    n_clusters_per_class=2)
```

```
# ANOVA SVM-C
# 1) anova filter, take 3 best ranked features
anova_filter = SelectKBest(f_regression, k=3)
# 2) svm
clf = svm.SVC(kernel='linear')

anova_svm = make_pipeline(anova_filter, clf)
anova_svm.fit(X, y)
anova_svm.predict(X)
```

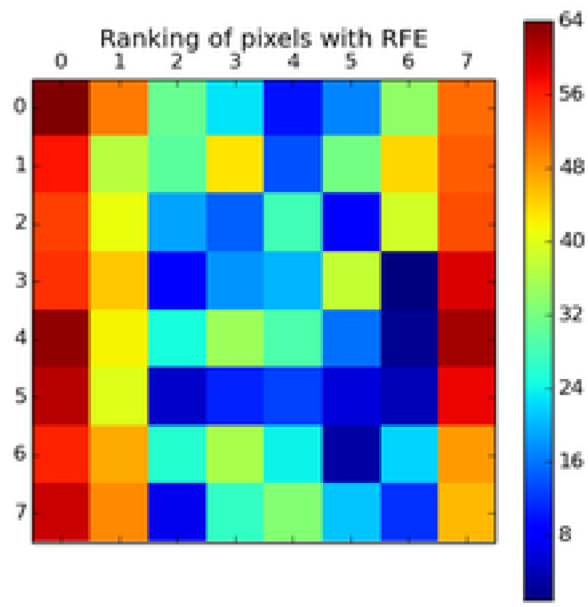
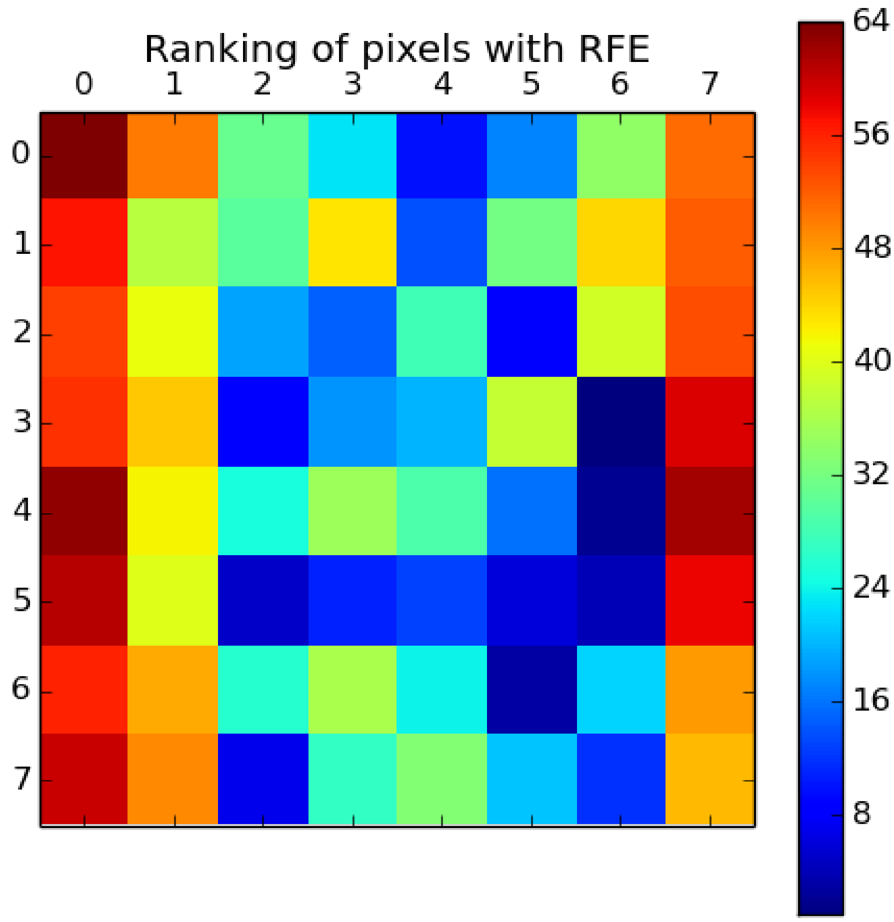


Figure 11.99: *Recursive feature elimination*

11.13.2 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*



Python source code: `plot_rfe_digits.py`

```
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)
plt.colorbar()
plt.title("Ranking of pixels with RFE")
plt.show()
```

Total running time of the example: 3.96 seconds (0 minutes 3.96 seconds)

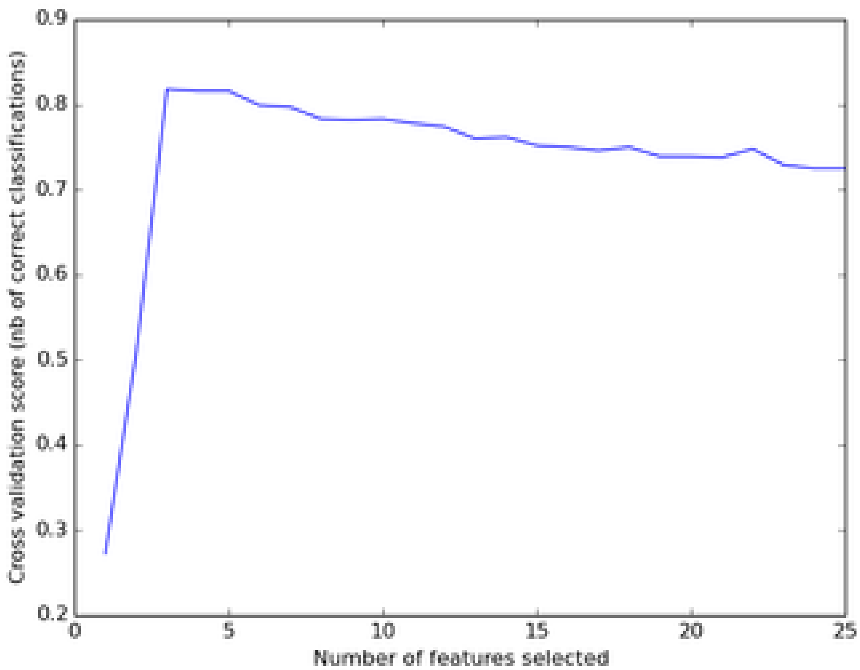
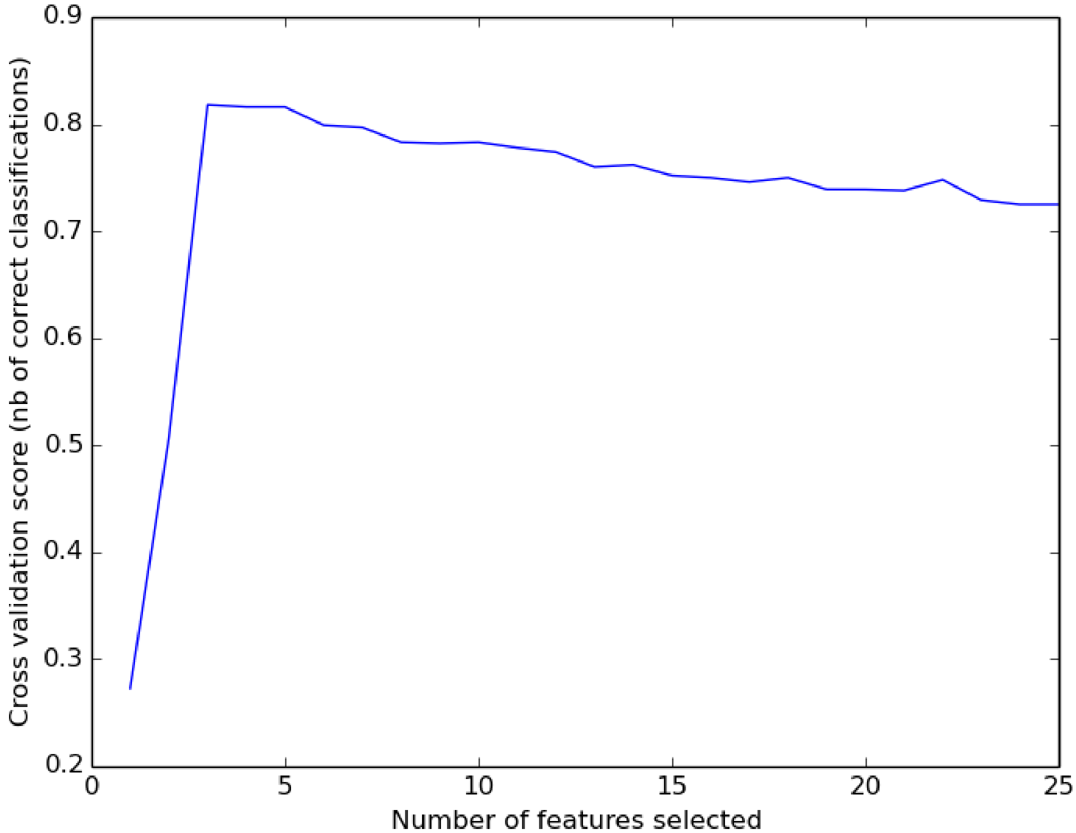


Figure 11.100: *Recursive feature elimination with cross-validation*

11.13.3 Recursive feature elimination with cross-validation

A recursive feature elimination example with automatic tuning of the number of features selected with cross-validation.



Script output:

```
Optimal number of features : 3
```

Python source code: `plot_rfe_with_cross_validation.py`

```
print(__doc__)

import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.cross_validation 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
# classifications
rfecv = RFECV(estimator=svc, step=1, cv=StratifiedKFold(y, 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 example: 2.63 seconds (0 minutes 2.63 seconds)

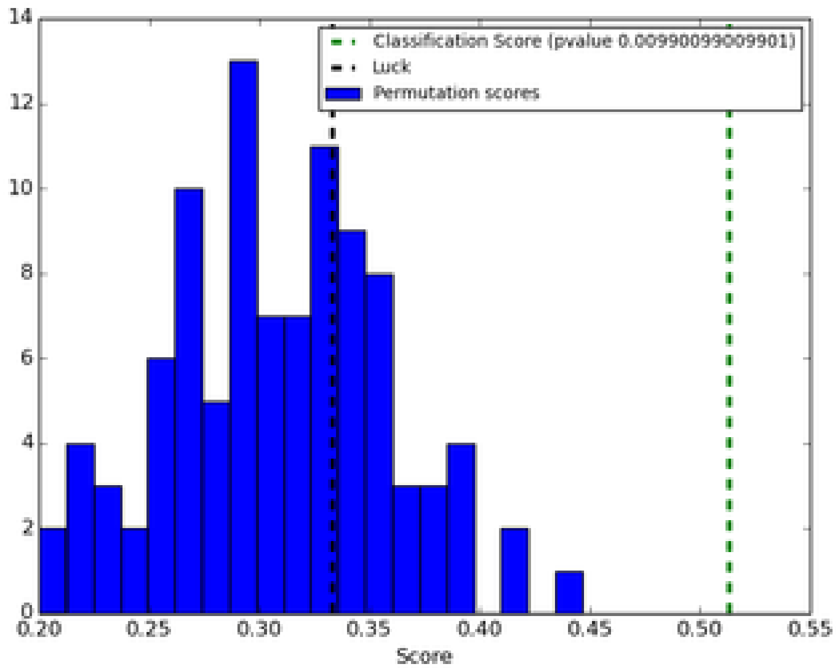
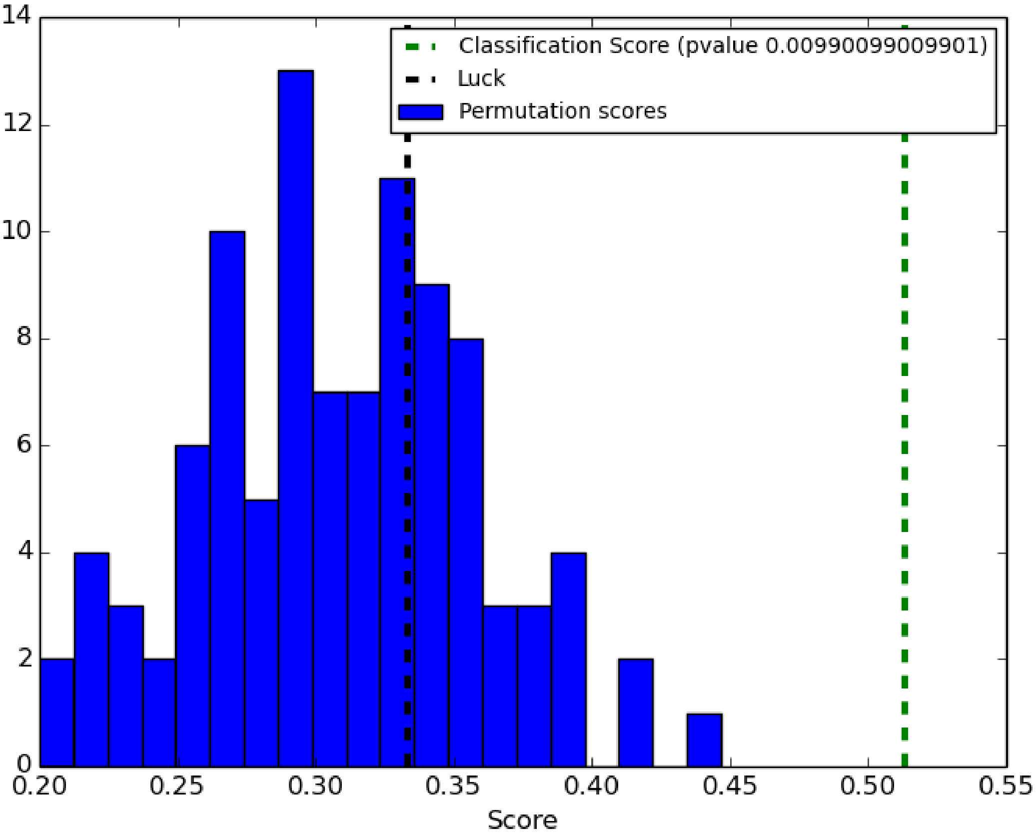


Figure 11.101: *Test with permutations the significance of a classification score*

11.13.4 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.



Script output:

```
Classification score 0.513333333333 (pvalue : 0.00990099009901)
```

Python source code: `plot_permutation_test_for_classification.py`

```
# 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.cross_validation import StratifiedKFold, permutation_test_score
from sklearn import datasets


##############################################################################
# Loading a dataset
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(y, 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')
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 example: 6.16 seconds (0 minutes 6.16 seconds)

11.13.5 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.

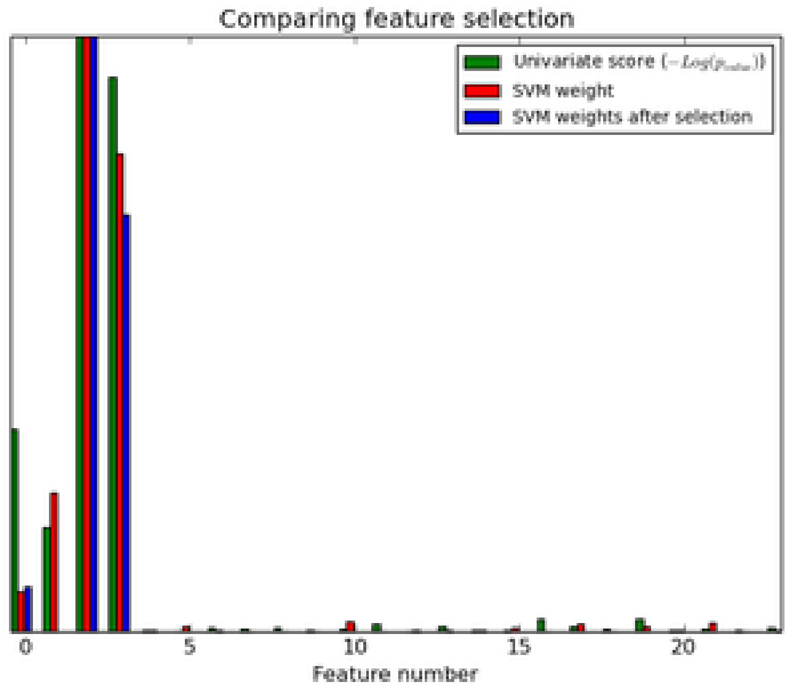
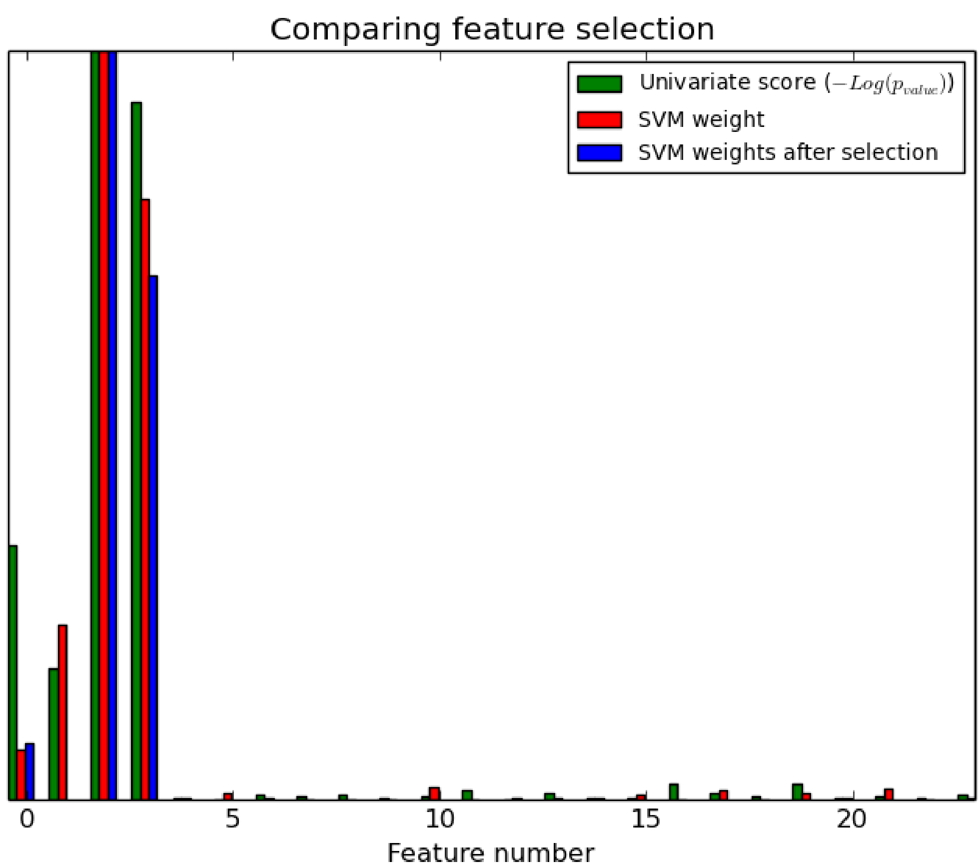


Figure 11.102: *Univariate Feature Selection*



Python source code: `plot_feature_selection.py`

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn import datasets, svm
from sklearn.feature_selection import SelectPercentile, f_classif

###############################################################################
# import some data to play with

# The iris dataset
iris = datasets.load_iris()

# Some noisy data not correlated
E = np.random.uniform(0, 0.1, size=(len(iris.data), 20))

# Add the noisy data to the informative features
X = np.hstack((iris.data, E))
y = iris.target

###############################################################################
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: the 10% most significant features
selector = SelectPercentile(f_classif, percentile=10)
selector.fit(X, y)
scores = -np.log10(selector.pvalues_)
scores /= scores.max()
plt.bar(X_indices - .45, scores, width=.2,
        label=r'Univariate score ($-Log(p_{value})$)', color='g')

###############################################################################
# Compare to the weights of an SVM
clf = svm.SVC(kernel='linear')
clf.fit(X, y)

svm_weights = (clf.coef_ ** 2).sum(axis=0)
svm_weights /= svm_weights.max()

plt.bar(X_indices - .25, svm_weights, width=.2, label='SVM weight', color='r')

clf_selected = svm.SVC(kernel='linear')
clf_selected.fit(selector.transform(X), y)

svm_weights_selected = (clf_selected.coef_ ** 2).sum(axis=0)
svm_weights_selected /= svm_weights_selected.max()

plt.bar(X_indices[selector.get_support()] - .05, svm_weights_selected,
        width=.2, label='SVM weights after selection', color='b')
```

```
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 example: 0.12 seconds (0 minutes 0.12 seconds)

11.14 Gaussian Process for Machine Learning

Examples concerning the `sklearn.gaussian_process` module.

Figure 11.103: *Gaussian Processes regression: goodness-of-fit on the 'diabetes' dataset*

11.14.1 Gaussian Processes regression: goodness-of-fit on the 'diabetes' dataset

In this example, we fit a Gaussian Process model onto the diabetes dataset.

We determine the correlation parameters with maximum likelihood estimation (MLE). We use an anisotropic squared exponential correlation model with a constant regression model. We also use a nugget of 1e-2 to account for the (strong) noise in the targets.

We compute a cross-validation estimate of the coefficient of determination (R2) without reperforming MLE, using the set of correlation parameters found on the whole dataset.

Python source code: `gp_diabetes_dataset.py`

```
print(__doc__)

# Author: Vincent Dubourg <vincent.dubourg@gmail.com>
# Licence: BSD 3 clause

from sklearn import datasets
from sklearn.gaussian_process import GaussianProcess
from sklearn.cross_validation import cross_val_score, KFold

# Load the dataset from scikit's data sets
diabetes = datasets.load_diabetes()
X, y = diabetes.data, diabetes.target
```

```
# Instanciate a GP model
gp = GaussianProcess(regr='constant', corr='absolute_exponential',
                     theta0=[1e-4] * 10, thetaL=[1e-12] * 10,
                     thetaU=[1e-2] * 10, nugget=1e-2, optimizer='Welch')

# Fit the GP model to the data performing maximum likelihood estimation
gp.fit(X, y)

# Deactivate maximum likelihood estimation for the cross-validation loop
gp.theta0 = gp.theta_  # Given correlation parameter = MLE
gp.thetaL, gp.thetaU = None, None  # None bounds deactivate MLE

# Perform a cross-validation estimate of the coefficient of determination using
# the cross_validation module using all CPUs available on the machine
K = 20  # folds
R2 = cross_val_score(gp, X, y=y, cv=KFold(y.size, K), n_jobs=1).mean()
print("The %d-Folds estimate of the coefficient of determination is R2 = %s"
      % (K, R2))
```

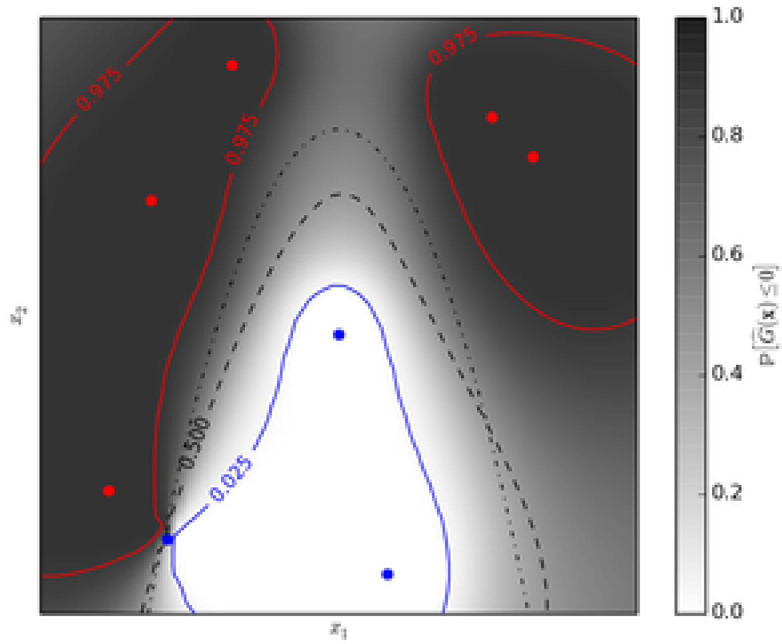
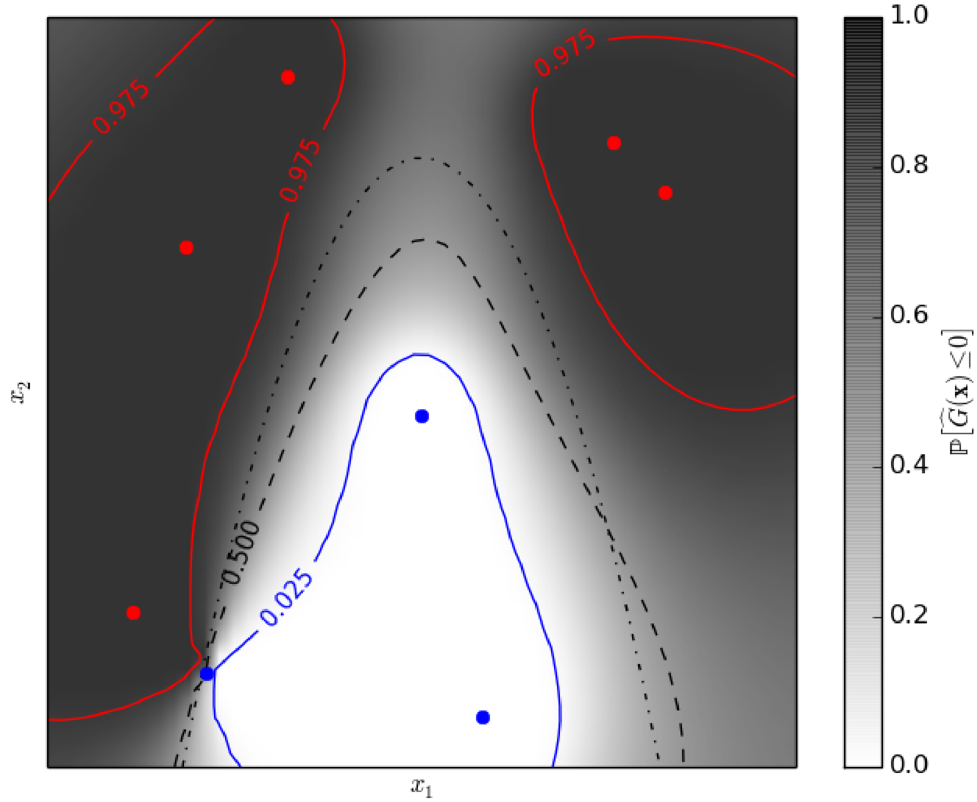


Figure 11.104: *Gaussian Processes classification example: exploiting the probabilistic output*

11.14.2 Gaussian Processes classification example: exploiting the probabilistic output

A two-dimensional regression exercise with a post-processing allowing for probabilistic classification thanks to the Gaussian property of the prediction.

The figure illustrates the probability that the prediction is negative with respect to the remaining uncertainty in the prediction. The red and blue lines corresponds to the 95% confidence interval on the prediction of the zero level set.



Python source code: `plot_gp_probabilistic_classification_after_regression.py`

```
print(__doc__)

# Author: Vincent Dubourg <vincent.dubourg@gmail.com>
# Licence: BSD 3 clause

import numpy as np
from scipy import stats
from sklearn.gaussian_process import GaussianProcess
from matplotlib import pyplot as pl
from matplotlib import cm

# Standard normal distribution functions
phi = stats.distributions.norm().pdf
PHI = stats.distributions.norm().cdf
PHIinv = stats.distributions.norm().ppf

# 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 = g(X)

# Instanciate and fit Gaussian Process Model
gp = GaussianProcess(theta0=5e-1)

# Don't perform MLE or you'll get a perfect prediction for this simple example!
gp.fit(X, y)

# Evaluate real function, the prediction and its MSE on a grid
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

y_true = g(xx)
y_pred, MSE = gp.predict(xx, eval_MSE=True)
sigma = np.sqrt(MSE)
y_true = y_true.reshape((res, res))
y_pred = y_pred.reshape((res, res))
sigma = sigma.reshape((res, res))
k = PHIinv(.975)

# Plot the probabilistic classification iso-values using the Gaussian property
# of the prediction
fig = pl.figure(1)
ax = fig.add_subplot(111)
ax.axes.set_aspect('equal')
pl.xticks([])
pl.yticks([])
ax.set_xticklabels([])
ax.set_yticklabels([])
pl.xlabel('$x_1$')
pl.ylabel('$x_2$')

cax = pl.imshow(np.flipud(PHI(- y_pred / sigma)), cmap=cm.gray_r, alpha=0.8,
                extent=(- lim, lim, - lim, lim))
norm = pl.matplotlib.colors.Normalize(vmin=0., vmax=0.9)
cb = pl.colorbar(cax, ticks=[0., 0.2, 0.4, 0.6, 0.8, 1.], norm=norm)
cb.set_label('${\\rm \mathbb{P}}\left[\widehat{G}(\mathbf{x}) \leq 0\\right]$')

pl.plot(X[y <= 0, 0], X[y <= 0, 1], 'r.', markersize=12)

pl.plot(X[y > 0, 0], X[y > 0, 1], 'b.', markersize=12)

cs = pl.contour(x1, x2, y_true, [0.], colors='k', linestyles='dashdot')

cs = pl.contour(x1, x2, PHI(- y_pred / sigma), [0.025], colors='b',
```

```
                linestyles='solid')
pl.clabel(cs, fontsize=11)

cs = pl.contour(x1, x2, PHI(- y_pred / sigma), [0.5], colors='k',
                linestyles='dashed')
pl.clabel(cs, fontsize=11)

cs = pl.contour(x1, x2, PHI(- y_pred / sigma), [0.975], colors='r',
                linestyles='solid')
pl.clabel(cs, fontsize=11)

pl.show()
```

Total running time of the example: 0.12 seconds (0 minutes 0.12 seconds)

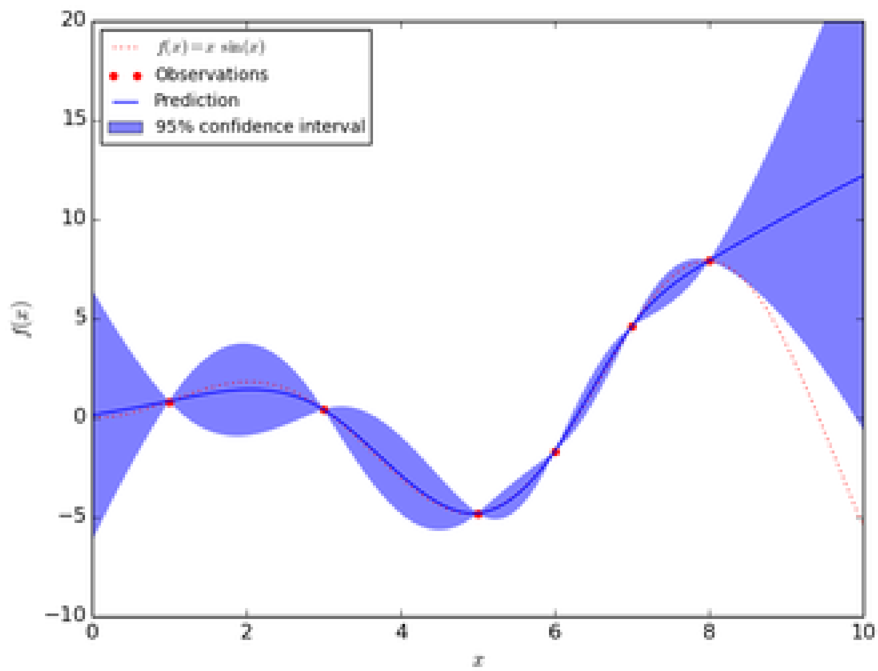


Figure 11.105: *Gaussian Processes regression: basic introductory example*

11.14.3 Gaussian Processes regression: basic introductory example

A simple one-dimensional regression exercise computed in two different ways:

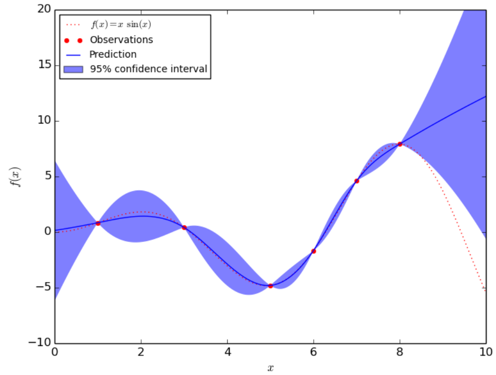
1. A noise-free case with a cubic correlation model
2. A noisy case with a squared Euclidean correlation model

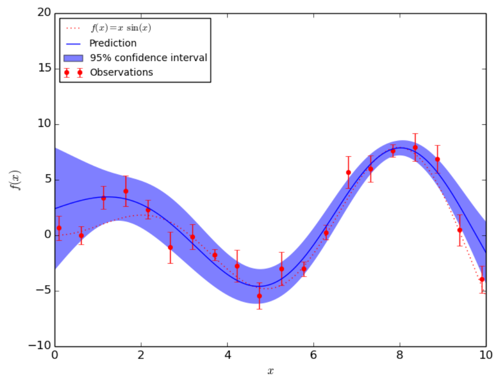
In both cases, the model parameters are estimated using the maximum likelihood principle.

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 `nugget` is applied as a Tikhonov regularization of the assumed covariance between the training points. In the special case of the squared euclidean correlation model, nugget is mathematically equivalent to a normalized variance: That is

$$\mathrm{nugget}_i = \left[\frac{\sigma_i}{y_i}\right]^2$$

- 

- 

Python source code: `plot_gp_regression.py`

```
print(__doc__)

# Author: Vincent Dubourg <vincent.dubourg@gmail.com>
#         Jake Vanderplas <vanderplas@astro.washington.edu>
# Licence: BSD 3 clause

import numpy as np
from sklearn.gaussian_process import GaussianProcess
from matplotlib import pyplot as pl

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()

# 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

# Instanciate a Gaussian Process model
gp = GaussianProcess(corr='cubic', theta0=1e-2, thetaL=1e-4, thetaU=1e-1,
                     random_start=100)

# 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, MSE = gp.predict(x, eval_MSE=True)
sigma = np.sqrt(MSE)

# Plot the function, the prediction and the 95% confidence interval based on
# the MSE
fig = pl.figure()
pl.plot(x, f(x), 'r:', label=u'$f(x) = x\,\sin(x)$')
pl.plot(X, y, 'r.', markersize=10, label=u'Observations')
pl.plot(x, y_pred, 'b-', label=u'Prediction')
pl.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')
pl.xlabel('$x$')
pl.ylabel('$f(x)$')
pl.ylim(-10, 20)
pl.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

# 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

# Instanciate a Gaussian Process model
gp = GaussianProcess(corr='squared_exponential', theta0=1e-1,
                     thetaL=1e-3, thetaU=1,
                     nugget=(dy / y) ** 2,
                     random_start=100)

# 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, MSE = gp.predict(x, eval_MSE=True)
sigma = np.sqrt(MSE)

# Plot the function, the prediction and the 95% confidence interval based on
# the MSE
fig = pl.figure()
pl.plot(x, f(x), 'r:', label=u'$f(x) = x\,\sin(x)$')
pl.errorbar(X.ravel(), y, dy, fmt='r.', markersize=10, label=u'Observations')
pl.plot(x, y_pred, 'b-', label=u'Prediction')
pl.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')
pl.xlabel('$x$')
pl.ylabel('$f(x)$')
pl.ylim(-10, 20)
pl.legend(loc='upper left')

pl.show()
```

Total running time of the example: 0.74 seconds (0 minutes 0.74 seconds)

11.15 Generalized Linear Models

Examples concerning the `sklearn.linear_model` module.

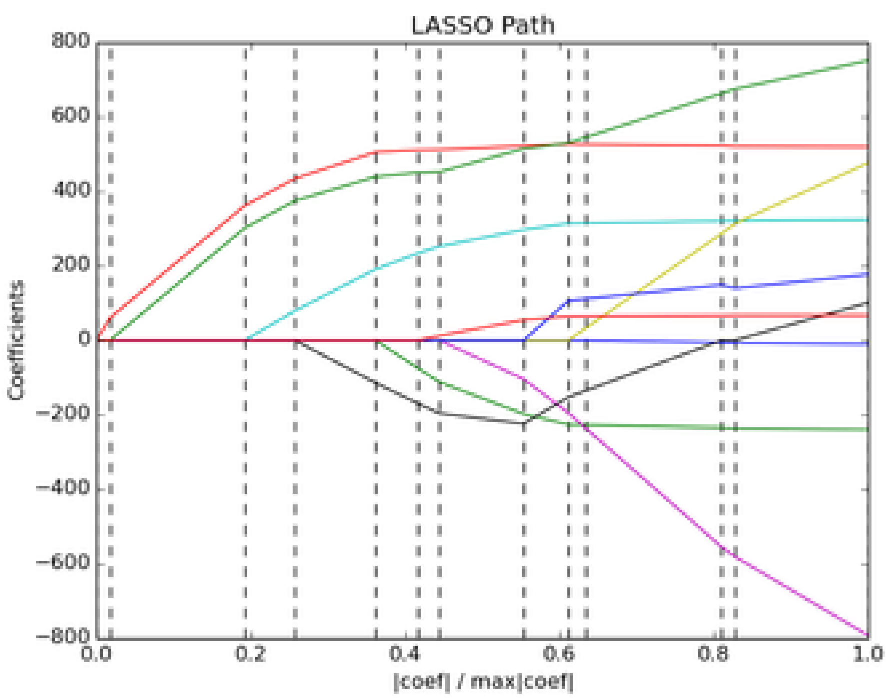
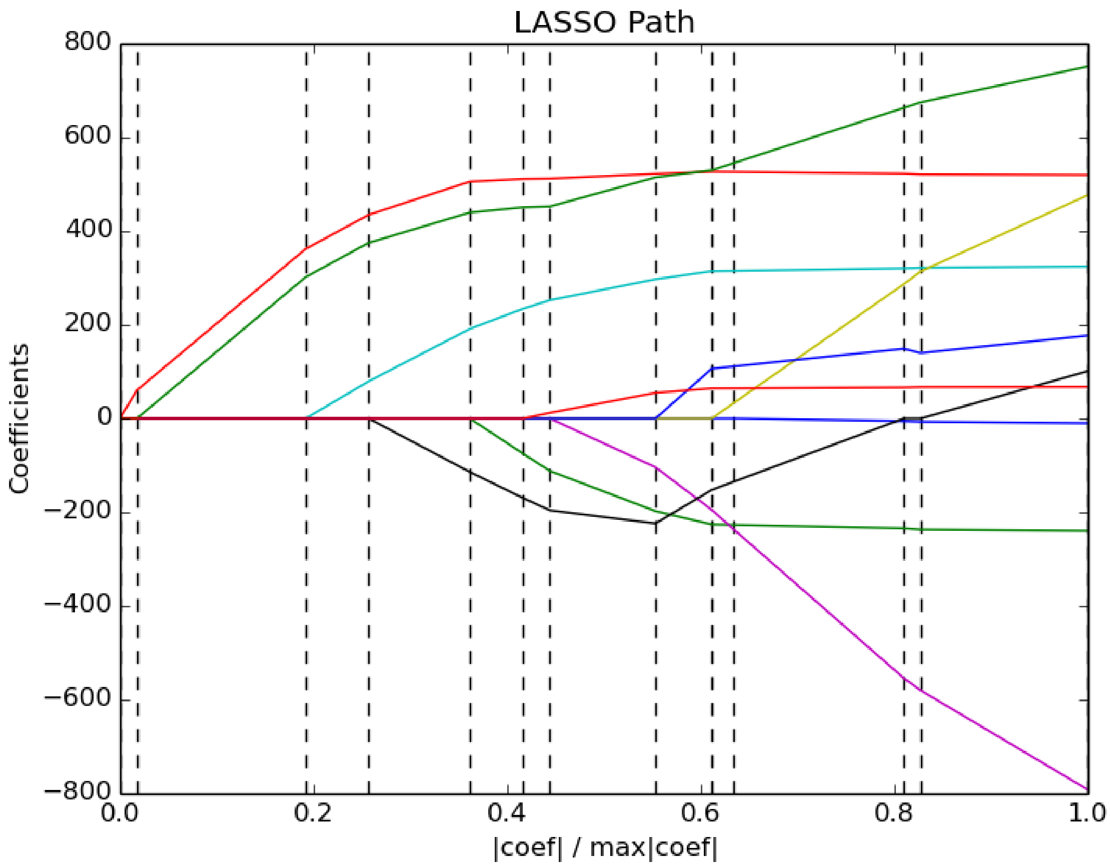


Figure 11.106: *Lasso path using LARS*

11.15.1 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.



Script output:

```
Computing regularization path using the LARS ...
.
```

Python source code: `plot_lasso_lars.py`

```
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

diabetes = datasets.load_diabetes()
```

```
X = diabetes.data
y = diabetes.target

print("Computing regularization path using the LARS ...")
alphas, _, 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 example: 0.09 seconds (0 minutes 0.09 seconds)

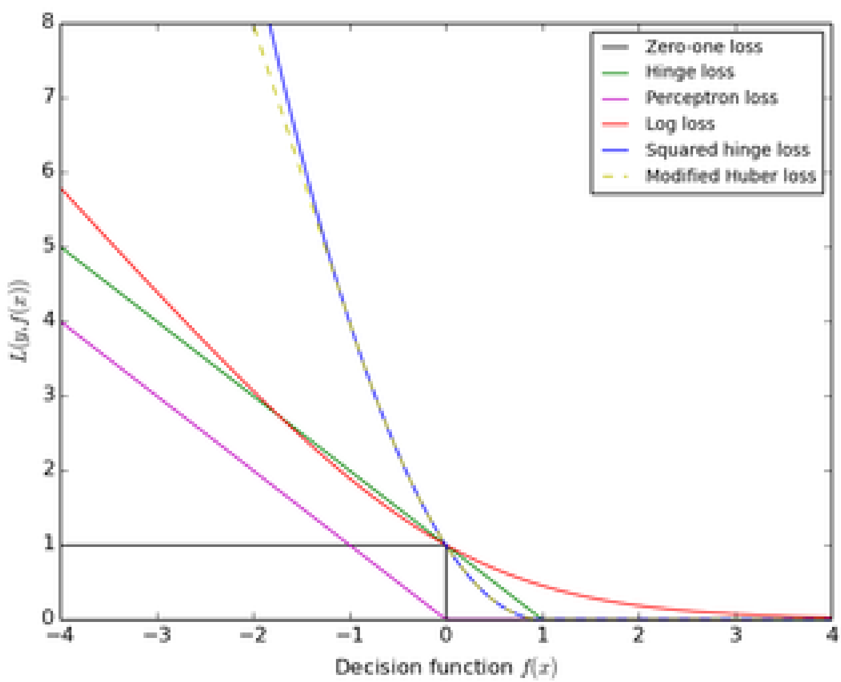
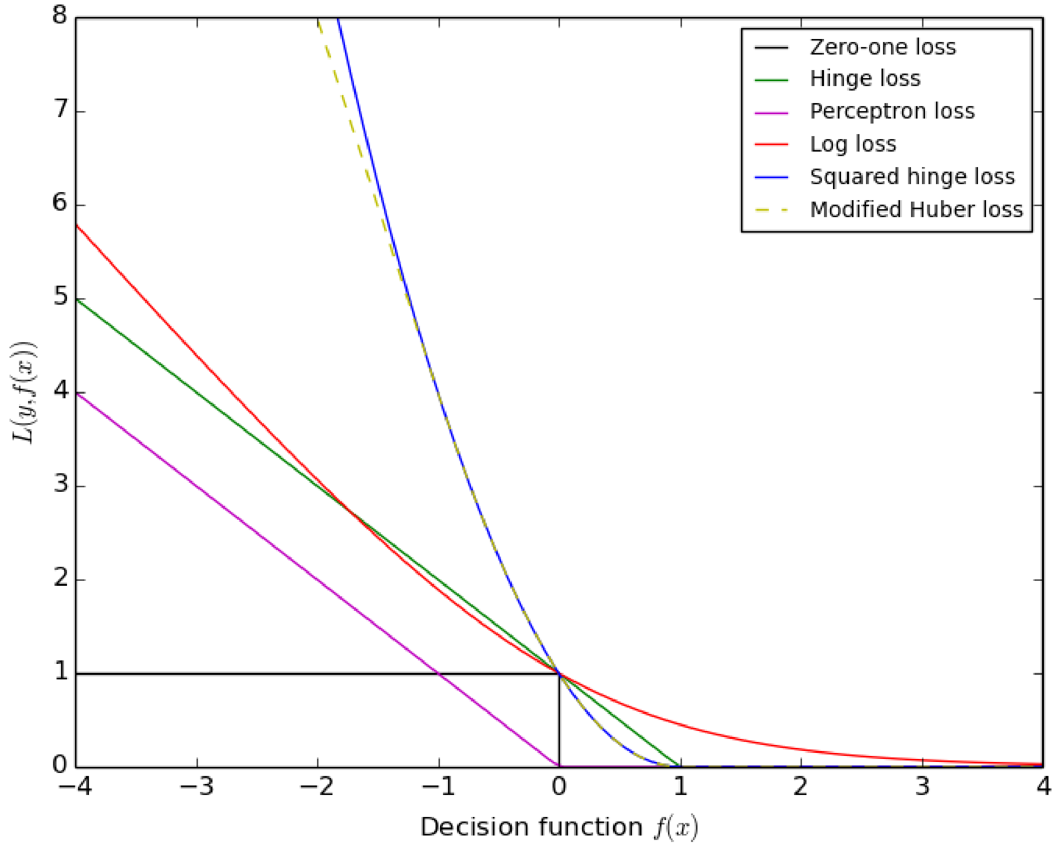


Figure 11.107: *SGD: convex loss functions*

11.15.2 SGD: convex loss functions

A plot that compares the various convex loss functions supported by `sklearn.linear_model.SGDClassifier`
.



Python source code: `plot_sgd_loss_functions.py`

```
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)
plt.plot([xmin, 0, 0, xmax], [1, 1, 0, 0], 'k-',
         label="Zero-one loss")
plt.plot(xx, np.where(xx < 1, 1 - xx, 0), 'g-',
         label="Hinge loss")
plt.plot(xx, -np.minimum(xx, 0), 'm-',
         label="Perceptron loss")
plt.plot(xx, np.log2(1 + np.exp(-xx)), 'r-',
         label="Log loss")
```

```
plt.plot(xx, np.where(xx < 1, 1 - xx, 0) ** 2, 'b-',
         label="Squared hinge loss")
plt.plot(xx, modified_huber_loss(xx, 1), 'y--',
         label="Modified Huber loss")
plt.ylim((0, 8))
plt.legend(loc="upper right")
plt.xlabel(r"Decision function $f(x)$")
plt.ylabel("$L(y, f(x))$")
plt.show()
```

Total running time of the example: 0.09 seconds (0 minutes 0.09 seconds)

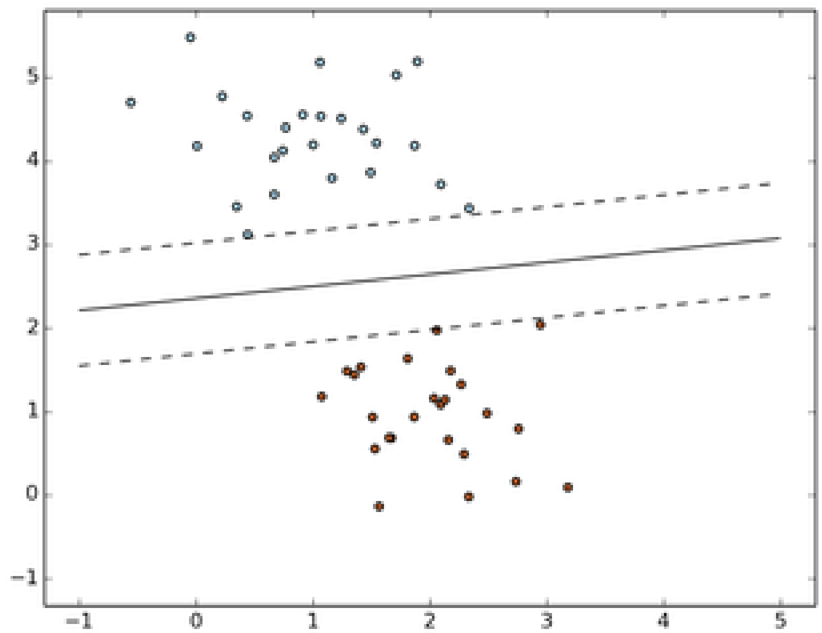
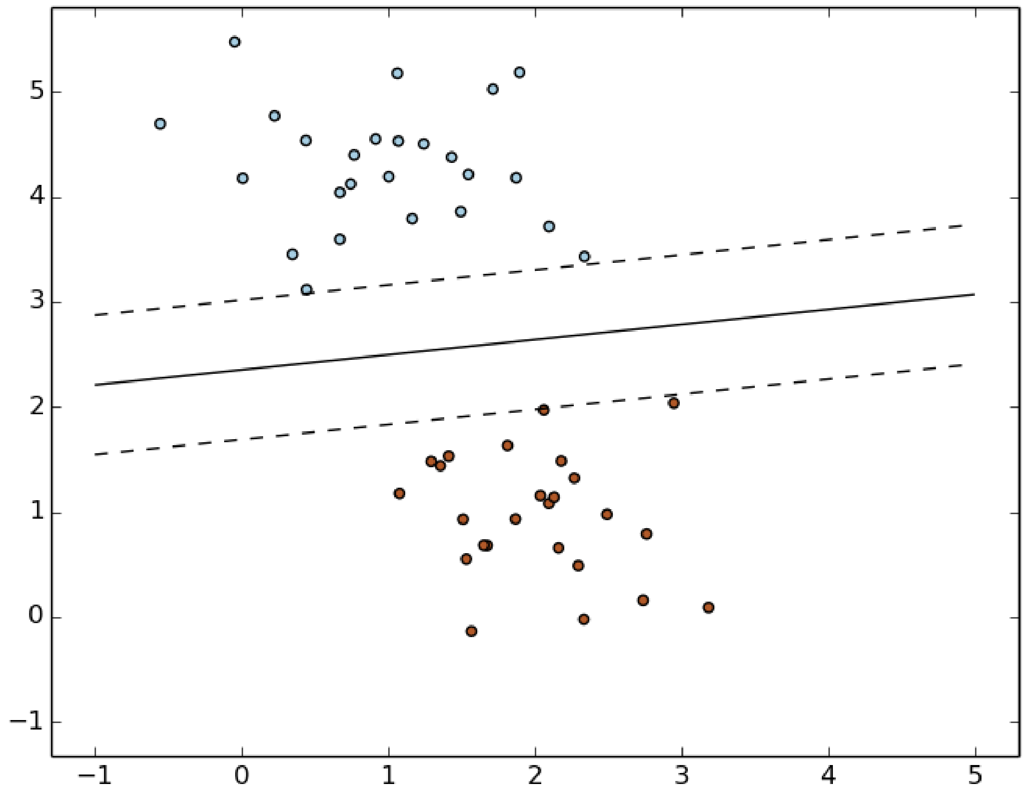


Figure 11.108: *SGD: Maximum margin separating hyperplane*

11.15.3 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.



Python source code: `plot_sgd_separating_hyperplane.py`

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDClassifier
from sklearn.datasets.samples_generator 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, n_iter=200, fit_intercept=True)
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])
```

```
    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)

plt.axis('tight')
plt.show()
```

Total running time of the example: 0.09 seconds (0 minutes 0.09 seconds)

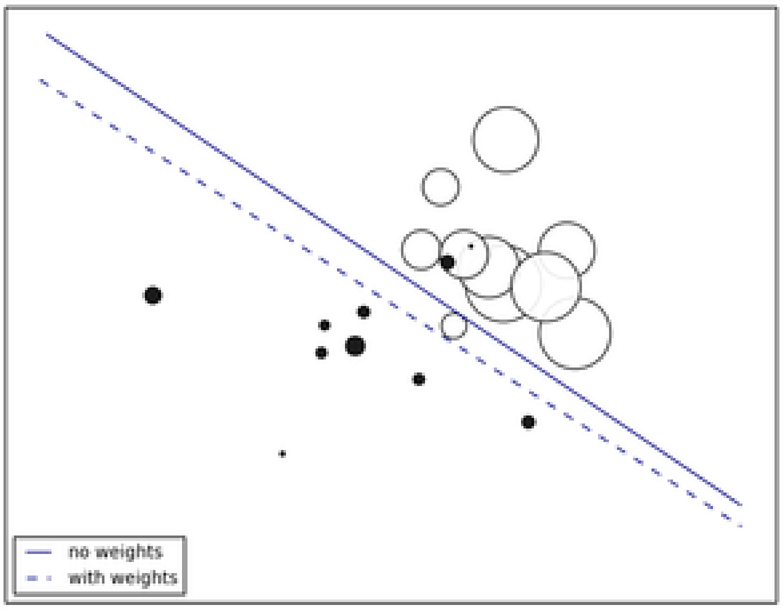
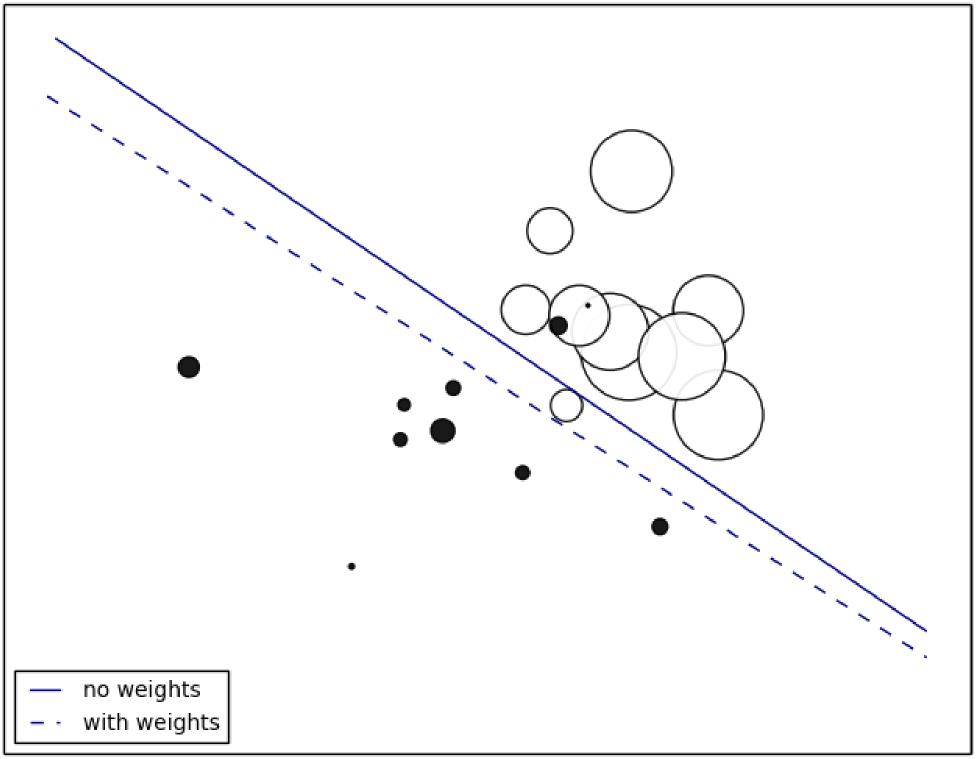


Figure 11.109: *SGD: Weighted samples*

11.15.4 SGD: Weighted samples

Plot decision function of a weighted dataset, where the size of points is proportional to its weight.



Python source code: `plot_sgd_weighted_samples.py`

```
print(__doc__)

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)

## fit the unweighted model
clf = linear_model.SGDClassifier(alpha=0.01, n_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, n_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 example: 0.11 seconds (0 minutes 0.11 seconds)

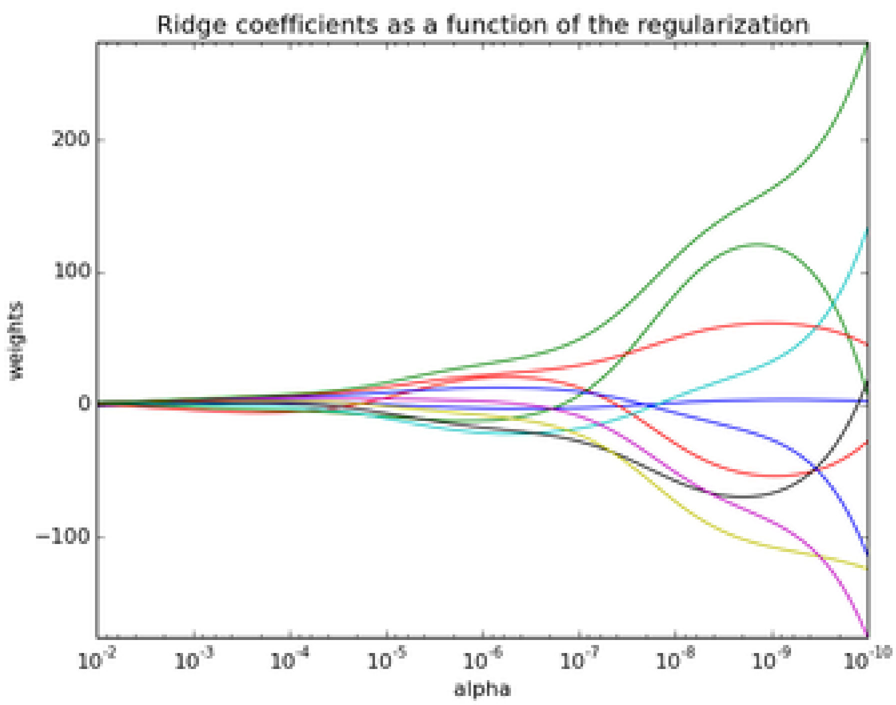


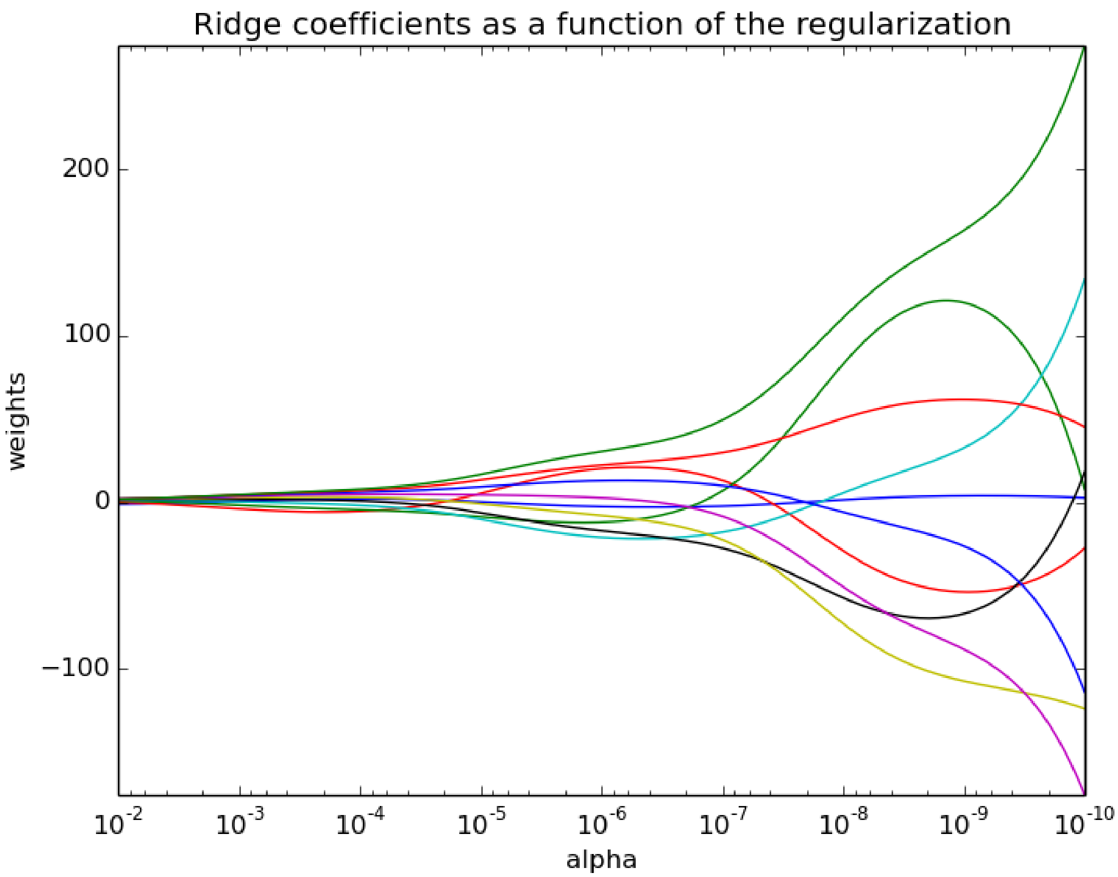
Figure 11.110: *Plot Ridge coefficients as a function of the regularization*

11.15.5 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.

At the end of the path, as alpha tends toward zero and the solution tends towards the ordinary least squares, coefficients exhibit big oscillations.



Python source code: `plot_ridge_path.py`

```
# 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)
clf = linear_model.Ridge(fit_intercept=False)

coefs = []
for a in alphas:
    clf.set_params(alpha=a)
    clf.fit(X, y)
```

```
    coefs.append(clf.coef_)

###############################################################################
# Display results

ax = plt.gca()
ax.set_color_cycle(['b', 'r', 'g', 'c', 'k', 'y', 'm'])

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 example: 0.12 seconds (0 minutes 0.12 seconds)

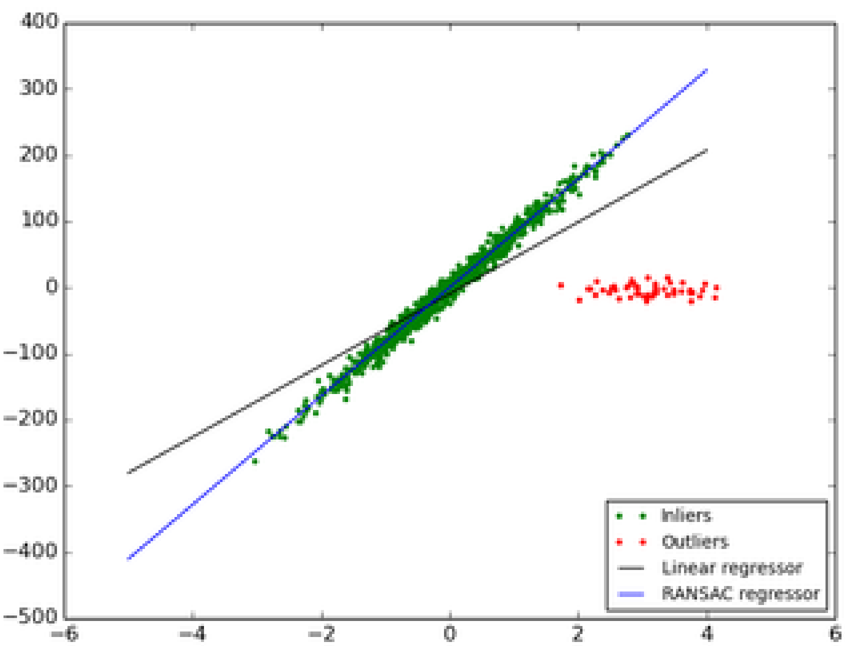
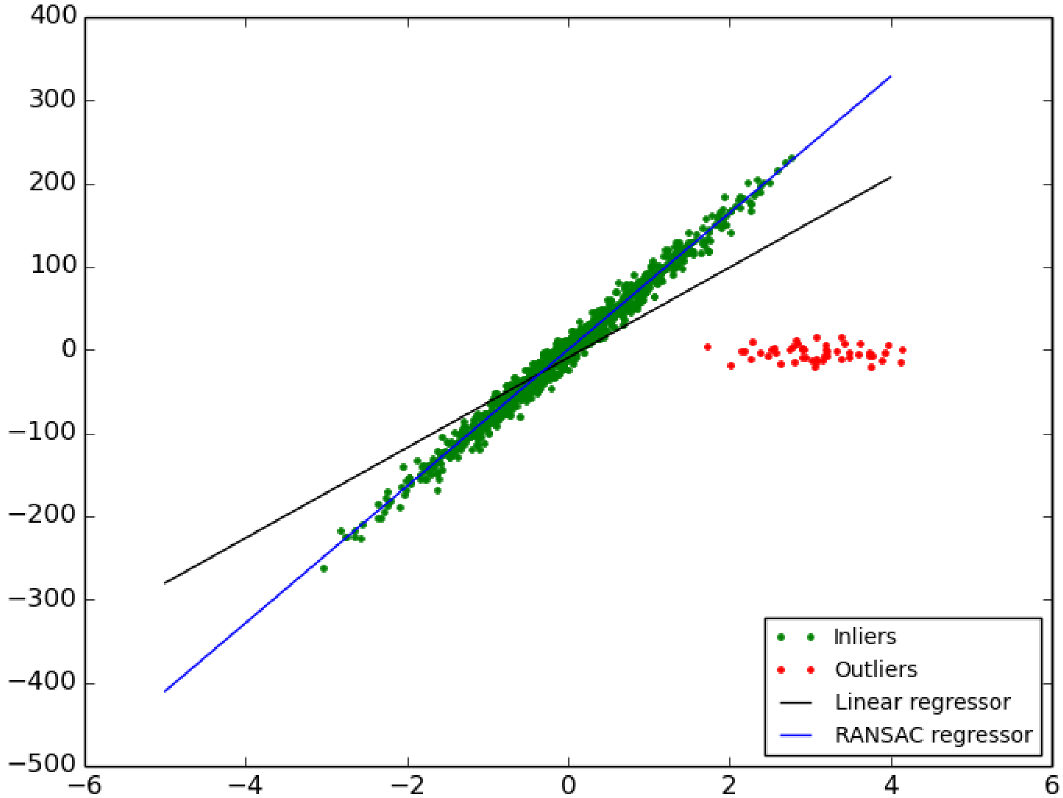


Figure 11.111: *Robust linear model estimation using RANSAC*

11.15.6 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.



Script output:

```
Estimated coefficients (true, normal, RANSAC):
82.1903908408 [ 54.17236387] [[ 82.08533159]]
```

Python source code: `plot_ransac.py`

```python
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
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
model = linear_model.LinearRegression()
model.fit(X, y)

# Robustly fit linear model with RANSAC algorithm
model_ransac = linear_model.RANSACRegressor(linear_model.LinearRegression())
model_ransac.fit(X, y)
inlier_mask = model_ransac.inlier_mask_
outlier_mask = np.logical_not(inlier_mask)

# Predict data of estimated models
line_X = np.arange(-5, 5)
line_y = model.predict(line_X[:, np.newaxis])
line_y_ransac = model_ransac.predict(line_X[:, np.newaxis])

# Compare estimated coefficients
print("Estimated coefficients (true, normal, RANSAC):")
print(coef, model.coef_, model_ransac.estimator_.coef_)

plt.plot(X[inlier_mask], y[inlier_mask], '.g', label='Inliers')
plt.plot(X[outlier_mask], y[outlier_mask], '.r', label='Outliers')
plt.plot(line_X, line_y, '-k', label='Linear regressor')
plt.plot(line_X, line_y_ransac, '-b', label='RANSAC regressor')
plt.legend(loc='lower right')
plt.show()
```

Total running time of the example: 0.10 seconds (0 minutes 0.10 seconds)

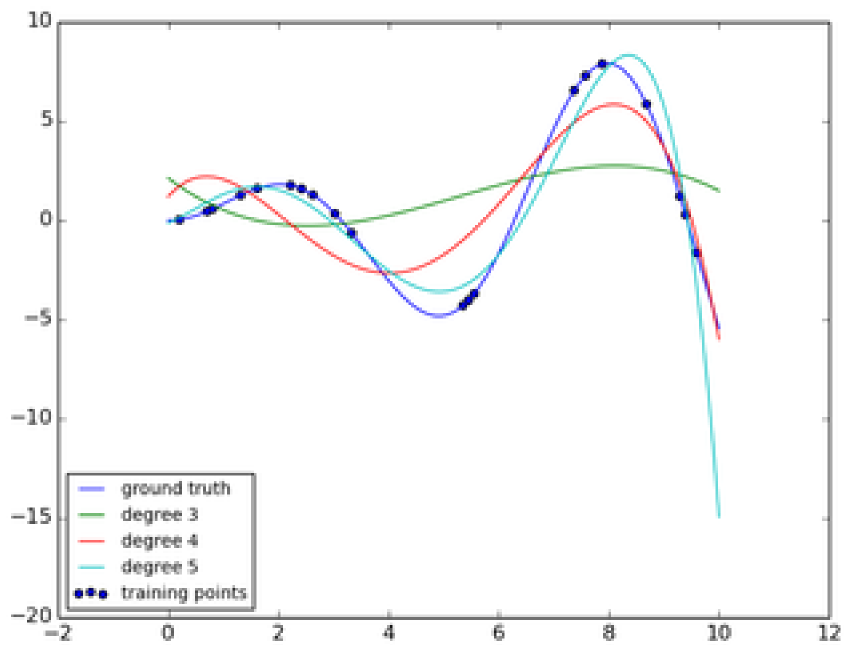


Figure 11.112: *Polynomial interpolation*

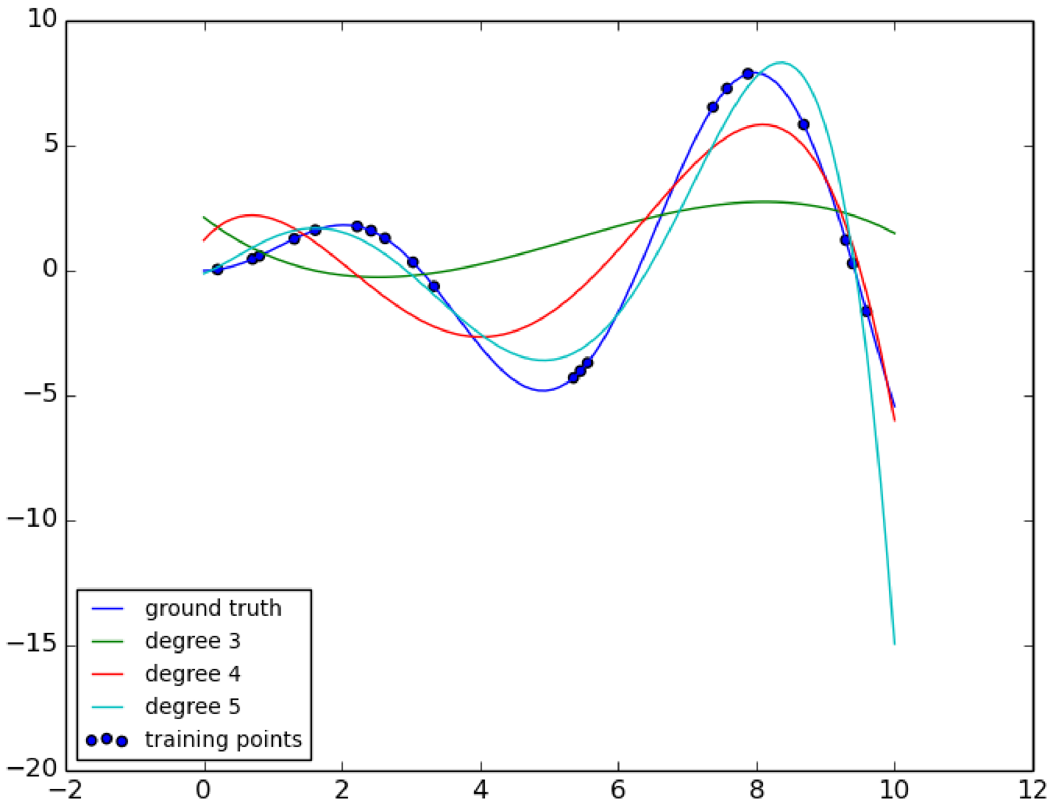
11.15.7 Polynomial interpolation

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.



Python source code: `plot_polynomial_interpolation.py`

```
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)
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]

plt.plot(x_plot, f(x_plot), label="ground truth")
plt.scatter(x, y, label="training points")

for degree in [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, label="degree %d" % degree)

plt.legend(loc='lower left')

plt.show()
```

Total running time of the example: 0.09 seconds (0 minutes 0.09 seconds)

11.15.8 Logistic Regression 3-class Classifier

Show below is a logistic-regression classifiers decision boundaries on the iris dataset. The datapoints are colored according to their labels.

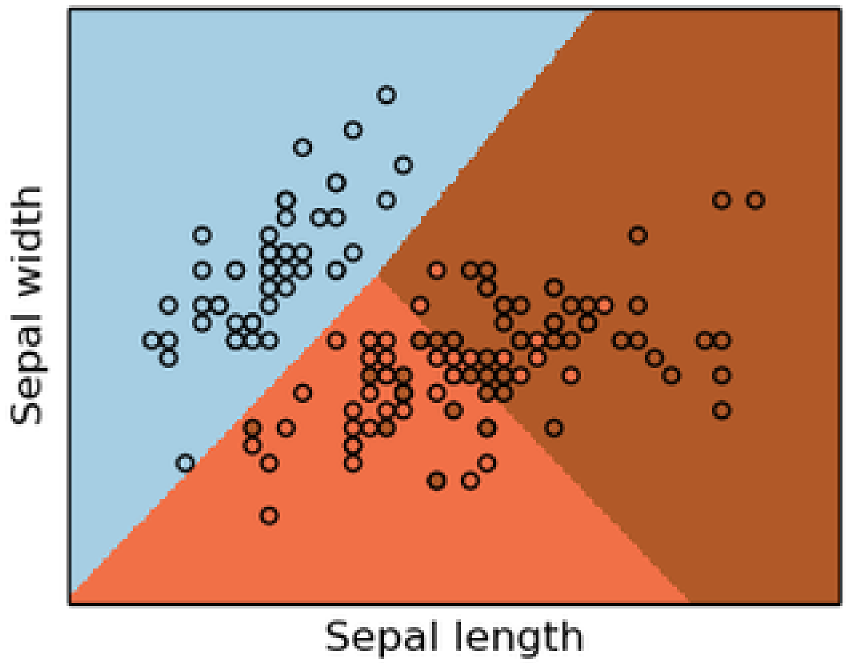
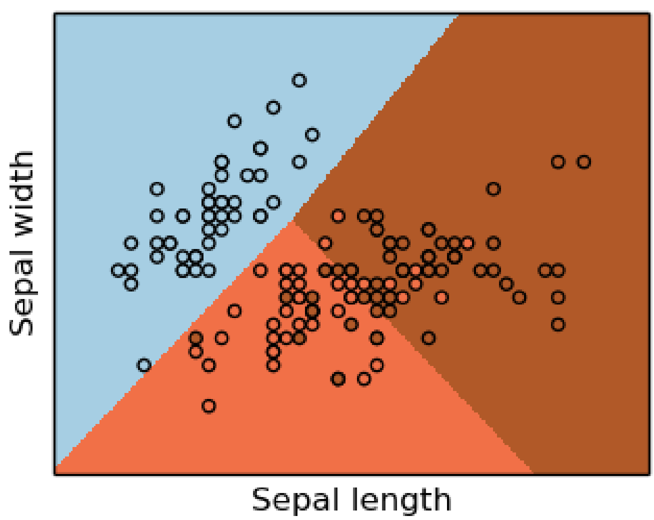


Figure 11.113: *Logistic Regression 3-class Classifier*



Python source code: `plot_iris_logistic.py`

```
print(__doc__)


# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause
```

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model, 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

h = .02  # step size in the mesh

logreg = linear_model.LogisticRegression(C=1e5)

# we create an instance of Neighbours 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, m_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
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')
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 example: 0.09 seconds (0 minutes 0.09 seconds)

11.15.9 Path with L1- Logistic Regression

Computes path on IRIS dataset.

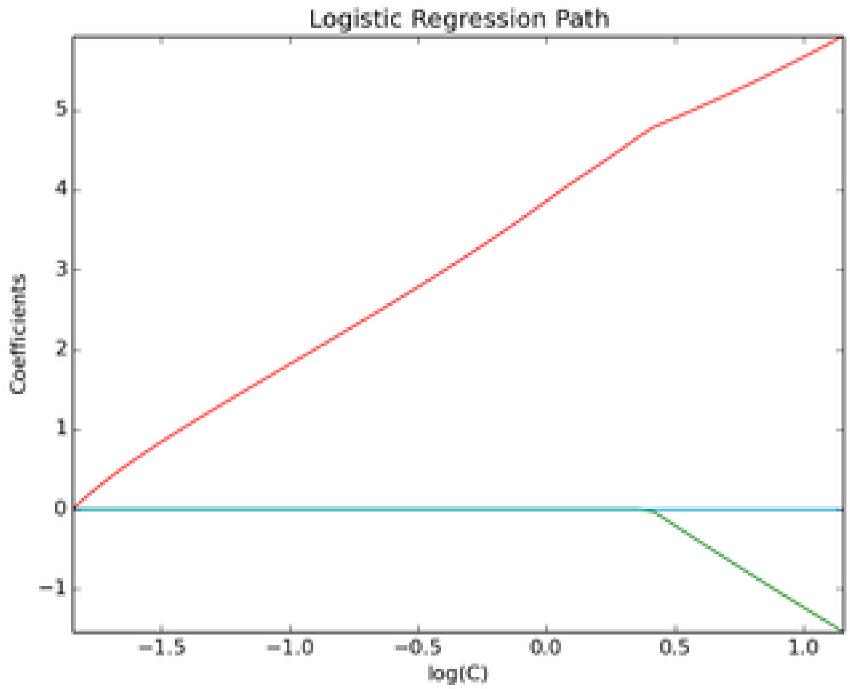
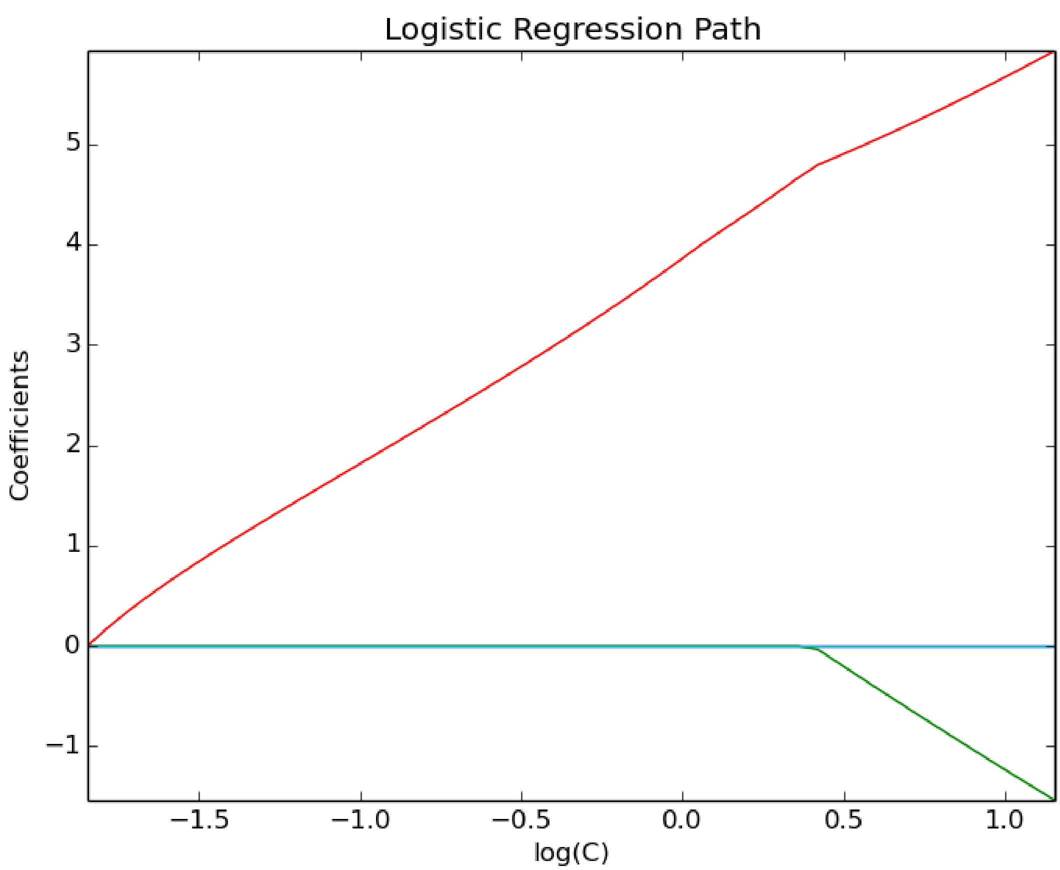


Figure 11.114: *Path with L1- Logistic Regression*



Script output:

```
Computing regularization path ...
This took  0:00:00.016999
```

Python source code: `plot_logistic_path.py`

```
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

from datetime import datetime
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
y = iris.target

X = X[y != 2]
y = y[y != 2]

X -= np.mean(X, 0)

###############################################################################
# Demo path functions

cs = l1_min_c(X, y, loss='log') * np.logspace(0, 3)


print("Computing regularization path ...")
start = datetime.now()
clf = linear_model.LogisticRegression(C=1.0, penalty='l1', tol=1e-6)
coefs_ = []
for c in cs:
    clf.set_params(C=c)
    clf.fit(X, y)
    coefs_.append(clf.coef_.ravel().copy())
print("This took ", datetime.now() - start)

coefs_ = np.array(coefs_)
plt.plot(np.log10(cs), coefs_)
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 example: 0.10 seconds (0 minutes 0.10 seconds)

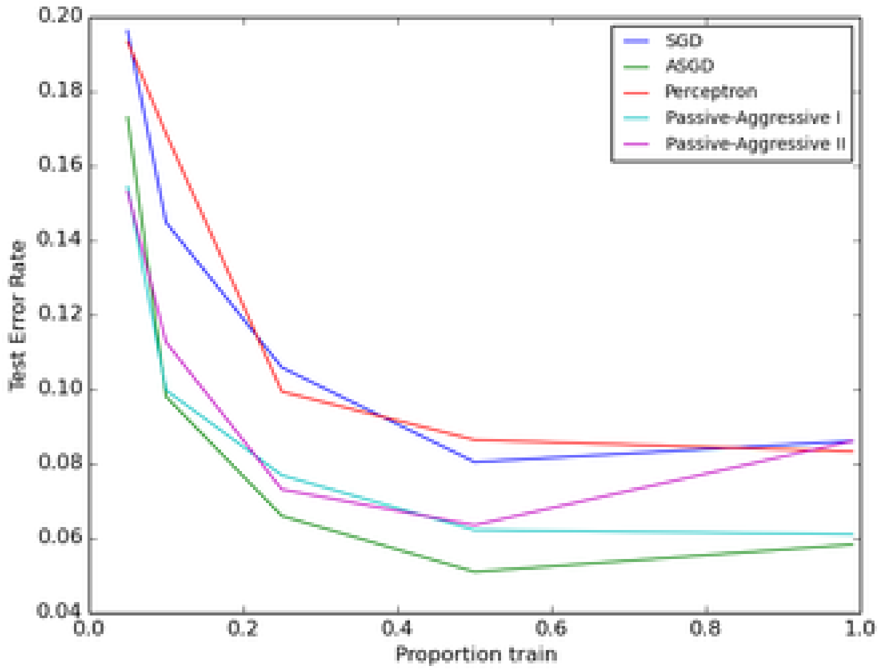
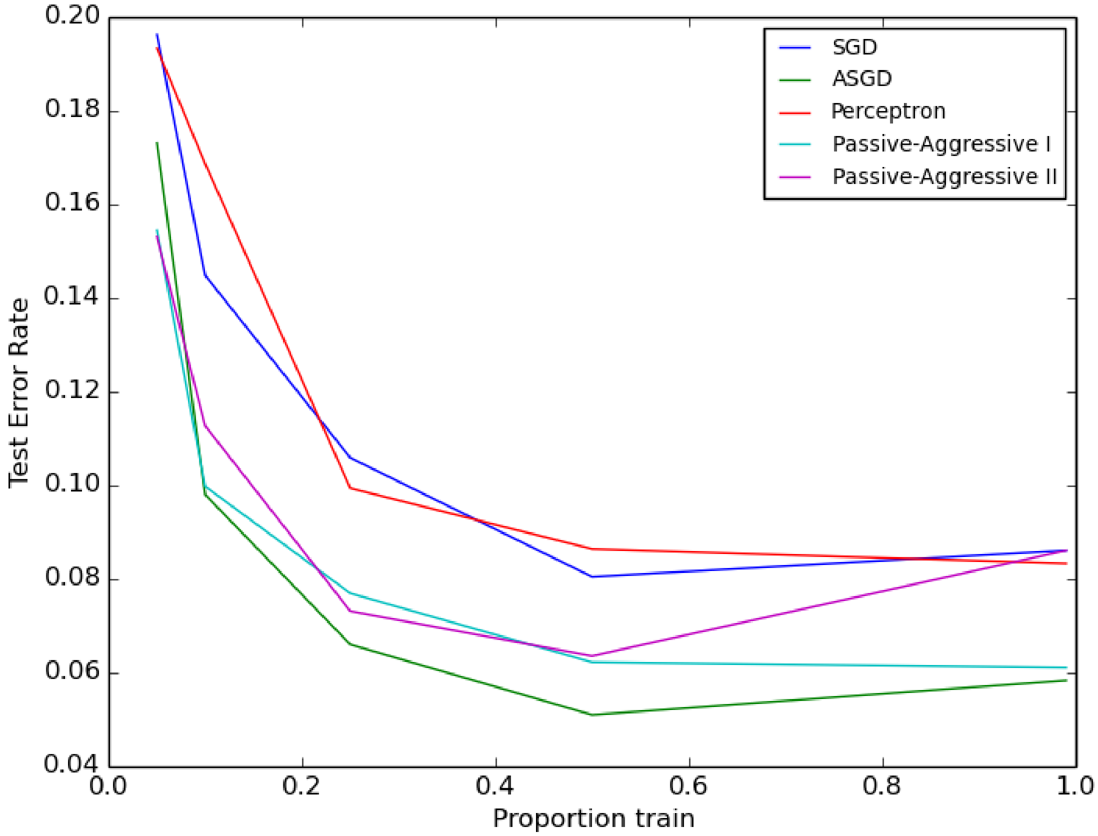


Figure 11.115: *Comparing various online solvers*

11.15.10 Comparing various online solvers

An example showing how different online solvers perform on the hand-written digits dataset.



Python source code: `plot_sgd_comparison.py`

```
# 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.cross_validation import train_test_split
from sklearn.linear_model import SGDClassifier, Perceptron
from sklearn.linear_model import PassiveAggressiveClassifier

heldout = [0.95, 0.90, 0.75, 0.50, 0.01]
rounds = 20
digits = datasets.load_digits()
X, y = digits.data, digits.target

classifiers = [
    ("SGD", SGDClassifier()),
    ("ASGD", SGDClassifier(average=True)),
    ("Perceptron", Perceptron()),
    ("Passive-Aggressive I", PassiveAggressiveClassifier(loss='hinge',
                                                         C=1.0)),
    ("Passive-Aggressive II", PassiveAggressiveClassifier(loss='squared_hinge',
                                                          C=1.0)),
```

```
]

xx = 1. - np.array(heldout)

for name, clf in classifiers:
    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 example: 3.54 seconds (0 minutes 3.54 seconds)

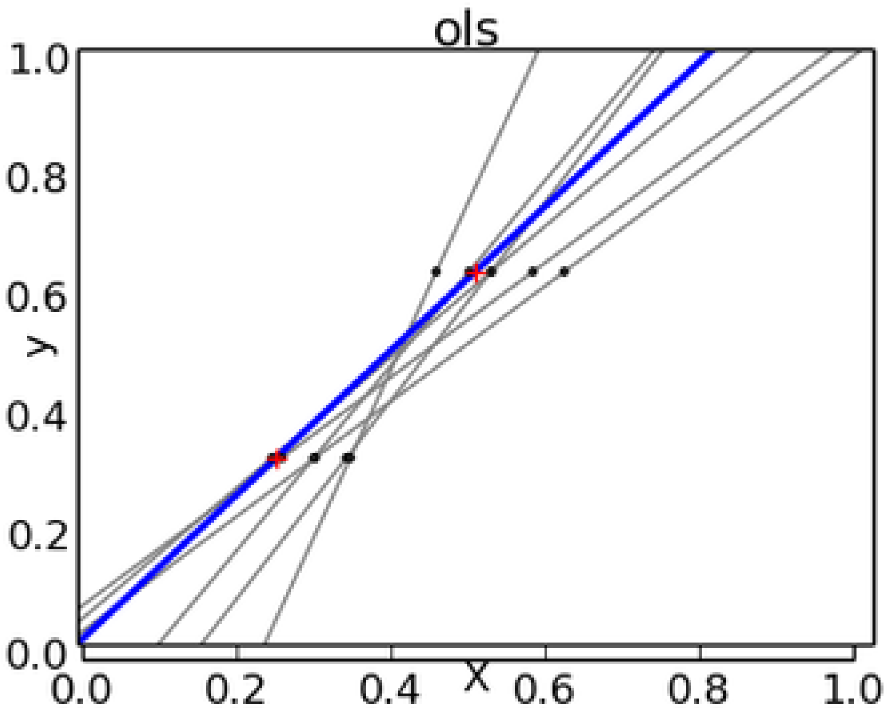
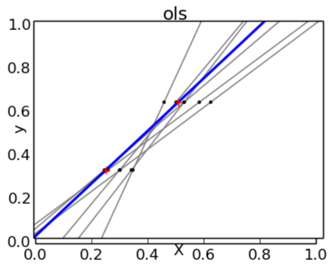


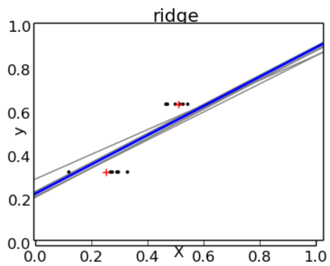
Figure 11.116: *Ordinary Least Squares and Ridge Regression Variance*

11.15.11 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

- 

- 

Python source code: `plot_ols_ridge_variance.py`

```
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

np.random.seed(0)

classifiers = dict(ols=linear_model.LinearRegression(),
                   ridge=linear_model.Ridge(alpha=.1))

fignum = 1
for name, clf in classifiers.items():
    fig = plt.figure(fignum, figsize=(4, 3))
    plt.clf()
    plt.title(name)
```

```
    ax = plt.axes([.12, .12, .8, .8])

    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='.5')
        ax.scatter(this_X, y_train, s=3, c='.5', 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='r', marker='+', zorder=10)

    ax.set_xticks(())
    ax.set_yticks(())
    ax.set_ylim((0, 1.6))
    ax.set_xlabel('X')
    ax.set_ylabel('y')
    ax.set_xlim(0, 2)
    fignum += 1

plt.show()
```

Total running time of the example: 0.18 seconds (0 minutes 0.18 seconds)

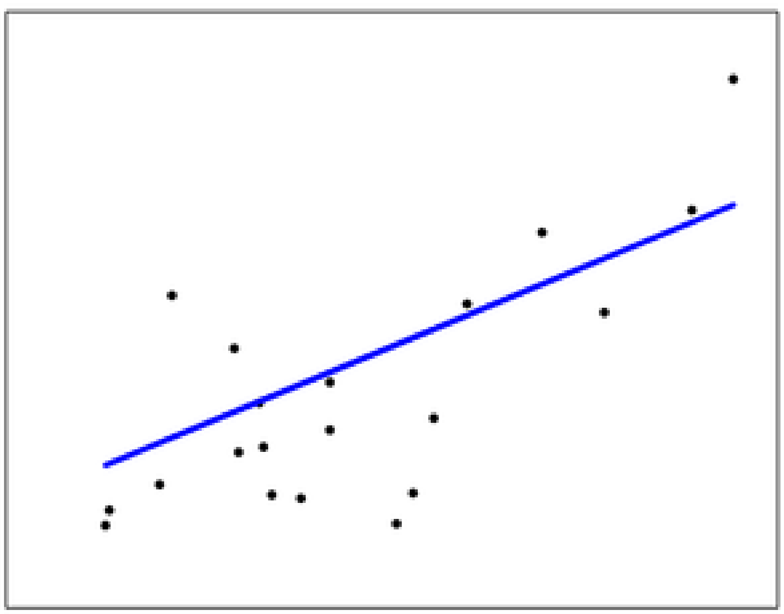
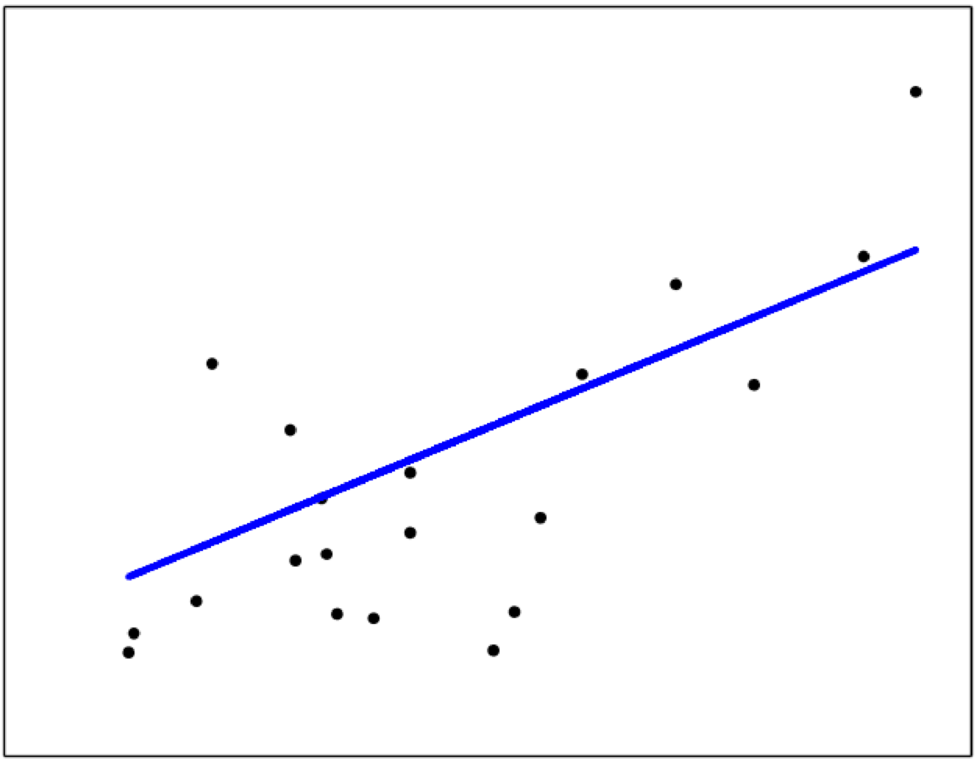


Figure 11.117: *Linear Regression Example*

11.15.12 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 variance score are also calculated.



Script output:

```
Coefficients:
 [ 938.23786125]
Residual sum of squares: 2548.07
Variance score: 0.47
```

Python source code: `plot_ols.py`

```
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

# Load the diabetes dataset
diabetes = datasets.load_diabetes()


# Use only one feature
diabetes_X = diabetes.data[:, np.newaxis]
diabetes_X_temp = diabetes_X[:, :, 2]

# Split the data into training/testing sets
diabetes_X_train = diabetes_X_temp[:-20]
diabetes_X_test = diabetes_X_temp[-20:]

# Split the targets into training/testing sets
diabetes_y_train = diabetes.target[:-20]
diabetes_y_test = diabetes.target[-20:]

# Create linear regression object
regr = linear_model.LinearRegression()

# Train the model using the training sets
regr.fit(diabetes_X_train, diabetes_y_train)

# The coefficients
print('Coefficients: \n', regr.coef_)
# The mean square error
print("Residual sum of squares: %.2f"
      % np.mean((regr.predict(diabetes_X_test) - diabetes_y_test) ** 2))
# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % regr.score(diabetes_X_test, diabetes_y_test))

# Plot outputs
plt.scatter(diabetes_X_test, diabetes_y_test,  color='black')
plt.plot(diabetes_X_test, regr.predict(diabetes_X_test), color='blue',
         linewidth=3)

plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the example: 0.09 seconds (0 minutes 0.09 seconds)

11.15.13 Logit function

Show 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 logit-curve.

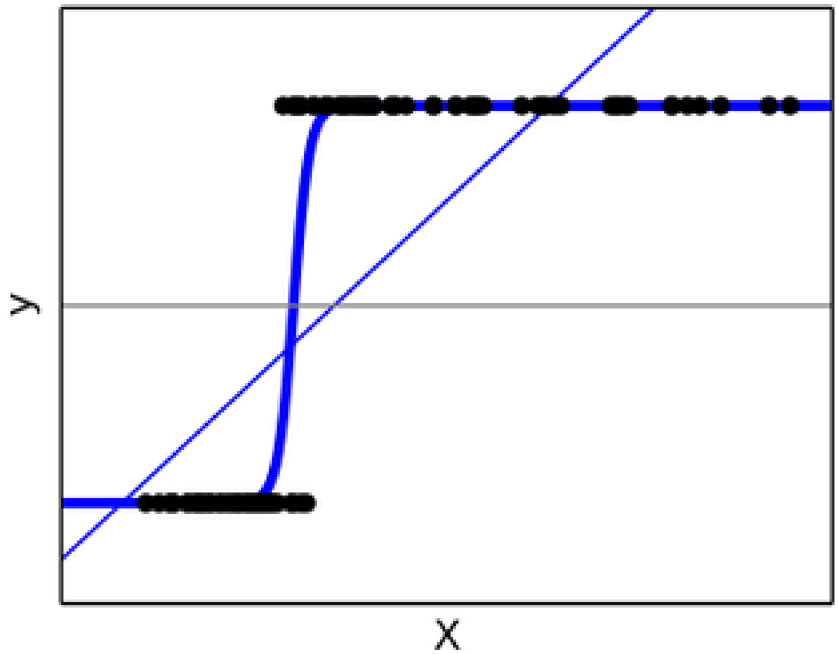
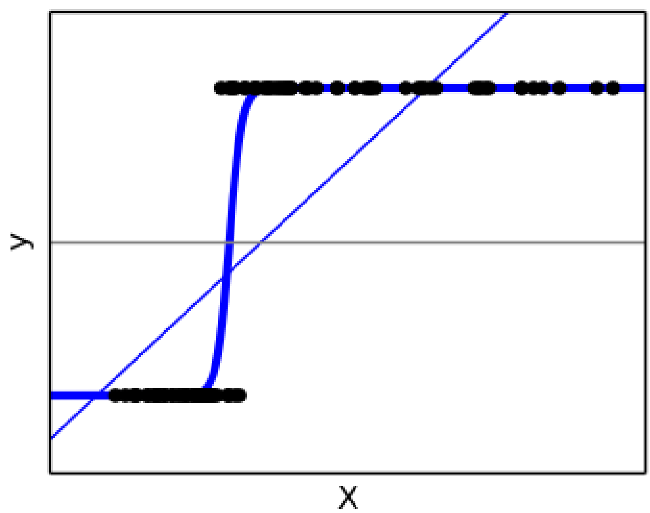


Figure 11.118: *Logit function*



Python source code: `plot_logistic.py`

```
print(__doc__)


# Code source: Gael Varoquaux
# License: BSD 3 clause

import numpy as np
```

```python
import matplotlib.pyplot as plt

from sklearn import linear_model

# this is our test set, 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]
# run 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)


def model(x):
    return 1 / (1 + np.exp(-x))
loss = model(X_test * clf.coef_ + clf.intercept_).ravel()
plt.plot(X_test, loss, color='blue', linewidth=3)

ols = linear_model.LinearRegression()
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(())
plt.yticks(())
plt.ylim(-.25, 1.25)
plt.xlim(-4, 10)

plt.show()
```

Total running time of the example: 0.09 seconds (0 minutes 0.09 seconds)

11.15.14 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.

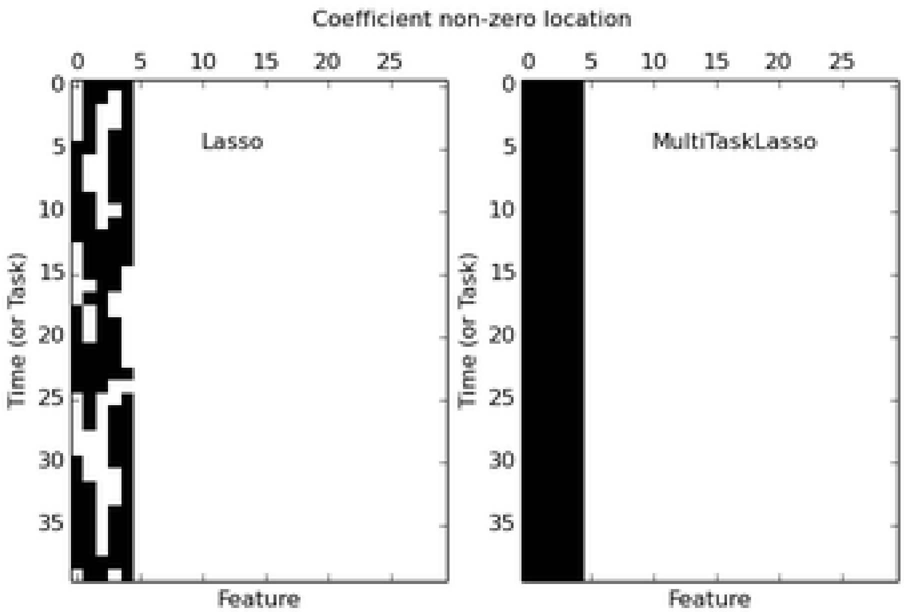
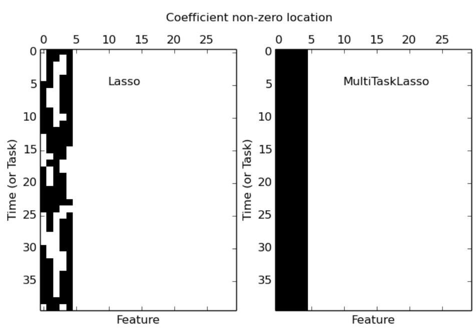
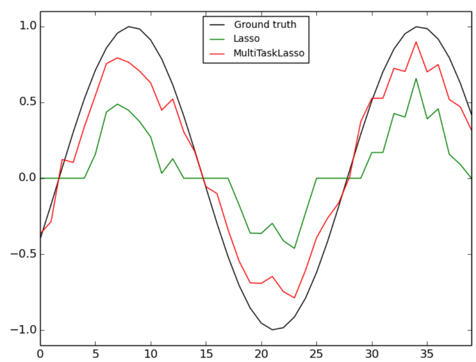


Figure 11.119: *Joint feature selection with multi-task Lasso*

- 

- 

Python source code: `plot_multi_task_lasso_support.py`

```python
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_

###############################################################################
# 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()
plt.plot(coef[:, feature_to_plot], 'k', label='Ground truth')
plt.plot(coef_lasso_[:, feature_to_plot], 'g', label='Lasso')
plt.plot(coef_multi_task_lasso_[:, feature_to_plot],
         'r', label='MultiTaskLasso')
plt.legend(loc='upper center')
plt.axis('tight')
plt.ylim([-1.1, 1.1])
plt.show()
```

Total running time of the example: 0.14 seconds (0 minutes 0.14 seconds)

11.15.15 SGD: Penalties

Plot the contours of the three penalties.

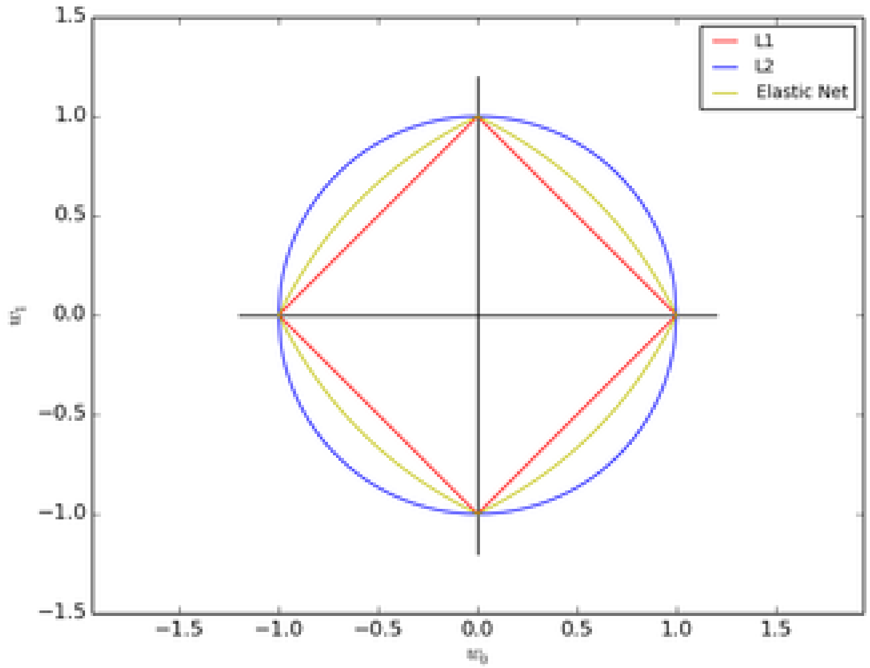
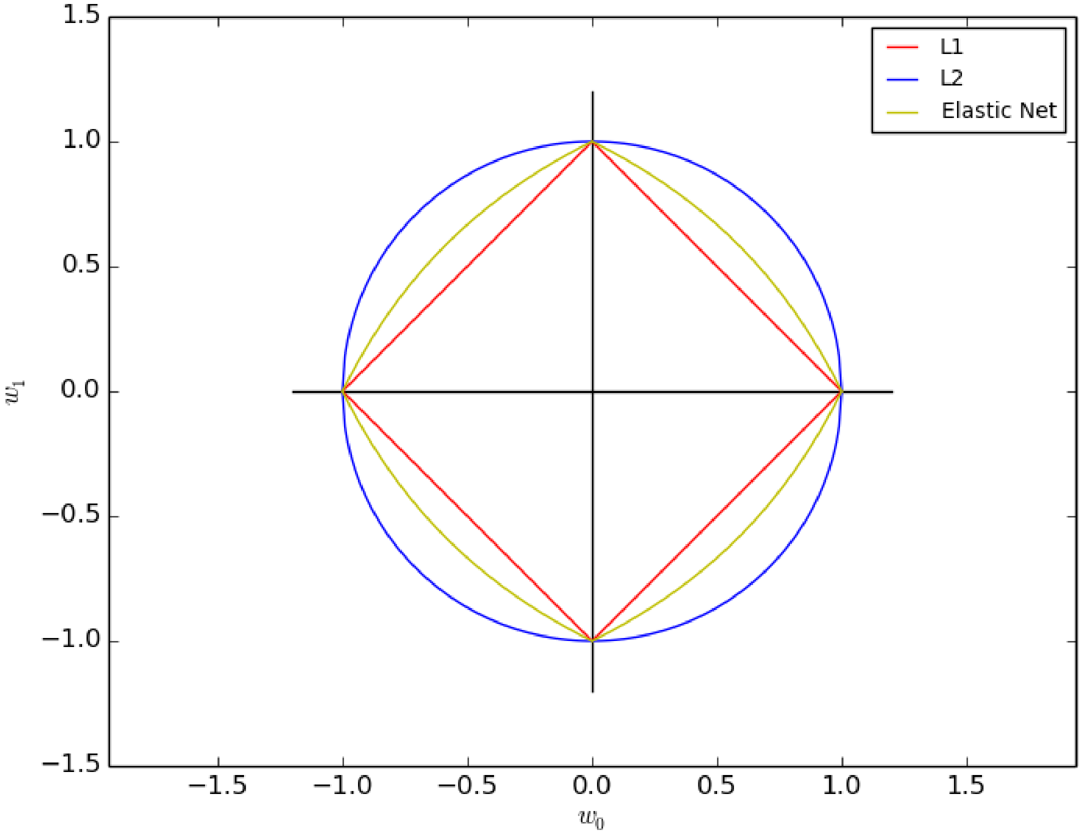


Figure 11.120: *SGD: Penalties*

All of the above are supported by `sklearn.linear_model.stochastic_gradient`.



Python source code: `plot_sgd_penalties.py`

```
from __future__ import division
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt


def l1(xs):
    return np.array([np.sqrt((1 - np.sqrt(x ** 2.0)) ** 2.0) for x in xs])


def l2(xs):
    return np.array([np.sqrt(1.0 - x ** 2.0) for x in xs])


def el(xs, z):
    return np.array([(2 - 2 * x - 2 * z + 4 * x * z -
                      (4 * z ** 2
                       - 8 * x * z ** 2
                       + 8 * x ** 2 * z ** 2
                       - 16 * x ** 2 * z ** 3
                       + 8 * x * z ** 3 + 4 * x ** 2 * z ** 4) ** (1. / 2)
                      - 2 * x * z ** 2) / (2 - 4 * z) for x in xs])
```

```
def cross(ext):
    plt.plot([-ext, ext], [0, 0], "k-")
    plt.plot([0, 0], [-ext, ext], "k-")

xs = np.linspace(0, 1, 100)

alpha = 0.501  # 0.5 division throuh zero

cross(1.2)

plt.plot(xs, l1(xs), "r-", label="L1")
plt.plot(xs, -1.0 * l1(xs), "r-")
plt.plot(-1 * xs, l1(xs), "r-")
plt.plot(-1 * xs, -1.0 * l1(xs), "r-")

plt.plot(xs, l2(xs), "b-", label="L2")
plt.plot(xs, -1.0 * l2(xs), "b-")
plt.plot(-1 * xs, l2(xs), "b-")
plt.plot(-1 * xs, -1.0 * l2(xs), "b-")

plt.plot(xs, el(xs, alpha), "y-", label="Elastic Net")
plt.plot(xs, -1.0 * el(xs, alpha), "y-")
plt.plot(-1 * xs, el(xs, alpha), "y-")
plt.plot(-1 * xs, -1.0 * el(xs, alpha), "y-")

plt.xlabel(r"$w_0$")
plt.ylabel(r"$w_1$")
plt.legend()

plt.axis("equal")
plt.show()
```

Total running time of the example: 0.10 seconds (0 minutes 0.10 seconds)

Figure 11.121: *Lasso on dense and sparse data*

11.15.16 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.

Python source code: `lasso_dense_vs_sparse_data.py`

```
print(__doc__)

from time import time
from scipy import sparse
from scipy import linalg

from sklearn.datasets.samples_generator 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_))
```

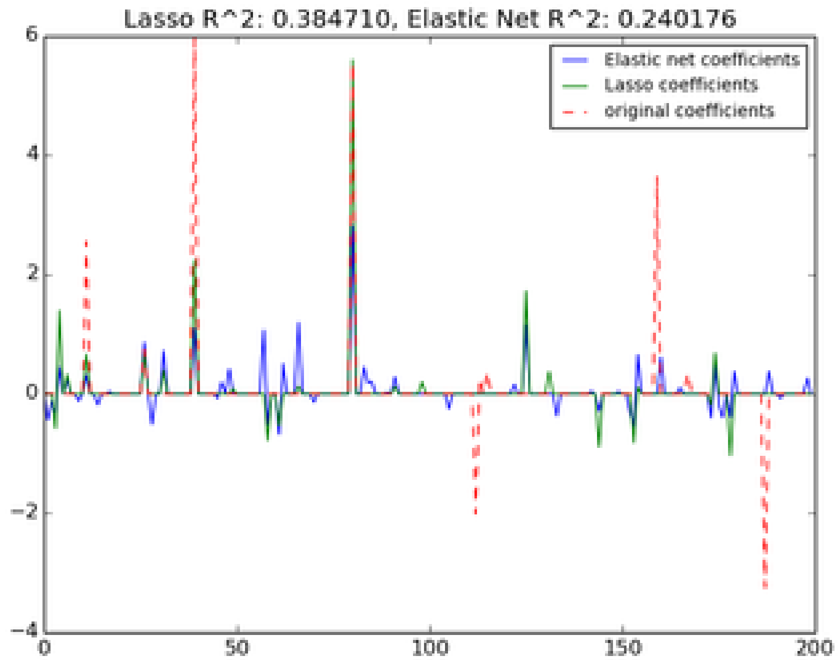
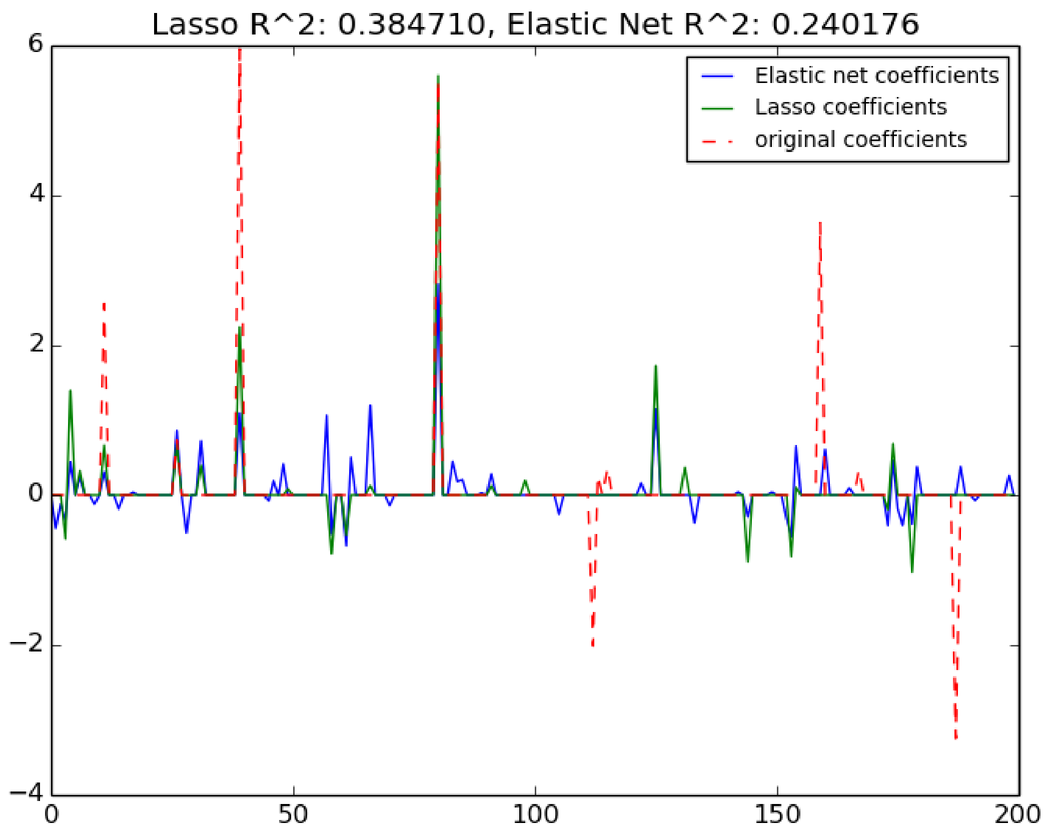


Figure 11.122: *Lasso and Elastic Net for Sparse Signals*

11.15.17 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.



Script output:

```
Lasso(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=None,
   selection='cyclic', tol=0.0001, warm_start=False)
r^2 on test data : 0.384710
ElasticNet(alpha=0.1, copy_X=True, fit_intercept=True, l1_ratio=0.7,
      max_iter=1000, normalize=False, positive=False, precompute=False,
      random_state=None, selection='cyclic', tol=0.0001, warm_start=False)
r^2 on test data : 0.240176
```

Python source code: `plot_lasso_and_elasticnet.py`

```
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, 200
X = np.random.randn(n_samples, n_features)
coef = 3 * np.random.randn(n_features)
```

```
inds = np.arange(n_features)
np.random.shuffle(inds)
coef[inds[10:]] = 0  # sparsify coef
y = np.dot(X, coef)

# add noise
y += 0.01 * np.random.normal((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)

plt.plot(enet.coef_, label='Elastic net coefficients')
plt.plot(lasso.coef_, label='Lasso coefficients')
plt.plot(coef, '--', label='original coefficients')
plt.legend(loc='best')
plt.title("Lasso R^2: %f, Elastic Net R^2: %f"
          % (r2_score_lasso, r2_score_enet))
plt.show()
```

Total running time of the example: 0.09 seconds (0 minutes 0.09 seconds)

11.15.18 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.

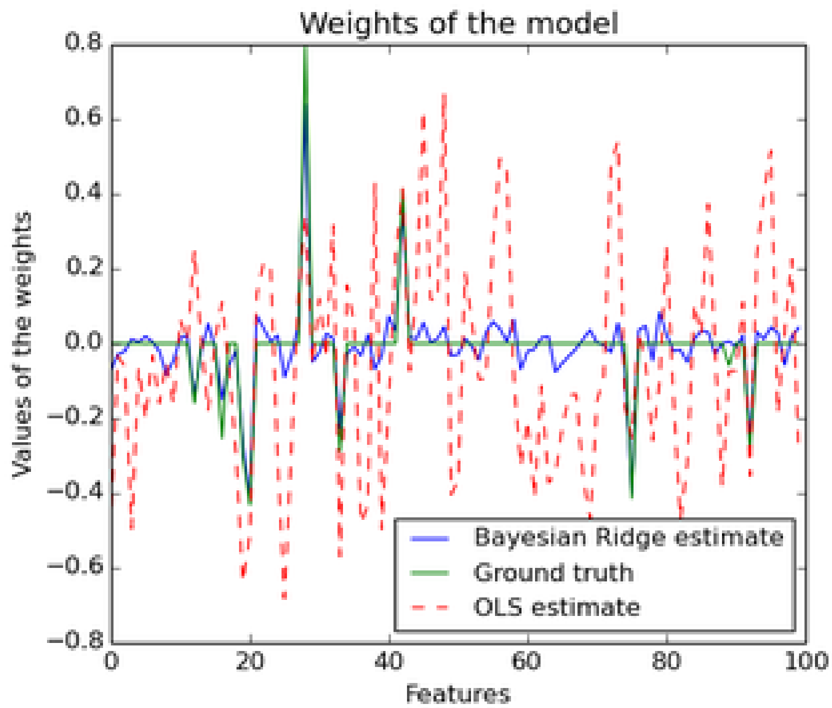
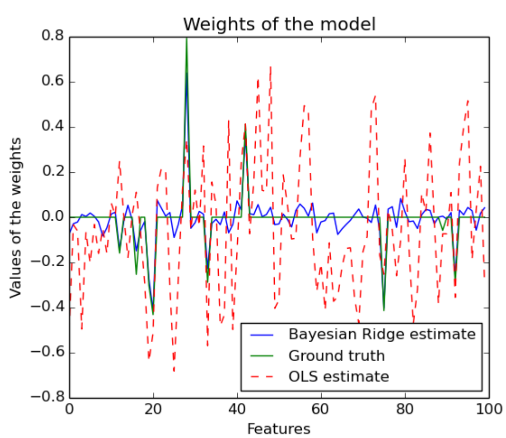
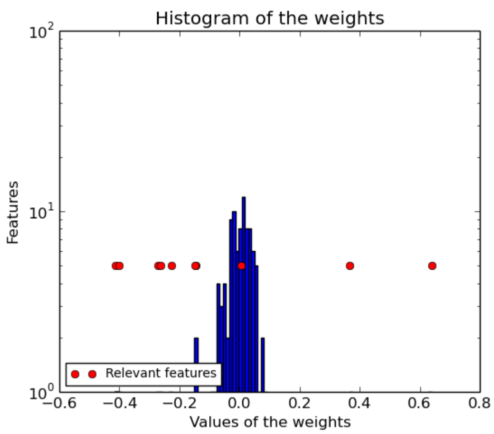
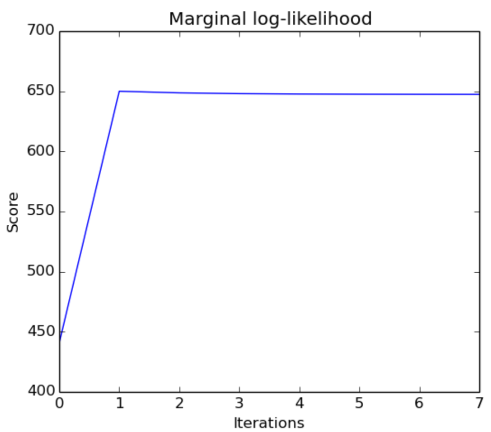


Figure 11.123: *Bayesian Ridge Regression*

- 

- 

- 

Python source code: `plot_bayesian_ridge.py`

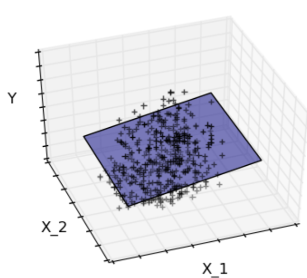
```
print(__doc__)

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 weigthts
np.random.seed(0)
n_samples, n_features = 100, 100
X = np.random.randn(n_samples, n_features)  # Create Gaussian data
# Create weigts 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
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 and histogram of the weights
plt.figure(figsize=(6, 5))
plt.title("Weights of the model")
plt.plot(clf.coef_, 'b-', label="Bayesian Ridge estimate")
plt.plot(w, 'g-', label="Ground truth")
plt.plot(ols.coef_, 'r--', 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, log=True)
plt.plot(clf.coef_[relevant_features], 5 * np.ones(len(relevant_features)),
         'ro', label="Relevant features")
plt.ylabel("Features")
plt.xlabel("Values of the weights")
plt.legend(loc="lower left")

plt.figure(figsize=(6, 5))
plt.title("Marginal log-likelihood")
plt.plot(clf.scores_)
plt.ylabel("Score")
plt.xlabel("Iterations")
plt.show()
```

Total running time of the example: 0.20 seconds (0 minutes 0.20 seconds)

11.15.19 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

- 

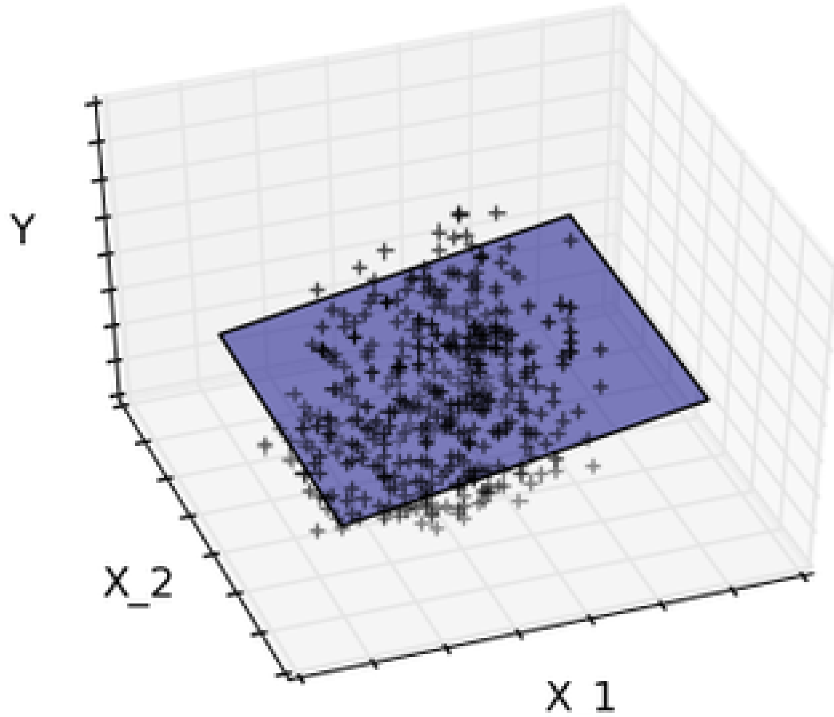
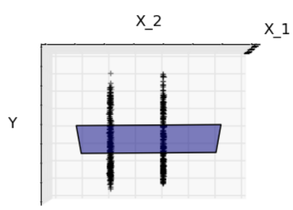
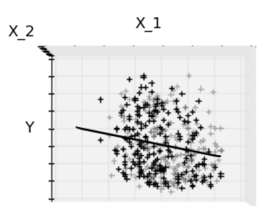


Figure 11.124: *Sparsity Example: Fitting only features 1 and 2*

- 

- 

Python source code: `plot_ols_3d.py`

```
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

diabetes = datasets.load_diabetes()
indices = (0, 1)

X_train = diabetes.data[:-20, indices]
X_test = diabetes.data[-20:, indices]
y_train = diabetes.target[:-20]
y_test = diabetes.target[-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
                                ).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 example: 0.19 seconds (0 minutes 0.19 seconds)

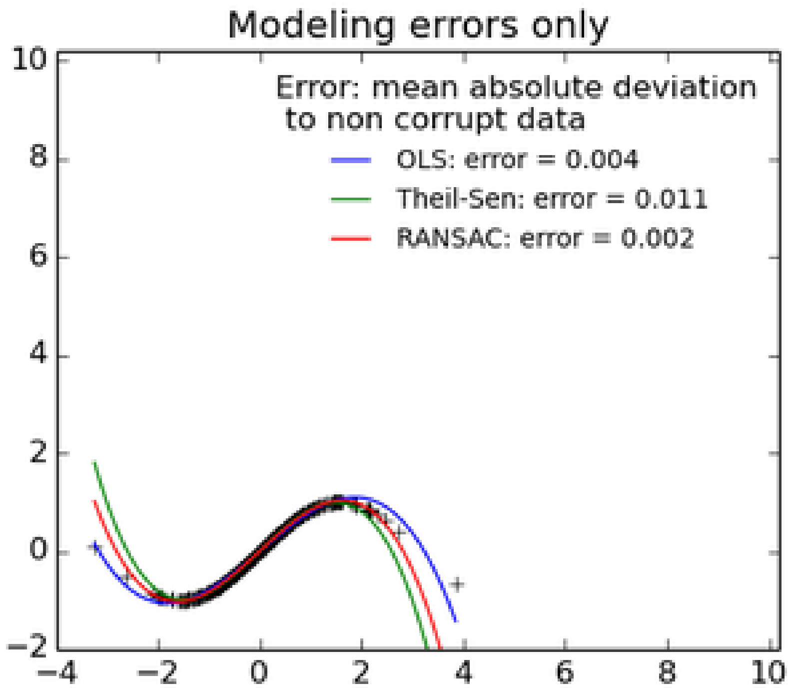


Figure 11.125: *Robust linear estimator fitting*

11.15.20 Robust linear estimator fitting

Here a sine function is fit with a polynomial of order 3, for values close to zero.

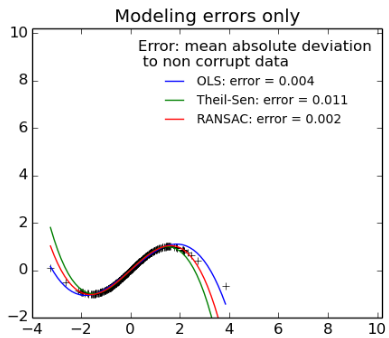
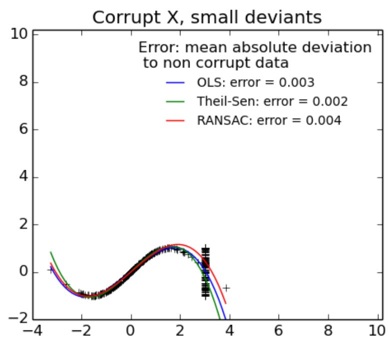
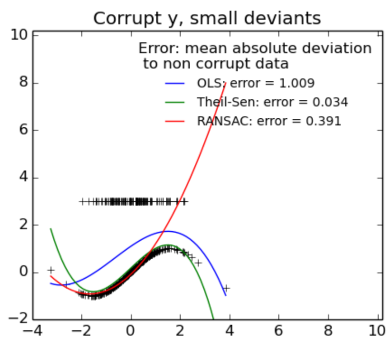
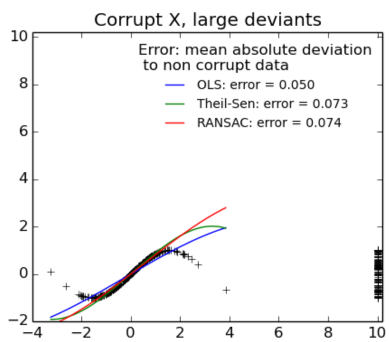
Robust fitting is demoed in different situations:

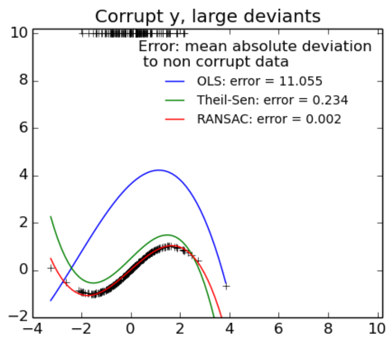
- No measurement errors, only modelling errors (fitting a sine with a polynomial)
- Measurement errors in X
- Measurement errors in y

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 worst than OLS.

- 
Modeling errors only
Error: mean absolute deviation
to non corrupt data
OLS: error = 0.004
Theil-Sen: error = 0.011
RANSAC: error = 0.002
- 
Corrupt X, small deviants
Error: mean absolute deviation
to non corrupt data
OLS: error = 0.003
Theil-Sen: error = 0.002
RANSAC: error = 0.004
- 
Corrupt y, small deviants
Error: mean absolute deviation
to non corrupt data
OLS: error = 1.009
Theil-Sen: error = 0.034
RANSAC: error = 0.391
- 
Corrupt X, large deviants
Error: mean absolute deviation
to non corrupt data
OLS: error = 0.050
Theil-Sen: error = 0.073
RANSAC: error = 0.074

- 

Python source code: `plot_robust_fit.py`

```
from matplotlib import pyplot as plt
import numpy as np

from sklearn import linear_model, metrics
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', linear_model.LinearRegression()),
              ('Theil-Sen', linear_model.TheilSenRegressor(random_state=42)),
              ('RANSAC', linear_model.RANSACRegressor(random_state=42)), ]

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, 'k+')
```

```
    for name, estimator in estimators:
        model = make_pipeline(PolynomialFeatures(3), estimator)
        model.fit(this_X, this_y)
        mse = metrics.mean_squared_error(model.predict(X_test), y_test)
        y_plot = model.predict(x_plot[:, np.newaxis])
        plt.plot(x_plot, y_plot,
                 label='%s: error = %.3f' % (name, mse))

    plt.legend(loc='best', frameon=False,
               title='Error: mean absolute deviation\n to non corrupt data')
    plt.xlim(-4, 10.2)
    plt.ylim(-2, 10.2)
    plt.title(title)
plt.show()
```

Total running time of the example: 3.59 seconds (0 minutes 3.59 seconds)

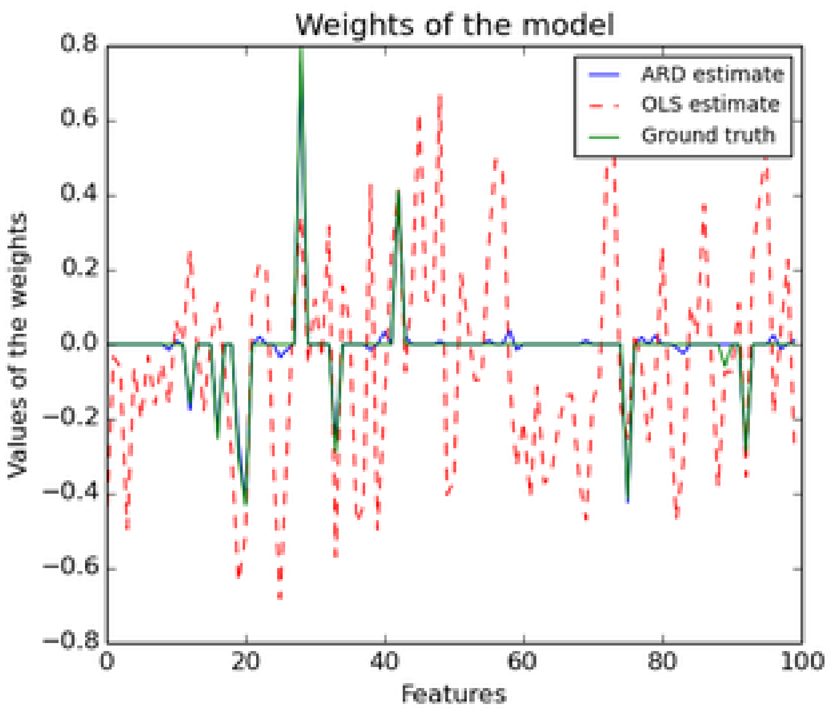


Figure 11.126: *Automatic Relevance Determination Regression (ARD)*

11.15.21 Automatic Relevance Determination Regression (ARD)

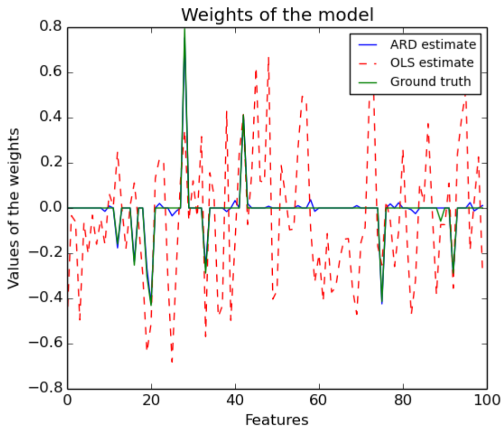
Fit regression model with Bayesian Ridge Regression.

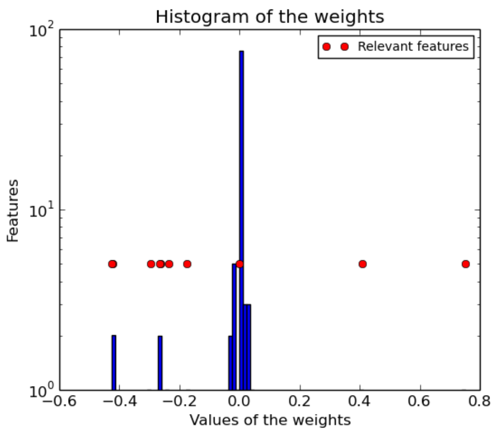
See *Bayesian Ridge Regression* for more information on the regressor.

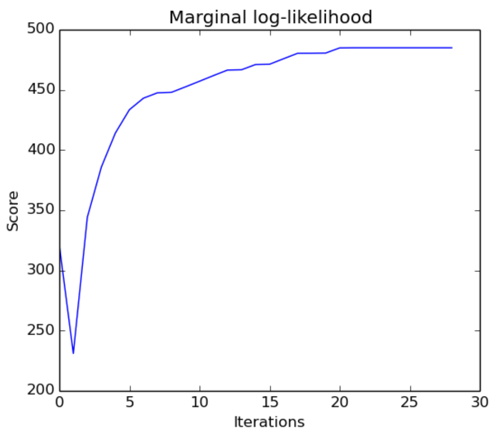
Compared to the OLS (ordinary least squares) estimator, the coefficient weights are slightly shifted toward zeros, which stabilises them.

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.

- 

- 

- 

Python source code: `plot_ard.py`

```
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 weigts 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 noite 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
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 and the histogram of the
# weights
plt.figure(figsize=(6, 5))
plt.title("Weights of the model")
plt.plot(clf.coef_, 'b-', label="ARD estimate")
plt.plot(ols.coef_, 'r--', label="OLS estimate")
plt.plot(w, 'g-', 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, log=True)
plt.plot(clf.coef_[relevant_features], 5 * np.ones(len(relevant_features)),
         'ro', 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_)
plt.ylabel("Score")
plt.xlabel("Iterations")
plt.show()
```

Total running time of the example: 0.30 seconds (0 minutes 0.30 seconds)

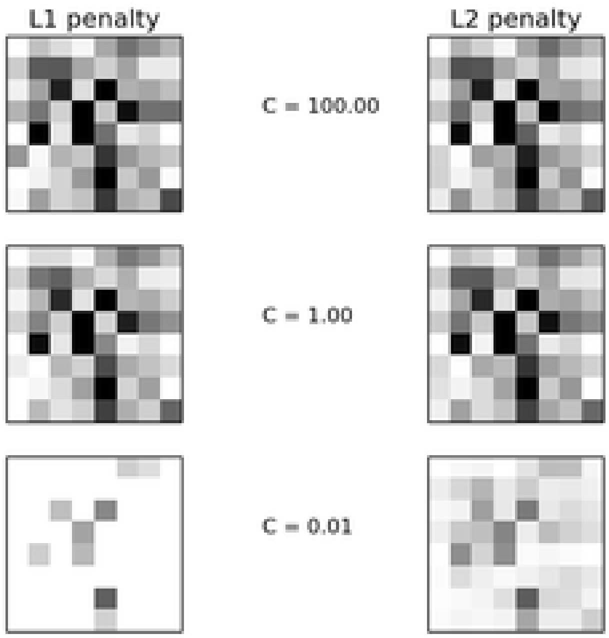
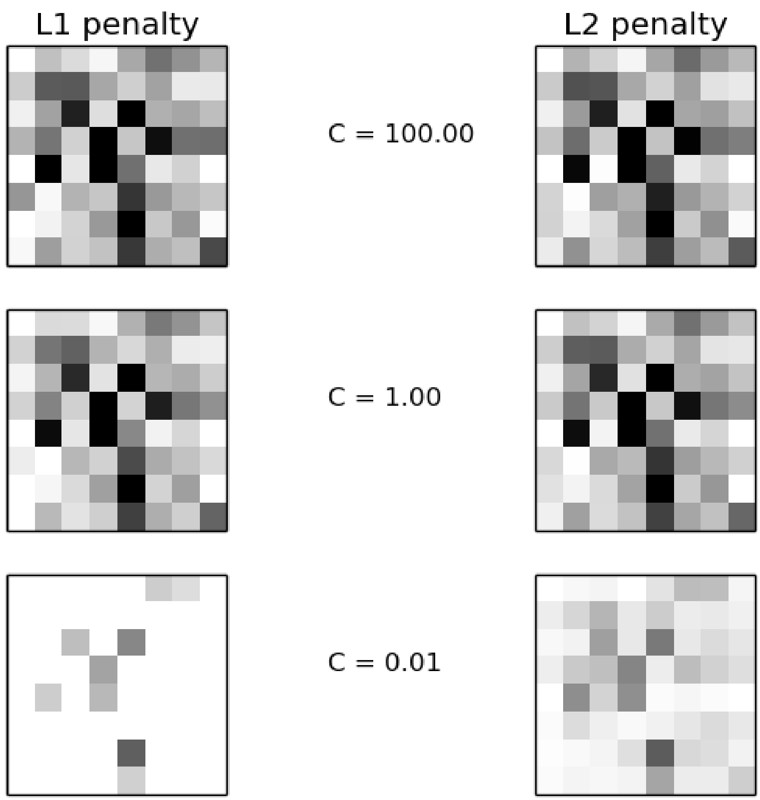


Figure 11.127: *L1 Penalty and Sparsity in Logistic Regression*

11.15.22 L1 Penalty and Sparsity in Logistic Regression

Comparison of the sparsity (percentage of zero coefficients) of solutions when L1 and L2 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.

We classify 8x8 images of digits into two classes: 0-4 against 5-9. The visualization shows coefficients of the models for varying C.



Script output:

```
C=100.00
Sparsity with L1 penalty: 6.25%
score with L1 penalty: 0.9104
Sparsity with L2 penalty: 4.69%
score with L2 penalty: 0.9098
C=1.00
Sparsity with L1 penalty: 10.94%
score with L1 penalty: 0.9098
Sparsity with L2 penalty: 4.69%
score with L2 penalty: 0.9093
C=0.01
Sparsity with L1 penalty: 85.94%
score with L1 penalty: 0.8614
Sparsity with L2 penalty: 4.69%
score with L2 penalty: 0.8915
```

Python source code: `plot_logistic_l1_l2_sparsity.py`

```python
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

digits = datasets.load_digits()

X, y = digits.data, digits.target
X = StandardScaler().fit_transform(X)

# classify small against large digits
y = (y > 4).astype(np.int)


# Set regularization parameter
for i, C in enumerate((100, 1, 0.01)):
    # turn down tolerance for short training time
    clf_l1_LR = LogisticRegression(C=C, penalty='l1', tol=0.01)
    clf_l2_LR = LogisticRegression(C=C, penalty='l2', tol=0.01)
    clf_l1_LR.fit(X, y)
    clf_l2_LR.fit(X, y)

    coef_l1_LR = clf_l1_LR.coef_.ravel()
    coef_l2_LR = clf_l2_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

    print("C=%.2f" % C)
    print("Sparsity with L1 penalty: %.2f%%" % sparsity_l1_LR)
    print("score with L1 penalty: %.4f" % clf_l1_LR.score(X, y))
    print("Sparsity with L2 penalty: %.2f%%" % sparsity_l2_LR)
    print("score with L2 penalty: %.4f" % clf_l2_LR.score(X, y))

    l1_plot = plt.subplot(3, 2, 2 * i + 1)
    l2_plot = plt.subplot(3, 2, 2 * (i + 1))
    if i == 0:
        l1_plot.set_title("L1 penalty")
        l2_plot.set_title("L2 penalty")

    l1_plot.imshow(np.abs(coef_l1_LR.reshape(8, 8)), interpolation='nearest',
                   cmap='binary', vmax=1, vmin=0)
    l2_plot.imshow(np.abs(coef_l2_LR.reshape(8, 8)), interpolation='nearest',
                   cmap='binary', vmax=1, vmin=0)
    plt.text(-8, 3, "C = %.2f" % C)

    l1_plot.set_xticks(())
    l1_plot.set_yticks(())
    l2_plot.set_xticks(())
    l2_plot.set_yticks(())

plt.show()
```

Total running time of the example: 0.32 seconds (0 minutes 0.32 seconds)

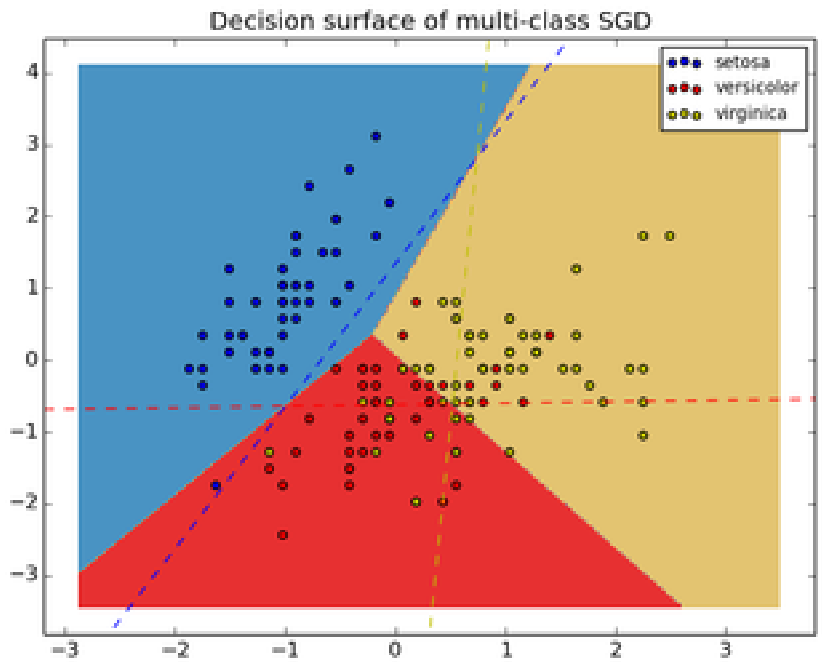
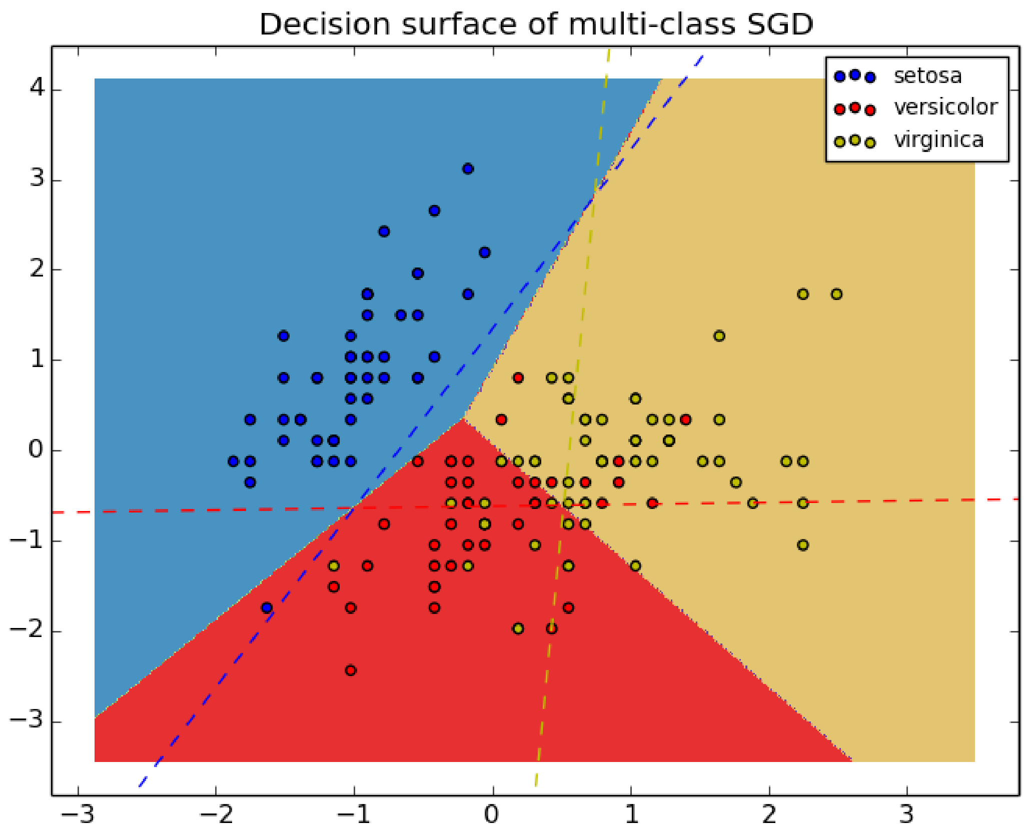


Figure 11.128: *Plot multi-class SGD on the iris dataset*

11.15.23 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.



Python source code: `plot_sgd_iris.py`

```
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()
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
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)
std = X.std(axis=0)
```

```
X = (X - mean) / std

h = .02  # step size in the mesh

clf = SGDClassifier(alpha=0.001, n_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, m_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)
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 example: 0.12 seconds (0 minutes 0.12 seconds)

11.15.24 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.

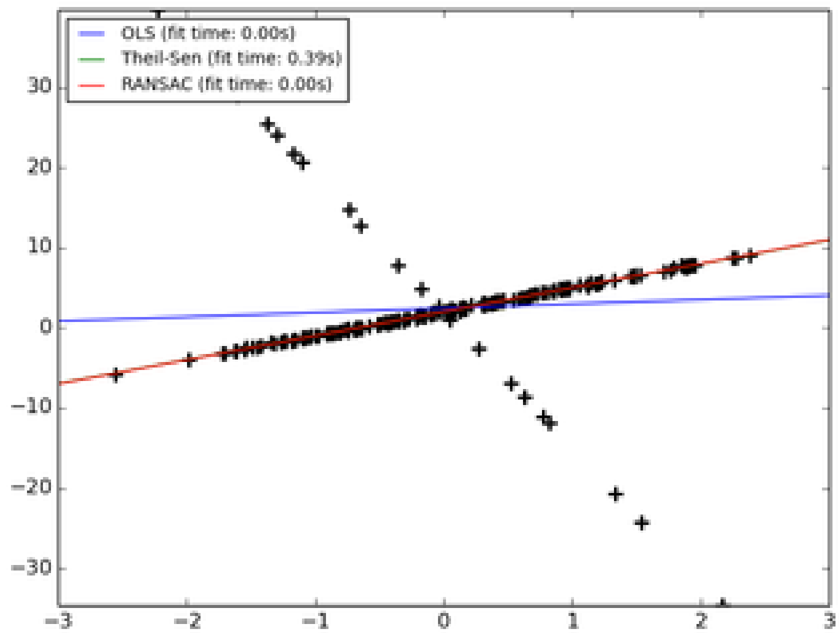
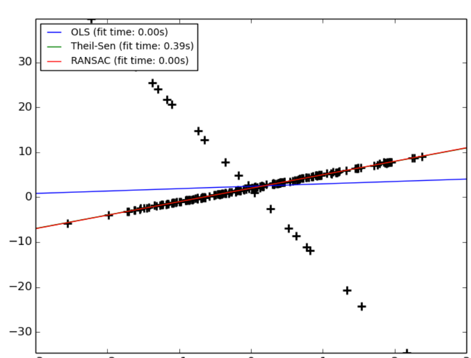
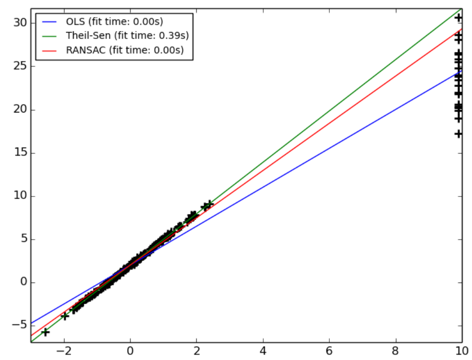


Figure 11.129: *Theil-Sen Regression*

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.

- 

- 

Python source code: `plot_theilsen.py`

```
# 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)), ]

##############################################################################
# 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)
y = w * x + c + noise
# 10% outliers
y[-20:] += -20 * x[-20:]
X = x[:, np.newaxis]

plt.plot(x, y, 'k+', mew=2, ms=8)
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,
             label='%s (fit time: %.2fs)' % (name, elapsed_time))

plt.axis('tight')
plt.legend(loc='upper left')
```

```
###############################################################################
# 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.plot(x, y, 'k+', mew=2, ms=8)

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,
             label='%s (fit time: %.2fs)' % (name, elapsed_time))

plt.axis('tight')
plt.legend(loc='upper left')
plt.show()
```

Total running time of the example: 0.90 seconds (0 minutes 0.90 seconds)

11.15.25 Orthogonal Matching Pursuit

Using orthogonal matching pursuit for recovering a sparse signal from a noisy measurement encoded with a dictionary

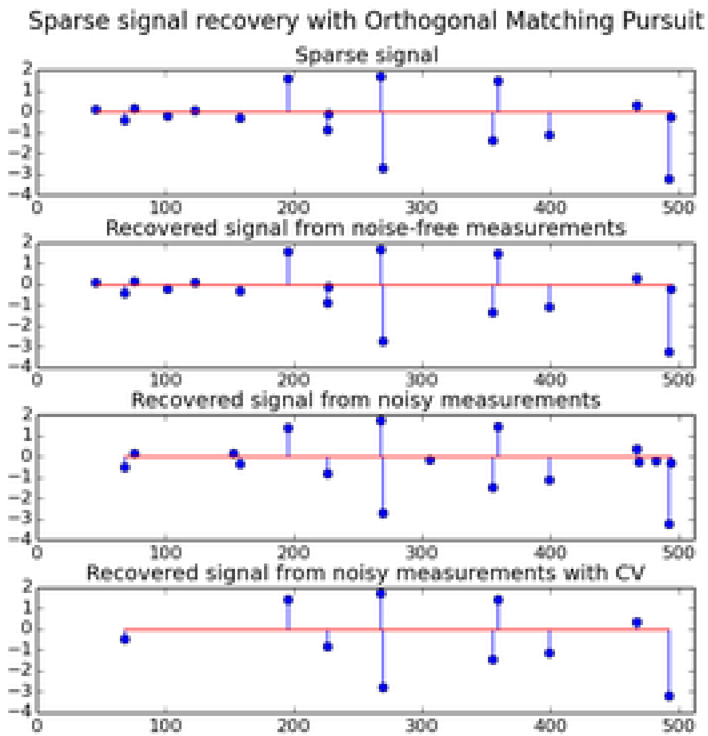
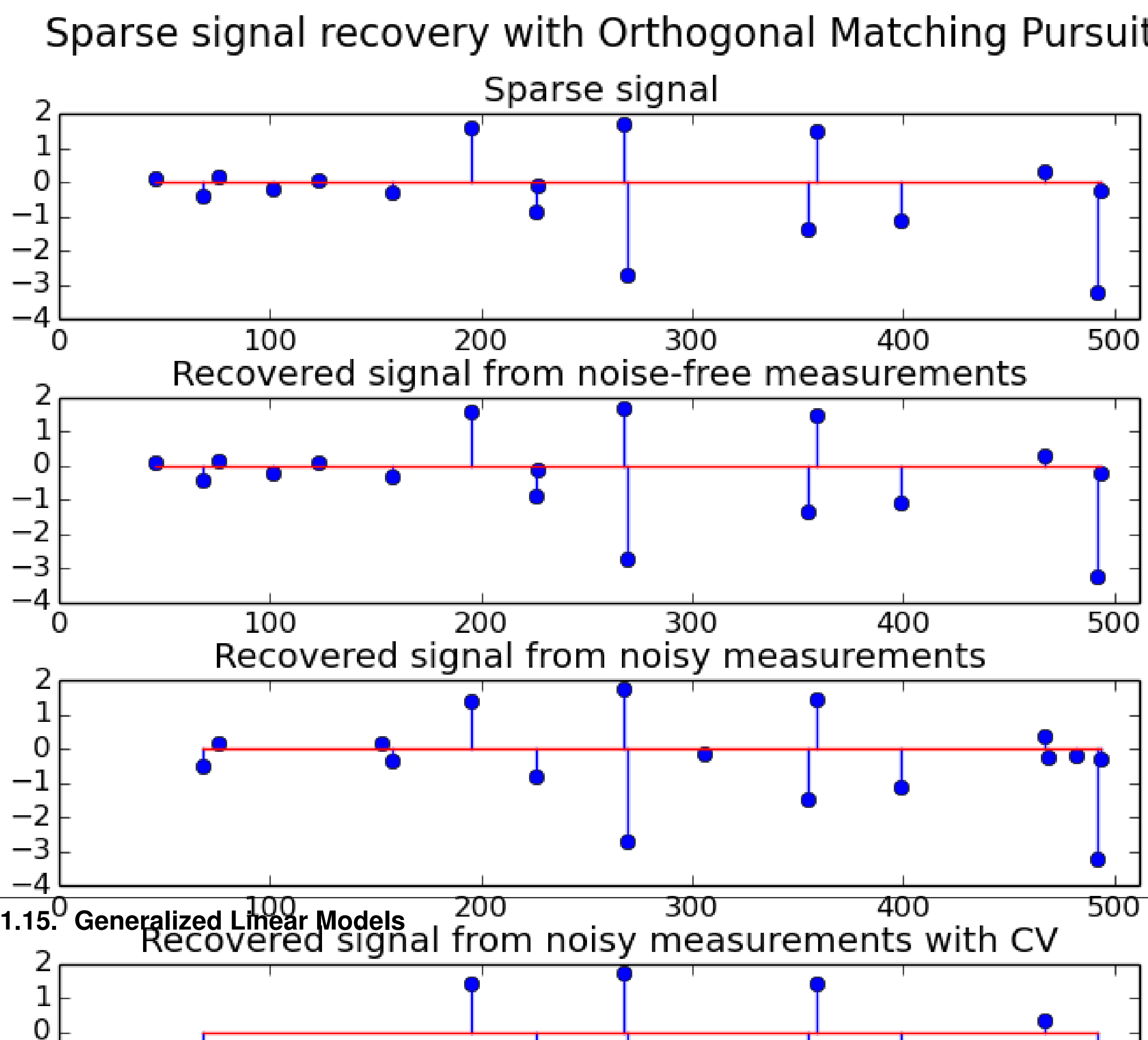


Figure 11.130: *Orthogonal Matching Pursuit*



Python source code: plot_omp.py

```
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

# 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])

# 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])

# 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])

# 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])

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 example: 0.22 seconds (0 minutes 0.22 seconds)

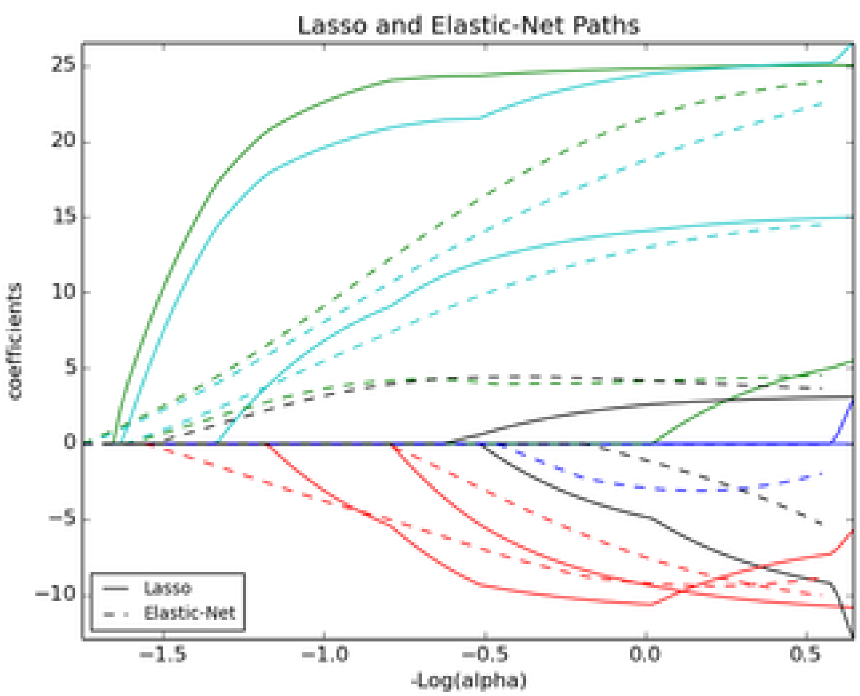
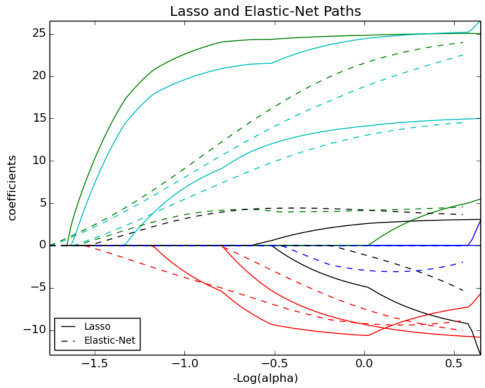


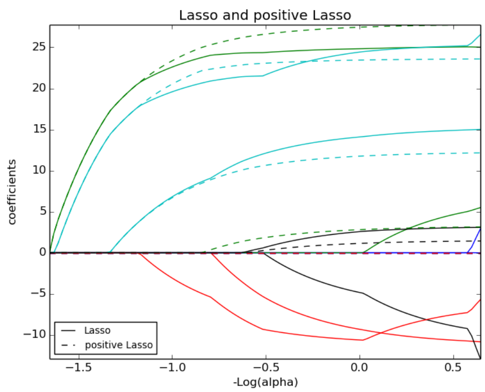
Figure 11.131: *Lasso and Elastic Net*

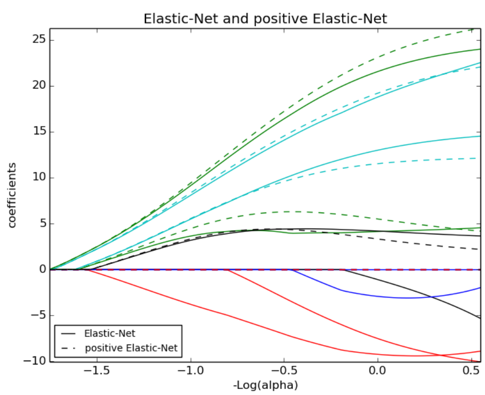
11.15.26 Lasso and Elastic Net

Lasso and elastic net (L1 and L2 penalisation) implemented using a coordinate descent.

The coefficients can be forced to be positive.

- 

- 

- 

Script output:

```
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 positve elastic net...
```

Python source code: `plot_lasso_coordinate_descent_path.py`

```
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause
```

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import lasso_path, enet_path
from sklearn import datasets

diabetes = datasets.load_diabetes()
X = diabetes.data
y = diabetes.target

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 positve 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)
ax = plt.gca()
ax.set_color_cycle(2 * ['b', 'r', 'g', 'c', 'k'])
l1 = plt.plot(-np.log10(alphas_lasso), coefs_lasso.T)
l2 = plt.plot(-np.log10(alphas_enet), coefs_enet.T, linestyle='--')

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)
ax = plt.gca()
ax.set_color_cycle(2 * ['b', 'r', 'g', 'c', 'k'])
l1 = plt.plot(-np.log10(alphas_lasso), coefs_lasso.T)
l2 = plt.plot(-np.log10(alphas_positive_lasso), coefs_positive_lasso.T,
              linestyle='--')

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)
ax = plt.gca()
ax.set_color_cycle(2 * ['b', 'r', 'g', 'c', 'k'])
l1 = plt.plot(-np.log10(alphas_enet), coefs_enet.T)
l2 = plt.plot(-np.log10(alphas_positive_enet), coefs_positive_enet.T,
              linestyle='--')

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 example: 0.18 seconds (0 minutes 0.18 seconds)

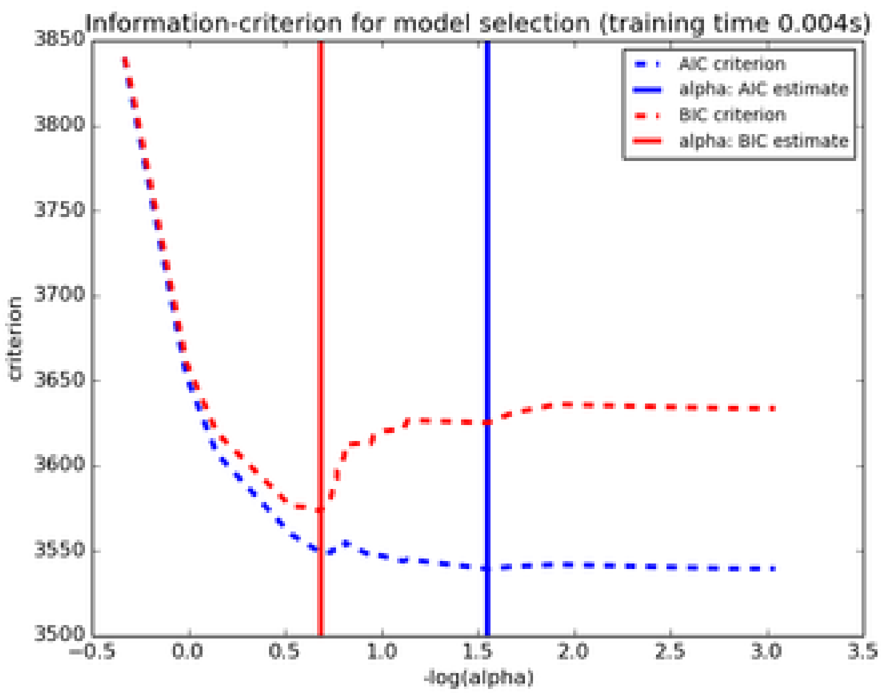


Figure 11.132: *Lasso model selection: Cross-Validation / AIC / BIC*

11.15.27 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.

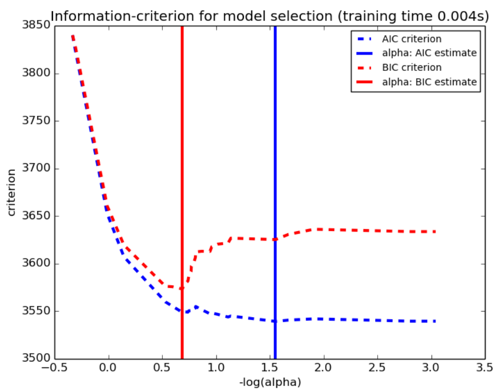
Results obtained with LassoLarsIC are based on AIC/BIC criteria.

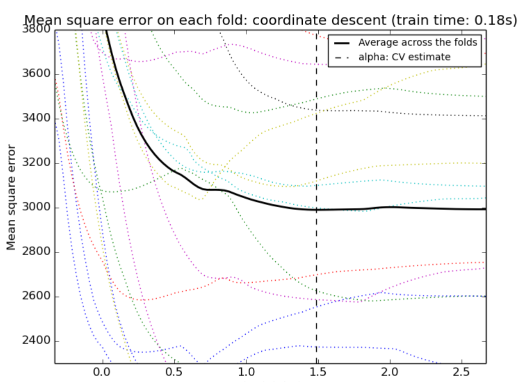
Information-criterion based model selection is very fast, but it relies on a proper estimation of degrees of freedom, are derived for large samples (asymptotic results) and assume the model is correct, i.e. that the data are actually generated by this model. They also tend to break when the problem is badly conditioned (more features than samples).

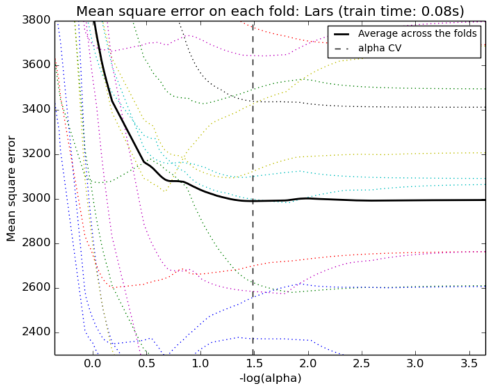
For cross-validation, we use 20-fold with 2 algorithms to compute the Lasso path: coordinate descent, as implemented by the LassoCV class, and Lars (least angle regression) as implemented by the LassoLarsCV class. Both algorithms give roughly the same results. They differ with regards to their execution 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.

- 

- 

- 

Script output:

```
Computing regularization path using the coordinate descent lasso...
Computing regularization path using the Lars lasso...
```

Python source code: `plot_lasso_model_selection.py`

```
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

diabetes = datasets.load_diabetes()
X = diabetes.data
y = diabetes.target

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_
    alphas_ = model.alphas_
    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_)

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)
plt.axvline(-np.log10(model.alpha_), 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_)
```

```
plt.figure()
plt.plot(m_log_alphas, model.cv_mse_path_, ':')
plt.plot(m_log_alphas, model.cv_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 example: 0.45 seconds (0 minutes 0.45 seconds)

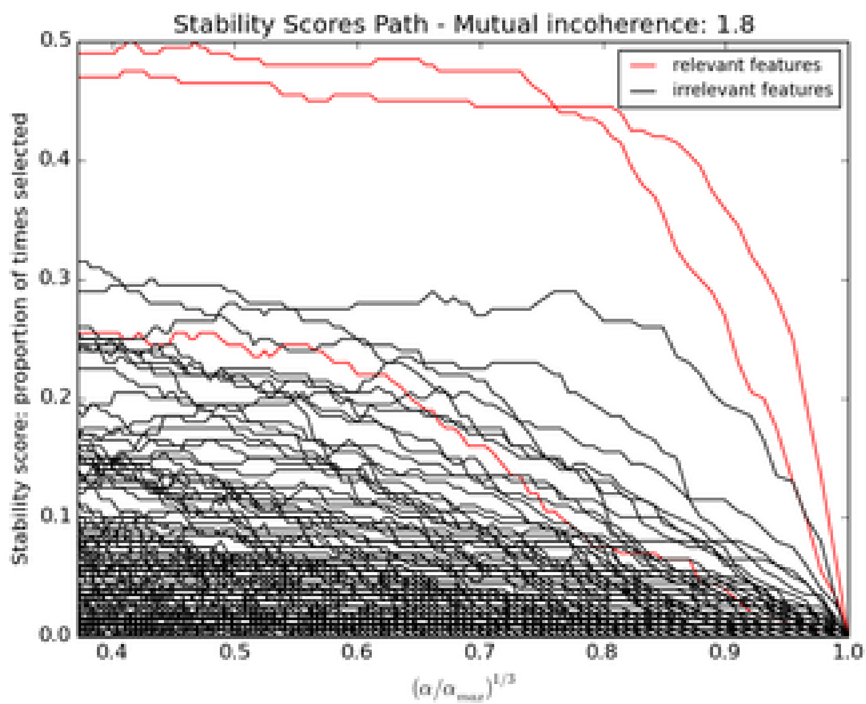


Figure 11.133: *Sparse recovery: feature selection for sparse linear models*

11.15.28 Sparse recovery: feature selection for sparse linear models

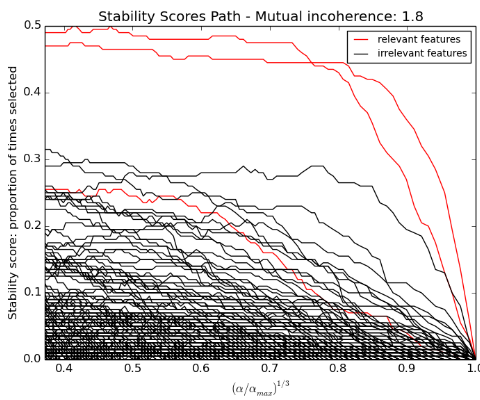
Given a small number of observations, we want to recover which features of X are relevant to explain y. For this *sparse linear models* can outperform standard statistical tests if the true model is sparse, i.e. if a small fraction of the features are relevant.

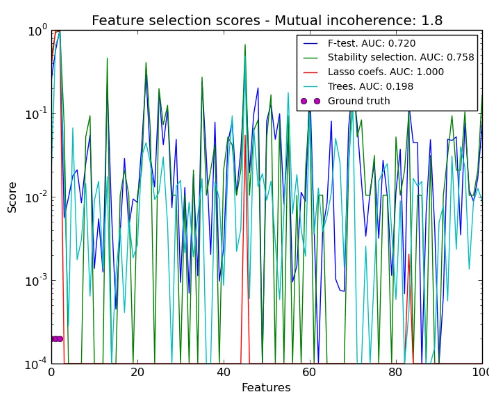
As detailed in *the compressive sensing notes*, the ability of L1-based approach to identify the relevant variables depends on the sparsity of the ground truth, the number of samples, the number of features, the conditioning of the design matrix on the signal subspace, the amount of noise, and the absolute value of the smallest non-zero coefficient [Wainwright2006] (http://statistics.berkeley.edu/tech-reports/709.pdf).

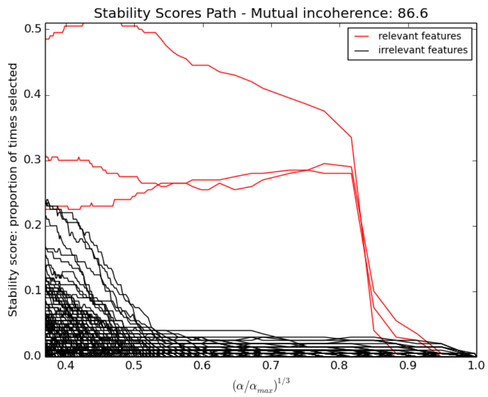
Here we keep all parameters constant and vary the conditioning of the design matrix. For a well-conditioned design matrix (small mutual incoherence) we are exactly in compressive sensing conditions (i.i.d Gaussian sensing matrix), and L1-recovery with the Lasso performs very well. For an ill-conditioned matrix (high mutual incoherence), regressors are very correlated, and the Lasso randomly selects one. However, randomized-Lasso can recover the ground truth well.

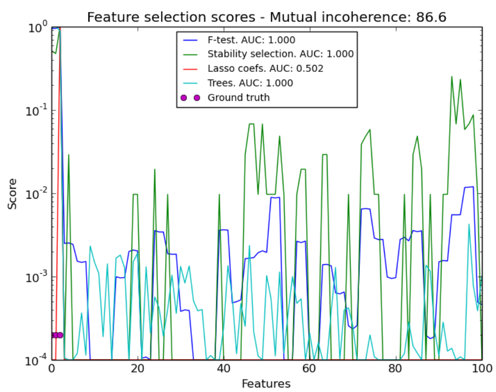
In each situation, we first vary the alpha parameter setting the sparsity of the estimated model and look at the stability scores of the randomized Lasso. This analysis, knowing the ground truth, shows an optimal regime in which relevant features stand out from the irrelevant ones. If alpha is chosen too small, non-relevant variables enter the model. On the opposite, if alpha is selected too large, the Lasso is equivalent to stepwise regression, and thus brings no advantage over a univariate F-test.

In a second time, we set alpha and compare the performance of different feature selection methods, using the area under curve (AUC) of the precision-recall.

- 

- 

- 

- 

Python source code: `plot_sparse_recovery.py`

```
print(__doc__)

# Author: Alexandre Gramfort and Gael Varoquaux
# License: BSD 3 clause

import warnings

import matplotlib.pyplot as plt
import numpy as np
from scipy import linalg

from sklearn.linear_model import (RandomizedLasso, lasso_stability_path,
                                  LassoLarsCV)
from sklearn.feature_selection import f_regression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import auc, precision_recall_curve
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.utils.extmath import pinvh
from sklearn.utils import ConvergenceWarning


def mutual_incoherence(X_relevant, X_irelevant):
    """Mutual incoherence, as defined by formula (26a) of [Wainwright2006].
    """
    projector = np.dot(np.dot(X_irelevant.T, X_relevant),
                       pinvh(np.dot(X_relevant.T, X_relevant)))
    return np.max(np.abs(projector).sum(axis=1))
```

```python
for conditioning in (1, 1e-4):
    ###########################################################################
    # Simulate regression data with a correlated design
    n_features = 501
    n_relevant_features = 3
    noise_level = .2
    coef_min = .2
    # The Donoho-Tanner phase transition is around n_samples=25: below we
    # will completely fail to recover in the well-conditioned case
    n_samples = 25
    block_size = n_relevant_features

    rng = np.random.RandomState(42)

    # The coefficients of our model
    coef = np.zeros(n_features)
    coef[:n_relevant_features] = coef_min + rng.rand(n_relevant_features)

    # The correlation of our design: variables correlated by blocs of 3
    corr = np.zeros((n_features, n_features))
    for i in range(0, n_features, block_size):
        corr[i:i + block_size, i:i + block_size] = 1 - conditioning
    corr.flat[::n_features + 1] = 1
    corr = linalg.cholesky(corr)

    # Our design
    X = rng.normal(size=(n_samples, n_features))
    X = np.dot(X, corr)
    # Keep [Wainwright2006] (26c) constant
    X[:n_relevant_features] /= np.abs(
        linalg.svdvals(X[:n_relevant_features])).max()
    X = StandardScaler().fit_transform(X.copy())

    # The output variable
    y = np.dot(X, coef)
    y /= np.std(y)
    # We scale the added noise as a function of the average correlation
    # between the design and the output variable
    y += noise_level * rng.normal(size=n_samples)
    mi = mutual_incoherence(X[:, :n_relevant_features],
                            X[:, n_relevant_features:])

    ###########################################################################
    # Plot stability selection path, using a high eps for early stopping
    # of the path, to save computation time
    alpha_grid, scores_path = lasso_stability_path(X, y, random_state=42,
                                                   eps=0.05)

    plt.figure()
    # We plot the path as a function of alpha/alpha_max to the power 1/3: the
    # power 1/3 scales the path less brutally than the log, and enables to
    # see the progression along the path
    hg = plt.plot(alpha_grid[1:] ** .333, scores_path[coef != 0].T[1:], 'r')
    hb = plt.plot(alpha_grid[1:] ** .333, scores_path[coef == 0].T[1:], 'k')
    ymin, ymax = plt.ylim()
    plt.xlabel(r'$(\alpha / \alpha_{max})^{1/3}$')
    plt.ylabel('Stability score: proportion of times selected')
```

```
    plt.title('Stability Scores Path - Mutual incoherence: %.1f' % mi)
    plt.axis('tight')
    plt.legend((hg[0], hb[0]), ('relevant features', 'irrelevant features'),
               loc='best')

    ###########################################################################
    # Plot the estimated stability scores for a given alpha

    # Use 6-fold cross-validation rather than the default 3-fold: it leads to
    # a better choice of alpha:
    # Stop the user warnings outputs- they are not necessary for the example
    # as it is specifically set up to be challenging.
    with warnings.catch_warnings():
        warnings.simplefilter('ignore', UserWarning)
        warnings.simplefilter('ignore', ConvergenceWarning)
        lars_cv = LassoLarsCV(cv=6).fit(X, y)

    # Run the RandomizedLasso: we use a paths going down to .1*alpha_max
    # to avoid exploring the regime in which very noisy variables enter
    # the model
    alphas = np.linspace(lars_cv.alphas_[0], .1 * lars_cv.alphas_[0], 6)
    clf = RandomizedLasso(alpha=alphas, random_state=42).fit(X, y)
    trees = ExtraTreesRegressor(100).fit(X, y)
    # Compare with F-score
    F, _ = f_regression(X, y)

    plt.figure()
    for name, score in [('F-test', F),
                        ('Stability selection', clf.scores_),
                        ('Lasso coefs', np.abs(lars_cv.coef_)),
                        ('Trees', trees.feature_importances_),
                        ]:
        precision, recall, thresholds = precision_recall_curve(coef != 0,
                                                               score)
        plt.semilogy(np.maximum(score / np.max(score), 1e-4),
                     label="%s. AUC: %.3f" % (name, auc(recall, precision)))

    plt.plot(np.where(coef != 0)[0], [2e-4] * n_relevant_features, 'mo',
             label="Ground truth")
    plt.xlabel("Features")
    plt.ylabel("Score")
    # Plot only the 100 first coefficients
    plt.xlim(0, 100)
    plt.legend(loc='best')
    plt.title('Feature selection scores - Mutual incoherence: %.1f'
              % mi)

plt.show()
```

Total running time of the example: 3.48 seconds (0 minutes 3.48 seconds)

11.16 Manifold learning

Examples concerning the `sklearn.manifold` module.

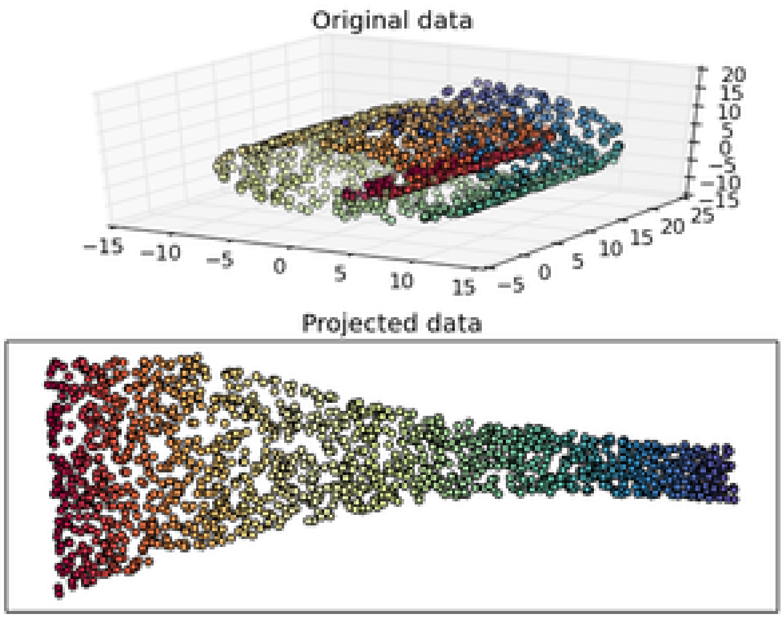
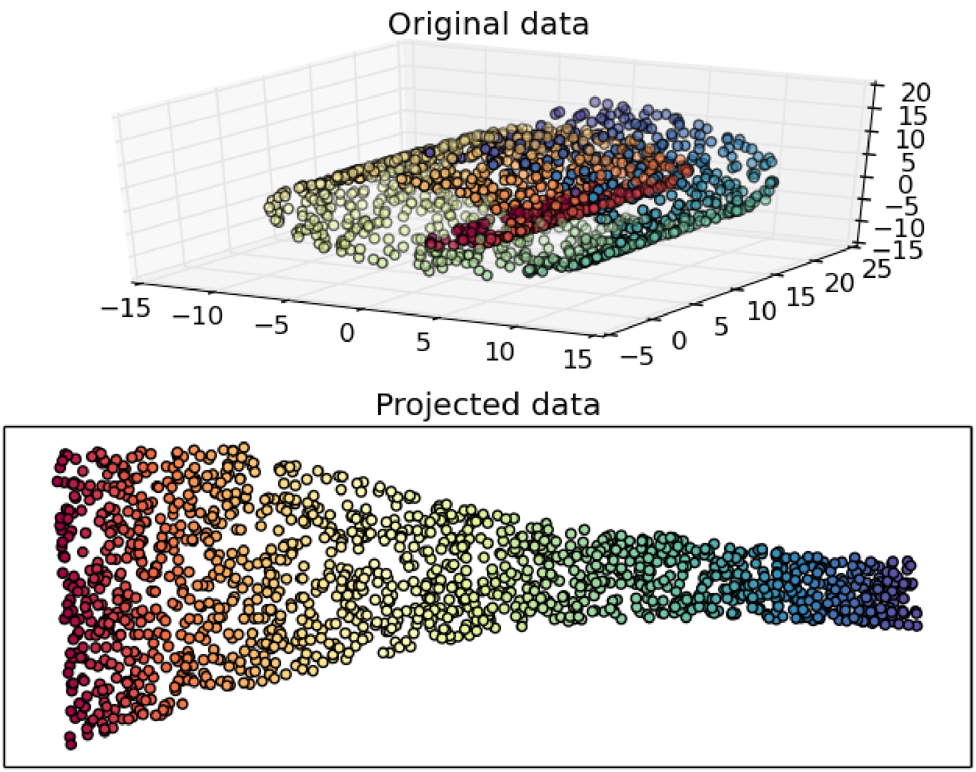


Figure 11.134: *Swiss Roll reduction with LLE*

11.16.1 Swiss Roll reduction with LLE

An illustration of Swiss Roll reduction with locally linear embedding



Script output:

```
Computing LLE embedding
Done. Reconstruction error: 9.9806e-08
```

Python source code: `plot_swissroll.py`

```python
# 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.samples_generator.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()
try:
    # compatibility matplotlib < 1.0
    ax = fig.add_subplot(211, projection='3d')
    ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral)
except:
    ax = fig.add_subplot(211)
    ax.scatter(X[:, 0], 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 example: 0.28 seconds (0 minutes 0.28 seconds)

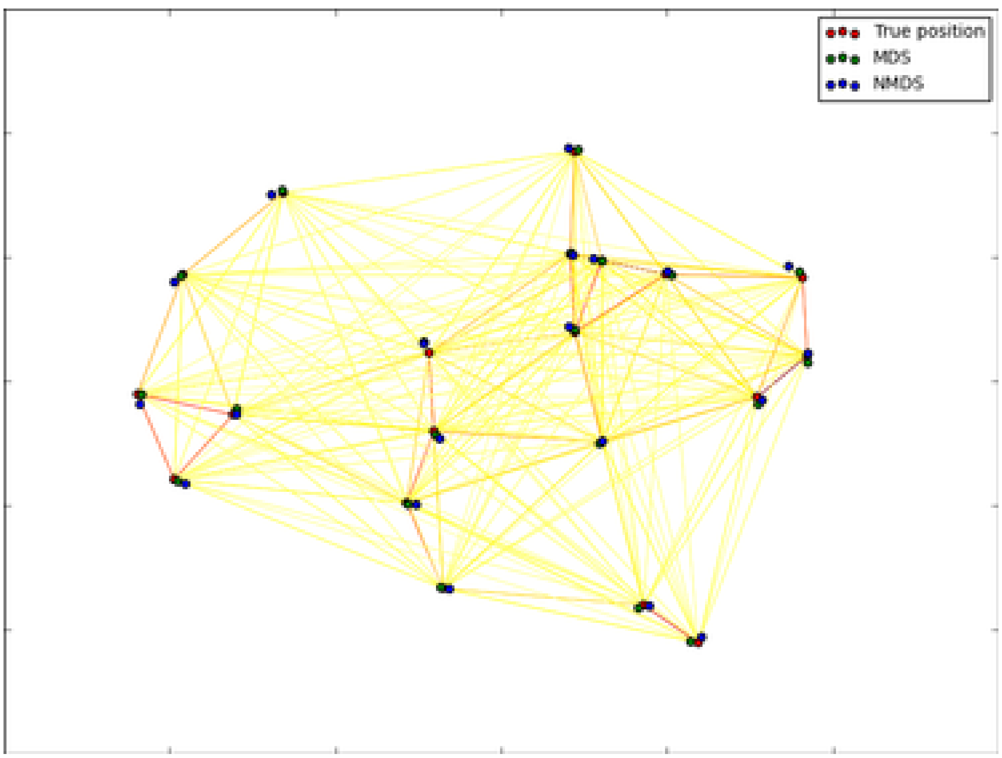
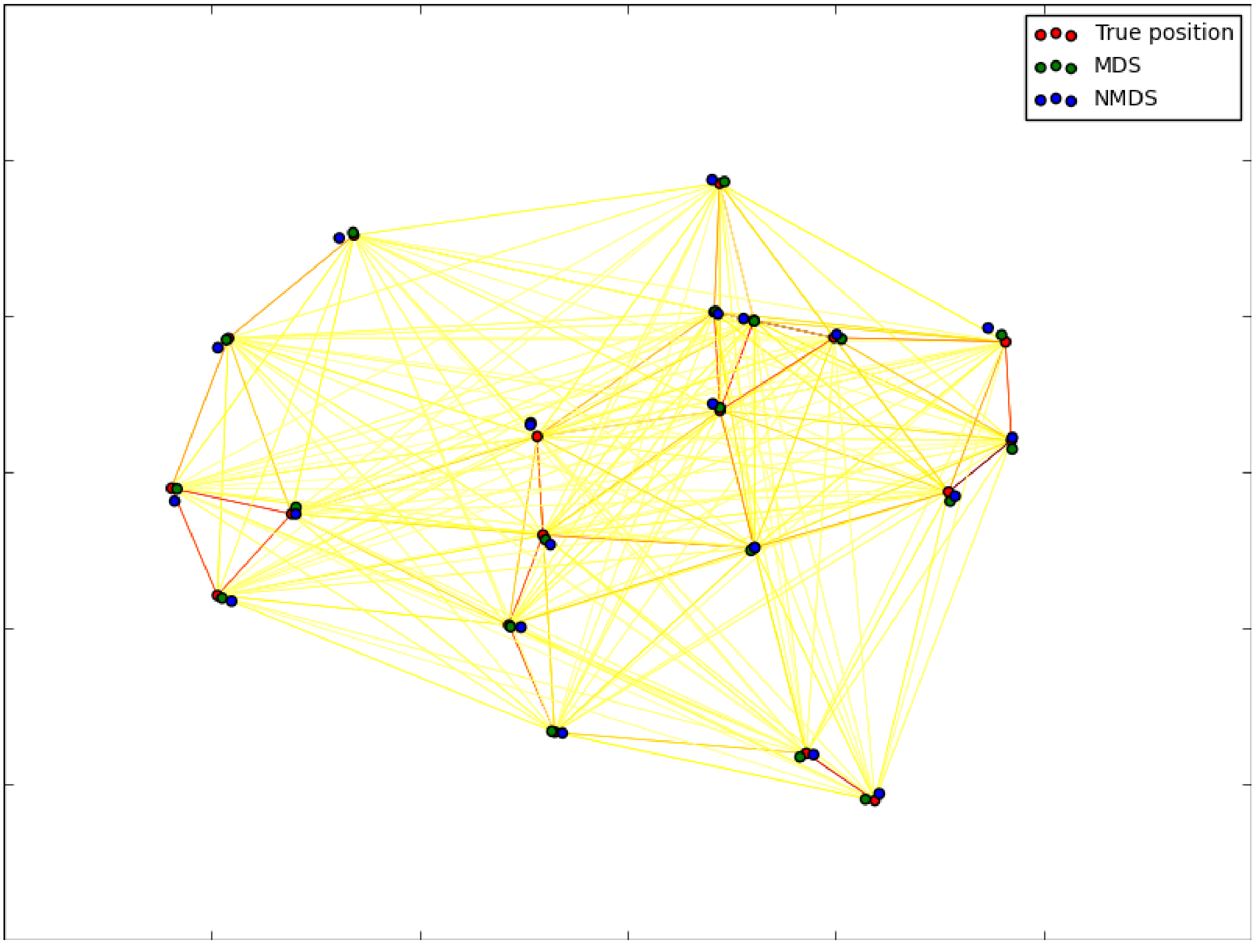


Figure 11.135: *Multi-dimensional scaling*

11.16.2 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.



Python source code: `plot_mds.py`

```
# Author: Nelle Varoquaux <nelle.varoquaux@gmail.com>
# Licence: 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

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
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.])

plt.scatter(X_true[:, 0], X_true[:, 1], c='r', s=20)
plt.scatter(pos[:, 0], pos[:, 1], s=20, c='g')
plt.scatter(npos[:, 0], npos[:, 1], s=20, c='b')
plt.legend(('True position', 'MDS', 'NMDS'), loc='best')

similarities = similarities.max() / similarities * 100
similarities[np.isinf(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.hot_r,
                    norm=plt.Normalize(0, values.max()))
lc.set_array(similarities.flatten())
lc.set_linewidths(0.5 * np.ones(len(segments)))
ax.add_collection(lc)

plt.show()
```

Total running time of the example: 0.11 seconds (0 minutes 0.11 seconds)

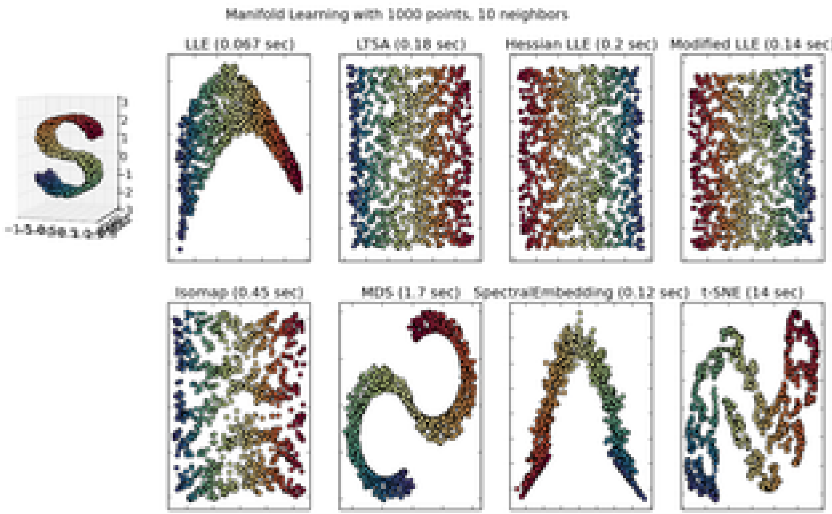


Figure 11.136: *Comparison of Manifold Learning methods*

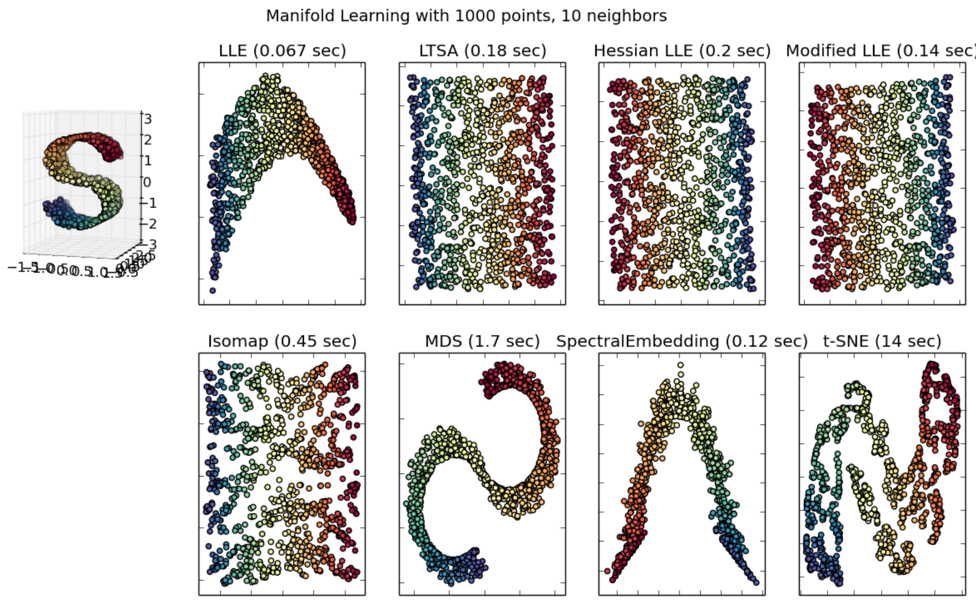
11.16.3 Comparison of Manifold Learning methods

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.



Script output:

```
standard: 0.067 sec
ltsa: 0.18 sec
hessian: 0.2 sec
modified: 0.14 sec
Isomap: 0.45 sec
MDS: 1.7 sec
SpectralEmbedding: 0.12 sec
t-SNE: 14 sec
```

Python source code: `plot_compare_methods.py`

```python
# Author: Jake Vanderplas -- <vanderplas@astro.washington.edu>

print(__doc__)

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
X, color = datasets.samples_generator.make_s_curve(n_points, random_state=0)
n_neighbors = 10
n_components = 2

fig = plt.figure(figsize=(15, 8))
plt.suptitle("Manifold Learning with %i points, %i neighbors"
             % (1000, n_neighbors), fontsize=14)
```

```
try:
    # compatibility matplotlib < 1.0
    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)
except:
    ax = fig.add_subplot(251, projection='3d')
    plt.scatter(X[:, 0], X[:, 2], c=color, cmap=plt.cm.Spectral)

methods = ['standard', 'ltsa', 'hessian', 'modified']
labels = ['LLE', 'LTSA', 'Hessian LLE', 'Modified LLE']

for i, method in enumerate(methods):
    t0 = time()
    Y = manifold.LocallyLinearEmbedding(n_neighbors, n_components,
                                        eigen_solver='auto',
                                        method=method).fit_transform(X)
    t1 = time()
    print("%s: %.2g sec" % (methods[i], t1 - t0))

    ax = fig.add_subplot(252 + i)
    plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
    plt.title("%s (%.2g sec)" % (labels[i], t1 - t0))
    ax.xaxis.set_major_formatter(NullFormatter())
    ax.yaxis.set_major_formatter(NullFormatter())
    plt.axis('tight')

t0 = time()
Y = manifold.Isomap(n_neighbors, n_components).fit_transform(X)
t1 = time()
print("Isomap: %.2g sec" % (t1 - t0))
ax = fig.add_subplot(257)
plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("Isomap (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')


t0 = time()
mds = manifold.MDS(n_components, max_iter=100, n_init=1)
Y = mds.fit_transform(X)
t1 = time()
print("MDS: %.2g sec" % (t1 - t0))
ax = fig.add_subplot(258)
plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("MDS (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')


t0 = time()
se = manifold.SpectralEmbedding(n_components=n_components,
                                n_neighbors=n_neighbors)
Y = se.fit_transform(X)
t1 = time()
print("SpectralEmbedding: %.2g sec" % (t1 - t0))
```

```
ax = fig.add_subplot(259)
plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("SpectralEmbedding (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

t0 = time()
tsne = manifold.TSNE(n_components=n_components, init='pca', random_state=0)
Y = tsne.fit_transform(X)
t1 = time()
print("t-SNE: %.2g sec" % (t1 - t0))
ax = fig.add_subplot(250)
plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("t-SNE (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

plt.show()
```

Total running time of the example: 17.42 seconds (0 minutes 17.42 seconds)

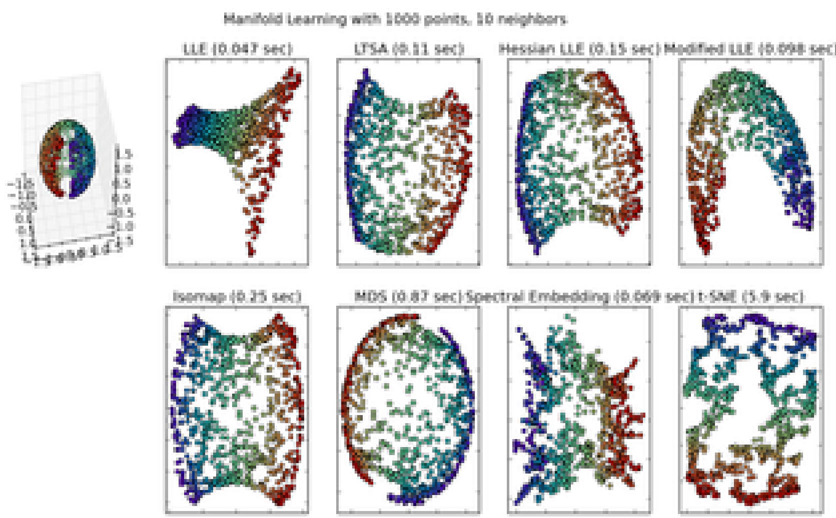


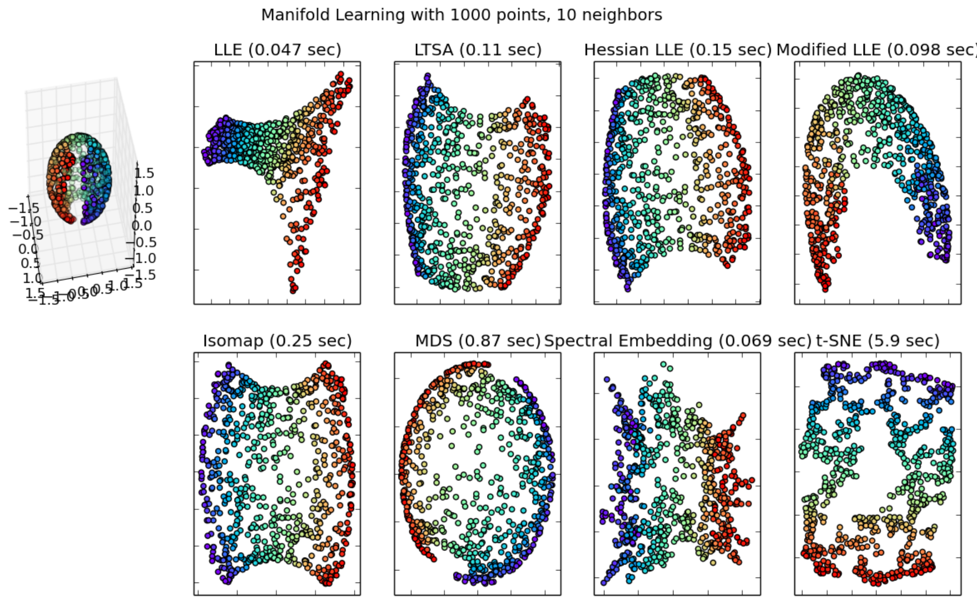
Figure 11.137: *Manifold Learning methods on a severed sphere*

11.16.4 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



Script output:

```
standard: 0.047 sec
ltsa: 0.11 sec
hessian: 0.15 sec
modified: 0.098 sec
ISO: 0.25 sec
MDS: 0.87 sec
Spectral Embedding: 0.069 sec
t-SNE: 5.9 sec
```

Python source code: `plot_manifold_sphere.py`

```
# Author: Jaques Grobler <jaques.grobler@inria.fr>
# License: BSD 3 clause

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)
try:
    # compatibility matplotlib < 1.0
    ax.view_init(40, -10)
except:
    pass

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))
    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(250)
plt.scatter(trans_data[0], trans_data[1],  c=colors, cmap=plt.cm.rainbow)
plt.title("t-SNE (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

plt.show()
```

Total running time of the example: 7.80 seconds (0 minutes 7.80 seconds)

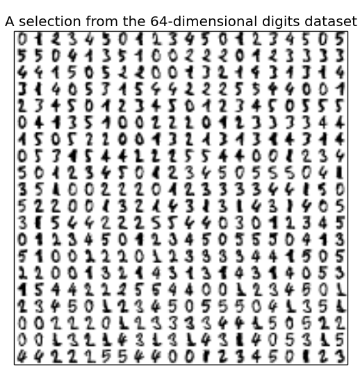
Figure 11.138: *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

11.16.5 Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...

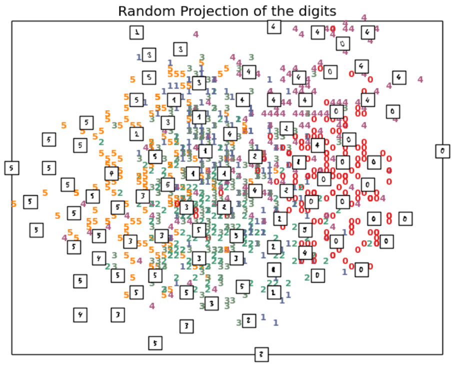
An illustration of various embeddings on the digits dataset.

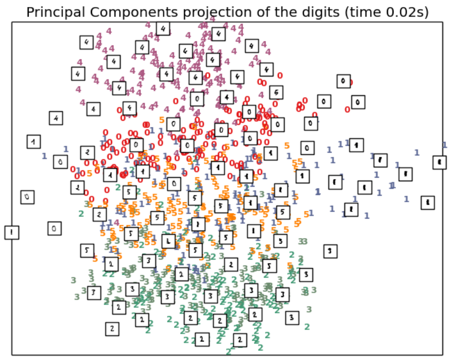
The RandomTreesEmbedding, from the `sklearn.ensemble` module, is not technically a manifold embedding method, as it learn a high-dimensional representation on which we apply a dimensionality reduction method. However, it is often useful to cast a dataset into a representation in which the classes are linearly-separable.

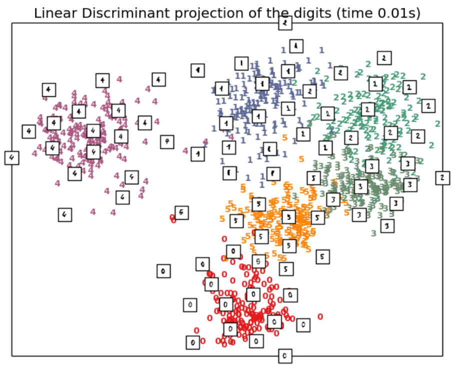
t-SNE will be initialized with the embedding that is generated by PCA in this example, which is not the default setting. It ensures global stability of the embedding, i.e., the embedding does not depend on random initialization.

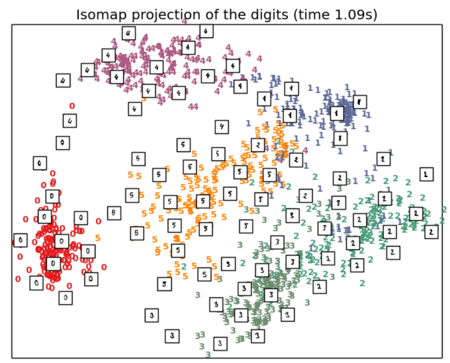


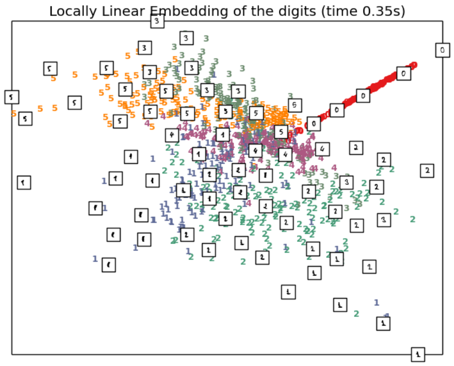
•

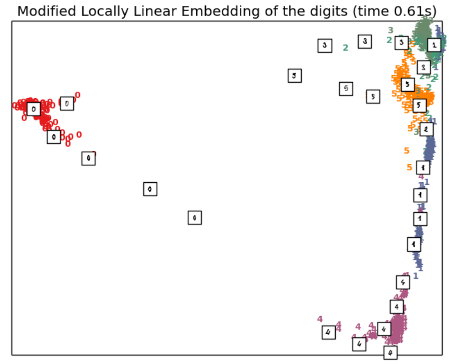
- 
Random Projection of the digits

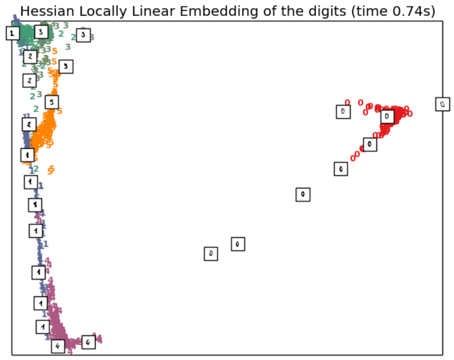
- 
Principal Components projection of the digits (time 0.02s)

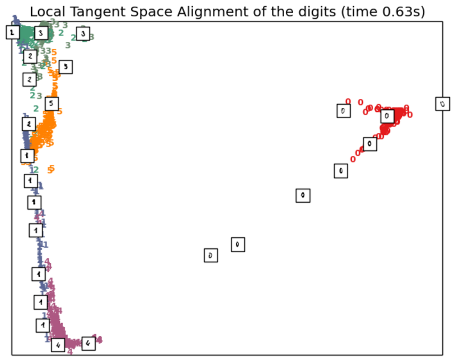
- 
Linear Discriminant projection of the digits (time 0.01s)

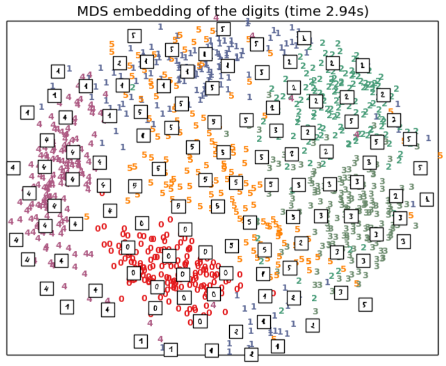
- 
Isomap projection of the digits (time 1.09s)

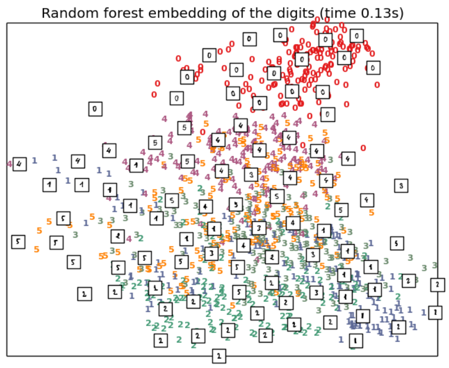
- 
Locally Linear Embedding of the digits (time 0.35s)

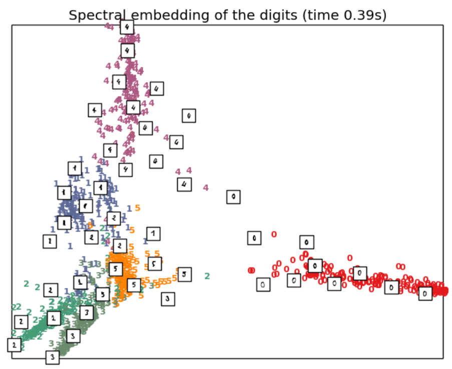
- 
Modified Locally Linear Embedding of the digits (time 0.61s)

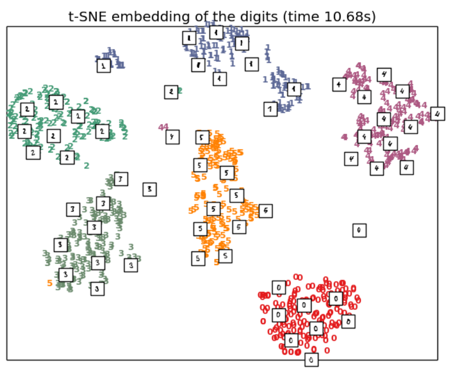
- 
Hessian Locally Linear Embedding of the digits (time 0.74s)

- 
Local Tangent Space Alignment of the digits (time 0.63s)

- 
MDS embedding of the digits (time 2.94s)

- 
Random forest embedding of the digits (time 0.13s)

- 
Spectral embedding of the digits (time 0.39s)

- 
t-SNE embedding of the digits (time 10.68s)

Script output:

```
Computing random projection
Computing PCA projection
Computing LDA projection
Computing Isomap embedding
Done.
Computing LLE embedding
Done. Reconstruction error: 1.63514e-06
Computing modified LLE embedding
Done. Reconstruction error: 0.36107
Computing Hessian LLE embedding
Done. Reconstruction error: 0.21292
Computing LTSA embedding
Done. Reconstruction error: 0.212808
Computing MDS embedding
Done. Stress: 147600495.378309
Computing Totally Random Trees embedding
Computing Spectral embedding
Computing t-SNE embedding
```

Python source code: `plot_lle_digits.py`

```python
# 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

print(__doc__)
from time import time

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import offsetbox
from sklearn import (manifold, datasets, decomposition, ensemble, lda,
                     random_projection)

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(digits.target[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(digits.data.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):
    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 LDA projection")
X2 = X.copy()
X2.flat[::X.shape[1] + 1] += 0.01  # Make X invertible
t0 = time()
```

```
X_lda = lda.LDA(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 embedding")
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")
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)" %
               (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))
```

```
plt.show()
```

Total running time of the example: 23.59 seconds (0 minutes 23.59 seconds)

11.17 Gaussian Mixture Models

Examples concerning the `sklearn.mixture` module.

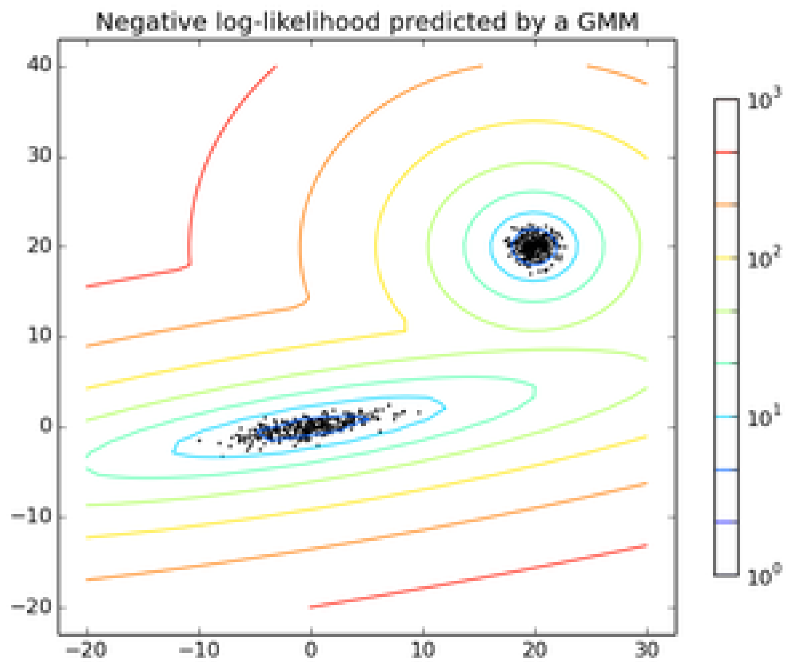
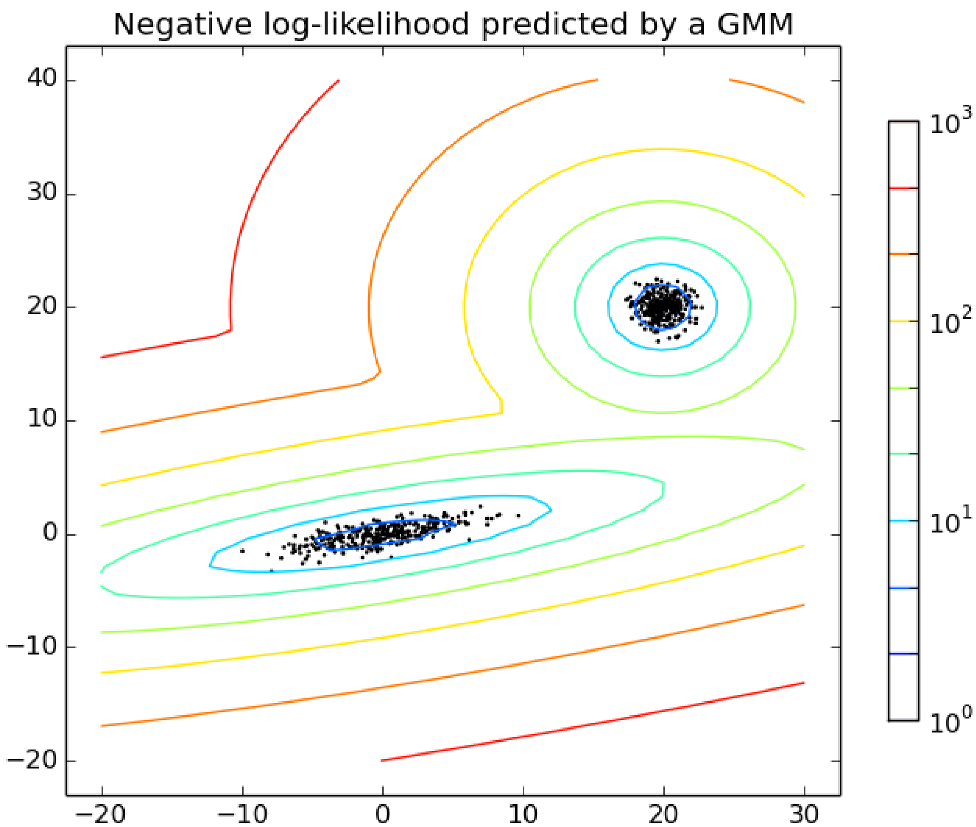


Figure 11.139: *Density Estimation for a mixture of Gaussians*

11.17.1 Density Estimation for a mixture of Gaussians

Plot the density estimation of a mixture of two Gaussians. Data is generated from two Gaussians with different centers and covariance matrices.



Python source code: `plot_gmm_pdf.py`

```
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.GMM(n_components=2, covariance_type='full')
clf.fit(X_train)
```

```
# display predicted scores by the model as a contour plot
x = np.linspace(-20.0, 30.0)
y = np.linspace(-20.0, 40.0)
X, Y = np.meshgrid(x, y)
XX = np.array([X.ravel(), Y.ravel()]).T
Z = -clf.score_samples(XX)[0]
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 example: 0.80 seconds (0 minutes 0.80 seconds)

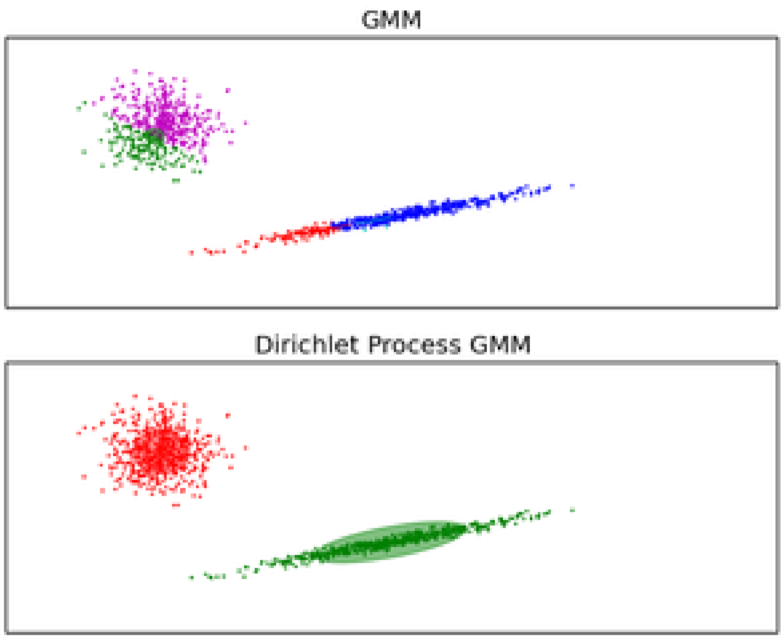


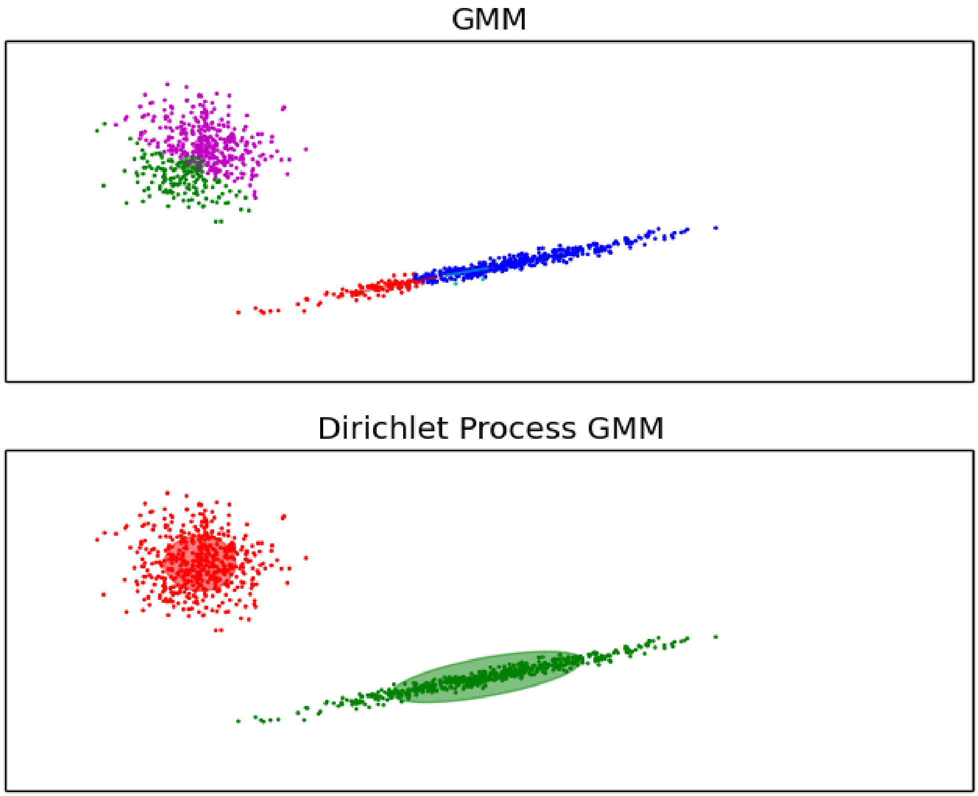
Figure 11.140: *Gaussian Mixture Model Ellipsoids*

11.17.2 Gaussian Mixture Model Ellipsoids

Plot the confidence ellipsoids of a mixture of two Gaussians with EM and variational Dirichlet process.

Both models have access to five components with which to fit the data. Note that the EM model will necessarily use all five components while the DP model will effectively only use as many as are needed for a good fit. This is a property of the Dirichlet Process prior. Here we can see that the EM 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.



Python source code: `plot_gmm.py`

```
import itertools

import numpy as np
from scipy import linalg
import matplotlib.pyplot as plt
import matplotlib as mpl

from sklearn import mixture

# 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 mixture of Gaussians with EM using five components
gmm = mixture.GMM(n_components=5, covariance_type='full')
```

```
gmm.fit(X)

# Fit a Dirichlet process mixture of Gaussians using five components
dpgmm = mixture.DPGMM(n_components=5, covariance_type='full')
dpgmm.fit(X)

color_iter = itertools.cycle(['r', 'g', 'b', 'c', 'm'])

for i, (clf, title) in enumerate([(gmm, 'GMM'),
                                  (dpgmm, 'Dirichlet Process GMM')]):
    splot = plt.subplot(2, 1, 1 + i)
    Y_ = clf.predict(X)
    for i, (mean, covar, color) in enumerate(zip(
            clf.means_, clf._get_covars(), color_iter)):
        v, w = linalg.eigh(covar)
        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(-10, 10)
    plt.ylim(-3, 6)
    plt.xticks(())
    plt.yticks(())
    plt.title(title)

plt.show()
```

Total running time of the example: 0.25 seconds (0 minutes 0.25 seconds)

11.17.3 Gaussian Mixture Model Sine Curve

This example highlights the advantages of the Dirichlet Process: complexity control and dealing with sparse data. The dataset is formed by 100 points loosely spaced following a noisy sine curve. The fit by the GMM class, using the expectation-maximization algorithm to fit a mixture of 10 Gaussian components, finds too-small components and very little structure. The fits by the Dirichlet process, however, show that the model can either learn a global structure for the data (small alpha) or easily interpolate to finding relevant local structure (large alpha), never falling into the problems shown by the GMM class.

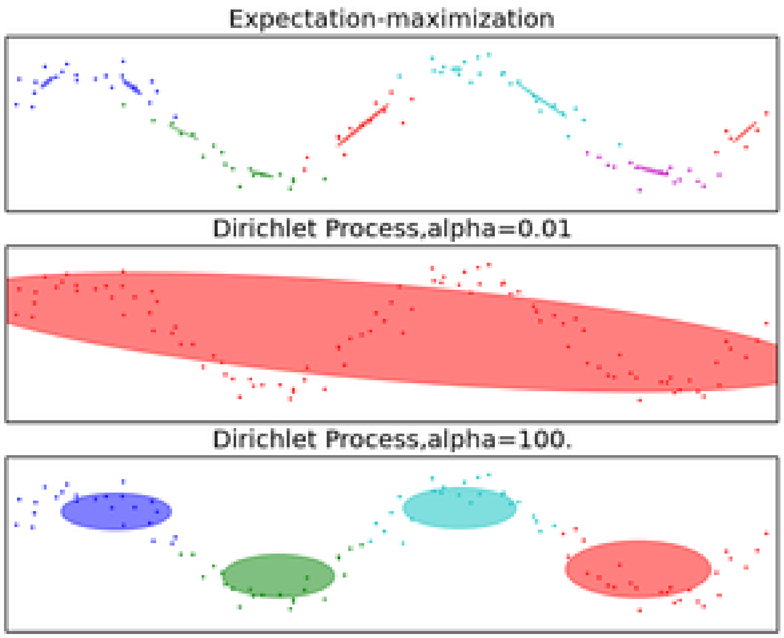
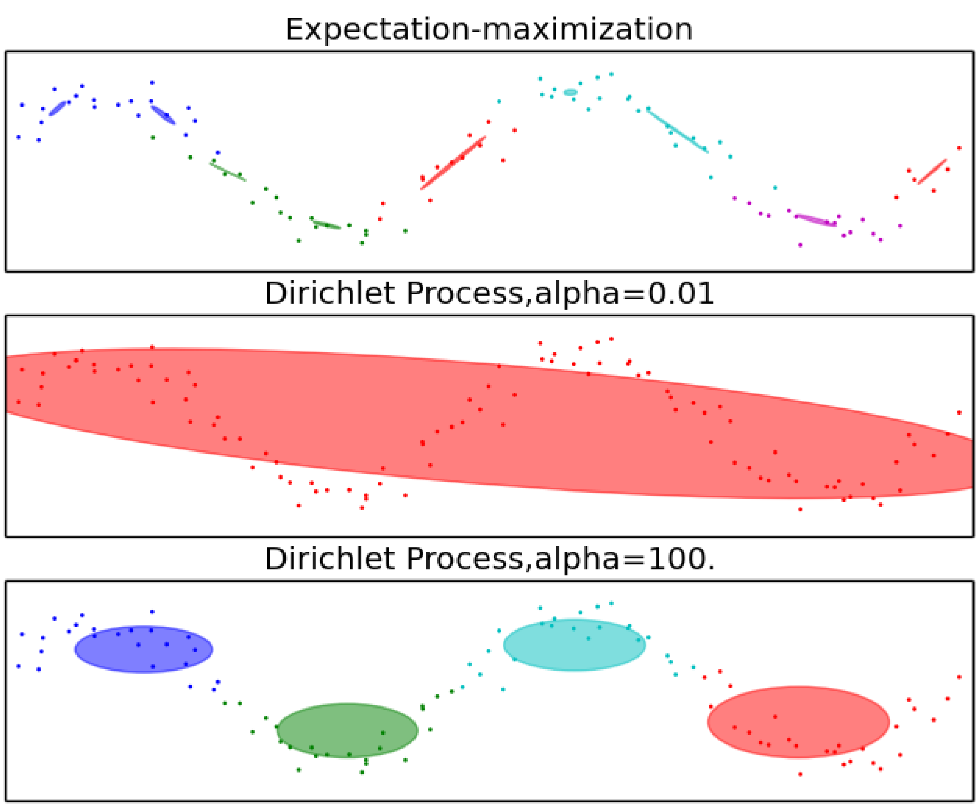


Figure 11.141: *Gaussian Mixture Model Sine Curve*



Python source code: plot_gmm_sin.py

```
import itertools

import numpy as np
from scipy import linalg
import matplotlib.pyplot as plt
import matplotlib as mpl

from sklearn import mixture
from sklearn.externals.six.moves import xrange

# Number of samples per component
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 xrange(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))


color_iter = itertools.cycle(['r', 'g', 'b', 'c', 'm'])


for i, (clf, title) in enumerate([
        (mixture.GMM(n_components=10, covariance_type='full', n_iter=100),
         "Expectation-maximization"),
        (mixture.DPGMM(n_components=10, covariance_type='full', alpha=0.01,
                       n_iter=100),
         "Dirichlet Process,alpha=0.01"),
        (mixture.DPGMM(n_components=10, covariance_type='diag', alpha=100.,
                       n_iter=100),
         "Dirichlet Process,alpha=100.")]):

    clf.fit(X)
    splot = plt.subplot(3, 1, 1 + i)
    Y_ = clf.predict(X)
    for i, (mean, covar, color) in enumerate(zip(
            clf.means_, clf._get_covars(), color_iter)):
        v, w = linalg.eigh(covar)
        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(())

plt.show()
```

Total running time of the example: 0.25 seconds (0 minutes 0.25 seconds)

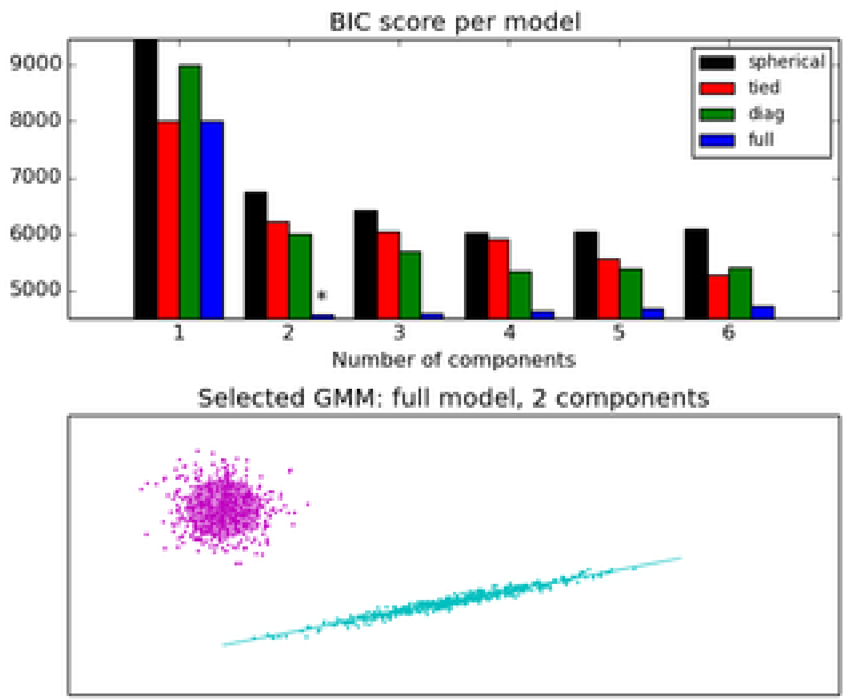
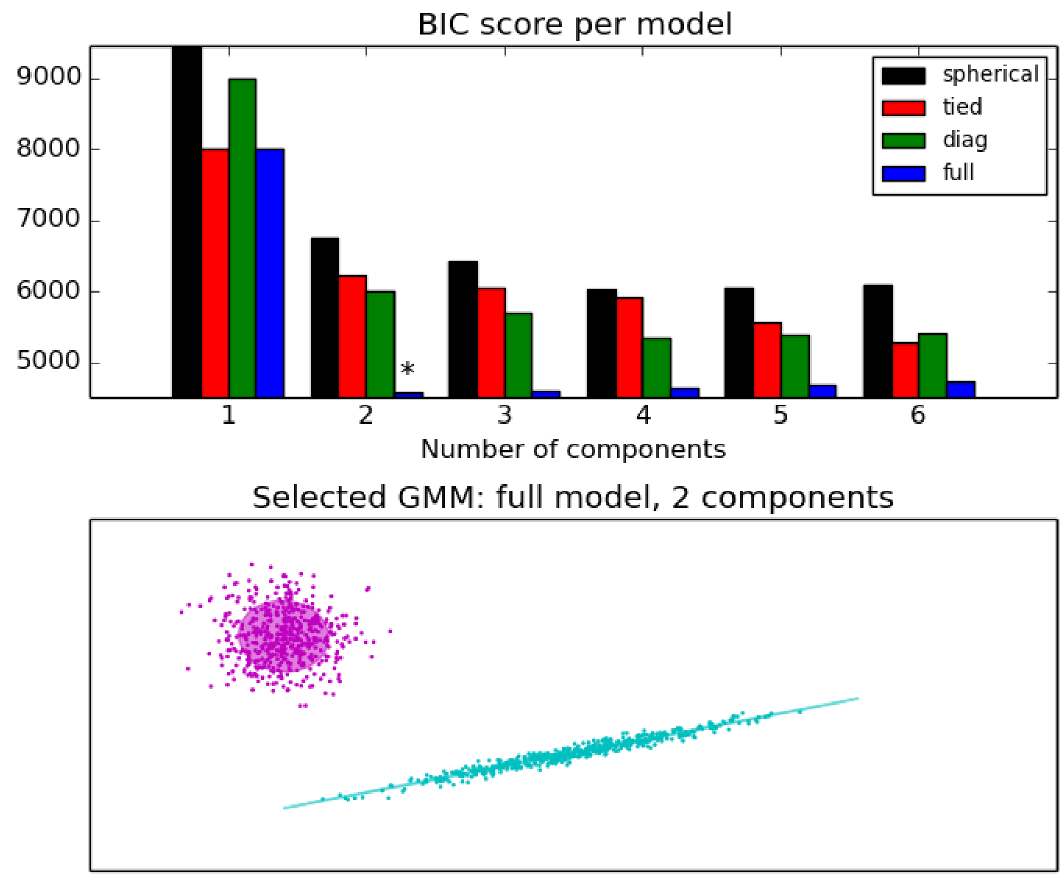


Figure 11.142: *Gaussian Mixture Model Selection*

11.17.4 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.



Python source code: `plot_gmm_selection.py`

```
print(__doc__)

import itertools

import numpy as np
from scipy import linalg
import matplotlib.pyplot as plt
import matplotlib as mpl

from sklearn import mixture

# 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:
    for n_components in n_components_range:
        # Fit a mixture of Gaussians with EM
        gmm = mixture.GMM(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(['k', 'r', 'g', 'b', 'c', 'm', 'y'])
clf = best_gmm
bars = []

# Plot the BIC scores
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, covar, color) in enumerate(zip(clf.means_, clf.covars_,
                                             color_iter)):
    v, w = linalg.eigh(covar)
    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 *= 4
    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.xlim(-10, 10)
plt.ylim(-3, 6)
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 example: 0.74 seconds (0 minutes 0.74 seconds)

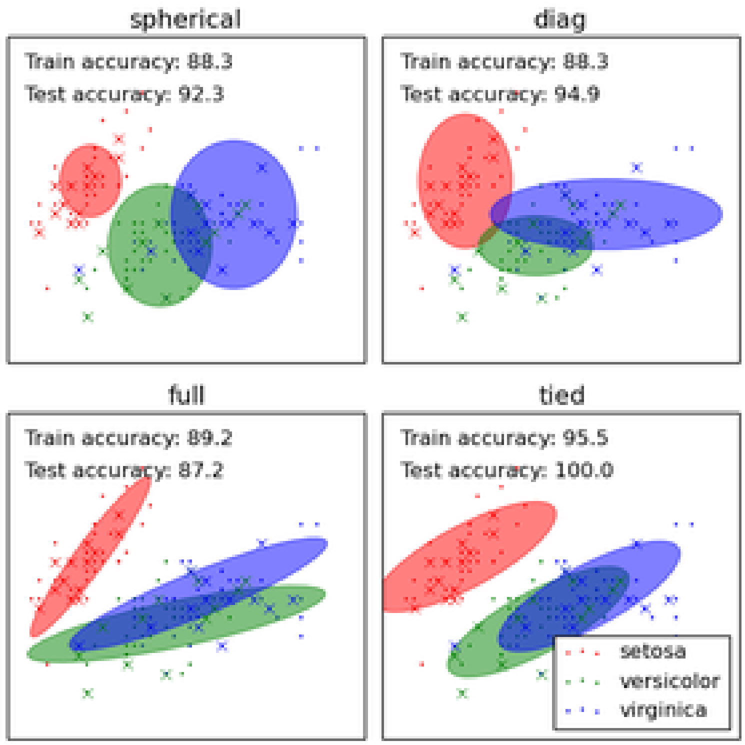


Figure 11.143: *GMM classification*

11.17.5 GMM classification

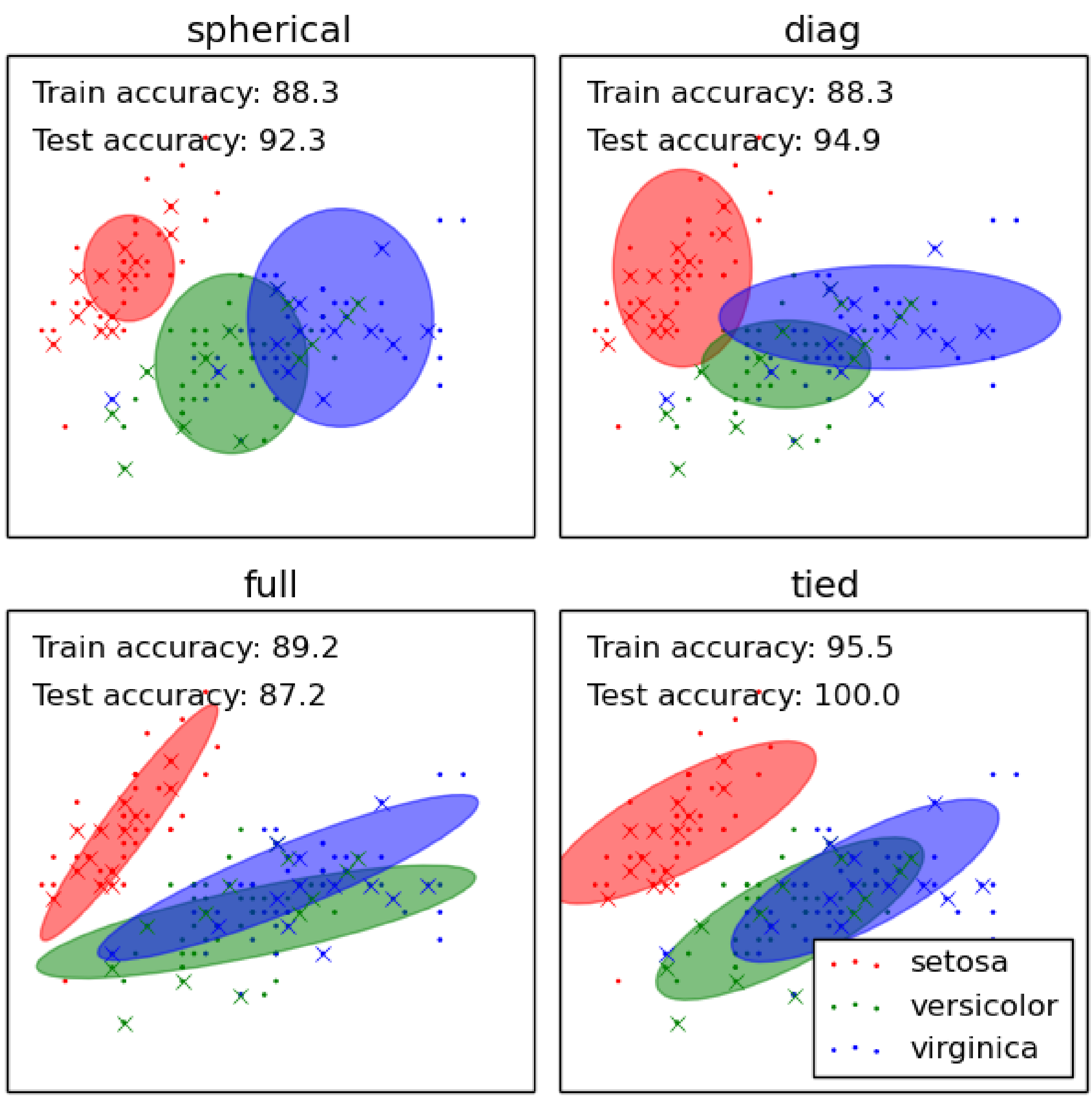
Demonstration of Gaussian mixture models for classification.

See *Gaussian mixture models* for more information on the estimator.

Plots predicted labels on both training and held out test data using a variety of GMM classifiers on the iris dataset.

Compares 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.



Python source code: `plot_gmm_classifier.py`

```python
print(__doc__)

# Author: Ron Weiss <ronweiss@gmail.com>, Gael Varoquaux
# License: BSD 3 clause

# $Id$

import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np

from sklearn import datasets
from sklearn.cross_validation import StratifiedKFold
from sklearn.externals.six.moves import xrange
from sklearn.mixture import GMM
```

```
def make_ellipses(gmm, ax):
    for n, color in enumerate('rgb'):
        v, w = np.linalg.eigh(gmm._get_covars()[n][:2, :2])
        u = w[0] / np.linalg.norm(w[0])
        angle = np.arctan2(u[1], u[0])
        angle = 180 * angle / np.pi  # convert to degrees
        v *= 9
        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)

iris = datasets.load_iris()

# Break up the dataset into non-overlapping training (75%) and testing
# (25%) sets.
skf = StratifiedKFold(iris.target, n_folds=4)
# Only take the first fold.
train_index, test_index = next(iter(skf))


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.
classifiers = dict((covar_type, GMM(n_components=n_classes,
                    covariance_type=covar_type, init_params='wc', n_iter=20))
                   for covar_type in ['spherical', 'diag', 'tied', 'full'])

n_classifiers = len(classifiers)

plt.figure(figsize=(3 * n_classifiers / 2, 6))
plt.subplots_adjust(bottom=.01, top=0.95, hspace=.15, wspace=.05,
                    left=.01, right=.99)


for index, (name, classifier) in enumerate(classifiers.items()):
    # Since we have class labels for the training data, we can
    # initialize the GMM parameters in a supervised manner.
    classifier.means_ = np.array([X_train[y_train == i].mean(axis=0)
                                  for i in xrange(n_classes)])

    # Train the other parameters using the EM algorithm.
    classifier.fit(X_train)

    h = plt.subplot(2, n_classifiers / 2, index + 1)
    make_ellipses(classifier, h)

    for n, color in enumerate('rgb'):
        data = iris.data[iris.target == n]
        plt.scatter(data[:, 0], data[:, 1], 0.8, color=color,
                    label=iris.target_names[n])
    # Plot the test data with crosses
```

```
    for n, color in enumerate('rgb'):
        data = X_test[y_test == n]
        plt.plot(data[:, 0], data[:, 1], 'x', color=color)

    y_train_pred = classifier.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 = classifier.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(loc='lower right', prop=dict(size=12))


plt.show()
```

Total running time of the example: 0.21 seconds (0 minutes 0.21 seconds)

11.18 Model Selection

Examples concerning model selection, mostly contained in the `sklearn.grid_search` and `sklearn.cross_validation` modules.

11.18.1 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.

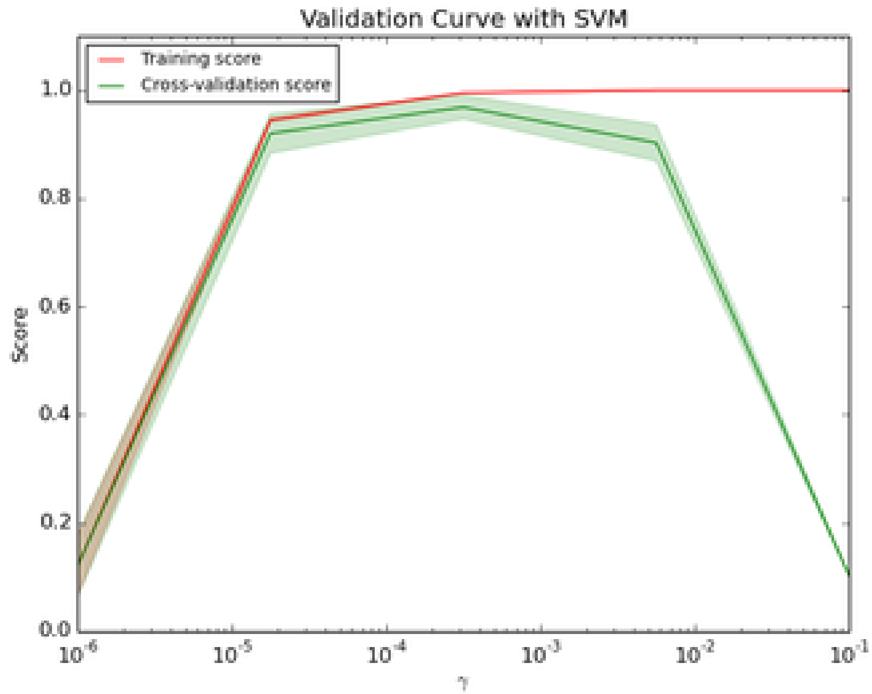
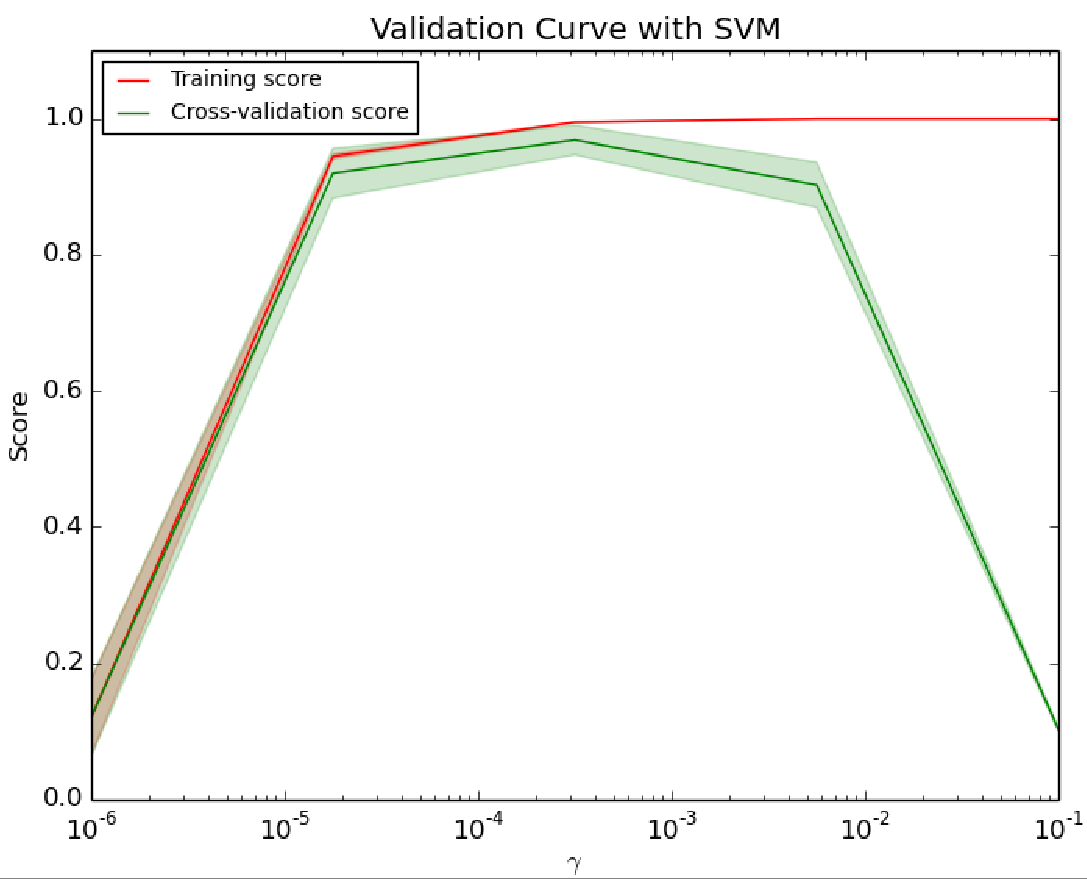


Figure 11.144: *Plotting Validation Curves*



Python source code: `plot_validation_curve.py`

```
print(__doc__)

import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_digits
from sklearn.svm import SVC
from sklearn.learning_curve import validation_curve

digits = load_digits()
X, y = digits.data, digits.target

param_range = np.logspace(-6, -1, 5)
train_scores, test_scores = validation_curve(
    SVC(), X, y, param_name="gamma", param_range=param_range,
    cv=10, 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("$\gamma$")
plt.ylabel("Score")
plt.ylim(0.0, 1.1)
plt.semilogx(param_range, train_scores_mean, label="Training score", color="r")
plt.fill_between(param_range, train_scores_mean - train_scores_std,
                 train_scores_mean + train_scores_std, alpha=0.2, color="r")
plt.semilogx(param_range, test_scores_mean, label="Cross-validation score",
             color="g")
plt.fill_between(param_range, test_scores_mean - test_scores_std,
                 test_scores_mean + test_scores_std, alpha=0.2, color="g")
plt.legend(loc="best")
plt.show()
```

Total running time of the example: 31.19 seconds (0 minutes 31.19 seconds)

11.18.2 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.

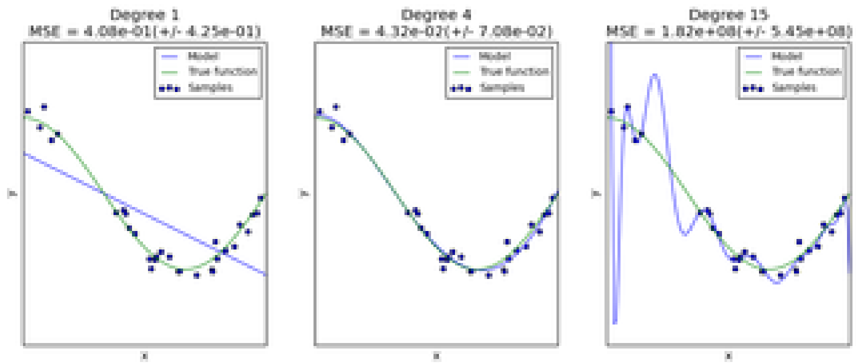
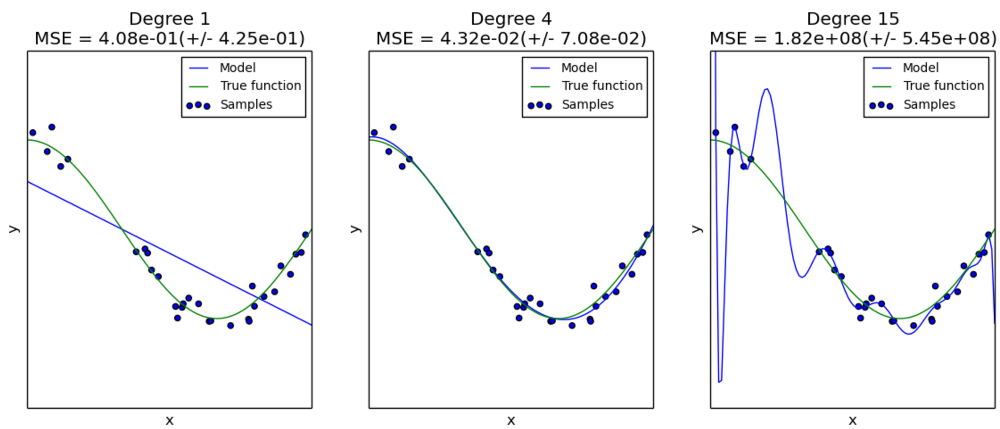


Figure 11.145: *Underfitting vs. Overfitting*



Python source code: `plot_underfitting_overfitting.py`

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn import cross_validation

np.random.seed(0)
```

```
n_samples = 30
degrees = [1, 4, 15]

true_fun = lambda X: np.cos(1.5 * np.pi * X)
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_validation.cross_val_score(pipeline,
        X[:, np.newaxis], y, scoring="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, 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 example: 0.18 seconds (0 minutes 0.18 seconds)

11.18.3 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 *Grid Search: Searching for estimator parameters*.

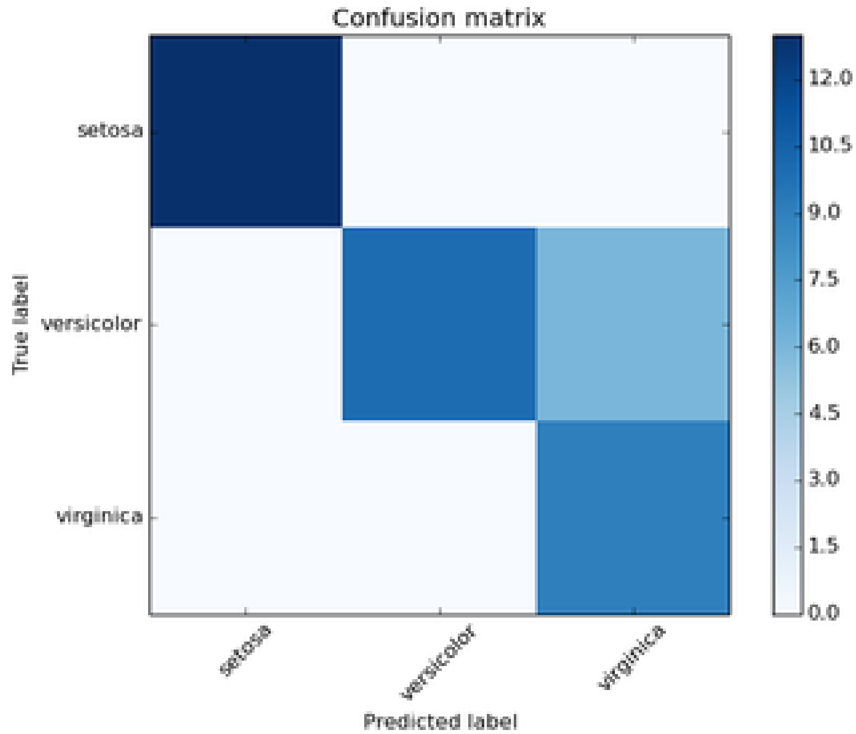
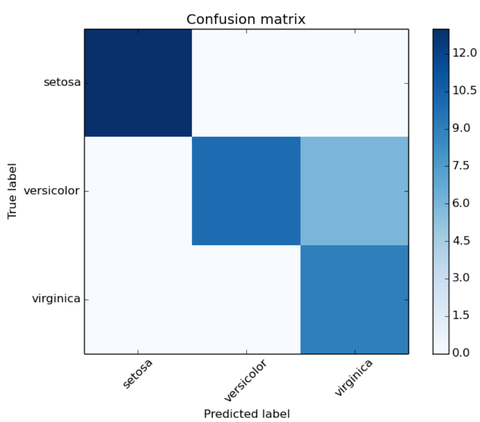
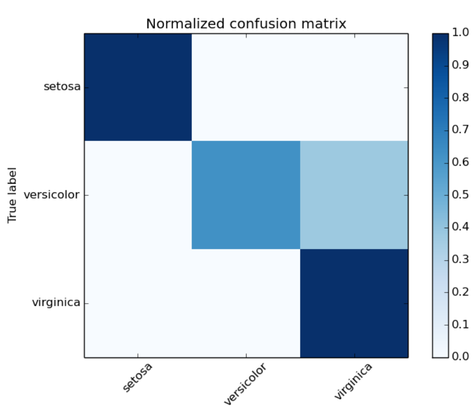


Figure 11.146: *Confusion matrix*

- 

- 

Script output:

```
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.  ]]
```

Python source code: plot_confusion_matrix.py

```python
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn import svm, datasets
from sklearn.cross_validation import train_test_split
from sklearn.metrics import confusion_matrix

# import some data to play with
iris = datasets.load_iris()
X = iris.data
y = iris.target

# 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)
y_pred = classifier.fit(X_train, y_train).predict(X_test)


def plot_confusion_matrix(cm, title='Confusion matrix', cmap=plt.cm.Blues):
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(iris.target_names))
    plt.xticks(tick_marks, iris.target_names, rotation=45)
    plt.yticks(tick_marks, iris.target_names)
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')


# Compute confusion matrix
cm = confusion_matrix(y_test, y_pred)
np.set_printoptions(precision=2)
print('Confusion matrix, without normalization')
print(cm)
plt.figure()
plot_confusion_matrix(cm)

# Normalize the confusion matrix by row (i.e by the number of samples
# in each class)
cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
```

```
print('Normalized confusion matrix')
print(cm_normalized)
plt.figure()
plot_confusion_matrix(cm_normalized, title='Normalized confusion matrix')

plt.show()
```

Total running time of the example: 0.20 seconds (0 minutes 0.20 seconds)

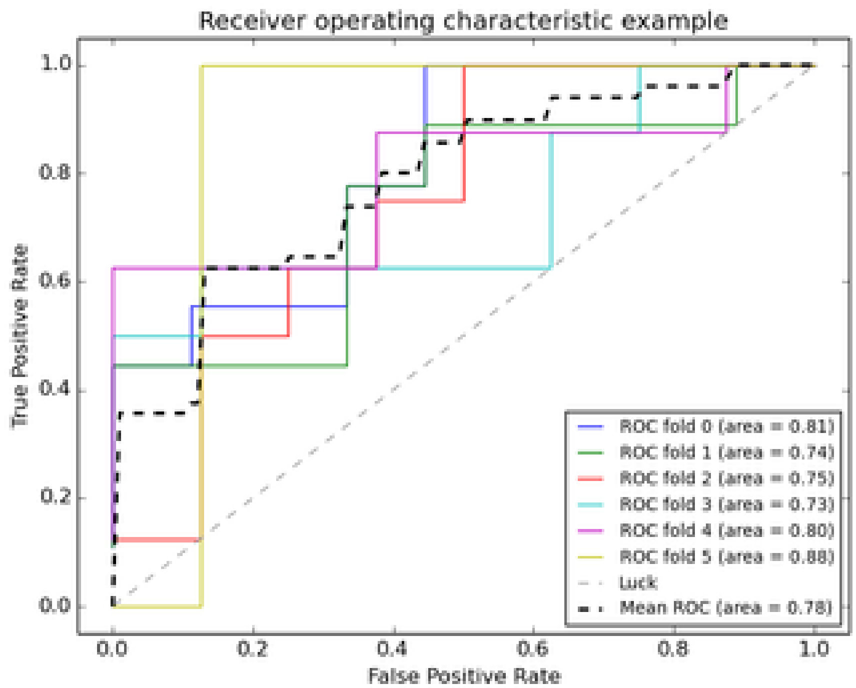


Figure 11.147: *Receiver Operating Characteristic (ROC) with cross validation*

11.18.4 Receiver Operating Characteristic (ROC) with cross validation

Example of Receiver Operating Characteristic (ROC) metric to evaluate classifier output quality using cross-validation.

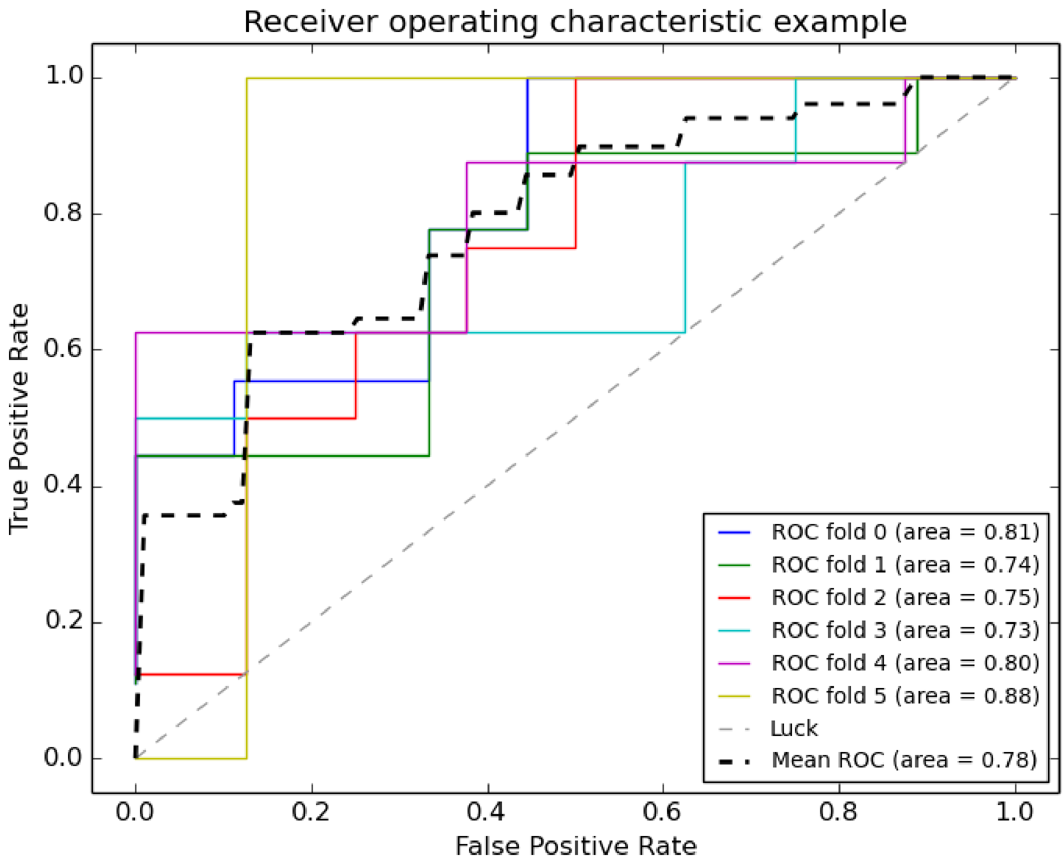
ROC curves typically feature true positive rate on the Y axis, and false positive rate on the X axis. This means that the top left corner of the plot is the "ideal" point - a false positive rate of zero, and a true positive rate of one. This is not very realistic, but it does mean that a larger area 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.auc_score**, sklearn.cross_validation.cross_val_score, *Receiver Operating Characteristic (ROC)*,



Python source code: plot_roc_crossval.py

```
print(__doc__)

import numpy as np
from scipy import interp
import matplotlib.pyplot as plt

from sklearn import svm, datasets
from sklearn.metrics import roc_curve, auc
from sklearn.cross_validation 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(y, n_folds=6)
classifier = svm.SVC(kernel='linear', probability=True,
                     random_state=random_state)

mean_tpr = 0.0
mean_fpr = np.linspace(0, 1, 100)
all_tpr = []

for i, (train, test) in enumerate(cv):
    probas_ = classifier.fit(X[train], y[train]).predict_proba(X[test])
    # Compute ROC curve and area the curve
    fpr, tpr, thresholds = roc_curve(y[test], probas_[:, 1])
    mean_tpr += interp(mean_fpr, fpr, tpr)
    mean_tpr[0] = 0.0
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, lw=1, label='ROC fold %d (area = %0.2f)' % (i, roc_auc))

plt.plot([0, 1], [0, 1], '--', color=(0.6, 0.6, 0.6), label='Luck')

mean_tpr /= len(cv)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
plt.plot(mean_fpr, mean_tpr, 'k--',
         label='Mean ROC (area = %0.2f)' % mean_auc, lw=2)

plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()
```

Total running time of the example: 0.28 seconds (0 minutes 0.28 seconds)

Figure 11.148: *Parameter estimation using grid search with cross-validation*

11.18.5 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.grid_search.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 *Grid Search: Searching for estimator parameters*.

Python source code: `grid_search_digits.py`

```
from __future__ import print_function

from sklearn import datasets
from sklearn.cross_validation import train_test_split
from sklearn.grid_search 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:
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(C=1), tuned_parameters, cv=5,
                       scoring='%s_weighted' % 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()
    for params, mean_score, scores in clf.grid_scores_:
```

```
        print("%0.3f (+/-%0.03f) for %r"
              % (mean_score, scores.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.
```

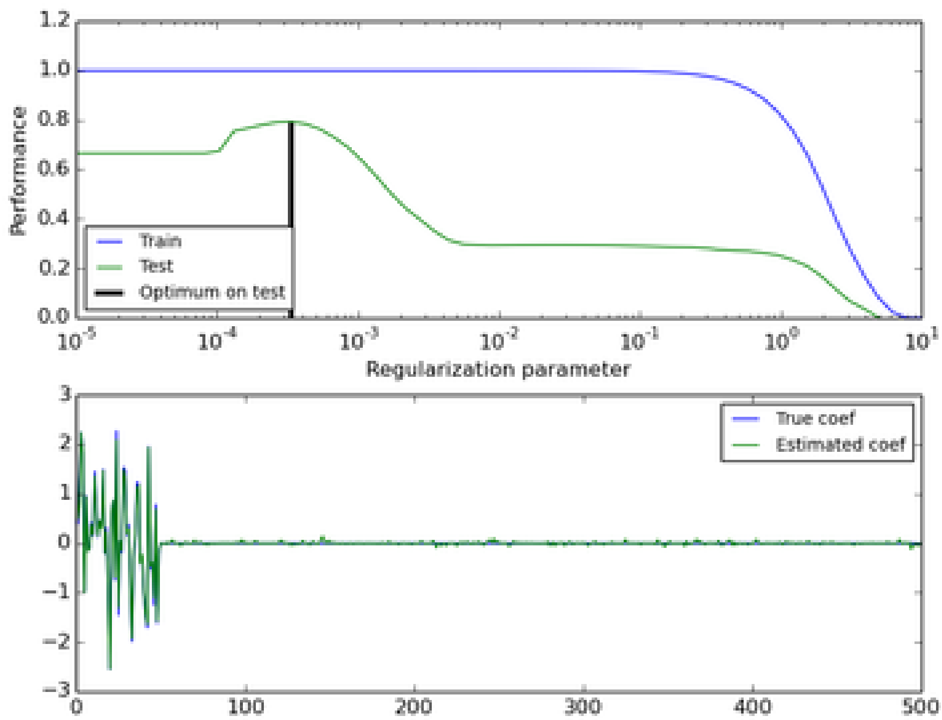
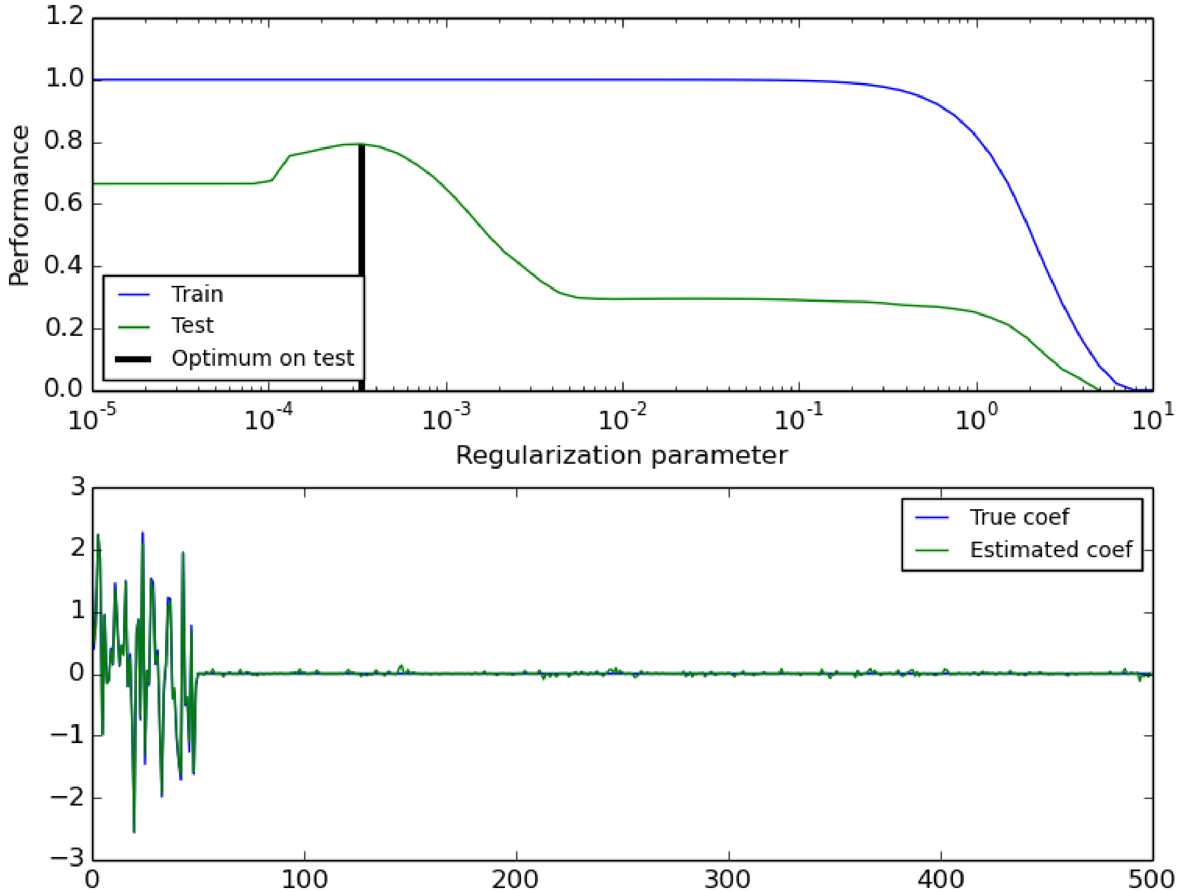


Figure 11.149: *Train error vs Test error*

11.18.6 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.



Script output:

```
Optimal regularization parameter : 0.000335292414925
```

Python source code: `plot_train_error_vs_test_error.py`

```
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)
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)
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 example: 1.76 seconds (0 minutes 1.76 seconds)

11.18.7 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 slightly worse for the randomized search, though this is most likely a noise effect and would not carry over to a held-out test set.

Figure 11.150: *Comparing randomized search and grid search for hyperparameter estimation*

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.

Python source code: randomized_search.py

```
print(__doc__)

import numpy as np

from time import time
from operator import itemgetter
from scipy.stats import randint as sp_randint

from sklearn.grid_search import GridSearchCV, RandomizedSearchCV
from sklearn.datasets import load_digits
from sklearn.ensemble import RandomForestClassifier

# get some data
iris = load_digits()
X, y = iris.data, iris.target

# build a classifier
clf = RandomForestClassifier(n_estimators=20)


# Utility function to report best scores
def report(grid_scores, n_top=3):
    top_scores = sorted(grid_scores, key=itemgetter(1), reverse=True)[:n_top]
    for i, score in enumerate(top_scores):
        print("Model with rank: {0}".format(i + 1))
        print("Mean validation score: {0:.3f} (std: {1:.3f})".format(
              score.mean_validation_score,
              np.std(score.cv_validation_scores)))
        print("Parameters: {0}".format(score.parameters))
        print("")


# specify parameters and distributions to sample from
param_dist = {"max_depth": [3, None],
              "max_features": sp_randint(1, 11),
              "min_samples_split": sp_randint(1, 11),
              "min_samples_leaf": sp_randint(1, 11),
```

```
              "bootstrap": [True, False],
              "criterion": ["gini", "entropy"]}

# 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.grid_scores_)

# use a full grid over all parameters
param_grid = {"max_depth": [3, None],
              "max_features": [1, 3, 10],
              "min_samples_split": [1, 3, 10],
              "min_samples_leaf": [1, 3, 10],
              "bootstrap": [True, False],
              "criterion": ["gini", "entropy"]}

# 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.grid_scores_)))
report(grid_search.grid_scores_)
```

11.18.8 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-class or 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).

Note:

See also **`sklearn.metrics.roc_auc_score`**, *Receiver Operating Characteristic (ROC) with cross validation*.

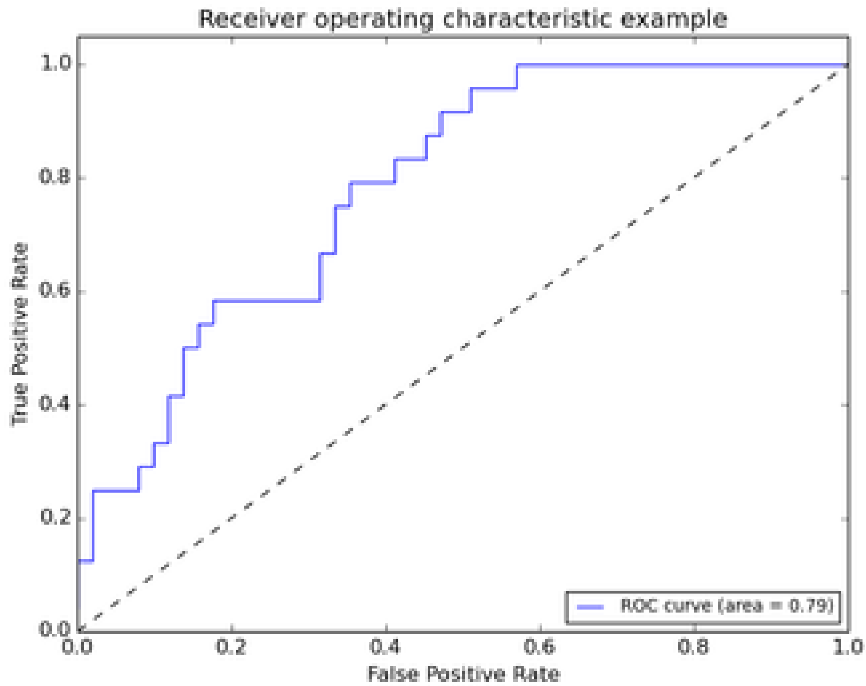
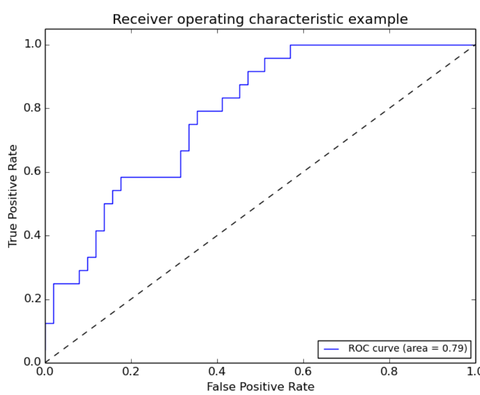
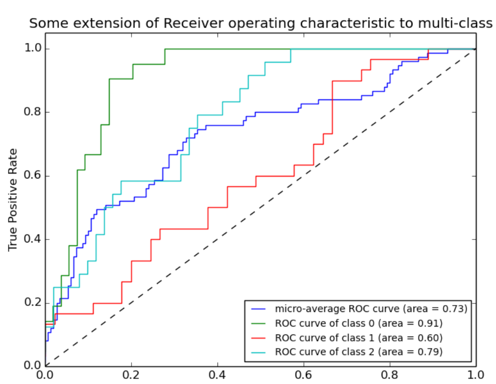


Figure 11.151: *Receiver Operating Characteristic (ROC)*

- 

- 

Python source code: `plot_roc.py`

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets
from sklearn.metrics import roc_curve, auc
from sklearn.cross_validation import train_test_split
from sklearn.preprocessing import label_binarize
from sklearn.multiclass import OneVsRestClassifier

# 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()
plt.plot(fpr[2], tpr[2], label='ROC curve (area = %0.2f)' % roc_auc[2])
plt.plot([0, 1], [0, 1], 'k--')
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")
plt.show()
```

```
# Plot ROC curve
plt.figure()
plt.plot(fpr["micro"], tpr["micro"],
         label='micro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["micro"]))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], label='ROC curve of class {0} (area = {1:0.2f})'
                                   ''.format(i, roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--')
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()
```

Total running time of the example: 0.21 seconds (0 minutes 0.21 seconds)

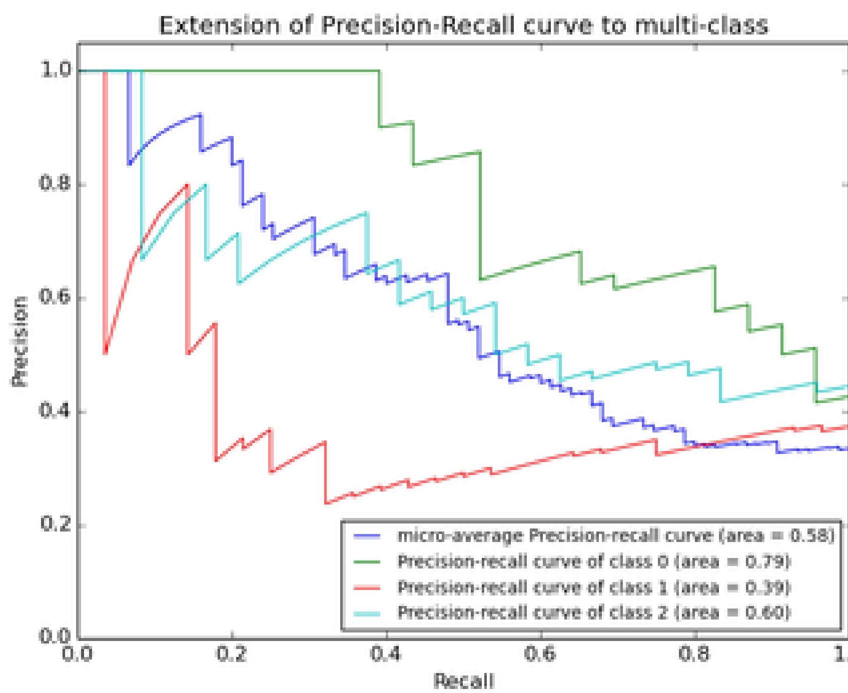


Figure 11.152: *Precision-Recall*

11.18.9 Precision-Recall

Example of Precision-Recall metric to evaluate classifier output quality.

In information retrieval, precision is a measure of result relevancy, while recall is a measure of how many truly relevant results are returned. 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.

$F1 = 2\frac{P \times R}{P+R}$

It is important to 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.

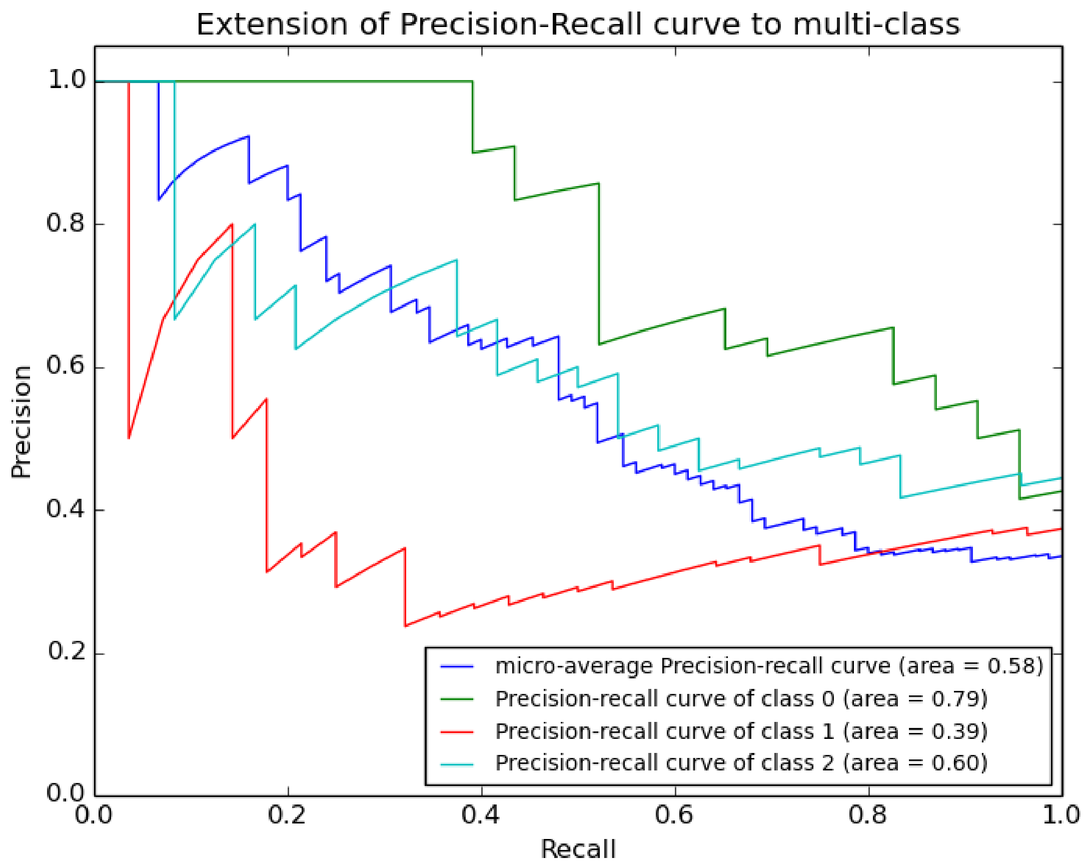
Recall is defined as $\frac{T_p}{T_p+F_n}$, where $T_p + F_n$ does not depend on the classifier threshold. This means that lowering the classifier threshold may increase recall, by increasing the number of true positive results. It is also possible that lowering the threshold may leave recall unchanged, 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. See the corner at recall = .59, precision = .8 for an example of this phenomenon.

Precision-recall curves are typically used in binary classification to study the output of a classifier. In order to extend 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`



Python source code: plot_precision_recall.py

```
print(__doc__)

import matplotlib.pyplot as plt
import numpy as np
from sklearn import svm, datasets
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import average_precision_score
from sklearn.cross_validation import train_test_split
from sklearn.preprocessing import label_binarize
from sklearn.multiclass import OneVsRestClassifier

# 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
random_state = np.random.RandomState(0)
n_samples, n_features = X.shape
X = np.c_[X, random_state.randn(n_samples, 200 * n_features)]
```

```
# 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)

# Run classifier
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 Precision-Recall and plot curve
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])

# Compute micro-average ROC curve and ROC area
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")

# Plot Precision-Recall curve
plt.clf()
plt.plot(recall[0], precision[0], label='Precision-Recall curve')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Precision-Recall example: AUC={0:0.2f}'.format(average_precision[0]))
plt.legend(loc="lower left")
plt.show()

# Plot Precision-Recall curve for each class
plt.clf()
plt.plot(recall["micro"], precision["micro"],
         label='micro-average Precision-recall curve (area = {0:0.2f})'
               ''.format(average_precision["micro"]))
for i in range(n_classes):
    plt.plot(recall[i], precision[i],
             label='Precision-recall curve of class {0} (area = {1:0.2f})'
                   ''.format(i, average_precision[i]))

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(loc="lower right")
plt.show()
```

Total running time of the example: 0.23 seconds (0 minutes 0.23 seconds)

Figure 11.153: *Sample pipeline for text feature extraction and evaluation*

11.18.10 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__n_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__n_iter: 50
    clf__penalty: 'elasticnet'
    tfidf__use_idf: True
    vect__max_n: 2
    vect__max_df: 0.75
    vect__max_features: 50000
```

Python source code: `grid_search_text_feature_extraction.py`

```
# Author: Olivier Grisel <olivier.grisel@ensta.org>
#         Peter Prettenhofer <peter.prettenhofer@gmail.com>
#         Mathieu Blondel <mathieu@mblondel.org>
# License: BSD 3 clause
```

```python
from __future__ import print_function

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.grid_search 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
#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__alpha': (0.00001, 0.000001),
    'clf__penalty': ('l2', 'elasticnet'),
    #'clf__n_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]))
```

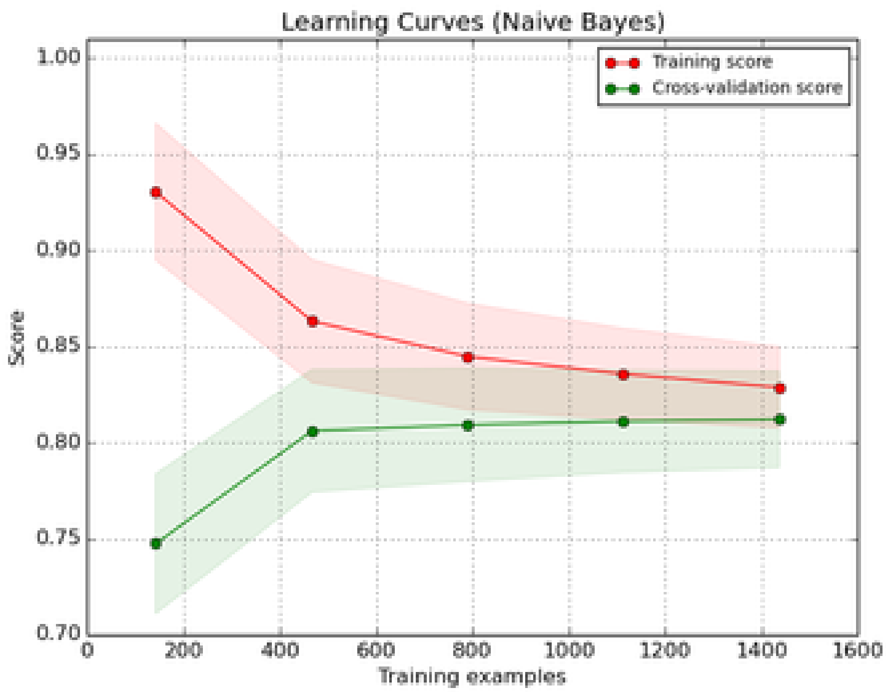
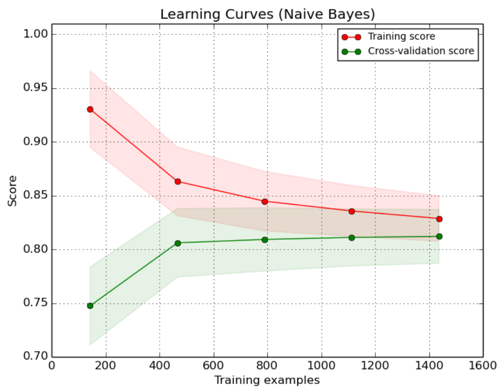
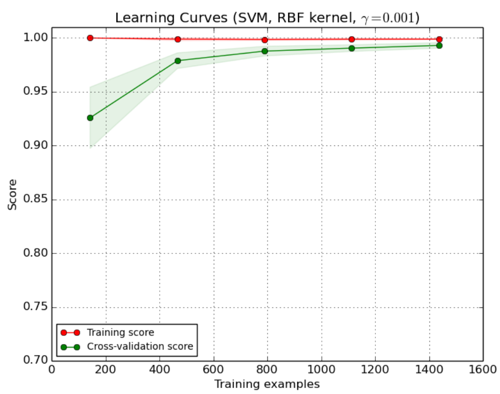


Figure 11.154: *Plotting Learning Curves*

11.18.11 Plotting Learning Curves

On the left side 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. On the right side 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.

- 

- 

Python source code: `plot_learning_curve.py`

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import cross_validation
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.datasets import load_digits
from sklearn.learning_curve import learning_curve


def plot_learning_curve(estimator, title, X, y, ylim=None, cv=None,
                        n_jobs=1, train_sizes=np.linspace(.1, 1.0, 5)):
    """
    Generate a simple plot of the test and traning learning 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.

    ylim : tuple, shape (ymin, ymax), optional
        Defines minimum and maximum yvalues plotted.

    cv : integer, cross-validation generator, optional
        If an integer is passed, it is the number of folds (defaults to 3).
        Specific cross-validation objects can be passed, see
        sklearn.cross_validation module for the list of possible objects

    n_jobs : integer, optional
        Number of jobs to run in parallel (default 1).
    """
    plt.figure()
    plt.title(title)
    if ylim is not None:
        plt.ylim(*ylim)
    plt.xlabel("Training examples")
    plt.ylabel("Score")
    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes)
    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.grid()

    plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="r")
    plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1, color="g")
    plt.plot(train_sizes, train_scores_mean, 'o-', color="r",
             label="Training score")
    plt.plot(train_sizes, test_scores_mean, 'o-', color="g",
             label="Cross-validation score")

    plt.legend(loc="best")
    return plt


digits = load_digits()
X, y = digits.data, digits.target


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 = cross_validation.ShuffleSplit(digits.data.shape[0], n_iter=100,
                                   test_size=0.2, random_state=0)
```

```
estimator = GaussianNB()
plot_learning_curve(estimator, title, X, y, ylim=(0.7, 1.01), cv=cv, n_jobs=4)

title = "Learning Curves (SVM, RBF kernel, $\gamma=0.001$)"
# SVC is more expensive so we do a lower number of CV iterations:
cv = cross_validation.ShuffleSplit(digits.data.shape[0], n_iter=10,
                                   test_size=0.2, random_state=0)
estimator = SVC(gamma=0.001)
plot_learning_curve(estimator, title, X, y, (0.7, 1.01), cv=cv, n_jobs=4)

plt.show()
```

Total running time of the example: 6.31 seconds (0 minutes 6.31 seconds)

11.19 Nearest Neighbors

Examples concerning the `sklearn.neighbors` module.

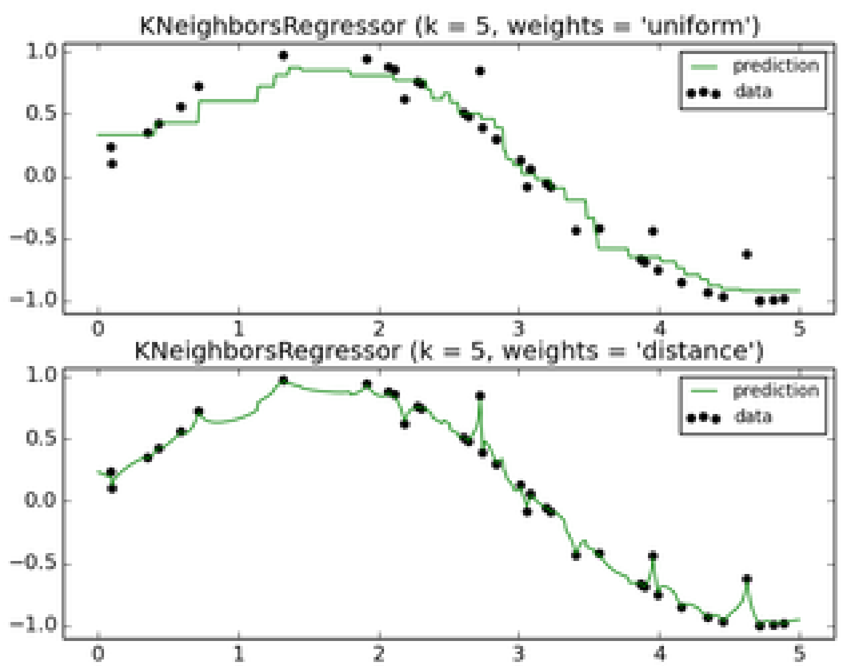
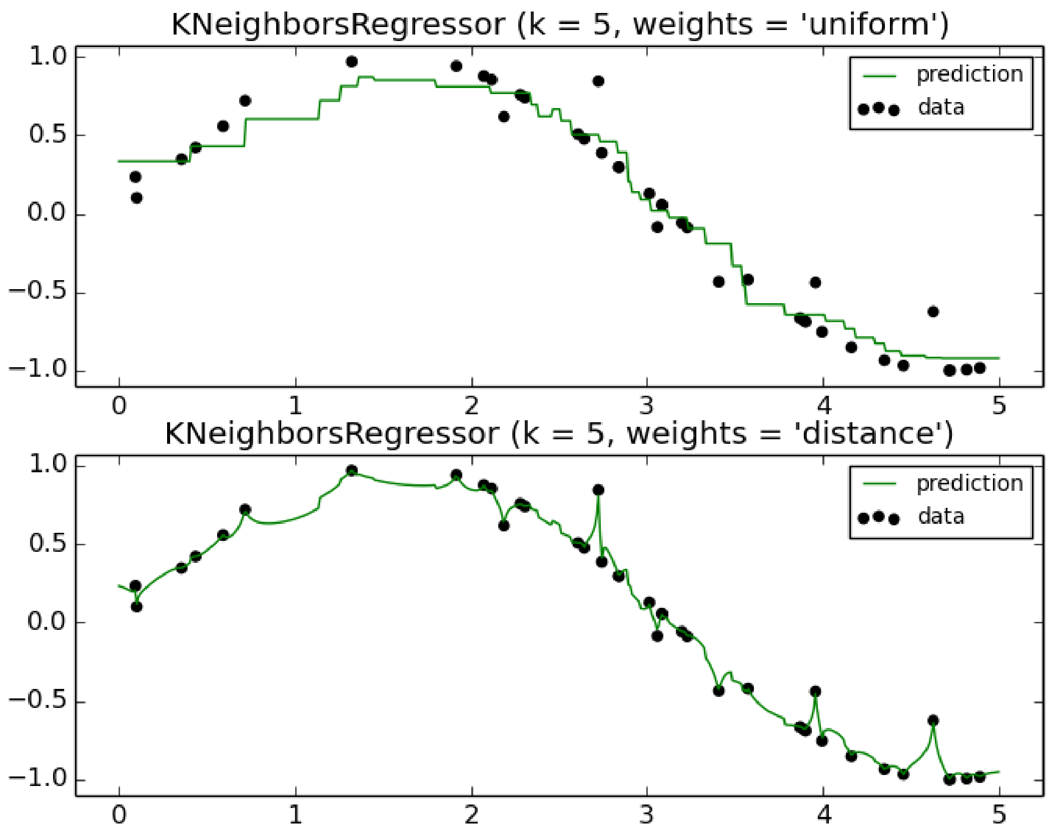


Figure 11.155: *Nearest Neighbors regression*

11.19.1 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.



Python source code: `plot_regression.py`

```
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, c='k', label='data')
    plt.plot(T, y_, c='g', label='prediction')
    plt.axis('tight')
    plt.legend()
    plt.title("KNeighborsRegressor (k = %i, weights = '%s')" % (n_neighbors,
                                                                weights))

plt.show()
```

Total running time of the example: 0.12 seconds (0 minutes 0.12 seconds)

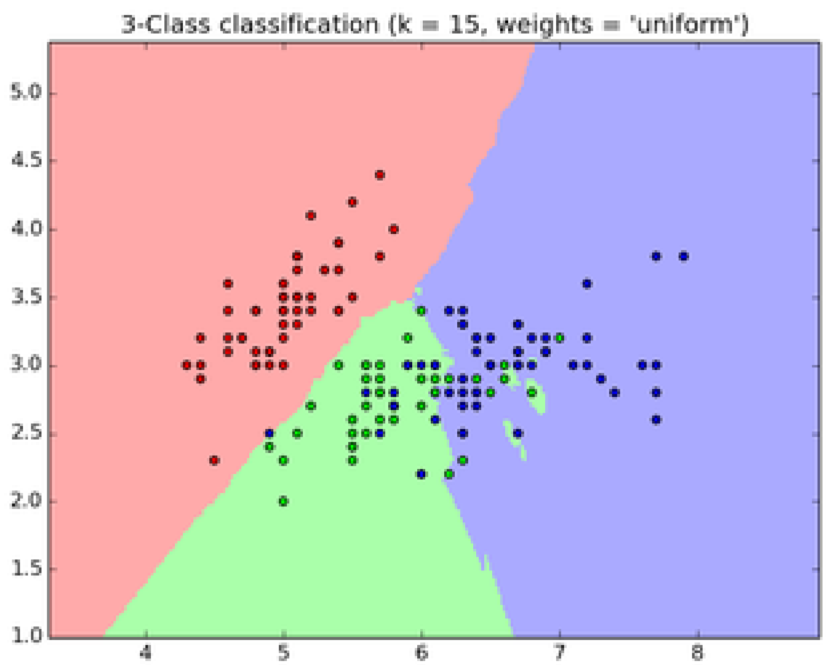
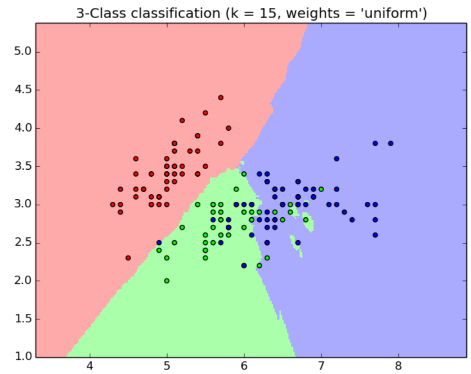
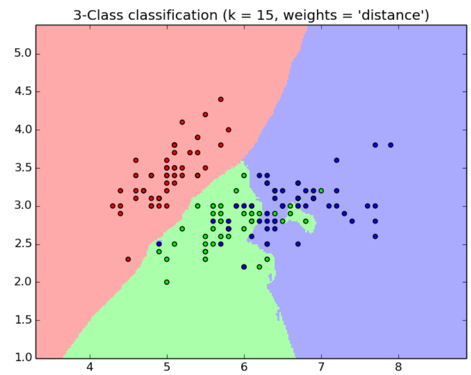


Figure 11.156: *Nearest Neighbors Classification*

11.19.2 Nearest Neighbors Classification

Sample usage of Nearest Neighbors classification. It will plot the decision boundaries for each class.

- 

- 

Python source code: `plot_classification.py`

```
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()
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

h = .02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

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)

    # Plot the decision boundary. For that, we will assign a color to each
```

```
    # point in the mesh [x_min, m_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)
    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 example: 0.36 seconds (0 minutes 0.36 seconds)

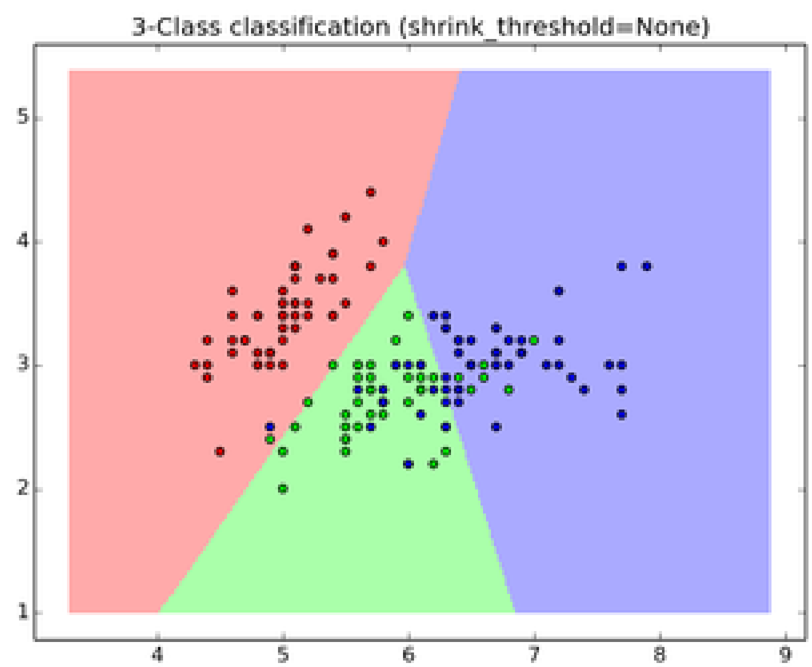
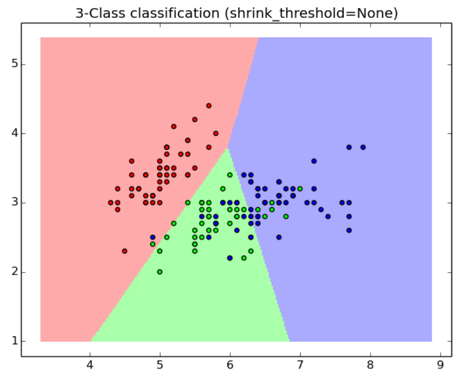
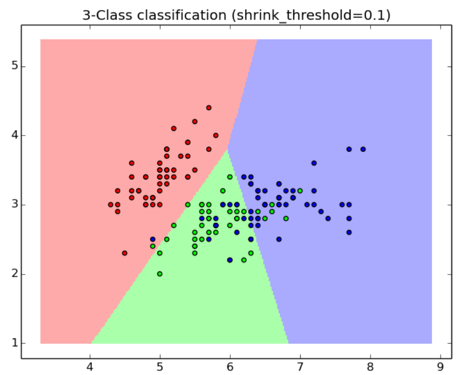


Figure 11.157: *Nearest Centroid Classification*

11.19.3 Nearest Centroid Classification

Sample usage of Nearest Centroid classification. It will plot the decision boundaries for each class.

- 

- 

Script output:

```
None 0.813333333333
0.1 0.813333333333
```

Python source code: `plot_nearest_centroid.py`

```python
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()
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

h = .02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])
```

```
for shrinkage in [None, 0.1]:
    # 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, m_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)
    plt.title("3-Class classification (shrink_threshold=%r)"
              % shrinkage)
    plt.axis('tight')

plt.show()
```

Total running time of the example: 0.13 seconds (0 minutes 0.13 seconds)

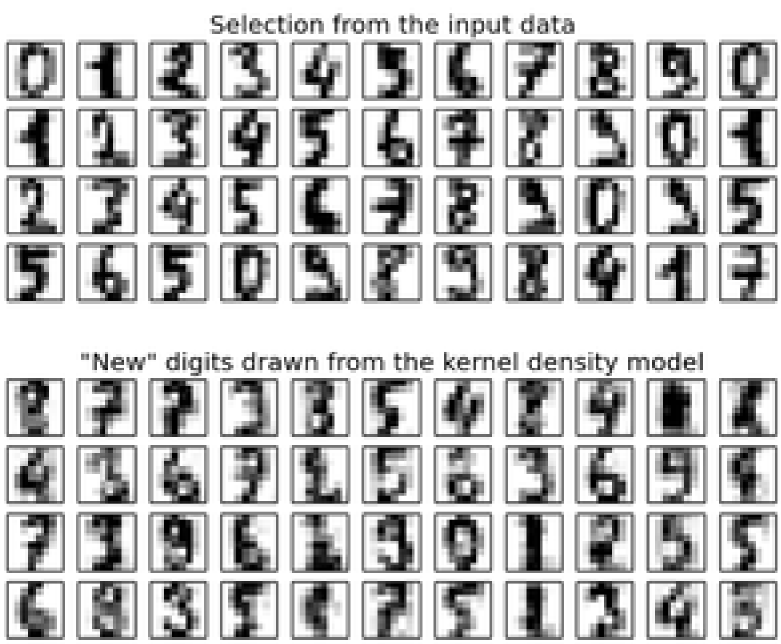
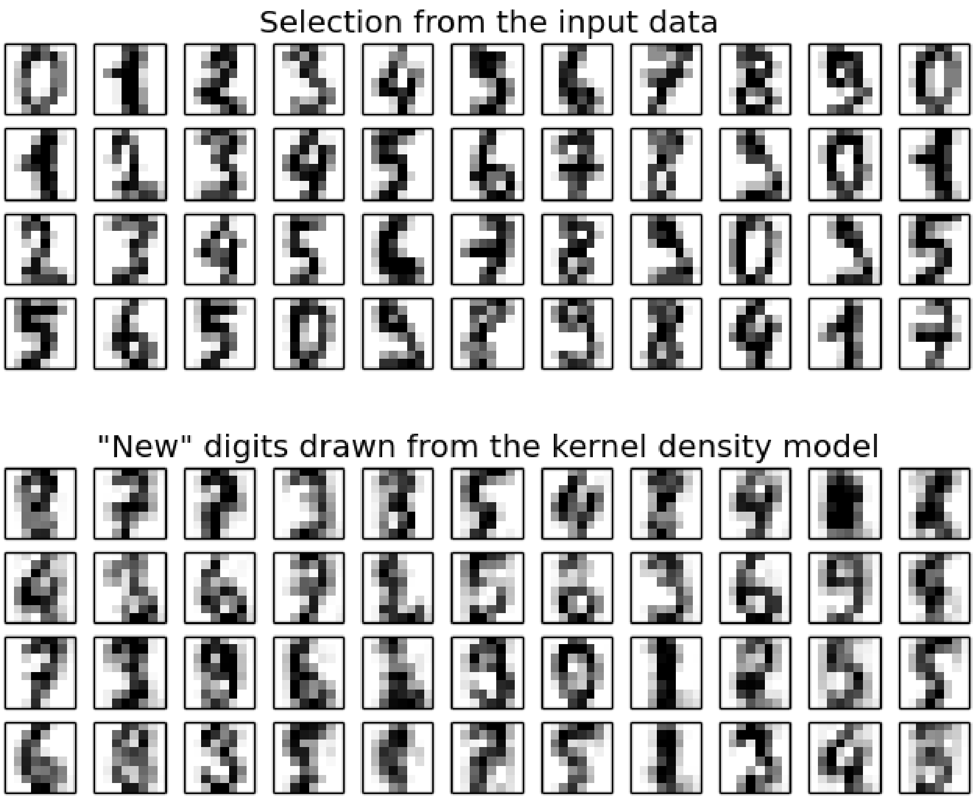


Figure 11.158: *Kernel Density Estimation*

11.19.4 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.



Script output:

```
best bandwidth: 3.79269019073
```

Python source code: `plot_digits_kde_sampling.py`

```python
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.grid_search import GridSearchCV

# load the data
digits = load_digits()
data = digits.data

# 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
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 example: 5.70 seconds (0 minutes 5.70 seconds)

11.19.5 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.

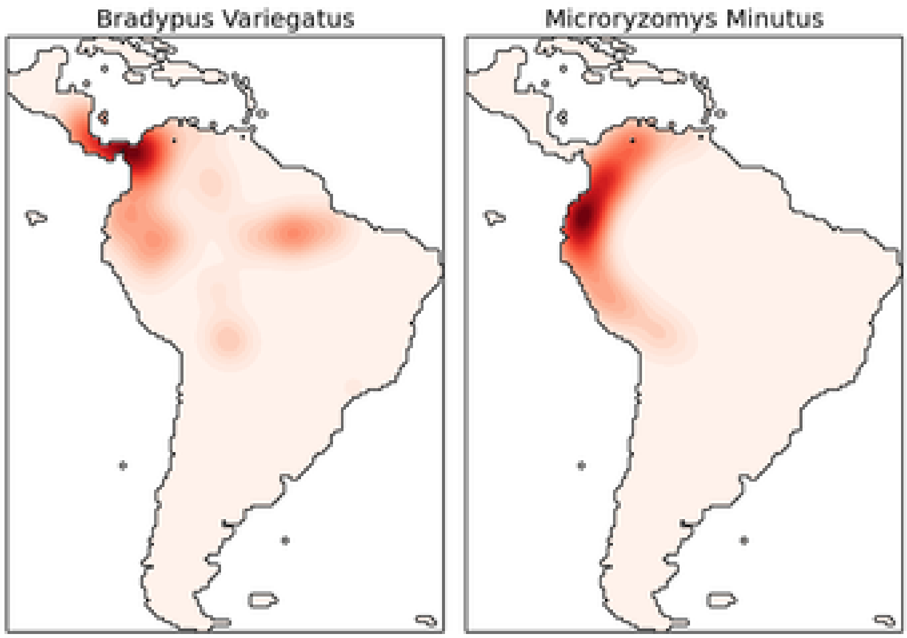
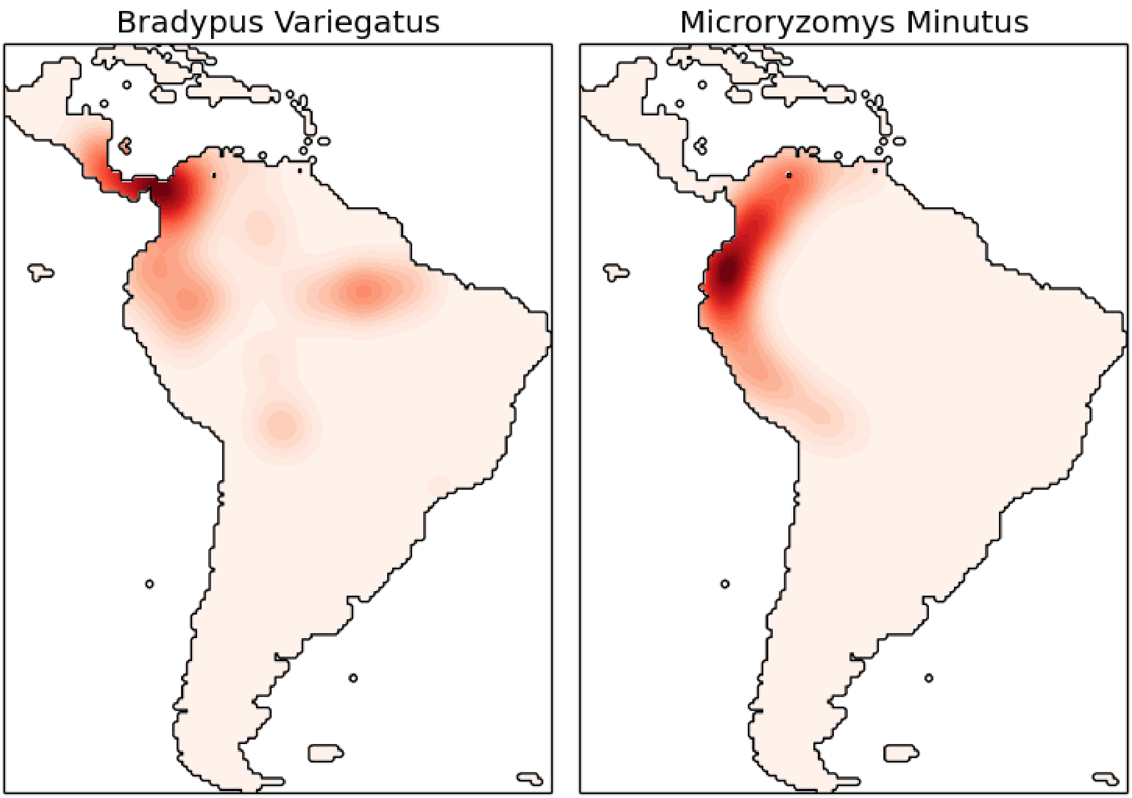


Figure 11.159: *Kernel Density Estimate of Species Distributions*

References

- "Maximum entropy modeling of species geographic distributions" S. J. Phillips, R. P. Anderson, R. E. Schapire - Ecological Modelling, 190:231-259, 2006.



Script output:

```
- computing KDE in spherical coordinates
 - plot coastlines from coverage
 - computing KDE in spherical coordinates
 - plot coastlines from coverage
```

Python source code: `plot_species_kde.py`

```python
# 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.datasets.species_distributions import construct_grids
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
```

```python
# Get matrices/arrays of species IDs and locations
data = fetch_species_distributions()
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 = -9999 + np.zeros(land_mask.shape[0])
    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=[-9999], colors="k",
                    linestyles="solid")
        plt.xticks([])
        plt.yticks([])

    plt.title(species_names[i])
```

```
plt.show()
```

Total running time of the example: 5.81 seconds (0 minutes 5.81 seconds)

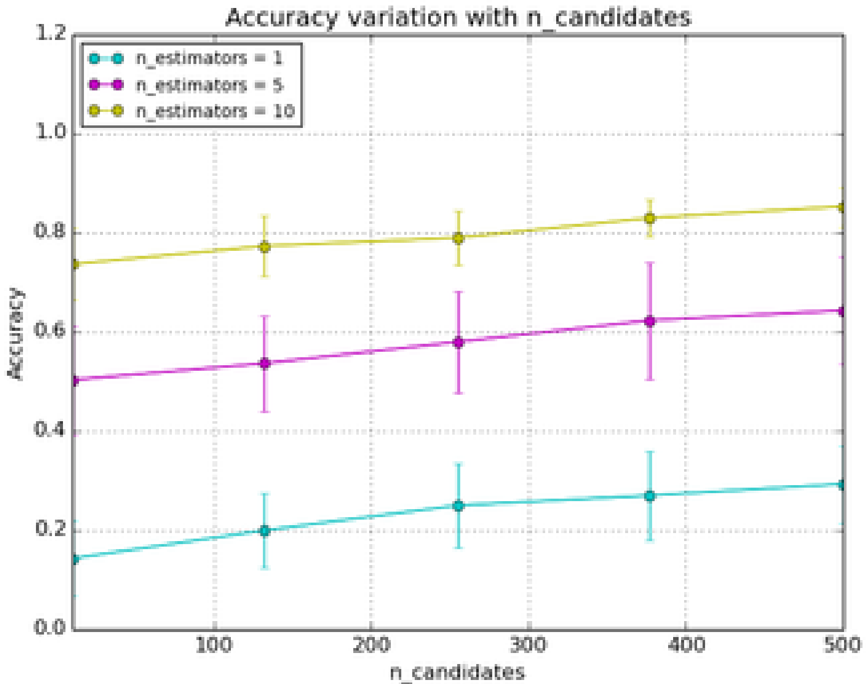


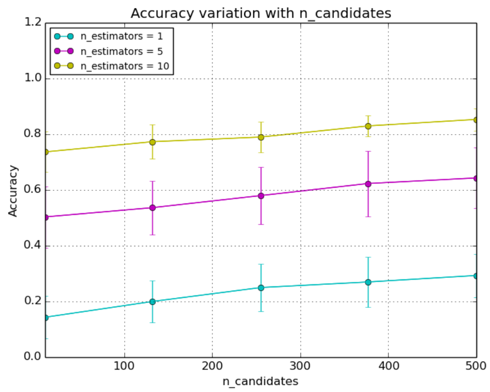
Figure 11.160: *Hyper-parameters of Approximate Nearest Neighbors*

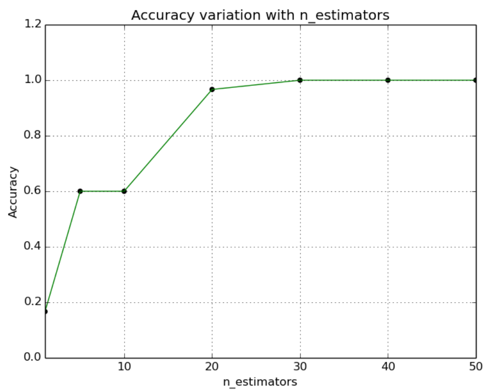
11.19.6 Hyper-parameters of Approximate Nearest Neighbors

This example demonstrates the behaviour of the accuracy of the nearest neighbor queries of Locality Sensitive Hashing Forest as the number of candidates and the number of estimators (trees) vary.

In the first plot, accuracy is measured with the number of candidates. Here, the term "number of candidates" refers to maximum bound for the number of distinct points retrieved from each tree to calculate the distances. Nearest neighbors are selected from this pool of candidates. Number of estimators is maintained at three fixed levels (1, 5, 10).

In the second plot, the number of candidates is fixed at 50. Number of trees is varied and the accuracy is plotted against those values. To measure the accuracy, the true nearest neighbors are required, therefore `sklearn.neighbors.NearestNeighbors` is used to compute the exact neighbors.

- 

- 

Python source code: `plot_approximate_nearest_neighbors_hyperparameters.py`

```python
from __future__ import division
print(__doc__)

# Author: Maheshakya Wijewardena <maheshakya.10@cse.mrt.ac.lk>
#
# License: BSD 3 clause


###############################################################################
import numpy as np
from sklearn.datasets.samples_generator import make_blobs
from sklearn.neighbors import LSHForest
from sklearn.neighbors import NearestNeighbors
import matplotlib.pyplot as plt


# Initialize size of the database, iterations and required neighbors.
n_samples = 10000
n_features = 100
n_queries = 30
rng = np.random.RandomState(42)

# Generate sample data
X, _ = make_blobs(n_samples=n_samples + n_queries,
                  n_features=n_features, centers=10,
                  random_state=0)
X_index = X[:n_samples]
```

```
X_query = X[n_samples:]
# Get exact neighbors
nbrs = NearestNeighbors(n_neighbors=1, algorithm='brute',
                        metric='cosine').fit(X_index)
neighbors_exact = nbrs.kneighbors(X_query, return_distance=False)

# Set `n_candidate` values
n_candidates_values = np.linspace(10, 500, 5).astype(np.int)
n_estimators_for_candidate_value = [1, 5, 10]
n_iter = 10
stds_accuracies = np.zeros((len(n_estimators_for_candidate_value),
                            n_candidates_values.shape[0]),
                           dtype=float)
accuracies_c = np.zeros((len(n_estimators_for_candidate_value),
                         n_candidates_values.shape[0]), dtype=float)

# LSH Forest is a stochastic index: perform several iteration to estimate
# expected accuracy and standard deviation displayed as error bars in
# the plots
for j, value in enumerate(n_estimators_for_candidate_value):
    for i, n_candidates in enumerate(n_candidates_values):
        accuracy_c = []
        for seed in range(n_iter):
            lshf = LSHForest(n_estimators=value,
                             n_candidates=n_candidates, n_neighbors=1,
                             random_state=seed)
            # Build the LSH Forest index
            lshf.fit(X_index)
            # Get neighbors
            neighbors_approx = lshf.kneighbors(X_query,
                                               return_distance=False)
            accuracy_c.append(np.sum(np.equal(neighbors_approx,
                                              neighbors_exact)) /
                              n_queries)

        stds_accuracies[j, i] = np.std(accuracy_c)
        accuracies_c[j, i] = np.mean(accuracy_c)

# Set `n_estimators` values
n_estimators_values = [1, 5, 10, 20, 30, 40, 50]
accuracies_trees = np.zeros(len(n_estimators_values), dtype=float)

# Calculate average accuracy for each value of `n_estimators`
for i, n_estimators in enumerate(n_estimators_values):
    lshf = LSHForest(n_estimators=n_estimators, n_neighbors=1)
    # Build the LSH Forest index
    lshf.fit(X_index)
    # Get neighbors
    neighbors_approx = lshf.kneighbors(X_query, return_distance=False)
    accuracies_trees[i] = np.sum(np.equal(neighbors_approx,
                                          neighbors_exact))/n_queries

###############################################################################
# Plot the accuracy variation with `n_candidates`
plt.figure()
colors = ['c', 'm', 'y']
for i, n_estimators in enumerate(n_estimators_for_candidate_value):
    label = 'n_estimators = %d ' % n_estimators
```

```
    plt.plot(n_candidates_values, accuracies_c[i, :],
             'o-', c=colors[i], label=label)
    plt.errorbar(n_candidates_values, accuracies_c[i, :],
                 stds_accuracies[i, :], c=colors[i])

plt.legend(loc='upper left', fontsize='small')
plt.ylim([0, 1.2])
plt.xlim(min(n_candidates_values), max(n_candidates_values))
plt.ylabel("Accuracy")
plt.xlabel("n_candidates")
plt.grid(which='both')
plt.title("Accuracy variation with n_candidates")

# Plot the accuracy variation with `n_estimators`
plt.figure()
plt.scatter(n_estimators_values, accuracies_trees, c='k')
plt.plot(n_estimators_values, accuracies_trees, c='g')
plt.ylim([0, 1.2])
plt.xlim(min(n_estimators_values), max(n_estimators_values))
plt.ylabel("Accuracy")
plt.xlabel("n_estimators")
plt.grid(which='both')
plt.title("Accuracy variation with n_estimators")

plt.show()
```

Total running time of the example: 39.33 seconds (0 minutes 39.33 seconds)

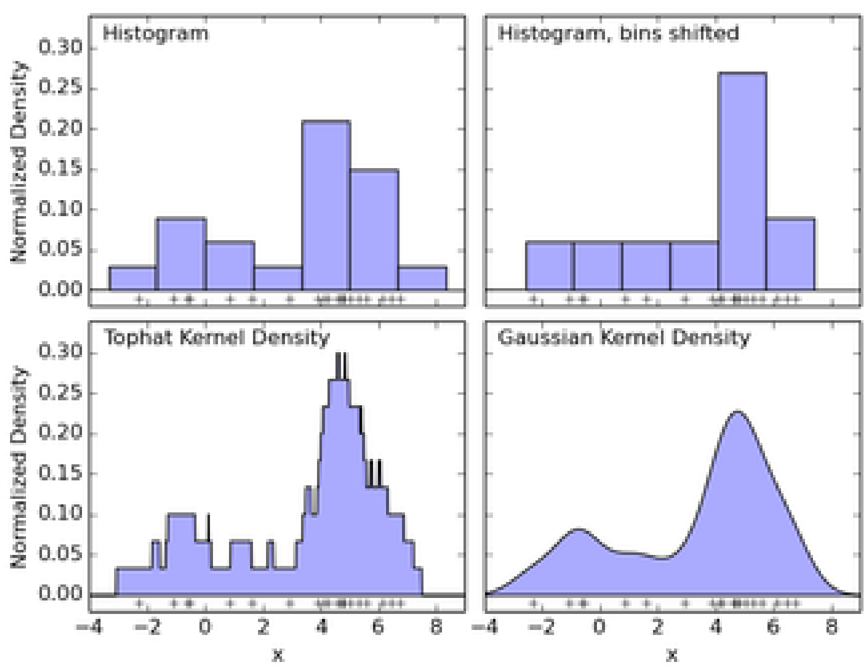


Figure 11.161: *Simple 1D Kernel Density Estimation*

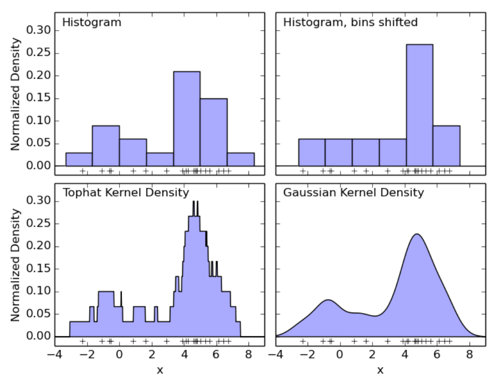
11.19.7 Simple 1D Kernel Density Estimation

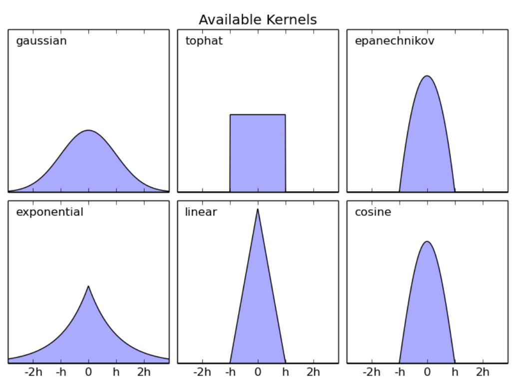
This example uses the sklearn.neighbors.KernelDensity class to demonstrate the principles of Kernel Density Estimation in one dimension.

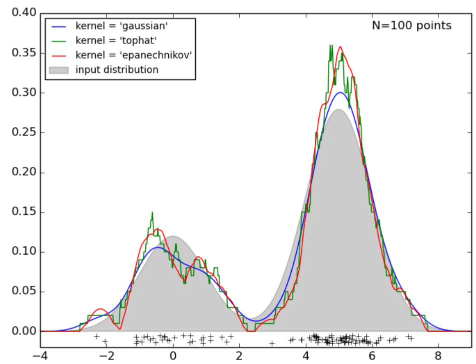
The first plot shows one of the problems with using histograms to visualize the density of points in 1D. Intuitively, a histogram can be thought of as a scheme in which a unit "block" is stacked above each point on a regular grid. As the top two panels show, however, the choice of gridding for these 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.

- 

- 

- 

Python source code: `plot_kde_1d.py`

```
# Author: Jake Vanderplas <jakevdp@cs.washington.edu>
#
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from sklearn.neighbors import KernelDensity


#----------------------------------------------------------------------
# Plot the progression of histograms to kernels
np.random.seed(1)
N = 20
X = np.concatenate((np.random.normal(0, 1, 0.3 * N),
                    np.random.normal(5, 1, 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)

# histogram 1
ax[0, 0].hist(X[:, 0], bins=bins, fc='#AAAAFF', normed=True)
ax[0, 0].text(-3.5, 0.31, "Histogram")

# histogram 2
ax[0, 1].hist(X[:, 0], bins=bins + 0.75, fc='#AAAAFF', normed=True)
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.zeros(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)

    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, 0.3 * N),
                    np.random.normal(5, 1, 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')

for kernel in ['gaussian', 'tophat', 'epanechnikov']:
    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), '-',
            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 example: 0.40 seconds (0 minutes 0.40 seconds)

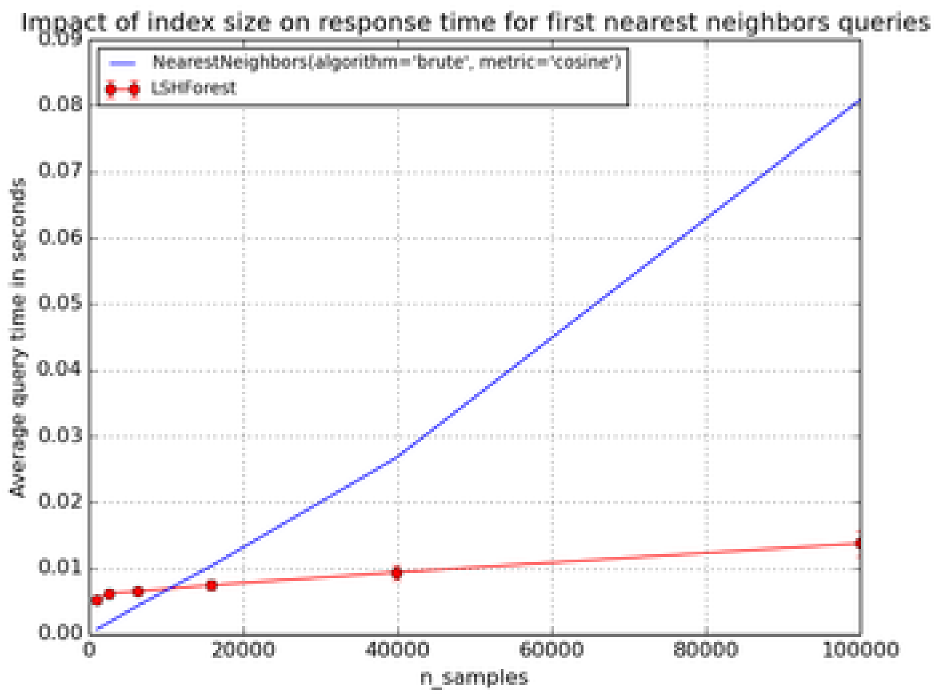


Figure 11.162: *Scalability of Approximate Nearest Neighbors*

11.19.8 Scalability of Approximate Nearest Neighbors

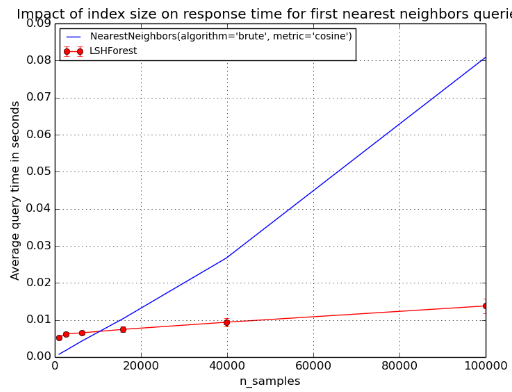
This example studies the scalability profile of approximate 10-neighbors queries using the LSHForest with `n_estimators=20` and `n_candidates=200` when varying the number of samples in the dataset.

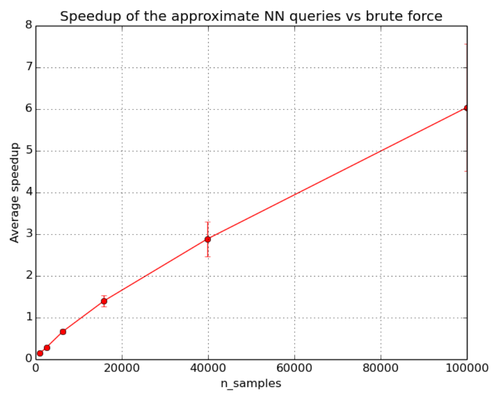
The first plot demonstrates the relationship between query time and index size of LSHForest. Query time is compared with the brute force method in exact nearest neighbor search for the same index sizes. The brute force queries have a very predictable linear scalability with the index (full scan). LSHForest index have sub-linear scalability profile but can be slower for small datasets.

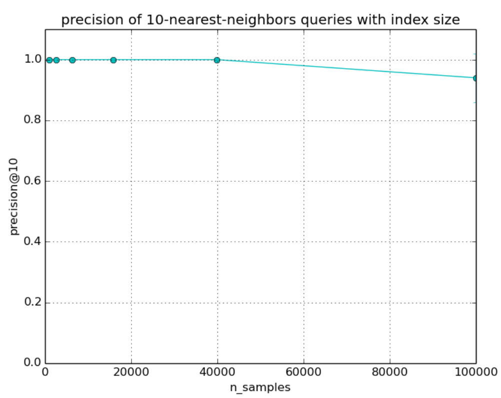
The second plot shows the speedup when using approximate queries vs brute force exact queries. The speedup tends to increase with the dataset size but should reach a plateau typically when doing queries on datasets with millions of samples and a few hundreds of dimensions. Higher dimensional datasets tends to benefit more from LSHForest indexing.

The break even point (speedup = 1) depends on the dimensionality and structure of the indexed data and the parameters of the LSHForest index.

The precision of approximate queries should decrease slowly with the dataset size. The speed of the decrease depends mostly on the LSHForest parameters and the dimensionality of the data.

- 

- 

- 

Script output:

```
Index size: 1000, exact: 0.001s, LSHF: 0.005s, speedup: 0.1, accuracy: 1.00 +/-0.00
Index size: 2511, exact: 0.002s, LSHF: 0.006s, speedup: 0.3, accuracy: 1.00 +/-0.00
Index size: 6309, exact: 0.004s, LSHF: 0.007s, speedup: 0.7, accuracy: 1.00 +/-0.00
Index size: 15848, exact: 0.010s, LSHF: 0.007s, speedup: 1.4, accuracy: 1.00 +/-0.00
Index size: 39810, exact: 0.027s, LSHF: 0.009s, speedup: 2.9, accuracy: 1.00 +/-0.00
Index size: 100000, exact: 0.081s, LSHF: 0.014s, speedup: 6.0, accuracy: 0.94 +/-0.08
```

Python source code: `plot_approximate_nearest_neighbors_scalability.py`

```python
from __future__ import division
print(__doc__)

# Authors: Maheshakya Wijewardena <maheshakya.10@cse.mrt.ac.lk>
#          Olivier Grisel <olivier.grisel@ensta.org>
#
# License: BSD 3 clause


###############################################################################
import time
import numpy as np
from sklearn.datasets.samples_generator import make_blobs
from sklearn.neighbors import LSHForest
from sklearn.neighbors import NearestNeighbors
import matplotlib.pyplot as plt

# Parameters of the study
n_samples_min = int(1e3)
n_samples_max = int(1e5)
n_features = 100
n_centers = 100
n_queries = 100
n_steps = 6
n_iter = 5

# Initialize the range of `n_samples`
n_samples_values = np.logspace(np.log10(n_samples_min),
                               np.log10(n_samples_max),
                               n_steps).astype(np.int)

# Generate some structured data
rng = np.random.RandomState(42)
all_data, _ = make_blobs(n_samples=n_samples_max + n_queries,
                         n_features=n_features, centers=n_centers, shuffle=True,
                         random_state=0)
queries = all_data[:n_queries]
index_data = all_data[n_queries:]

# Metrics to collect for the plots
average_times_exact = []
average_times_approx = []
std_times_approx = []
accuracies = []
std_accuracies = []
average_speedups = []
std_speedups = []

# Calculate the average query time
```

```
for n_samples in n_samples_values:
    X = index_data[:n_samples]
    # Initialize LSHForest for queries of a single neighbor
    lshf = LSHForest(n_estimators=20, n_candidates=200,
                     n_neighbors=10).fit(X)
    nbrs = NearestNeighbors(algorithm='brute', metric='cosine',
                            n_neighbors=10).fit(X)
    time_approx = []
    time_exact = []
    accuracy = []

    for i in range(n_iter):
        # pick one query at random to study query time variability in LSHForest
        query = queries[rng.randint(0, n_queries)]

        t0 = time.time()
        exact_neighbors = nbrs.kneighbors(query, return_distance=False)
        time_exact.append(time.time() - t0)

        t0 = time.time()
        approx_neighbors = lshf.kneighbors(query, return_distance=False)
        time_approx.append(time.time() - t0)

        accuracy.append(np.in1d(approx_neighbors, exact_neighbors).mean())

    average_time_exact = np.mean(time_exact)
    average_time_approx = np.mean(time_approx)
    speedup = np.array(time_exact) / np.array(time_approx)
    average_speedup = np.mean(speedup)
    mean_accuracy = np.mean(accuracy)
    std_accuracy = np.std(accuracy)
    print("Index size: %d, exact: %0.3fs, LSHF: %0.3fs, speedup: %0.1f, "
          "accuracy: %0.2f +/-%0.2f" %
          (n_samples, average_time_exact, average_time_approx, average_speedup,
           mean_accuracy, std_accuracy))

    accuracies.append(mean_accuracy)
    std_accuracies.append(std_accuracy)
    average_times_exact.append(average_time_exact)
    average_times_approx.append(average_time_approx)
    std_times_approx.append(np.std(time_approx))
    average_speedups.append(average_speedup)
    std_speedups.append(np.std(speedup))

# Plot average query time against n_samples
plt.figure()
plt.errorbar(n_samples_values, average_times_approx, yerr=std_times_approx,
             fmt='o-', c='r', label='LSHForest')
plt.plot(n_samples_values, average_times_exact, c='b',
         label="NearestNeighbors(algorithm='brute', metric='cosine')")
plt.legend(loc='upper left', fontsize='small')
plt.ylim(0, None)
plt.ylabel("Average query time in seconds")
plt.xlabel("n_samples")
plt.grid(which='both')
plt.title("Impact of index size on response time for first "
          "nearest neighbors queries")
```

```
# Plot average query speedup versus index size
plt.figure()
plt.errorbar(n_samples_values, average_speedups, yerr=std_speedups,
             fmt='o-', c='r')
plt.ylim(0, None)
plt.ylabel("Average speedup")
plt.xlabel("n_samples")
plt.grid(which='both')
plt.title("Speedup of the approximate NN queries vs brute force")

# Plot average precision versus index size
plt.figure()
plt.errorbar(n_samples_values, accuracies, std_accuracies, fmt='o-', c='c')
plt.ylim(0, 1.1)
plt.ylabel("precision@10")
plt.xlabel("n_samples")
plt.grid(which='both')
plt.title("precision of 10-nearest-neighbors queries with index size")

plt.show()
```

Total running time of the example: 13.29 seconds (0 minutes 13.29 seconds)

11.20 Neural Networks

Examples concerning the `sklearn.neural_network` module.

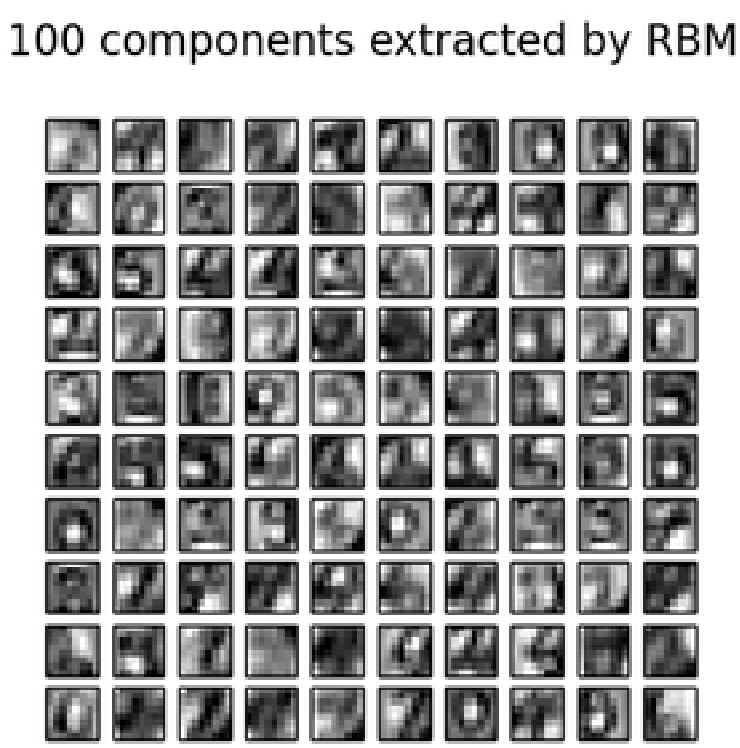


Figure 11.163: *Restricted Boltzmann Machine features for digit classification*

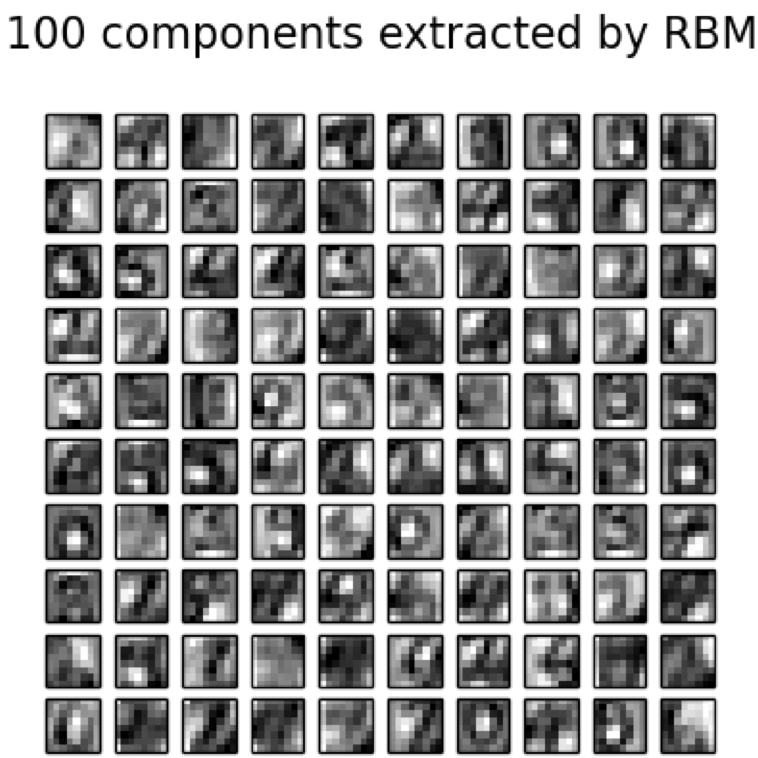
11.20.1 Restricted Boltzmann Machine features for digit classification

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.



Script output:

```
[BernoulliRBM] Iteration 1, pseudo-likelihood = -25.39, time = 0.38s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -23.77, time = 0.55s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -22.94, time = 0.55s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -21.91, time = 0.55s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -21.69, time = 0.55s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -21.06, time = 0.55s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -20.89, time = 0.55s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -20.64, time = 0.54s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -20.36, time = 0.55s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -20.09, time = 0.55s
[BernoulliRBM] Iteration 11, pseudo-likelihood = -20.08, time = 0.54s
[BernoulliRBM] Iteration 12, pseudo-likelihood = -19.82, time = 0.54s
[BernoulliRBM] Iteration 13, pseudo-likelihood = -19.64, time = 0.54s
[BernoulliRBM] Iteration 14, pseudo-likelihood = -19.61, time = 0.55s
```

```
[BernoulliRBM] Iteration 15, pseudo-likelihood = -19.57, time = 0.56s
[BernoulliRBM] Iteration 16, pseudo-likelihood = -19.41, time = 0.56s
[BernoulliRBM] Iteration 17, pseudo-likelihood = -19.30, time = 0.58s
[BernoulliRBM] Iteration 18, pseudo-likelihood = -19.25, time = 0.56s
[BernoulliRBM] Iteration 19, pseudo-likelihood = -19.27, time = 0.56s
[BernoulliRBM] Iteration 20, pseudo-likelihood = -19.01, time = 0.56s

Logistic regression using RBM features:
             precision    recall  f1-score   support

          0       0.99      0.99      0.99       174
          1       0.92      0.95      0.93       184
          2       0.95      0.98      0.97       166
          3       0.97      0.91      0.94       194
          4       0.97      0.95      0.96       186
          5       0.93      0.93      0.93       181
          6       0.98      0.97      0.97       207
          7       0.95      1.00      0.97       154
          8       0.90      0.88      0.89       182
          9       0.91      0.93      0.92       169

avg / total       0.95      0.95      0.95      1797


Logistic regression using raw pixel features:
             precision    recall  f1-score   support

          0       0.85      0.94      0.89       174
          1       0.57      0.55      0.56       184
          2       0.72      0.85      0.78       166
          3       0.76      0.74      0.75       194
          4       0.85      0.82      0.84       186
          5       0.74      0.75      0.75       181
          6       0.93      0.88      0.91       207
          7       0.86      0.90      0.88       154
          8       0.68      0.55      0.61       182
          9       0.71      0.74      0.72       169

avg / total       0.77      0.77      0.77      1797
```

Python source code: `plot_rbm_logistic_classification.py`

```python
from __future__ import print_function

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.cross_validation import train_test_split
from sklearn.neural_network import BernoulliRBM
from sklearn.pipeline import Pipeline
```

```
###############################################################################
# Setting up

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]]]

    shift = lambda x, w: 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
digits = datasets.load_digits()
X = np.asarray(digits.data, 'float32')
X, Y = nudge_dataset(X, digits.target)
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()
rbm = BernoulliRBM(random_state=0, verbose=True)

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 = 20
# More components tend to give better prediction performance, but larger
```

```
# fitting time
rbm.n_components = 100
logistic.C = 6000.0

# Training RBM-Logistic Pipeline
classifier.fit(X_train, Y_train)

# Training Logistic regression
logistic_classifier = linear_model.LogisticRegression(C=100.0)
logistic_classifier.fit(X_train, Y_train)

###############################################################################
# Evaluation

print()
print("Logistic regression using RBM features:\n%s\n" % (
    metrics.classification_report(
        Y_test,
        classifier.predict(X_test))))

print("Logistic regression using raw pixel features:\n%s\n" % (
    metrics.classification_report(
        Y_test,
        logistic_classifier.predict(X_test))))

###############################################################################
# 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 example: 36.55 seconds (0 minutes 36.55 seconds)

11.21 Semi Supervised Classification

Examples concerning the `sklearn.semi_supervised` module.

11.21.1 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.

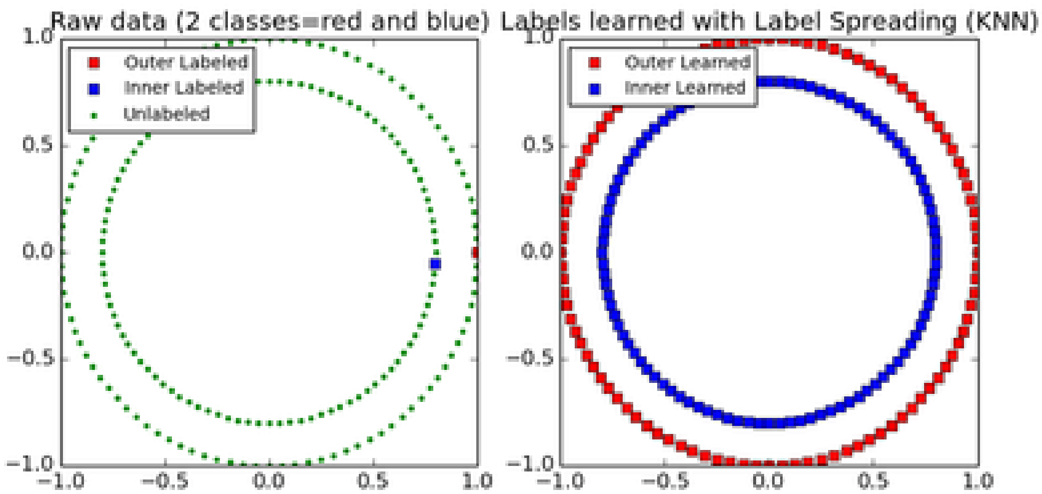
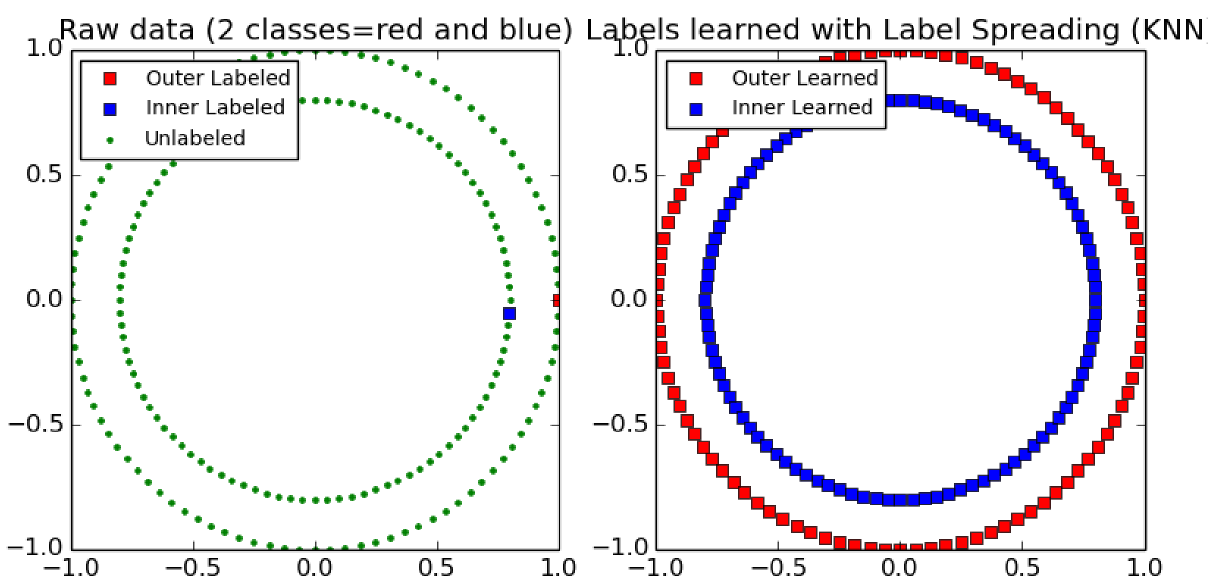


Figure 11.164: *Label Propagation learning a complex structure*



Python source code: `plot_label_propagation_structure.py`

```
print(__doc__)

# Authors: Clay Woolam <clay@woolam.org>
#          Andreas Mueller <amueller@ais.uni-bonn.de>
# Licence: BSD
```

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.semi_supervised import label_propagation
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.ones(n_samples)
labels[0] = outer
labels[-1] = inner

###############################################################################
# Learn with LabelSpreading
label_spread = label_propagation.LabelSpreading(kernel='knn', alpha=1.0)
label_spread.fit(X, labels)

###############################################################################
# Plot output labels
output_labels = label_spread.transduction_
plt.figure(figsize=(8.5, 4))
plt.subplot(1, 2, 1)
plot_outer_labeled, = plt.plot(X[labels == outer, 0],
                               X[labels == outer, 1], 'rs')
plot_unlabeled, = plt.plot(X[labels == -1, 0], X[labels == -1, 1], 'g.')
plot_inner_labeled, = plt.plot(X[labels == inner, 0],
                               X[labels == inner, 1], 'bs')
plt.legend((plot_outer_labeled, plot_inner_labeled, plot_unlabeled),
           ('Outer Labeled', 'Inner Labeled', 'Unlabeled'), 'upper left',
           numpoints=1, shadow=False)
plt.title("Raw data (2 classes=red and blue)")

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]
plot_outer, = plt.plot(X[outer_numbers, 0], X[outer_numbers, 1], 'rs')
plot_inner, = plt.plot(X[inner_numbers, 0], X[inner_numbers, 1], 'bs')
plt.legend((plot_outer, plot_inner), ('Outer Learned', 'Inner Learned'),
           'upper left', numpoints=1, shadow=False)
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 example: 0.28 seconds (0 minutes 0.28 seconds)

11.21.2 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.

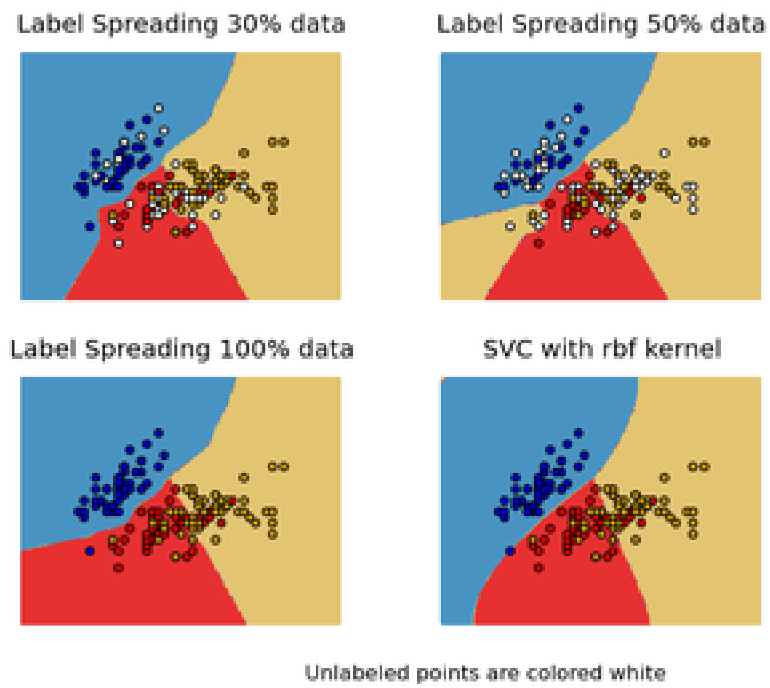
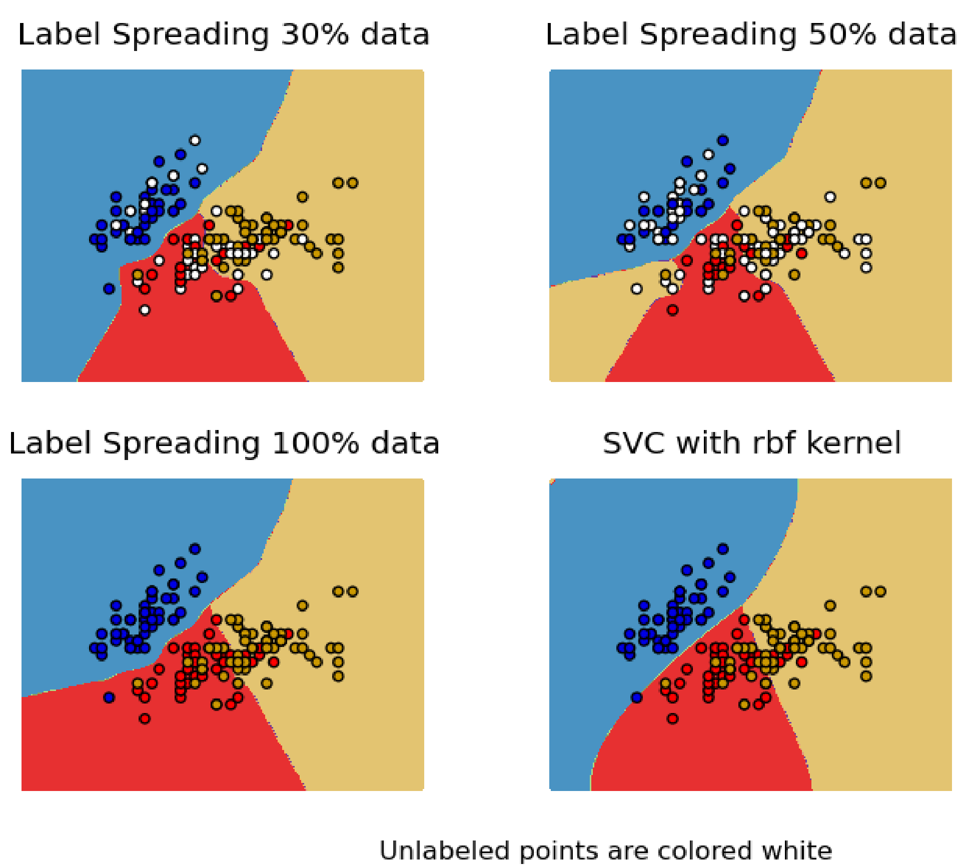


Figure 11.165: *Decision boundary of label propagation versus SVM on the Iris dataset*



Python source code: plot_label_propagation_versus_svm_iris.py

```
print(__doc__)

# Authors: Clay Woolam <clay@woolam.org>
# Licence: BSD

import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn import svm
from sklearn.semi_supervised import label_propagation

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
# 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 = (label_propagation.LabelSpreading().fit(X, y_30),
        y_30)
ls50 = (label_propagation.LabelSpreading().fit(X, y_50),
        y_50)
ls100 = (label_propagation.LabelSpreading().fit(X, y), y)
rbf_svc = (svm.SVC(kernel='rbf').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, m_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, cmap=plt.cm.Paired)

    plt.title(titles[i])

plt.text(.90, 0, "Unlabeled points are colored white")
plt.show()
```

Total running time of the example: 1.54 seconds (0 minutes 1.54 seconds)

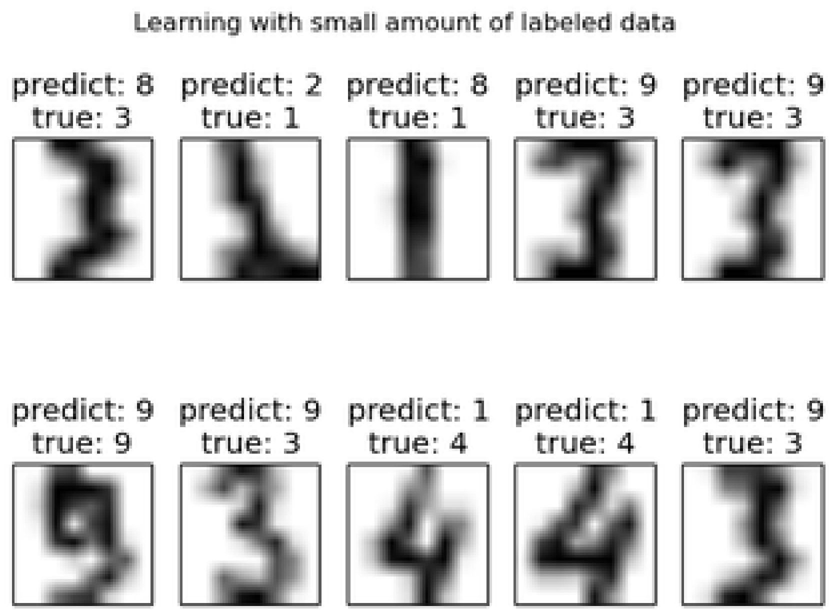


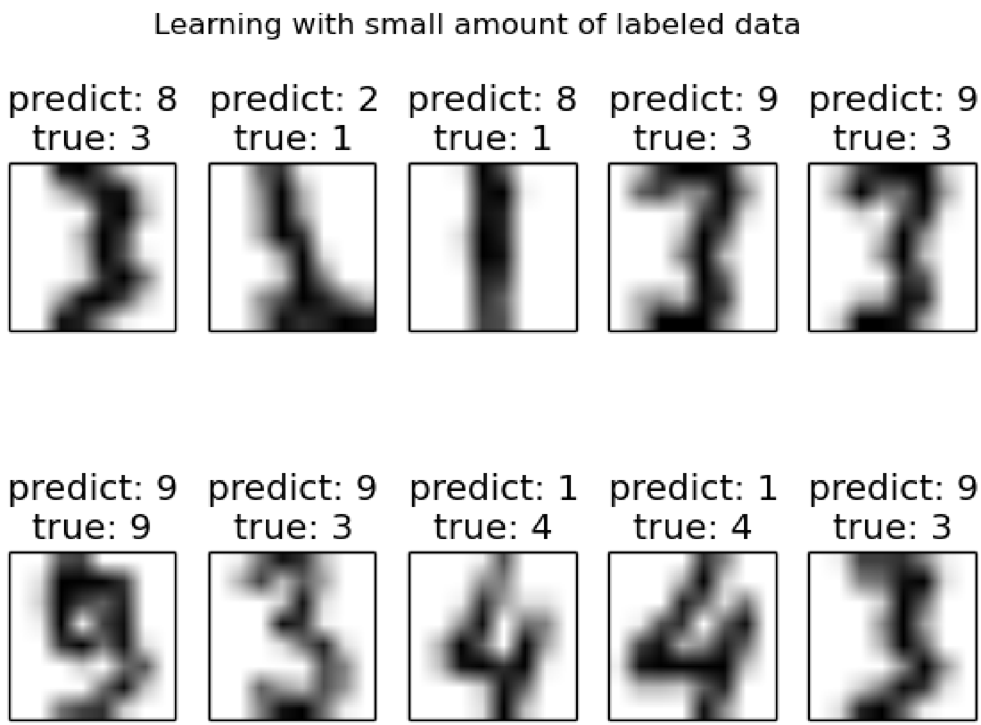
Figure 11.166: *Label Propagation digits: Demonstrating performance*

11.21.3 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.



Script output:

```
Label Spreading model: 30 labeled & 300 unlabeled points (330 total)
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        23
          1       0.58      0.54      0.56        28
          2       0.96      0.93      0.95        29
          3       0.00      0.00      0.00        28
          4       0.91      0.80      0.85        25
          5       0.96      0.79      0.87        33
          6       0.97      0.97      0.97        36
          7       0.89      1.00      0.94        34
          8       0.48      0.83      0.61        29
          9       0.54      0.77      0.64        35

avg / total       0.73      0.77      0.74       300

Confusion matrix
[[23  0  0  0  0  0  0  0  0]
 [ 0 15  1  0  0  1  0 11  0]
 [ 0  0 27  0  0  0  2  0  0]
 [ 0  5  0 20  0  0  0  0  0]
 [ 0  0  0  0 26  0  0  1  6]
 [ 0  1  0  0  0 35  0  0  0]
 [ 0  0  0  0  0  0 34  0  0]
 [ 0  5  0  0  0  0  0 24  0]
 [ 0  0  0  2  1  0  2  3 27]]
```

Python source code: `plot_label_propagation_digits.py`

```
print(__doc__)

# Authors: Clay Woolam <clay@woolam.org>
# Licence: BSD

import numpy as np
import matplotlib.pyplot as plt

from scipy import stats

from sklearn import datasets
from sklearn.semi_supervised import label_propagation

from sklearn.metrics import confusion_matrix, classification_report

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 = 30

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 = label_propagation.LabelSpreading(gamma=0.25, max_iter=5)
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)" %
      (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:]
```

```python
###############################################################################
# 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 example: 0.36 seconds (0 minutes 0.36 seconds)

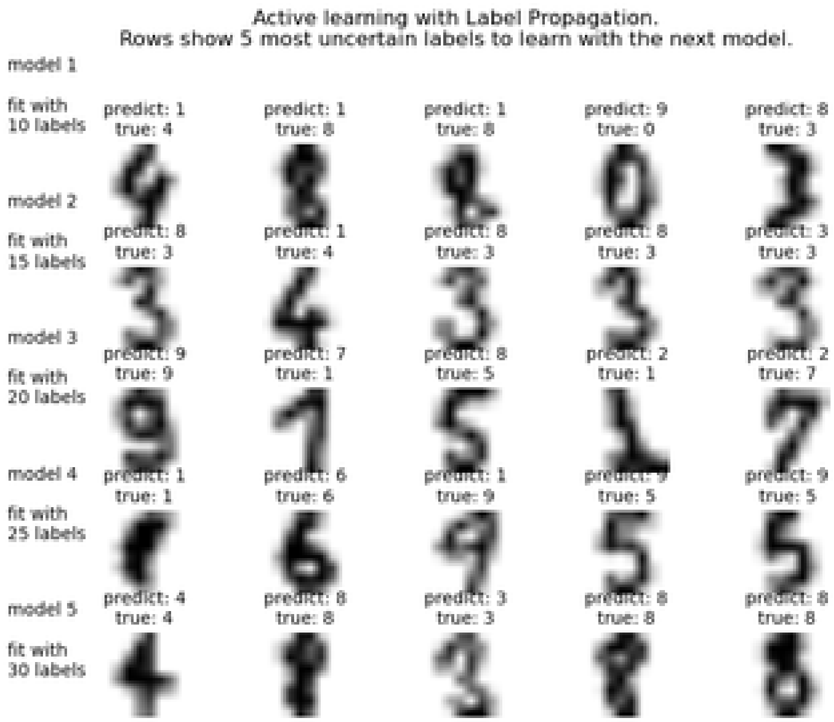


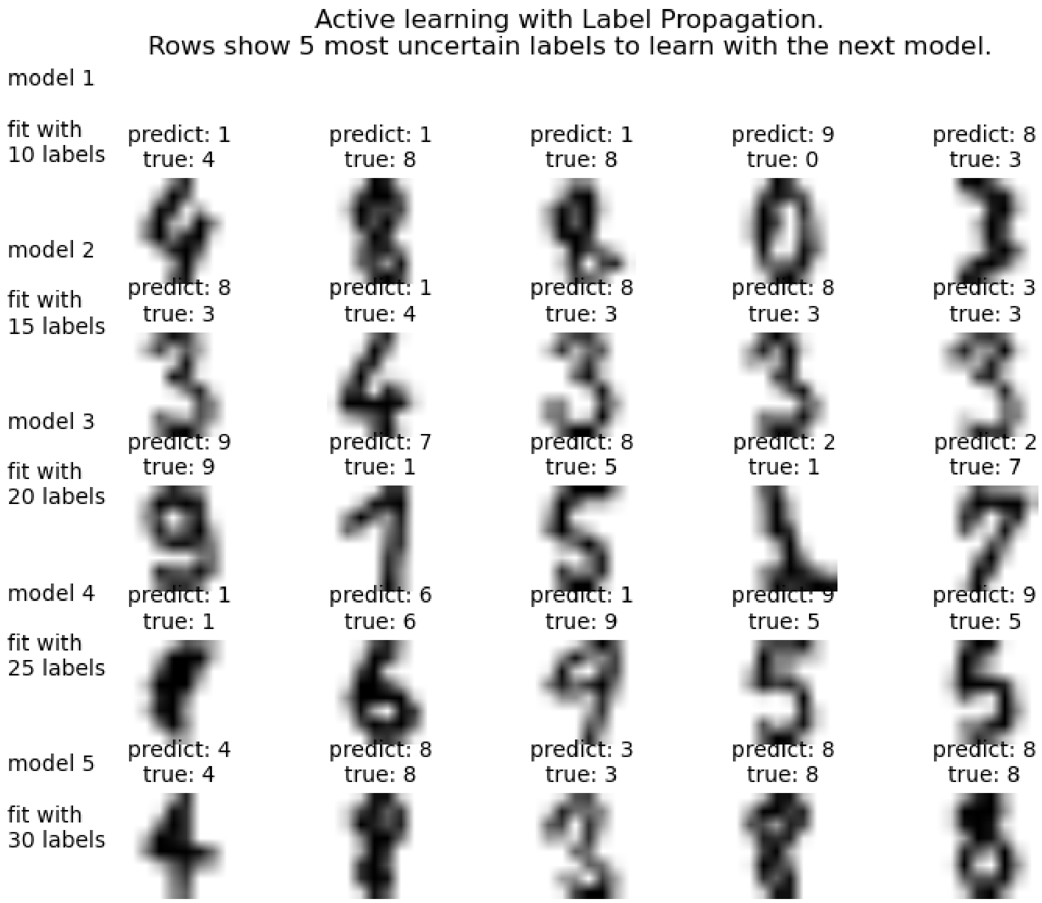
Figure 11.167: *Label Propagation digits active learning*

11.21.4 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.

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.



Script output:

```
Iteration 0 ______________________________________________________________________
Label Spreading model: 10 labeled & 320 unlabeled (330 total)
             precision    recall  f1-score   support

          0       0.00      0.00      0.00        24
          1       0.49      0.90      0.63        29
          2       0.88      0.97      0.92        31
          3       0.00      0.00      0.00        28
          4       0.00      0.00      0.00        27
          5       0.89      0.49      0.63        35
          6       0.86      0.95      0.90        40
          7       0.75      0.92      0.83        36
          8       0.54      0.79      0.64        33
          9       0.41      0.86      0.56        37

avg / total       0.52      0.63      0.55       320

Confusion matrix
[[26  1  0  0  1  0  1]
 [ 1 30  0  0  0  0  0]
 [ 0  0 17  6  0  2 10]
 [ 2  0  0 38  0  0  0]
 [ 0  3  0  0 33  0  0]
 [ 7  0  0  0  0 26  0]
```

```
 [ 0  0  2  0  0  3 32]]
Iteration 1 ______________________________________________________________________
Label Spreading model: 15 labeled & 315 unlabeled (330 total)
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        23
          1       0.61      0.59      0.60        29
          2       0.91      0.97      0.94        31
          3       1.00      0.56      0.71        27
          4       0.79      0.88      0.84        26
          5       0.89      0.46      0.60        35
          6       0.86      0.95      0.90        40
          7       0.97      0.92      0.94        36
          8       0.54      0.84      0.66        31
          9       0.70      0.81      0.75        37

avg / total       0.82      0.80      0.79       315

Confusion matrix
[[23  0  0  0  0  0  0  0  0  0]
 [ 0 17  1  0  2  0  0  1  7  1]
 [ 0  1 30  0  0  0  0  0  0  0]
 [ 0  0  0 15  0  0  0  0 10  2]
 [ 0  3  0  0 23  0  0  0  0  0]
 [ 0  0  0  0  1 16  6  0  2 10]
 [ 0  2  0  0  0  0 38  0  0  0]
 [ 0  0  2  0  1  0  0 33  0  0]
 [ 0  5  0  0  0  0  0  0 26  0]
 [ 0  0  0  0  2  2  0  0  3 30]]
Iteration 2 ______________________________________________________________________
Label Spreading model: 20 labeled & 310 unlabeled (330 total)
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        23
          1       0.68      0.59      0.63        29
          2       0.91      0.97      0.94        31
          3       0.96      1.00      0.98        23
          4       0.81      1.00      0.89        25
          5       0.89      0.46      0.60        35
          6       0.86      0.95      0.90        40
          7       0.97      0.92      0.94        36
          8       0.68      0.84      0.75        31
          9       0.75      0.81      0.78        37

avg / total       0.85      0.84      0.83       310

Confusion matrix
[[23  0  0  0  0  0  0  0  0  0]
 [ 0 17  1  0  2  0  0  1  7  1]
 [ 0  1 30  0  0  0  0  0  0  0]
 [ 0  0  0 23  0  0  0  0  0  0]
 [ 0  0  0  0 25  0  0  0  0  0]
 [ 0  0  0  1  1 16  6  0  2  9]
 [ 0  2  0  0  0  0 38  0  0  0]
 [ 0  0  2  0  1  0  0 33  0  0]
 [ 0  5  0  0  0  0  0  0 26  0]
 [ 0  0  0  0  2  2  0  0  3 30]]
Iteration 3 ______________________________________________________________________
```

```
Label Spreading model: 25 labeled & 305 unlabeled (330 total)
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        23
          1       0.70      0.85      0.77        27
          2       1.00      0.90      0.95        31
          3       1.00      1.00      1.00        23
          4       1.00      1.00      1.00        25
          5       0.96      0.74      0.83        34
          6       1.00      0.95      0.97        40
          7       0.90      1.00      0.95        35
          8       0.83      0.81      0.82        31
          9       0.75      0.83      0.79        36

avg / total       0.91      0.90      0.90       305

Confusion matrix
[[23  0  0  0  0  0  0  0  0  0]
 [ 0 23  0  0  0  0  0  0  4  0]
 [ 0  1 28  0  0  0  0  2  0  0]
 [ 0  0  0 23  0  0  0  0  0  0]
 [ 0  0  0  0 25  0  0  0  0  0]
 [ 0  0  0  0  0 25  0  0  0  9]
 [ 0  2  0  0  0  0 38  0  0  0]
 [ 0  0  0  0  0  0  0 35  0  0]
 [ 0  5  0  0  0  0  0  0 25  1]
 [ 0  2  0  0  0  1  0  2  1 30]]
Iteration 4 ______________________________________________________________________
Label Spreading model: 30 labeled & 300 unlabeled (330 total)
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        23
          1       0.77      0.88      0.82        26
          2       1.00      0.90      0.95        31
          3       1.00      1.00      1.00        23
          4       1.00      1.00      1.00        25
          5       0.94      0.97      0.95        32
          6       1.00      0.97      0.99        39
          7       0.90      1.00      0.95        35
          8       0.89      0.81      0.85        31
          9       0.94      0.89      0.91        35

avg / total       0.94      0.94      0.94       300

Confusion matrix
[[23  0  0  0  0  0  0  0  0  0]
 [ 0 23  0  0  0  0  0  0  3  0]
 [ 0  1 28  0  0  0  0  2  0  0]
 [ 0  0  0 23  0  0  0  0  0  0]
 [ 0  0  0  0 25  0  0  0  0  0]
 [ 0  0  0  0  0 31  0  0  0  1]
 [ 0  1  0  0  0  0 38  0  0  0]
 [ 0  0  0  0  0  0  0 35  0  0]
 [ 0  5  0  0  0  0  0  0 25  1]
 [ 0  0  0  0  0  2  0  2  0 31]]
```

Python source code: `plot_label_propagation_digits_active_learning.py`

```
print(__doc__)

# Authors: Clay Woolam <clay@woolam.org>
# Licence: BSD

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

from sklearn import datasets
from sklearn.semi_supervised import label_propagation
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 = 10

unlabeled_indices = np.arange(n_total_samples)[n_labeled_points:]
f = plt.figure()

for i in range(5):
    y_train = np.copy(y)
    y_train[unlabeled_indices] = -1

    lp_model = label_propagation.LabelSpreading(gamma=0.25, max_iter=5)
    lp_model.fit(X, y_train)

    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 five digit examples that the classifier is most uncertain about
    uncertainty_index = uncertainty_index = np.argsort(pred_entropies)[-5:]

    # keep track of indices that we get labels for
```

```
    delete_indices = np.array([])

    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]

        sub = f.add_subplot(5, 5, index + 1 + (5 * i))
        sub.imshow(image, cmap=plt.cm.gray_r)
        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 += 5

f.suptitle("Active learning with Label Propagation.\nRows show 5 most "
           "uncertain labels to learn with the next model.")
plt.subplots_adjust(0.12, 0.03, 0.9, 0.8, 0.2, 0.45)
plt.show()
```

Total running time of the example: 0.94 seconds (0 minutes 0.94 seconds)

11.22 Support Vector Machines

Examples concerning the `sklearn.svm` module.

11.22.1 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 learn by the SVC.

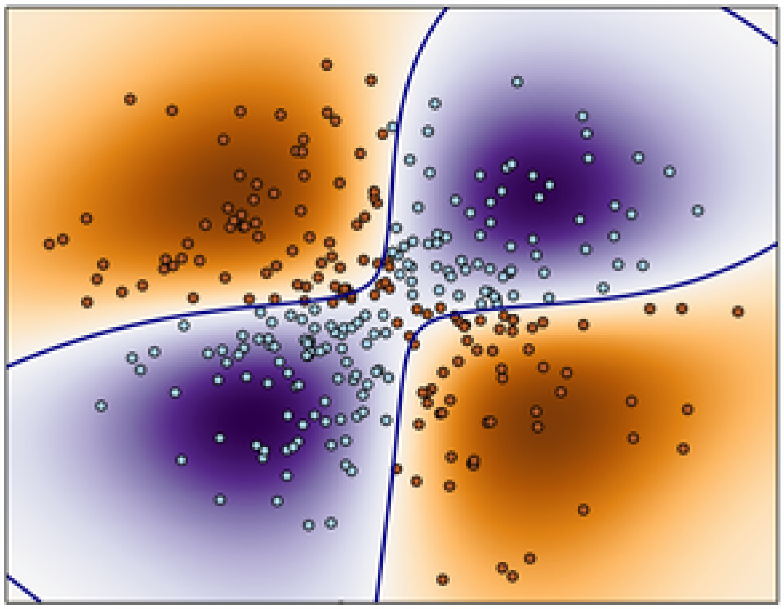
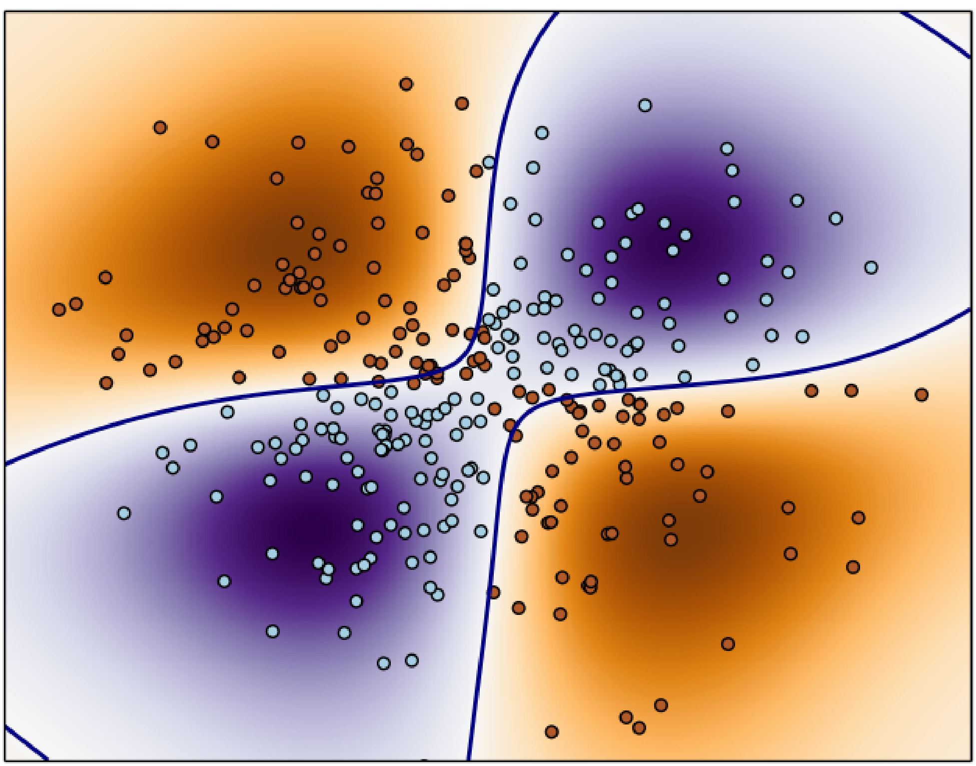


Figure 11.168: *Non-linear SVM*



Python source code: plot_svm_nonlinear.py

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

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()
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,
                       linetypes='--')
plt.scatter(X[:, 0], X[:, 1], s=30, c=Y, cmap=plt.cm.Paired)
plt.xticks(())
plt.yticks(())
plt.axis([-3, 3, -3, 3])
plt.show()
```

Total running time of the example: 1.13 seconds (0 minutes 1.13 seconds)

11.22.2 Support Vector Regression (SVR) using linear and non-linear kernels

Toy example of 1D regression using linear, polynomial and RBF kernels.

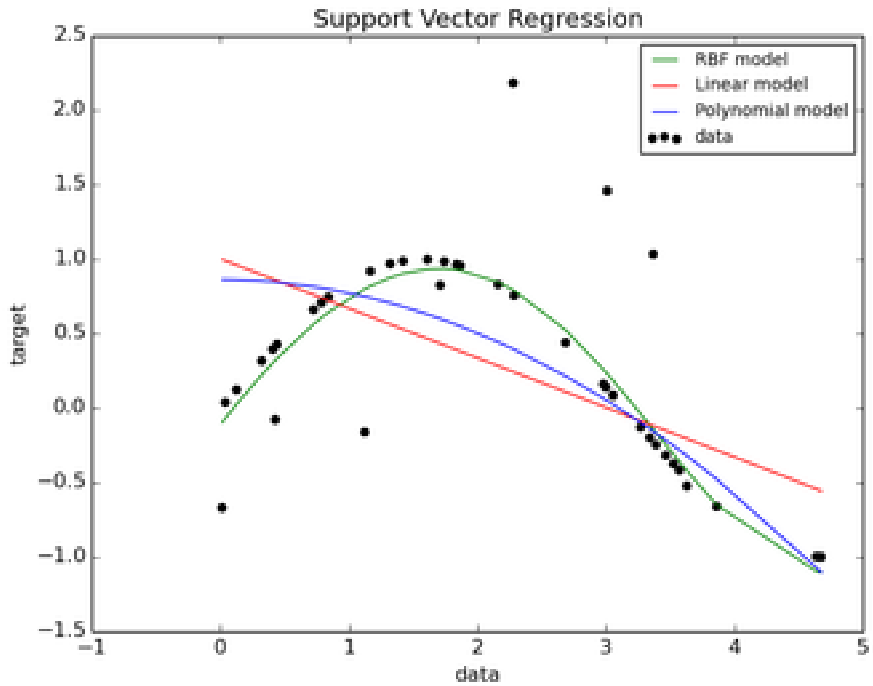
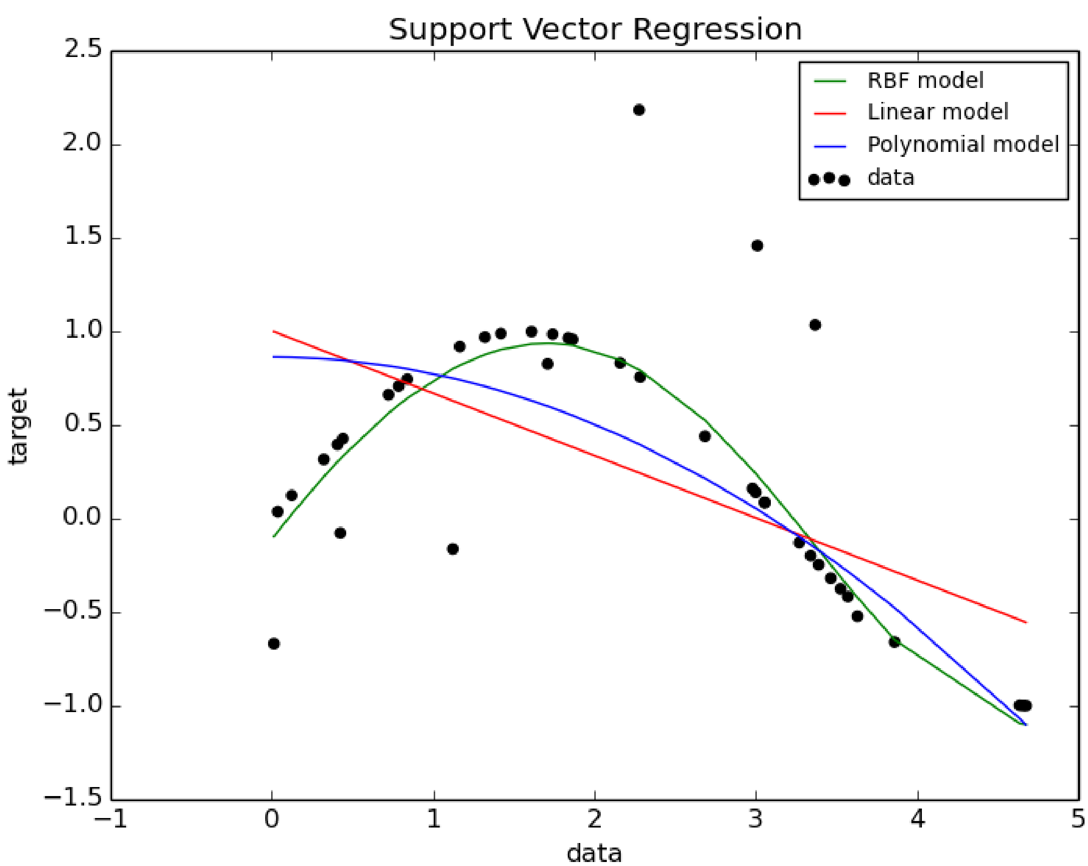


Figure 11.169: *Support Vector Regression (SVR) using linear and non-linear kernels*



Python source code: plot_svm_regression.py

```
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=1e3, gamma=0.1)
svr_lin = SVR(kernel='linear', C=1e3)
svr_poly = SVR(kernel='poly', C=1e3, degree=2)
y_rbf = svr_rbf.fit(X, y).predict(X)
y_lin = svr_lin.fit(X, y).predict(X)
y_poly = svr_poly.fit(X, y).predict(X)

###############################################################################
# look at the results
plt.scatter(X, y, c='k', label='data')
plt.hold('on')
plt.plot(X, y_rbf, c='g', label='RBF model')
plt.plot(X, y_lin, c='r', label='Linear model')
plt.plot(X, y_poly, c='b', label='Polynomial model')
plt.xlabel('data')
plt.ylabel('target')
plt.title('Support Vector Regression')
plt.legend()
plt.show()
```

Total running time of the example: 0.97 seconds (0 minutes 0.97 seconds)

11.22.3 SVM: Maximum margin separating hyperplane

Plot the maximum margin separating hyperplane within a two-class separable dataset using a Support Vector Machines classifier with linear kernel.

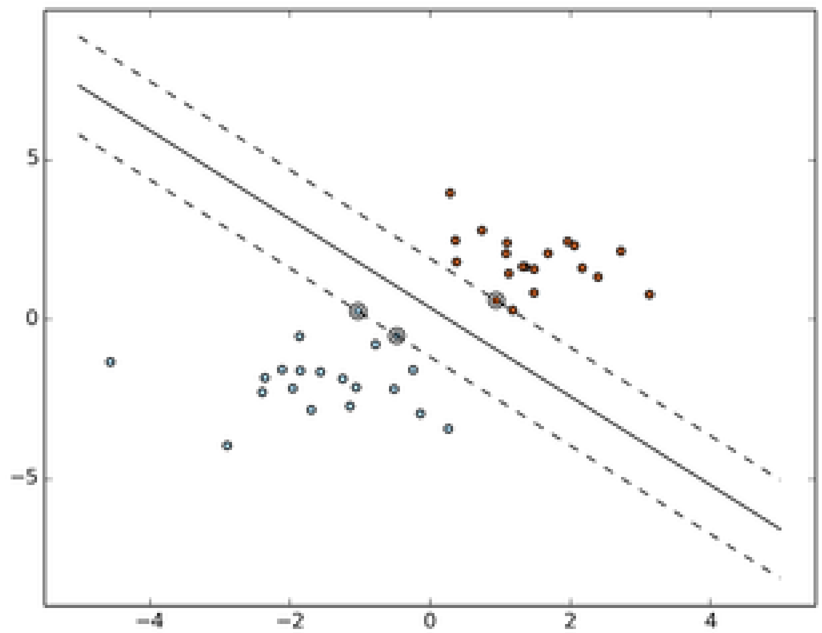
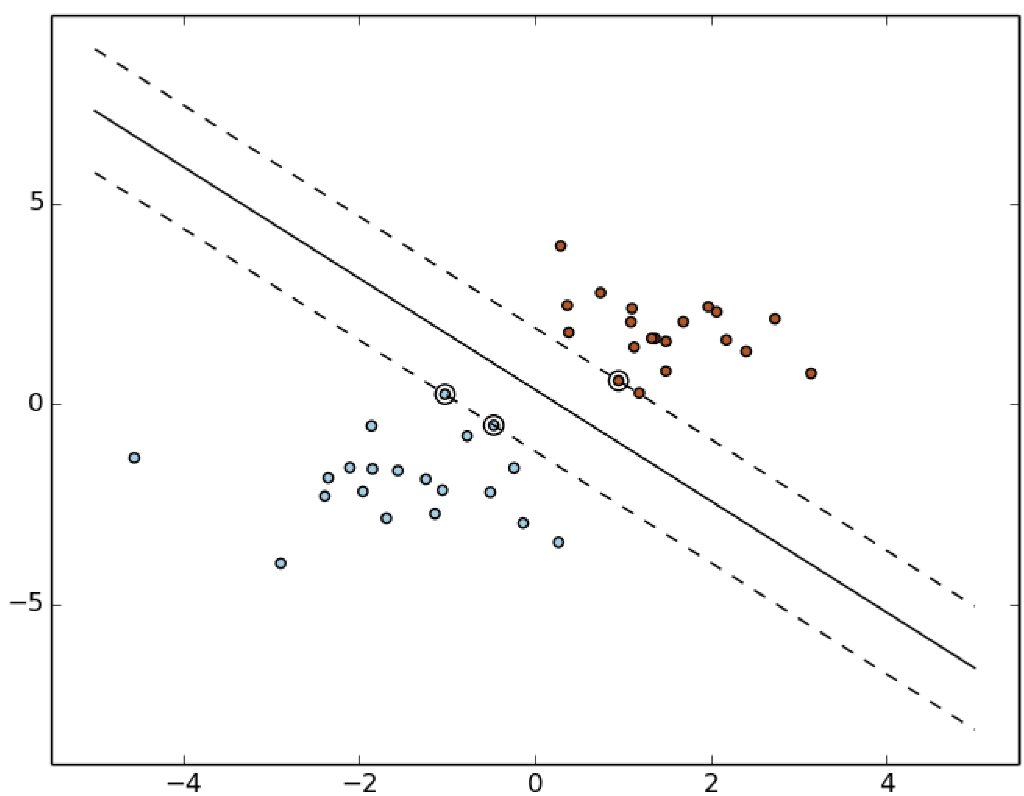


Figure 11.170: *SVM: Maximum margin separating hyperplane*



Python source code: `plot_separating_hyperplane.py`

```
print(__doc__)

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

# fit the model
clf = svm.SVC(kernel='linear')
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
b = clf.support_vectors_[0]
yy_down = a * xx + (b[1] - a * b[0])
b = clf.support_vectors_[-1]
yy_up = a * xx + (b[1] - a * b[0])

# plot the line, the points, and the nearest vectors to the plane
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')
plt.scatter(X[:, 0], X[:, 1], c=Y, cmap=plt.cm.Paired)

plt.axis('tight')
plt.show()
```

Total running time of the example: 0.09 seconds (0 minutes 0.09 seconds)

11.22.4 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`:

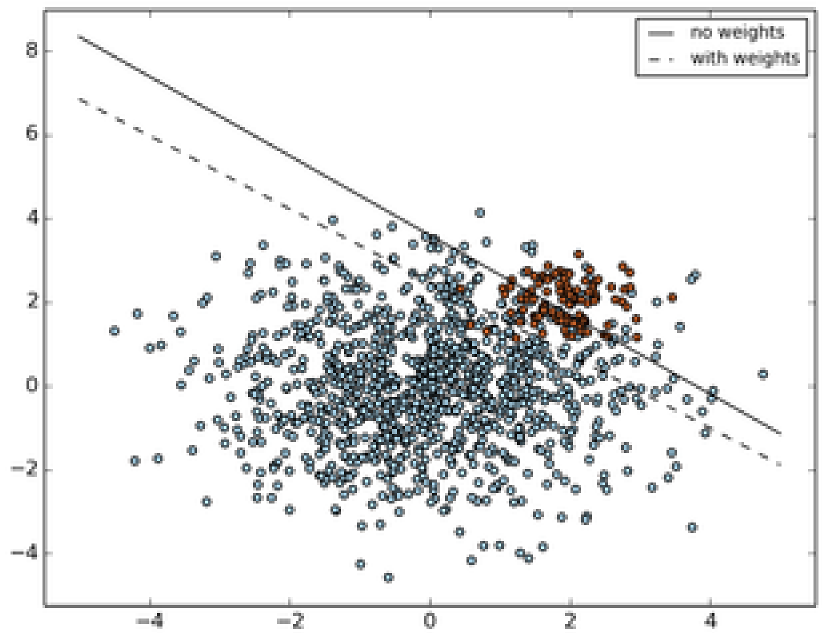
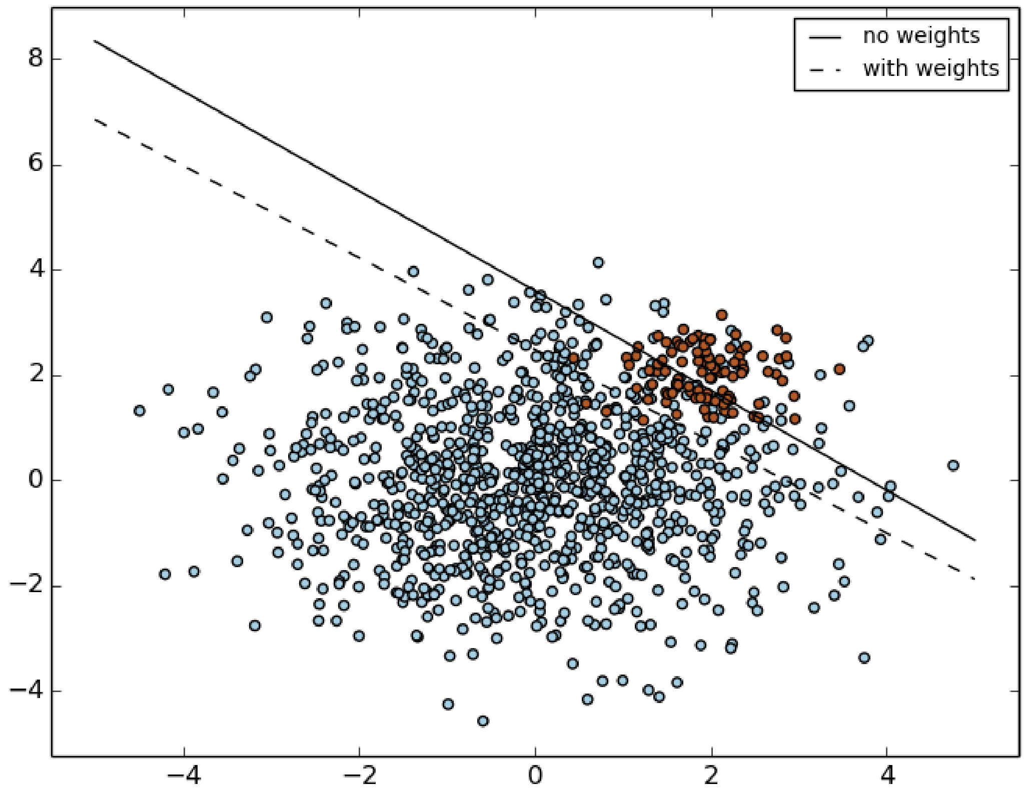


Figure 11.171: *SVM: Separating hyperplane for unbalanced classes*

```
clf = SGDClassifier(n_iter=100, alpha=0.01)
```



Python source code: `plot_separating_hyperplane_unbalanced.py`

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
#from sklearn.linear_model import SGDClassifier

# we create 40 separable points
rng = np.random.RandomState(0)
n_samples_1 = 1000
n_samples_2 = 100
X = np.r_[1.5 * rng.randn(n_samples_1, 2),
          0.5 * rng.randn(n_samples_2, 2) + [2, 2]]
y = [0] * (n_samples_1) + [1] * (n_samples_2)

# fit the model and get the separating hyperplane
clf = svm.SVC(kernel='linear', C=1.0)
clf.fit(X, y)

w = clf.coef_[0]
a = -w[0] / w[1]
xx = np.linspace(-5, 5)
yy = a * xx - clf.intercept_[0] / w[1]
```

```
# get the separating hyperplane using weighted classes
wclf = svm.SVC(kernel='linear', class_weight={1: 10})
wclf.fit(X, y)

ww = wclf.coef_[0]
wa = -ww[0] / ww[1]
wyy = wa * xx - wclf.intercept_[0] / ww[1]

# plot separating hyperplanes and samples
h0 = plt.plot(xx, yy, 'k-', label='no weights')
h1 = plt.plot(xx, wyy, 'k--', label='with weights')
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)
plt.legend()

plt.axis('tight')
plt.show()
```

Total running time of the example: 0.10 seconds (0 minutes 0.10 seconds)

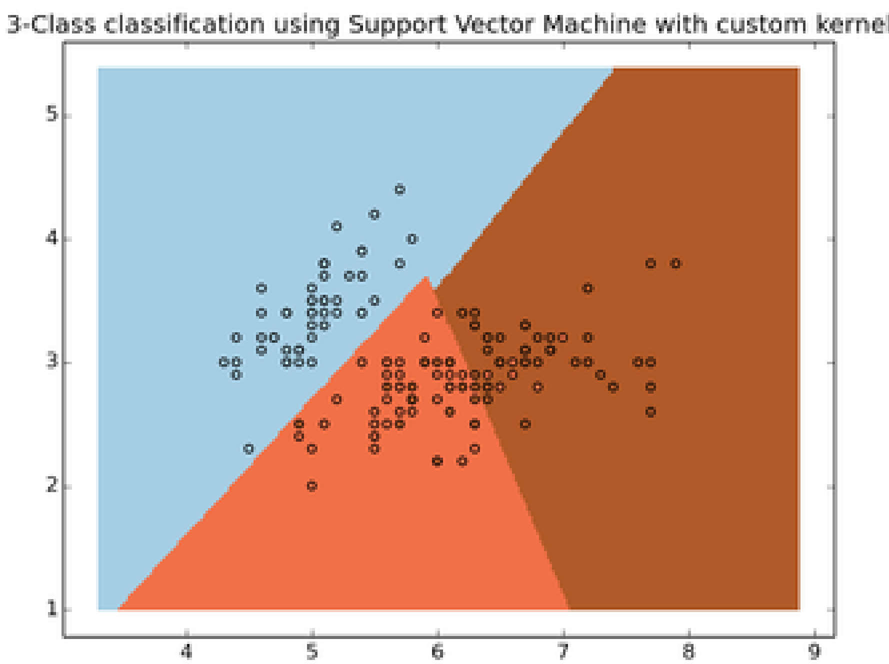
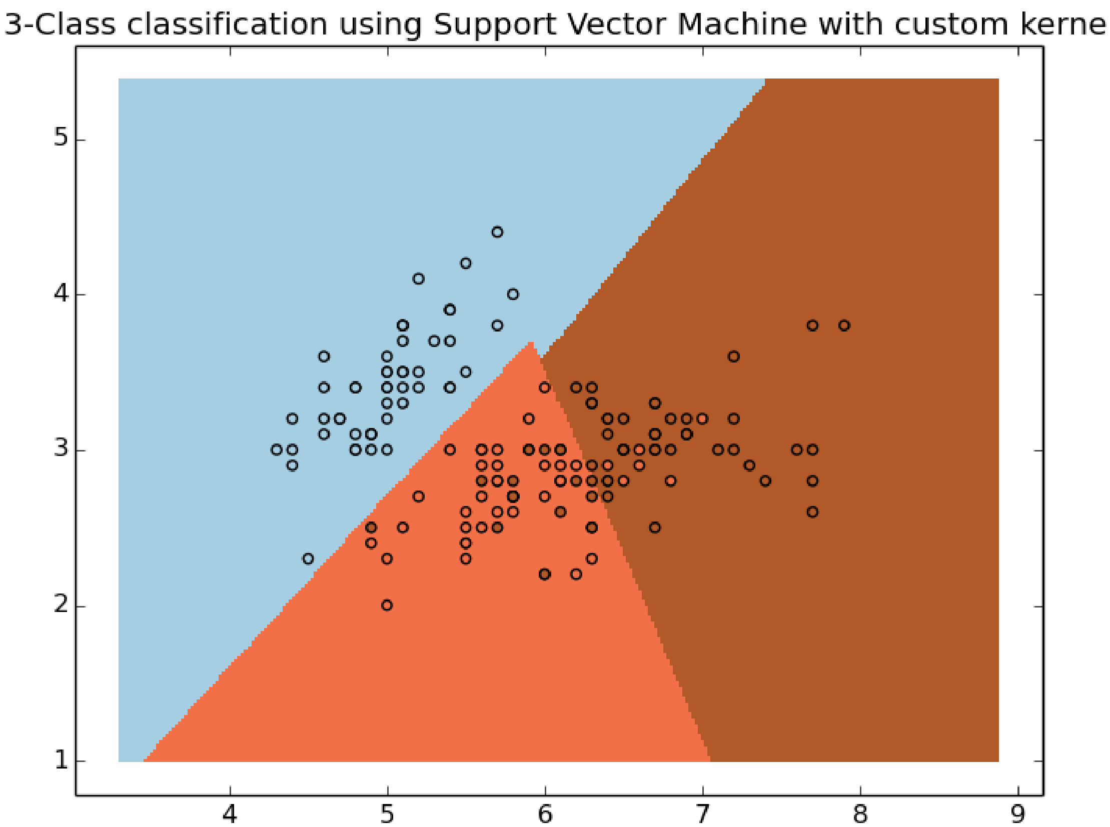


Figure 11.172: *SVM with custom kernel*

11.22.5 SVM with custom kernel

Simple usage of Support Vector Machines to classify a sample. It will plot the decision surface and the support vectors.



Python source code: `plot_custom_kernel.py`

```
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, m_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)
plt.title('3-Class classification using Support Vector Machine with custom'
          ' kernel')
plt.axis('tight')
plt.show()
```

Total running time of the example: 0.19 seconds (0 minutes 0.19 seconds)

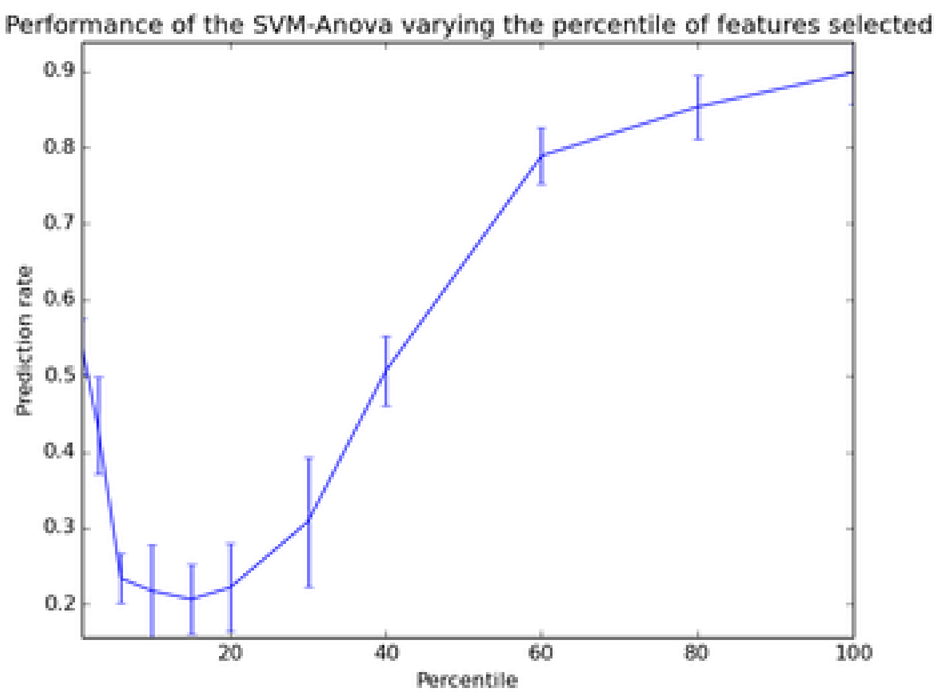
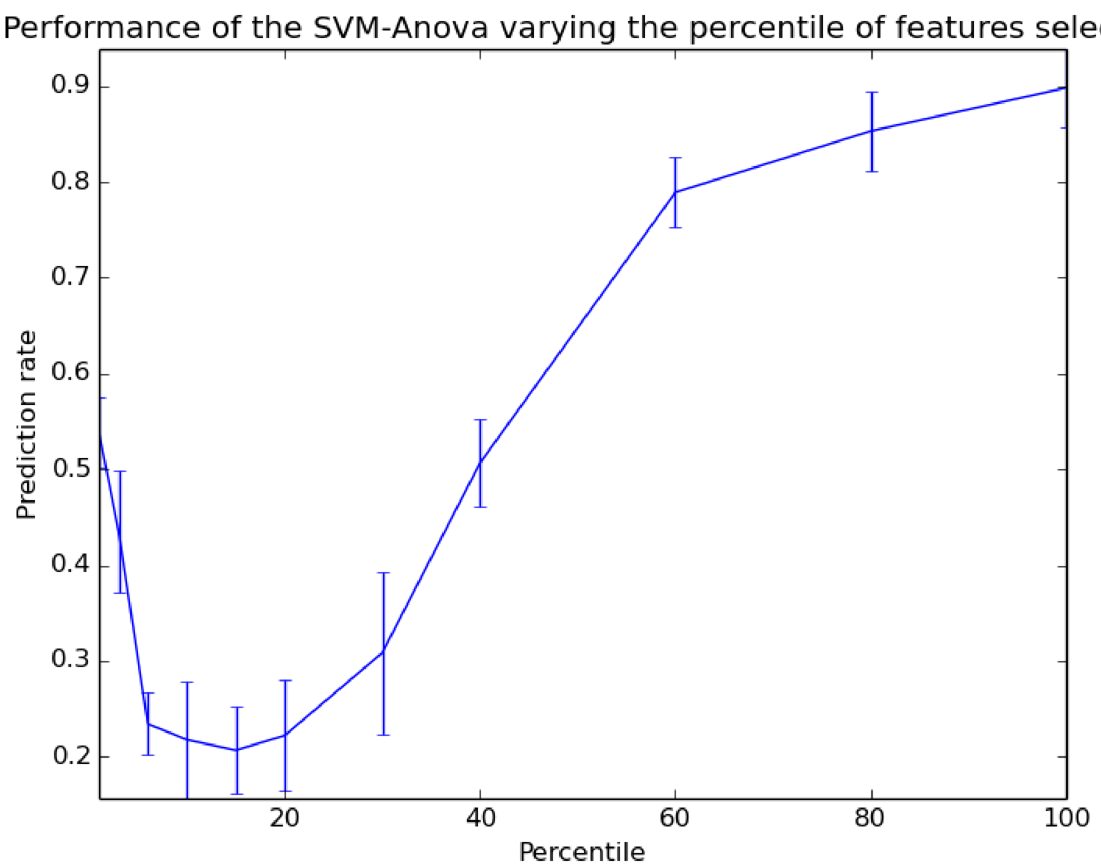


Figure 11.173: *SVM-Anova: SVM with univariate feature selection*

11.22.6 SVM-Anova: SVM with univariate feature selection

This example shows how to perform univariate feature before running a SVC (support vector classifier) to improve the classification scores.



Python source code: plot_svm_anova.py

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets, feature_selection, cross_validation
from sklearn.pipeline import Pipeline

###############################################################################
# Import some data to play with
digits = datasets.load_digits()
y = digits.target
# Throw away data, to be in the curse of dimension settings
y = y[:200]
X = digits.data[:200]
n_samples = len(y)
X = X.reshape((n_samples, -1))
# add 200 non-informative features
X = np.hstack((X, 2 * np.random.random((n_samples, 200))))
```

```
###############################################################################
# Create a feature-selection transform and an instance of SVM that we
# combine together to have an full-blown estimator

transform = feature_selection.SelectPercentile(feature_selection.f_classif)

clf = Pipeline([('anova', transform), ('svc', svm.SVC(C=1.0))])

###############################################################################
# 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)
    # Compute cross-validation score using all CPUs
    this_scores = cross_validation.cross_val_score(clf, X, y, n_jobs=1)
    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.xlabel('Percentile')
plt.ylabel('Prediction rate')

plt.axis('tight')
plt.show()
```

Total running time of the example: 0.42 seconds (0 minutes 0.42 seconds)

11.22.7 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 emphasis the effect here, we particularly weight outliers, making the deformation of the decision boundary very visible.

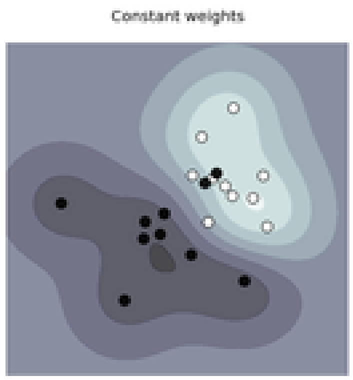
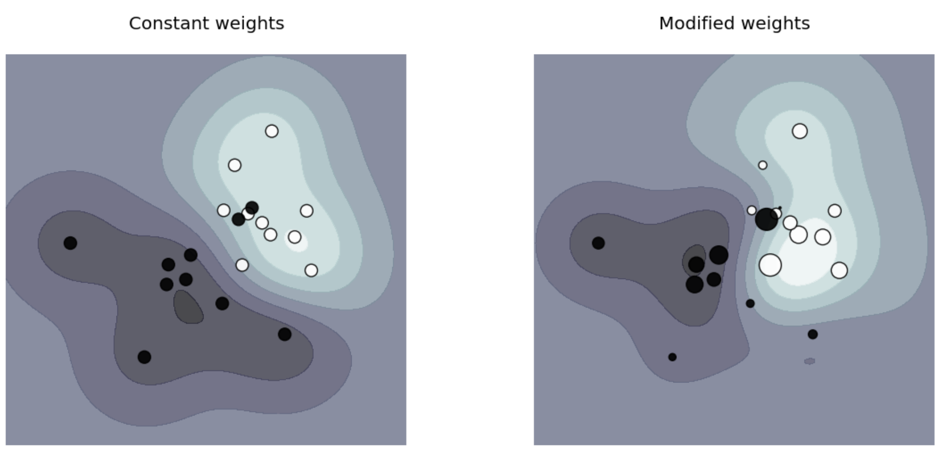


Figure 11.174: *SVM: Weighted samples*



Python source code: `plot_weighted_samples.py`

```
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)

    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
sample_weight_last_ten[9] *= 15

# for reference, first fit without class weights

# fit the model
clf_weights = svm.SVC()
clf_weights.fit(X, Y, sample_weight=sample_weight_last_ten)

clf_no_weights = svm.SVC()
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 example: 0.42 seconds (0 minutes 0.42 seconds)

11.22.8 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.

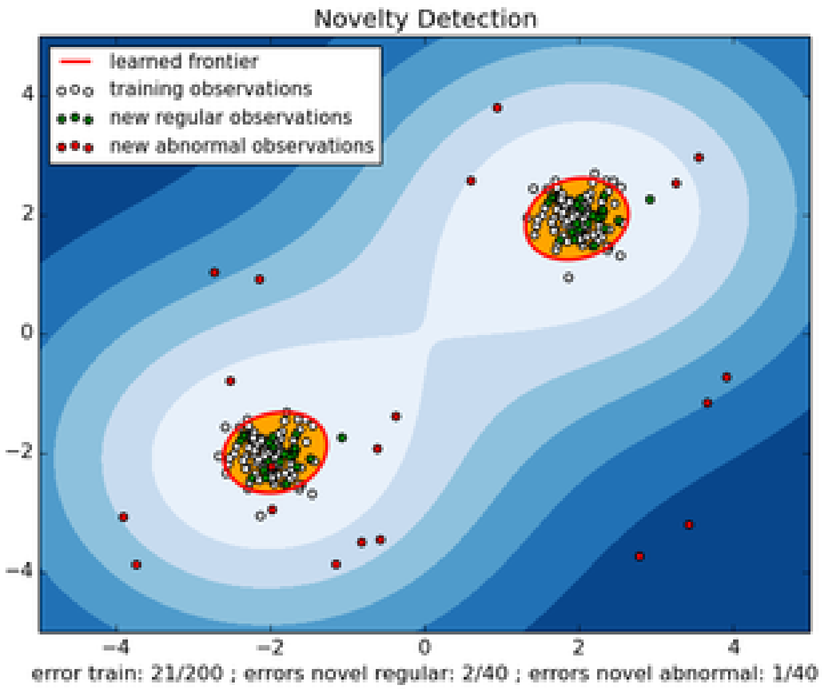
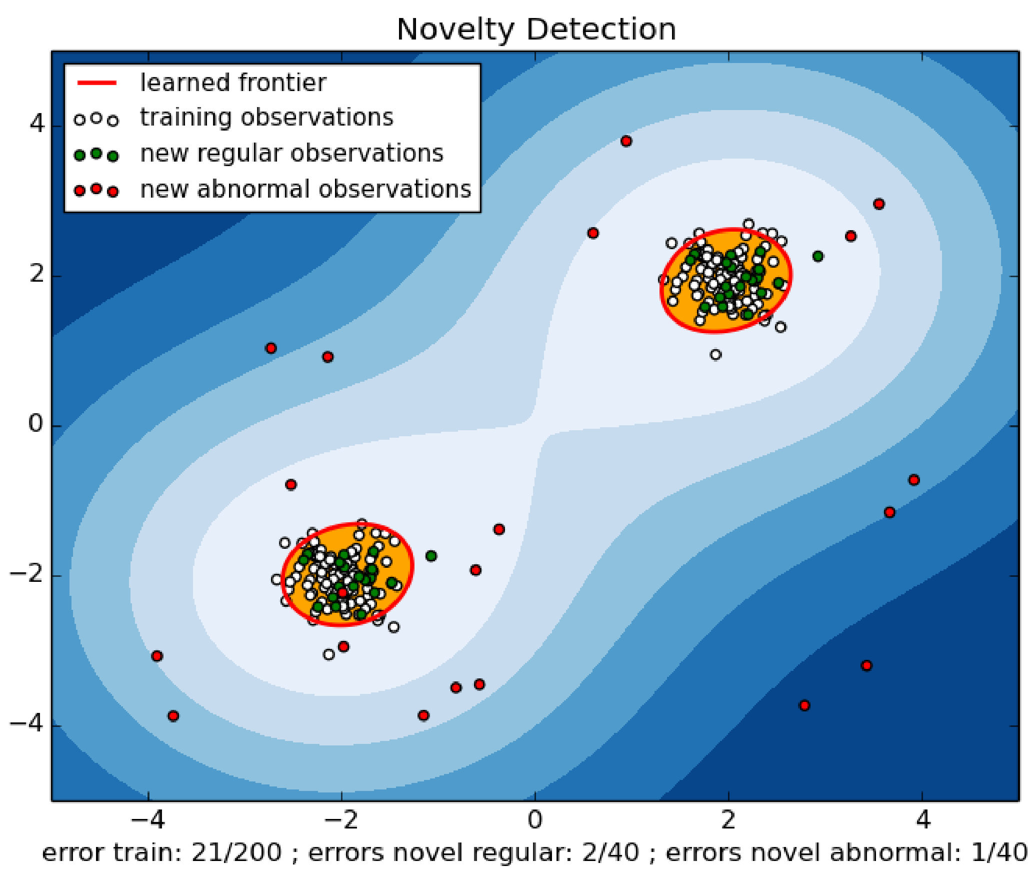


Figure 11.175: *One-class SVM with non-linear kernel (RBF)*



Python source code: `plot_oneclass.py`

```
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)
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.Blues_r)
a = plt.contour(xx, yy, Z, levels=[0], linewidths=2, colors='red')
plt.contourf(xx, yy, Z, levels=[0, Z.max()], colors='orange')

b1 = plt.scatter(X_train[:, 0], X_train[:, 1], c='white')
b2 = plt.scatter(X_test[:, 0], X_test[:, 1], c='green')
c = plt.scatter(X_outliers[:, 0], X_outliers[:, 1], c='red')
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 example: 0.26 seconds (0 minutes 0.26 seconds)

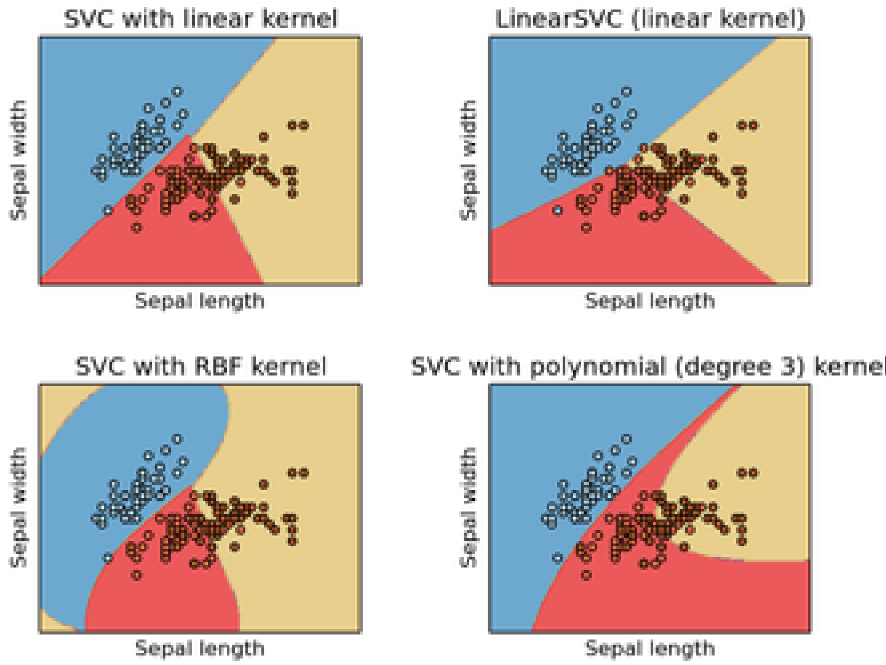


Figure 11.176: *Plot different SVM classifiers in the iris dataset*

11.22.9 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

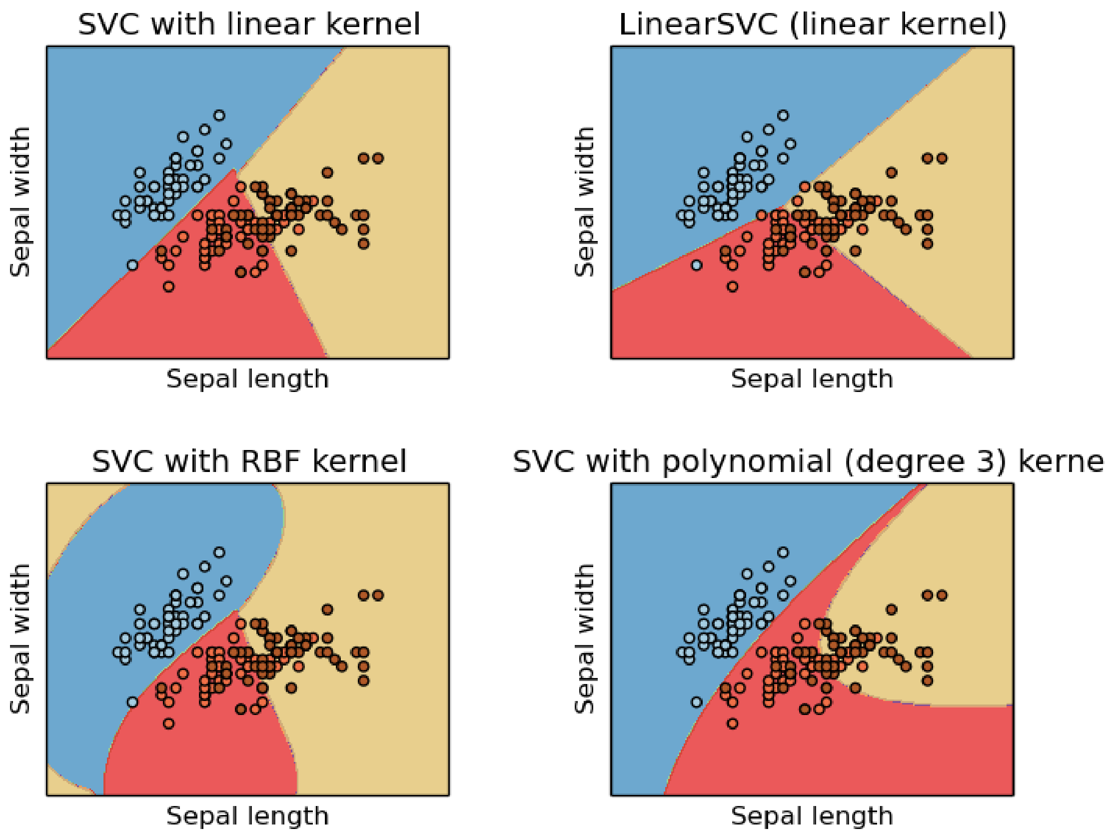
This example shows how to plot the decision surface for four SVM classifiers with different kernels.

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 problem.



Python source code: `plot_iris.py`

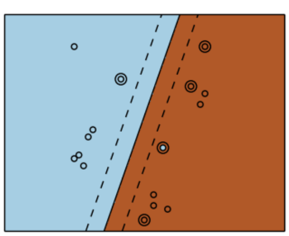
```
print(__doc__)

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

h = .02  # step size in the mesh

# 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
svc = svm.SVC(kernel='linear', C=C).fit(X, y)
rbf_svc = svm.SVC(kernel='rbf', gamma=0.7, C=C).fit(X, y)
poly_svc = svm.SVC(kernel='poly', degree=3, C=C).fit(X, y)
lin_svc = svm.LinearSVC(C=C).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))

# title for the plots
titles = ['SVC with linear kernel',
          'LinearSVC (linear kernel)',
          'SVC with RBF kernel',
          'SVC with polynomial (degree 3) kernel']


for i, clf in enumerate((svc, lin_svc, rbf_svc, poly_svc)):
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, m_max]x[y_min, y_max].
    plt.subplot(2, 2, i + 1)
    plt.subplots_adjust(wspace=0.4, hspace=0.4)

    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, alpha=0.8)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)
    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(titles[i])

plt.show()
```

Total running time of the example: 0.48 seconds (0 minutes 0.48 seconds)

11.22.10 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.

- 

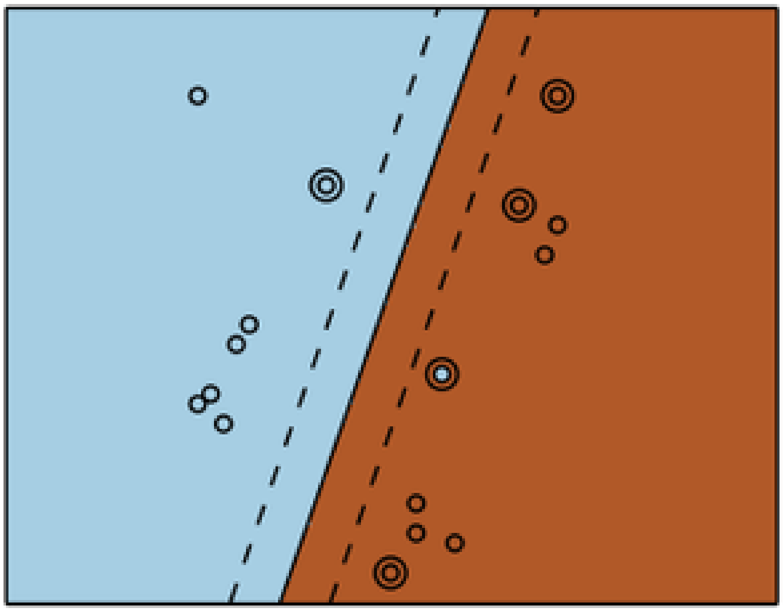
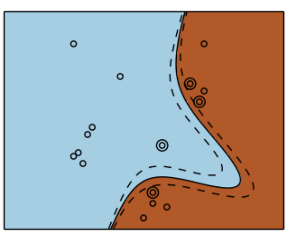
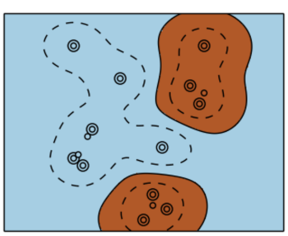


Figure 11.177: *SVM-Kernels*

- 
- 

Python source code: `plot_svm_kernels.py`

```
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)
    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)
    plt.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=plt.cm.Paired)

    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 example: 0.19 seconds (0 minutes 0.19 seconds)

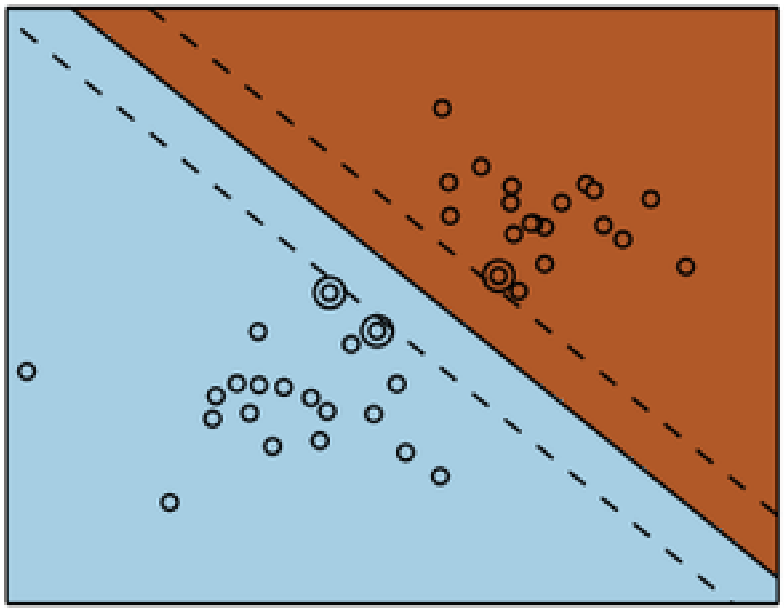
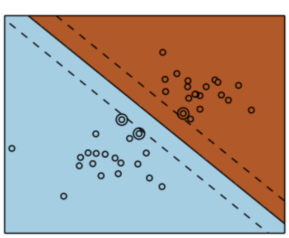


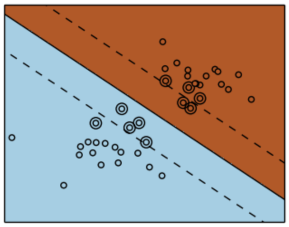
Figure 11.178: *SVM Margins Example*

11.22.11 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.

- 

- 

Python source code: `plot_svm_margin.py`

```
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
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 = 1 / np.sqrt(np.sum(clf.coef_ ** 2))
    yy_down = yy + a * margin
    yy_up = yy - a * 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)
    plt.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=plt.cm.Paired)
```

```
    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 example: 0.15 seconds (0 minutes 0.15 seconds)

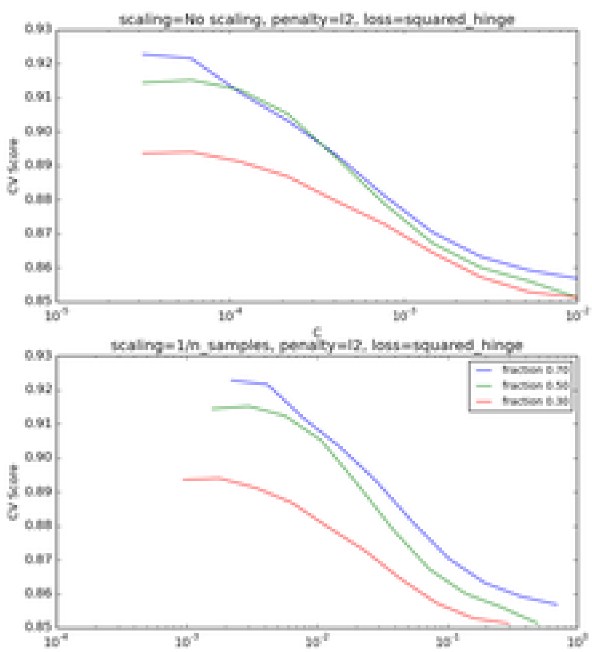


Figure 11.179: *Scaling the regularization parameter for SVCs*

11.22.12 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

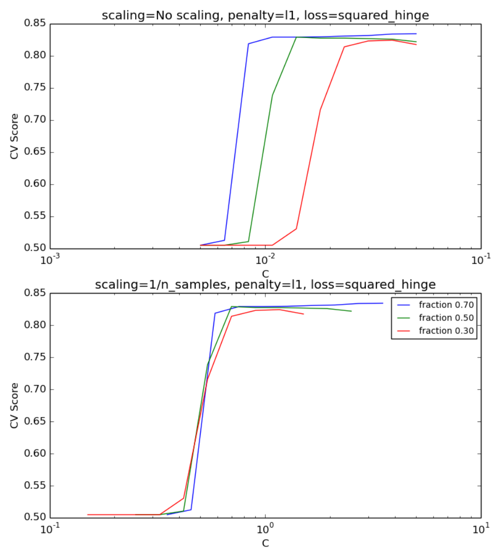
The two figures below plot the values of *C* on the *x-axis* and the corresponding cross-validation scores on the *y-axis*, for several different fractions of a generated data-set.

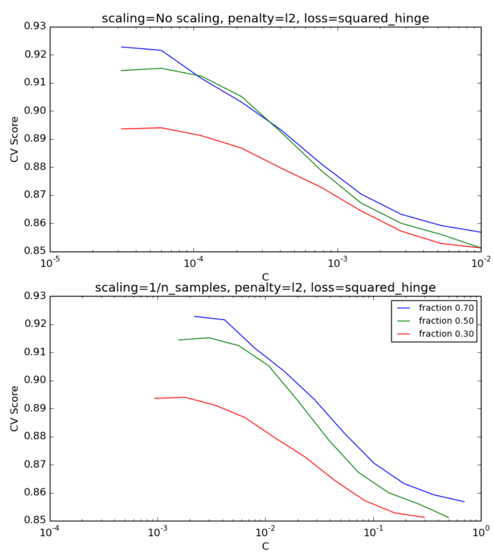
In the *l1* penalty case, the cross-validation-error correlates best with the test-error, when scaling our *C* with the number of samples, *n*, which can be seen in the first figure.

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.

- 

- 

Python source code: `plot_svm_scale_c.py`

```
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
from sklearn.cross_validation import ShuffleSplit
```

```
from sklearn.grid_search 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,
                       tol=1e-4),
             np.logspace(-4.5, -2, 10), X_2, y_2)]

colors = ['b', 'g', 'r', 'c']

for fignum, (clf, cs, X, y) in enumerate(clf_sets):
    # set up the plot for each regressor
    plt.figure(fignum, 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(n=n_samples, train_size=train_size,
                                            n_iter=250, random_state=1))
        grid.fit(X, y)
        scores = [x[1] for x in grid.grid_scores_]

        scales = [(1, 'No scaling'),
                  ((n_samples * train_size), '1/n_samples'),
                  ]

        for subplotnum, (scaler, name) in enumerate(scales):
            plt.subplot(2, 1, subplotnum + 1)
            plt.xlabel('C')
            plt.ylabel('CV Score')
            grid_cs = cs * float(scaler)  # scale the C's
            plt.semilogx(grid_cs, scores, label="fraction %.2f" %
                         train_size)
            plt.title('scaling=%s, penalty=%s, loss=%s' %
                      (name, clf.penalty, clf.loss))
```

```
    plt.legend(loc="best")
plt.show()
```

Total running time of the example: 12.17 seconds (0 minutes 12.17 seconds)

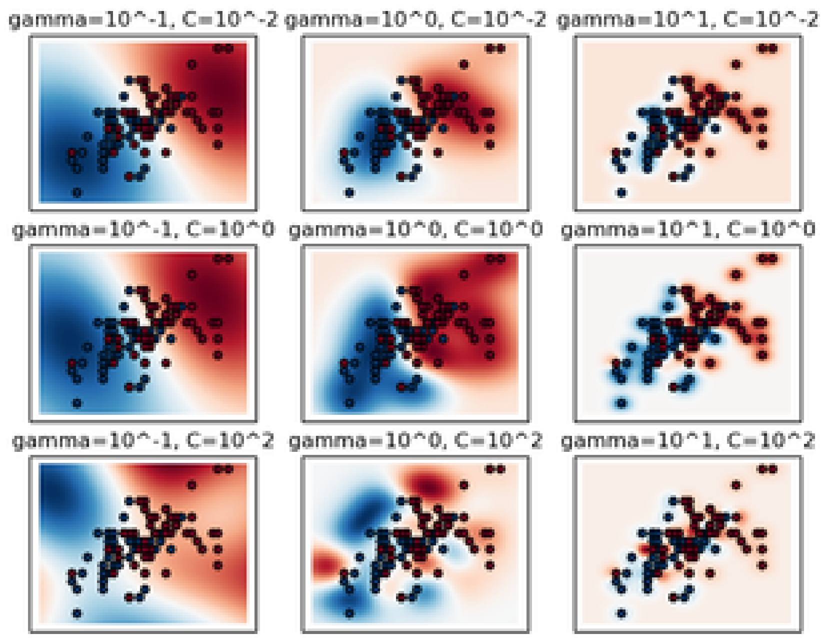


Figure 11.180: *RBF SVM parameters*

11.22.13 RBF SVM parameters

This example illustrates the effect of the parameters `gamma` and `C` of the Radius 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 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 by give the model freedom to select more samples as support vectors.

The first plot is a visualization of the decision function for a variety of parameter values on 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.

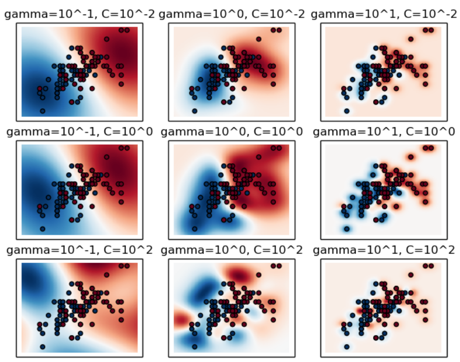
The behavior of the model is very sensitive to the `gamma` parameter. If `gamma` is too large, the radius of the area of influence of the support vectors only includes the support vector it-self and no amount of regularization with `C` will be able to prevent of overfitting.

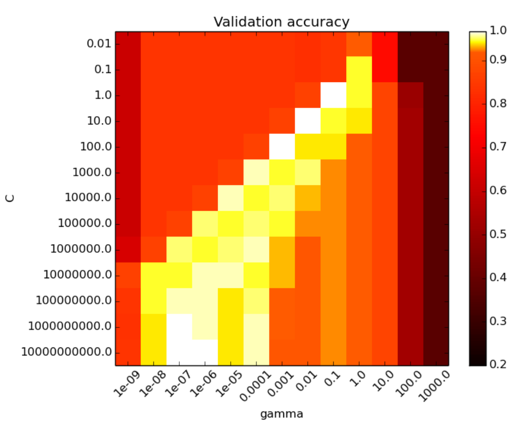
When `gamma` is very small, the model is too constrained and cannot capture the complexity or "shape" of the data. The region of influence of any selected support vector would include the whole training set. The resulting model will behave similarly to a linear model with a set of hyperplanes that separate the centers of high density of any pair of two classes.

For intermediate values, we can see on a 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 selecting a larger number of support vectors (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 limiting the number support vectors. The radius of the RBF kernel alone acts as a good structural regularizer. In practice though it might still be interesting to limit the number of support vectors 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 smoothed out by increasing the number of CV iterations `n_iter` 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.

- 

- 

Script output:

```
The best parameters are {'C': 1.0, 'gamma': 0.10000000000000001} with a score of 0.97
```

Python source code: `plot_rbf_parameters.py`

```
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.cross_validation import StratifiedShuffleSplit
from sklearn.grid_search import GridSearchCV


# Utility function to move the midpoint of a colormap to be around
# the values of interest.

class MidpointNormalize(Normalize):

    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 class to has
# to 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
```

```python
# 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(y, n_iter=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]
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)
    plt.xticks(())
    plt.yticks(())
    plt.axis('tight')

# plot the scores of the grid
# grid_scores_ contains parameter settings and scores
# We extract just the scores
scores = [x[1] for x in grid.grid_scores_]
scores = np.array(scores).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 example: 4.62 seconds (0 minutes 4.62 seconds)

11.23 Working with text documents

Examples concerning the `sklearn.feature_extraction.text` module.

Figure 11.181: *FeatureHasher and DictVectorizer Comparison*

11.23.1 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.

Python source code: `hashing_vs_dict_vectorizer.py`

```
# Author: Lars Buitinck <L.J.Buitinck@uva.nl>
# License: BSD 3 clause

from __future__ import print_function
```

```python
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:
    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).data
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))
```


Figure 11.182: *Classification of text documents: using a MLComp dataset*

11.23.2 Classification of text documents: using a MLComp dataset

This is an example showing how the 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 instead of standard numpy arrays.

The dataset used in this example is the 20 newsgroups dataset and should be downloaded from the http://mlcomp.org (free registration required):

> http://mlcomp.org/datasets/379

Once downloaded unzip the archive somewhere on your filesystem. For instance in:

```
% mkdir -p ~/data/mlcomp
% cd  ~/data/mlcomp
% unzip /path/to/dataset-379-20news-18828_XXXXX.zip
```

You should get a folder `~/data/mlcomp/379` with a file named `metadata` and subfolders `raw`, `train` and `test` holding the text documents organized by newsgroups.

Then set the `MLCOMP_DATASETS_HOME` environment variable pointing to the root folder holding the uncompressed archive:

```
% export MLCOMP_DATASETS_HOME="~/data/mlcomp"
```

Then you are ready to run this example using your favorite python shell:

```
% ipython examples/mlcomp_sparse_document_classification.py
```

Python source code: `mlcomp_sparse_document_classification.py`

```python
# Author: Olivier Grisel <olivier.grisel@ensta.org>
# License: BSD 3 clause

from __future__ import print_function

from time import time
import sys
import os
import numpy as np
import scipy.sparse as sp
import pylab as pl

from sklearn.datasets import load_mlcomp
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.naive_bayes import MultinomialNB


print(__doc__)

if 'MLCOMP_DATASETS_HOME' not in os.environ:
    print("MLCOMP_DATASETS_HOME not set; please follow the above instructions")
    sys.exit(0)

# Load the training set
print("Loading 20 newsgroups training set... ")
news_train = load_mlcomp('20news-18828', 'train')
print(news_train.DESCR)
print("%d documents" % len(news_train.filenames))
print("%d categories" % len(news_train.target_names))

print("Extracting features from the dataset using a sparse vectorizer")
t0 = time()
vectorizer = TfidfVectorizer(encoding='latin1')
X_train = vectorizer.fit_transform((open(f).read()
                                    for f in news_train.filenames))
print("done in %fs" % (time() - t0))
print("n_samples: %d, n_features: %d" % X_train.shape)
```

```
assert sp.issparse(X_train)
y_train = news_train.target

print("Loading 20 newsgroups test set... ")
news_test = load_mlcomp('20news-18828', 'test')
t0 = time()
print("done in %fs" % (time() - t0))

print("Predicting the labels of the test set...")
print("%d documents" % len(news_test.filenames))
print("%d categories" % len(news_test.target_names))

print("Extracting features from the dataset using the same vectorizer")
t0 = time()
X_test = vectorizer.transform((open(f).read() for f in news_test.filenames))
y_test = news_test.target
print("done in %fs" % (time() - t0))
print("n_samples: %d, n_features: %d" % X_test.shape)


###############################################################################
# Benchmark classifiers
def benchmark(clf_class, params, name):
    print("parameters:", params)
    t0 = time()
    clf = clf_class(**params).fit(X_train, y_train)
    print("done in %fs" % (time() - t0))

    if hasattr(clf, 'coef_'):
        print("Percentage of non zeros coef: %f"
              % (np.mean(clf.coef_ != 0) * 100))
    print("Predicting the outcomes of the testing set")
    t0 = time()
    pred = clf.predict(X_test)
    print("done in %fs" % (time() - t0))

    print("Classification report on test set for classifier:")
    print(clf)
    print()
    print(classification_report(y_test, pred,
                                target_names=news_test.target_names))

    cm = confusion_matrix(y_test, pred)
    print("Confusion matrix:")
    print(cm)

    # Show confusion matrix
    pl.matshow(cm)
    pl.title('Confusion matrix of the %s classifier' % name)
    pl.colorbar()


print("Testbenching a linear classifier...")
parameters = {
    'loss': 'hinge',
    'penalty': 'l2',
    'n_iter': 50,
    'alpha': 0.00001,
```

```
    'fit_intercept': True,
}

benchmark(SGDClassifier, parameters, 'SGD')

print("Testbenching a MultinomialNB classifier...")
parameters = {'alpha': 0.01}

benchmark(MultinomialNB, parameters, 'MultinomialNB')

pl.show()
```


Figure 11.183: *Clustering text documents using k-means*

11.23.3 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.

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.

Python source code: document_clustering.py

```
# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#         Lars Buitinck <L.J.Buitinck@uva.nl>
# License: BSD 3 clause

from __future__ import print_function

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.")

print(__doc__)
op.print_help()

(opts, args) = op.parse_args()
```

```
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', non_negative=True,
                                   norm=None, binary=False)
        vectorizer = make_pipeline(hasher, TfidfTransformer())
    else:
        vectorizer = HashingVectorizer(n_features=opts.n_features,
                                       stop_words='english',
                                       non_negative=False, norm='l2',
                                       binary=False)
else:
    vectorizer = TfidfVectorizer(max_df=0.5, max_features=opts.n_features,
                                 min_df=2, stop_words='english',
                                 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)
    lsa = make_pipeline(svd, Normalizer(copy=False))

    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.n_components or opts.use_hashing):
    print("Top terms per cluster:")
    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()
```

11.23.4 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.

Figure 11.184: *Classification of text documents using sparse features*

The dataset used in this example is the 20 newsgroups dataset. It will be automatically downloaded, then cached.

The bar plot indicates the accuracy, training time (normalized) and test time (normalized) of each classifier.

Python source code: `document_classification_20newsgroups.py`

```
# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#         Olivier Grisel <olivier.grisel@ensta.org>
#         Mathieu Blondel <mathieu@mblondel.org>
#         Lars Buitinck <L.J.Buitinck@uva.nl>
# License: BSD 3 clause

from __future__ import print_function

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 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, MultinomialNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neighbors import NearestCentroid
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')
```

```
# parse commandline arguments
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.")

(opts, args) = op.parse_args()
if len(args) > 0:
    op.error("this script takes no arguments.")
    sys.exit(1)

print(__doc__)
op.print_help()
print()


###############################################################################
# Load some categories from the training set
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')

categories = data_train.target_names    # for case categories == None


def size_mb(docs):
    return sum(len(s.encode('utf-8')) for s in docs) / 1e6

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(categories))
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', non_negative=True,
                                   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()

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] + "..."


###############################################################################
# Benchmark classifiers
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, category in enumerate(categories):
                top10 = np.argsort(clf.coef_[i])[-10:]
                print(trim("%s: %s"
                      % (category, " ".join(feature_names[top10]))))
        print()

    if opts.print_report:
        print("classification report:")
        print(metrics.classification_report(y_test, pred,
                                            target_names=categories))
```

```
    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="lsqr"), "Ridge Classifier"),
        (Perceptron(n_iter=50), "Perceptron"),
        (PassiveAggressiveClassifier(n_iter=50), "Passive-Aggressive"),
        (KNeighborsClassifier(n_neighbors=10), "kNN"),
        (RandomForestClassifier(n_estimators=100), "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(loss='l2', penalty=penalty,
                                            dual=False, tol=1e-3)))

    # Train SGD model
    results.append(benchmark(SGDClassifier(alpha=.0001, n_iter=50,
                                           penalty=penalty)))

# Train SGD with Elastic Net penalty
print('=' * 80)
print("Elastic-Net penalty")
results.append(benchmark(SGDClassifier(alpha=.0001, n_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)))

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', LinearSVC(penalty="l1", dual=False, tol=1e-3)),
  ('classification', LinearSVC())
])))

# make some plots
```

```
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='r')
plt.barh(indices + .3, training_time, .2, label="training time", color='g')
plt.barh(indices + .6, test_time, .2, label="test time", color='b')
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()
```

11.24 Decision Trees

Examples concerning the `sklearn.tree` module.

11.24.1 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.

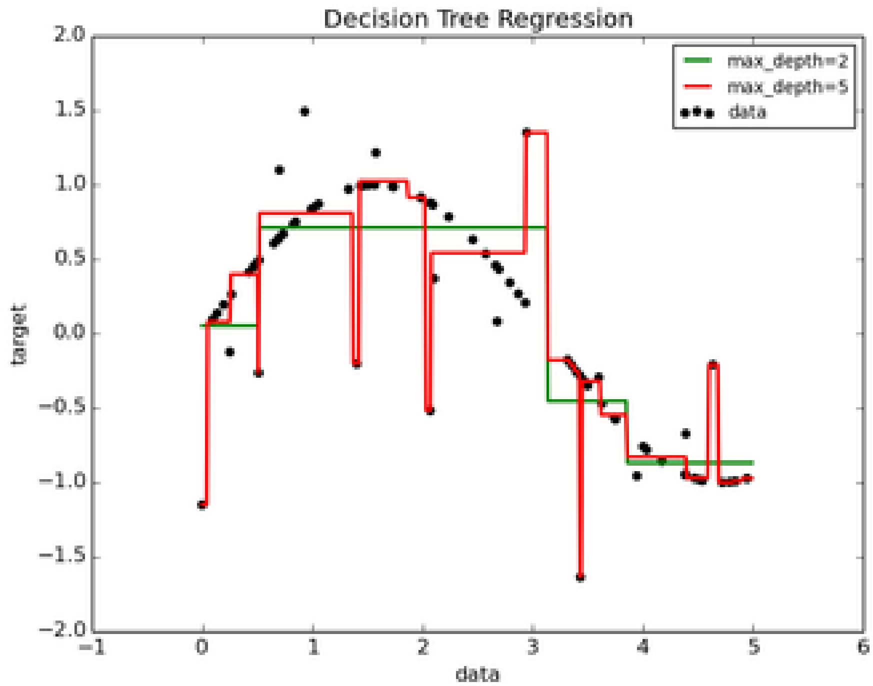
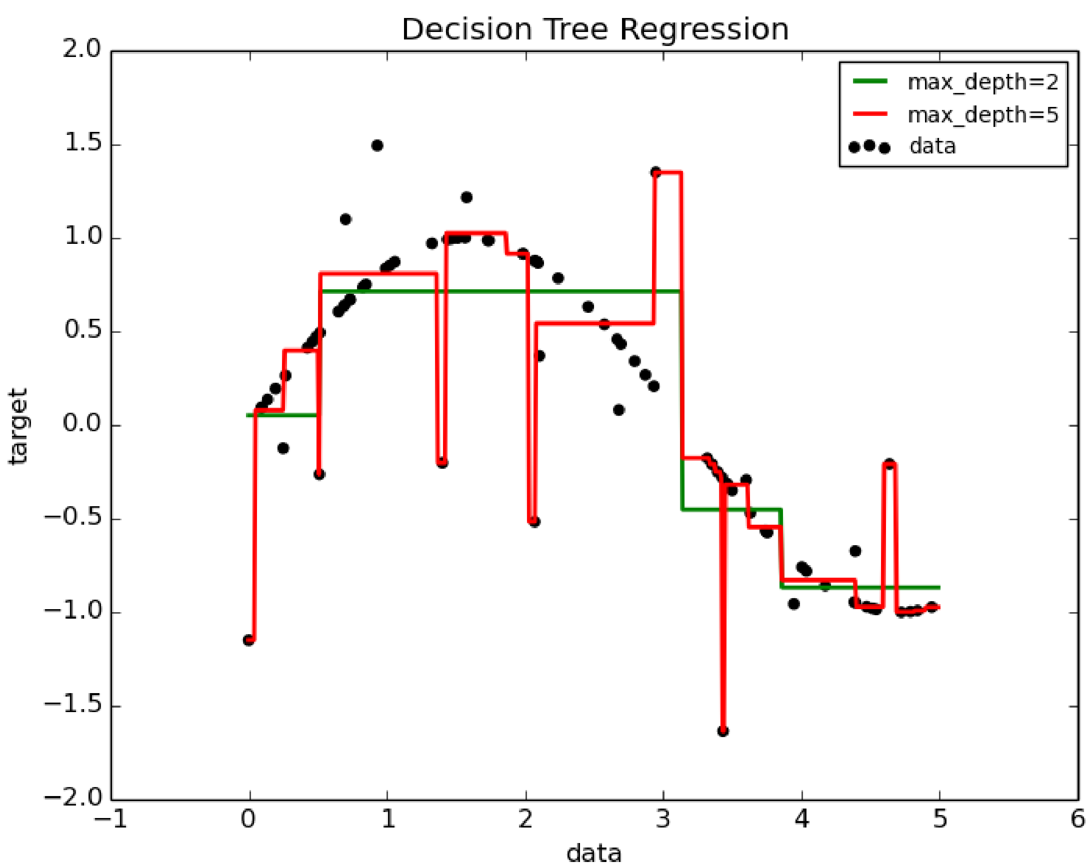


Figure 11.185: *Decision Tree Regression*



Python source code: plot_tree_regression.py

```
print(__doc__)

# Import the necessary modules and libraries
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
clf_1 = DecisionTreeRegressor(max_depth=2)
clf_2 = DecisionTreeRegressor(max_depth=5)
clf_1.fit(X, y)
clf_2.fit(X, y)

# Predict
X_test = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]
y_1 = clf_1.predict(X_test)
y_2 = clf_2.predict(X_test)

# Plot the results
plt.figure()
plt.scatter(X, y, c="k", label="data")
plt.plot(X_test, y_1, c="g", label="max_depth=2", linewidth=2)
plt.plot(X_test, y_2, c="r", 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 example: 0.11 seconds (0 minutes 0.11 seconds)

11.24.2 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.

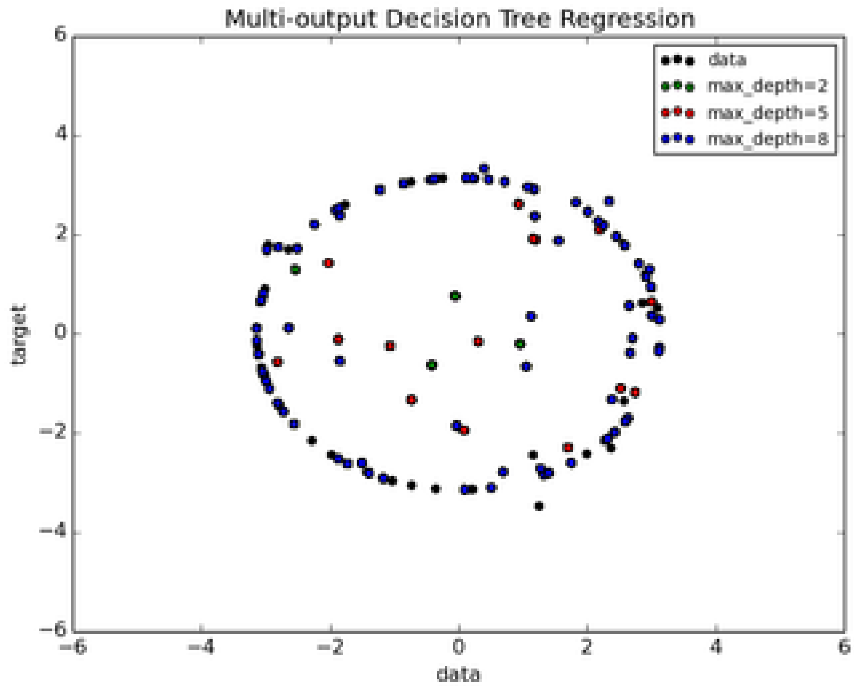
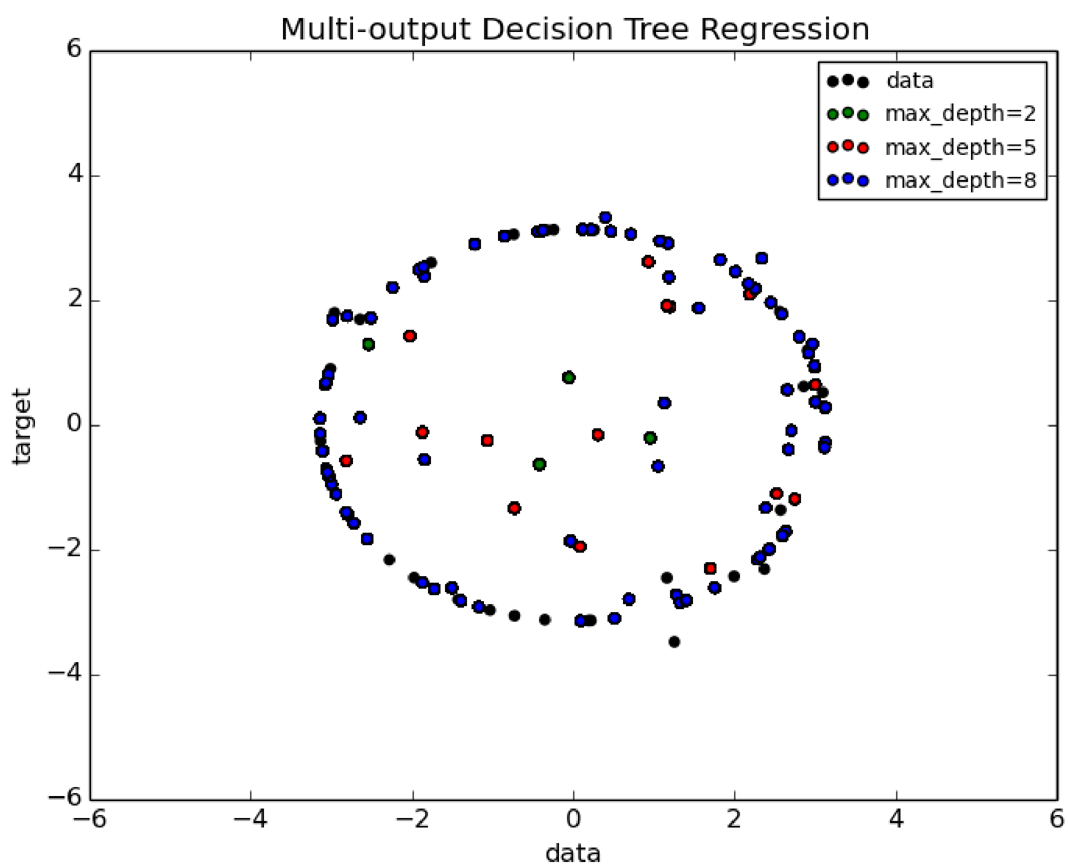


Figure 11.186: *Multi-output Decision Tree Regression*



Python source code: plot_tree_regression_multioutput.py

```
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
clf_1 = DecisionTreeRegressor(max_depth=2)
clf_2 = DecisionTreeRegressor(max_depth=5)
clf_3 = DecisionTreeRegressor(max_depth=8)
clf_1.fit(X, y)
clf_2.fit(X, y)
clf_3.fit(X, y)

# Predict
X_test = np.arange(-100.0, 100.0, 0.01)[:, np.newaxis]
y_1 = clf_1.predict(X_test)
y_2 = clf_2.predict(X_test)
y_3 = clf_3.predict(X_test)

# Plot the results
plt.figure()
plt.scatter(y[:, 0], y[:, 1], c="k", label="data")
plt.scatter(y_1[:, 0], y_1[:, 1], c="g", label="max_depth=2")
plt.scatter(y_2[:, 0], y_2[:, 1], c="r", label="max_depth=5")
plt.scatter(y_3[:, 0], y_3[:, 1], c="b", label="max_depth=8")
plt.xlim([-6, 6])
plt.ylim([-6, 6])
plt.xlabel("data")
plt.ylabel("target")
plt.title("Multi-output Decision Tree Regression")
plt.legend()
plt.show()
```

Total running time of the example: 0.28 seconds (0 minutes 0.28 seconds)

11.24.3 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.

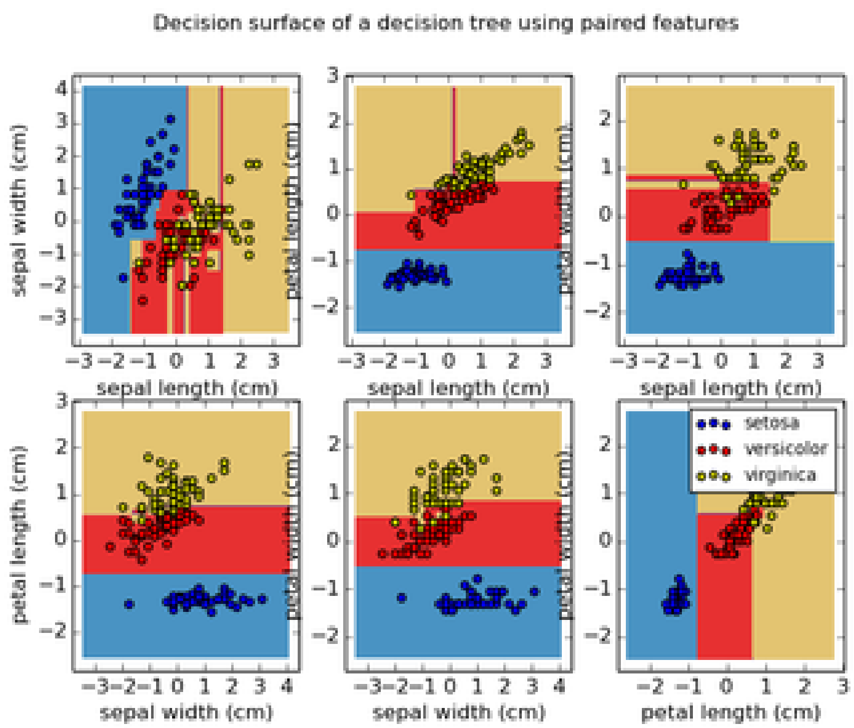
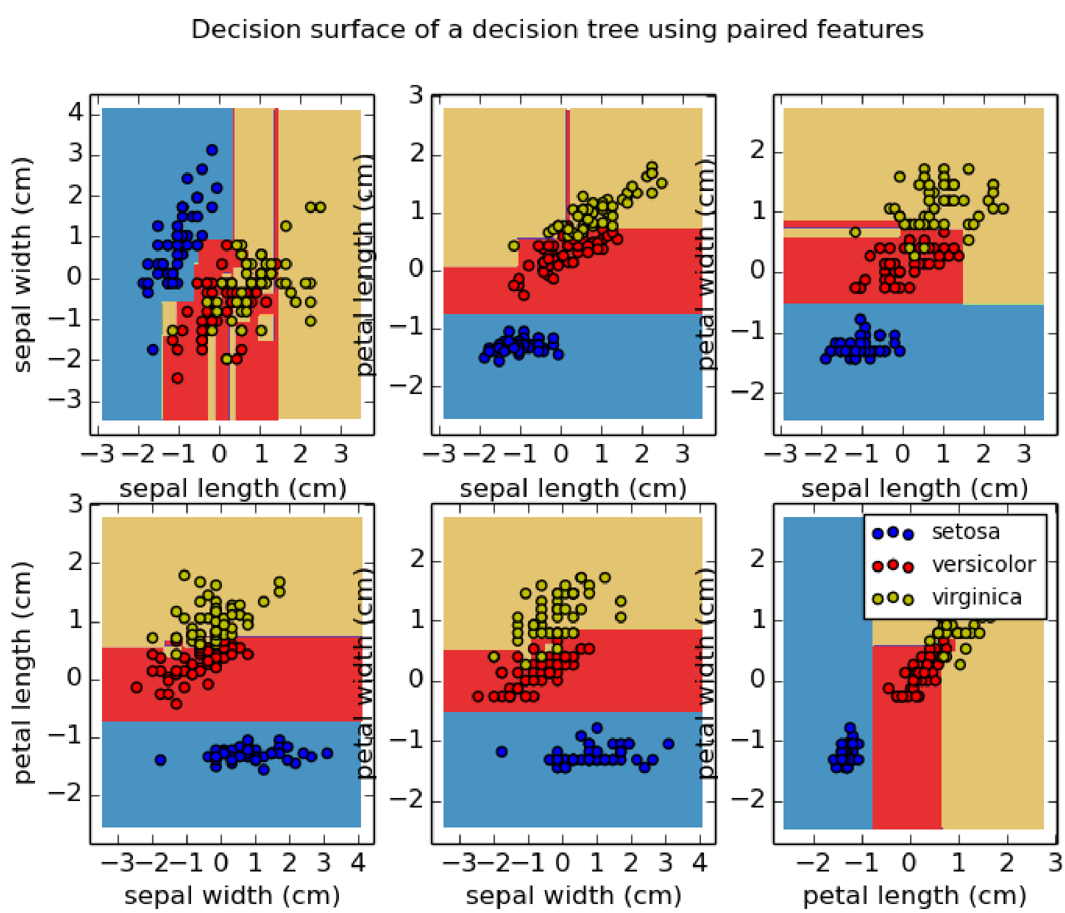


Figure 11.187: *Plot the decision surface of a decision tree on the iris dataset*



Python source code: plot_iris.py

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

# Parameters
n_classes = 3
plot_colors = "bry"
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

    # 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)
    std = X.std(axis=0)
    X = (X - mean) / std

    # Train
    clf = DecisionTreeClassifier().fit(X, y)

    # Plot the decision boundary
    plt.subplot(2, 3, pairidx + 1)

    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 = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    cs = plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)

    plt.xlabel(iris.feature_names[pair[0]])
    plt.ylabel(iris.feature_names[pair[1]])
    plt.axis("tight")

    # 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.Paired)

    plt.axis("tight")

plt.suptitle("Decision surface of a decision tree using paired features")
plt.legend()
plt.show()
```

Total running time of the example: 0.32 seconds (0 minutes 0.32 seconds)

CHAPTER

TWELVE

FREQUENTLY ASKED QUESTIONS

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

12.1 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 where previously used.

12.2 How do you pronounce the project name?

sy-kit learn. sci stands for science!

12.3 Why scikit?

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

12.4 How can I contribute to scikit-learn?

See *Contributing*.

12.5 Can I add this new algorithm that I (or someone else) just published?

No. As a rule we only add well-established algorithms. A rule of thumb is at least 3 years since publications, 200+ citations and wide use and usefullness. A technique that provides a clear-cut improvement (e.g. an enhanced data structure or efficient approximation) on a widely-used method will also be considered for inclusion. Your implementation doesn't need to be in scikit-learn to be used together with scikit-learn tools, though. Implement your favorite algorithm in a scikit-learn compatible way, upload it to github and we will list it under *related_projects*. Also see selectiveness.

12.6 Can I add this classical algorithm from the 80s?

Depends. If there is a common usecase within the scope of scikit-learn, such as classification, regression or clustering, where it outperforms methods that are already implemented in scikit-learn, we will consider it.

12.7 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. Also see this thread on the mailing list.

12.8 Why did you remove HMMs from scikit-learn?

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

12.9 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)

12.10 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.

12.11 Do you support PyPy?

In case you didn't know, PyPy is the new, fast, just-in-time compiling Python implementation. We don't support it. When the NumPy support in PyPy is complete or near-complete, and SciPy is ported over as well, we can start thinking of a port. We use too much of NumPy to work with a partial implementation.

12.12 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]])
>>> dbscan(X, metric=lev_metric, eps=5, min_samples=2)
([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.

CHAPTER

THIRTEEN

SUPPORT

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

13.1 Mailing List

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

13.2 User questions

- Some scikit-learn developers support users on StackOverflow using the [scikit-learn] tag.
- For general theoretical or methodological Machine Learning questions metaoptimize.com/qa 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.

13.3 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.

13.4 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: http://webchat.freenode.net

13.5 Documentation resources

This documentation is relative to 0.16.1. Documentation for other versions can be found here:

- 0.15
- 0.14
- 0.13
- 0.12
- 0.11
- 0.10
- 0.9
- 0.8
- 0.7
- 0.6
- 0.5

Printable pdf documentation for all versions can be found here.

CHAPTER

FOURTEEN

0.16.1

14.1 Changelog

14.1.1 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

CHAPTER

FIFTEEN

0.16

15.1 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.

15.2 Changelog

15.2.1 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 `lda.LDA` 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 untom.
- Added `calibration.CalibratedClassifierCV`, an approach for calibrating the predicted probabilities of a classifier. By Alexandre Gramfort, Jan Hendrik Metzen, Mathieu Blondel and Balazs Kegl.

15.2.2 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 paramter. 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 R V 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 eigen vectors. 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.

15.2.3 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*

15.2.4 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.

15.3 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 R V.
- 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.PassiveAgressiveClassivier` and `linear_model.PassiveAgressiveRegressor` now defaults to `True`.
- `cluster.DBSCAN` now uses a deterministic initialization. The *random_state* parameter is deprecated. By Eric Schubert.

CHAPTER

SIXTEEN

0.15.2

16.1 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 lda.LDA 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.

CHAPTER

SEVENTEEN

0.15.1

17.1 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.

CHAPTER

EIGHTEEN

0.15

18.1 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.

18.2 Changelog

18.2.1 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.

18.2.2 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.
- 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.

18.2.3 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.

18.2.4 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.

18.3 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.

18.4 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 Saeger
- 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

CHAPTER

NINETEEN

0.14

19.1 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

19.2 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.

19.3 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

CHAPTER

TWENTY

0.13.1

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

20.1 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.

20.2 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

CHAPTER

TWENTYONE

0.13

21.1 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,

21.2 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.

21.3 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__.

21.4 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

CHAPTER TWENTYTWO

0.12.1

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

22.1 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

22.2 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

CHAPTER

TWENTYTHREE

0.12

23.1 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*.

23.2 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: qda.QDA.decision_function and lda.LDA.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 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.

23.3 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

CHAPTER

TWENTYFOUR

0.11

24.1 Changelog

24.1.1 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 models*: 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.

24.1.2 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 *Grid Search: Searching for estimator parameters* 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 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`.

24.2 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.

24.3 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

CHAPTER

TWENTYFIVE

0.10

25.1 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 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 multi-class).
- 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.

25.2 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.

25.3 People

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

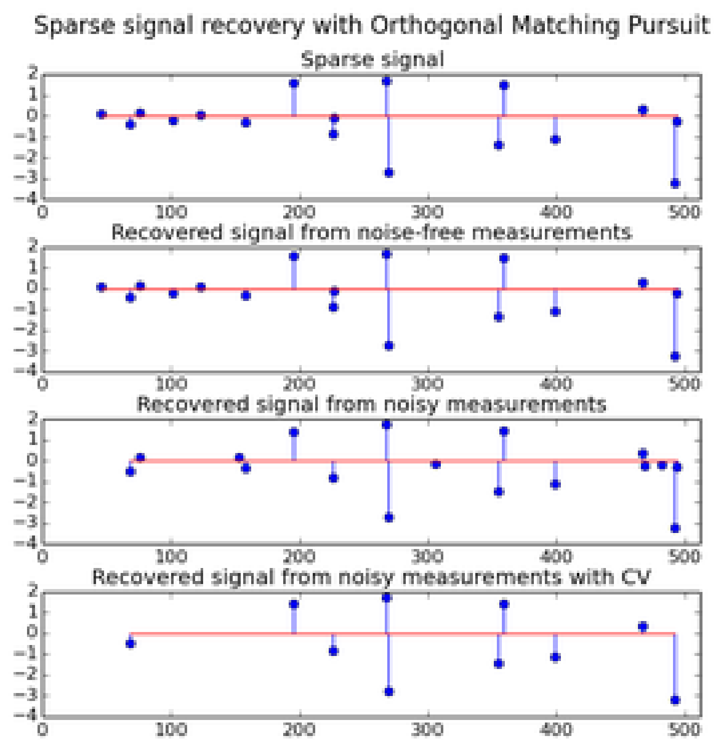
- 246 Andreas Müller
- 242 Olivier Grisel
- 220 Gilles Louppe
- 183 Brian Holt
- 166 Gael Varoquaux
- 144 Lars Buitinck
- 73 Vlad Niculae
- 65 Peter Prettenhofer
- 64 Fabian Pedregosa
- 60 Robert Layton
- 55 Mathieu Blondel
- 52 Jake Vanderplas
- 44 Noel Dawe
- 38 Alexandre Gramfort
- 24 Virgile Fritsch
- 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

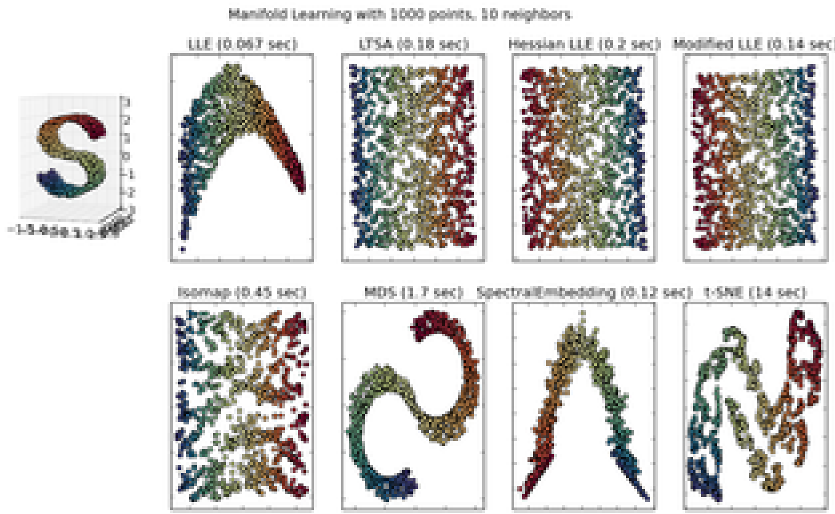
CHAPTER TWENTYSIX

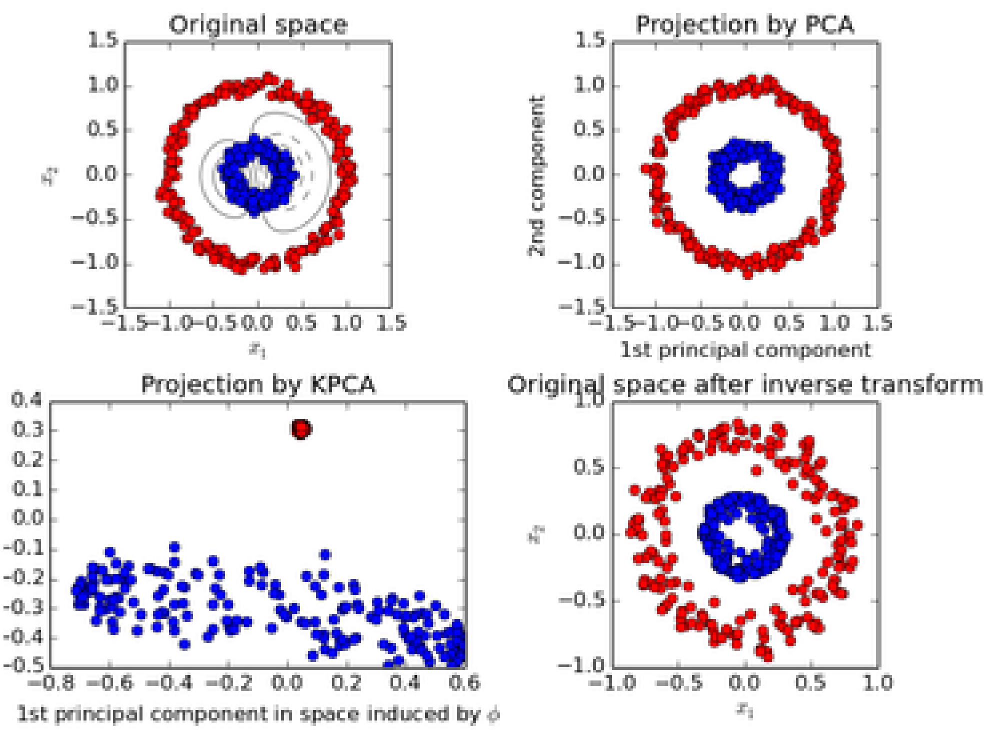
0.9

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.







26.1 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.
- *Sample generators* 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.
- *Downloading datasets from the mldata.org repository* utilities by Pietro Berkes.
- *The Olivetti faces dataset* by David Warde-Farley.

26.2 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.

26.3 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

CHAPTER

TWENTYSEVEN

0.8

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.

27.1 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 `lda.LDA` 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.

27.2 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

CHAPTER

TWENTYEIGHT

0.7

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.

28.1 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]

28.2 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

CHAPTER

TWENTYNINE

0.6

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.

29.1 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 *Gaussian Processes regression: basic introductory example* or *Gaussian Processes classification example: exploiting the probabilistic output* 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.

29.2 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

CHAPTER

THIRTY

0.5

30.1 Changelog

30.2 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

30.3 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.

30.4 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.

30.5 Examples

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

30.6 External dependencies

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

30.7 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.

30.8 Misc

- New sphinx theme for the web page.

30.9 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

CHAPTER

THIRTYONE

0.4

31.1 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 ...

31.2 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

CHAPTER THIRTYTWO

EARLIER VERSIONS

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

CHAPTER

THIRTYTHREE

EXTERNAL RESOURCES, VIDEOS AND TALKS

For written tutorials, see the *Tutorial section* of the documentation.

33.1 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.

33.2 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

33.3 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 the scikit, 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.

CHAPTER THIRTYFOUR

ABOUT US

This is a community effort, and as such many people have contributed to it over the years.

34.1 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.

34.2 People

- David Cournapeau
- Jarrod Millman
- Matthieu Brucher
- Fabian Pedregosa
- Gael Varoquaux
- Jake VanderPlas
- Alexandre Gramfort
- Olivier Grisel
- Bertrand Thirion
- Vincent Michel
- Chris Filo Gorgolewski
- Angel Soler Gollonet
- Yaroslav Halchenko
- Ron Weiss
- Virgile Fritsch
- Mathieu Blondel
- Peter Prettenhofer
- Vincent Dubourg
- Alexandre Passos
- Vlad Niculae
- Edouard Duchesnay
- Thouis (Ray) Jones
- Lars Buitinck
- Paolo Losi
- Nelle Varoquaux

- Brian Holt
- Robert Layton
- Gilles Louppe
- Andreas Müller (release manager)
- Satra Ghosh
- Wei Li
- Arnaud Joly
- Kemal Eren
- Michael Becker

34.3 Citing scikit-learn

If you use scikit-learn in 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}
}
```

34.4 Artwork

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

34.5 Funding

INRIA actively supports this project. It has provided funding for Fabian Pedregosa (2010-2012), Jaques Grobler (2012-2013) and Olivier Grisel (2013-2015) 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).

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.

It also provided funding for sprints and events around scikit-learn. If you would like to participate in the next Google Summer of code program, please see this page

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

The PSF helped find and manage funding for our 2011 Granada sprint. More information can be found here

tinyclues funded the 2011 international Granada sprint.

34.5.1 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

34.5.2 The 2013' Paris international sprint

Figure 34.1: IAP VII/19 - DYSCO

For more information on this sprint, see here

[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.

34.6 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 Shining Panda for free CPU time on their Continuous Integration server.

CHAPTER

THIRTYFIVE

DOCUMENTATION OF SCIKIT-LEARN 0.16.1

CHAPTER

THIRTYSIX

DATASET LOADING UTILITIES

The sklearn.datasets package embeds some small toy datasets as introduced in the *Getting Started* section.

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.

This package also features helpers to fetch larger datasets commonly used by the machine learning community to benchmark algorithm on data that comes from the 'real world'.

36.1 General dataset API

There are three distinct kinds of dataset interfaces for different types of datasets. The simplest one is the interface for sample images, which is described below in the *Sample images* section.

The dataset generation functions and the svmlight loader share a simplistic interface, returning 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.

The toy datasets as well as the 'real world' datasets and the datasets fetched from mldata.org have more sophisticated structure. These 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.

The datasets also contain a description in DESCR and some contain feature_names and target_names. See the dataset descriptions below for details.

36.2 Toy datasets

scikit-learn comes with a few small standard datasets that do not require to download any file from some external website.

load_boston()	Load and return the boston house-prices dataset (regression).
load_iris()	Load and return the iris dataset (classification).
load_diabetes()	Load and return the diabetes dataset (regression).
load_digits([n_class])	Load and return the digits dataset (classification).
load_linnerud()	Load and return the linnerud dataset (multivariate regression).

These datasets are useful to quickly illustrate the behavior of the various algorithms implemented in the scikit. They are however often too small to be representative of real world machine learning tasks.

36.3 Sample images

The scikit also embed a couple of sample JPEG images published under Creative Commons license by their authors. Those image 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

Original image (96,615 colors)

Warning: The default coding of images is based on the uint8 dtype to spare memory. Ofte algorithms work best if the input is converted to a floating point representation first. Also, pylab.imshow don't forget to scale to the range 0 - 1 as done in the following example.

Examples:

- *Color Quantization using K-Means*

36.4 Sample generators

In addition, scikit-learn includes various random sample generators that can be used to build artificial datasets of controlled size and complexity.

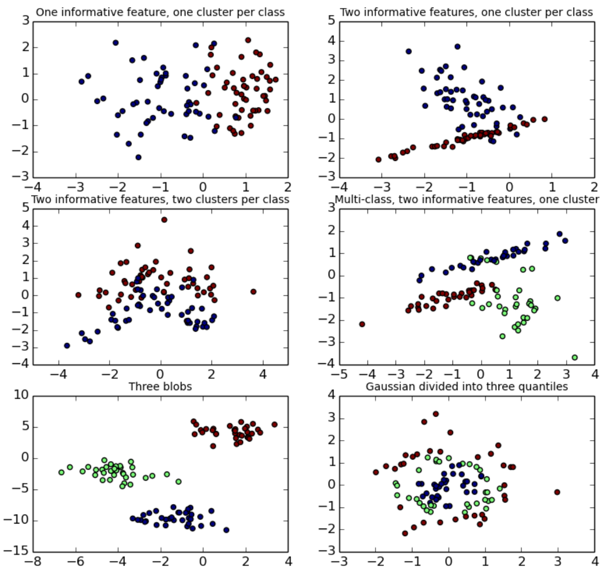
36.4.1 Generators for classification and clustering

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.

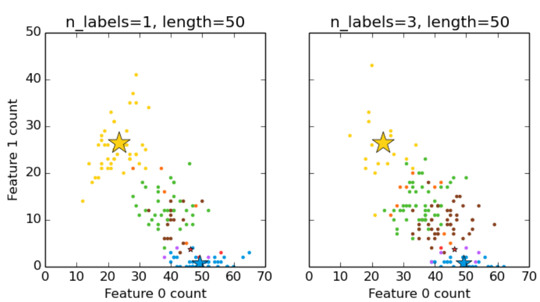


make_circles and make_moons generate 2d binary classification datasets that are challenging to certain algorithms (e.g. centroid-based clustering or linear classification), including optional Gaussian noise. They are useful for visualisation. produces Gaussian data with a spherical decision boundary for binary classification.

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.

Continued on next page

Table 36.3 – continued from previous page

make_checkerboard(shape, n_clusters[, ...])	Generate an array with block checkerboard structure for biclustering.

36.4.2 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.

36.4.3 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.

36.4.4 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.

36.5 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: http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
Faster API-compatible implementation: https://github.com/mblondel/svmlight-loader

36.6 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 website describing the original dataset is now defunct, but archived copies can be accessed through the Internet Archive's Wayback Machine. 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).

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.

36.7 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.

36.7.1 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))
['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([12,  6,  9,  8,  6,  7,  9,  2, 13, 19])
```

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([1, 1, 1, 0, 1, 0, 0, 1, 1, 1])
```

36.7.2 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',
...               '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])
159.01327433628319
```

`sklearn.datasets.fetch_20newsgroups_vectorized` is a function which returns ready-to-use tfidf features instead of file names.

36.7.3 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)
>>> pred = clf.predict(vectors_test)
>>> metrics.f1_score(newsgroups_test.target, pred, average='weighted')
0.88251152461278892
```

(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: sgi livesey atheists writes people caltech com god keith edu
comp.graphics: organization thanks files subject com image lines university edu graphics
sci.space: toronto moon gov com alaska access henry nasa edu space
talk.religion.misc: article writes kent people christian jesus sandvik edu com god
```

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='weighted')
0.78409163025839435
```

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 = BernoulliNB(alpha=.01)
>>> clf.fit(vectors, newsgroups_train.target)
>>> vectors_test = vectorizer.transform(newsgroups_test.data)
>>> pred = clf.predict(vectors_test)
>>> metrics.f1_score(newsgroups_test.target, pred, average='weighted')
0.73160869205141166
```

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*

36.8 Downloading datasets from the mldata.org repository

mldata.org is a public repository for machine learning data, supported by the PASCAL network .

The sklearn.datasets package is able to directly download data sets from the repository using the function sklearn.datasets.fetch_mldata.

For example, to download the MNIST digit recognition database:

```
>>> from sklearn.datasets import fetch_mldata
>>> mnist = fetch_mldata('MNIST original', data_home=custom_data_home)
```

The MNIST database contains a total of 70000 examples of handwritten digits of size 28x28 pixels, labeled from 0 to 9:

```
>>> mnist.data.shape
(70000, 784)
>>> mnist.target.shape
(70000,)
>>> np.unique(mnist.target)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

After the first download, the dataset is cached locally in the path specified by the data_home keyword argument, which defaults to ~/scikit_learn_data/:

```
>>> os.listdir(os.path.join(custom_data_home, 'mldata'))
['mnist-original.mat']
```

Data sets in mldata.org do not adhere to a strict naming or formatting convention. sklearn.datasets.fetch_mldata is able to make sense of the most common cases, but allows to tailor the defaults to individual datasets:

- The data arrays in mldata.org are most often shaped as (n_features, n_samples). This is the opposite of the scikit-learn convention, so sklearn.datasets.fetch_mldata transposes the matrix by default. The transpose_data keyword controls this behavior:

```
>>> iris = fetch_mldata('iris', data_home=custom_data_home)
>>> iris.data.shape
(150, 4)
>>> iris = fetch_mldata('iris', transpose_data=False,
...                     data_home=custom_data_home)
>>> iris.data.shape
(4, 150)
```

- For datasets with multiple columns, sklearn.datasets.fetch_mldata tries to identify the target and data columns and rename them to target and data. This is done by looking for arrays named label and data in the dataset, and failing that by choosing the first array to be target and the second to be data. This behavior can be changed with the target_name and data_name keywords, setting them to a specific name or index number (the name and order of the columns in the datasets can be found at its mldata.org under the tab "Data":

```
>>> iris2 = fetch_mldata('datasets-UCI iris', target_name=1, data_name=0,
...                      data_home=custom_data_home)
>>> iris3 = fetch_mldata('datasets-UCI iris', target_name='class',
...                      data_name='double0', data_home=custom_data_home)
```

36.9 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.

36.9.1 Usage

`scikit-learn` provides two loaders that will automatically download, cache, parse the metadata files, decode the jpeg and convert the interesting slices into memmaped 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 memmaped 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.

36.9.2 Examples

Faces recognition example using eigenfaces and SVMs

36.10 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.

`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.

CHAPTER

THIRTYSEVEN

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.

`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.

CHAPTER

THIRTYEIGHT

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.

38.1 Usage

`scikit-learn` provides two loaders that will automatically download, cache, parse the metadata files, decode the jpeg and convert the interesting slices into memmaped 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 memmaped 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.

38.2 Examples

Faces recognition example using eigenfaces and SVMs

CHAPTER

THIRTYNINE

DOWNLOADING DATASETS FROM THE MLDATA.ORG REPOSITORY

mldata.org is a public repository for machine learning data, supported by the PASCAL network .

The `sklearn.datasets` package is able to directly download data sets from the repository using the function `sklearn.datasets.fetch_mldata`.

For example, to download the MNIST digit recognition database:

```
>>> from sklearn.datasets import fetch_mldata
>>> mnist = fetch_mldata('MNIST original', data_home=custom_data_home)
```

The MNIST database contains a total of 70000 examples of handwritten digits of size 28x28 pixels, labeled from 0 to 9:

```
>>> mnist.data.shape
(70000, 784)
>>> mnist.target.shape
(70000,)
>>> np.unique(mnist.target)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

After the first download, the dataset is cached locally in the path specified by the `data_home` keyword argument, which defaults to `~/scikit_learn_data/`:

```
>>> os.listdir(os.path.join(custom_data_home, 'mldata'))
['mnist-original.mat']
```

Data sets in mldata.org do not adhere to a strict naming or formatting convention. `sklearn.datasets.fetch_mldata` is able to make sense of the most common cases, but allows to tailor the defaults to individual datasets:

- The data arrays in mldata.org are most often shaped as `(n_features, n_samples)`. This is the opposite of the `scikit-learn` convention, so `sklearn.datasets.fetch_mldata` transposes the matrix by default. The `transpose_data` keyword controls this behavior:

  ```
  >>> iris = fetch_mldata('iris', data_home=custom_data_home)
  >>> iris.data.shape
  (150, 4)
  >>> iris = fetch_mldata('iris', transpose_data=False,
  ...                     data_home=custom_data_home)
  >>> iris.data.shape
  (4, 150)
  ```

- For datasets with multiple columns, `sklearn.datasets.fetch_mldata` tries to identify the target and data columns and rename them to `target` and `data`. This is done by looking for arrays named `label` and `data` in the dataset, and failing that by choosing the first array to be `target` and the second to be `data`. This behavior can be changed with the `target_name` and `data_name` keywords, setting them to a specific name or index number (the name and order of the columns in the datasets can be found at its mldata.org under the tab "Data":

```
>>> iris2 = fetch_mldata('datasets-UCI iris', target_name=1, data_name=0,
...                      data_home=custom_data_home)
>>> iris3 = fetch_mldata('datasets-UCI iris', target_name='class',
...                      data_name='double0', data_home=custom_data_home)
```

CHAPTER

FORTY

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 website describing the original dataset is now defunct, but archived copies can be accessed through the Internet Archive's Wayback Machine. 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).

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.

CHAPTER

FORTYONE

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.

41.1 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))
['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([12,  6,  9,  8,  6,  7,  9,  2, 13, 19])
```

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([1, 1, 1, 0, 1, 0, 0, 1, 1, 1])
```

41.2 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',
...               '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])
159.01327433628319
```

sklearn.datasets.fetch_20newsgroups_vectorized is a function which returns ready-to-use tfidf features instead of file names.

41.3 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)
>>> pred = clf.predict(vectors_test)
>>> metrics.f1_score(newsgroups_test.target, pred, average='weighted')
0.88251152461278892
```

(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: sgi livesey atheists writes people caltech com god keith edu
comp.graphics: organization thanks files subject com image lines university edu graphics
sci.space: toronto moon gov com alaska access henry nasa edu space
talk.religion.misc: article writes kent people christian jesus sandvik edu com god
```

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='weighted')
0.78409163025839435
```

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 = BernoulliNB(alpha=.01)
>>> clf.fit(vectors, newsgroups_train.target)
>>> vectors_test = vectorizer.transform(newsgroups_test.data)
>>> pred = clf.predict(vectors_test)
>>> metrics.f1_score(newsgroups_test.target, pred, average='weighted')
0.73160869205141166
```

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*

CHAPTER

FORTYTWO

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.

42.1 `sklearn.base`: Base classes and utility functions

Base classes for all estimators.

42.1.1 Base classes

`base.BaseEstimator`	Base class for all 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.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`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.

`__init__`()

static **`get_params`**(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.base.BaseEstimator`

Figure 42.1: *Feature Union with Heterogeneous Data Sources*

`sklearn.base`.ClassifierMixin

class `sklearn.base.`**`ClassifierMixin`**

Mixin class for all classifiers in scikit-learn.

Methods

`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.

`__init__`()

static score (*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.

ParametersX : 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.

Returnsscore : 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`(X[, y])	Performs clustering on X and returns cluster labels.

`__init__`()

static **`fit_predict`**(*X*, *y=None*)

Performs clustering on X and returns cluster labels.

ParametersX : ndarray, shape (n_samples, n_features)

Input data.

Returnsy : ndarray, shape (n_samples,)

cluster labels

`sklearn.base`.RegressorMixin

class `sklearn.base.`**`RegressorMixin`**

Mixin class for all regression estimators in scikit-learn.

Methods

`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.

`__init__`()

static **`score`**(*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

sklearn.base.TransformerMixin

class sklearn.base.**TransformerMixin**

Mixin class for all transformers in scikit-learn.

Methods

fit_transform(X[, y])	Fit to data, then transform it.

__init__()

static fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

Examples using sklearn.base.TransformerMixin

42.1.2 Functions

base.clone(estimator[, safe])	Constructs a new estimator with the same parameters.

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.

Figure 42.2: *Feature Union with Heterogeneous Data Sources*

Parametersestimator: 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 deepcopy on objects that are not estimators.

42.2 `sklearn.cluster`: Clustering

The `sklearn.cluster` module gathers popular unsupervised clustering algorithms.

User guide: See the *Clustering* section for further details.

42.2.1 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.SpectralClustering`([n_clusters, ...])	Apply clustering to a projection to the normalized laplacian.
`cluster.Ward`([n_clusters, memory, ...])	Ward hierarchical clustering: constructs a tree and cuts it.

`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.

Parametersdamping : float, optional, default: 0.5

Damping factor between 0.5 and 1.

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.

affinity : string, optional, default=‘‘euclidean‘‘

Which affinity to use. At the moment `precomputed` and `euclidean` are supported. `euclidean` uses the negative squared euclidean distance between points.

verbose : boolean, optional, default: False

Whether to be verbose.

Attributescluster_centers_indices_ : array, shape (n_clusters,)

Indices of cluster centers

cluster_centers_ : array, shape (n_clusters, n_features)

Cluster centers (if affinity != `precomputed`).

labels_ : array, shape (n_samples,)

Labels of each point

affinity_matrix_ : array, shape (n_samples, n_samples)

Stores the affinity matrix used in `fit`.

n_iter_ : int

Number of iterations taken to converge.

Notes

See examples/cluster/plot_affinity_propagation.py for an example.

The algorithmic complexity of affinity propagation is quadratic in the number of points.

References

Brendan J. Frey and Delbert Dueck, “Clustering by Passing Messages Between Data Points”, Science Feb. 2007

Methods

`fit`(X[, y])	Create affinity matrix from negative euclidean distances, then apply affinity propagation clustering.
`fit_predict`(X[, y])	Performs clustering on X and returns cluster labels.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict the closest cluster each sample in X belongs to.
`set_params`(**params)	Set the parameters of this estimator.

static __init__ (*damping=0.5*, *max_iter=200*, *convergence_iter=15*, *copy=True*, *preference=None*, *affinity='euclidean'*, *verbose=False*)

static fit (*X*, *y=None*)

Create affinity matrix from negative euclidean distances, then apply affinity propagation clustering.

ParametersX: array-like, shape (n_samples, n_features) or (n_samples, n_samples) :

Data matrix or, if affinity is `precomputed`, matrix of similarities / affinities.

fit_predict (*X*, *y=None*)

Performs clustering on X and returns cluster labels.

ParametersX : ndarray, shape (n_samples, n_features)

Input data.

Returnsy : ndarray, shape (n_samples,)

cluster labels

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **predict** (*X*)

Predict the closest cluster each sample in X belongs to.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

New data to predict.

Returnslabels : array, shape (n_samples,)

Index of the cluster each sample belongs to.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.cluster.AffinityPropagation`

`sklearn.cluster`.AgglomerativeClustering

class sklearn.cluster.**AgglomerativeClustering** (*n_clusters=2, affinity='euclidean', memory=Memory(cachedir=None), connectivity=None, n_components=None, compute_full_tree='auto', linkage='ward', pooling_func=<function mean at 0x2ad3c0a06f50>*)

Agglomerative Clustering

Recursively merges the pair of clusters that minimally increases a given linkage distance.

Parametersn_clusters : int, default=2

The number of clusters to find.

connectivity : array-like or callable, optional

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.

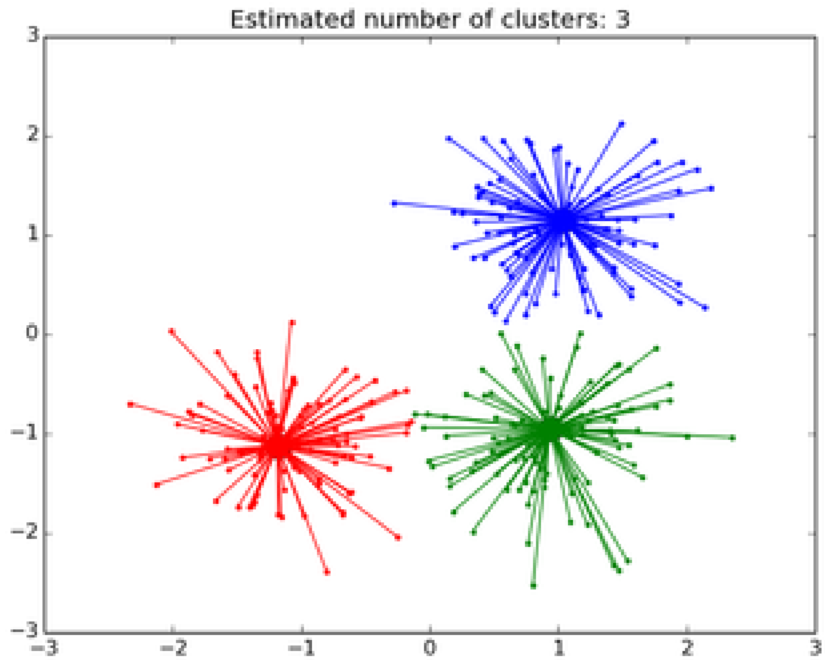


Figure 42.3: *Demo of affinity propagation clustering algorithm*

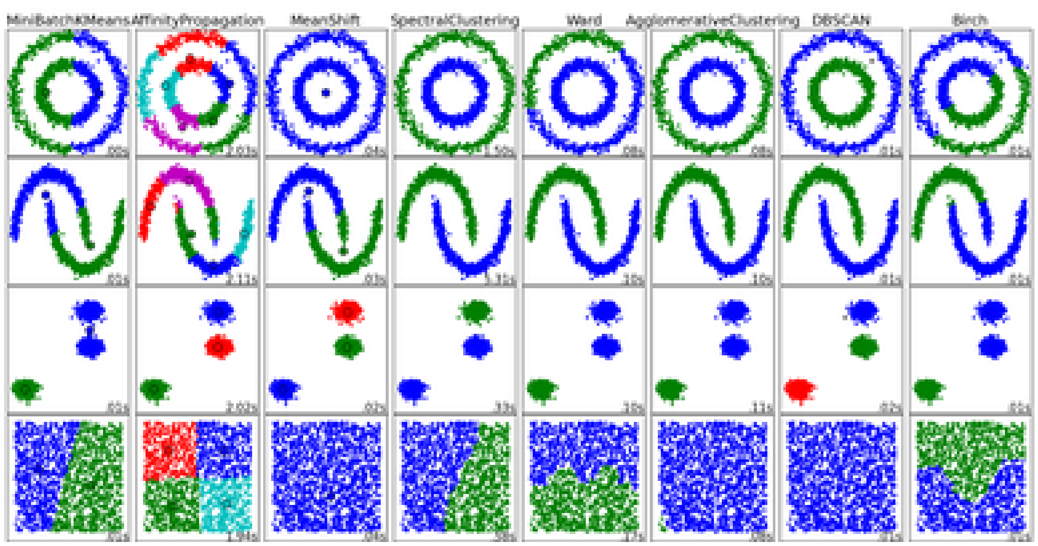


Figure 42.4: *Comparing different clustering algorithms on toy datasets*

affinity : string 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 : Instance of joblib.Memory or string (optional)

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.

n_components : int (optional)

Number of connected components. If None the number of connected components is estimated from the connectivity matrix. NOTE: This parameter is now directly determined from the connectivity matrix and will be removed in 0.18

compute_full_tree : bool or 'auto' (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. 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.

linkage : {"ward", "complete", "average"}, optional, 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.

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].

Attributeslabels_ : array [n_samples]

cluster labels for each point

n_leaves_ : int

Number of leaves in the hierarchical tree.

n_components_ : int

The estimated number of connected components in the graph.

children_ : array-like, 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*

Methods

`fit`(X[, y])	Fit the hierarchical clustering on the data
`fit_predict`(X[, y])	Performs clustering on X and returns cluster labels.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*n_clusters=2*, *affinity='euclidean'*, *memory=Memory(cachedir=None)*, *connectivity=None*, *n_components=None*, *compute_full_tree='auto'*, *linkage='ward'*, *pooling_func=<function mean at 0x2ad3c0a06f50>*)

static `fit`(*X*, *y=None*)

Fit the hierarchical clustering on the data

ParametersX : array-like, shape = [n_samples, n_features]

The samples a.k.a. observations.

Returnsself :

`fit_predict`(*X*, *y=None*)

Performs clustering on X and returns cluster labels.

ParametersX : ndarray, shape (n_samples, n_features)

Input data.

Returnsy : ndarray, shape (n_samples,)

cluster labels

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.cluster.AgglomerativeClustering`

`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.

Every new sample is inserted into the root of the Clustering Feature Tree. It is then clubbed together with the subcluster that has the centroid closest to the new sample. This is done recursively till it ends up at the subcluster of the leaf of the tree has the closest centroid.

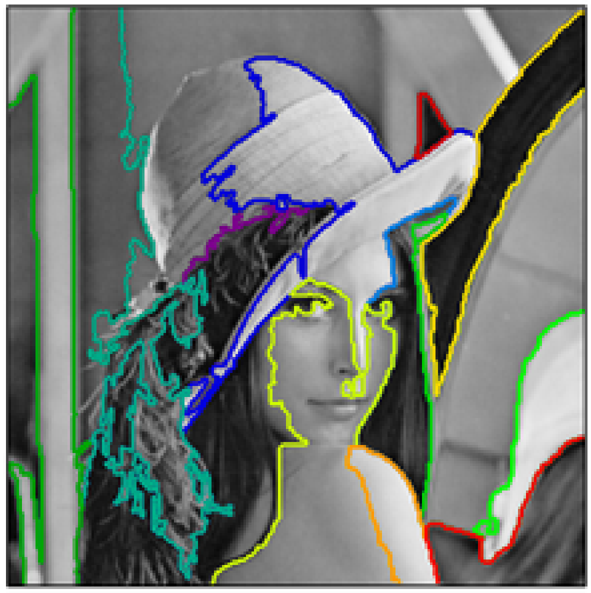


Figure 42.5: *A demo of structured Ward hierarchical clustering on Lena image*

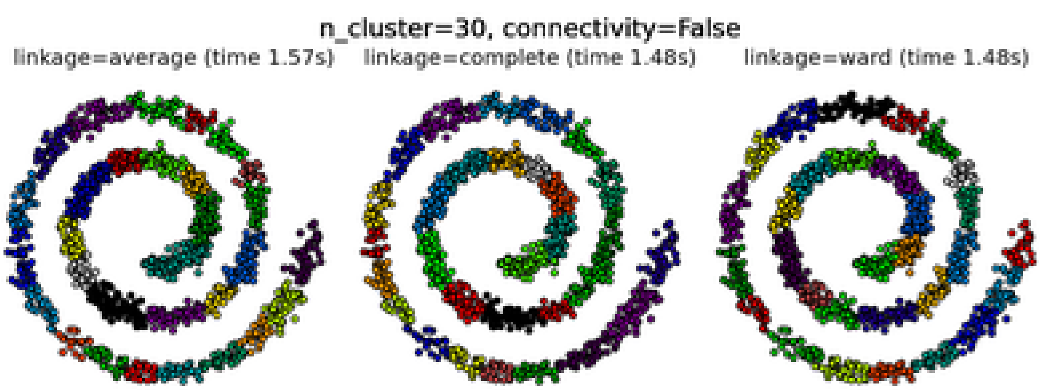


Figure 42.6: *Agglomerative clustering with and without structure*

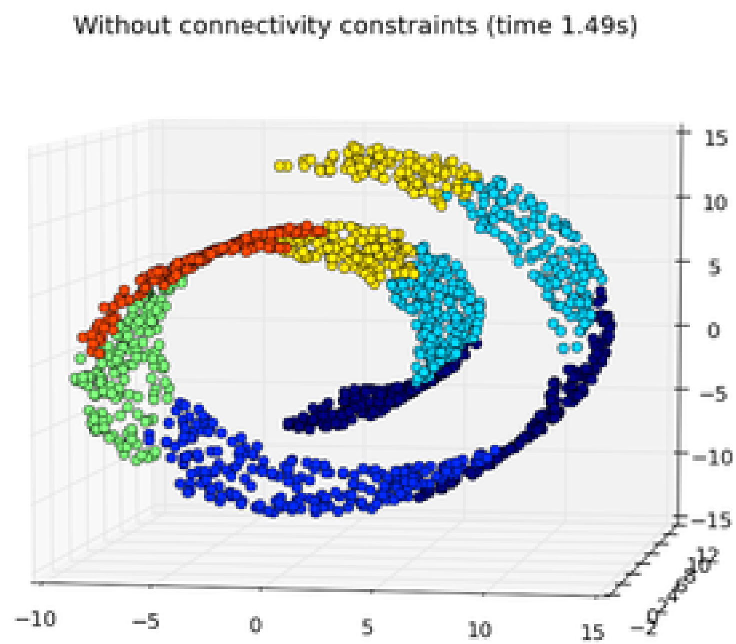


Figure 42.7: *Hierarchical clustering: structured vs unstructured ward*

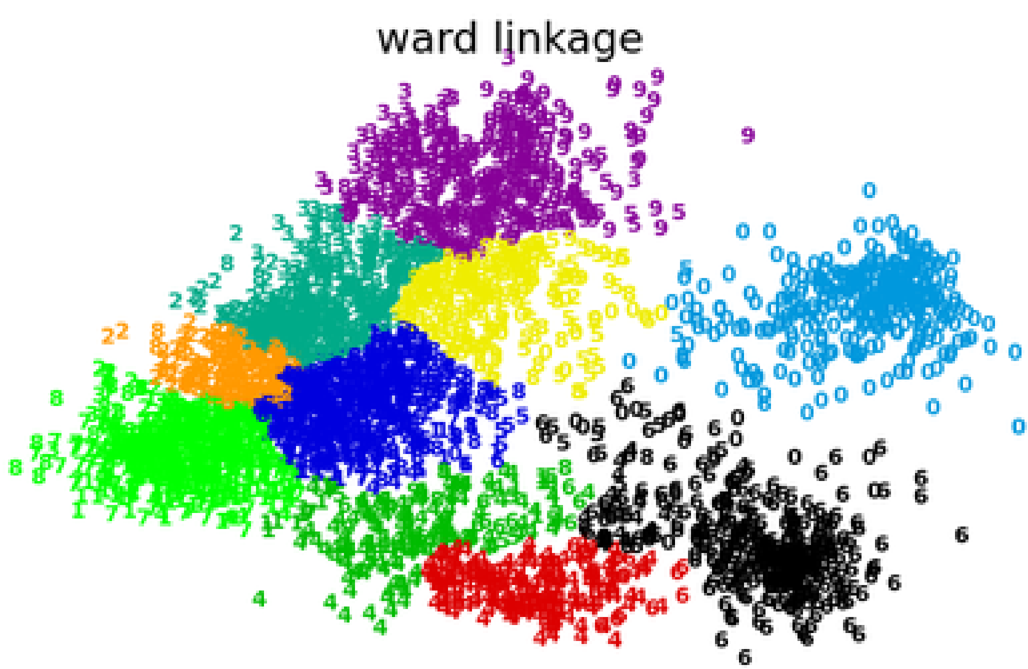


Figure 42.8: *Various Agglomerative Clustering on a 2D embedding of digits*

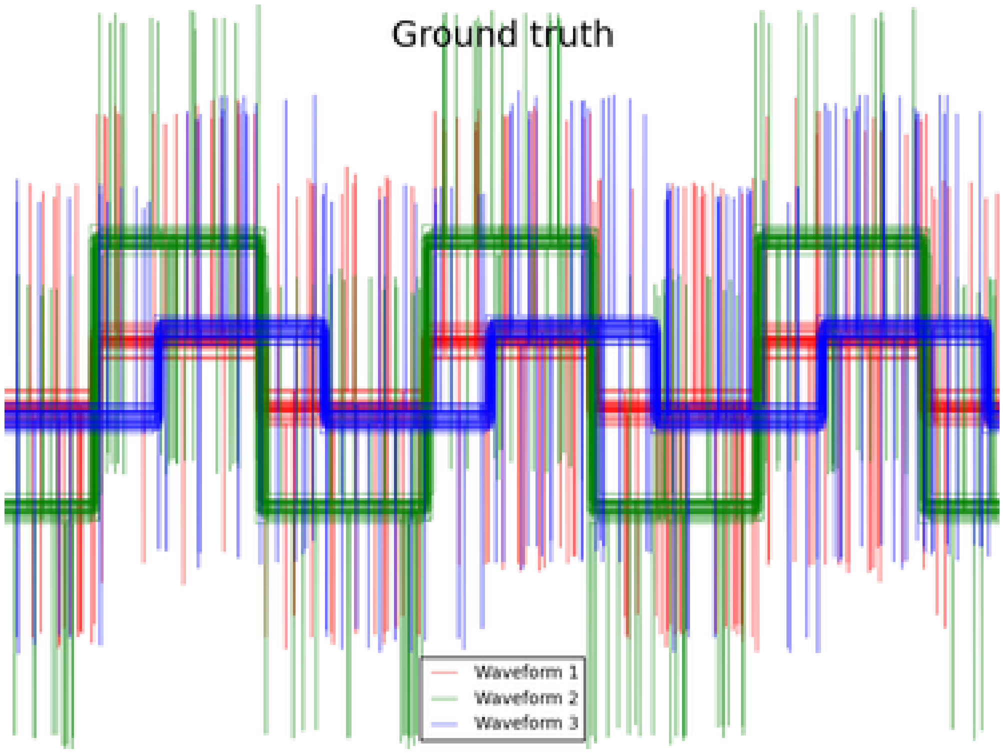


Figure 42.9: *Agglomerative clustering with different metrics*

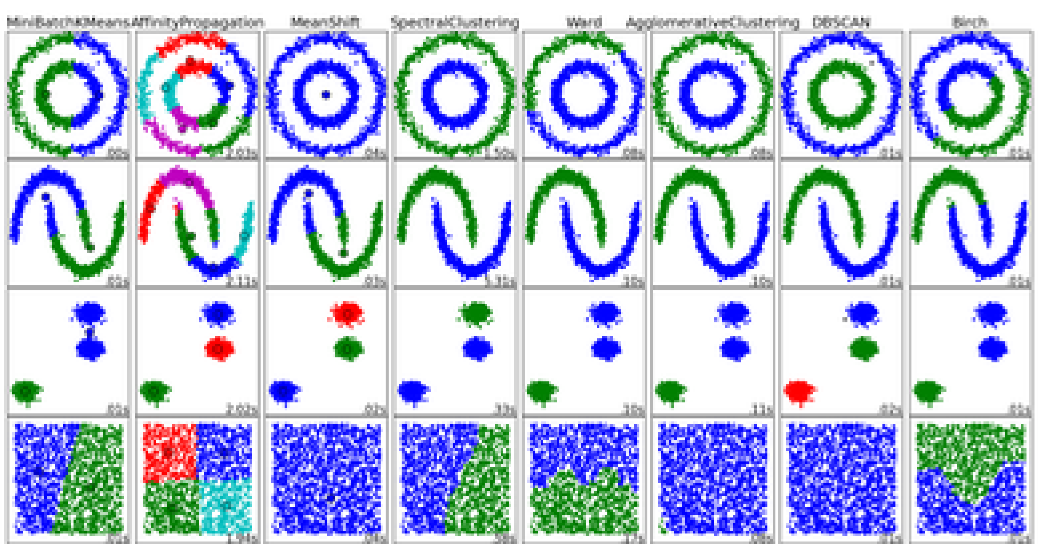


Figure 42.10: *Comparing different clustering algorithms on toy datasets*

Parametersthreshold : 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.

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 the node has to be split. The corresponding parent also has to be split and if the number of subclusters in the parent is greater than the branching factor, then it has to be split recursively.

n_clusters : int, instance of sklearn.cluster model, default None

Number of clusters after the final clustering step, which treats the subclusters from the leaves as new samples. By default, this final clustering step is not performed and the subclusters are returned as they are. 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. If an int is provided, the model fit is AgglomerativeClustering with n_clusters set 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.

Attributesroot_ : _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.

References

- Tian Zhang, Raghu Ramakrishnan, Maron Livny BIRCH: An efficient data clustering method for large databases. http://www.cs.sfu.ca/CourseCentral/459/han/papers/zhang96.pdf
- Roberto Perdisci JBirch - Java implementation of BIRCH clustering algorithm https://code.google.com/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(branching_factor=50, n_clusters=None, threshold=0.5,
... compute_labels=True)
>>> brc.fit(X)
Birch(branching_factor=50, compute_labels=True, copy=True, n_clusters=None,
   threshold=0.5)
>>> brc.predict(X)
array([0, 0, 0, 1, 1, 1])
```

Methods

`fit`(X[, y])	Build a CF Tree for the input data.
`fit_predict`(X[, y])	Performs clustering on X and returns cluster labels.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`partial_fit`([X, y])	Online learning.
`predict`(X)	Predict data using the `centroids_` of subclusters.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y])	Transform X into subcluster centroids dimension.

static `__init__` (*threshold=0.5, branching_factor=50, n_clusters=3, compute_labels=True, copy=True*)

static `fit` (*X, y=None*)

Build a CF Tree for the input data.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

Input data.

`fit_predict` (*X, y=None*)

Performs clustering on X and returns cluster labels.

ParametersX : ndarray, shape (n_samples, n_features)

Input data.

Returnsy : ndarray, shape (n_samples,)

cluster labels

`fit_transform` (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **partial_fit** (*X=None*, *y=None*)
Online learning. Prevents rebuilding of CFTree from scratch.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features), None

Input data. If X is not provided, only the global clustering step is done.

static **predict** (*X*)
Predict data using the `centroids_` of subclusters.

Avoid computation of the row norms of X.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

Input data.

Returnslabels: ndarray, shape(n_samples) :

Labelled data.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **transform** (*X*, *y=None*)
Transform X into subcluster centroids dimension.

Each dimension represents the distance from the sample point to each cluster centroid.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

Input data.

ReturnsX_trans : {array-like, sparse matrix}, shape (n_samples, n_clusters)

Transformed data.

Examples using `sklearn.cluster.Birch`

`sklearn.cluster`.DBSCAN

class `sklearn.cluster.`**DBSCAN** (*eps=0.5*, *min_samples=5*, *metric='euclidean'*, *algorithm='auto'*, *leaf_size=30*, *p=None*, *random_state=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.

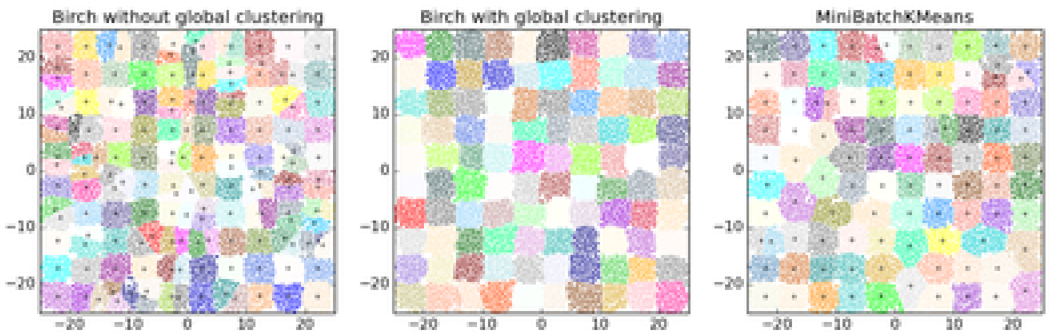


Figure 42.11: *Compare BIRCH and MiniBatchKMeans*

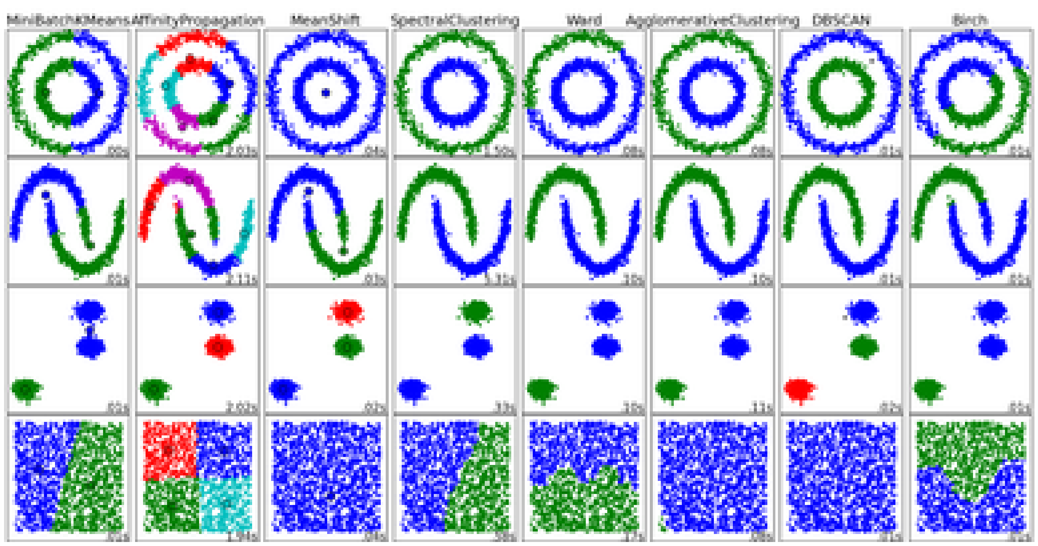


Figure 42.12: *Comparing different clustering algorithms on toy datasets*

Parameterseps : float, optional

The maximum distance between two samples for them to be considered as in the same neighborhood.

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 metrics.pairwise.calculate_distance for its metric parameter. If metric is "precomputed", X is assumed to be a distance matrix and must be square.

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.

random_state: numpy.RandomState, optional :

Deprecated and ignored as of version 0.16, will be removed in version 0.18. DBSCAN does not use random initialization.

Attributescore_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.

Notes

See examples/cluster/plot_dbscan.py for an example.

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).

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

Methods

`fit`(X[, y, sample_weight])	Perform DBSCAN clustering from features or distance matrix.
`fit_predict`(X[, y, sample_weight])	Performs clustering on X and returns cluster labels.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*eps=0.5*, *min_samples=5*, *metric='euclidean'*, *algorithm='auto'*, *leaf_size=30*, *p=None*, *random_state=None*)

static `fit`(*X*, *y=None*, *sample_weight=None*)

Perform DBSCAN clustering from features or distance matrix.

ParametersX : 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'`.

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.

static `fit_predict`(*X*, *y=None*, *sample_weight=None*)

Performs clustering on X and returns cluster labels.

ParametersX : 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'`.

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.

Returnsy : ndarray, shape (n_samples,)

cluster labels

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.cluster.DBSCAN`

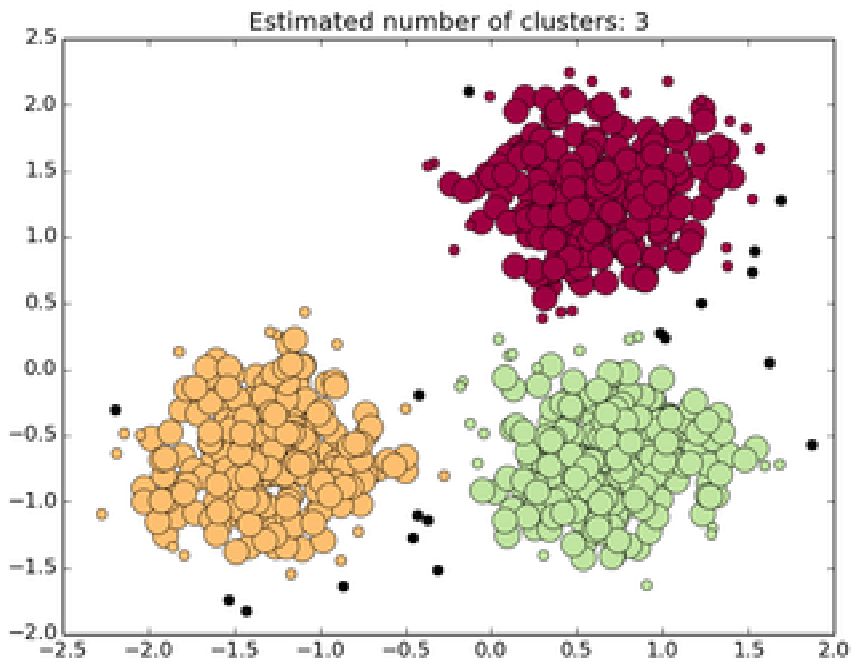


Figure 42.13: *Demo of DBSCAN clustering algorithm*

`sklearn.cluster`.FeatureAgglomeration

class `sklearn.cluster.`**`FeatureAgglomeration`**(*n_clusters=2, affinity='euclidean', memory=Memory(cachedir=None), connectivity=None, n_components=None, compute_full_tree='auto', linkage='ward', pooling_func=<function mean at 0x2ad3c0a06f50>*)

Agglomerate features.

Similar to AgglomerativeClustering, but recursively merges features instead of samples.

Parametersn_clusters : int, default 2

The number of clusters to find.

connectivity : array-like or callable, optional

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.

affinity : string 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 : Instance of joblib.Memory or string, optional

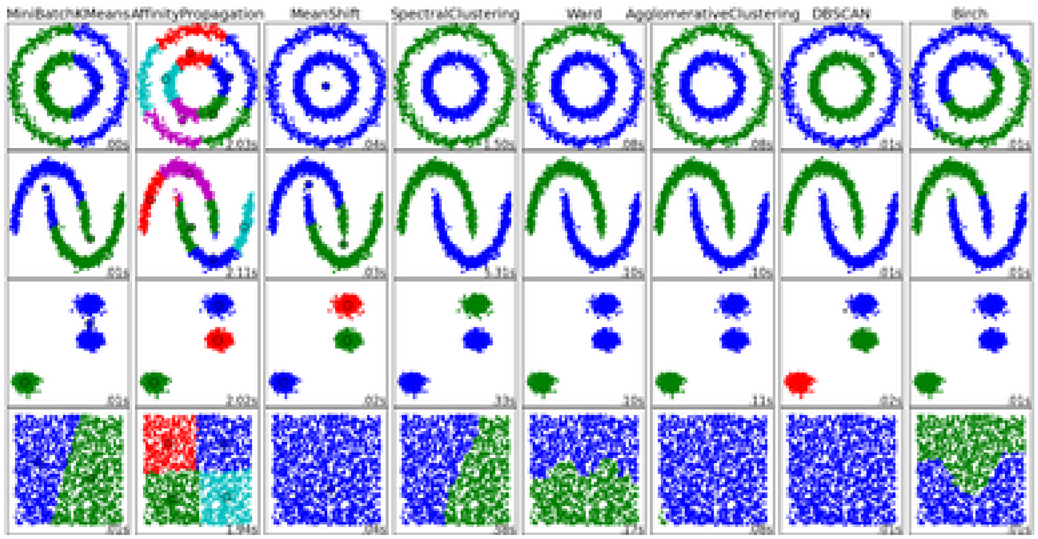


Figure 42.14: *Comparing different clustering algorithms on toy datasets*

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.

n_components : int (optional)

Number of connected components. If None the number of connected components is estimated from the connectivity matrix. NOTE: This parameter is now directly determined from the connectivity matrix and will be removed in 0.18

compute_full_tree : bool or 'auto', 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.

linkage : {"ward", "complete", "average"}, optional, 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.

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].

Attributeslabels_ : array-like, (n_features,)

cluster labels for each feature.

n_leaves_ : int

Number of leaves in the hierarchical tree.

n_components_ : int

The estimated number of connected components in the graph.

children_ : array-like, 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*

Methods

`fit`(X[, y])	Fit the hierarchical clustering on the data
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(Xred)	Inverse the transformation.
`pooling_func`(a[, axis, dtype, out, keepdims])	Compute the arithmetic mean along the specified axis.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, pooling_func])	Transform a new matrix using the built clustering

`__init__`(*n_clusters=2*, *affinity='euclidean'*, *memory=Memory(cachedir=None)*, *connectivity=None*, *n_components=None*, *compute_full_tree='auto'*, *linkage='ward'*, *pooling_func=<function mean at 0x2ad3c0a06f50>*)

static `fit`(*X*, *y=None*, ***params*)

Fit the hierarchical clustering on the data

ParametersX : array-like, shape = [n_samples, n_features]

The data

Returnsself :

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

inverse_transform(*Xred*)
Inverse the transformation. Return a vector of size nb_features with the values of Xred assigned to each group of features

ParametersXred : array-like, shape=[n_samples, n_clusters] or [n_clusters,]

The values to be assigned to each cluster of samples

ReturnsX : 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.

pooling_func(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

Parametersa : array_like

Array containing numbers whose mean is desired. If *a* is not an array, a conversion is attempted.

axis : int, optional

Axis along which the means are computed. The default is to compute the mean of the flattened array.

dtype : data-type, optional

Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.

out : ndarray, optional

Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returnsm : ndarray, see dtype parameter above

If *out=None*, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See also:

averageWeighted average

std, var, nanmean, nanstd, nanvar

Notes

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32* (see example below). Specifying a higher-precision accumulator using the *dtype* keyword can alleviate this issue.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([ 2.,  3.])
>>> np.mean(a, axis=1)
array([ 1.5,  3.5])
```

In single precision, *mean* can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.546875
```

Computing the mean in float64 is more accurate:

```
>>> np.mean(a, dtype=np.float64)
0.55000000074505806
```

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*, *pooling_func=None*)

Transform a new matrix using the built clustering

ParametersX : array-like, shape = [n_samples, n_features] or [n_features]

A M by N array of M observations in N dimensions or a length M array of M one-dimensional observations.

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].

ReturnsY : array, shape = [n_samples, n_clusters] or [n_clusters]

The pooled values for each feature cluster.

Examples using `sklearn.cluster.FeatureAgglomeration`

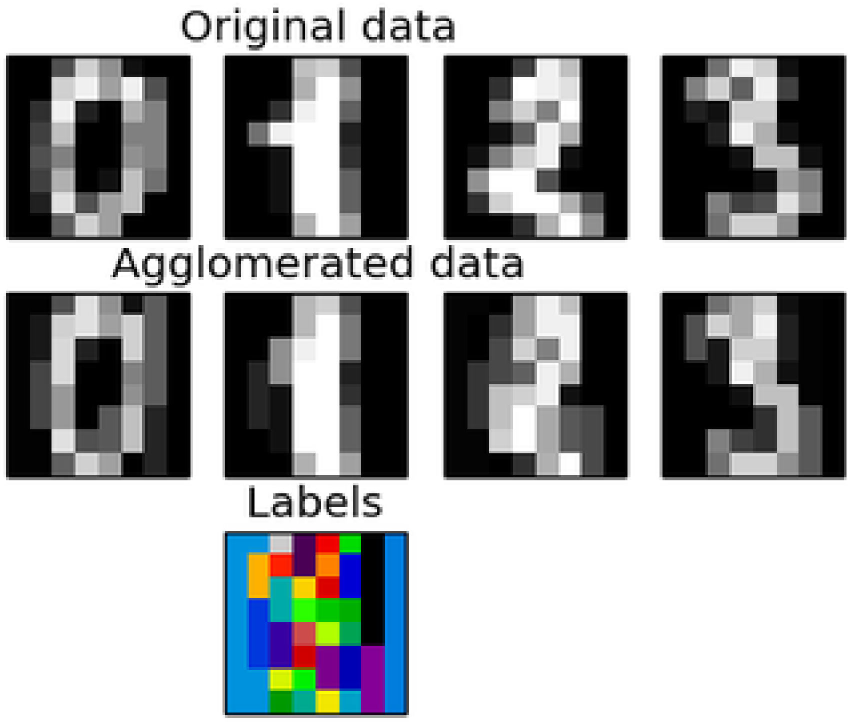


Figure 42.15: *Feature agglomeration*

`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=1*)

K-Means clustering

Parametersn_clusters : int, optional, default: 8

The number of clusters to form as well as the number of centroids to generate.

max_iter : int, default: 300

Maximum number of iterations of the k-means algorithm for a single run.

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.

init : {'k-means++', 'random' or an ndarray}

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.

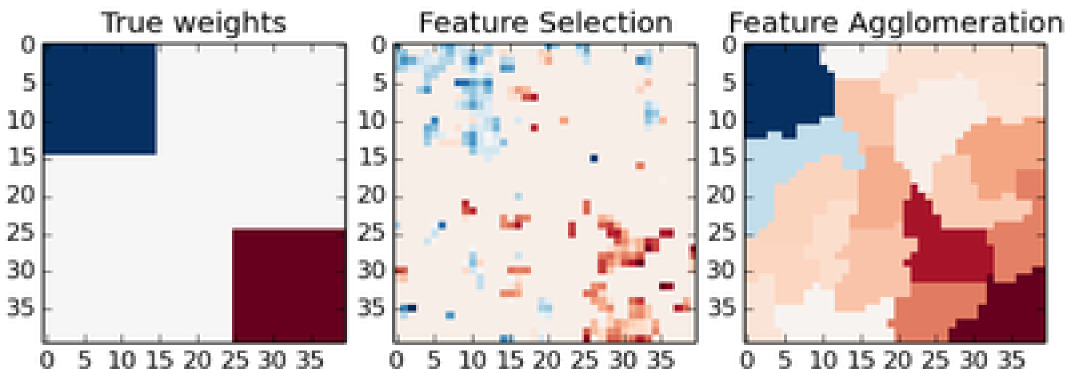


Figure 42.16: *Feature agglomeration vs. univariate selection*

If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.

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

tol : float, default: 1e-4

Relative tolerance with regards to inertia to declare convergence

n_jobs : int

The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

random_state : integer or numpy.RandomState, optional

The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

verbose : int, default 0

Verbosity mode.

copy_x : boolean, default True

When pre-computing distances it is more numerically accurate to center the data first. If copy_x is True, then the original data is not modified. 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.

Attributescluster_centers_ : array, [n_clusters, n_features]

Coordinates of cluster centers

labels_ : :

Labels of each point

inertia_ : float

Sum of distances of samples to their closest cluster center.

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 to than the default batch implementation.

Notes

The k-means problem is solved using Lloyd'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.

Methods

`fit`(X[, y])	Compute k-means clustering.
`fit_predict`(X[, y])	Compute cluster centers and predict cluster index for each sample.
`fit_transform`(X[, y])	Compute clustering and transform X to cluster-distance space.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict the closest cluster each sample in X belongs to.
`score`(X[, y])	Opposite of the value of X on the K-means objective.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y])	Transform X to a cluster-distance space.

static `__init__`(*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=1*)

static `fit`(*X*, *y=None*)

Compute k-means clustering.

ParametersX : array-like or sparse matrix, shape=(n_samples, n_features)

static **fit_predict**(*X*, *y=None*)
Compute cluster centers and predict cluster index for each sample.

Convenience method; equivalent to calling fit(X) followed by predict(X).

static **fit_transform**(*X*, *y=None*)
Compute clustering and transform X to cluster-distance space.

Equivalent to fit(X).transform(X), but more efficiently implemented.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **predict**(*X*)
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.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

New data to predict.

Returnslabels : array, shape [n_samples,]

Index of the cluster each sample belongs to.

static **score**(*X*, *y=None*)
Opposite of the value of X on the K-means objective.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

New data.

Returnsscore : float

Opposite of the value of X on the K-means objective.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static **transform**(*X*, *y=None*)
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.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

New data to transform.

ReturnsX_new : array, shape [n_samples, k]

X transformed in the new space.

Examples using `sklearn.cluster.KMeans`

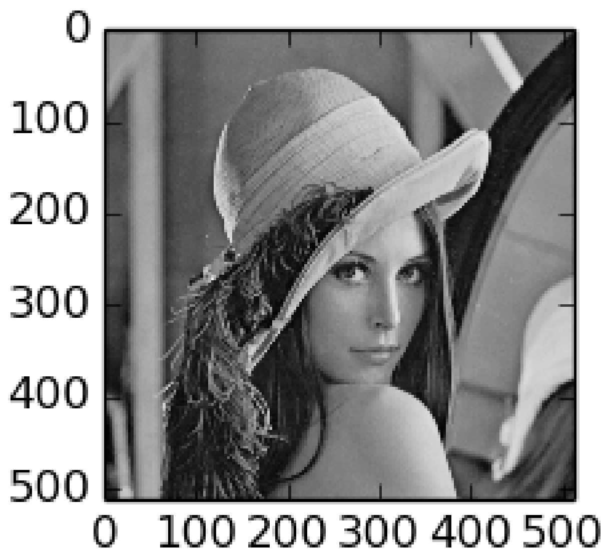


Figure 42.17: *Vector Quantization Example*

`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

Parametersn_clusters : int, optional, default: 8

The number of clusters to form as well as the number of centroids to generate.

max_iter : int, optional

Maximum number of iterations over the complete dataset before stopping independently of any early stopping criterion heuristics.

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.

tol : float, default: 0.0

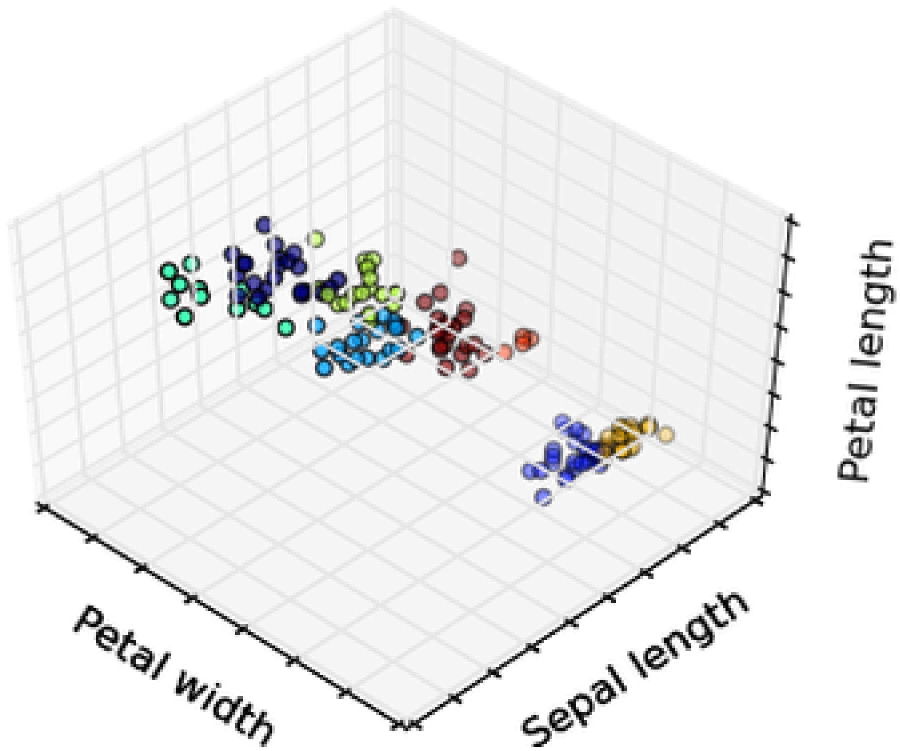


Figure 42.18: *K-means Clustering*

Figure 42.19: *Color Quantization using K-Means*

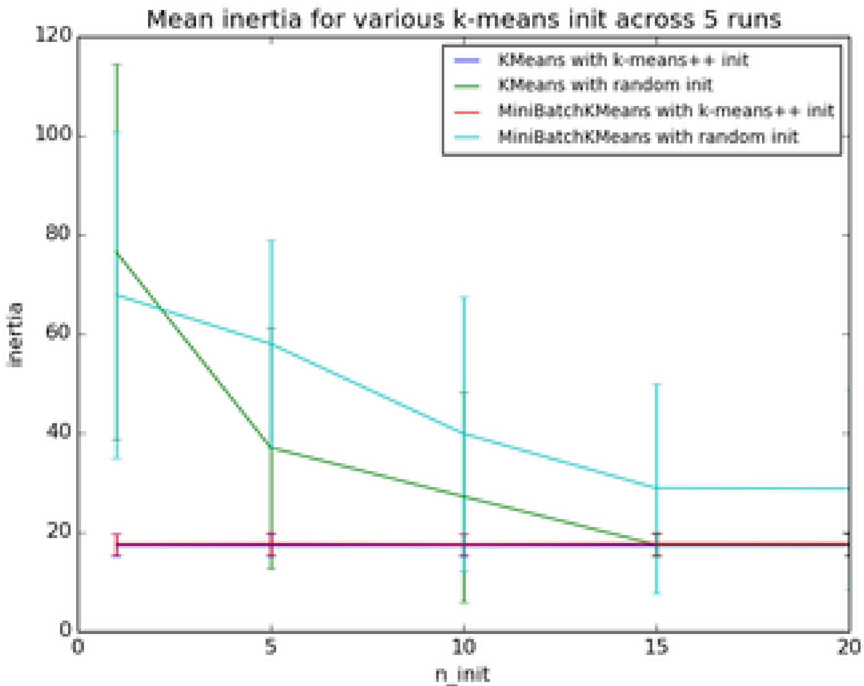


Figure 42.20: *Empirical evaluation of the impact of k-means initialization*

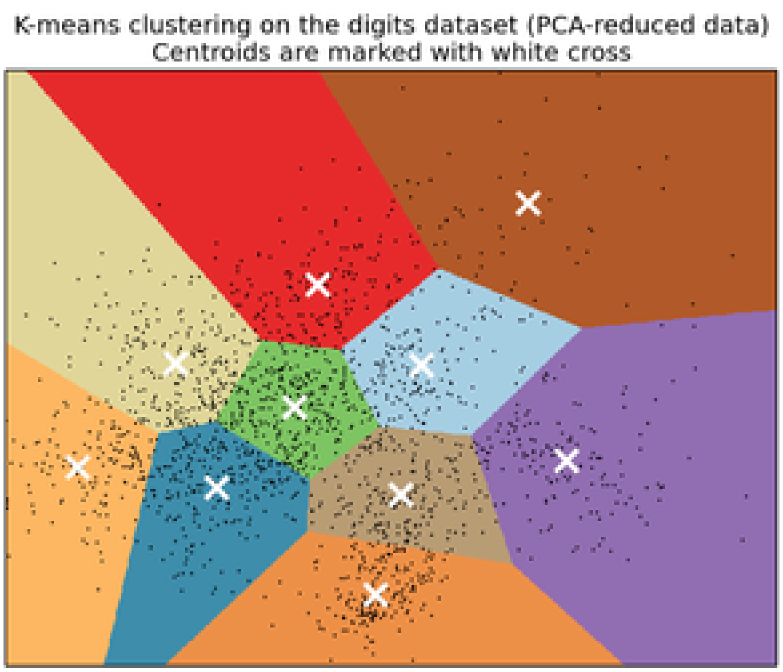


Figure 42.21: *A demo of K-Means clustering on the handwritten digits data*

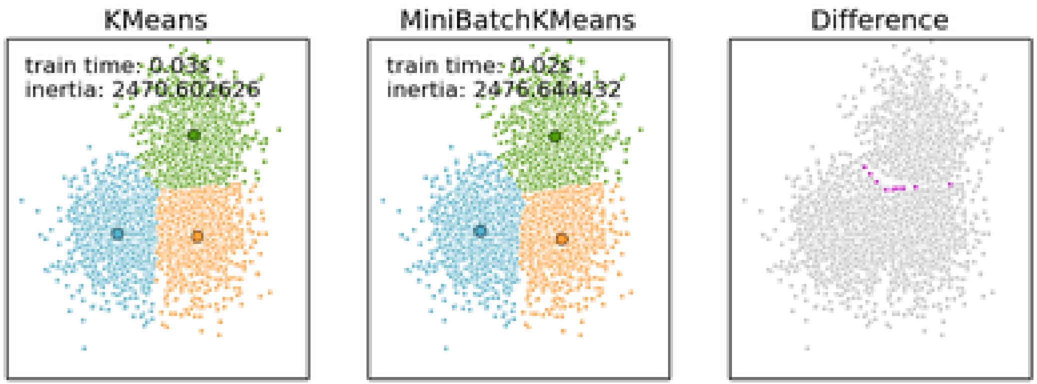


Figure 42.22: *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*

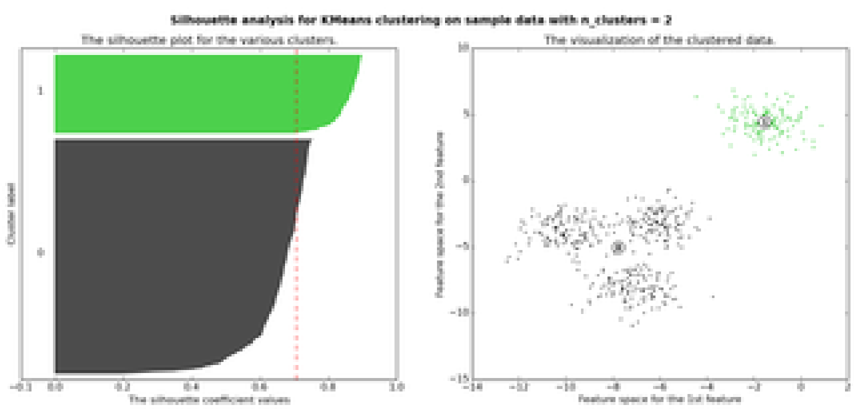


Figure 42.23: *Selecting the number of clusters with silhouette analysis on KMeans clustering*

Figure 42.24: *Clustering text documents using k-means*

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).

batch_size : int, optional, default: 100

Size of the mini batches.

init_size : int, optional, default: 3 * batch_size

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.

init : {'k-means++', 'random' or an ndarray}, 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=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.

compute_labels : boolean, default=True

Compute label assignment and inertia for the complete dataset once the minibatch optimization has converged in fit.

random_state : integer or numpy.RandomState, optional

The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

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.

verbose : boolean, optional

Verbosity mode.

Attributescluster_centers_ : array, [n_clusters, n_features]

Coordinates of cluster centers

labels_ : :

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.

Notes

See http://www.eecs.tufts.edu/~dsculley/papers/fastkmeans.pdf

Methods

`fit`(X[, y])	Compute the centroids on X by chunking it into mini-batches.
`fit_predict`(X[, y])	Compute cluster centers and predict cluster index for each sample.
`fit_transform`(X[, y])	Compute clustering and transform X to cluster-distance space.
`get_params`([deep])	Get parameters for this estimator.
`partial_fit`(X[, y])	Update k means estimate on a single mini-batch X.
`predict`(X)	Predict the closest cluster each sample in X belongs to.
`score`(X[, y])	Opposite of the value of X on the K-means objective.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y])	Transform X to a cluster-distance space.

static `__init__`(*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*)

static `fit`(*X, y=None*)

Compute the centroids on X by chunking it into mini-batches.

ParametersX : array-like, shape = [n_samples, n_features]

Coordinates of the data points to cluster

`fit_predict`(*X, y=None*)

Compute cluster centers and predict cluster index for each sample.

Convenience method; equivalent to calling fit(X) followed by predict(X).

`fit_transform`(*X, y=None*)

Compute clustering and transform X to cluster-distance space.

Equivalent to fit(X).transform(X), but more efficiently implemented.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **partial_fit**(*X*, *y=None*)
Update k means estimate on a single mini-batch X.

ParametersX : array-like, shape = [n_samples, n_features]

Coordinates of the data points to cluster.

static **predict**(*X*)
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.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

New data to predict.

Returnslabels : array, shape [n_samples,]

Index of the cluster each sample belongs to.

score(*X*, *y=None*)
Opposite of the value of X on the K-means objective.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

New data.

Returnsscore : float

Opposite of the value of X on the K-means objective.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*, *y=None*)
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.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

New data to transform.

ReturnsX_new : array, shape [n_samples, k]

X transformed in the new space.

Examples using `sklearn.cluster.MiniBatchKMeans`

Figure 42.25: *Biclustering documents with the Spectral Co-clustering algorithm*

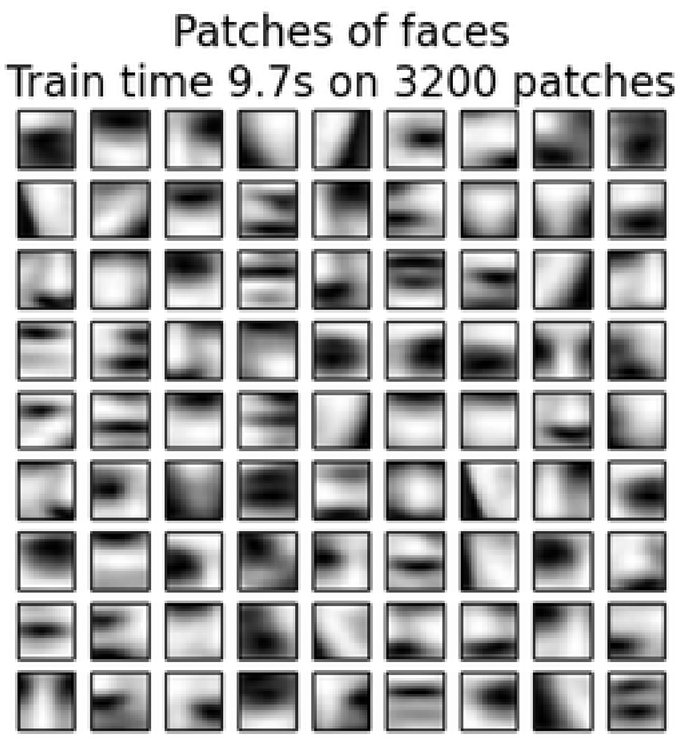


Figure 42.26: *Online learning of a dictionary of parts of faces*

`sklearn.cluster`.MeanShift

class `sklearn.cluster.`**`MeanShift`**(*bandwidth=None*, *seeds=None*, *bin_seeding=False*, *min_bin_freq=1*, *cluster_all=True*)

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.

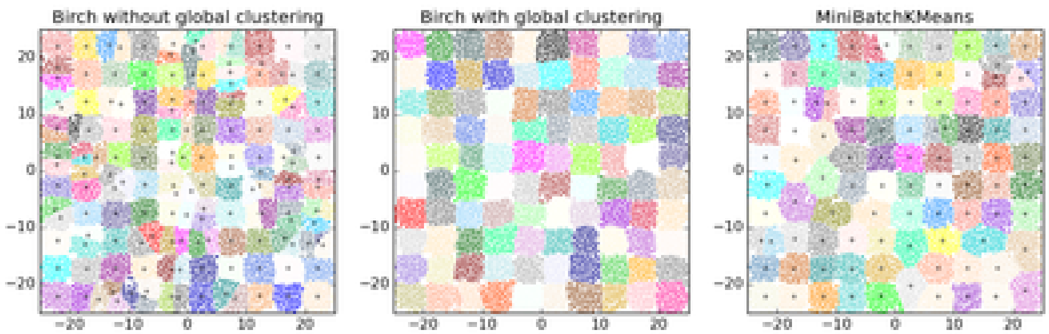


Figure 42.27: *Compare BIRCH and MiniBatchKMeans*

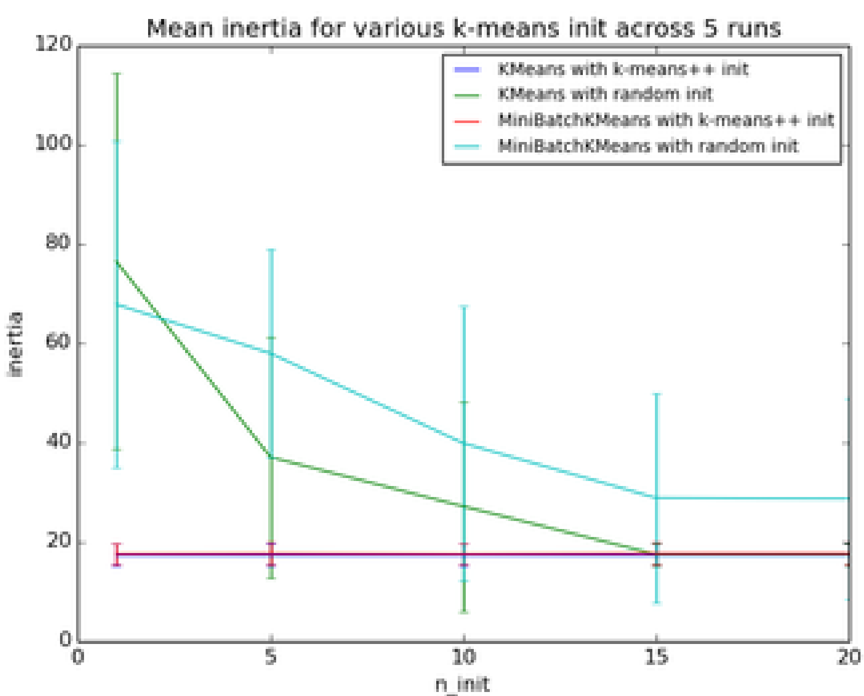


Figure 42.28: *Empirical evaluation of the impact of k-means initialization*

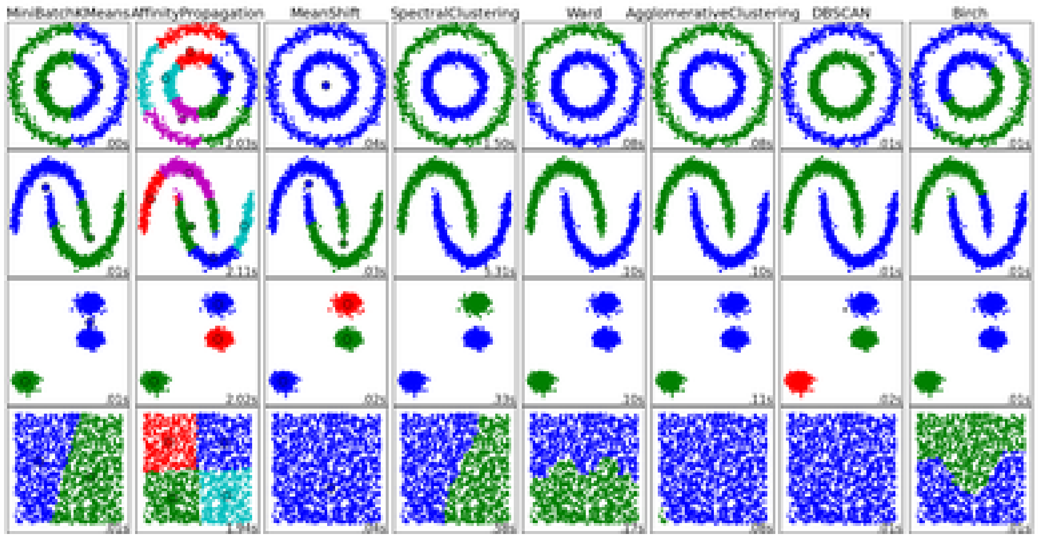


Figure 42.29: *Comparing different clustering algorithms on toy datasets*

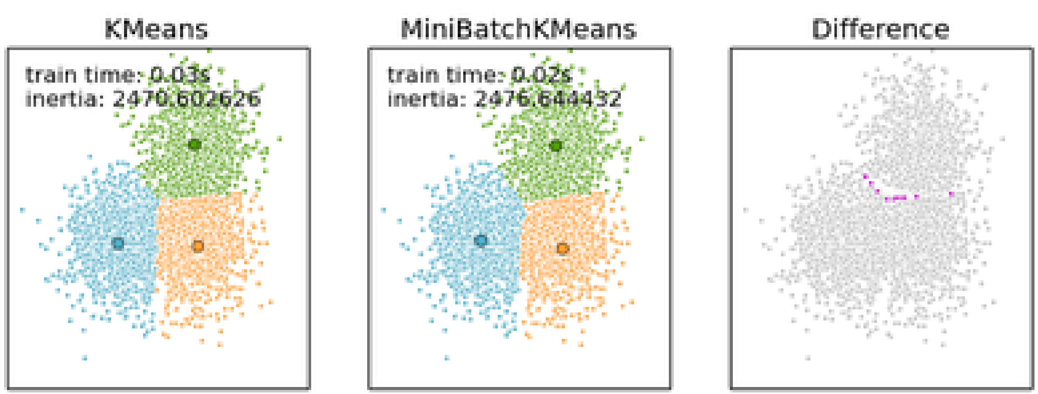


Figure 42.30: *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*

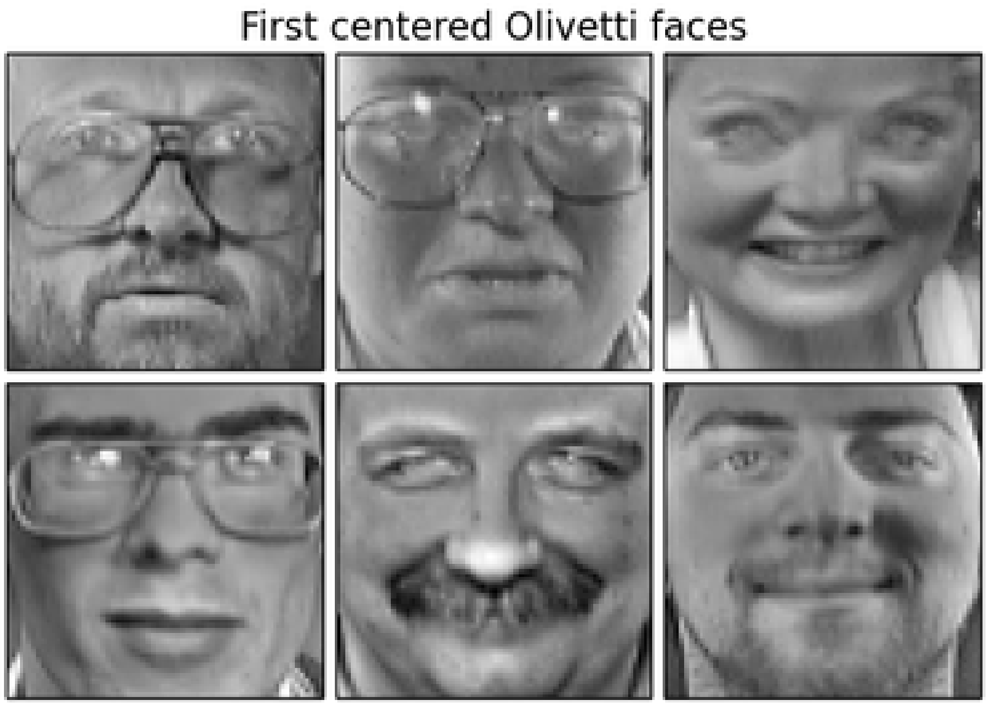


Figure 42.31: *Faces dataset decompositions*

Figure 42.32: *Clustering text documents using k-means*

Seeding is performed using a binning technique for scalability.

Parametersbandwidth : 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.

Attributescluster_centers_ : array, [n_clusters, n_features]

Coordinates of cluster centers.

labels_ : :

Labels of each point.

Notes

Scalability:

Because this implementation uses a flat kernel and a Ball Tree to look up members of each kernel, the complexity will is to 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.

Methods

fit(X[, y])	Perform clustering.
fit_predict(X[, y])	Performs clustering on X and returns cluster labels.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict the closest cluster each sample in X belongs to.
set_params(**params)	Set the parameters of this estimator.

static __init__(*bandwidth=None*, *seeds=None*, *bin_seeding=False*, *min_bin_freq=1*, *cluster_all=True*)

static fit(*X*, *y=None*)

Perform clustering.

ParametersX : array-like, shape=[n_samples, n_features]

Samples to cluster.

fit_predict(*X*, *y=None*)

Performs clustering on X and returns cluster labels.

ParametersX : ndarray, shape (n_samples, n_features)

Input data.

Returnsy : ndarray, shape (n_samples,)

cluster labels

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static predict(*X*)

Predict the closest cluster each sample in X belongs to.

ParametersX : {array-like, sparse matrix}, shape=[n_samples, n_features]

New data to predict.

Returnslabels : array, shape [n_samples,]

Index of the cluster each sample belongs to.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

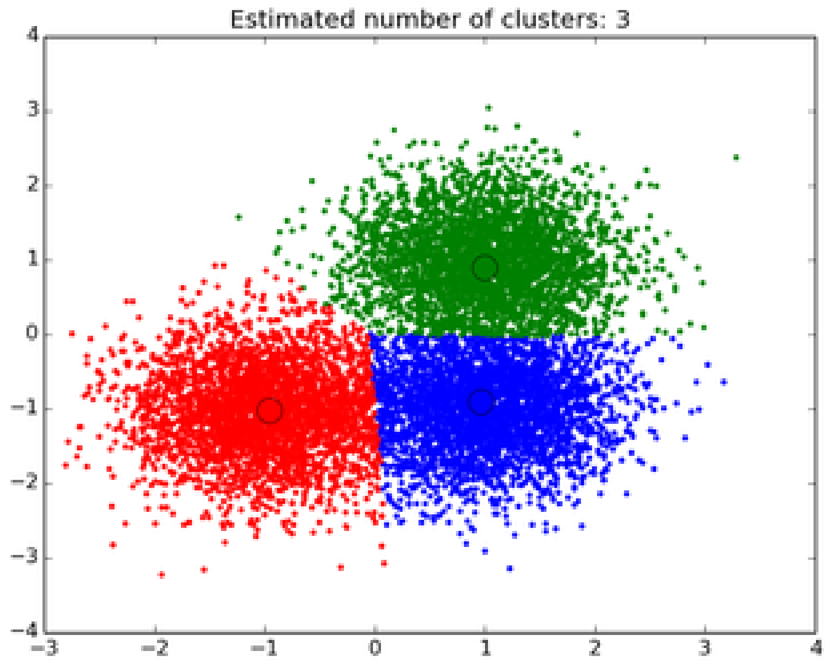


Figure 42.33: *A demo of the mean-shift clustering algorithm*

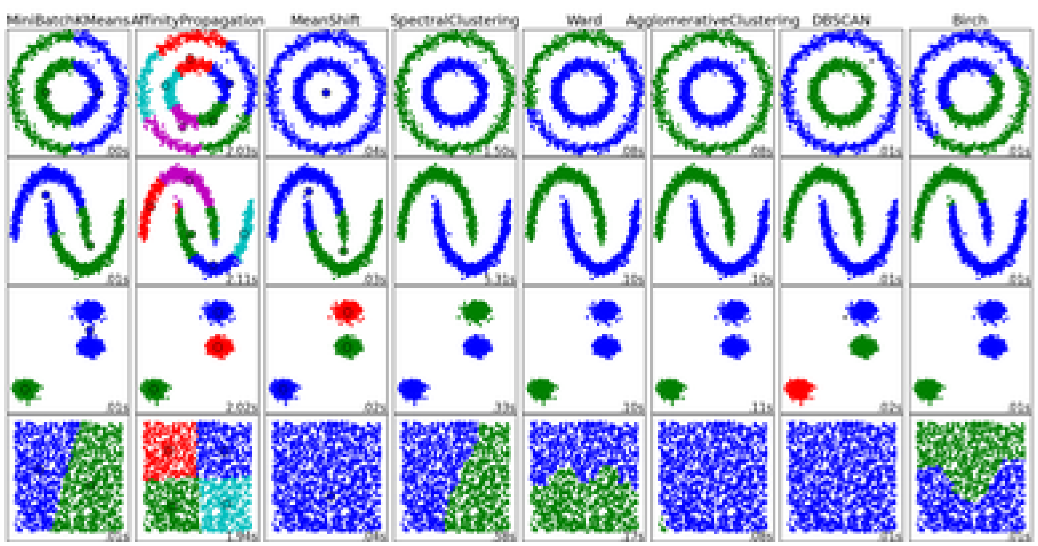


Figure 42.34: *Comparing different clustering algorithms on toy datasets*

Examples using `sklearn.cluster.MeanShift`

`sklearn.cluster`.SpectralClustering

class `sklearn.cluster.`**`SpectralClustering`**(*n_clusters=8, eigen_solver=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*)

Apply clustering to a projection to 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 plan.

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.

Parametersn_clusters : integer, optional

The dimension of the projection subspace.

affinity : string, array-like or callable, default 'rbf'

If a string, this may be one of 'nearest_neighbors', 'precomputed', 'rbf' or one of the kernels supported by *sklearn.metrics.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.

gamma : float

Scaling factor of RBF, polynomial, exponential chi^2 and sigmoid affinity kernel. Ignored for `affinity='nearest_neighbors'`.

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.

n_neighbors : integer

Number of neighbors to use when constructing the affinity matrix using the nearest neighbors method. Ignored for `affinity='rbf'`.

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 seed, 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.

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.

kernel_params : dictionary of string to any, optional

Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.

Attributesaffinity_matrix_ : array-like, shape (n_samples, n_samples)

Affinity matrix used for clustering. Available only if after calling `fit`.

labels_ : :

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(- X ** 2 / (2. * delta ** 2))
```

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 http://www1.icsi.berkeley.edu/~stellayu/publication/doc/2003kwayICCV.pdf

Methods

`fit`(X[, y])	Creates an affinity matrix for X using the selected affinity, then applies spectral clustering to this affinity
`fit_predict`(X[, y])	Performs clustering on X and returns cluster labels.

Continued on nex

Table 42.17 – continued from previous page

get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

static __init__ (*n_clusters=8, eigen_solver=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*)

static fit (*X, y=None*)

Creates an affinity matrix for X using the selected affinity, then applies spectral clustering to this affinity matrix.

ParametersX : array-like or sparse matrix, shape (n_samples, n_features)

OR, if affinity==‘precomputed‘, a precomputed affinity matrix of shape (n_samples, n_samples)

fit_predict (*X, y=None*)

Performs clustering on X and returns cluster labels.

ParametersX : ndarray, shape (n_samples, n_features)

Input data.

Returnsy : ndarray, shape (n_samples,)

cluster labels

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using sklearn.cluster.SpectralClustering

sklearn.cluster.Ward

class sklearn.cluster.**Ward** (*n_clusters=2, memory=Memory(cachedir=None), connectivity=None, n_components=None, compute_full_tree='auto', pooling_func=<function mean at 0x2ad3c0a06f50>*)

Ward hierarchical clustering: constructs a tree and cuts it.

Recursively merges the pair of clusters that minimally increases within-cluster variance.

Parametersn_clusters : int or ndarray

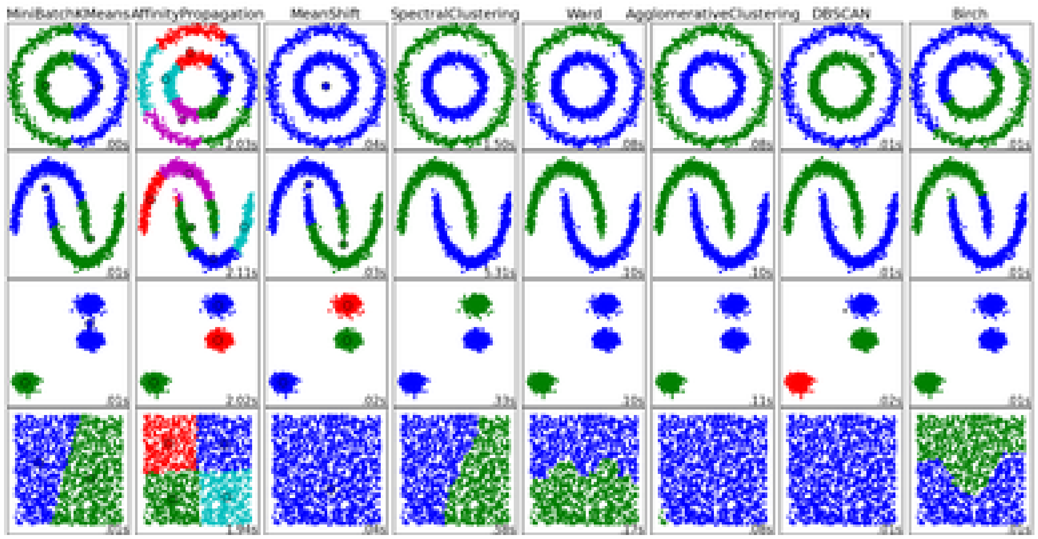


Figure 42.35: *Comparing different clustering algorithms on toy datasets*

The number of clusters to find.

connectivity : sparse matrix (optional)

Connectivity matrix. Defines for each sample the neighboring samples following a given structure of the data. Default is None, i.e, the hierarchical clustering algorithm is unstructured.

memory : Instance of joblib.Memory or string (optional)

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.

n_components : int (optional)

The number of connected components in the graph defined by the connectivity matrix. If not set, it is estimated.

compute_full_tree : bool or 'auto' (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. 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.

Attributeslabels_ : array [n_features]

cluster labels for each feature

n_leaves_ : int

Number of leaves in the hierarchical tree.

n_components_ : int

The estimated number of connected components in the graph.

children_ : array-like, shape (n_nodes-1, 2)

The children of each non-leaf node. Values less than *n_samples* refer to leaves of the tree. A greater value *i* indicates a node with children *children_[i - n_samples]*.

See also:

AgglomerativeClusteringagglomerative hierarchical clustering

Methods

fit(X[, y])	Fit the hierarchical clustering on the data
fit_predict(X[, y])	Performs clustering on X and returns cluster labels.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

static __init__(*n_clusters=2*, *memory=Memory(cachedir=None)*, *connectivity=None*, *n_components=None*, *compute_full_tree='auto'*, *pooling_func=<function mean at 0x2ad3c0a06f50>*)

fit(*X*, *y=None*)

Fit the hierarchical clustering on the data

ParametersX : array-like, shape = [n_samples, n_features]

The samples a.k.a. observations.

Returnsself :

fit_predict(*X*, *y=None*)

Performs clustering on X and returns cluster labels.

ParametersX : ndarray, shape (n_samples, n_features)

Input data.

Returnsy : ndarray, shape (n_samples,)

cluster labels

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

42.2.2 Functions

`cluster.estimate_bandwidth`(X[, quantile, ...])	Estimate the bandwidth to use with the mean-shift algorithm.
`cluster.k_means`(X, n_clusters[, init, ...])	K-means clustering algorithm.
`cluster.ward_tree`(X[, connectivity, ...])	Ward clustering based on a Feature matrix.
`cluster.affinity_propagation`(S[, ...])	Perform Affinity Propagation Clustering of data
`cluster.dbscan`(X[, eps, min_samples, ...])	Perform DBSCAN clustering from vector array or distance matrix.
`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 to the normalized laplacian.

`sklearn.cluster`.estimate_bandwidth

`sklearn.cluster.`**`estimate_bandwidth`**(*X*, *quantile=0.3*, *n_samples=None*, *random_state=0*)

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.

ParametersX : array-like, 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 or RandomState

Pseudo-random number generator state used for random sampling.

Returnsbandwidth : float

The bandwidth parameter.

Examples using `sklearn.cluster.estimate_bandwidth`

`sklearn.cluster`.k_means

`sklearn.cluster.`**`k_means`**(*X*, *n_clusters*, *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=1*, *return_n_iter=False*)

K-means clustering algorithm.

ParametersX : array-like or sparse matrix, shape (n_samples, n_features)

The observations to cluster.

n_clusters : int

The number of clusters to form as well as the number of centroids to generate.

max_iter : int, optional, default 300

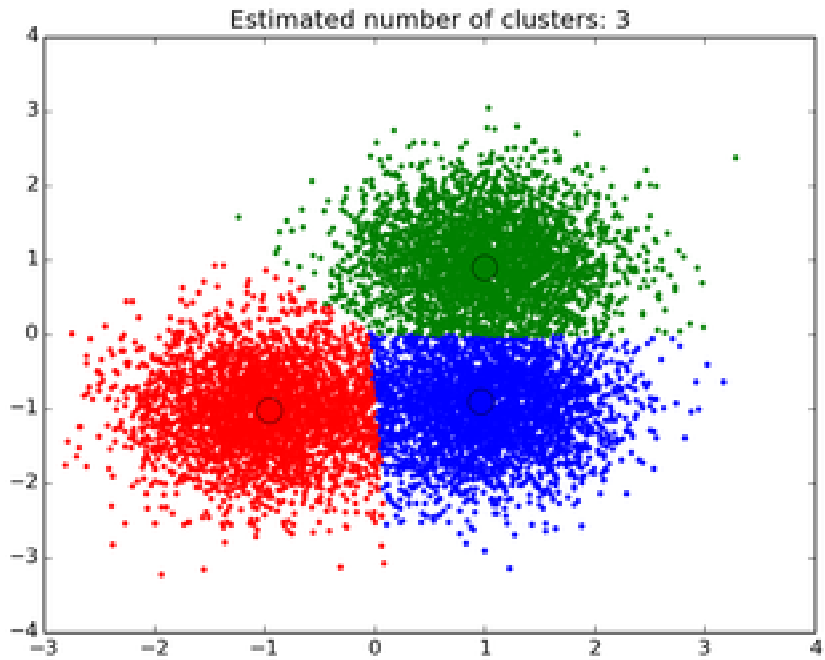


Figure 42.36: *A demo of the mean-shift clustering algorithm*

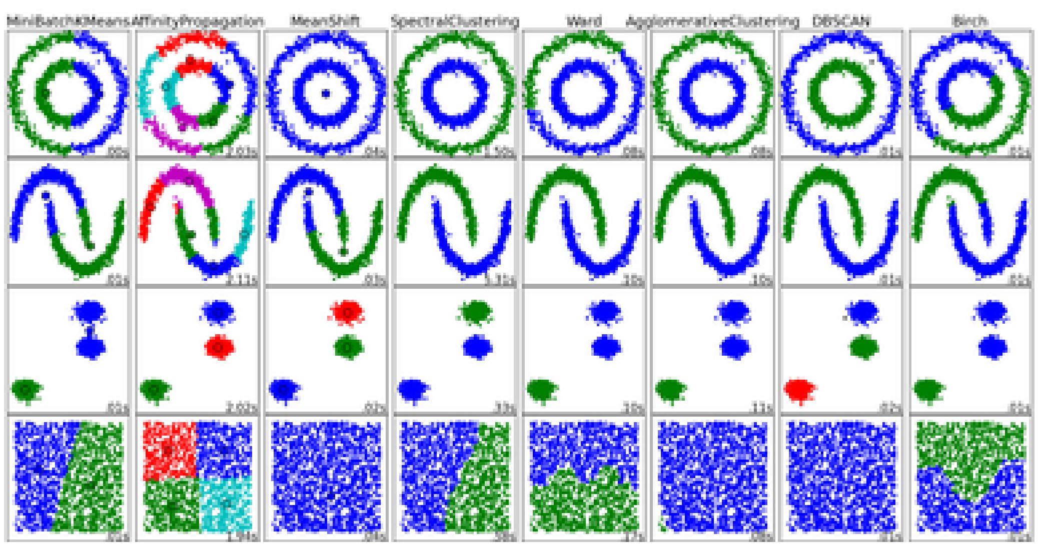


Figure 42.37: *Comparing different clustering algorithms on toy datasets*

Maximum number of iterations of the k-means algorithm to run.

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.

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': generate k centroids from a Gaussian with mean and variance estimated from the data.

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

tol : float, optional

The relative increment in the results before declaring convergence.

verbose : boolean, optional

Verbosity mode.

random_state : integer or numpy.RandomState, optional

The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

copy_x : boolean, optional

When pre-computing distances it is more numerically accurate to center the data first. If copy_x is True, then the original data is not modified. 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.

n_jobs : int

The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

return_n_iter : bool, optional

Whether or not to return the number of iterations.

Returnscentroid : 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`.ward_tree

`sklearn.cluster.`**`ward_tree`**(*X*, *connectivity=None*, *n_components=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.

ParametersX : 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_components : int (optional)

Number of connected components. If None the number of connected components is estimated from the connectivity matrix. NOTE: This parameter is now directly determined directly from the connectivity matrix and will be removed in 0.18

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.

Returnschildren : 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_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.

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

ParametersS : 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.

Returnscluster_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

See examples/cluster/plot_affinity_propagation.py for an example.

References

Brendan J. Frey and Delbert Dueck, "Clustering by Passing Messages Between Data Points", Science Feb. 2007

Examples using `sklearn.cluster.affinity_propagation`

`sklearn.cluster`.dbscan

`sklearn.cluster.`**`dbscan`**(*X*, *eps=0.5*, *min_samples=5*, *metric='minkowski'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *sample_weight=None*, *random_state=None*)

Perform DBSCAN clustering from vector array or distance matrix.

ParametersX : 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 them to be considered as in the same neighborhood.

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 metrics.pairwise.pairwise_distances for its metric parameter. If metric is "precomputed", X is assumed to be a distance matrix and must be square.

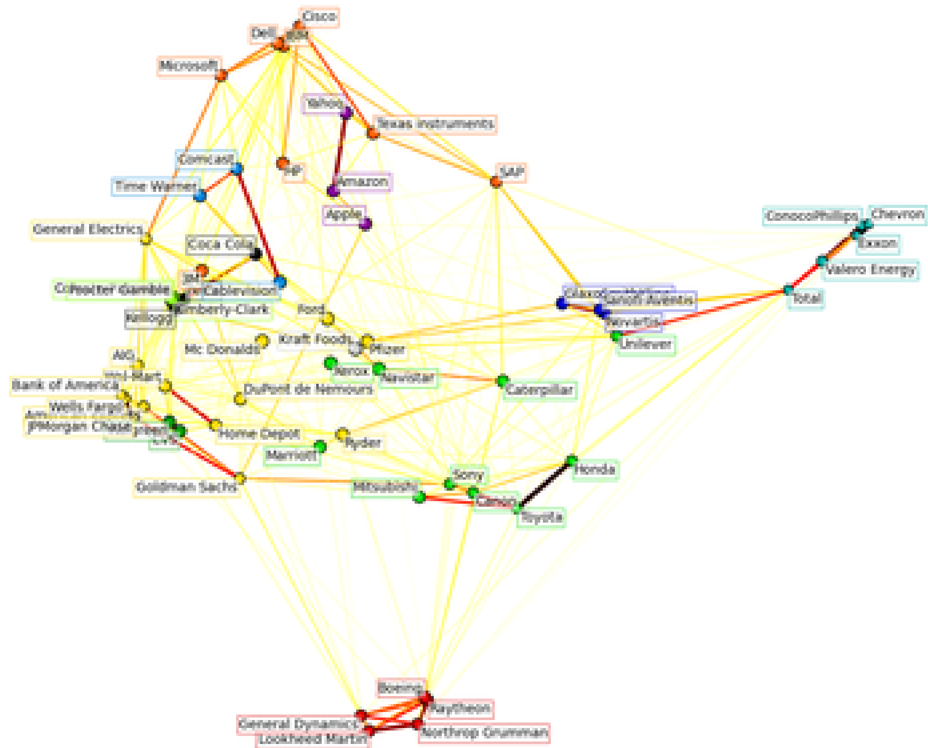


Figure 42.38: *Visualizing the stock market structure*

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.

random_state: numpy.RandomState, optional :

Deprecated and ignored as of version 0.16, will be removed in version 0.18. DBSCAN does not use random initialization.

Returnscore_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.

Notes

See examples/cluster/plot_dbscan.py for an example.

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).

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

`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*, *max_iterations=None*)

Perform mean shift clustering of data using a flat kernel.

Parameters **X** : array-like, 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, 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.

Returnscluster_centers : array, shape=[n_clusters, n_features]

Coordinates of cluster centers.

labels : array, shape=[n_samples]

Cluster labels for each point.

Notes

See examples/cluster/plot_meanshift.py for an example.

`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 to 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 plan.

If affinity is the adjacency matrix of a graph, this method can be used to find normalized graph cuts.

Parametersaffinity : 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 seed, 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.

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.

Returnslabels : 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 http://www1.icsi.berkeley.edu/~stellayu/publication/doc/2003kwayICCV.pdf

Examples using `sklearn.cluster.spectral_clustering`

42.3 `sklearn.cluster.bicluster`: Biclustering

Spectral biclustering algorithms.

Authors : Kemal Eren License: BSD 3 clause

User guide: See the *Biclustering* section for further details.

42.3.1 Classes

`SpectralBiclustering`([n_clusters, method, ...])	Spectral biclustering (Kluger, 2003).
`SpectralCoclustering`([n_clusters, ...])	Spectral Co-Clustering algorithm (Dhillon, 2001).

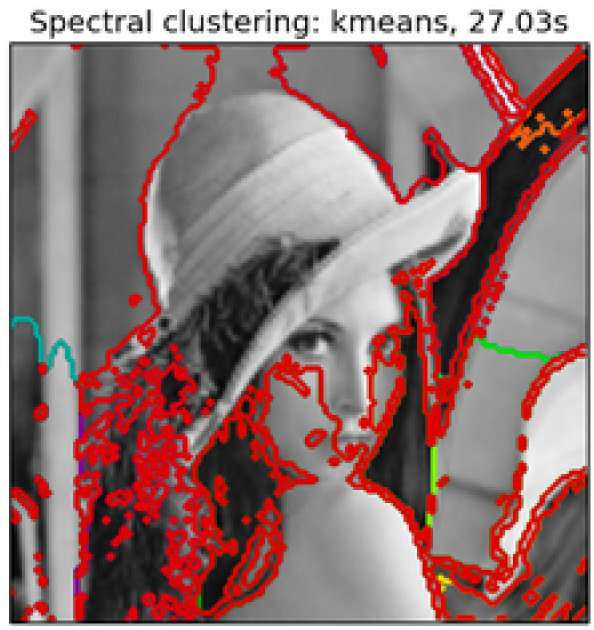


Figure 42.39: *Segmenting the picture of Lena in regions*

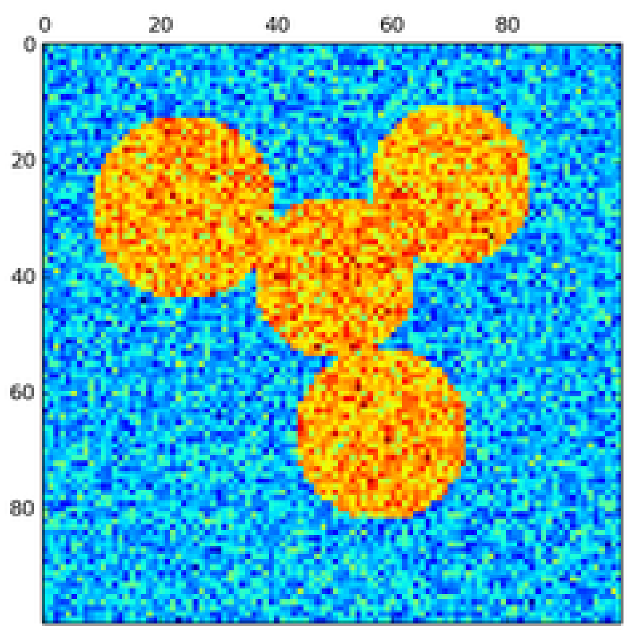


Figure 42.40: *Spectral clustering for image segmentation*

`sklearn.cluster.bicluster`.SpectralBiclustering

class `sklearn.cluster.bicluster.`**`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=1*, *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.

Parameters

n_clusters : integer or tuple (n_row_clusters, n_column_clusters)

The number of row and column clusters in the checkerboard structure.

method : string, optional, 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'. CAUTION: if *method='log'*, the data must not be sparse.

n_components : integer, optional, default: 6

Number of singular vectors to check.

n_best : integer, optional, default: 3

Number of best singular vectors to which to project the data for clustering.

svd_method : string, optional, default: 'randomized'

Selects the algorithm for finding singular vectors. May be 'randomized' or 'arpack'. If 'randomized', uses *sklearn.utils.extmath.randomized_svd*, which may be faster for large matrices. If 'arpack', uses *sklearn.utils.arpack.svds*, which is more accurate, but possibly slower in some cases.

n_svd_vecs : int, optional, 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, optional, default: False

Whether to use mini-batch k-means, which is faster but may get different results.

init : {'k-means++', 'random' or an ndarray}

Method for initialization of k-means algorithm; defaults to 'k-means++'.

n_init : int, optional, 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, optional, default: 1

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.

If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

random_state : int seed, RandomState instance, or None (default)

A pseudo random number generator used by the K-Means initialization.

Attributesrows_ : array-like, 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, shape (n_column_clusters, n_columns)

Results of the clustering, like *rows*.

row_labels_ : array-like, shape (n_rows,)

Row partition labels.

column_labels_ : array-like, shape (n_cols,)

Column partition labels.

References

•Kluger, Yuval, et. al., 2003. Spectral biclustering of microarray data: coclustering genes and conditions.

Methods

`fit`(X)	Creates a biclustering for X.
`get_indices`(i)	Row and column indices of the i'th bicluster.
`get_params`([deep])	Get parameters for this estimator.
`get_shape`(i)	Shape of the i'th bicluster.
`get_submatrix`(i, data)	Returns the submatrix corresponding to bicluster *i*.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*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=1, random_state=None*)

`fit`(*X*)

Creates a biclustering for X.

ParametersX : array-like, shape (n_samples, n_features)

`get_indices`(*i*)

Row and column indices of the i'th bicluster.

Only works if `rows_` and `columns_` attributes exist.

Returnsrow_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(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_shape(*i*)
Shape of the i'th bicluster.

Returnsshape : (int, int)

Number of rows and columns (resp.) in the bicluster.

get_submatrix(*i*, *data*)
Returns the submatrix corresponding to bicluster *i*.

Works with sparse matrices. Only works if `rows_` and `columns_` attributes exist.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

sklearn.cluster.bicluster.SpectralCoclustering

class sklearn.cluster.bicluster.**SpectralCoclustering**(*n_clusters=3*, *svd_method='randomized'*, *n_svd_vecs=None*, *mini_batch=False*, *init='k-means++'*, *n_init=10*, *n_jobs=1*, *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.

Parametersn_clusters : integer, optional, default: 3

The number of biclusters to find.

svd_method : string, optional, 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 `sklearn.utils.arpack.svds`, which is more accurate, but possibly slower in some cases.

n_svd_vecs : int, optional, 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, optional, default: False

Whether to use mini-batch k-means, which is faster but may get different results.

init : {'k-means++', 'random' or an ndarray}

Method for initialization of k-means algorithm; defaults to 'k-means++'.

n_init : int, optional, 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, optional, default: 1

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.

If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

random_state : int seed, RandomState instance, or None (default)

A pseudo random number generator used by the K-Means initialization.

Attributesrows_ : array-like, 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, shape (n_column_clusters, n_columns)

Results of the clustering, like *rows*.

row_labels_ : array-like, shape (n_rows,)

The bicluster label of each row.

column_labels_ : array-like, shape (n_cols,)

The bicluster label of each column.

References

•Dhillon, Inderjit S, 2001. Co-clustering documents and words using bipartite spectral graph partitioning.

Methods

`fit`(X)	Creates a biclustering for X.
`get_indices`(i)	Row and column indices of the i'th bicluster.
`get_params`([deep])	Get parameters for this estimator.
`get_shape`(i)	Shape of the i'th bicluster.
`get_submatrix`(i, data)	Returns the submatrix corresponding to bicluster *i*.

Continued on next page

Table 42.22 – continued from previous page

set_params(**params)	Set the parameters of this estimator.

static __init__(*n_clusters=3, svd_method='randomized', n_svd_vecs=None, mini_batch=False, init='k-means++', n_init=10, n_jobs=1, random_state=None*)

fit(*X*)
Creates a biclustering for X.

Parameters X : array-like, shape (n_samples, n_features)

get_indices(*i*)
Row and column indices of the i'th bicluster.

Only works if `rows_` and `columns_` attributes exist.

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(*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.

get_shape(*i*)
Shape of the i'th bicluster.

Returns shape : (int, int)

Number of rows and columns (resp.) in the bicluster.

get_submatrix(*i, data*)
Returns the submatrix corresponding to bicluster *i*.

Works with sparse matrices. Only works if `rows_` and `columns_` attributes exist.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns self :

42.4 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.GraphLasso([alpha, mode, tol, ...])	Sparse inverse covariance estimation with an l1-penalized estimator.
covariance.GraphLassoCV([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

42.4.1 sklearn.covariance.EmpiricalCovariance

class sklearn.covariance.**EmpiricalCovariance**(*store_precision=True*, *assume_centered=False*)

Maximum likelihood covariance estimator

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 **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)

Methods

error_norm(comp_cov[, norm, scaling, squared])	Computes the Mean Squared Error between two covariance estimators.
fit(X[, y])	Fits the Maximum Likelihood Estimator covariance model according to the giv
get_params([deep])	Get parameters for this estimator.
get_precision()	Getter for the precision matrix.
mahalanobis(observations)	Computes the squared Mahalanobis distances of given observations.
score(X_test[, y])	Computes the log-likelihood of a Gaussian data set with *self.covariance_* as an
set_params(**params)	Set the parameters of this estimator.

static __init__(*store_precision=True*, *assume_centered=False*)

static error_norm(*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, 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.

ReturnsThe Mean Squared Error (in the sense of the Frobenius norm) between :

'self' and 'comp_cov' covariance estimators. :

static fit (*X*, *y=None*)

Fits the Maximum Likelihood Estimator covariance model according to the given training data and parameters.

ParametersX : 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 : not used, present for API consistence purpose.

Returnsself : object

Returns self.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static get_precision ()

Getter for the precision matrix.

Returnsprecision_ : array-like,

The precision matrix associated to the current covariance object.

static mahalanobis (*observations*)

Computes the squared Mahalanobis distances of given observations.

Parametersobservations : array-like, shape = [n_observations, 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.

Returnsmahalanobis_distance : array, shape = [n_observations,]

Squared Mahalanobis distances of the observations.

static score (*X_test*, *y=None*)

Computes the log-likelihood of a Gaussian data set with *self.covariance_* as an estimator of its covariance matrix.

ParametersX_test : array-like, 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.

Returnsres : float

The likelihood of the data set with *self.covariance_* as an estimator of its covariance matrix.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.covariance.EmpiricalCovariance`

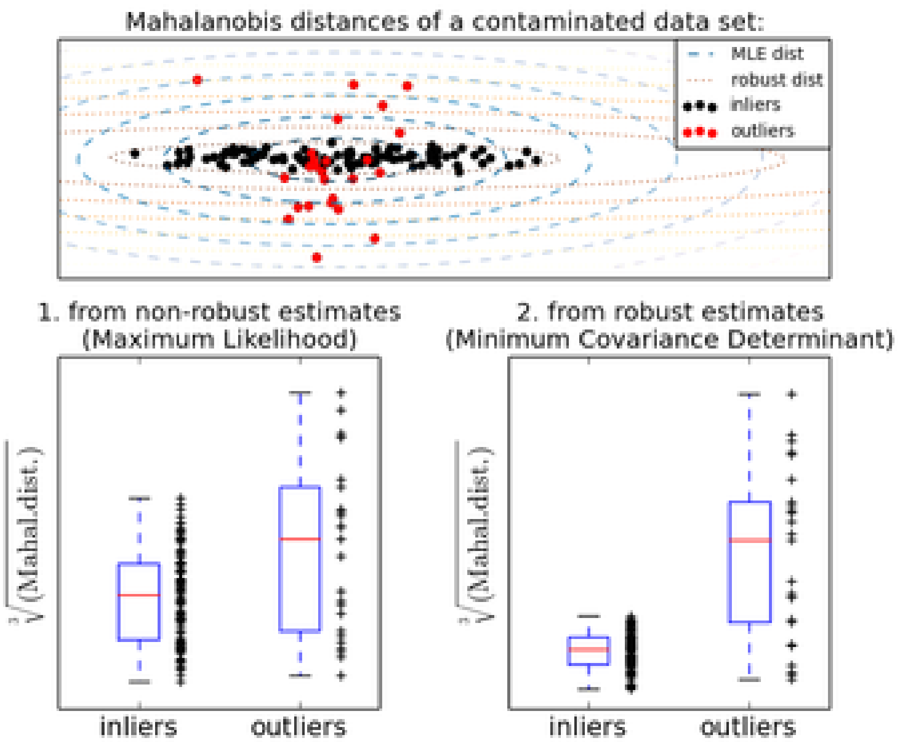


Figure 42.41: *Robust covariance estimation and Mahalanobis distances relevance*

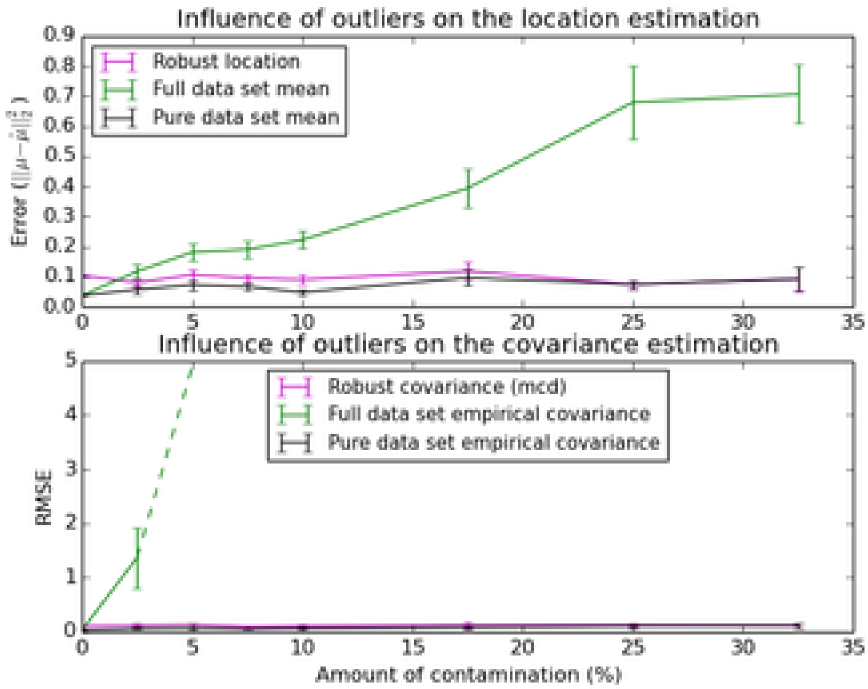


Figure 42.42: *Robust vs Empirical covariance estimate*

42.4.2 `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.

Parametersstore_precision : bool

Specify if the estimated precision is stored.

assume_centered : Boolean

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, 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*.

contamination : float, 0. < contamination < 0.5

The amount of contamination of the data set, i.e. the proportion of outliers in the data set.

Attributes'contamination' : float, 0. < contamination < 0.5

The amount of contamination of the data set, i.e. the proportion of outliers in the data set.

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.

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

Methods

`correct_covariance`(data)	Apply a correction to raw Minimum Covariance Determinant estimates.
`decision_function`(X[, raw_values])	Compute the decision function of the given observations.
`error_norm`(comp_cov[, norm, scaling, squared])	Computes the Mean Squared Error between two covariance estimators.
`fit`(X[, y])	
`get_params`([deep])	Get parameters for this estimator.
`get_precision`()	Getter for the precision matrix.
`mahalanobis`(observations)	Computes the squared Mahalanobis distances of given observations.
`predict`(X)	Outlyingness of observations in X according to the fitted model.
`reweight_covariance`(data)	Re-weight raw Minimum Covariance Determinant estimates.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.

static __init__(*store_precision=True*, *assume_centered=False*, *support_fraction=None*, *contamination=0.1*, *random_state=None*)

correct_covariance(*data*)

Apply a correction to raw Minimum Covariance Determinant estimates.

Correction using the empirical correction factor suggested by Rousseeuw and Van Driessen in [Rouseeuw1984].

Parametersdata : 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.

Returnscovariance_corrected : array-like, shape (n_features, n_features)

Corrected robust covariance estimate.

decision_function(*X*, *raw_values=False*)

Compute the decision function of the given observations.

ParametersX : array-like, shape (n_samples, n_features)

raw_values : bool

Whether or not to consider raw Mahalanobis distances as the decision function. Must be False (default) for compatibility with the others outlier detection tools.

Returnsdecision : array-like, shape (n_samples,)

The values of the decision function for each observations. It is equal to the Mahalanobis distances if *raw_values* is True. By default (`raw_values=True`), it is equal to the cubic root of the shifted Mahalanobis distances. In that case, the threshold for being an outlier is 0, which ensures a compatibility with other outlier detection tools such as the One-Class SVM.

error_norm(*comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameterscomp_cov : array-like, 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.

ReturnsThe Mean Squared Error (in the sense of the Frobenius norm) between :

'self' and 'comp_cov' covariance estimators. :

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_precision()

Getter for the precision matrix.

Returnsprecision_ : array-like,

The precision matrix associated to the current covariance object.

mahalanobis (*observations*)

Computes the squared Mahalanobis distances of given observations.

Parametersobservations : array-like, shape = [n_observations, 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.

Returnsmahalanobis_distance : array, shape = [n_observations,]

Squared Mahalanobis distances of the observations.

predict (*X*)

Outlyingness of observations in X according to the fitted model.

ParametersX : array-like, shape = (n_samples, n_features)

Returnsis_outliers : array, shape = (n_samples,), dtype = bool

For each observations, tells whether or not it should be considered as an outlier according to the fitted model.

threshold : float,

The values of the less outlying point's decision function.

reweight_covariance (*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). [Rouseeuw1984]

Parametersdata : 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.

Returnslocation_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.

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.covariance.EllipticEnvelope`

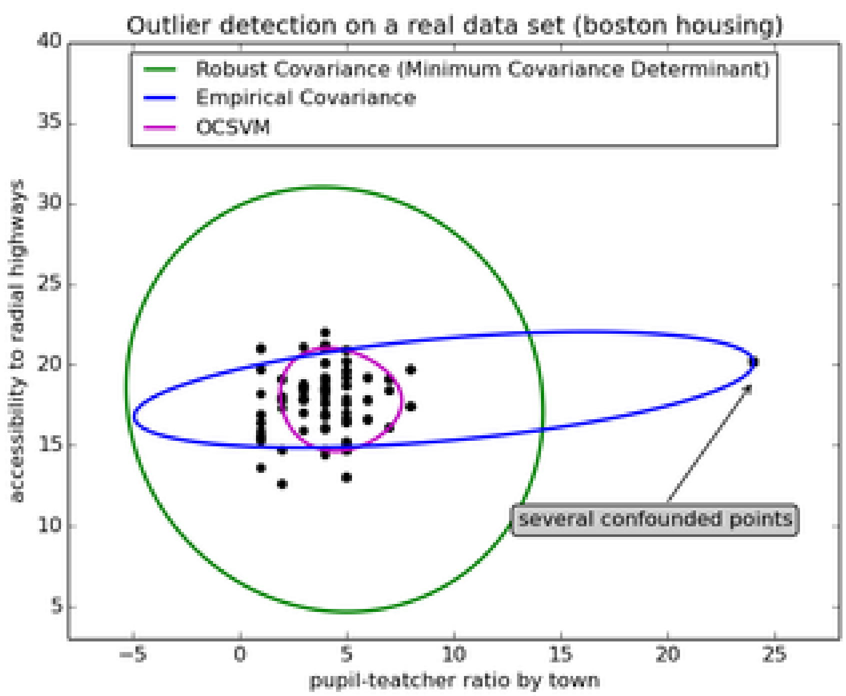


Figure 42.43: *Outlier detection on a real data set*

42.4.3 `sklearn.covariance`.GraphLasso

class `sklearn.covariance.`**`GraphLasso`**(*alpha=0.01*, *mode='cd'*, *tol=0.0001*, *max_iter=100*, *verbose=False*, *assume_centered=False*)

Sparse inverse covariance estimation with an l1-penalized estimator.

Parametersalpha : positive float, default 0.01

The regularization parameter: the higher alpha, the more regularization, the sparser the inverse covariance.

mode : {'cd', 'lars'}, default 'cd'

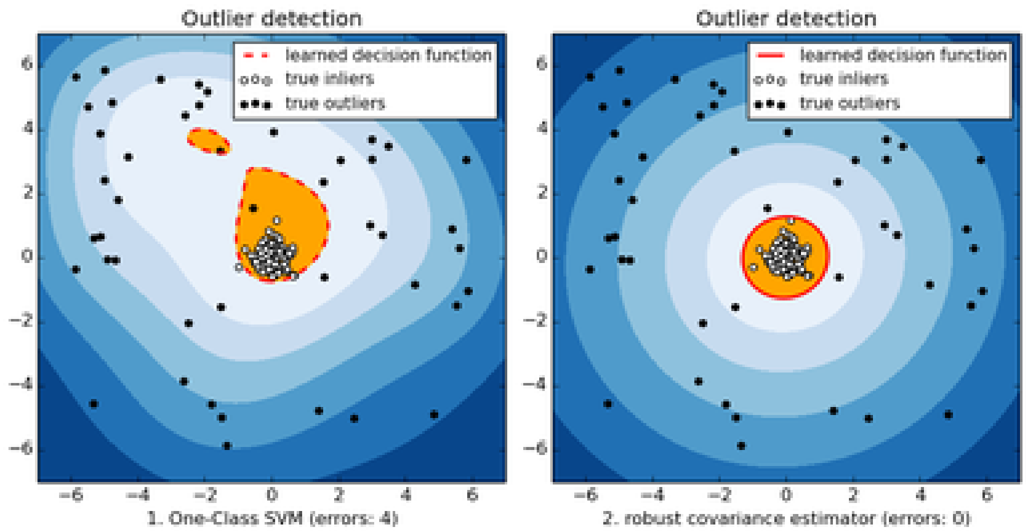


Figure 42.44: *Outlier detection with several methods.*

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.

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.

Attributescovariance_ : 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:

graph_lasso, GraphLassoCV

Methods

error_norm(comp_cov[, norm, scaling, squared])	Computes the Mean Squared Error between two covariance estimators.
fit(X[, y])	
get_params([deep])	Get parameters for this estimator.
get_precision()	Getter for the precision matrix.
mahalanobis(observations)	Computes the squared Mahalanobis distances of given observations.
score(X_test[, y])	Computes the log-likelihood of a Gaussian data set with *self.covariance_* as an
set_params(**params)	Set the parameters of this estimator.

static __init__(*alpha=0.01*, *mode='cd'*, *tol=0.0001*, *max_iter=100*, *verbose=False*, *assume_centered=False*)

error_norm(*comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameterscomp_cov : array-like, 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.

ReturnsThe Mean Squared Error (in the sense of the Frobenius norm) between :

'self' and 'comp_cov' covariance estimators. :

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_precision()

Getter for the precision matrix.

Returnsprecision_ : array-like,

The precision matrix associated to the current covariance object.

mahalanobis (*observations*)

Computes the squared Mahalanobis distances of given observations.

Parametersobservations : array-like, shape = [n_observations, 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.

Returnsmahalanobis_distance : array, shape = [n_observations,]

Squared Mahalanobis distances of the observations.

score (*X_test, y=None*)

Computes the log-likelihood of a Gaussian data set with *self.covariance_* as an estimator of its covariance matrix.

ParametersX_test : array-like, 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.

Returnsres : float

The likelihood of the data set with *self.covariance_* as an estimator of its covariance matrix.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

42.4.4 sklearn.covariance.GraphLassoCV

class sklearn.covariance.**GraphLassoCV** (*alphas=4, n_refinements=4, cv=None, tol=0.0001, max_iter=100, mode='cd', n_jobs=1, verbose=False, assume_centered=False*)

Sparse inverse covariance w/ cross-validated choice of the l1 penalty

Parametersalphas : 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 : cross-validation generator, optional

see sklearn.cross_validation module. If None is passed, defaults to a 3-fold strategy

tol: positive float, optional :

The tolerance to declare convergence: if the dual gap goes below this value, iterations are stopped.

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, optional :

number of jobs to run in parallel (default 1).

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.

Attributescovariance_ : 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:

`graph_lasso`, `GraphLasso`

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.

Methods

`error_norm`(comp_cov[, norm, scaling, squared])	Computes the Mean Squared Error between two covariance estimators.
`fit`(X[, y])	Fits the GraphLasso covariance model to X.
`get_params`([deep])	Get parameters for this estimator.
`get_precision`()	Getter for the precision matrix.
`mahalanobis`(observations)	Computes the squared Mahalanobis distances of given observations.
`score`(X_test[, y])	Computes the log-likelihood of a Gaussian data set with *self.covariance_* as an
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*alphas=4*, *n_refinements=4*, *cv=None*, *tol=0.0001*, *max_iter=100*, *mode='cd'*, *n_jobs=1*, *verbose=False*, *assume_centered=False*)

`error_norm`(*comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameterscomp_cov : array-like, 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.

ReturnsThe Mean Squared Error (in the sense of the Frobenius norm) between :

'self' and 'comp_cov' covariance estimators. :

static `fit`(*X*, *y=None*)

Fits the GraphLasso covariance model to X.

ParametersX : ndarray, shape (n_samples, n_features)

Data from which to compute the covariance estimate

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

`get_precision`()

Getter for the precision matrix.

Returnsprecision_ : array-like,

The precision matrix associated to the current covariance object.

mahalanobis (*observations*)
Computes the squared Mahalanobis distances of given observations.

Parametersobservations : array-like, shape = [n_observations, 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.

Returnsmahalanobis_distance : array, shape = [n_observations,]

Squared Mahalanobis distances of the observations.

score (*X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with *self.covariance_* as an estimator of its covariance matrix.

ParametersX_test : array-like, 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.

Returnsres : float

The likelihood of the data set with *self.covariance_* as an estimator of its covariance matrix.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.covariance.GraphLassoCV`

42.4.5 `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.

Parametersstore_precision : bool, default=True

Specify if the estimated precision is stored.

assume_centered : bool, 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 (default), data are centered before computation.

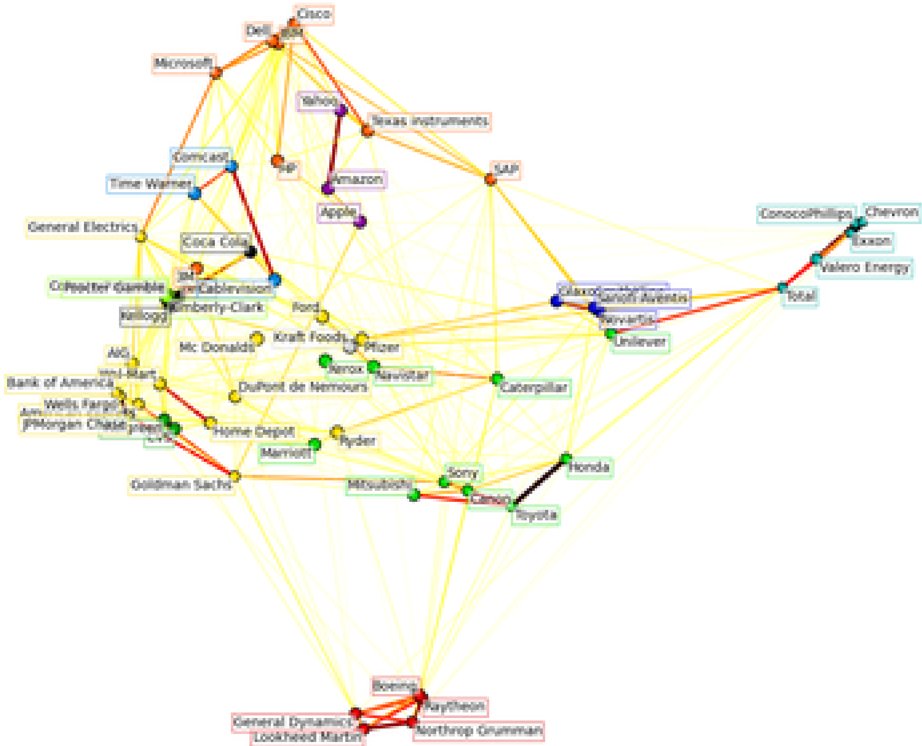


Figure 42.45: *Visualizing the stock market structure*

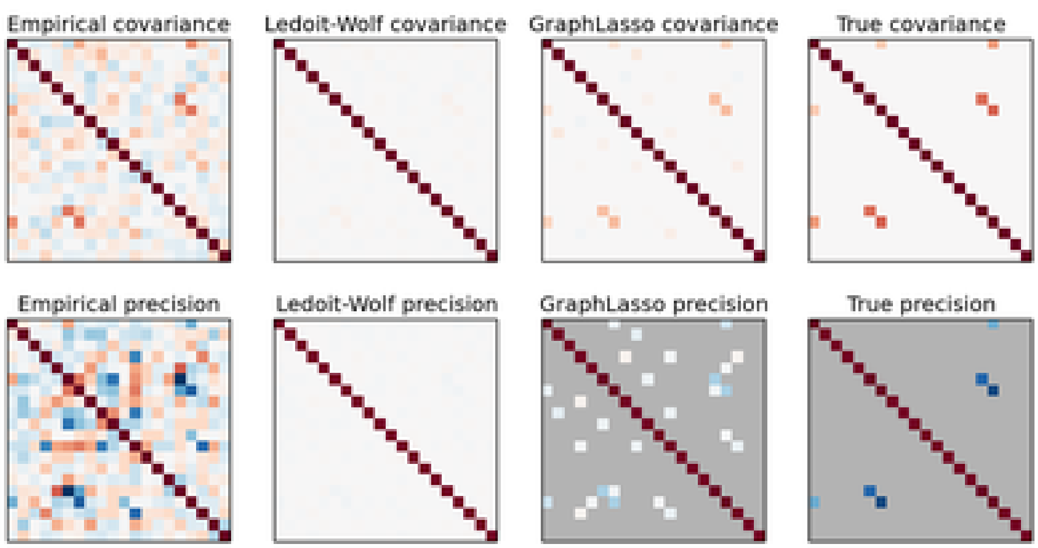


Figure 42.46: *Sparse inverse covariance estimation*

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.

Attributescovariance_ : 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.

Methods

`error_norm`(comp_cov[, norm, scaling, squared])	Computes the Mean Squared Error between two covariance estimators.
`fit`(X[, y])	Fits the Ledoit-Wolf shrunk covariance model according to the given training d
`get_params`([deep])	Get parameters for this estimator.
`get_precision`()	Getter for the precision matrix.
`mahalanobis`(observations)	Computes the squared Mahalanobis distances of given observations.
`score`(X_test[, y])	Computes the log-likelihood of a Gaussian data set with *self.covariance_* as an
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*store_precision=True*, *assume_centered=False*, *block_size=1000*)

`error_norm`(*comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameterscomp_cov : array-like, 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.

ReturnsThe Mean Squared Error (in the sense of the Frobenius norm) between :

'self' and 'comp_cov' covariance estimators. :

static fit(*X*, *y=None*)

Fits the Ledoit-Wolf shrunk covariance model according to the given training data and parameters.

ParametersX : 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 : not used, present for API consistence purpose.

Returnsself : object

Returns self.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_precision()

Getter for the precision matrix.

Returnsprecision_ : array-like,

The precision matrix associated to the current covariance object.

mahalanobis(*observations*)

Computes the squared Mahalanobis distances of given observations.

Parametersobservations : array-like, shape = [n_observations, 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.

Returnsmahalanobis_distance : array, shape = [n_observations,]

Squared Mahalanobis distances of the observations.

score(*X_test*, *y=None*)

Computes the log-likelihood of a Gaussian data set with *self.covariance_* as an estimator of its covariance matrix.

ParametersX_test : array-like, 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.

Returnsres : float

The likelihood of the data set with *self.covariance_* as an estimator of its covariance matrix.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.covariance.LedoitWolf`

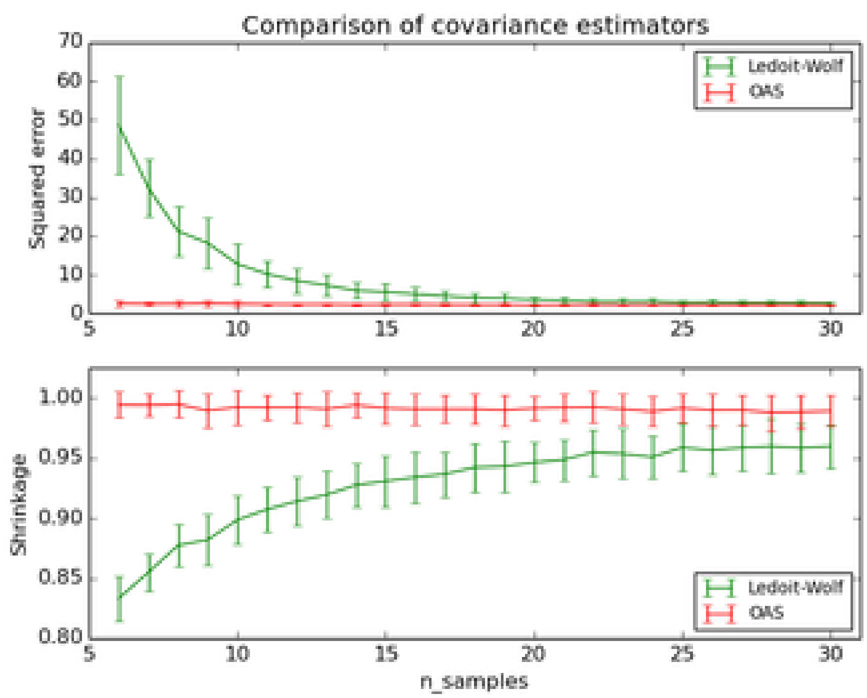


Figure 42.47: *Ledoit-Wolf vs OAS estimation*

42.4.6 `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.

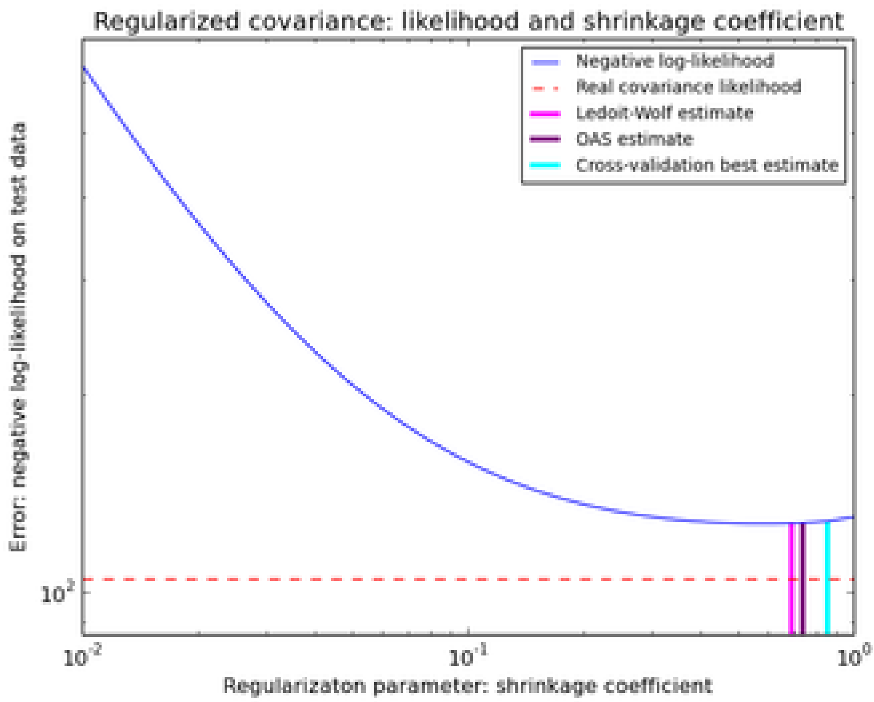


Figure 42.48: *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*

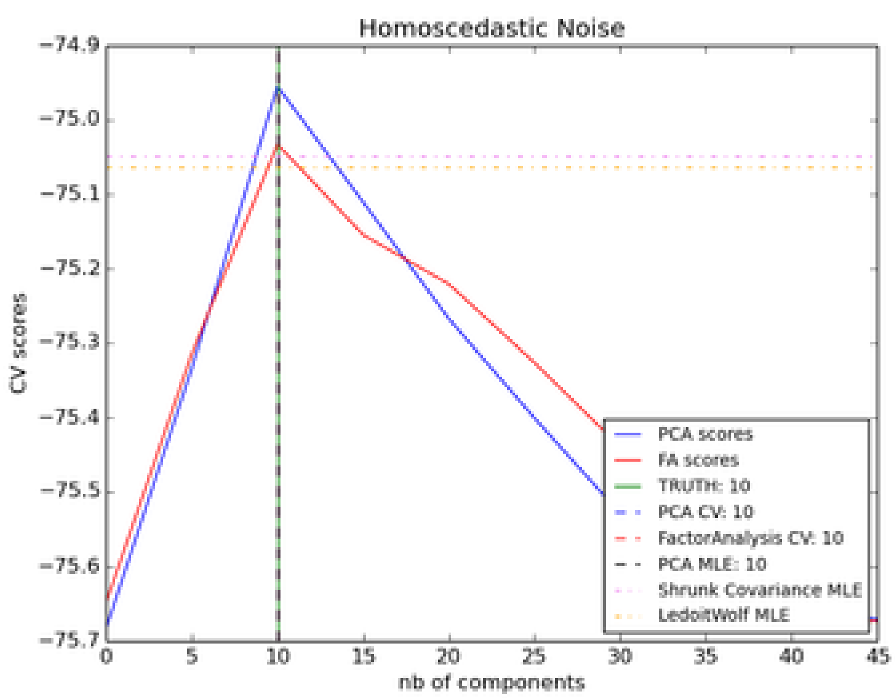


Figure 42.49: *Model selection with Probabilistic (PCA) and Factor Analysis (FA)*

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.

Parametersstore_precision : bool

Specify if the estimated precision is stored.

assume_centered : Boolean

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 : integer or numpy.RandomState, optional

The random generator used. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

Attributesraw_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

[Rouseeuw1984], [Rouseeuw1999], [Butler1993]

Methods

correct_covariance(data)	Apply a correction to raw Minimum Covariance Determinant estimates.
error_norm(comp_cov[, norm, scaling, squared])	Computes the Mean Squared Error between two covariance estimators.
fit(X[, y])	Fits a Minimum Covariance Determinant with the FastMCD algorithm.
get_params([deep])	Get parameters for this estimator.
get_precision()	Getter for the precision matrix.
mahalanobis(observations)	Computes the squared Mahalanobis distances of given observations.
reweight_covariance(data)	Re-weight raw Minimum Covariance Determinant estimates.
score(X_test[, y])	Computes the log-likelihood of a Gaussian data set with *self.covariance_* as an
set_params(**params)	Set the parameters of this estimator.

static __init__(*store_precision=True*, *assume_centered=False*, *support_fraction=None*, *random_state=None*)

static correct_covariance(*data*)

Apply a correction to raw Minimum Covariance Determinant estimates.

Correction using the empirical correction factor suggested by Rousseeuw and Van Driessen in [Rouseeuw1984].

Parametersdata : 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.

Returnscovariance_corrected : array-like, shape (n_features, n_features)

Corrected robust covariance estimate.

error_norm(*comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameterscomp_cov : array-like, 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.

ReturnsThe Mean Squared Error (in the sense of the Frobenius norm) between :

'self' and 'comp_cov' covariance estimators. :

static fit(*X*, *y=None*)

Fits a Minimum Covariance Determinant with the FastMCD algorithm.

ParametersX : 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 : not used, present for API consistence purpose.

Returnsself : object

Returns self.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_precision()
Getter for the precision matrix.

Returnsprecision_ : array-like,

The precision matrix associated to the current covariance object.

mahalanobis(*observations*)
Computes the squared Mahalanobis distances of given observations.

Parametersobservations : array-like, shape = [n_observations, 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.

Returnsmahalanobis_distance : array, shape = [n_observations,]

Squared Mahalanobis distances of the observations.

static reweight_covariance(*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). [Rouseeuw1984]

Parametersdata : 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.

Returnslocation_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.

score(*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, 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(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returns self :

Examples using `sklearn.covariance.MinCovDet`

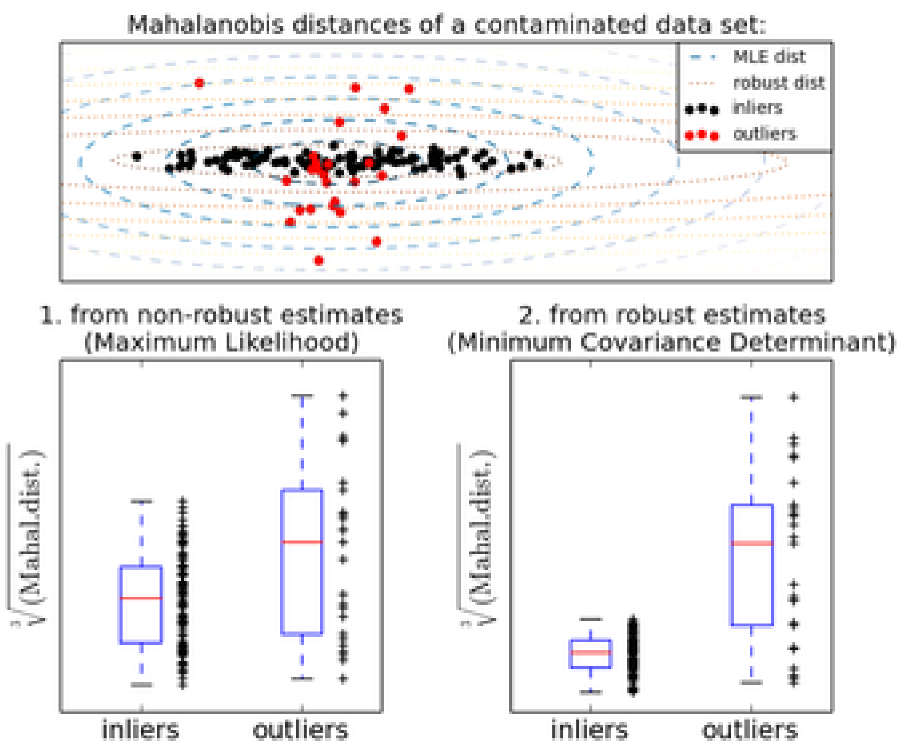


Figure 42.50: *Robust covariance estimation and Mahalanobis distances relevance*

42.4.7 `sklearn.covariance.OAS`

class `sklearn.covariance.`**OAS**(*store_precision=True*, *assume_centered=False*)

Oracle Approximating Shrinkage Estimator

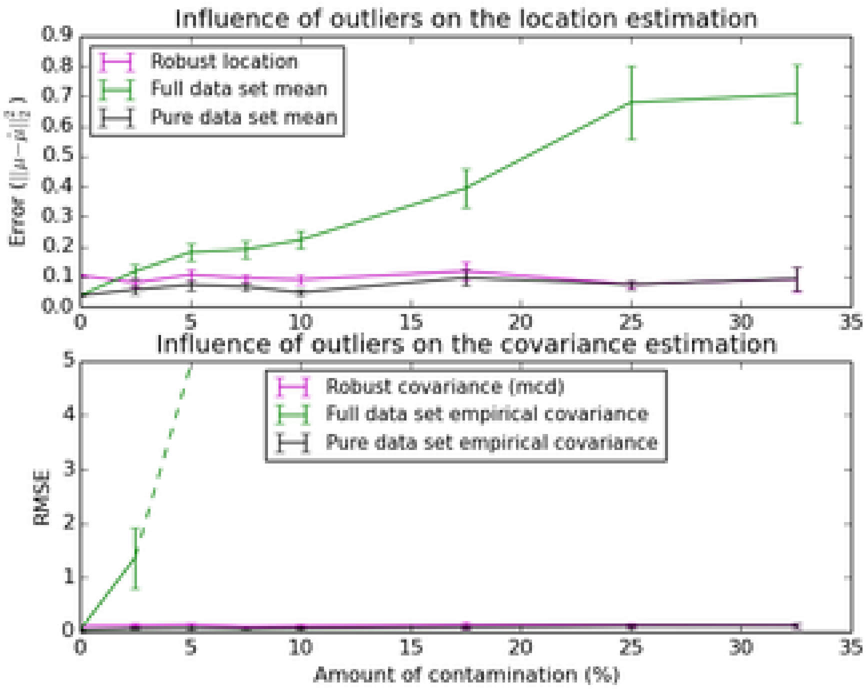


Figure 42.51: *Robust vs Empirical covariance estimate*

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. It has been taken from the Matlab program available from the authors' webpage (https://tbayes.eecs.umich.edu/yilun/covestimation).

Parametersstore_precision : bool, default=True

Specify if the estimated precision is stored.

assume_centered: bool, 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 (default), data are centered before computation.

Attributescovariance_ : 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(comp_cov[, norm, scaling, squared])	Computes the Mean Squared Error between two covariance estimators.
fit(X[, y])	Fits the Oracle Approximating Shrinkage covariance model according to the g
get_params([deep])	Get parameters for this estimator.
get_precision()	Getter for the precision matrix.
mahalanobis(observations)	Computes the squared Mahalanobis distances of given observations.
score(X_test[, y])	Computes the log-likelihood of a Gaussian data set with *self.covariance_* as an
set_params(**params)	Set the parameters of this estimator.

__init__(*store_precision=True*, *assume_centered=False*)

error_norm(*comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameterscomp_cov : array-like, 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.

ReturnsThe Mean Squared Error (in the sense of the Frobenius norm) between :

‘self‘ and ‘comp_cov‘ covariance estimators. :

static **fit**(*X*, *y=None*)

Fits the Oracle Approximating Shrinkage covariance model according to the given training data and parameters.

ParametersX : 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 : not used, present for API consistence purpose.

Returnsself: object :

Returns self.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_precision()

Getter for the precision matrix.

Returnsprecision_ : array-like,

The precision matrix associated to the current covariance object.

mahalanobis(*observations*)

Computes the squared Mahalanobis distances of given observations.

Parametersobservations : array-like, shape = [n_observations, 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.

Returnsmahalanobis_distance : array, shape = [n_observations,]

Squared Mahalanobis distances of the observations.

score(*X_test*, *y=None*)

Computes the log-likelihood of a Gaussian data set with *self.covariance_* as an estimator of its covariance matrix.

ParametersX_test : array-like, 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.

Returnsres : float

The likelihood of the data set with *self.covariance_* as an estimator of its covariance matrix.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

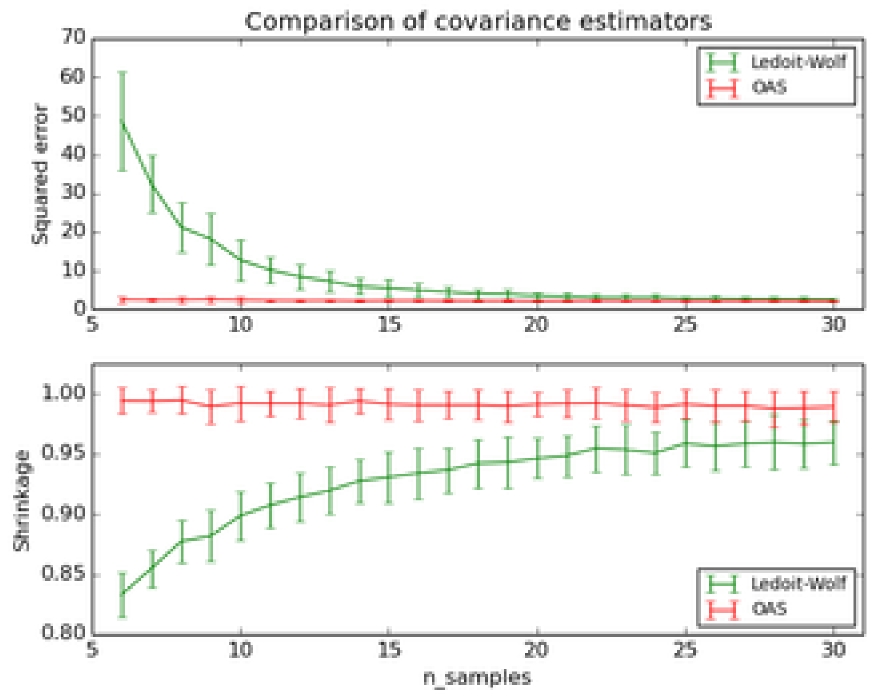


Figure 42.52: *Ledoit-Wolf vs OAS estimation*

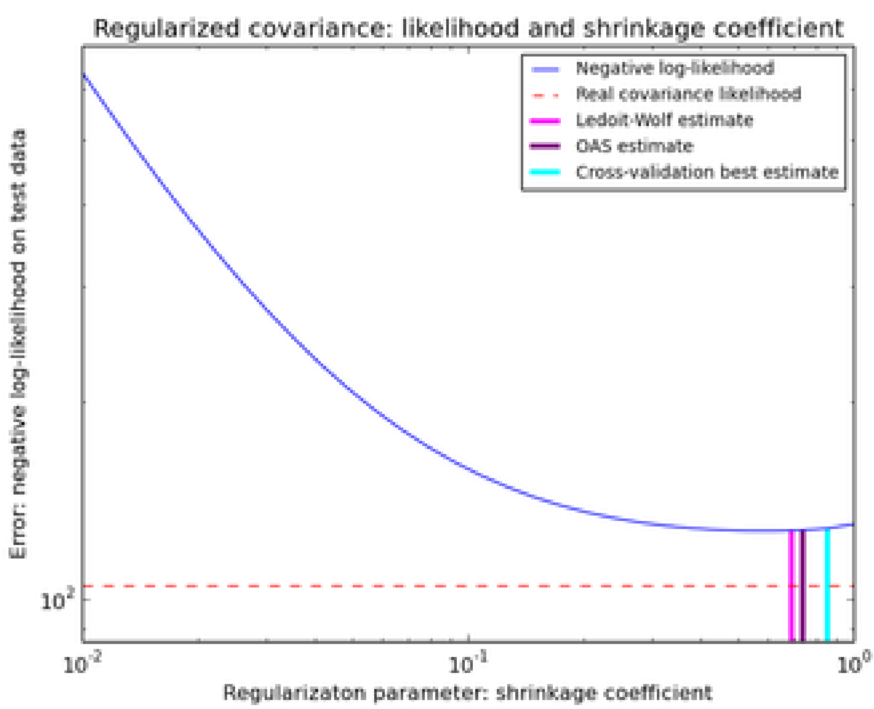


Figure 42.53: *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*

Examples using `sklearn.covariance.OAS`

42.4.8 `sklearn.covariance`.ShrunkCovariance

class `sklearn.covariance.`**`ShrunkCovariance`**(*store_precision=True*, *assume_centered=False*, *shrinkage=0.1*)

Covariance estimator with shrinkage

Parametersstore_precision : boolean, default True

Specify if the estimated precision is stored

shrinkage : float, 0 <= shrinkage <= 1, default 0.1

Coefficient in the convex combination used for the computation of the shrunk estimate.

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.

Attributescovariance_ : 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 regularized covariance is given by

(1 - shrinkage)*cov

•shrinkage*mu*np.identity(n_features)

where mu = trace(cov) / n_features

Methods

`error_norm`(comp_cov[, norm, scaling, squared])	Computes the Mean Squared Error between two covariance estimators.
`fit`(X[, y])	Fits the shrunk covariance model according to the given training data and para
`get_params`([deep])	Get parameters for this estimator.
`get_precision`()	Getter for the precision matrix.
`mahalanobis`(observations)	Computes the squared Mahalanobis distances of given observations.
`score`(X_test[, y])	Computes the log-likelihood of a Gaussian data set with *self.covariance_* as an
`set_params`(**params)	Set the parameters of this estimator.

static **`__init__`**(*store_precision=True*, *assume_centered=False*, *shrinkage=0.1*)

`error_norm`(*comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameterscomp_cov : array-like, 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.

ReturnsThe Mean Squared Error (in the sense of the Frobenius norm) between :

'self' and 'comp_cov' covariance estimators. :

static fit (*X*, *y=None*)

Fits the shrunk covariance model according to the given training data and parameters.

ParametersX : 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 : not used, present for API consistence purpose.

Returnsself : object

Returns self.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_precision ()

Getter for the precision matrix.

Returnsprecision_ : array-like,

The precision matrix associated to the current covariance object.

mahalanobis (*observations*)

Computes the squared Mahalanobis distances of given observations.

Parametersobservations : array-like, shape = [n_observations, 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.

Returnsmahalanobis_distance : array, shape = [n_observations,]

Squared Mahalanobis distances of the observations.

score (*X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with *self.covariance_* as an estimator of its covariance matrix.

ParametersX_test : array-like, 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.

Returnsres : float

The likelihood of the data set with *self.covariance_* as an estimator of its covariance matrix.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.covariance.ShrunkCovariance`

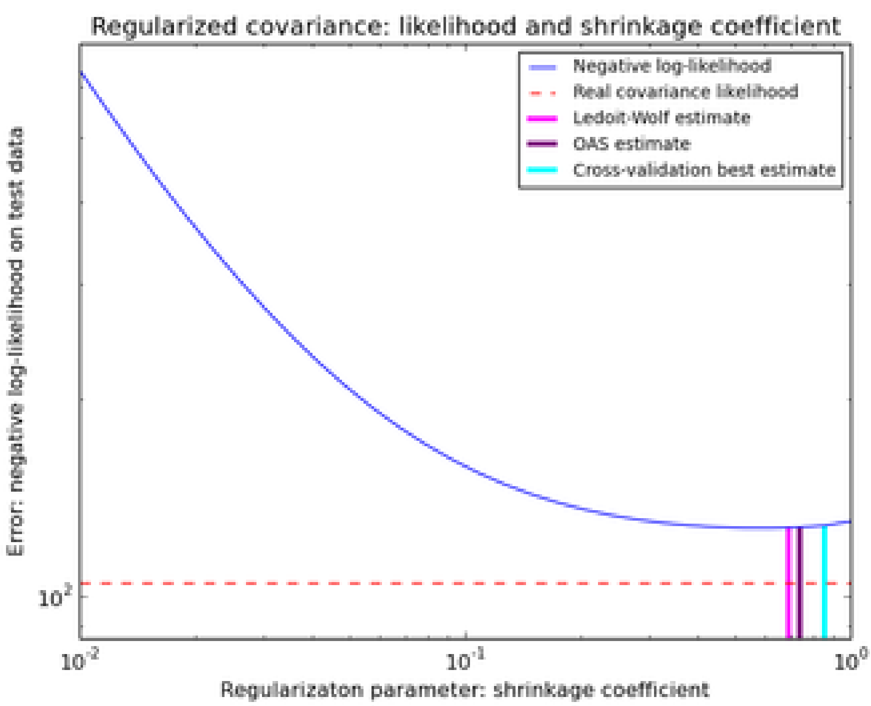


Figure 42.54: *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*

`covariance.empirical_covariance`(X[, ...])	Computes the Maximum likelihood covariance estimator

Continued on next page

Table 42.32 – continued from previous page

covariance.ledoit_wolf(X[, assume_centered, ...])	Estimates the shrunk Ledoit-Wolf covariance matrix.
covariance.shrunk_covariance(emp_cov[, ...])	Calculates a covariance matrix shrunk on the diagonal
covariance.oas(X[, assume_centered])	Estimate covariance with the Oracle Approximating Shrinkage algorith
covariance.graph_lasso(emp_cov, alpha[, ...])	l1-penalized covariance estimator

42.4.9 sklearn.covariance.empirical_covariance

sklearn.covariance.**empirical_covariance**(*X*, *assume_centered=False*)

Computes the Maximum likelihood covariance estimator

ParametersX : ndarray, shape (n_samples, n_features)

Data from which to compute the covariance estimate

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.

Returnscovariance : 2D ndarray, shape (n_features, n_features)

Empirical covariance (Maximum Likelihood Estimator).

Examples using sklearn.covariance.empirical_covariance

42.4.10 sklearn.covariance.ledoit_wolf

sklearn.covariance.**ledoit_wolf**(*X*, *assume_centered=False*, *block_size=1000*)

Estimates the shrunk Ledoit-Wolf covariance matrix.

ParametersX : array-like, shape (n_samples, n_features)

Data from which to compute the covariance estimate

assume_centered : boolean, default=False

If True, data are not centered before computation. Useful to work with data whose mean is significantly equal to zero but is not exactly zero. If False, data are 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.

Returnsshrunk_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

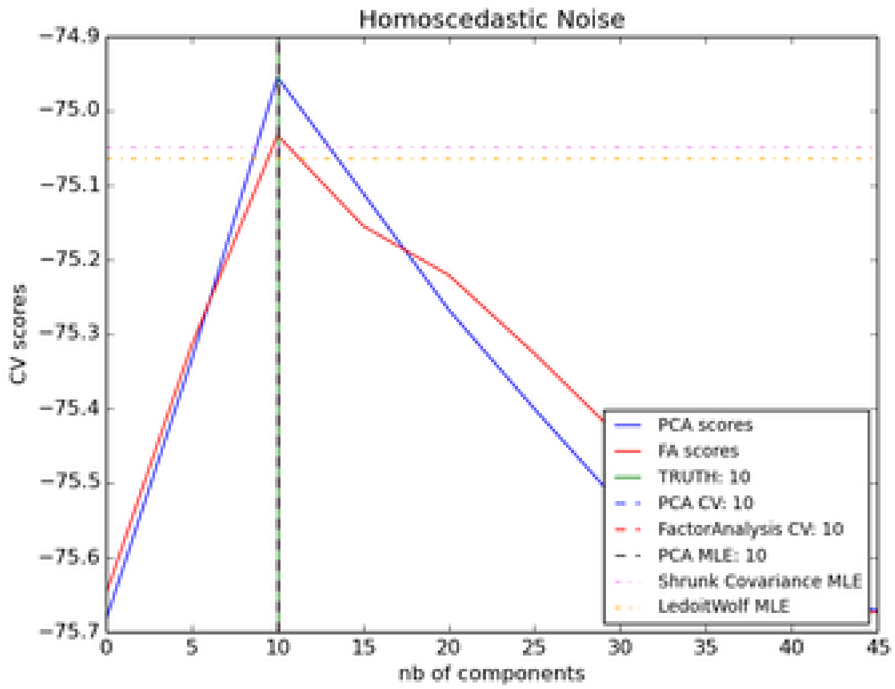


Figure 42.55: *Model selection with Probabilistic (PCA) and Factor Analysis (FA)*

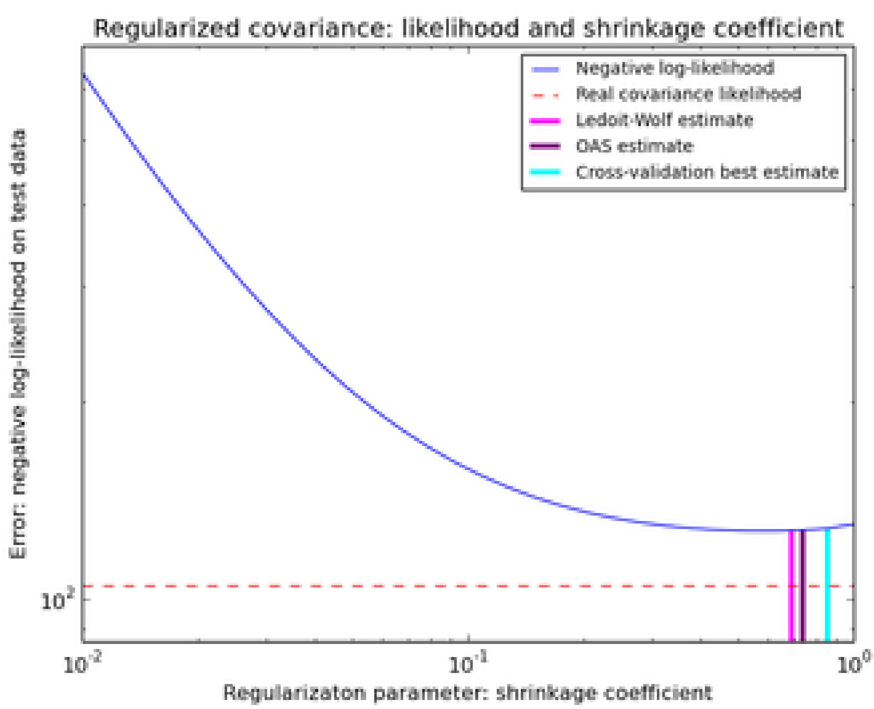


Figure 42.56: *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*

•shrinkage * mu * np.identity(n_features)

where mu = trace(cov) / n_features

Examples using `sklearn.covariance.ledoit_wolf`

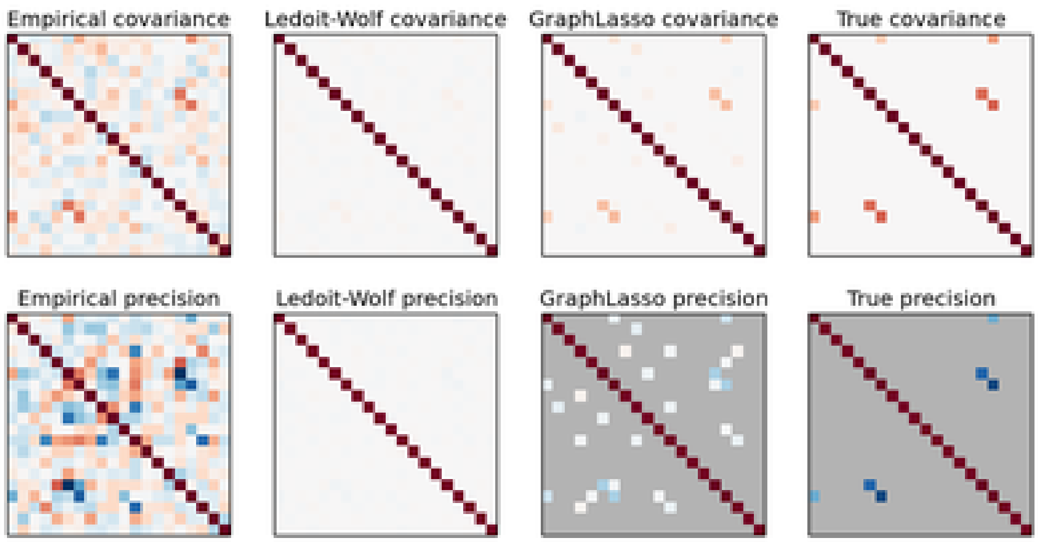


Figure 42.57: *Sparse inverse covariance estimation*

42.4.11 `sklearn.covariance.shrunk_covariance`

`sklearn.covariance.`**`shrunk_covariance`**(*emp_cov*, *shrinkage=0.1*)

Calculates a covariance matrix shrunk on the diagonal

Parametersemp_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.

Returnsshrunk_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

42.4.12 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 are not centered before computation. Useful to work with data whose mean is significantly equal to zero but is not exactly zero. If False, data are 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 does not correspond to the one given in the article. It has been taken from the MATLAB program available from the author's webpage (https://tbayes.eecs.umich.edu/yilun/covestimation).

42.4.13 sklearn.covariance.graph_lasso

`sklearn.covariance.graph_lasso` (*emp_cov*, *alpha*, *cov_init=None*, *mode='cd'*, *tol=0.0001*, *max_iter=100*, *verbose=False*, *return_costs=False*, *eps=2.220446049250313e-16*, *return_n_iter=False*)

l1-penalized covariance estimator

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.

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.

Returnscovariance : 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:

`GraphLasso`, `GraphLassoCV`

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.

42.5 `sklearn.cross_validation`: Cross Validation

The `sklearn.cross_validation` module includes utilities for cross- validation and performance evaluation.

User guide: See the *Cross-validation: evaluating estimator performance* section for further details.

cross_validation.KFold(n[, n_folds, ...])	K-Folds cross validation iterator.
cross_validation.LeaveOneLabelOut(labels[, ...])	Leave-One-Label_Out cross-validation iterator
cross_validation.LeaveOneOut(n[, indices])	Leave-One-Out cross validation iterator.
cross_validation.LeavePLabelOut(labels, p[, ...])	Leave-P-Label_Out cross-validation iterator
cross_validation.LeavePOut(n, p[, indices])	Leave-P-Out cross validation iterator
cross_validation.PredefinedSplit(test_fold)	Predefined split cross validation iterator
cross_validation.StratifiedKFold(y[, ...])	Stratified K-Folds cross validation iterator
cross_validation.ShuffleSplit(n[, n_iter, ...])	Random permutation cross-validation iterator.
cross_validation.StratifiedShuffleSplit(y[, ...])	Stratified ShuffleSplit cross validation iterator

42.5.1 sklearn.cross_validation.KFold

class sklearn.cross_validation.**KFold**(*n*, *n_folds=3*, *indices=None*, *shuffle=False*, *random_state=None*)

K-Folds cross validation iterator.

Provides train/test indices to split data in train test sets. Split dataset into k consecutive folds (without shuffling).

Each fold is then used a validation set once while the k - 1 remaining fold form the training set.

Parametersn : int

Total number of elements.

n_folds : int, default=3

Number of folds. Must be at least 2.

shuffle : boolean, optional

Whether to shuffle the data before splitting into batches.

random_state : None, int or RandomState

Pseudo-random number generator state used for random sampling. If None, use default numpy RNG for shuffling

See also:

StratifiedKFoldtake label information into account to avoid building

folds, classification

Notes

The first n % n_folds folds have size n // n_folds + 1, other folds have size n // n_folds.

Examples

```
>>> from sklearn import cross_validation
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([1, 2, 3, 4])
>>> kf = cross_validation.KFold(4, n_folds=2)
>>> len(kf)
2
>>> print(kf)
```

```
sklearn.cross_validation.KFold(n=4, n_folds=2, shuffle=False,
                               random_state=None)
>>> for train_index, test_index in kf:
...    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: [0 1] TEST: [2 3]
.. automethod:: __init__
```

Examples using `sklearn.cross_validation.KFold`

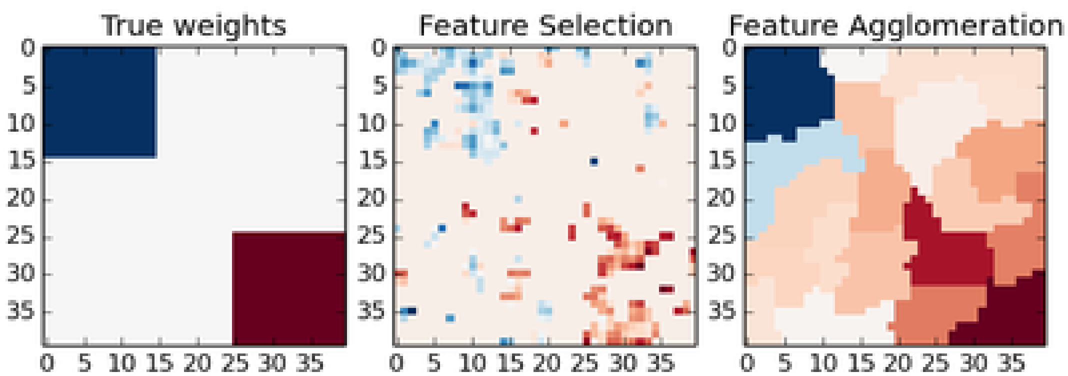


Figure 42.58: *Feature agglomeration vs. univariate selection*

42.5.2 `sklearn.cross_validation`.LeaveOneLabelOut

class `sklearn.cross_validation.`**`LeaveOneLabelOut`**(*labels*, *indices=None*)

Leave-One-Label_Out cross-validation iterator

Provides train/test indices to split data according to a third-party provided label. This label information can be used to encode arbitrary domain specific stratifications of the samples as integers.

For instance the labels could be the year of collection of the samples and thus allow for cross-validation against time-based splits.

Parameterslabels : array-like of int with shape (n_samples,)

Arbitrary domain-specific stratification of the data to be used to draw the splits.

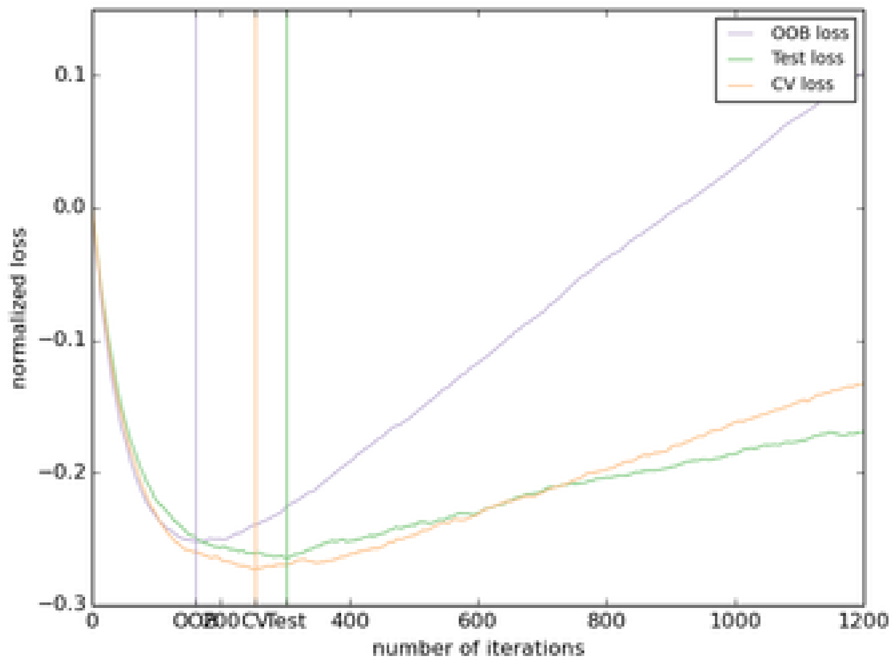


Figure 42.59: *Gradient Boosting Out-of-Bag estimates*

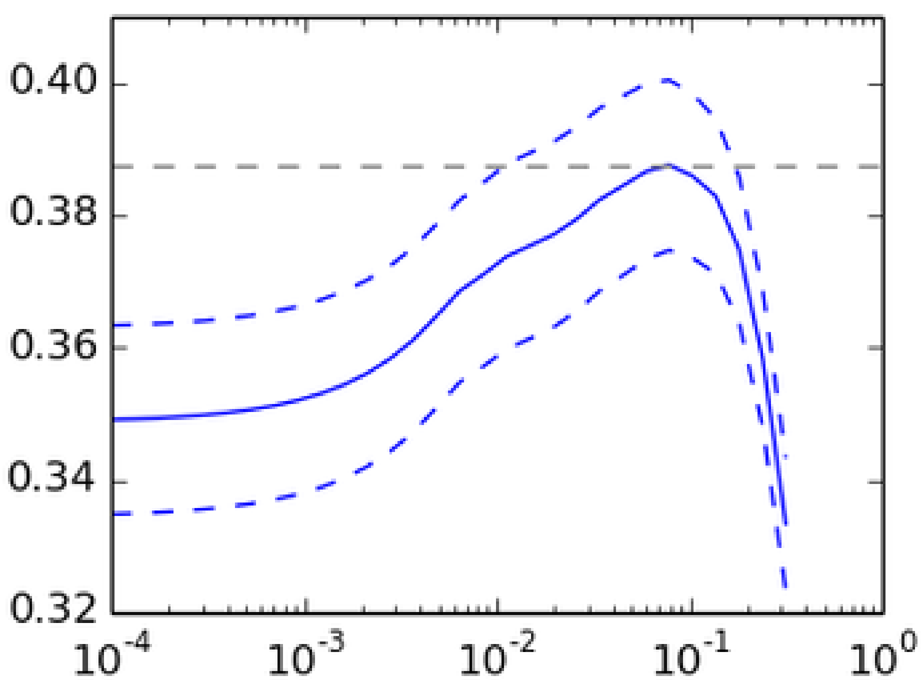


Figure 42.60: *Cross-validation on diabetes Dataset Exercise*

Figure 42.61: *Gaussian Processes regression: goodness-of-fit on the 'diabetes' dataset*

Examples

```
>>> from sklearn import cross_validation
>>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
>>> y = np.array([1, 2, 1, 2])
>>> labels = np.array([1, 1, 2, 2])
>>> lol = cross_validation.LeaveOneLabelOut(labels)
>>> len(lol)
2
>>> print(lol)
sklearn.cross_validation.LeaveOneLabelOut(labels=[1 1 2 2])
>>> for train_index, test_index in lol:
...     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]
.. automethod:: __init__
```

42.5.3 `sklearn.cross_validation`.LeaveOneOut

class `sklearn.cross_validation.`**`LeaveOneOut`**(*n*, *indices=None*)

Leave-One-Out cross validation iterator.

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(n)` is equivalent to `KFold(n, n_folds=n)` and `LeavePOut(n, p=1)`.

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, StratifiedKFold or ShuffleSplit.

Parametersn : int

Total number of elements in dataset.

See also:

`LeaveOneLabelOut`, `domain-specific`

Examples

```
>>> from sklearn import cross_validation
>>> X = np.array([[1, 2], [3, 4]])
>>> y = np.array([1, 2])
>>> loo = cross_validation.LeaveOneOut(2)
>>> len(loo)
2
>>> print(loo)
sklearn.cross_validation.LeaveOneOut(n=2)
>>> for train_index, test_index in loo:
...    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]
.. automethod:: __init__
```

42.5.4 `sklearn.cross_validation.LeavePLabelOut`

class `sklearn.cross_validation.`**`LeavePLabelOut`**(*labels*, *p*, *indices=None*)

Leave-P-Label_Out cross-validation iterator

Provides train/test indices to split data according to a third-party provided label. This label information can be used to encode arbitrary domain specific stratifications of the samples as integers.

For instance the labels could be the year of collection of the samples and thus allow for cross-validation against time-based splits.

The difference between LeavePLabelOut and LeaveOneLabelOut is that the former builds the test sets with all the samples assigned to `p` different values of the labels while the latter uses samples all assigned the same labels.

Parameterslabels : array-like of int with shape (n_samples,)

Arbitrary domain-specific stratification of the data to be used to draw the splits.

p : int

Number of samples to leave out in the test split.

Examples

```
>>> from sklearn import cross_validation
>>> X = np.array([[1, 2], [3, 4], [5, 6]])
>>> y = np.array([1, 2, 1])
>>> labels = np.array([1, 2, 3])
>>> lpl = cross_validation.LeavePLabelOut(labels, p=2)
>>> len(lpl)
```

```
3
>>> print(lpl)
sklearn.cross_validation.LeavePLabelOut(labels=[1 2 3], p=2)
>>> for train_index, test_index in lpl:
...     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]
.. automethod:: __init__
```

42.5.5 `sklearn.cross_validation`.LeavePOut

class `sklearn.cross_validation.`**`LeavePOut`**(*n*, *p*, *indices=None*)

Leave-P-Out cross validation iterator

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(n, p)` is NOT equivalent to `KFold(n, n_folds=n // 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.

Parametersn : int

Total number of elements in dataset.

p : int

Size of the test sets.

Examples

```
>>> from sklearn import cross_validation
>>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
>>> y = np.array([1, 2, 3, 4])
>>> lpo = cross_validation.LeavePOut(4, 2)
>>> len(lpo)
6
>>> print(lpo)
sklearn.cross_validation.LeavePOut(n=4, p=2)
>>> for train_index, test_index in lpo:
...     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]
.. automethod:: __init__
```

42.5.6 sklearn.cross_validation.PredefinedSplit

class sklearn.cross_validation.**PredefinedSplit**(*test_fold*, *indices=None*)

Predefined split cross validation iterator

Splits the data into training/test set folds according to a predefined scheme. Each sample can be assigned to at most one test set fold, as specified by the user through the test_fold parameter.

Parameterstest_fold : "array-like, shape (n_samples,)

test_fold[i] gives the test set fold of sample i. A value of -1 indicates that the corresponding sample is not part of any test set folds, but will instead always be put into the training fold.

Examples

```
>>> from sklearn.cross_validation import PredefinedSplit
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 1, 1])
>>> ps = PredefinedSplit(test_fold=[0, 1, -1, 1])
>>> len(ps)
2
>>> print(ps)
sklearn.cross_validation.PredefinedSplit(test_fold=[ 0  1 -1  1])
>>> for train_index, test_index in ps:
...    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]
.. automethod:: __init__
```

42.5.7 sklearn.cross_validation.StratifiedKFold

class sklearn.cross_validation.**StratifiedKFold**(*y*, *n_folds=3*, *indices=None*, *shuffle=False*, *random_state=None*)

Stratified K-Folds cross validation iterator

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.

Parametersy : array-like, [n_samples]

Samples to split in K folds.

n_folds : int, default=3

Number of folds. Must be at least 2.

shuffle : boolean, optional

Whether to shuffle each stratification of the data before splitting into batches.

random_state : None, int or RandomState

Pseudo-random number generator state used for random sampling. If None, use default numpy RNG for shuffling

Notes

All the folds have size trunc(n_samples / n_folds), the last one has the complementary.

Examples

```
>>> from sklearn import cross_validation
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 1, 1])
>>> skf = cross_validation.StratifiedKFold(y, n_folds=2)
>>> len(skf)
2
>>> print(skf)
sklearn.cross_validation.StratifiedKFold(labels=[0 0 1 1], n_folds=2,
                                          shuffle=False, random_state=None)
>>> for train_index, test_index in skf:
...    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]
.. automethod:: __init__
```

Examples using `sklearn.cross_validation.StratifiedKFold`

42.5.8 `sklearn.cross_validation`.ShuffleSplit

class `sklearn.cross_validation.`**`ShuffleSplit`**(*n*, *n_iter=10*, *test_size=0.1*, *train_size=None*, *indices=None*, *random_state=None*, *n_iterations=None*)

Random permutation cross-validation iterator.

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.

Parametersn : int

Total number of elements in the dataset.

n_iter : int (default 10)

Number of re-shuffling & splitting iterations.

test_size : float (default 0.1), int, or None

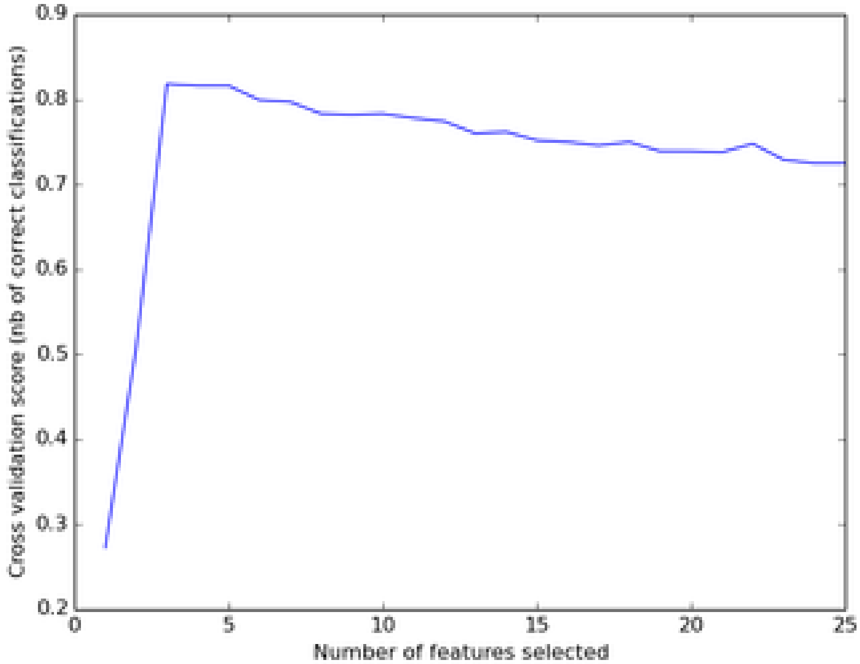


Figure 42.62: *Recursive feature elimination with cross-validation*

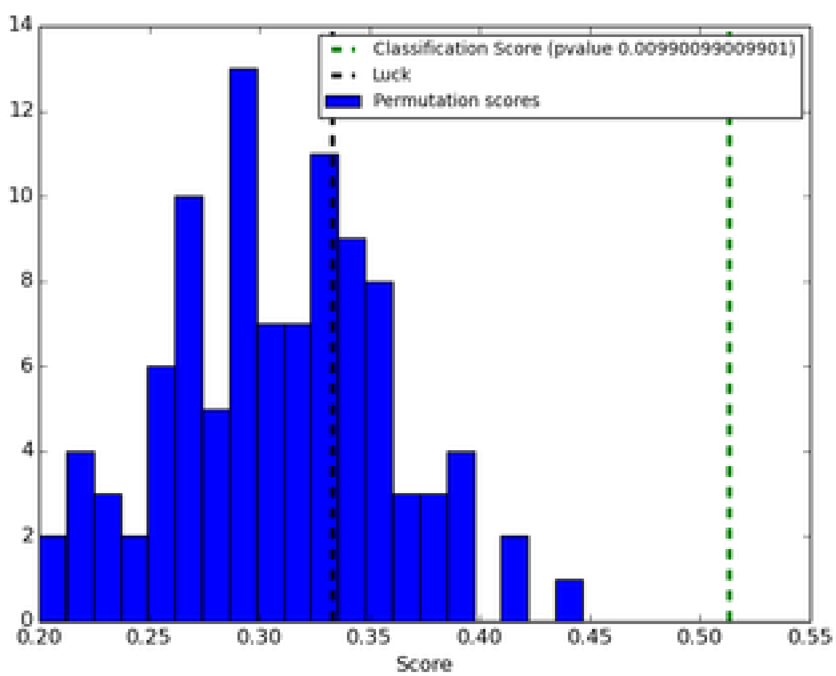


Figure 42.63: *Test with permutations the significance of a classification score*

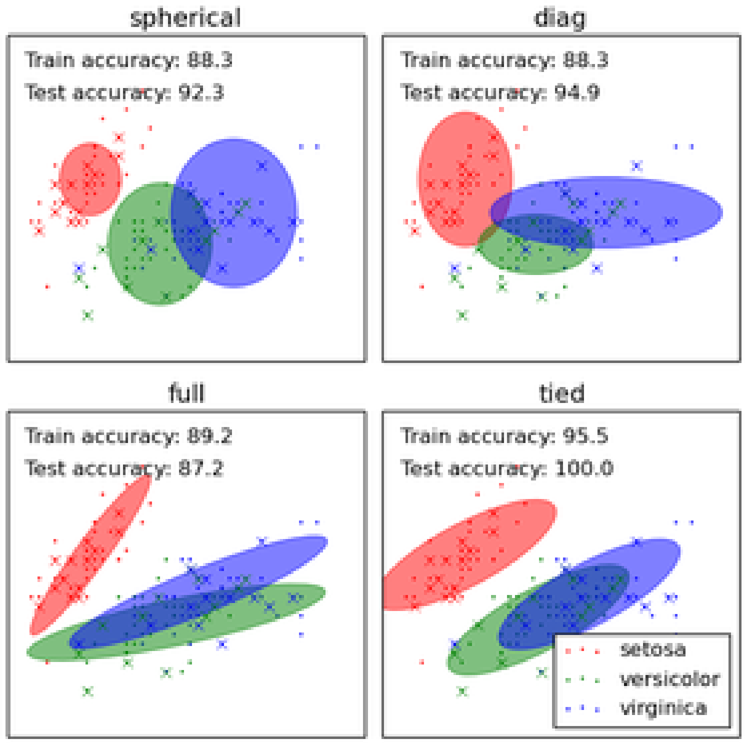


Figure 42.64: *GMM classification*

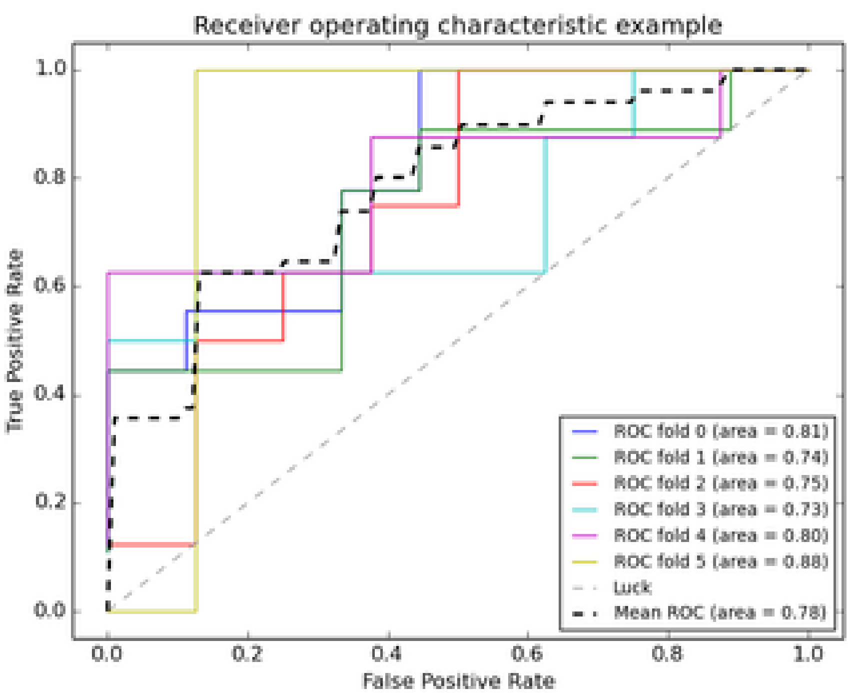


Figure 42.65: *Receiver Operating Characteristic (ROC) with cross validation*

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 automatically set to the complement of the 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 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 or RandomState

Pseudo-random number generator state used for random sampling.

See also:

Bootstrapcross-validation using re-sampling with replacement.

Examples

```
>>> from sklearn import cross_validation
>>> rs = cross_validation.ShuffleSplit(4, n_iter=3,
...     test_size=.25, random_state=0)
>>> len(rs)
3
>>> print(rs)
...
ShuffleSplit(4, n_iter=3, test_size=0.25, ...)
>>> for train_index, test_index in rs:
...    print("TRAIN:", train_index, "TEST:", test_index)
...
TRAIN: [3 1 0] TEST: [2]
TRAIN: [2 1 3] TEST: [0]
TRAIN: [0 2 1] TEST: [3]

>>> rs = cross_validation.ShuffleSplit(4, n_iter=3,
...     train_size=0.5, test_size=.25, random_state=0)
>>> for train_index, test_index in rs:
...    print("TRAIN:", train_index, "TEST:", test_index)
...
TRAIN: [3 1] TEST: [2]
TRAIN: [2 1] TEST: [0]
TRAIN: [0 2] TEST: [3]
.. automethod:: __init__
```

Examples using `sklearn.cross_validation.ShuffleSplit`

42.5.9 `sklearn.cross_validation`.StratifiedShuffleSplit

class `sklearn.cross_validation.`**`StratifiedShuffleSplit`**(*y*, *n_iter=10*, *test_size=0.1*, *train_size=None*, *indices=None*, *random_state=None*, *n_iterations=None*)

Stratified ShuffleSplit cross validation iterator

Provides train/test indices to split data in train test sets.

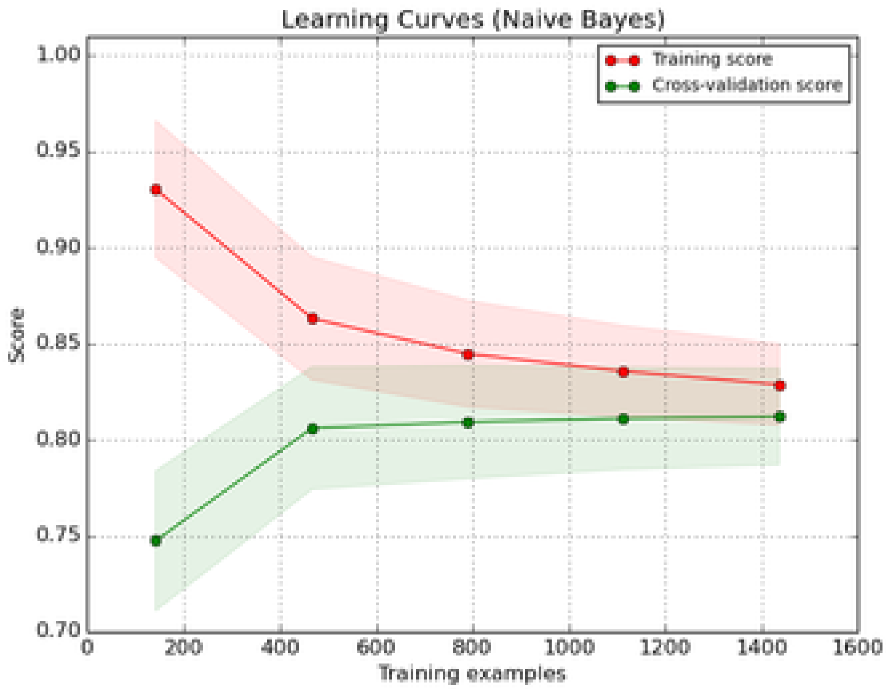


Figure 42.66: *Plotting Learning Curves*

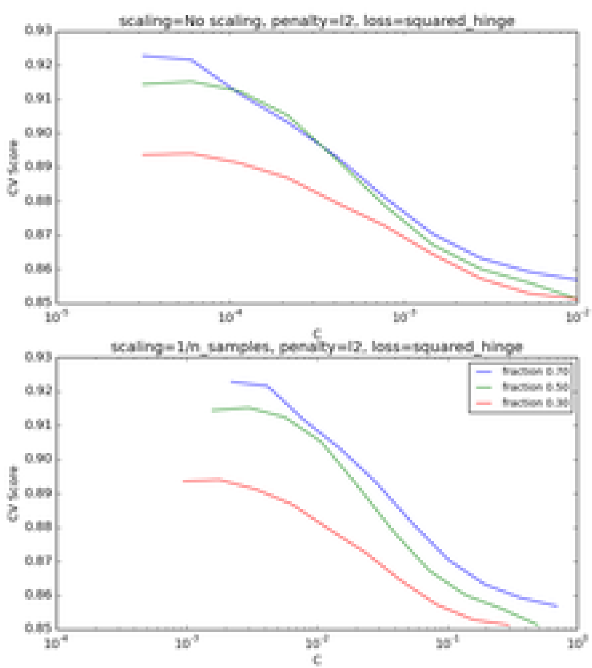


Figure 42.67: *Scaling the regularization parameter for SVCs*

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.

Parametersy : array, [n_samples]

Labels of samples.

n_iter : int (default 10)

Number of re-shuffling & splitting iterations.

test_size : float (default 0.1), int, or 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 automatically set to the complement of the 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 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 or RandomState

Pseudo-random number generator state used for random sampling.

Examples

```
>>> from sklearn.cross_validation import StratifiedShuffleSplit
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 1, 1])
>>> sss = StratifiedShuffleSplit(y, 3, test_size=0.5, random_state=0)
>>> len(sss)
3
>>> print(sss)
StratifiedShuffleSplit(labels=[0 0 1 1], n_iter=3, ...)
>>> for train_index, test_index in sss:
...    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: [3 0]
TRAIN: [0 2] TEST: [1 3]
TRAIN: [0 2] TEST: [3 1]
.. automethod:: __init__
```

Examples using `sklearn.cross_validation.StratifiedShuffleSplit`

`cross_validation.train_test_split`(*arrays, ...)	Split arrays or matrices into random train and test subsets
`cross_validation.cross_val_score`(estimator, X)	Evaluate a score by cross-validation
`cross_validation.cross_val_predict`(estimator, X)	Generate cross-validated estimates for each input data point
`cross_validation.permutation_test_score`(...)	Evaluate the significance of a cross-validated score with permutati
`cross_validation.check_cv`(cv[, X, y, classifier])	Input checker utility for building a CV in a user friendly way.

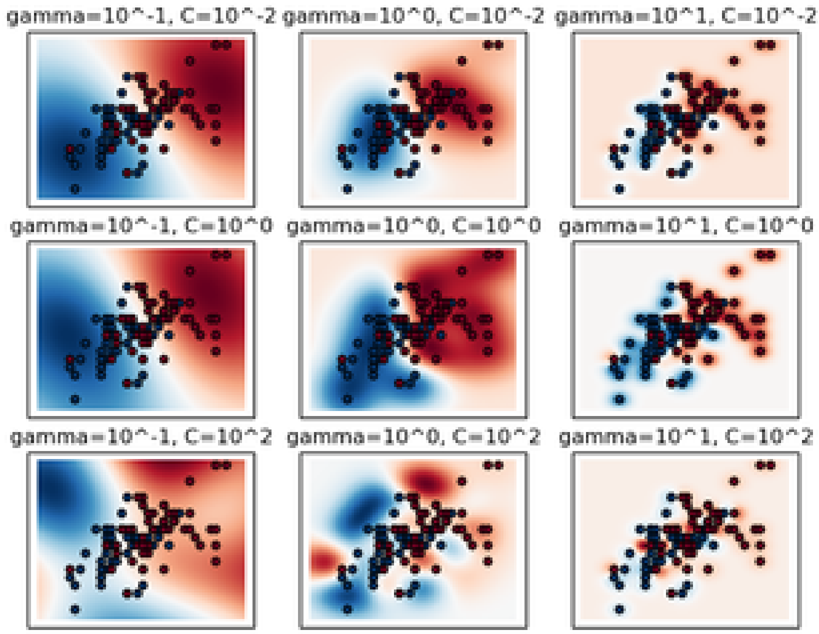


Figure 42.68: *RBF SVM parameters*

42.5.10 `sklearn.cross_validation`.train_test_split

`sklearn.cross_validation.`**`train_test_split`**(**arrays*, ***options*)

Split arrays or matrices into random train and test subsets

Quick utility that wraps input validation and `next(iter(ShuffleSplit(n_samples)))` and application to input data into a single call for splitting (and optionally subsampling) data in a oneliner.

Parameters*arrays : sequence of arrays or scipy.sparse matrices with same shape[0]

Python lists or tuples occurring in arrays are converted to 1D numpy arrays.

test_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 test split. If int, represents the absolute number of test samples. If None, the value is automatically set to the complement of the train size. If train size is also None, test size is set to 0.25.

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 or RandomState

Pseudo-random number generator state used for random sampling.

Returnssplitting : list of arrays, length=2 * len(arrays)

List containing train-test split of input array.

Examples

```
>>> import numpy as np
>>> from sklearn.cross_validation import train_test_split
>>> a, b = np.arange(10).reshape((5, 2)), range(5)
>>> a
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> list(b)
[0, 1, 2, 3, 4]

>>> a_train, a_test, b_train, b_test = train_test_split(
...     a, b, test_size=0.33, random_state=42)
...
>>> a_train
array([[4, 5],
       [0, 1],
       [6, 7]])
>>> b_train
[2, 0, 3]
>>> a_test
array([[2, 3],
       [8, 9]])
>>> b_test
[1, 4]
```

Examples using `sklearn.cross_validation.train_test_split`

Figure 42.69: *Faces recognition example using eigenfaces and SVMs*

42.5.11 `sklearn.cross_validation`.cross_val_score

`sklearn.cross_validation.`**`cross_val_score`**(*estimator*, *X*, *y=None*, *scoring=None*, *cv=None*, *n_jobs=1*, *verbose=0*, *fit_params=None*, *pre_dispatch='2*n_jobs'*)

Evaluate a score by cross-validation

Parametersestimator : estimator object implementing 'fit'

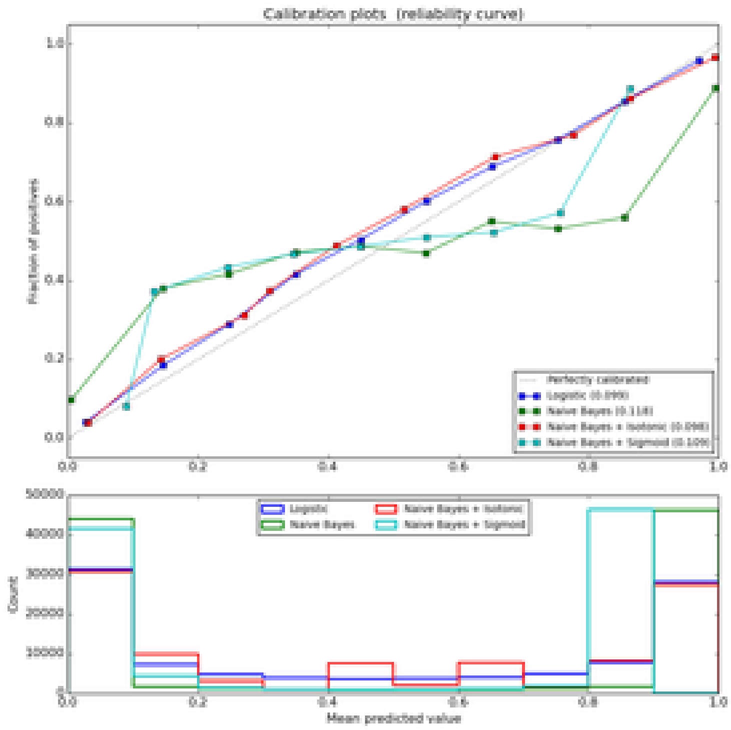


Figure 42.70: *Probability Calibration curves*

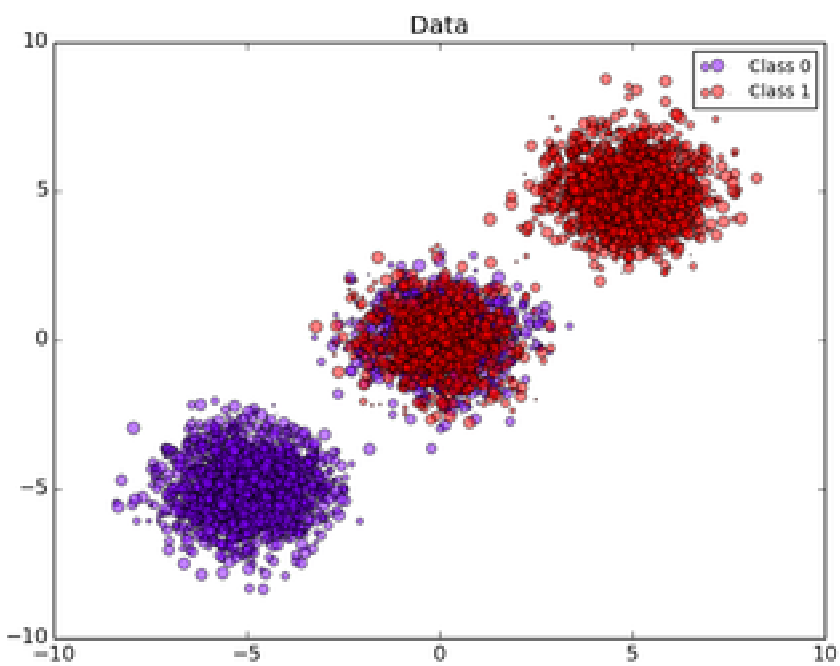


Figure 42.71: *Probability calibration of classifiers*

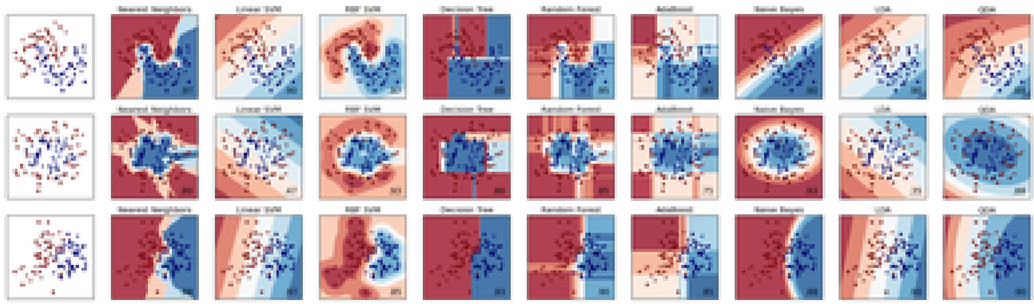


Figure 42.72: *Classifier comparison*

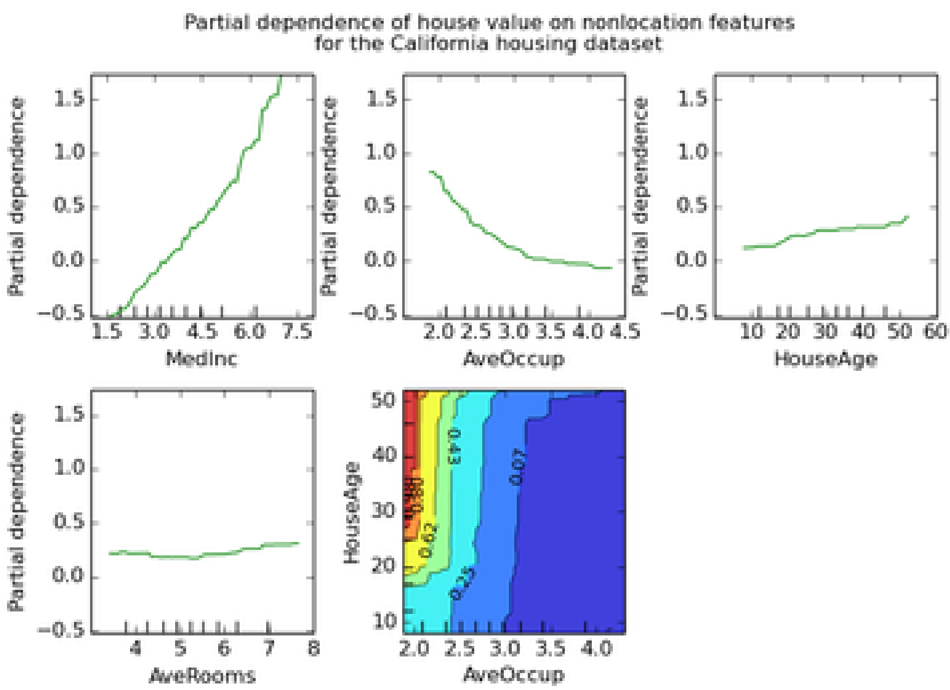


Figure 42.73: *Partial Dependence Plots*

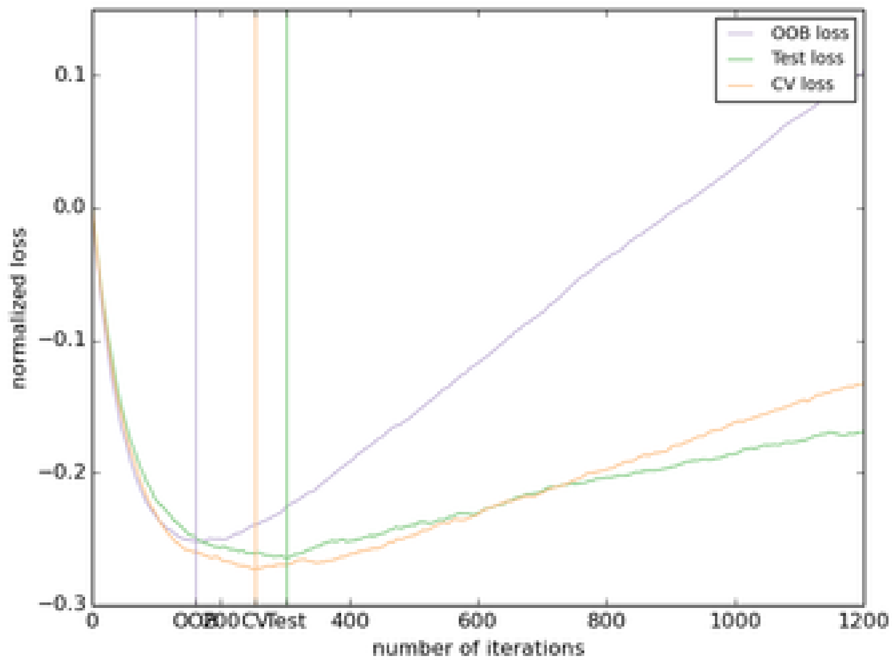


Figure 42.74: *Gradient Boosting Out-of-Bag estimates*

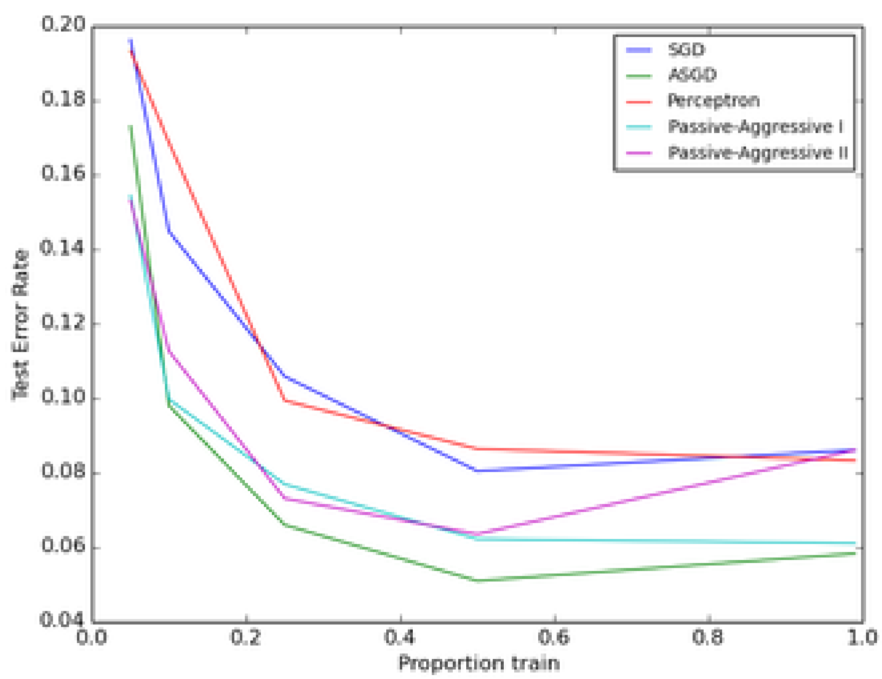


Figure 42.75: *Comparing various online solvers*

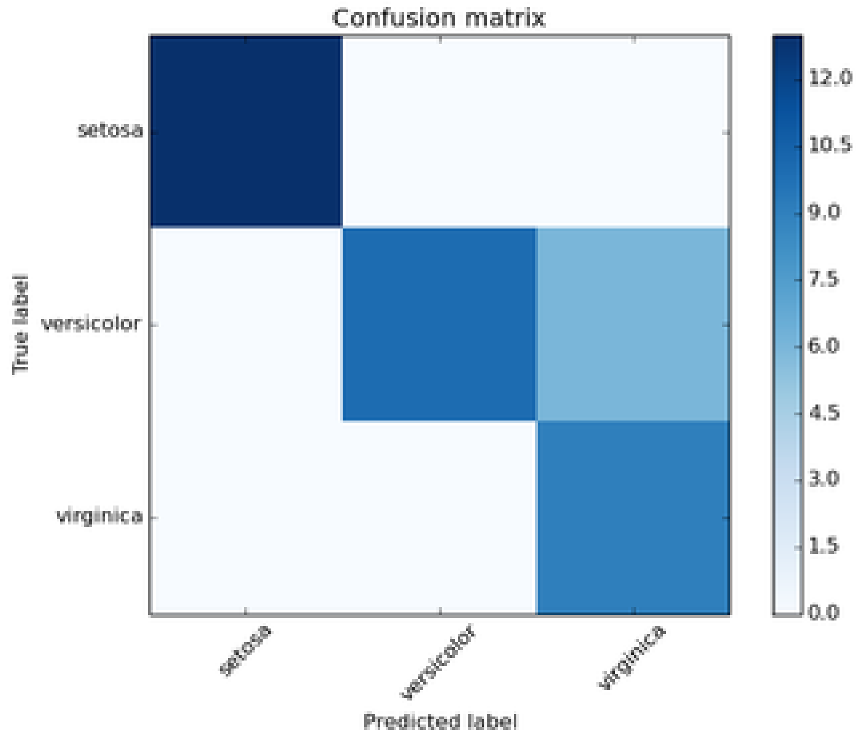


Figure 42.76: *Confusion matrix*

Figure 42.77: *Parameter estimation using grid search with cross-validation*

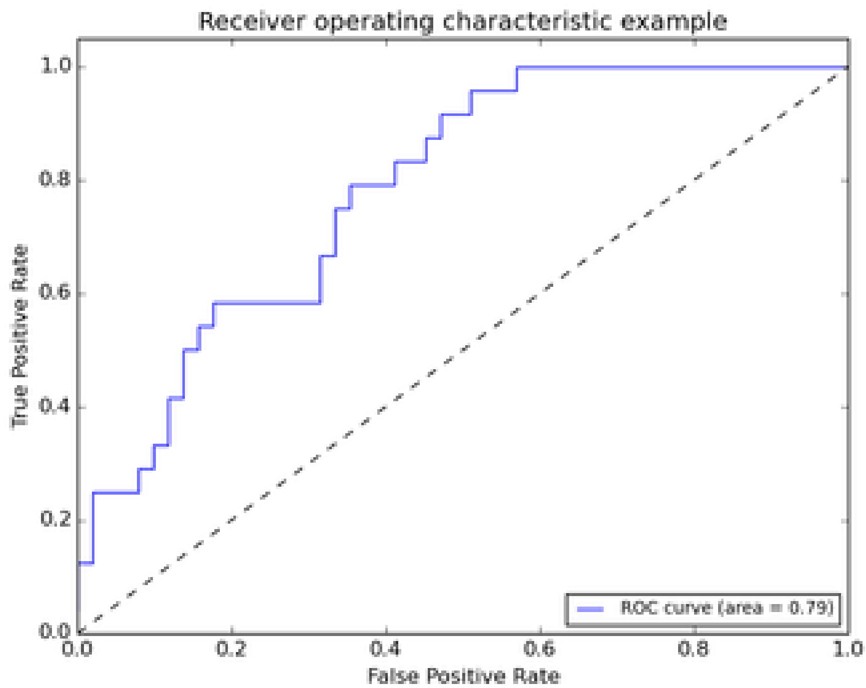


Figure 42.78: *Receiver Operating Characteristic (ROC)*

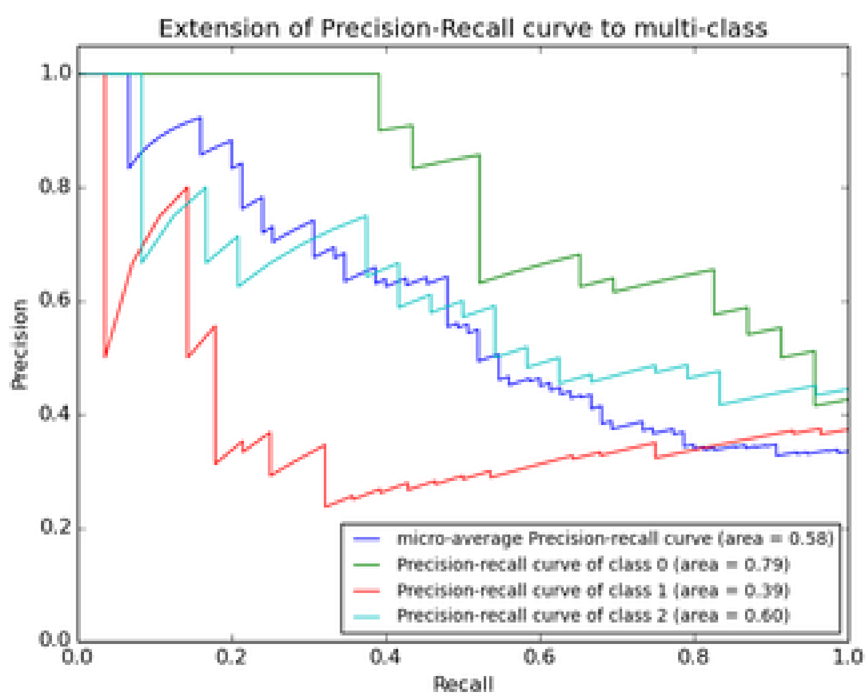


Figure 42.79: *Precision-Recall*

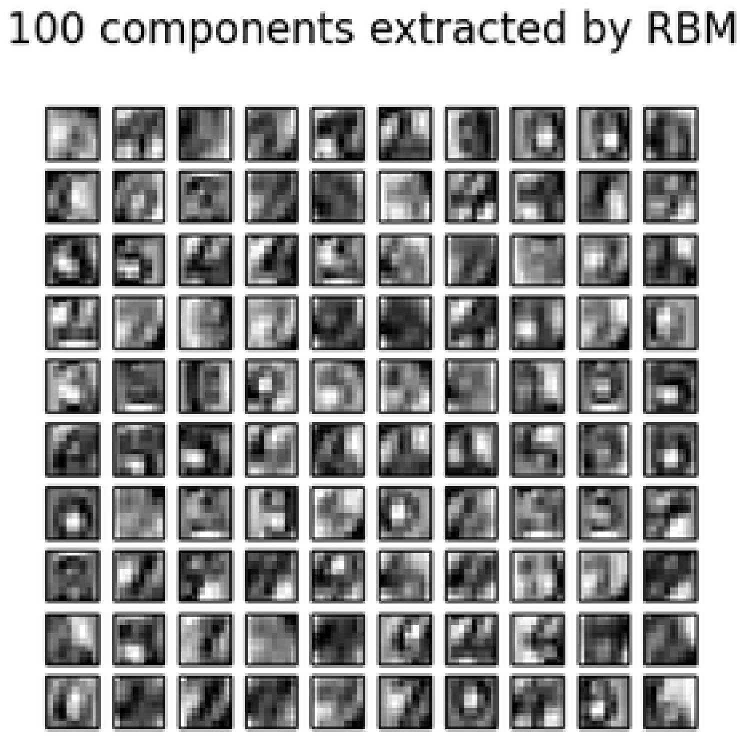


Figure 42.80: *Restricted Boltzmann Machine features for digit classification*

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.

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)`.

cv : cross-validation generator or int, optional, default: None

A cross-validation generator to use. If int, determines the number of folds in StratifiedKFold if y is binary or multiclass and estimator is a classifier, or the number of folds in KFold otherwise. If None, it is equivalent to cv=3.

n_jobs : integer, optional

The number of CPUs to use to do the computation. -1 means 'all CPUs'.

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'

Returnsscores : array of float, shape=(len(list(cv)),)

Array of scores of the estimator for each run of the cross validation.

Examples using `sklearn.cross_validation.cross_val_score`

Figure 42.81: *Imputing missing values before building an estimator*

42.5.12 `sklearn.cross_validation.cross_val_predict`

`sklearn.cross_validation.`**`cross_val_predict`**(*estimator*, *X*, *y=None*, *cv=None*, *n_jobs=1*, *verbose=0*, *fit_params=None*, *pre_dispatch='2*n_jobs'*)

Generate cross-validated estimates for each input data point

Parametersestimator : 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.

cv : cross-validation generator or int, optional, default: None

A cross-validation generator to use. If int, determines the number of folds in StratifiedKFold if y is binary or multiclass and estimator is a classifier, or the number of folds in KFold otherwise. If None, it is equivalent to cv=3. This generator must include all elements in the test set exactly once. Otherwise, a ValueError is raised.

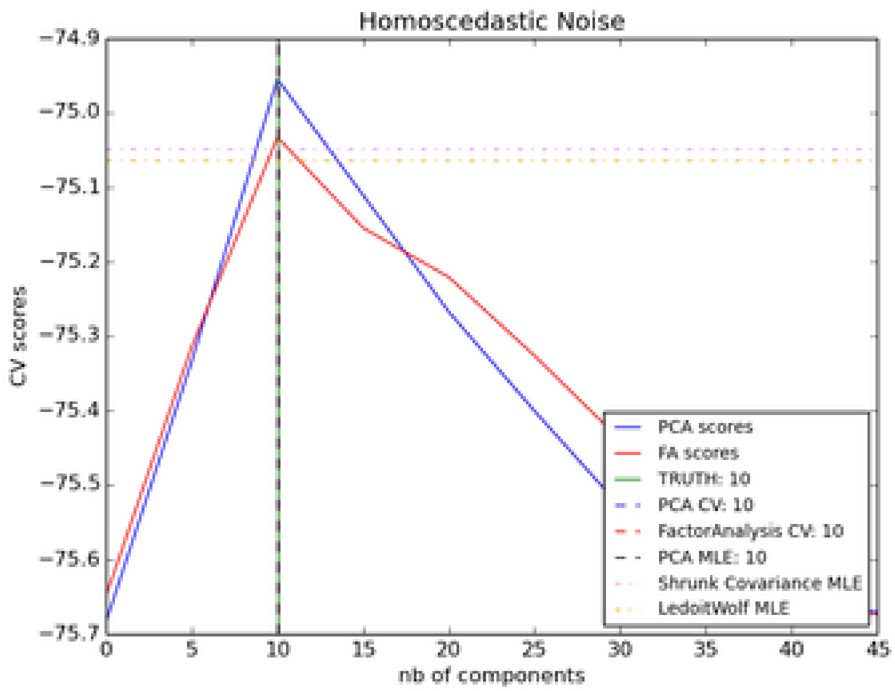


Figure 42.82: *Model selection with Probabilistic (PCA) and Factor Analysis (FA)*

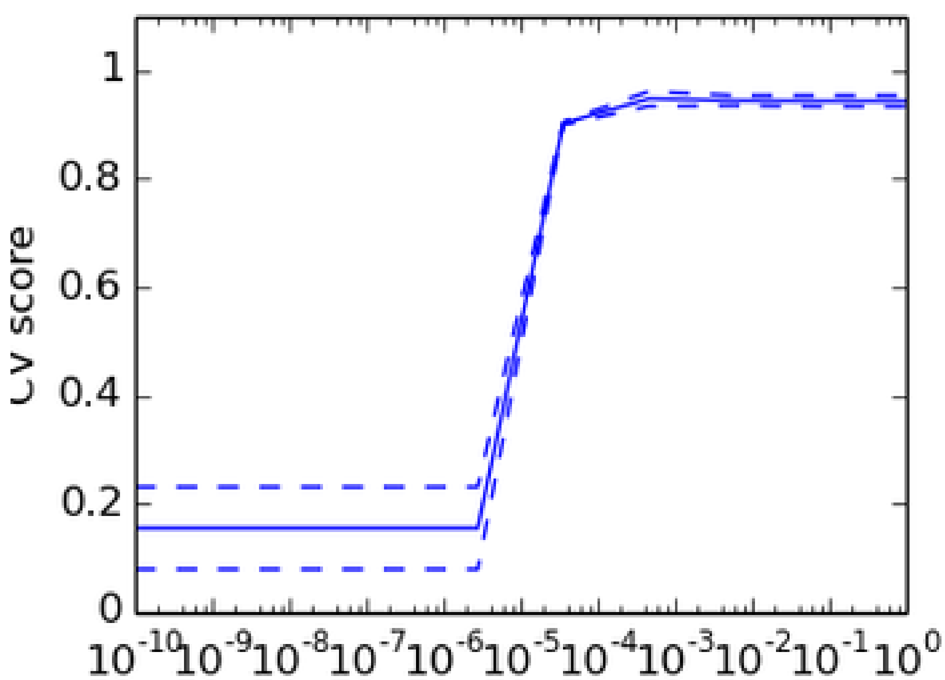


Figure 42.83: *Cross-validation on Digits Dataset Exercise*

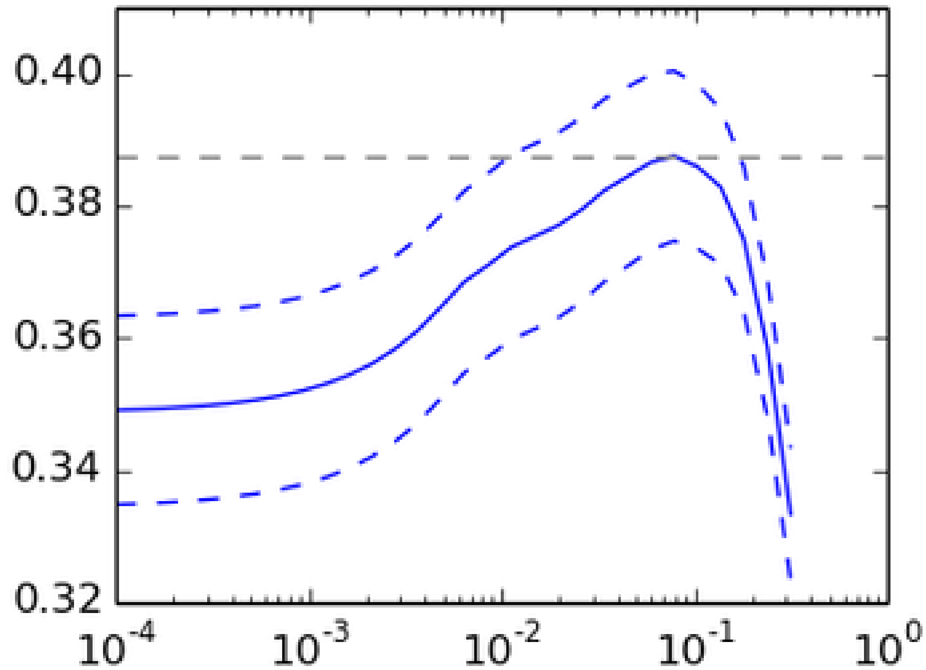


Figure 42.84: *Cross-validation on diabetes Dataset Exercise*

Figure 42.85: *Gaussian Processes regression: goodness-of-fit on the 'diabetes' dataset*

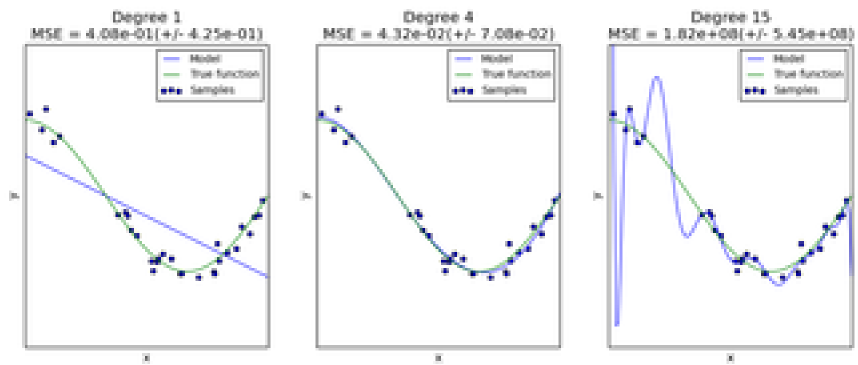


Figure 42.86: *Underfitting vs. Overfitting*

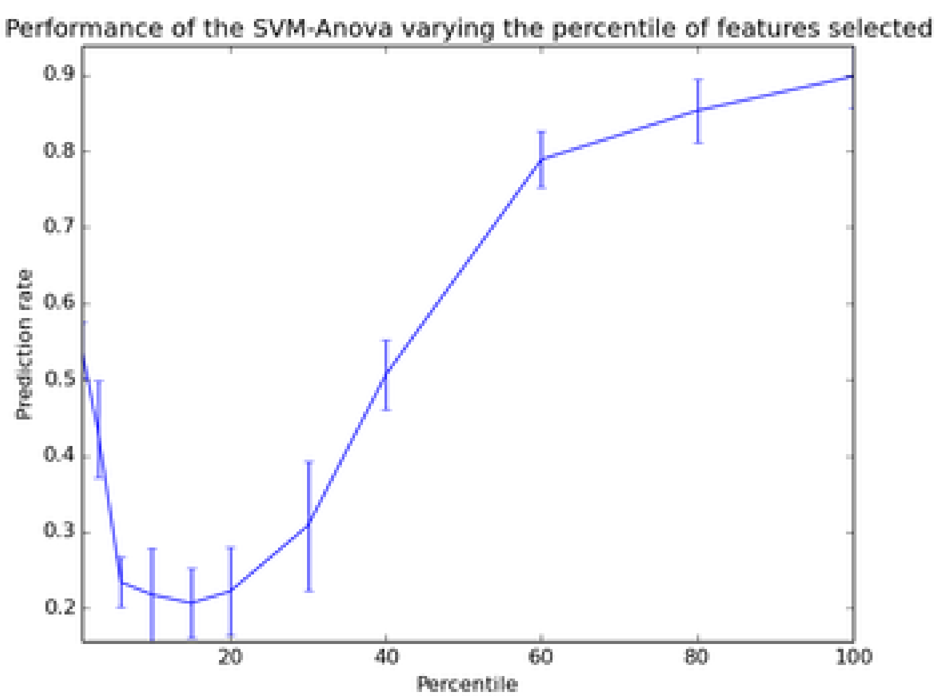


Figure 42.87: *SVM-Anova: SVM with univariate feature selection*

n_jobs : integer, optional

The number of CPUs to use to do the computation. -1 means 'all CPUs'.

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'

Returnspreds : ndarray

This is the result of calling 'predict'

Examples using `sklearn.cross_validation.cross_val_predict`

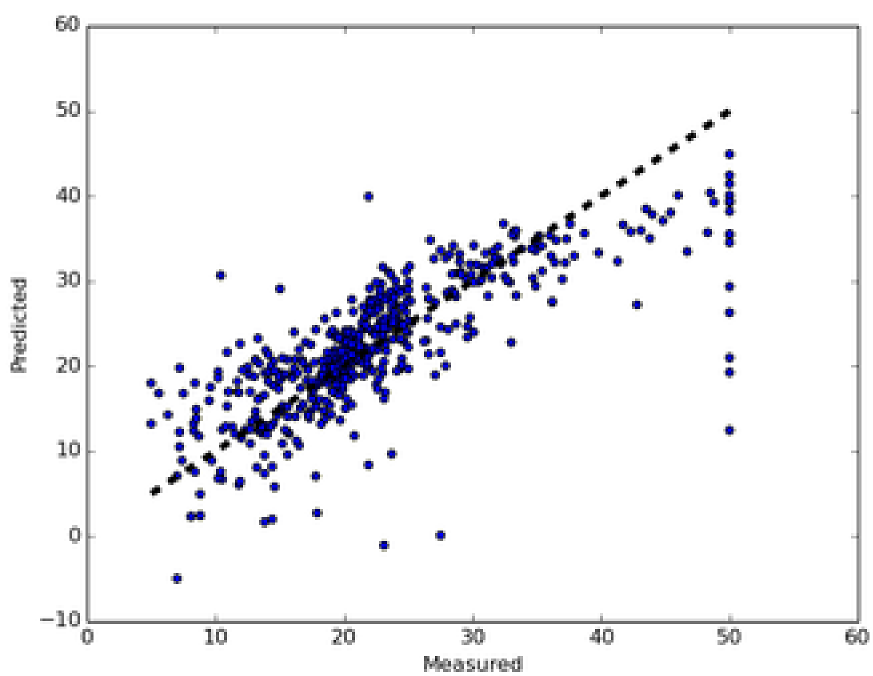


Figure 42.88: *Plotting Cross-Validated Predictions*

42.5.13 `sklearn.cross_validation`.permutation_test_score

`sklearn.cross_validation.`**`permutation_test_score`**(*estimator*, *X*, *y*, *cv=None*, *n_permutations=100*, *n_jobs=1*, *labels=None*, *random_state=0*, *verbose=0*, *scoring=None*)

Evaluate the significance of a cross-validated score with permutations

Parametersestimator : 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.

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)`.

cv : integer or cross-validation generator, optional

If an integer is passed, it is the number of fold (default 3). Specific cross-validation objects can be passed, see sklearn.cross_validation module for the list of possible objects.

n_permutations : integer, optional

Number of times to permute `y`.

n_jobs : integer, optional

The number of CPUs to use to do the computation. -1 means 'all CPUs'.

labels : array-like of shape [n_samples] (optional)

Labels constrain the permutation among groups of samples with a same label.

random_state : RandomState or an int seed (0 by default)

A random number generator instance to define the state of the random permutations generator.

verbose : integer, optional

The verbosity level.

Returnsscore : float

The true score without permuting targets.

permutation_scores : array, shape (n_permutations,)

The scores obtained for each permutations.

pvalue : float

The returned value equals p-value if *scoring* returns bigger numbers for better scores (e.g., accuracy_score). If *scoring* is rather a loss function (i.e. when lower is better such as with *mean_squared_error*) then this is actually the complement of the p-value: 1 - p-value.

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.cross_validation.permutation_test_score`

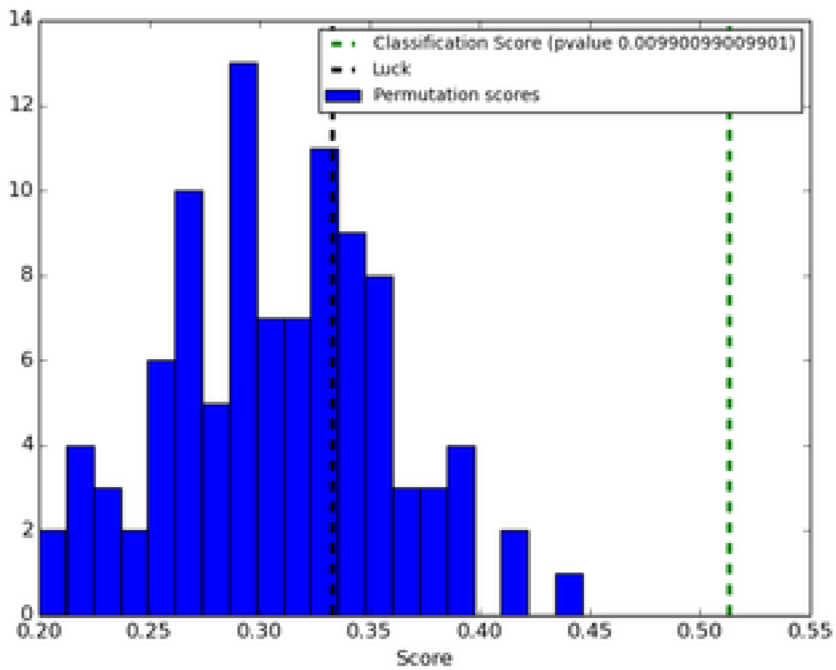


Figure 42.89: *Test with permutations the significance of a classification score*

42.5.14 `sklearn.cross_validation.check_cv`

`sklearn.cross_validation.`**`check_cv`**(*cv*, *X=None*, *y=None*, *classifier=False*)

Input checker utility for building a CV in a user friendly way.

Parameterscv : int, a cv generator instance, or None

> The input specifying which cv generator to use. It can be an integer, in which case it is the number of folds in a KFold, None, in which case 3 fold is used, or another object, that will then be used as a cv generator.

X : array-like

> The data the cross-val object will be applied on.

y : array-like

> The target variable for a supervised learning problem.

classifier : boolean optional

Whether the task is a classification task, in which case stratified KFold will be used.

Returnschecked_cv: a cross-validation generator instance. :

The return value is guaranteed to be a cv generator instance, whatever the input type.

42.6 `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.

42.6.1 Loaders

`datasets.clear_data_home`([data_home])	Delete all the content of the data home cache.
`datasets.get_data_home`([data_home])	Return the path of the scikit-learn data dir.
`datasets.fetch_20newsgroups`([data_home, ...])	Load the filenames and data from the 20 newsgroups dataset.
`datasets.fetch_20newsgroups_vectorized`([...])	Load the 20 newsgroups dataset and transform it into tf-idf vectors.
`datasets.load_boston`()	Load and return the boston house-prices dataset (regression).
`datasets.load_diabetes`()	Load and return the diabetes dataset (regression).
`datasets.load_digits`([n_class])	Load and return the digits dataset (classification).
`datasets.load_files`(container_path[, ...])	Load text files with categories as subfolder names.
`datasets.load_iris`()	Load and return the iris dataset (classification).
`datasets.load_lfw_pairs`([download_if_missing])	Alias for fetch_lfw_pairs(download_if_missing=False)
`datasets.fetch_lfw_pairs`([subset, ...])	Loader for the Labeled Faces in the Wild (LFW) pairs dataset
`datasets.load_lfw_people`([download_if_missing])	Alias for fetch_lfw_people(download_if_missing=False)
`datasets.fetch_lfw_people`([data_home, ...])	Loader for the Labeled Faces in the Wild (LFW) people dataset
`datasets.load_linnerud`()	Load and return the linnerud dataset (multivariate regression).
`datasets.mldata_filename`(dataname)	Convert a raw name for a data set in a mldata.org filename.
`datasets.fetch_mldata`(dataname[, ...])	Fetch an mldata.org data set
`datasets.fetch_olivetti_faces`([data_home, ...])	Loader for the Olivetti faces data-set from AT&T.
`datasets.fetch_california_housing`([...])	Loader for the California housing dataset from StatLib.
`datasets.fetch_covtype`([data_home, ...])	Load the covertype dataset, downloading it if necessary.
`datasets.load_mlcomp`(name_or_id[, **set_**, ...])	Load a datasets as downloaded from http://mlcomp.org
`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.dump_svmlight_file`(X, y, f[, ...])	Dump the dataset in svmlight / libsvm file format.

`sklearn.datasets`.clear_data_home

`sklearn.datasets.`**`clear_data_home`**(*data_home=None*)

Delete all the content of the data home cache.

`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.

Examples using `sklearn.datasets.get_data_home`

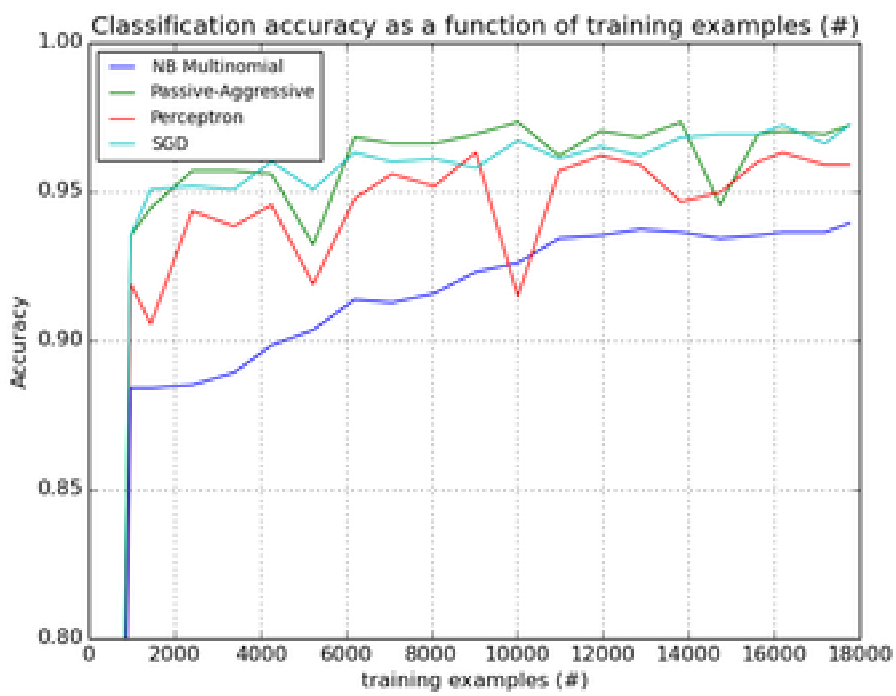


Figure 42.90: *Out-of-core classification of text documents*

`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*)

Load the filenames and data from the 20 newsgroups dataset.

Parameterssubset: '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.

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.

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: numpy random number generator or seed integer :

Used to shuffle the dataset.

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.

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.

Examples using `sklearn.datasets.fetch_20newsgroups`

Figure 42.91: *Feature Union with Heterogeneous Data Sources*

`sklearn.datasets`.fetch_20newsgroups_vectorized

`sklearn.datasets.`**`fetch_20newsgroups_vectorized`**(*subset='train'*, *remove=()*, *data_home=None*)

Load the 20 newsgroups dataset and transform it into tf-idf vectors.

This is a convenience function; the tf-idf transformation is done using the default settings for *sklearn.feature_extraction.text.Vectorizer*. For more advanced usage (stopword filtering, n-gram extraction, etc.), combine fetch_20newsgroups with a custom *Vectorizer* or *CountVectorizer*.

Parameterssubset: 'train' or 'test', 'all', optional :

Figure 42.92: *Topics extraction with Non-Negative Matrix Factorization*

Figure 42.93: *Biclustering documents with the Spectral Co-clustering algorithm*

Figure 42.94: *Sample pipeline for text feature extraction and evaluation*

Figure 42.95: *FeatureHasher and DictVectorizer Comparison*

Figure 42.96: *Clustering text documents using k-means*

Figure 42.97: *Classification of text documents using sparse features*

Select the dataset to load: 'train' for the training set, 'test' for the test set, 'all' for both, with shuffled ordering.

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.

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.

Returnsbunch : Bunch object

bunch.data: sparse matrix, shape [n_samples, n_features] bunch.target: array, shape [n_samples] bunch.target_names: list, length [n_classes]

Examples using `sklearn.datasets.fetch_20newsgroups_vectorized`

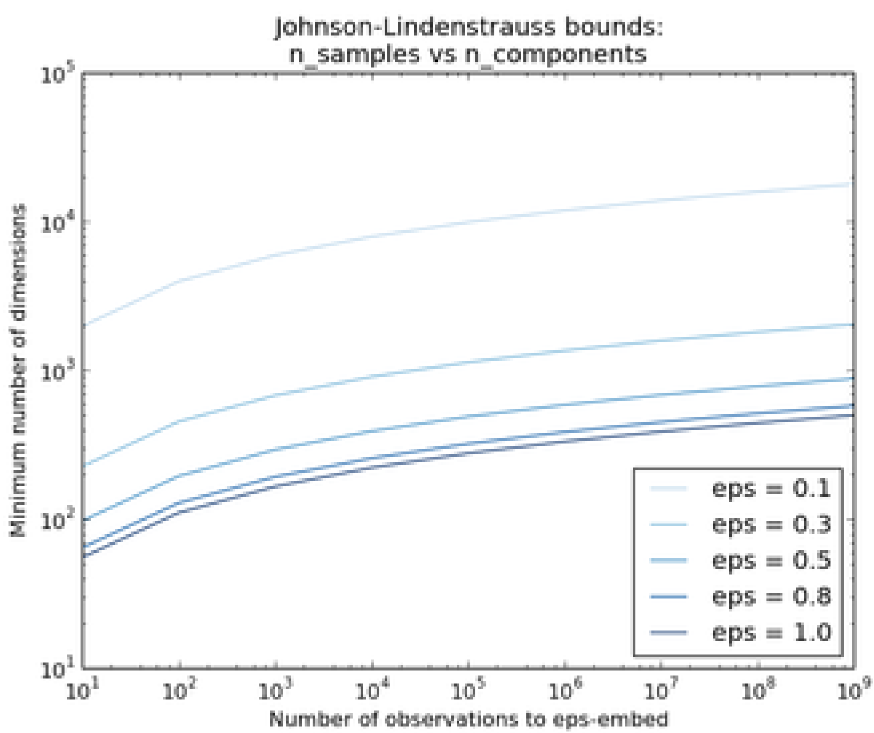


Figure 42.98: *The Johnson-Lindenstrauss bound for embedding with random projections*

`sklearn.datasets`.load_boston

`sklearn.datasets.load_boston()`

Load and return the boston house-prices dataset (regression).

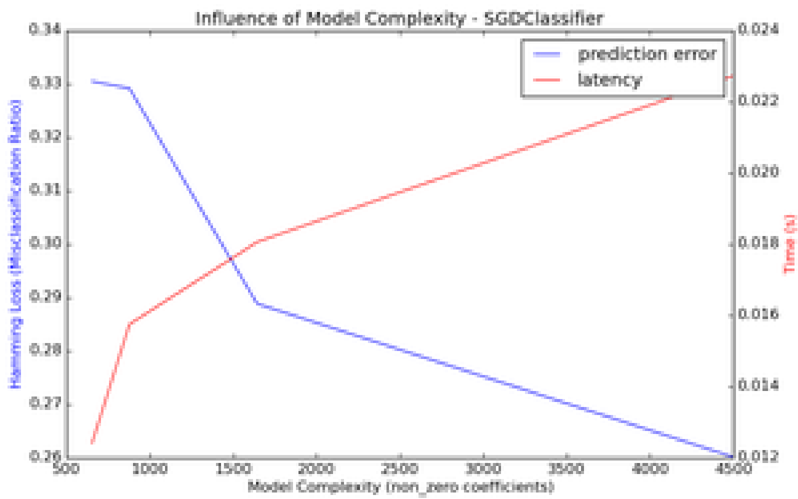


Figure 42.99: *Model Complexity Influence*

Samples total	506
Dimensionality	13
Features	real, positive
Targets	real 5. - 50.

Returnsdata : Bunch

Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the regression targets, and 'DESCR', the full description of the dataset.

Examples

```
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()
>>> print(boston.data.shape)
(506, 13)
```

Examples using `sklearn.datasets.load_boston`

`sklearn.datasets`.load_diabetes

`sklearn.datasets.load_diabetes()`

Load and return the diabetes dataset (regression).

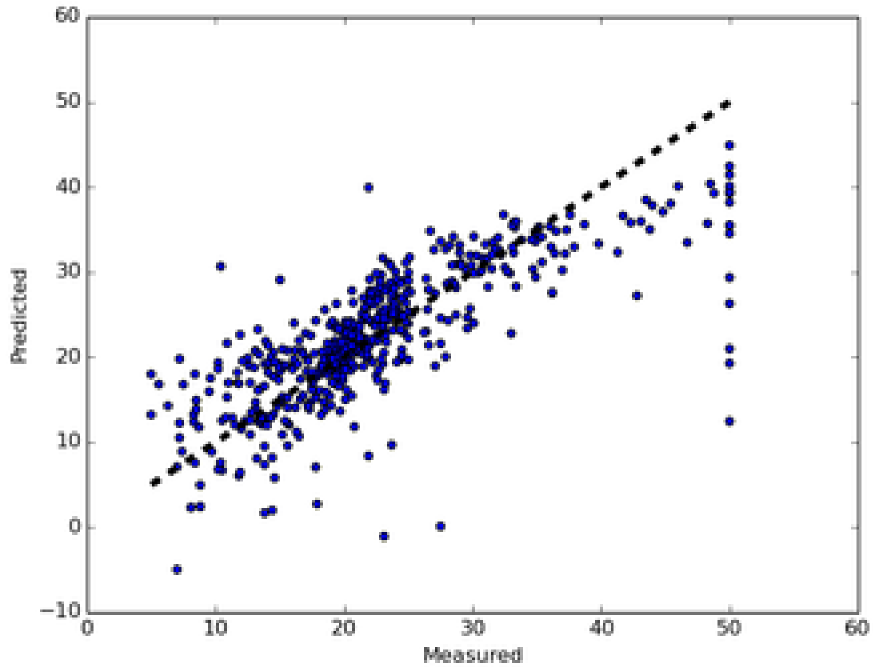


Figure 42.100: *Plotting Cross-Validated Predictions*

Figure 42.101: *Imputing missing values before building an estimator*

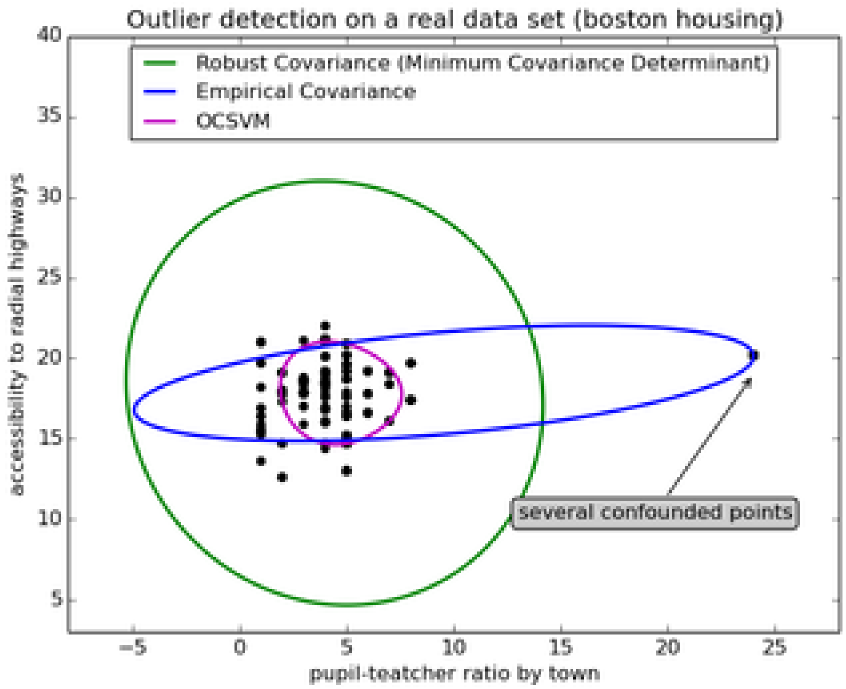


Figure 42.102: *Outlier detection on a real data set*

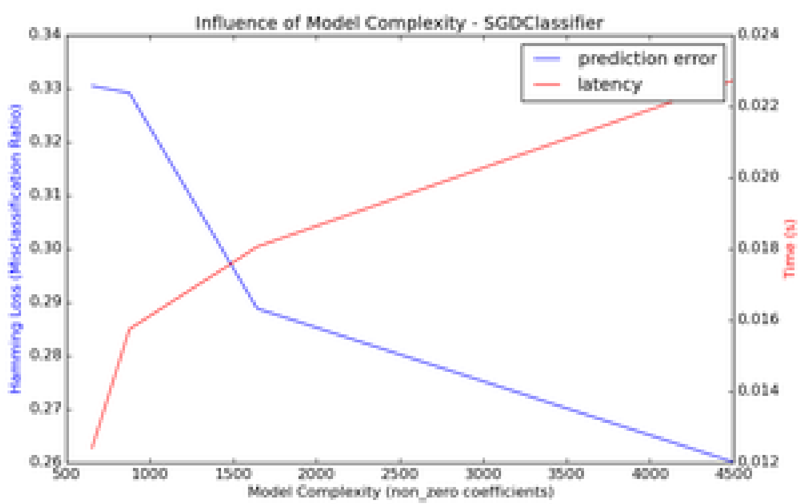


Figure 42.103: *Model Complexity Influence*

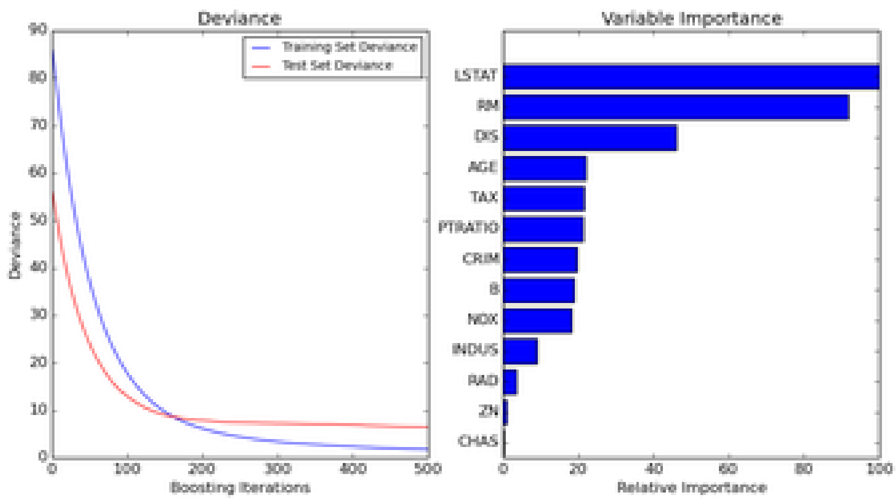


Figure 42.104: *Gradient Boosting regression*

Samples total	442
Dimensionality	10
Features	real, -.2 < x < .2
Targets	integer 25 - 346

Returnsdata : Bunch

Dictionary-like object, the interesting attributes are: 'data', the data to learn and 'target', the regression target for each sample.

Examples using `sklearn.datasets.load_diabetes`

`sklearn.datasets`.load_digits

`sklearn.datasets.`**`load_digits`**(*n_class=10*)

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

Parametersn_class : integer, between 0 and 10, optional (default=10)

The number of classes to return.

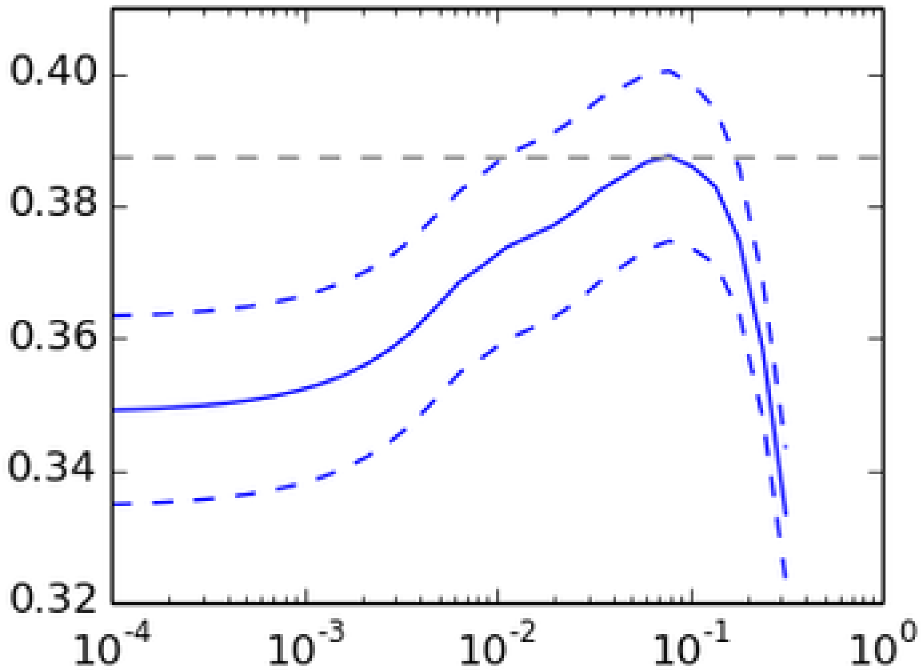


Figure 42.105: *Cross-validation on diabetes Dataset Exercise*

Figure 42.106: *Gaussian Processes regression: goodness-of-fit on the 'diabetes' dataset*

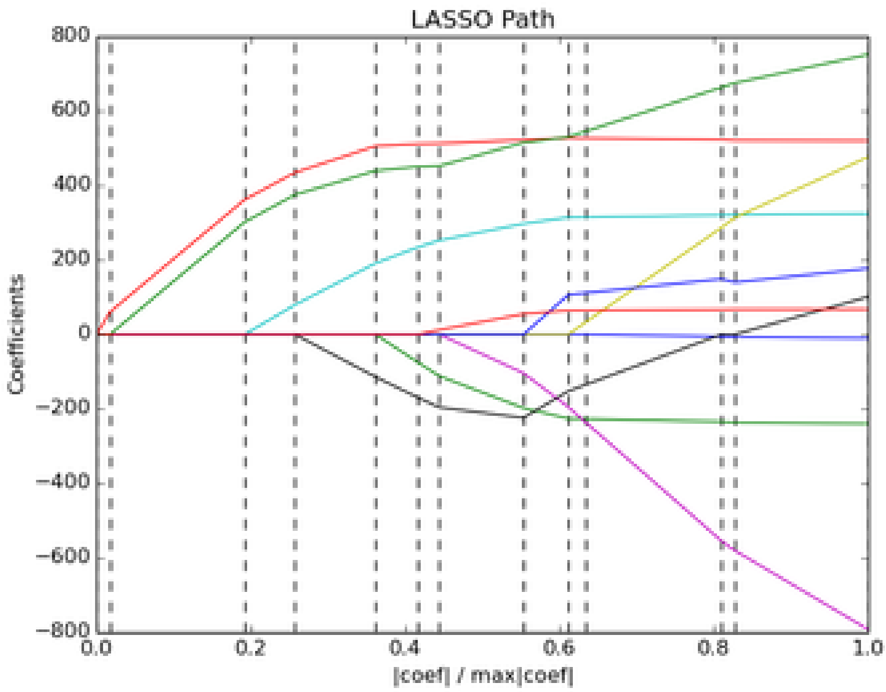


Figure 42.107: *Lasso path using LARS*

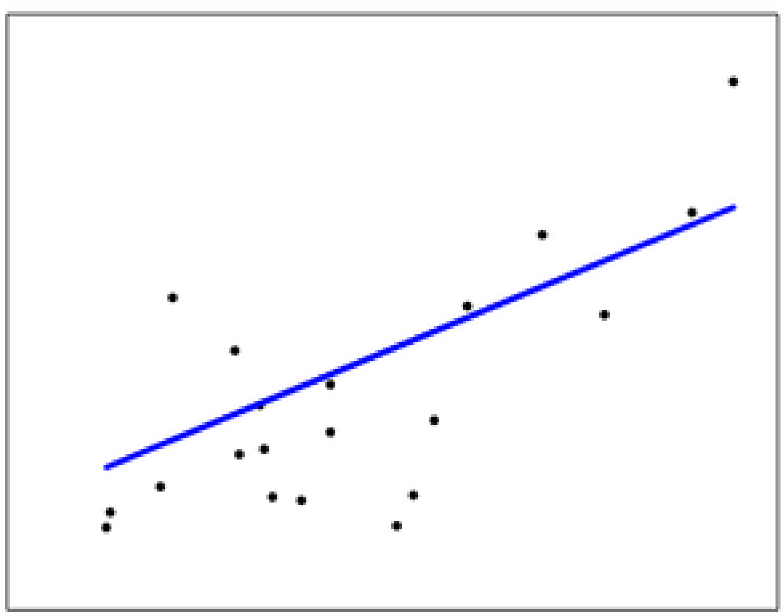


Figure 42.108: *Linear Regression Example*

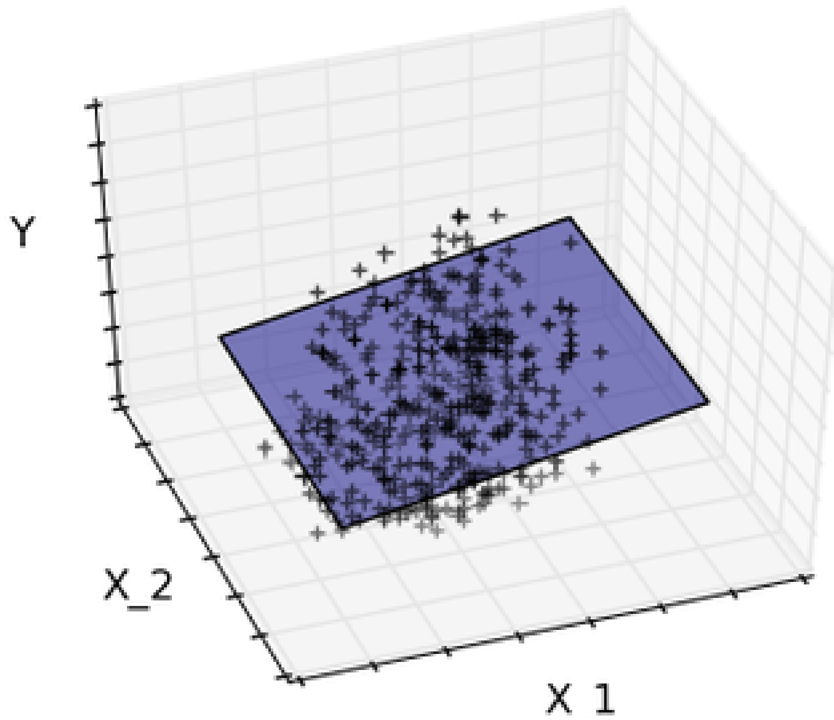


Figure 42.109: *Sparsity Example: Fitting only features 1 and 2*

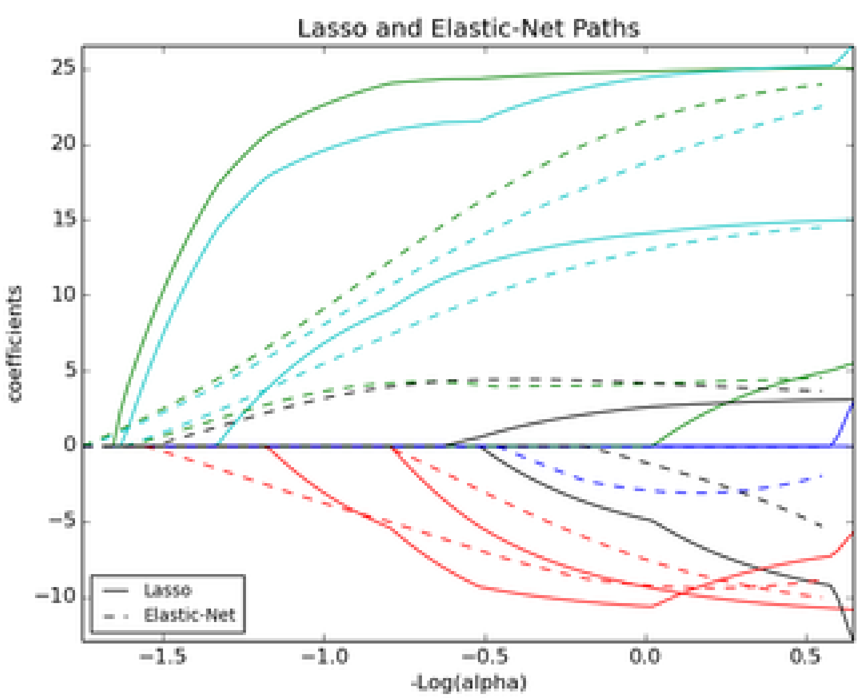


Figure 42.110: *Lasso and Elastic Net*

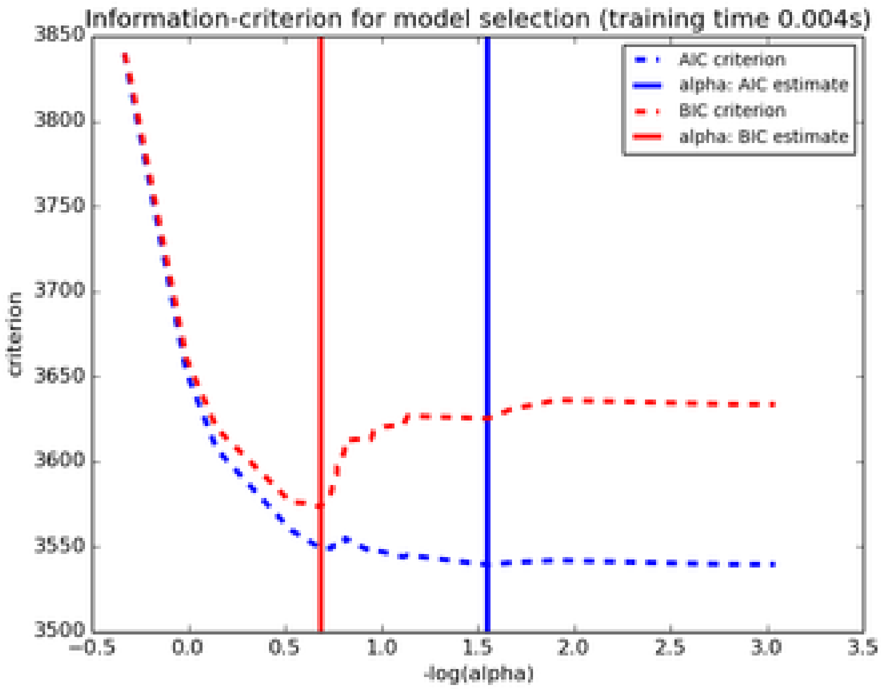


Figure 42.111: *Lasso model selection: Cross-Validation / AIC / BIC*

Returnsdata : 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.

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 pylab as pl
>>> pl.gray()
>>> pl.matshow(digits.images[0])
>>> pl.show()
```

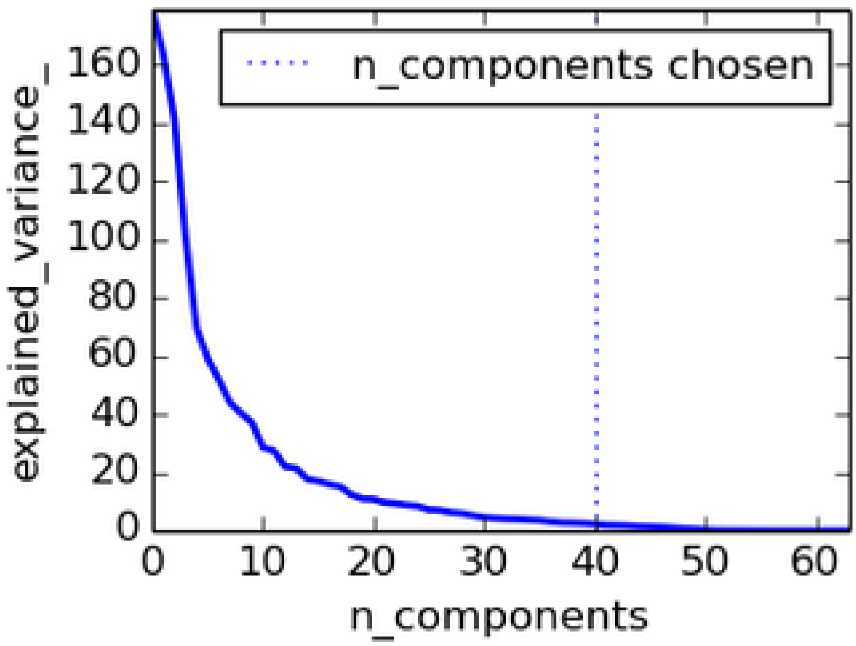


Figure 42.112: *Pipelining: chaining a PCA and a logistic regression*

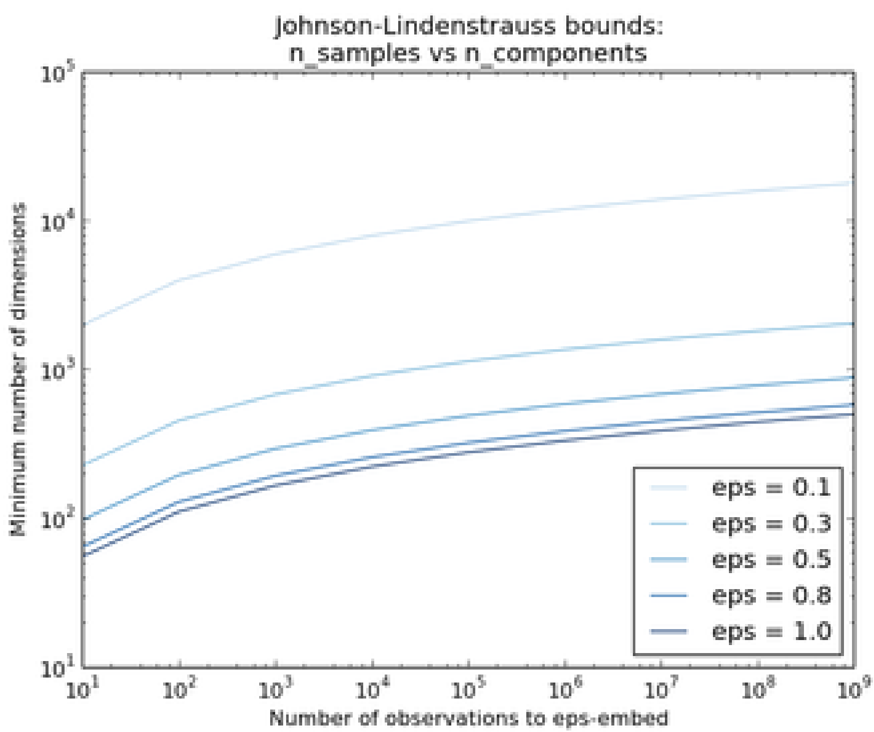


Figure 42.113: *The Johnson-Lindenstrauss bound for embedding with random projections*

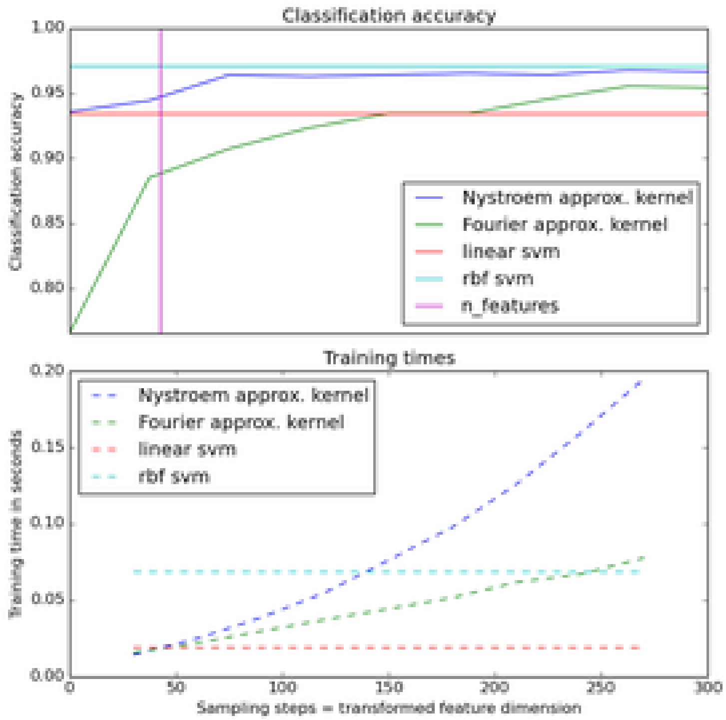


Figure 42.114: *Explicit feature map approximation for RBF kernels*

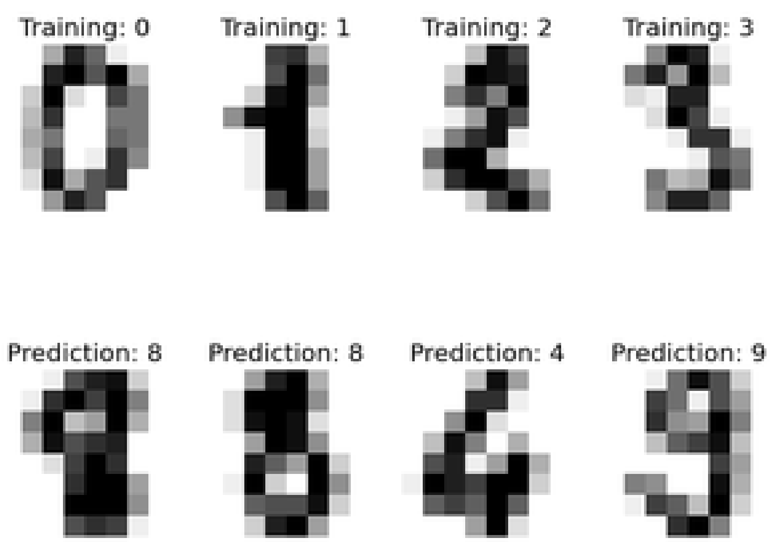


Figure 42.115: *Recognizing hand-written digits*

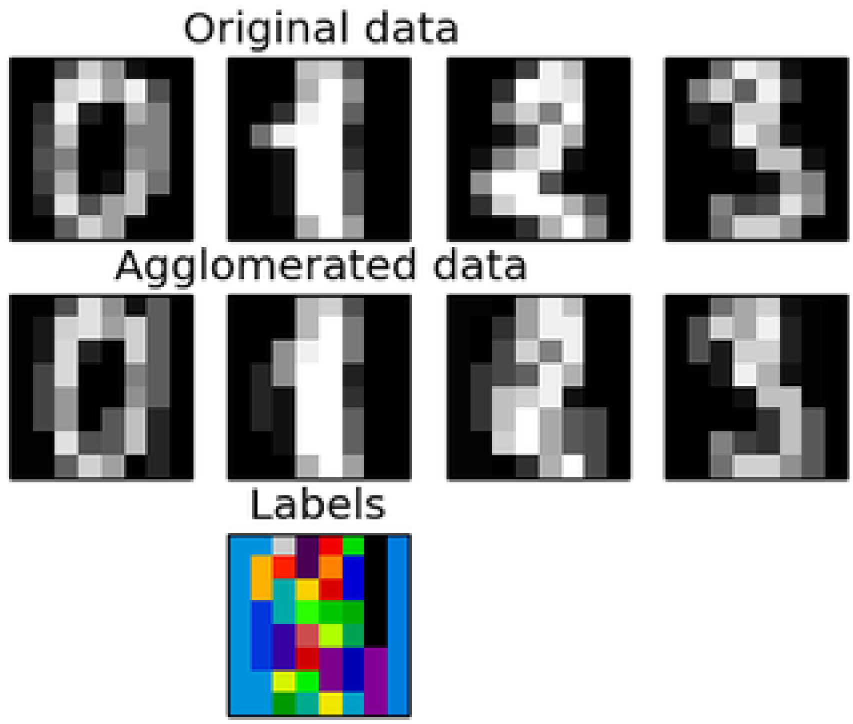


Figure 42.116: *Feature agglomeration*

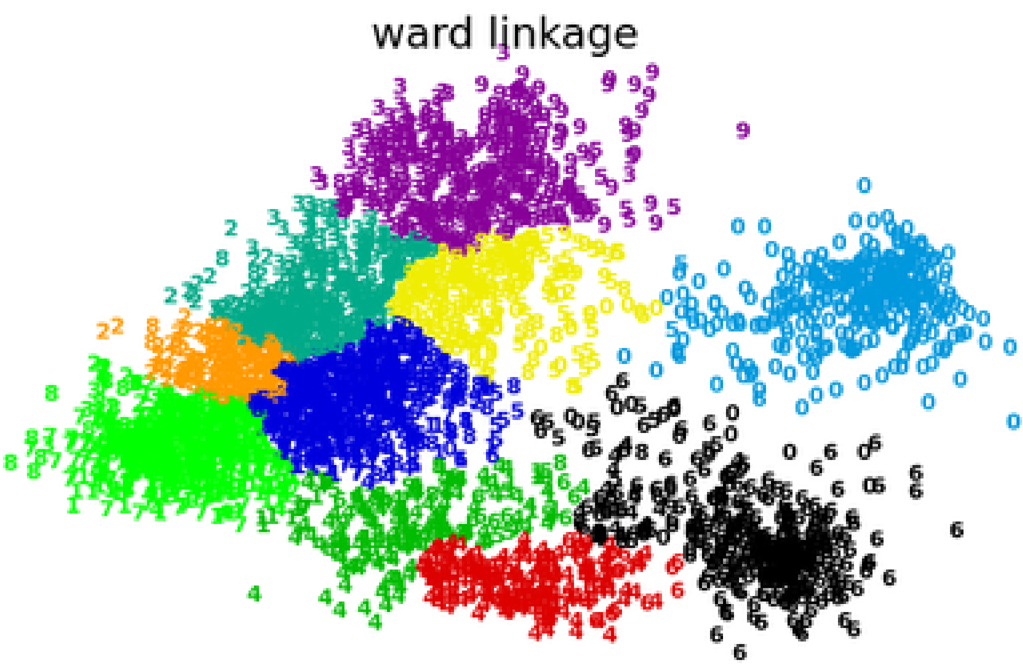


Figure 42.117: *Various Agglomerative Clustering on a 2D embedding of digits*

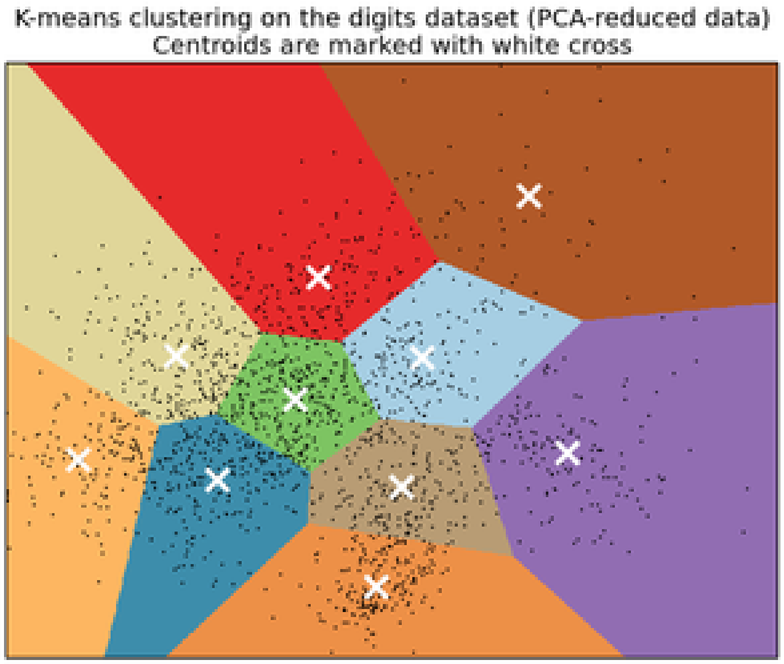


Figure 42.118: *A demo of K-Means clustering on the handwritten digits data*

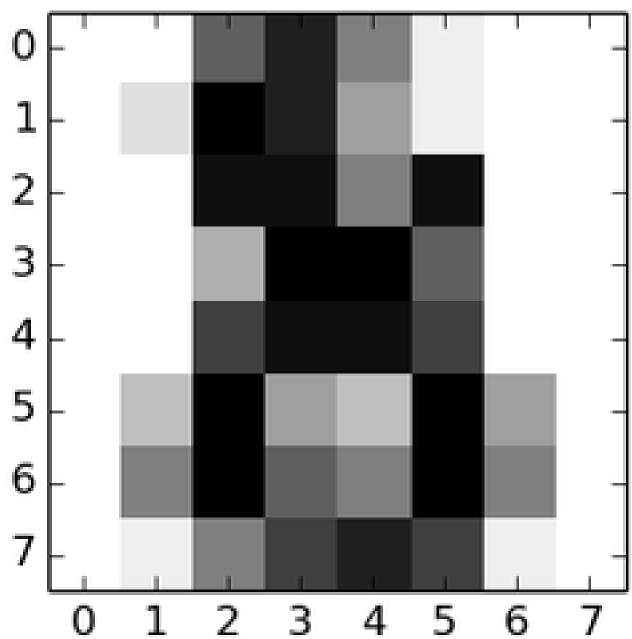


Figure 42.119: *The Digit Dataset*

Figure 42.120: *Digits Classification Exercise*

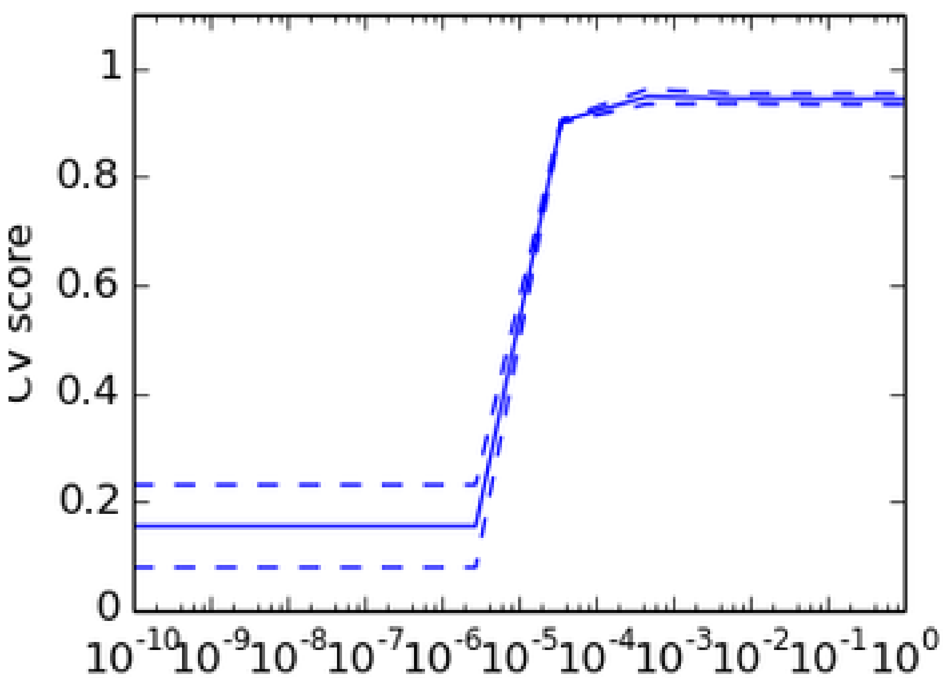


Figure 42.121: *Cross-validation on Digits Dataset Exercise*

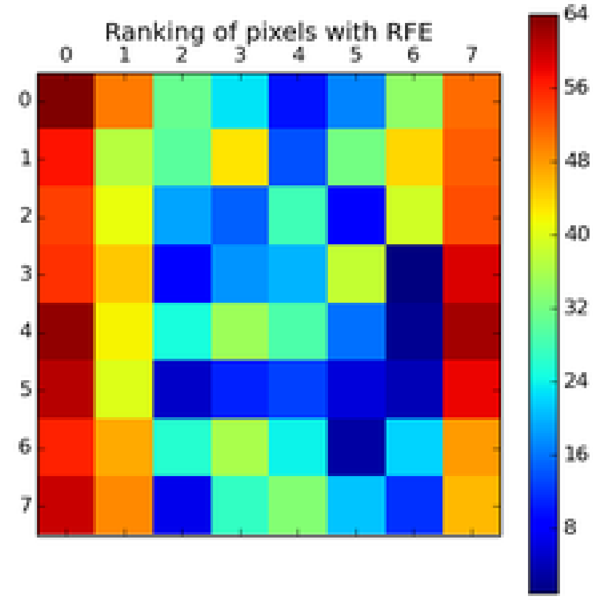


Figure 42.122: *Recursive feature elimination*

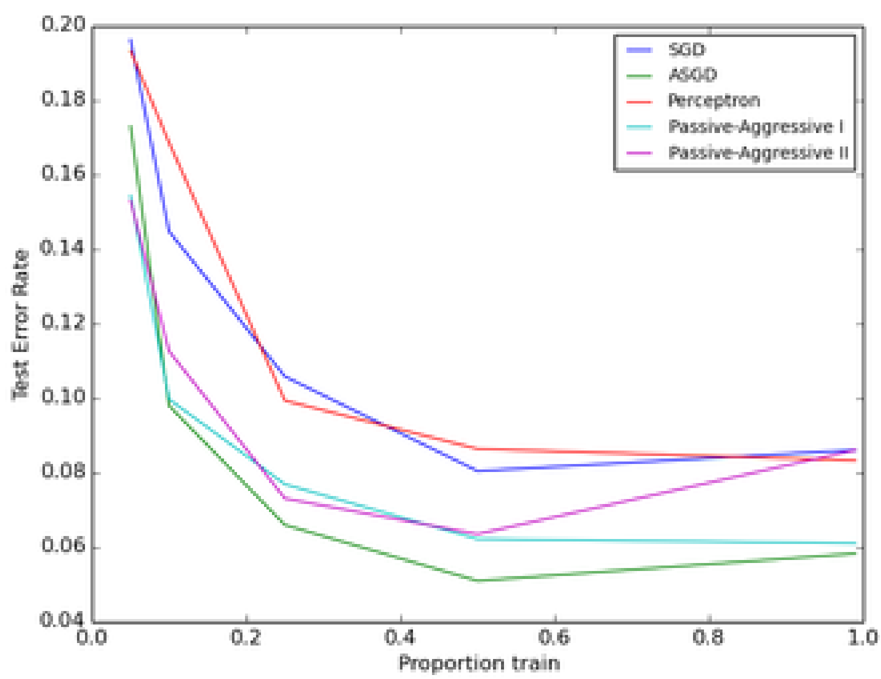


Figure 42.123: *Comparing various online solvers*

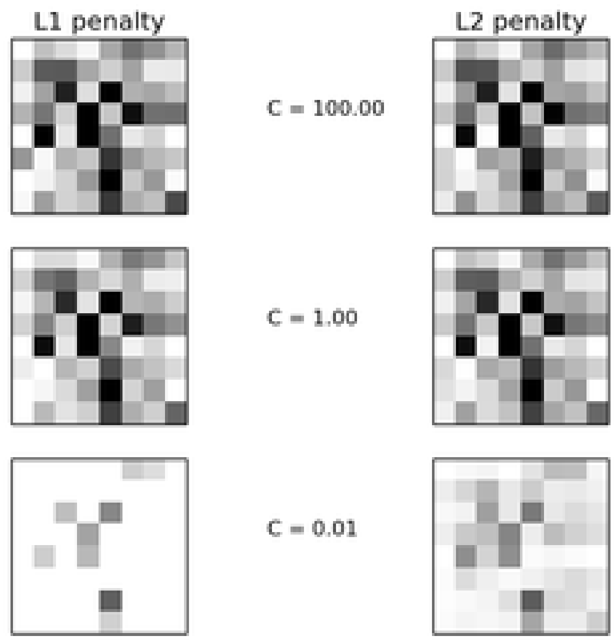


Figure 42.124: *L1 Penalty and Sparsity in Logistic Regression*

Figure 42.125: *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

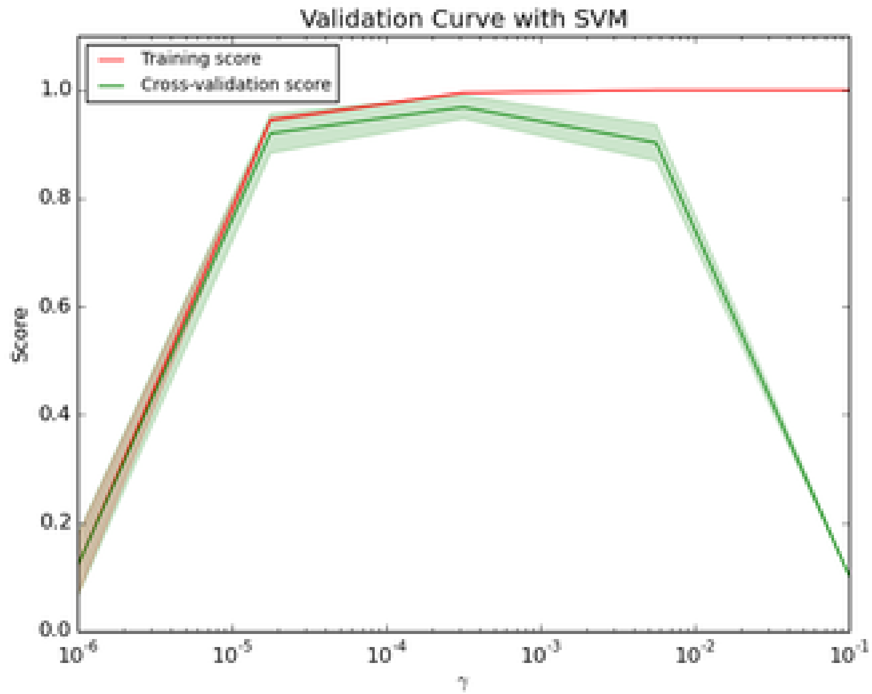


Figure 42.126: *Plotting Validation Curves*

Figure 42.127: *Parameter estimation using grid search with cross-validation*

Figure 42.128: *Comparing randomized search and grid search for hyperparameter estimation*

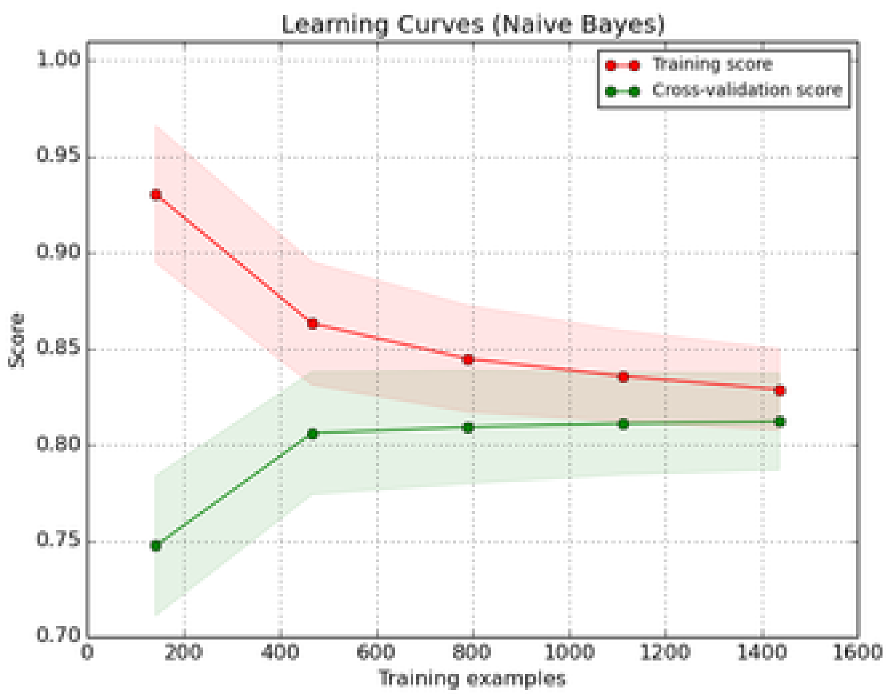


Figure 42.129: *Plotting Learning Curves*

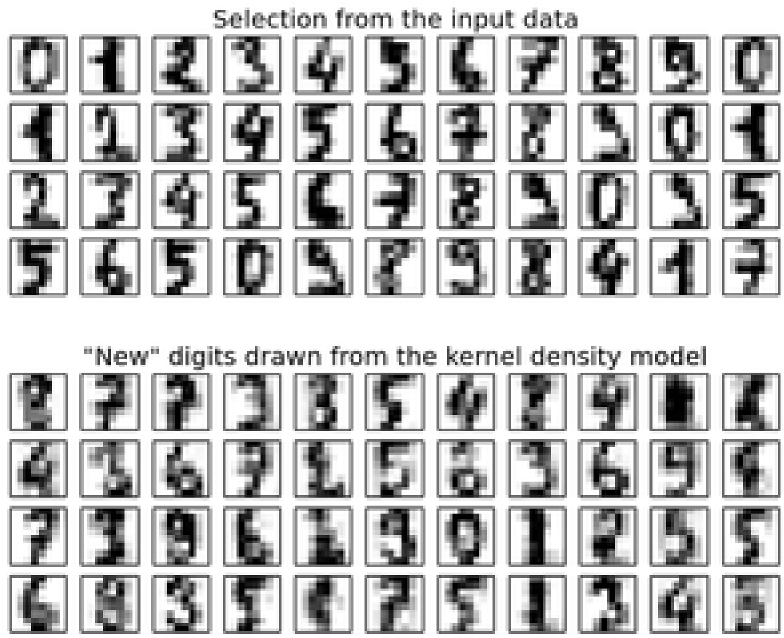


Figure 42.130: *Kernel Density Estimation*

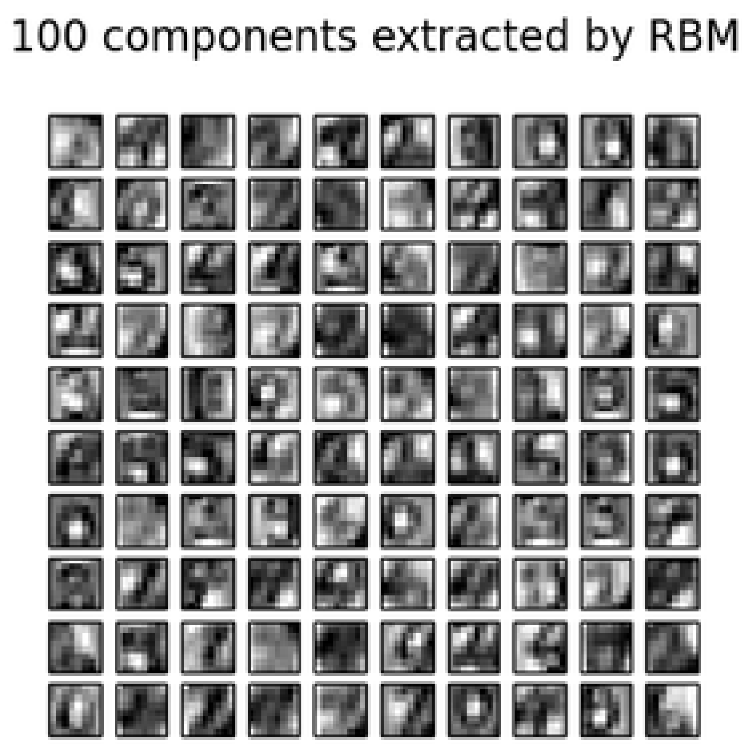


Figure 42.131: *Restricted Boltzmann Machine features for digit classification*

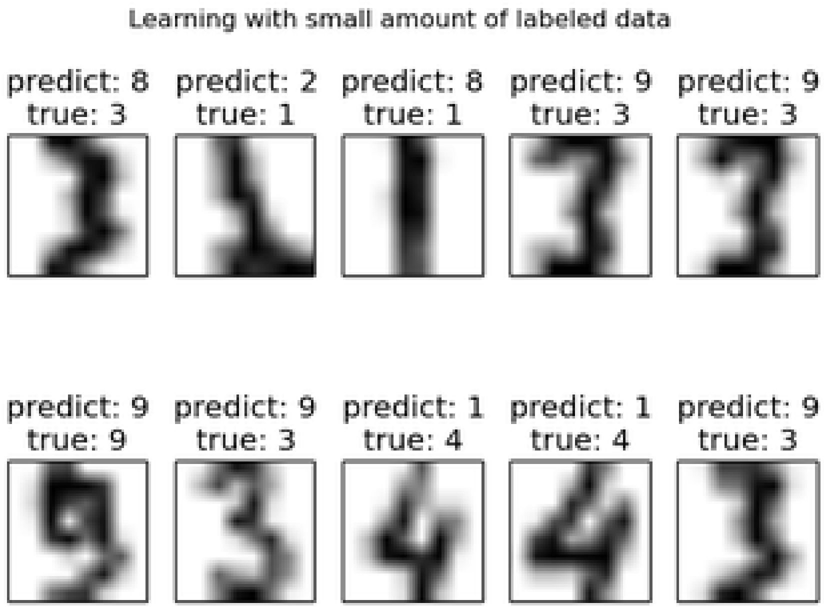


Figure 42.132: *Label Propagation digits: Demonstrating performance*

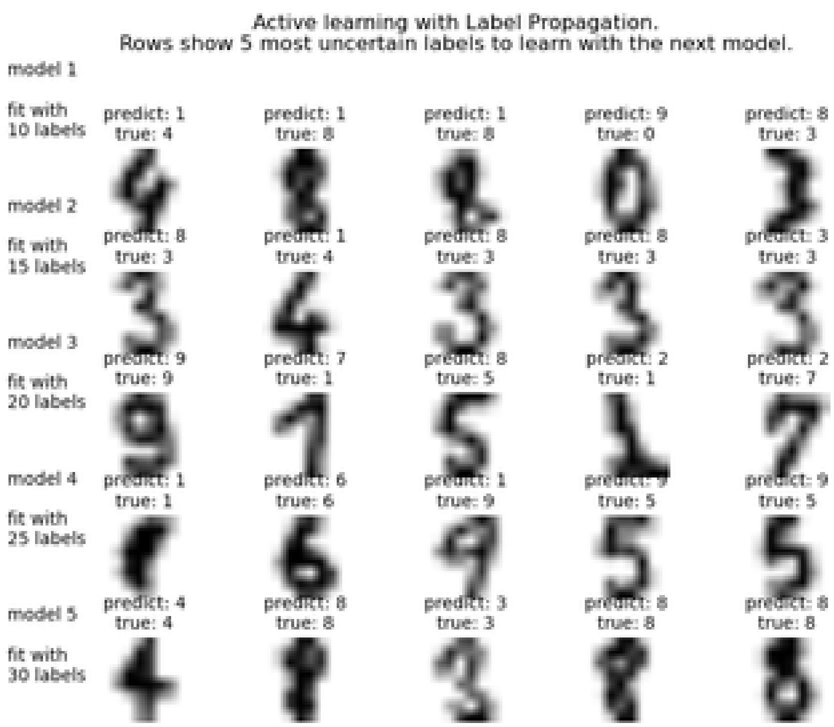


Figure 42.133: *Label Propagation digits active learning*

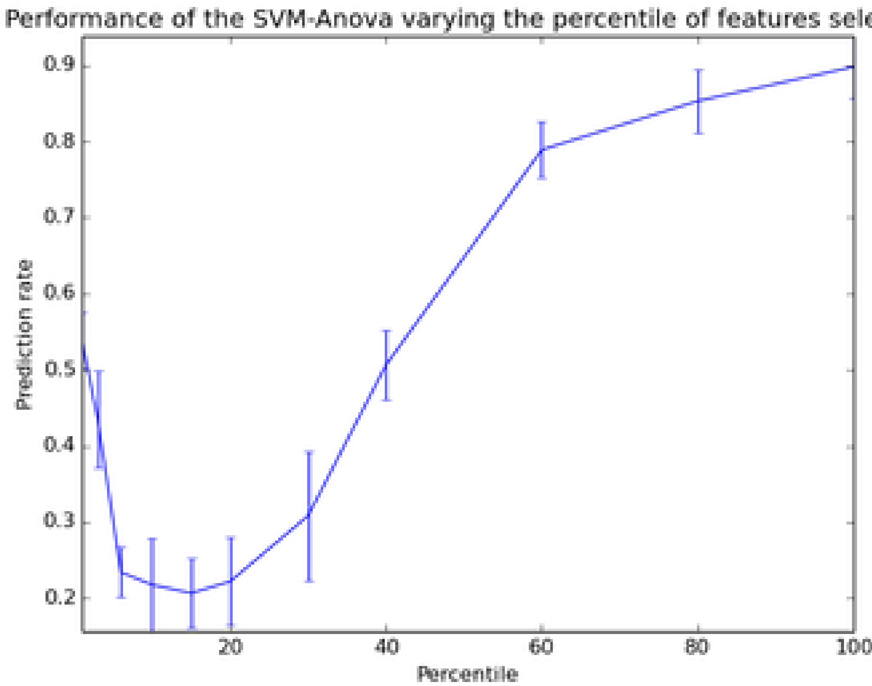


Figure 42.134: *SVM-Anova: SVM with univariate feature selection*

Examples using `sklearn.datasets.load_digits`

`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 *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 *sklearn.feature_extraction.text*.

Similar feature extractors should be built for other kind of unstructured data input such as images, audio, video, ...

Parameterscontainer_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.

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.

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.

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*.

Returnsdata : 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`**`()`

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

Returnsdata : 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.

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

Figure 42.135: *Concatenating multiple feature extraction methods*

sklearn.datasets.load_lfw_pairs

sklearn.datasets.**load_lfw_pairs**(*download_if_missing=False*, ***kwargs*)

Alias for fetch_lfw_pairs(download_if_missing=False)

Check fetch_lfw_pairs.__doc__ for the documentation and parameter list.

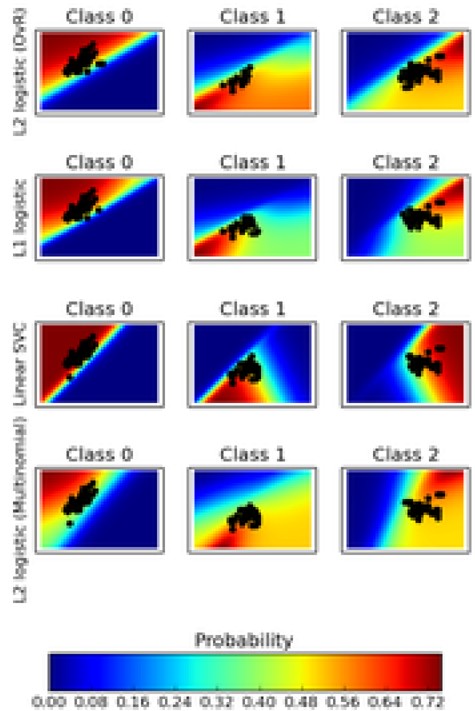


Figure 42.136: *Plot classification probability*

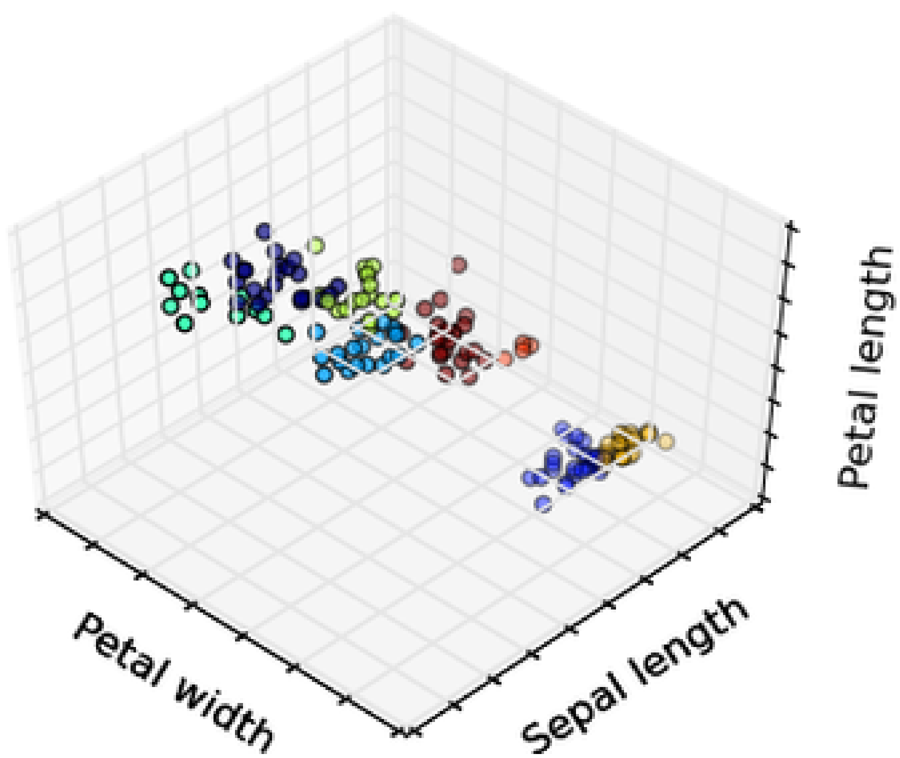


Figure 42.137: *K-means Clustering*

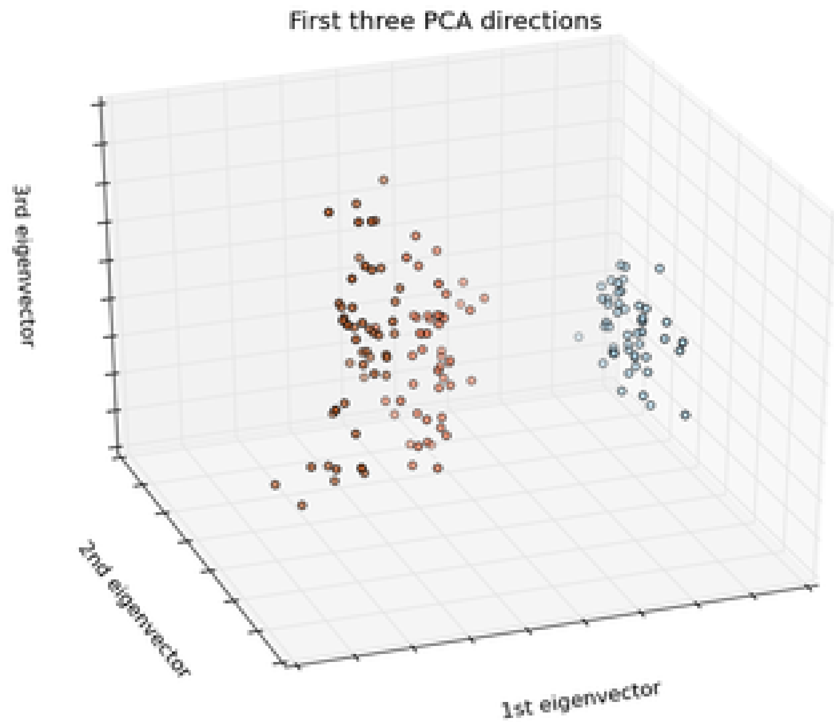


Figure 42.138: *The Iris Dataset*

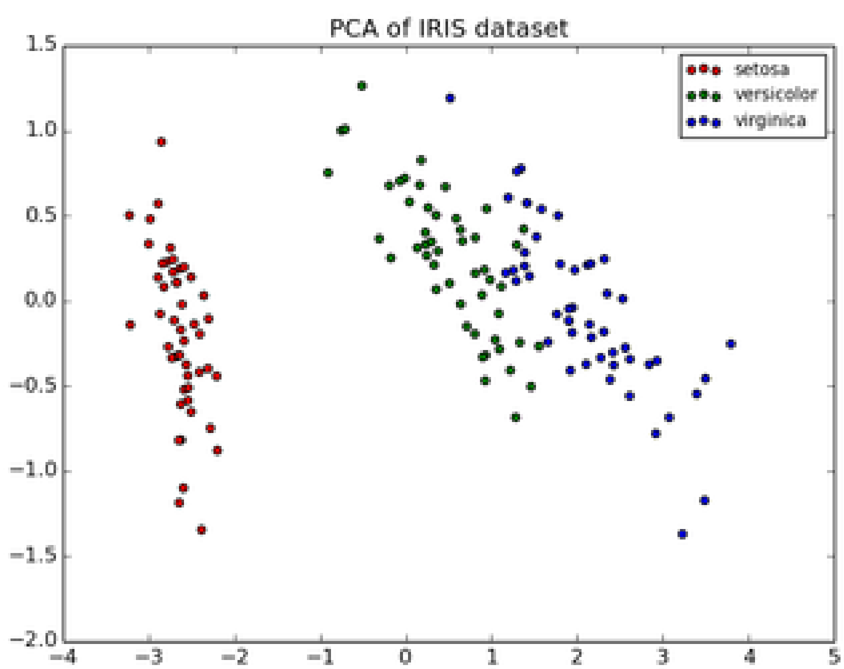


Figure 42.139: *Comparison of LDA and PCA 2D projection of Iris dataset*

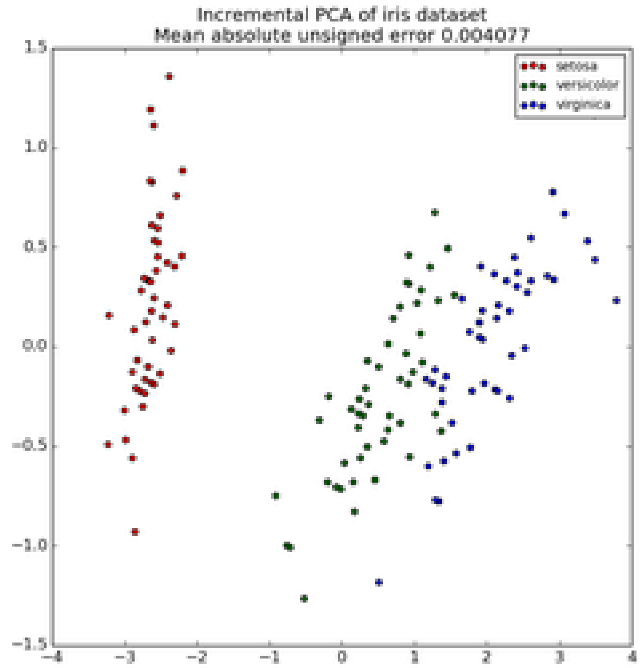


Figure 42.140: *Incremental PCA*

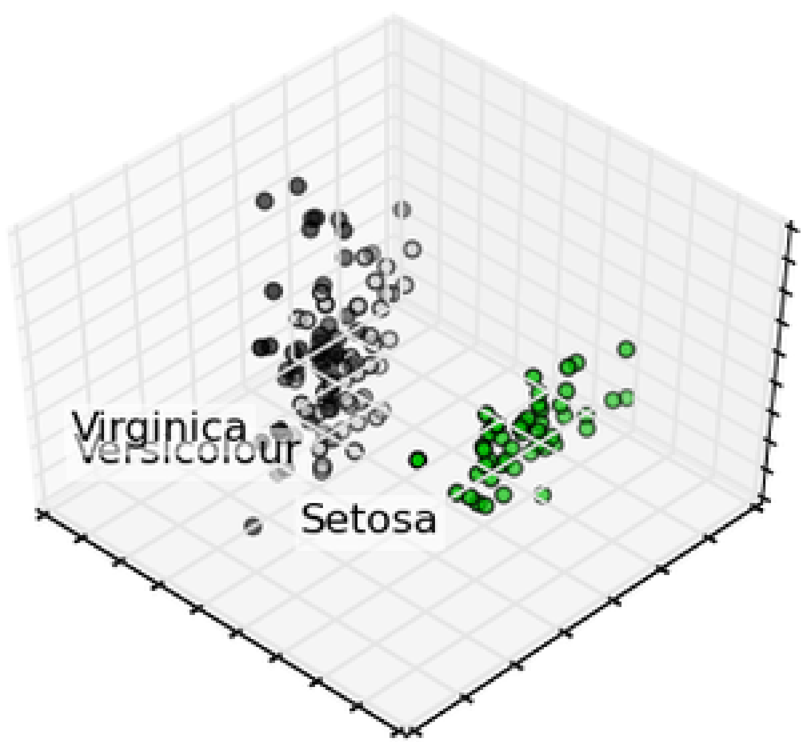


Figure 42.141: *PCA example with Iris Data-set*

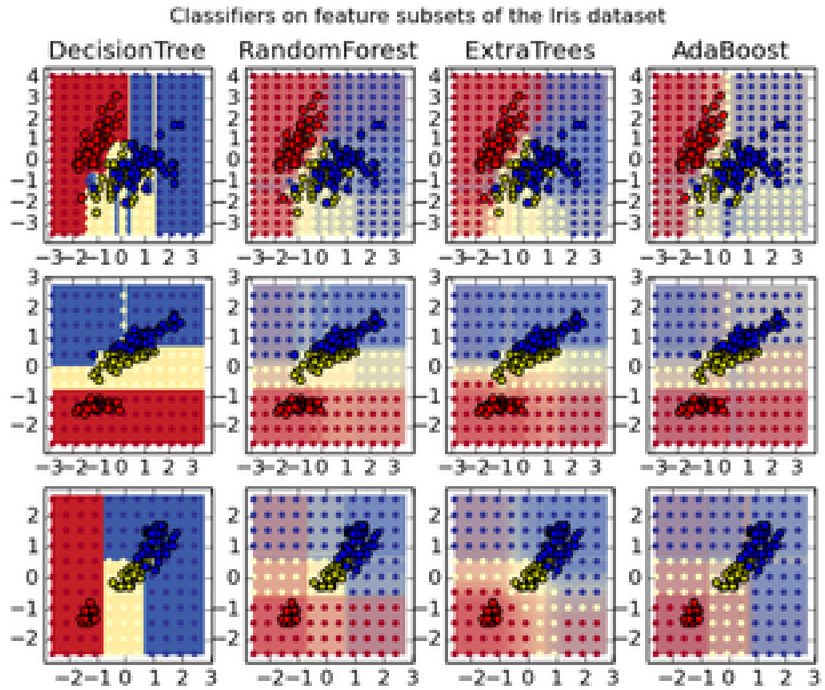


Figure 42.142: *Plot the decision surfaces of ensembles of trees on the iris dataset*

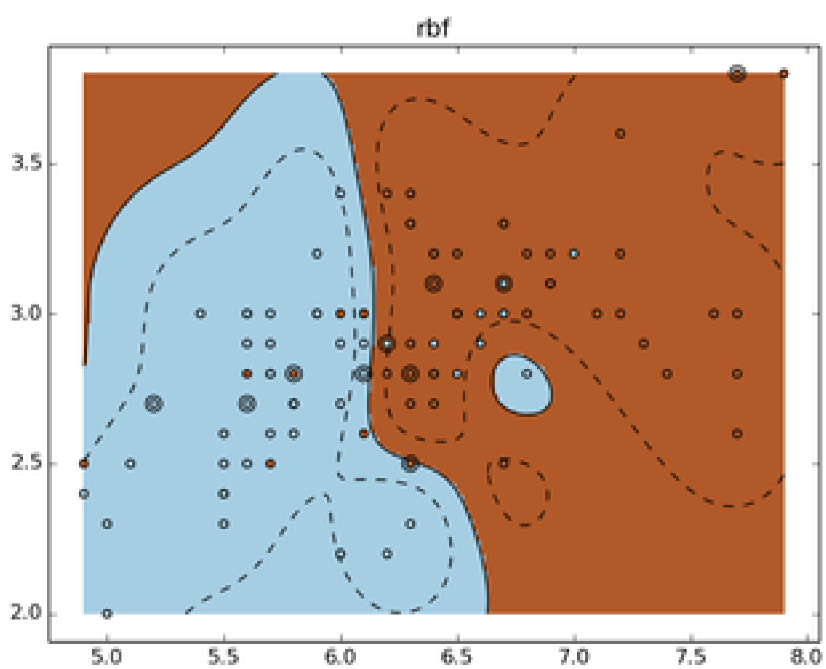


Figure 42.143: *SVM Exercise*

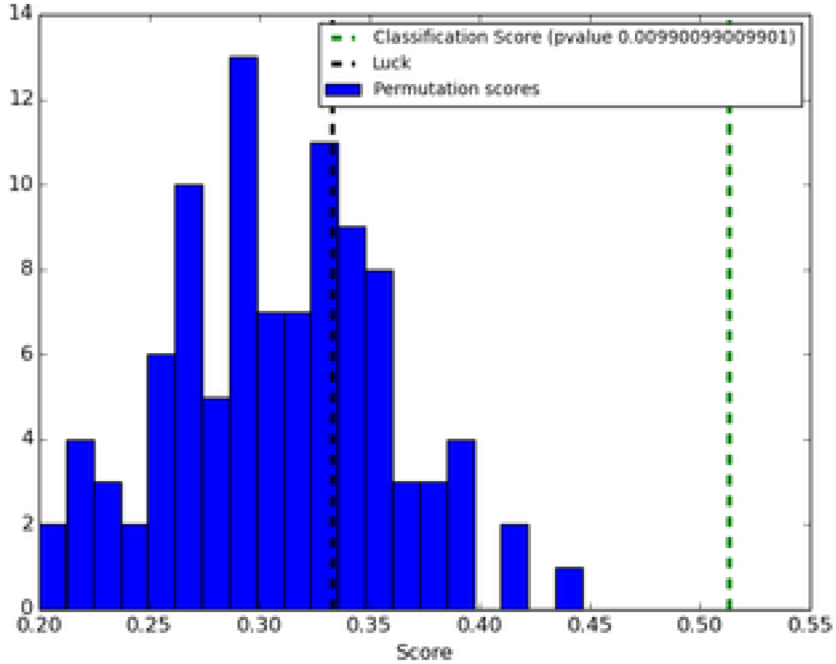


Figure 42.144: *Test with permutations the significance of a classification score*

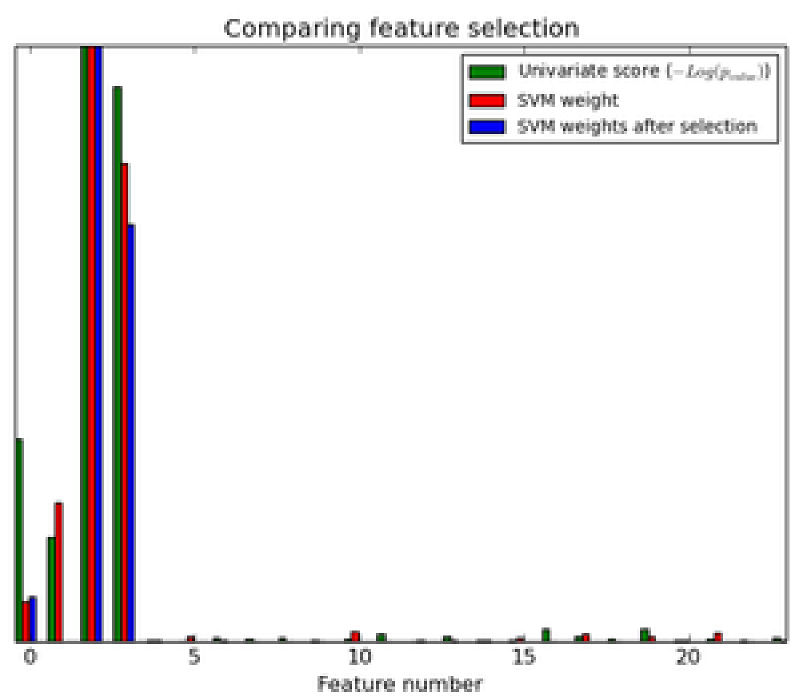


Figure 42.145: *Univariate Feature Selection*

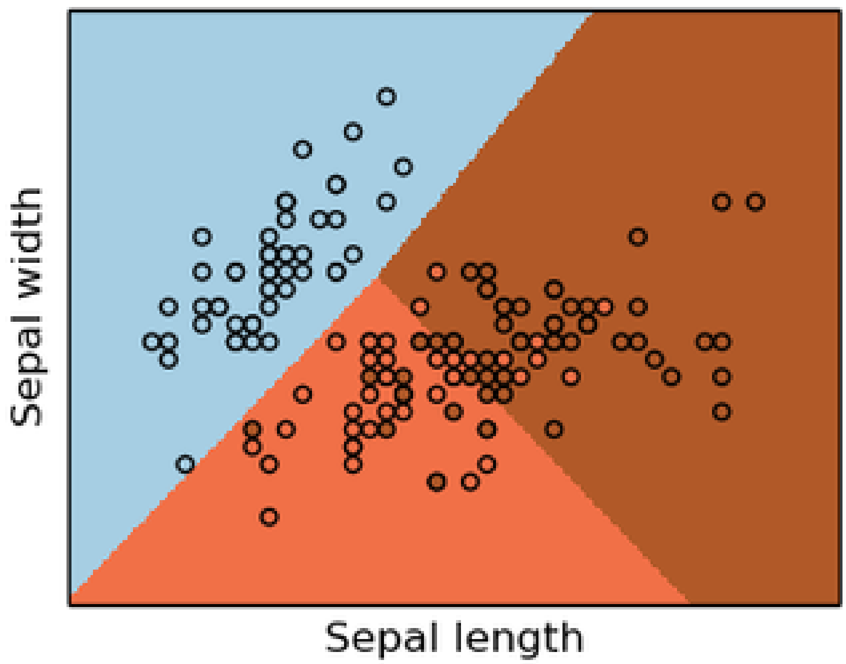


Figure 42.146: *Logistic Regression 3-class Classifier*

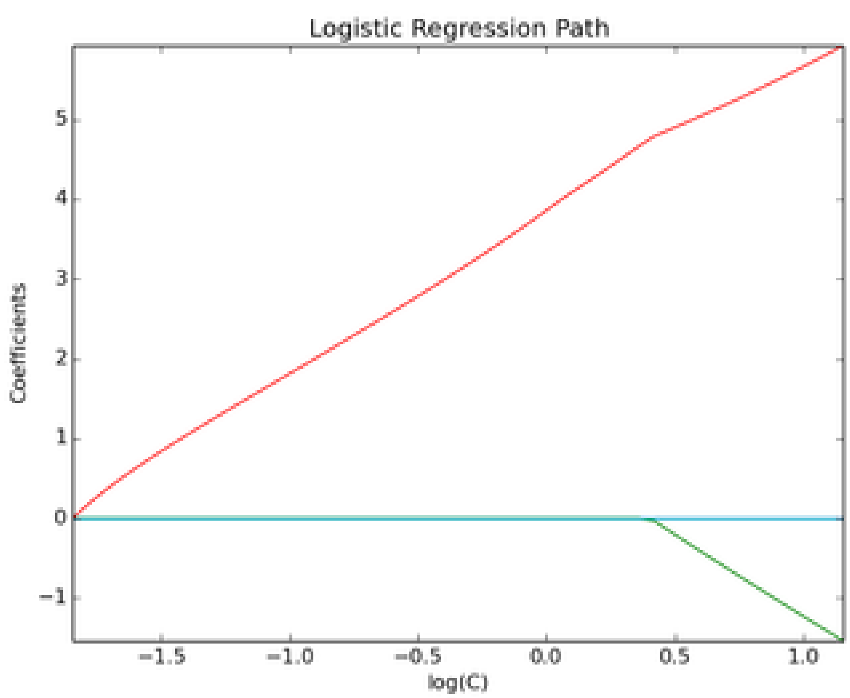


Figure 42.147: *Path with L1- Logistic Regression*

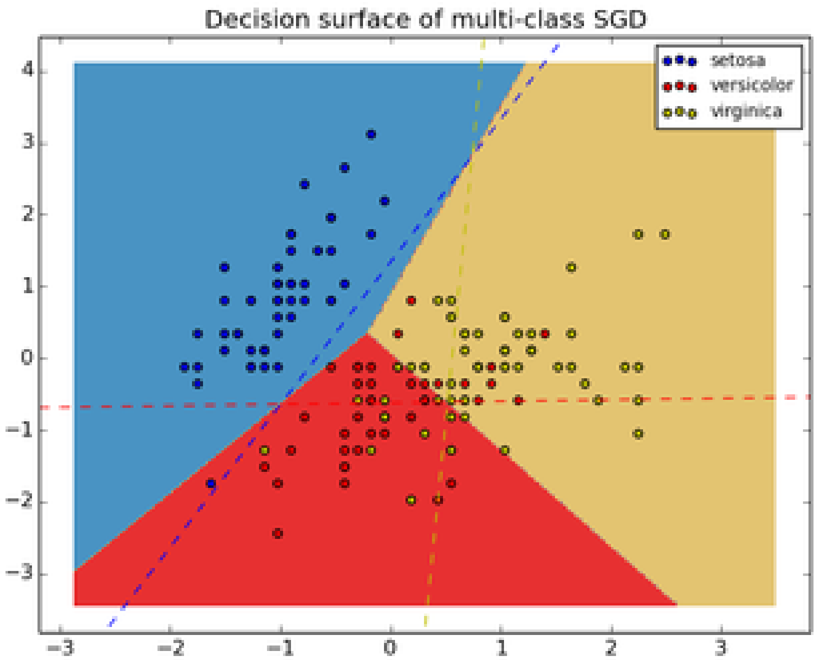


Figure 42.148: *Plot multi-class SGD on the iris dataset*

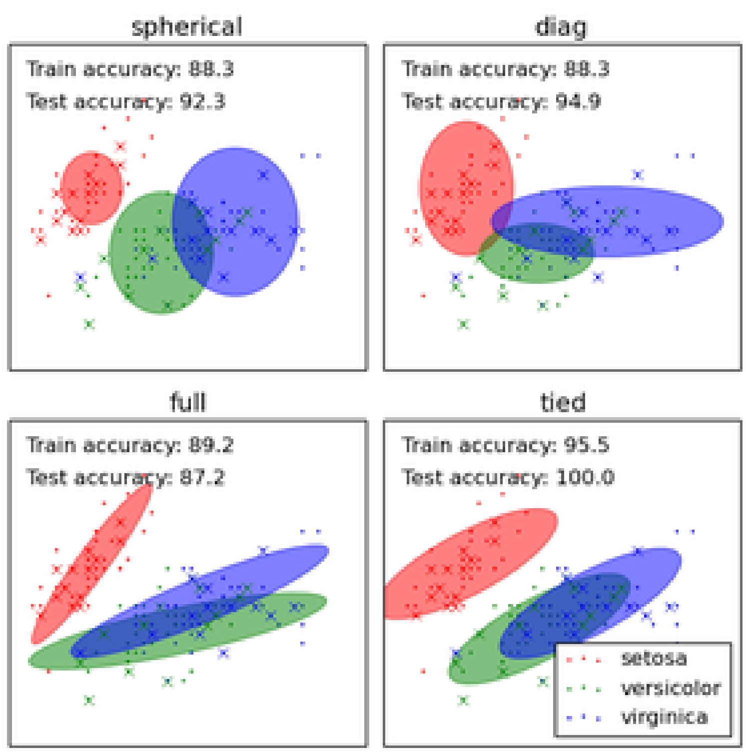


Figure 42.149: *GMM classification*

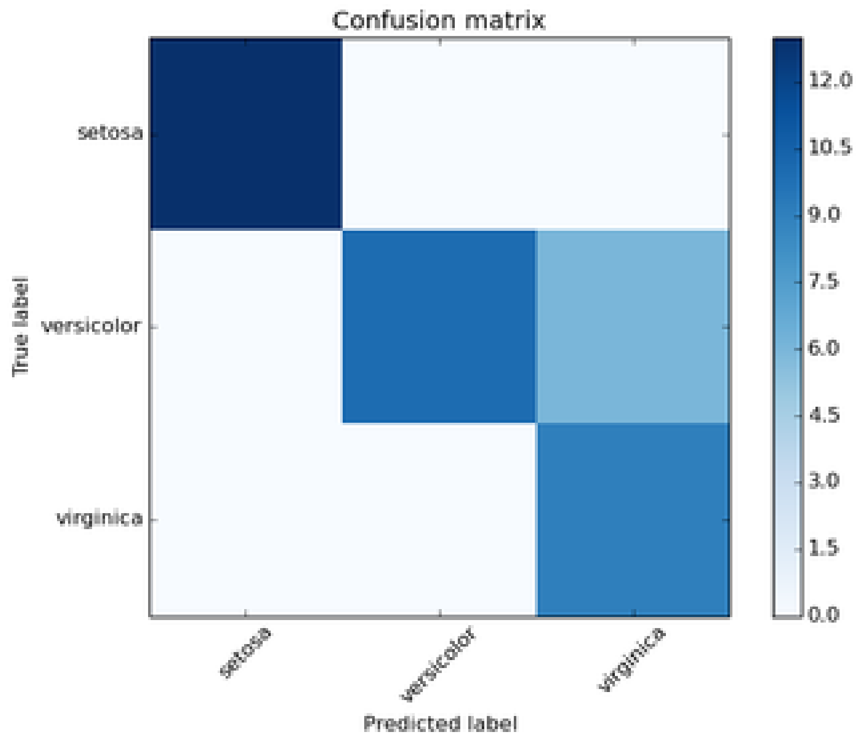


Figure 42.150: *Confusion matrix*

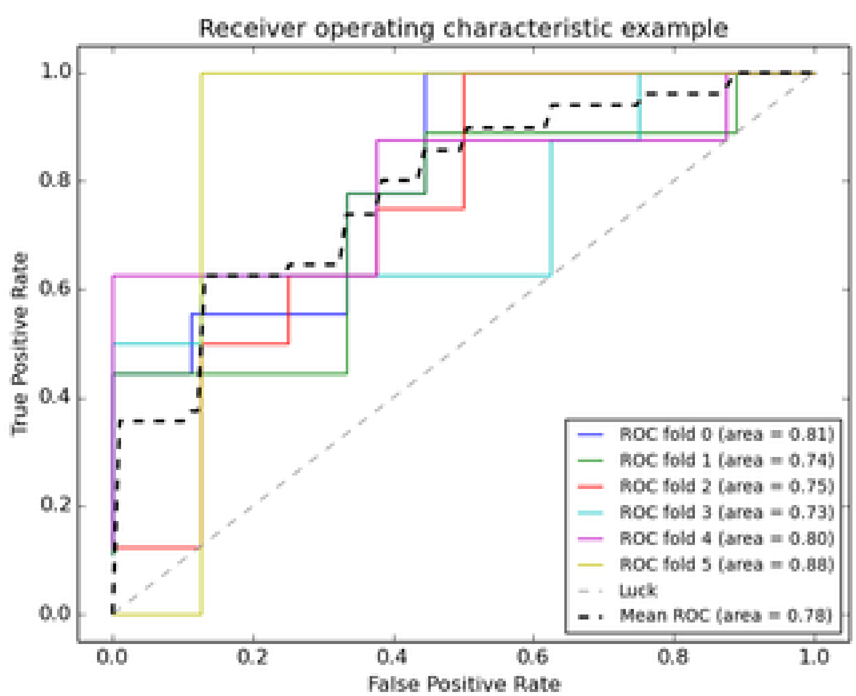


Figure 42.151: *Receiver Operating Characteristic (ROC) with cross validation*

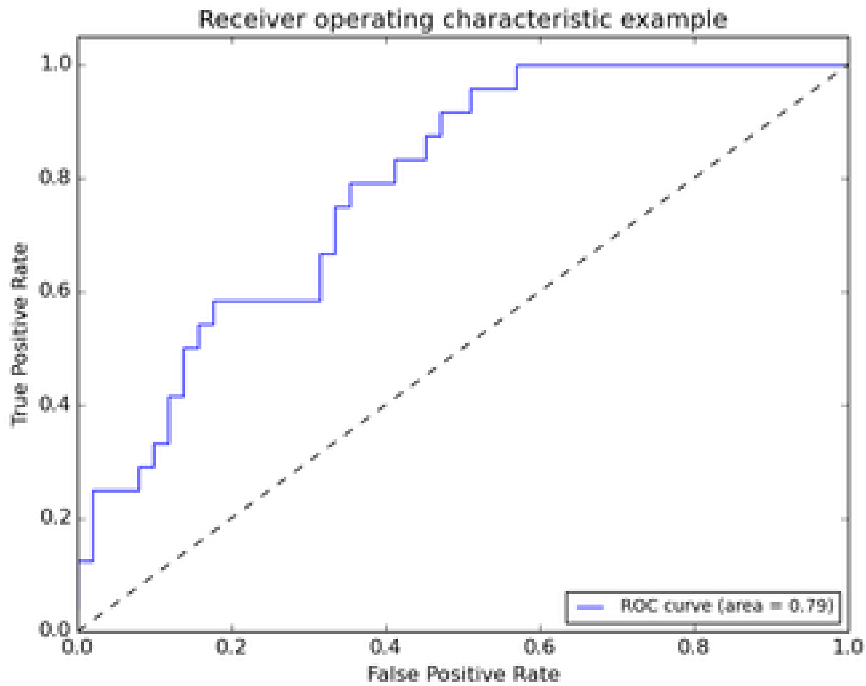


Figure 42.152: *Receiver Operating Characteristic (ROC)*

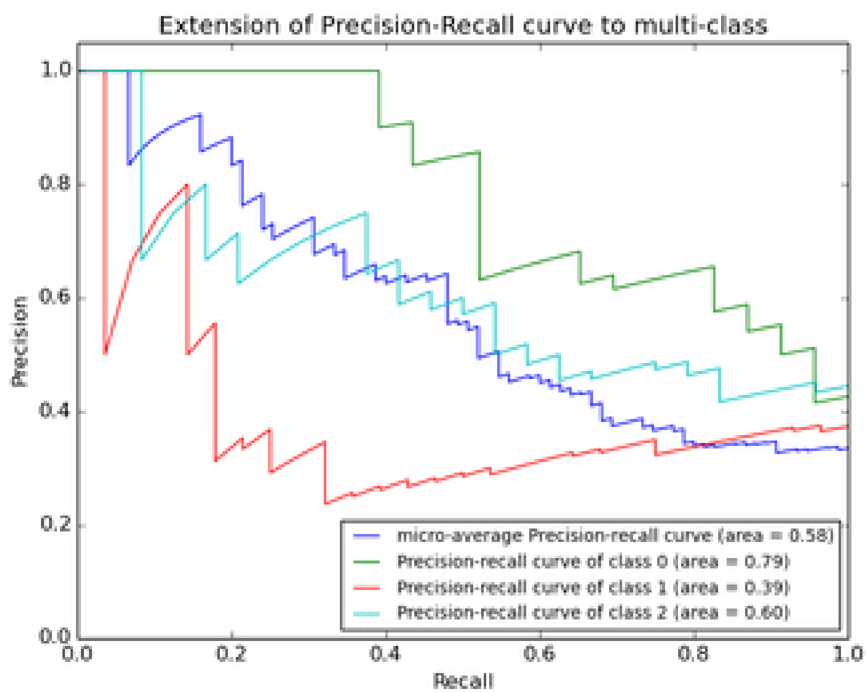


Figure 42.153: *Precision-Recall*

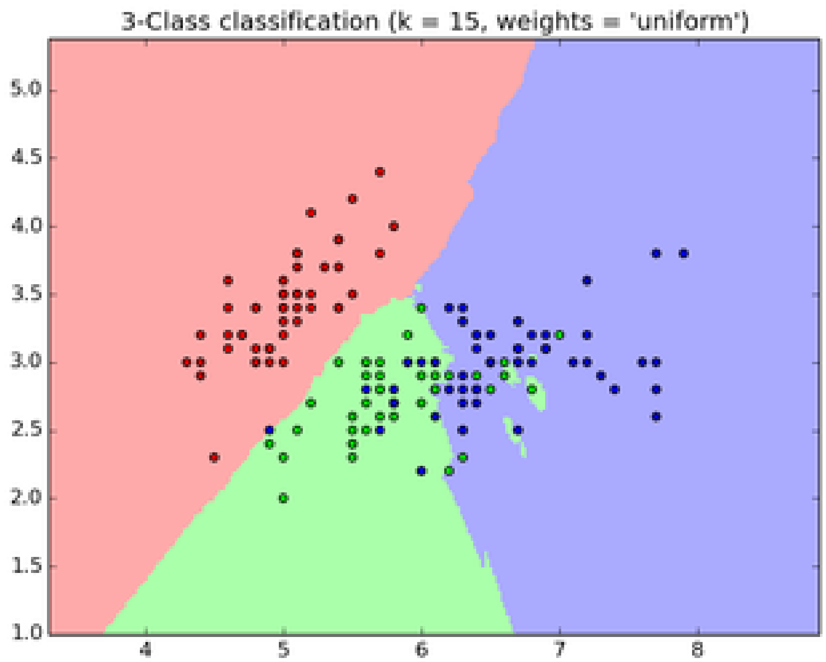


Figure 42.154: *Nearest Neighbors Classification*

Figure 42.155: *Nearest Centroid Classification*

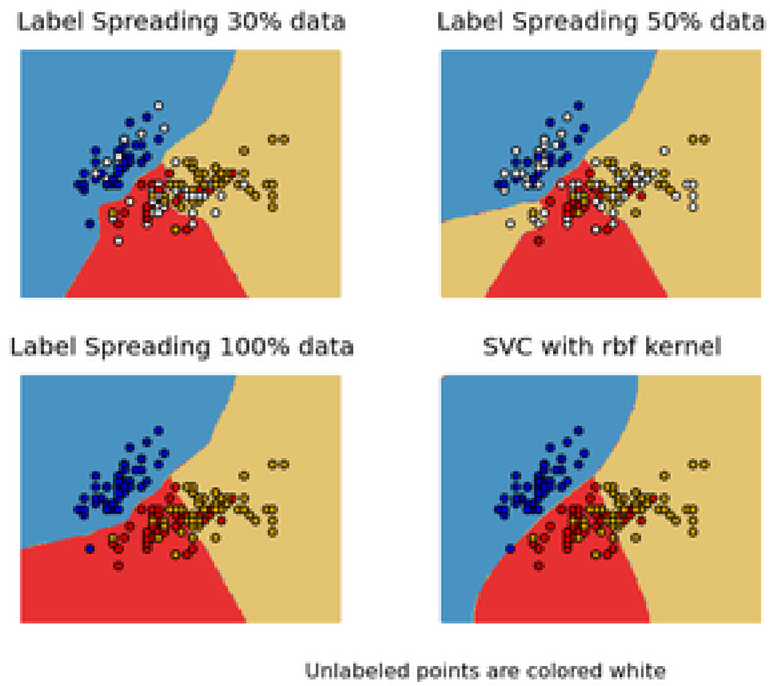


Figure 42.156: *Decision boundary of label propagation versus SVM on the Iris dataset*

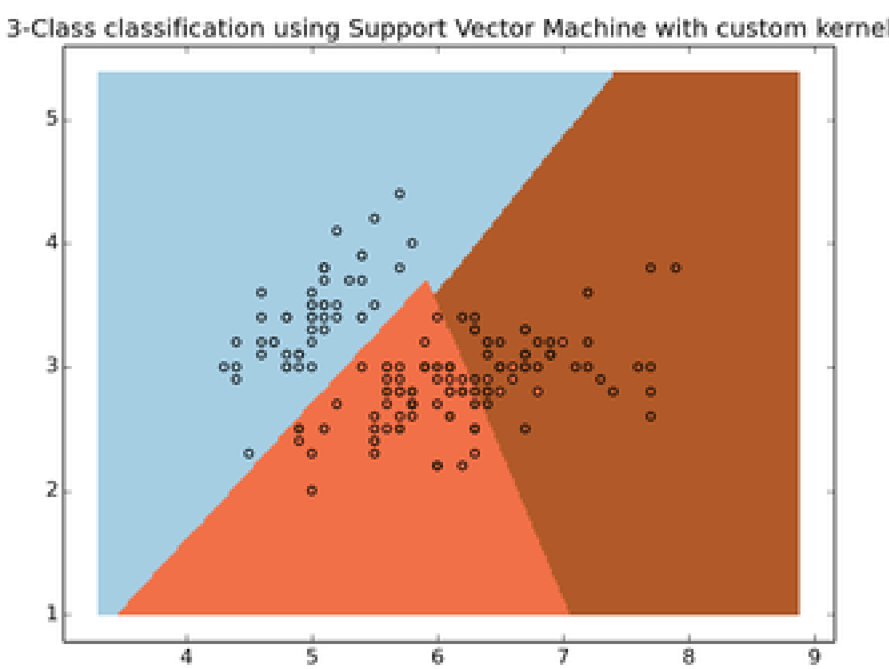


Figure 42.157: *SVM with custom kernel*

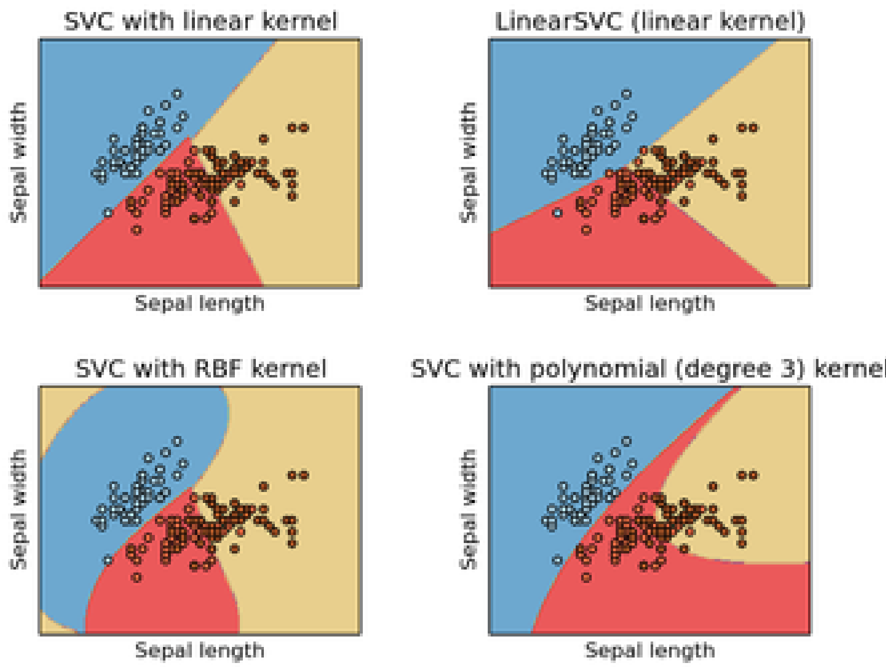


Figure 42.158: *Plot different SVM classifiers in the iris dataset*

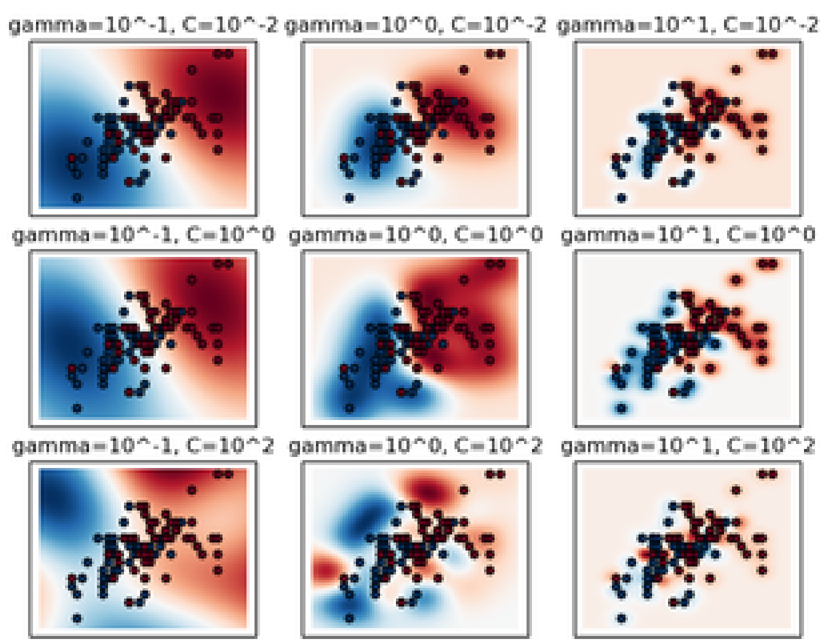


Figure 42.159: *RBF SVM parameters*

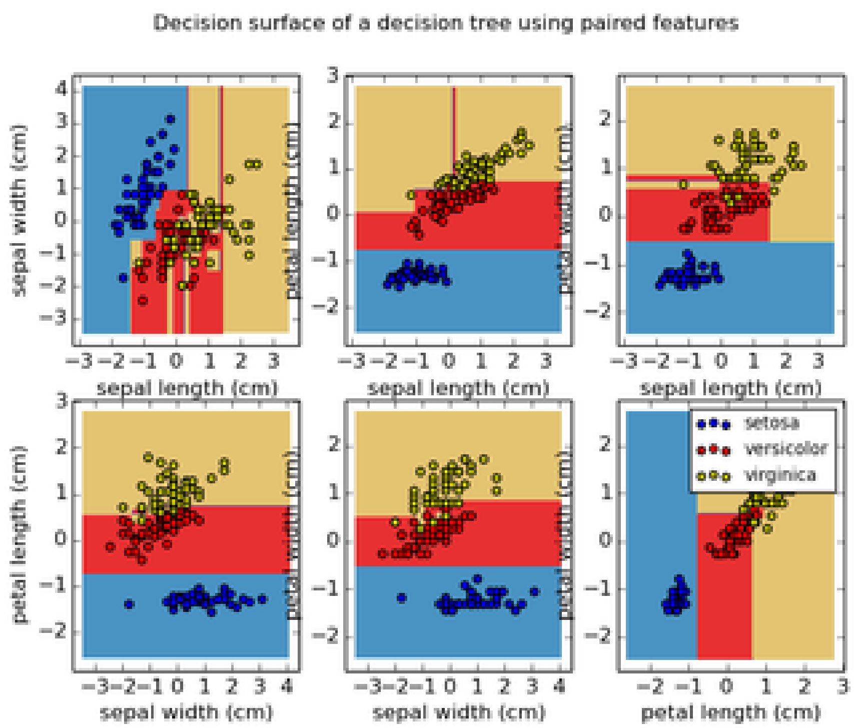


Figure 42.160: *Plot the decision surface of a decision tree on the iris dataset*

`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*)

Loader for the Labeled Faces in the Wild (LFW) pairs dataset

This dataset is a collection of JPEG pictures of famous people collected on 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. Each pixel of each channel (color in RGB) is encoded by a float in range 0.0 - 1.0.

The 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.

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 74.

Parameterssubset : 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 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.

ReturnsThe data is returned as a Bunch object with the following attributes: :

data : numpy array of shape (2200, 5828)

Each row corresponds to 2 ravel'd face images of original size 62 x 47 pixels. Changing the `slice_` or resize parameters will change the shape of the output.

pairs : numpy array of shape (2200, 2, 62, 47)

Each row has 2 face images corresponding to same or different person from the dataset containing 5749 people. Changing the `slice_` or resize parameters will change the shape of the output.

target : numpy array of shape (13233,)

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.load_lfw_people

`sklearn.datasets.load_lfw_people`(*download_if_missing=False*, ***kwargs*)

Alias for fetch_lfw_people(download_if_missing=False)

Check fetch_lfw_people.__doc__ for the documentation and parameter list.

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*)

Loader for the Labeled Faces in the Wild (LFW) people dataset

This dataset is a collection of JPEG pictures of famous people collected on 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. Each pixel of each channel (color in RGB) is encoded by a float in range 0.0 - 1.0.

The task is called Face Recognition (or Identification): given the picture of a face, find the name of the person given a training set (gallery).

The original images are 250 x 250 pixels, but the default slice and resize arguments reduce them to 62 x 74.

Parametersdata_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 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.

Returnsdataset : 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.

Examples using `sklearn.datasets.fetch_lfw_people`

Figure 42.161: *Faces recognition example using eigenfaces and SVMs*

`sklearn.datasets`.load_linnerud

`sklearn.datasets.`**`load_linnerud`**`()`

Load and return the linnerud dataset (multivariate regression).

Samples total: 20 Dimensionality: 3 for both data and targets Features: integer Targets: integer

Returnsdata : Bunch

Dictionary-like object, the interesting attributes are: 'data' and 'targets', the two multivariate datasets, with 'data' corresponding to the exercise and 'targets' corresponding to the physiological measurements, as well as 'feature_names' and 'target_names'.

`sklearn.datasets`.mldata_filename

`sklearn.datasets.`**`mldata_filename`**`(`*dataname*`)`

Convert a raw name for a data set in a mldata.org filename.

`sklearn.datasets`.fetch_mldata

`sklearn.datasets.`**`fetch_mldata`**`(`*dataname*, *target_name='label'*, *data_name='data'*, *transpose_data=True*, *data_home=None*`)`

Fetch an mldata.org data set

If the file does not exist yet, it is downloaded from mldata.org .

mldata.org does not have an enforced convention for storing data or naming the columns in a data set. The default behavior of this function works well with the most common cases:

1. data values are stored in the column 'data', and target values in the column 'label'
2. alternatively, the first column stores target values, and the second data values
3. the data array is stored as *n_features x n_samples* , and thus needs to be transposed to match the *sklearn* standard

Keyword arguments allow to adapt these defaults to specific data sets (see parameters *target_name*, *data_name*, *transpose_data*, and the examples below).

mldata.org data sets may have multiple columns, which are stored in the Bunch object with their original name.

Parametersdataname: :

Name of the data set on mldata.org, e.g.: "leukemia", "Whistler Daily Snowfall", etc. The raw name is automatically converted to a mldata.org URL .

target_name: optional, default: 'label' :

Name or index of the column containing the target values.

data_name: optional, default: 'data' :

Name or index of the column containing the data.

transpose_data: optional, default: True :

If True, transpose the downloaded data array.

data_home: optional, 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.

Returnsdata : Bunch

Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'DESCR', the full description of the dataset, and 'COL_NAMES', the original names of the dataset columns.

Examples

Load the 'iris' dataset from mldata.org:

```
>>> from sklearn.datasets.mldata import fetch_mldata
>>> import tempfile
>>> test_data_home = tempfile.mkdtemp()

>>> iris = fetch_mldata('iris', data_home=test_data_home)
>>> iris.target.shape
(150,)
>>> iris.data.shape
(150, 4)
```

Load the 'leukemia' dataset from mldata.org, which needs to be transposed to respects the sklearn axes convention:

```
>>> leuk = fetch_mldata('leukemia', transpose_data=True,
...                     data_home=test_data_home)
>>> leuk.data.shape
(72, 7129)
```

Load an alternative 'iris' dataset, which has different names for the columns:

```
>>> iris2 = fetch_mldata('datasets-UCI iris', target_name=1,
...                      data_name=0, data_home=test_data_home)
>>> iris3 = fetch_mldata('datasets-UCI iris',
...                      target_name='class', data_name='double0',
...                      data_home=test_data_home)

>>> import shutil
>>> shutil.rmtree(test_data_home)
```

sklearn.datasets.fetch_olivetti_faces

sklearn.datasets.**fetch_olivetti_faces**(*data_home=None*, *shuffle=False*, *random_state=0*, *download_if_missing=True*)

Loader for the Olivetti faces data-set from AT&T.

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.

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.

random_state : optional, integer or RandomState object

The seed or the random number generator used to shuffle the data.

Returns An object with the following attributes: :

data : numpy array of shape (400, 4096)

Each row corresponds to a ravelled face image of original size 64 x 64 pixels.

images : numpy array of shape (400, 64, 64)

Each row is a face image corresponding to one of the 40 subjects of the dataset.

target : numpy array of 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.

Notes

This dataset consists of 10 pictures each of 40 individuals. The original database was available from (now defunct)

http://www.uk.research.att.com/facedatabase.html

The version retrieved here comes in MATLAB format from the personal web page of Sam Roweis:

http://www.cs.nyu.edu/~roweis/

Examples using sklearn.datasets.fetch_olivetti_faces

sklearn.datasets.fetch_california_housing

sklearn.datasets.**fetch_california_housing**(*data_home=None*, *download_if_missing=True*)

Loader for the California housing dataset from StatLib.

Figure 42.162: *Face completion with a multi-output estimators*

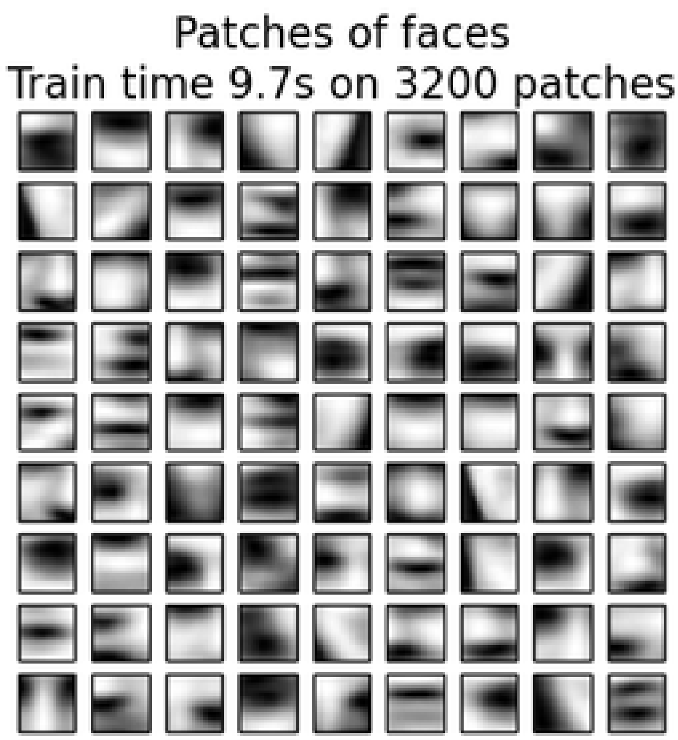


Figure 42.163: *Online learning of a dictionary of parts of faces*

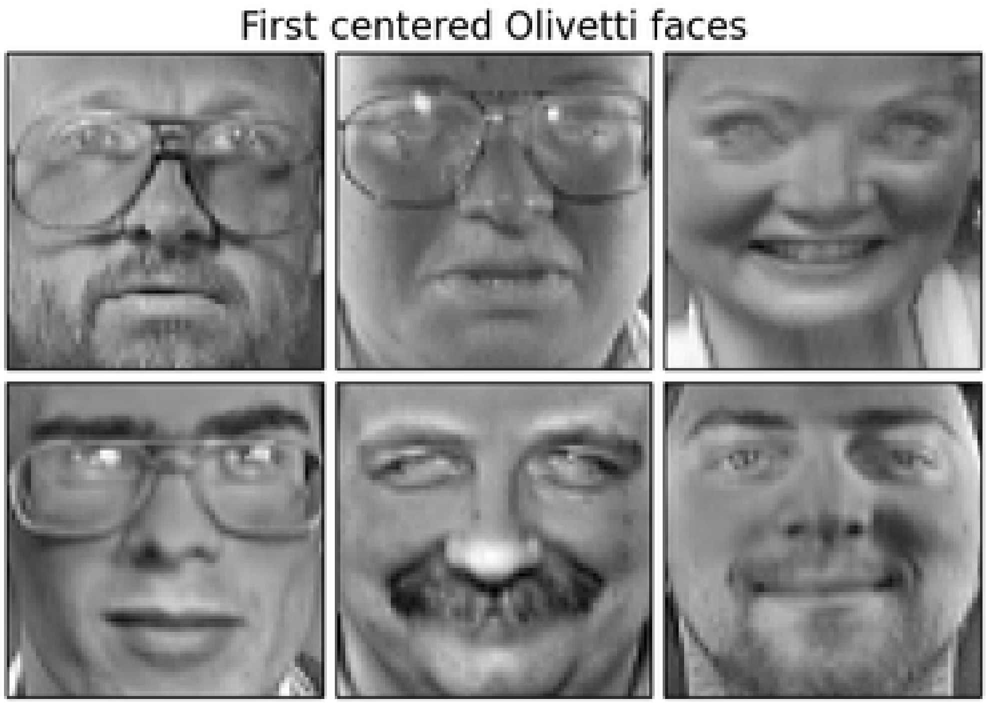


Figure 42.164: *Faces dataset decompositions*

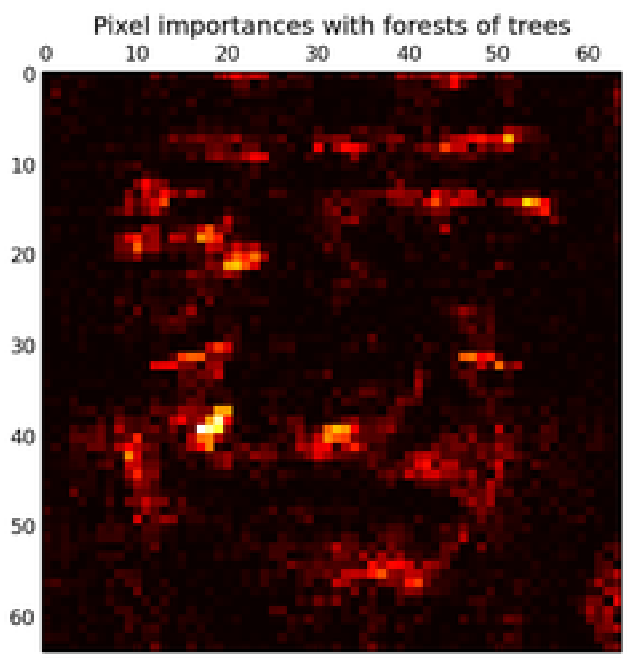


Figure 42.165: *Pixel importances with a parallel forest of trees*

Parametersdata_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.

Returnsdataset : 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.

Notes

This dataset consists of 20,640 samples and 9 features.

Examples using `sklearn.datasets.fetch_california_housing`

`sklearn.datasets`.fetch_covtype

`sklearn.datasets.`**`fetch_covtype`**(*data_home=None*, *download_if_missing=True*, *random_state=None*, *shuffle=False*)

Load the covertype dataset, downloading it if necessary.

Parametersdata_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, optional (default=None)

Random state for shuffling the dataset. 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 : bool, default=False

Whether to shuffle dataset.

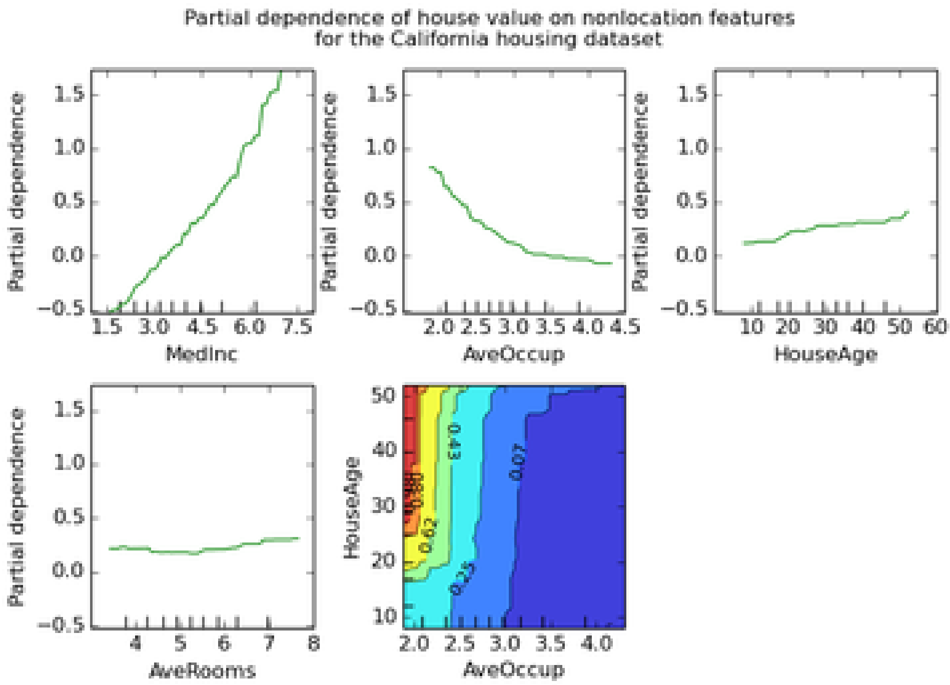


Figure 42.166: *Partial Dependence Plots*

Returnsdataset : 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.

sklearn.datasets.load_mlcomp

sklearn.datasets.**load_mlcomp**(*name_or_id*, *set_='raw'*, *mlcomp_root=None*, ***kwargs*)

Load a datasets as downloaded from http://mlcomp.org

Parametersname_or_id : the integer id or the string name metadata of the MLComp

dataset to load

set_ : select the portion to load: 'train', 'test' or 'raw'

mlcomp_root : the filesystem path to the root folder where MLComp datasets

are stored, if mlcomp_root is None, the MLCOMP_DATASETS_HOME environment variable is looked up instead.

****kwargs** : domain specific kwargs to be passed to the dataset loader.

Returnsdata : Bunch

Dictionary-like object, the interesting attributes are: 'filenames', the files holding the raw to learn, 'target', the classification labels (integer index), 'target_names', the meaning of the labels, and 'DESCR', the full description of the dataset.

Note on the lookup process: depending on the type of name_or_id, :

will choose between integer id lookup or metadata name lookup by :

looking at the unzipped archives and metadata file. :

TODO: implement zip dataset loading too :

Examples using `sklearn.datasets.load_mlcomp`

Figure 42.167: *Classification of text documents: using a MLComp dataset*

`sklearn.datasets`.load_sample_image

`sklearn.datasets.`**`load_sample_image`**(*image_name*)

Load the numpy array of a single sample image

Parametersimage_name: {'china.jpg', 'flower.jpg'} :

The name of the sample image loaded

Returnsimg: 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')
>>> china.dtype
dtype('uint8')
>>> china.shape
(427, 640, 3)
>>> flower = load_sample_image('flower.jpg')
>>> flower.dtype
dtype('uint8')
```

```
>>> flower.shape
(427, 640, 3)
```

Examples using `sklearn.datasets.load_sample_image`

Figure 42.168: *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`.

Returnsdata : 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()
>>> len(dataset.images)
2
>>> first_img_data = dataset.images[0]
```

```
>>> first_img_data.shape
(427, 640, 3)
>>> first_img_data.dtype
dtype('uint8')
```

sklearn.datasets.load_svmlight_file

sklearn.datasets.**load_svmlight_file**(*f*, *n_features=None*, *dtype=<type 'numpy.float64'>*, *multilabel=False*, *zero_based='auto'*, *query_id=False*)

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

Parametersf : {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.

multilabel : boolean, optional, default False

Samples may have several labels each (see http://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.

query_id : boolean, default False

If True, will return the query_id array for each file.

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`.

ReturnsX: 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_filessimilar function for loading multiple files in this

`format,enforcing`

Examples

To use joblib.Memory to cache the svmlight file:

```
from sklearn.externals.joblib import Memory
from sklearn.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=<type 'numpy.float64'>*, *multilabel=False*, *zero_based='auto'*, *query_id=False*)

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.

Parametersfiles : 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.

multilabel: boolean, optional :

Samples may have several labels each (see http://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.

query_id: boolean, defaults to False :

If True, will return the query_id array for each file.

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`.

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.dump_svmlight_file

`sklearn.datasets.dump_svmlight_file`(*X*, *y*, *f*, *zero_based=True*, *comment=None*, *query_id=None*)

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.

ParametersX : {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.

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, shape = [n_samples]

Array containing pairwise preference constraints (qid in svmlight format).

Examples using `sklearn.datasets.dump_svmlight_file`

Figure 42.169: *Libsvm GUI*

42.6.2 Samples generator

`datasets.make_blobs`([n_samples, n_features, ...])	Generate isotropic Gaussian blobs for clustering.
`datasets.make_classification`([n_samples, ...])	Generate a random n-class classification problem.
`datasets.make_circles`([n_samples, shuffle, ...])	Make a large circle containing a smaller circle in 2d.
`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 value
`datasets.make_moons`([n_samples, shuffle, ...])	Make two interleaving half circles
`datasets.make_multilabel_classification`([...])	Generate a random multilabel classification problem.

Continued on nex

Table 42.36 – continued from previous page

datasets.make_regression([n_samples, ...])	Generate a random regression problem.
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 de
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.
datasets.make_biclusters(shape, n_clusters)	Generate an array with constant block diagonal structure for biclus
datasets.make_checkerboard(shape, n_clusters)	Generate an array with block checkerboard structure for biclusterin

sklearn.datasets.make_blobs

sklearn.datasets.**make_blobs**(*n_samples=100*, *n_features=2*, *centers=3*, *cluster_std=1.0*, *center_box=(-10.0, 10.0)*, *shuffle=True*, *random_state=None*)

Generate isotropic Gaussian blobs for clustering.

Parametersn_samples : int, optional (default=100)

The total number of points equally divided among clusters.

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=3) The number of centers to generate, or the fixed center locations.

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, 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*.

ReturnsX : 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_classificationa more intricate variant

Examples

```
>>> from sklearn.datasets.samples_generator 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])
```

Examples using `sklearn.datasets.make_blobs`

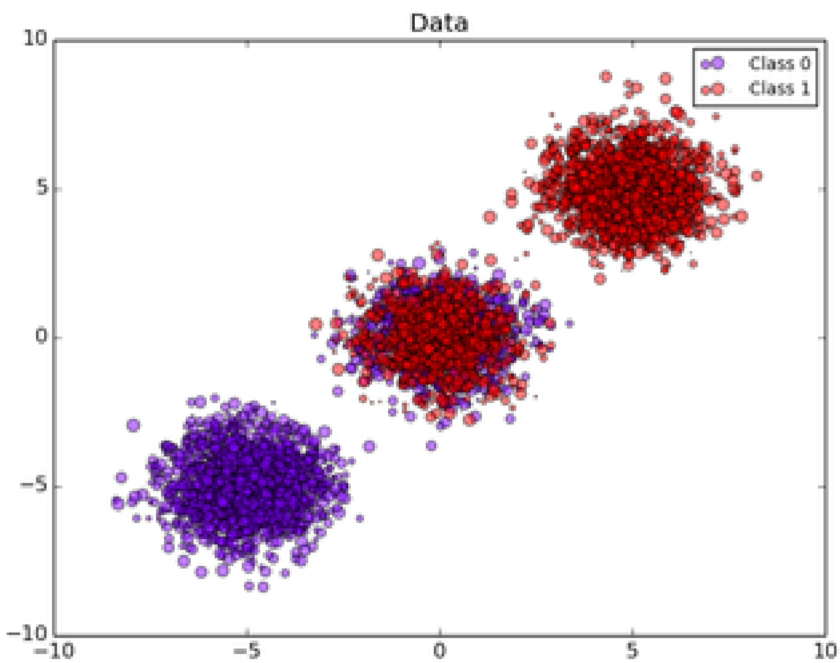


Figure 42.170: *Probability calibration of classifiers*

`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 a *2 * class_sep*-sided hypercube, 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.

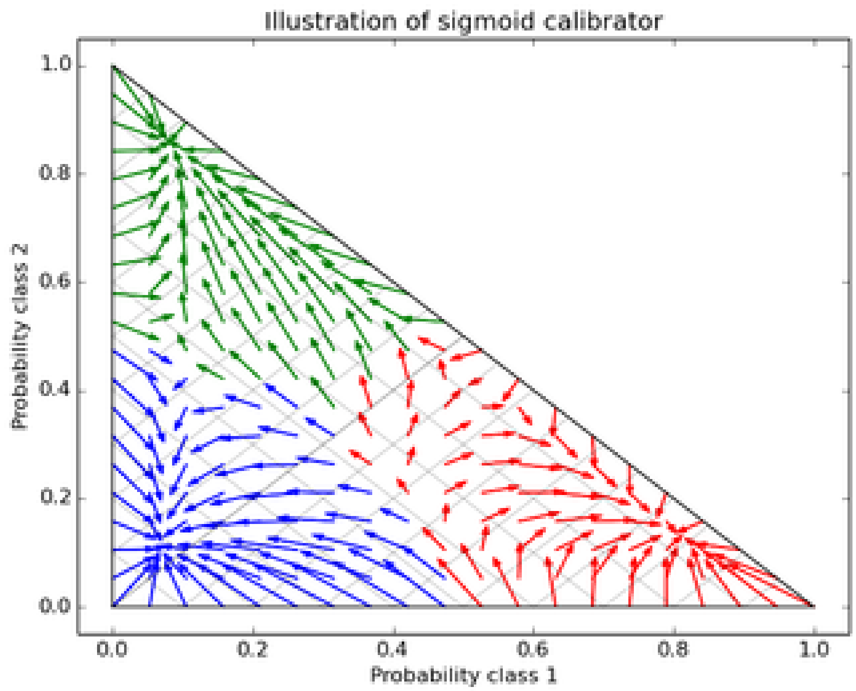


Figure 42.171: *Probability Calibration for 3-class classification*

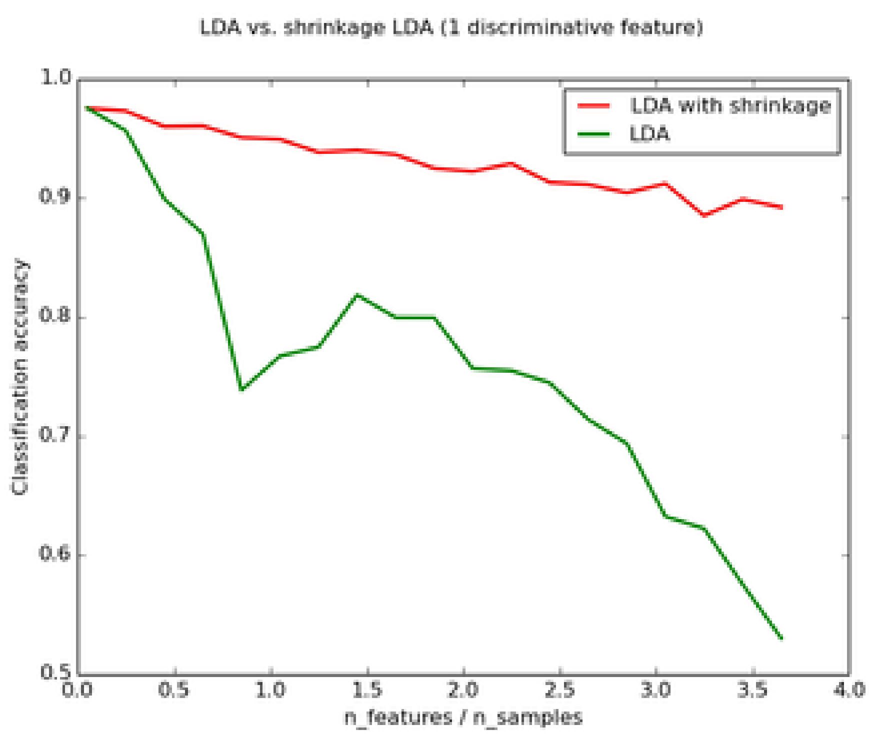


Figure 42.172: *Normal and Shrinkage Linear Discriminant Analysis for classification*

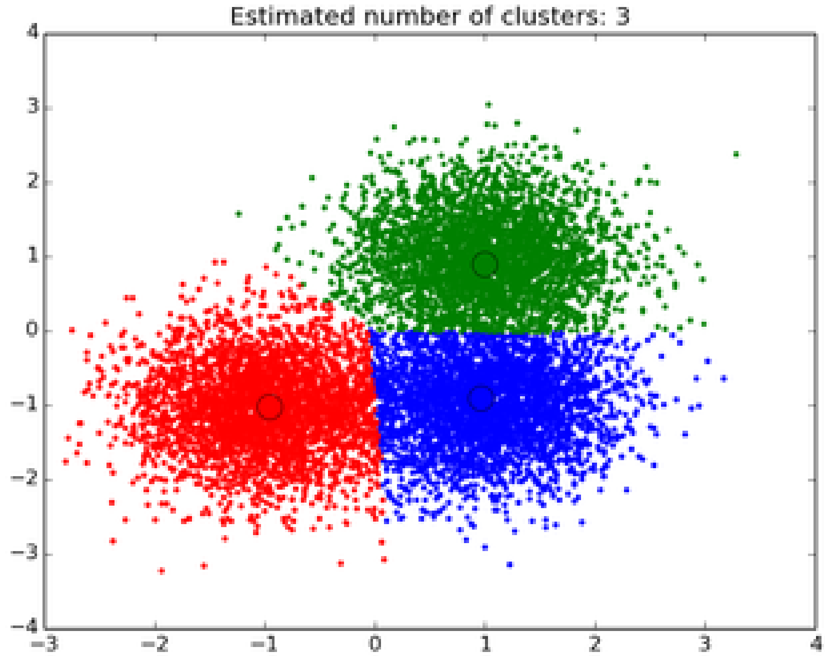


Figure 42.173: *A demo of the mean-shift clustering algorithm*

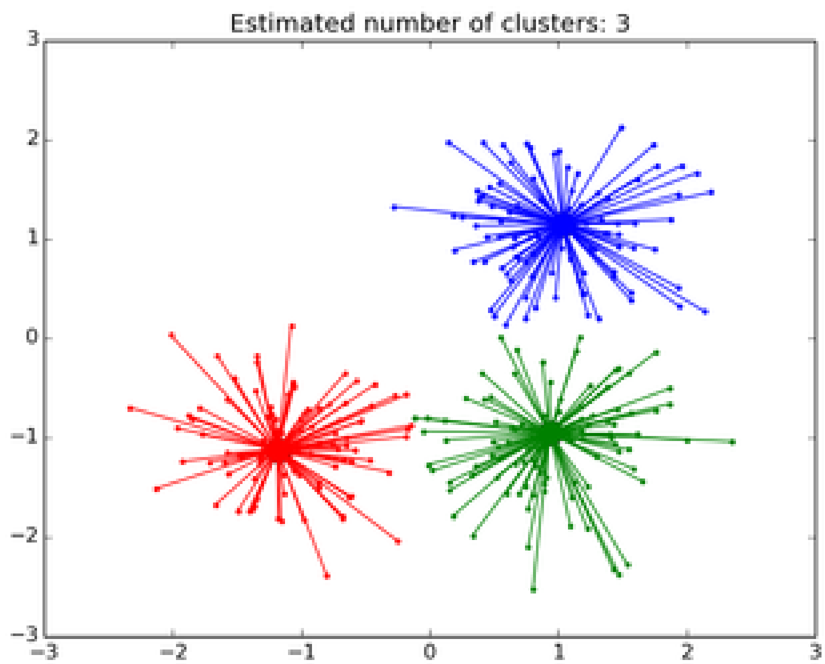


Figure 42.174: *Demo of affinity propagation clustering algorithm*

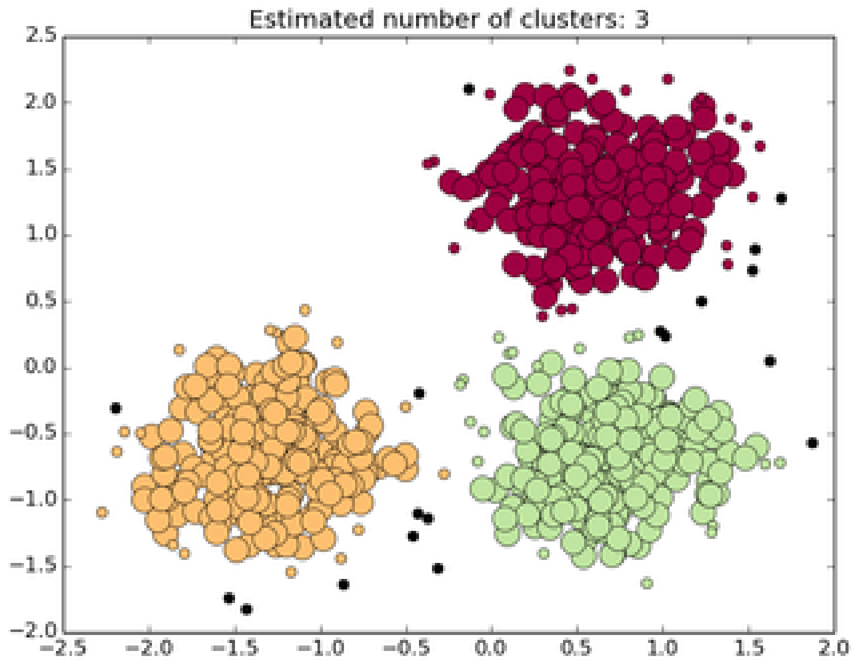


Figure 42.175: *Demo of DBSCAN clustering algorithm*

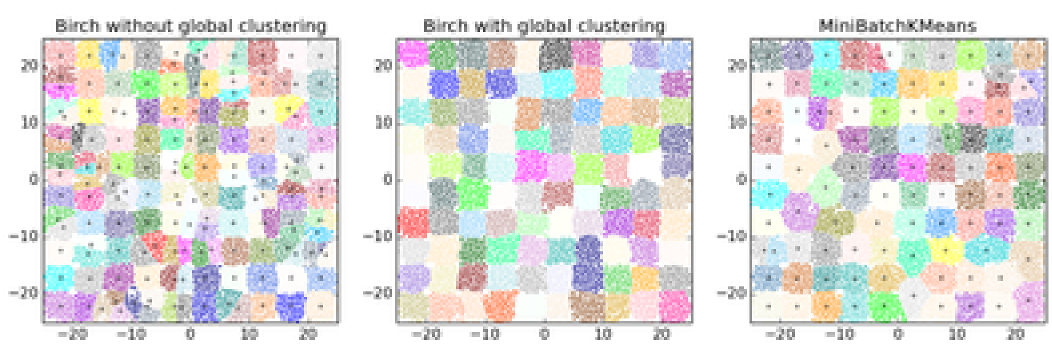


Figure 42.176: *Compare BIRCH and MiniBatchKMeans*

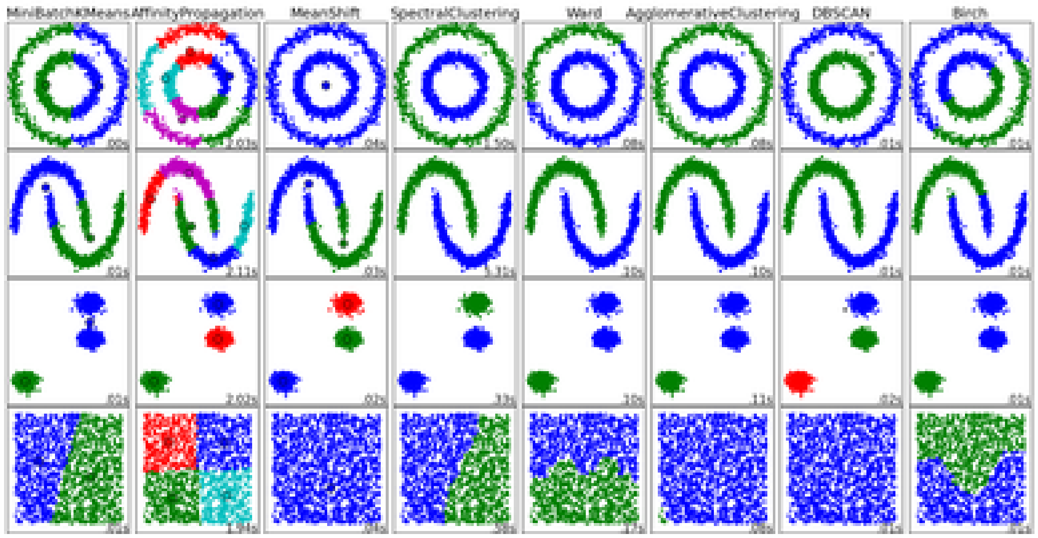


Figure 42.177: *Comparing different clustering algorithms on toy datasets*

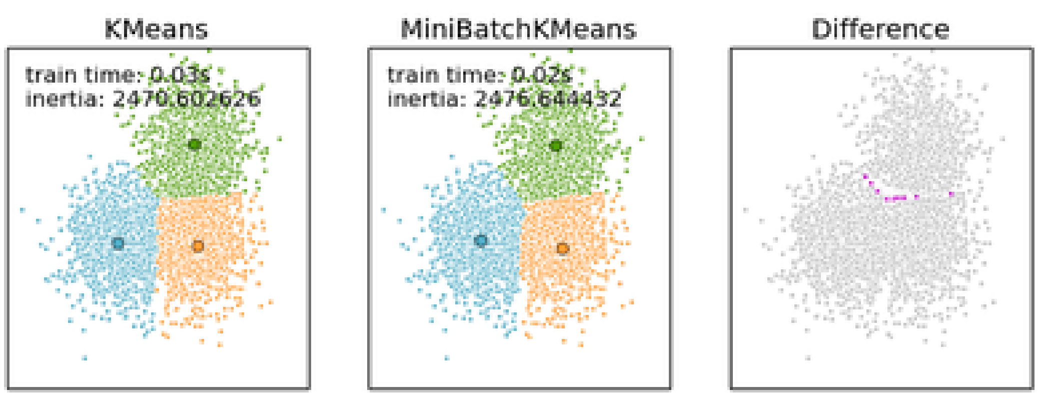


Figure 42.178: *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*

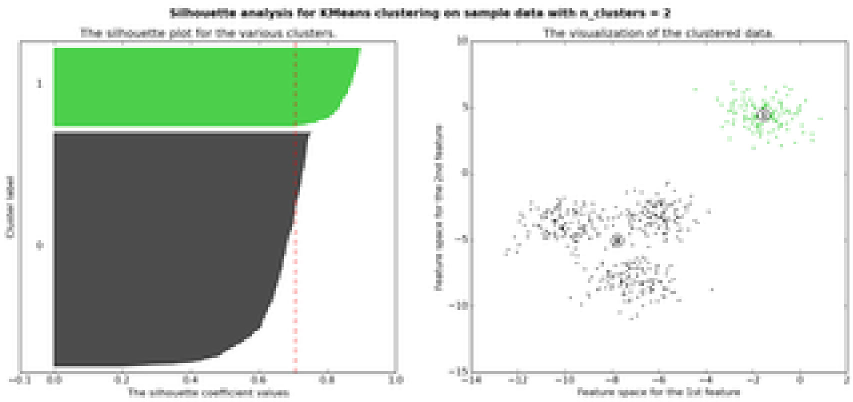


Figure 42.179: *Selecting the number of clusters with silhouette analysis on KMeans clustering*

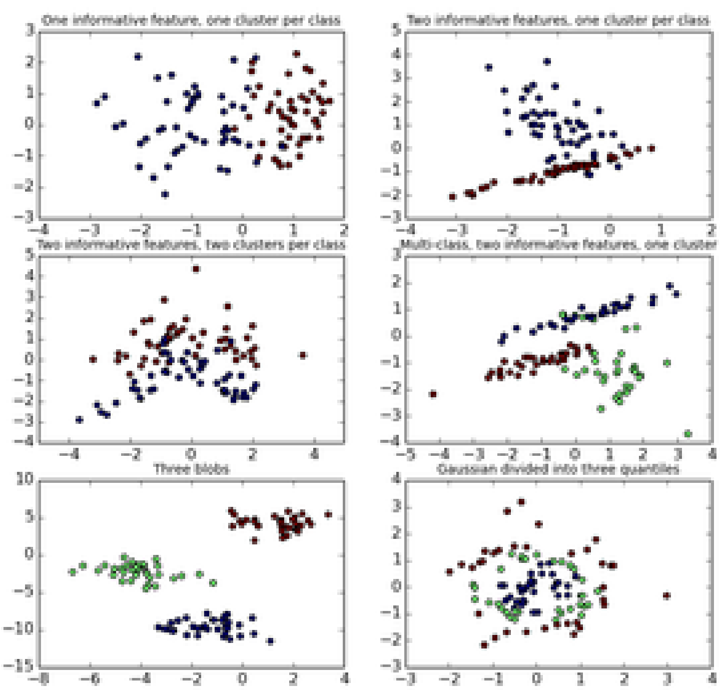


Figure 42.180: *Plot randomly generated classification dataset*

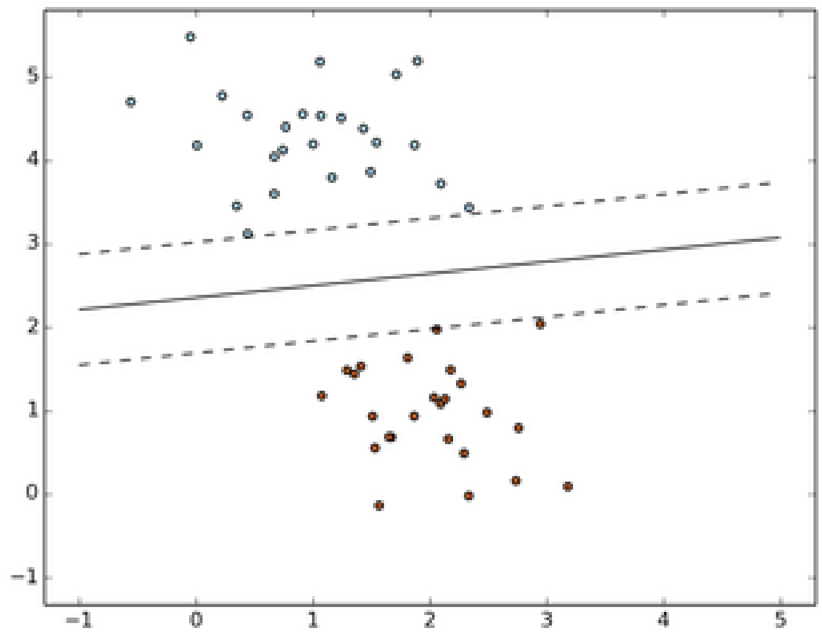


Figure 42.181: *SGD: Maximum margin separating hyperplane*

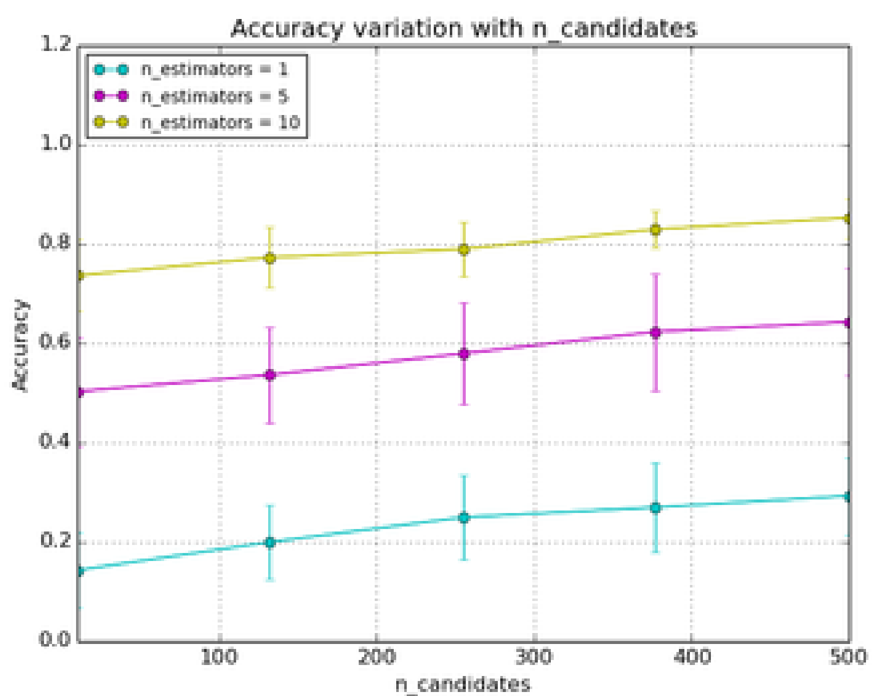


Figure 42.182: *Hyper-parameters of Approximate Nearest Neighbors*

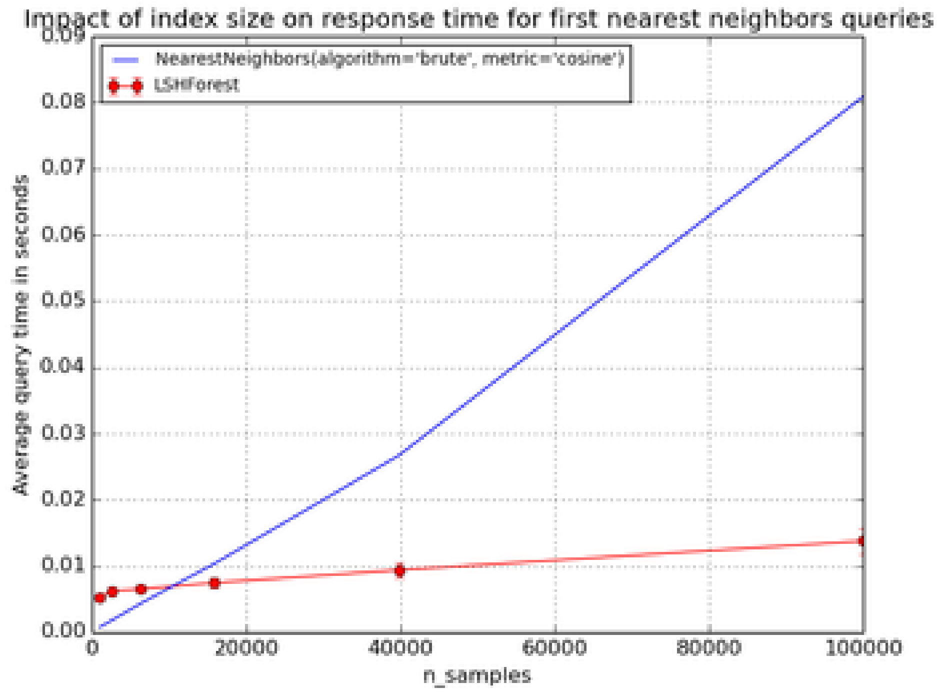


Figure 42.183: *Scalability of Approximate Nearest Neighbors*

Prior to shuffling, *X* stacks a number of these primary "informative" features, "redundant" linear combinations of these, "repeated" duplicates of sampled features, and arbitrary noise for and remaining features.

Parametersn_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 : list of floats or None (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 are randomly exchanged.

class_sep : float, optional (default=1.0)

The factor multiplying the hypercube dimension.

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, 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*.

ReturnsX : 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

[R109]

Examples using `sklearn.datasets.make_classification`

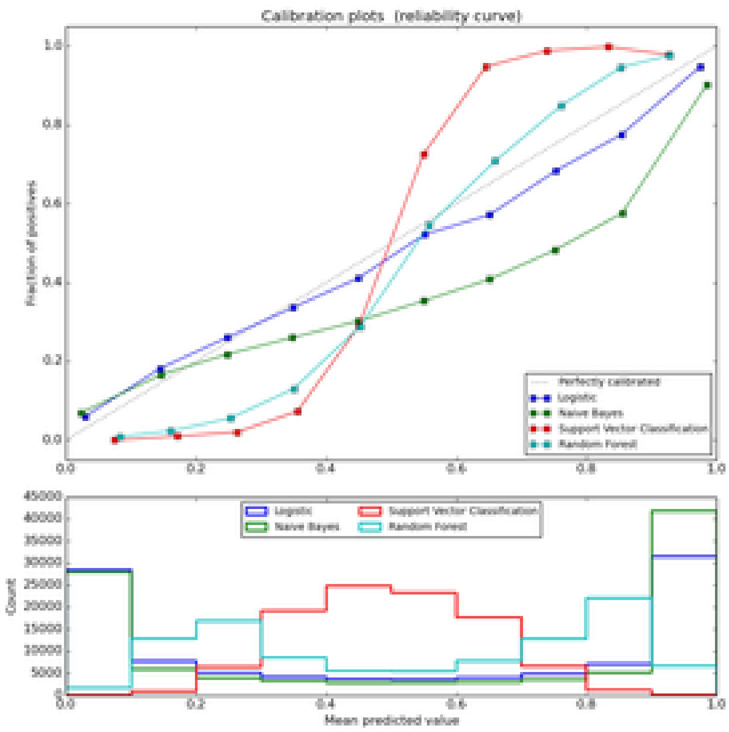


Figure 42.184: *Comparison of Calibration of Classifiers*

`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.

Parametersn_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.

factor : double < 1 (default=.8)

Scale factor between inner and outer circle.

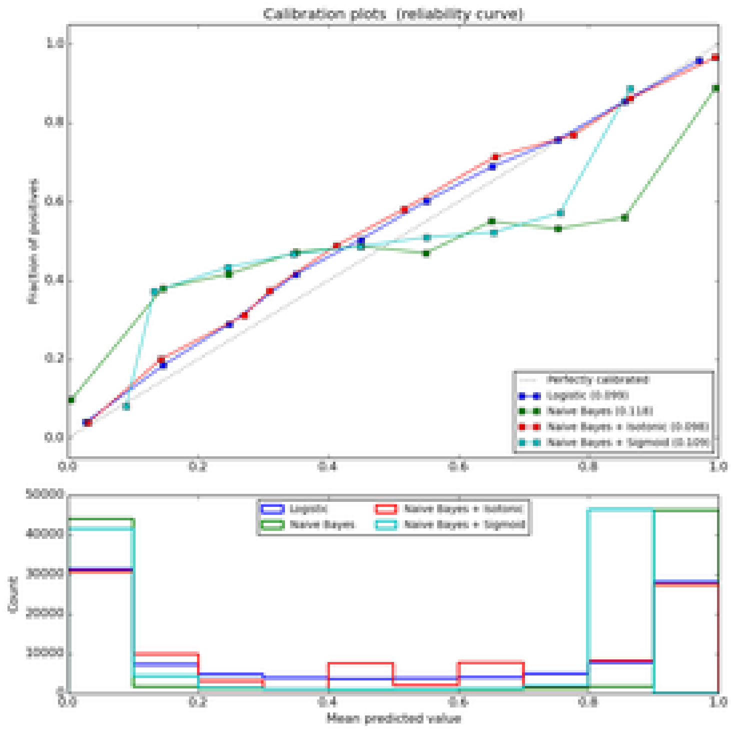


Figure 42.185: *Probability Calibration curves*

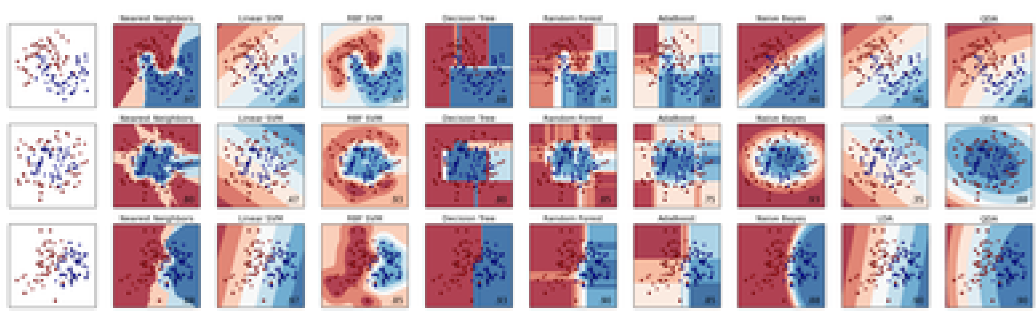


Figure 42.186: *Classifier comparison*

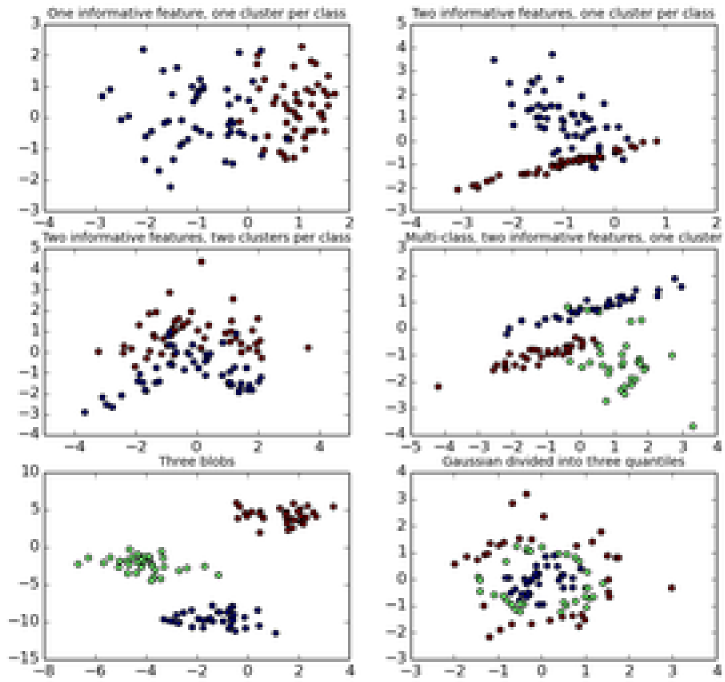


Figure 42.187: *Plot randomly generated classification dataset*

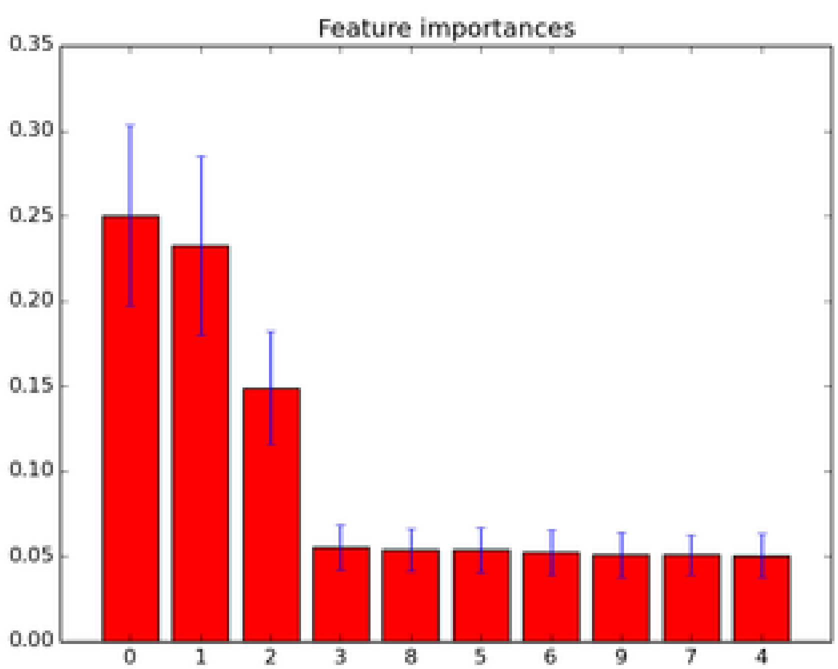


Figure 42.188: *Feature importances with forests of trees*

Figure 42.189: *Pipeline Anova SVM*

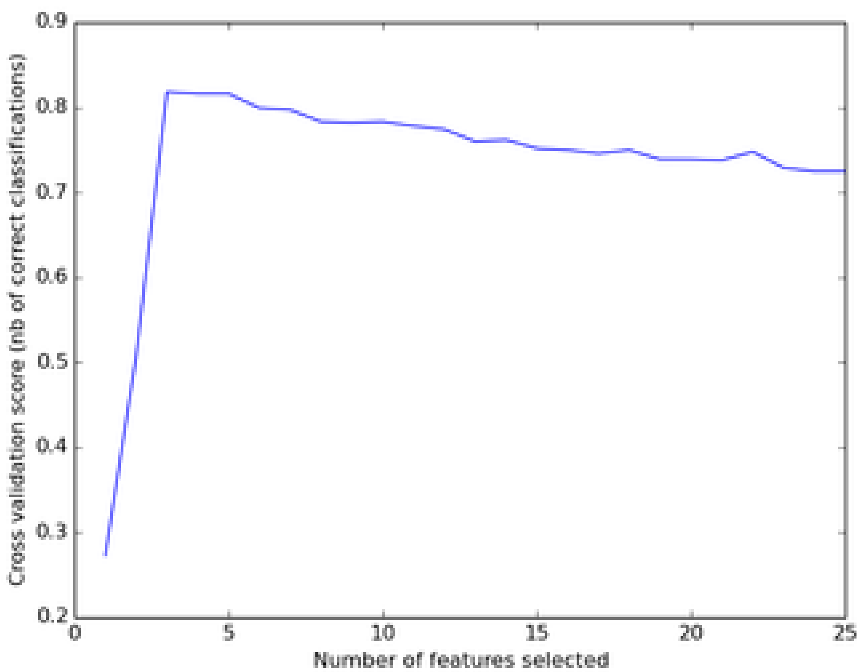


Figure 42.190: *Recursive feature elimination with cross-validation*

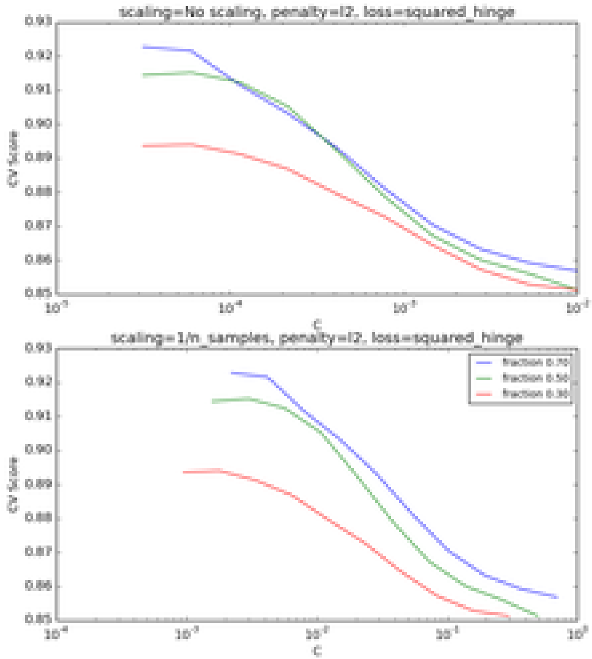


Figure 42.191: *Scaling the regularization parameter for SVCs*

ReturnsX : 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`

`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]
```

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.

Parametersn_samples : int, optional (default=100)

The number of samples.

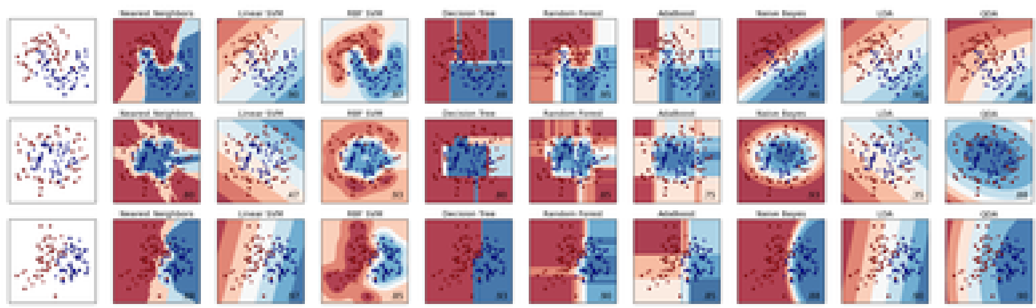


Figure 42.192: *Classifier comparison*

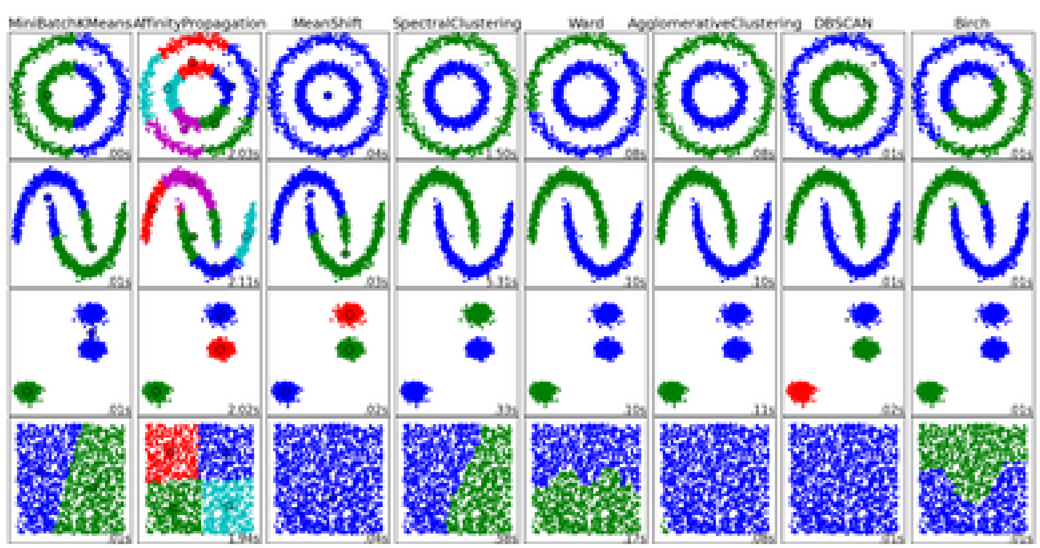


Figure 42.193: *Comparing different clustering algorithms on toy datasets*

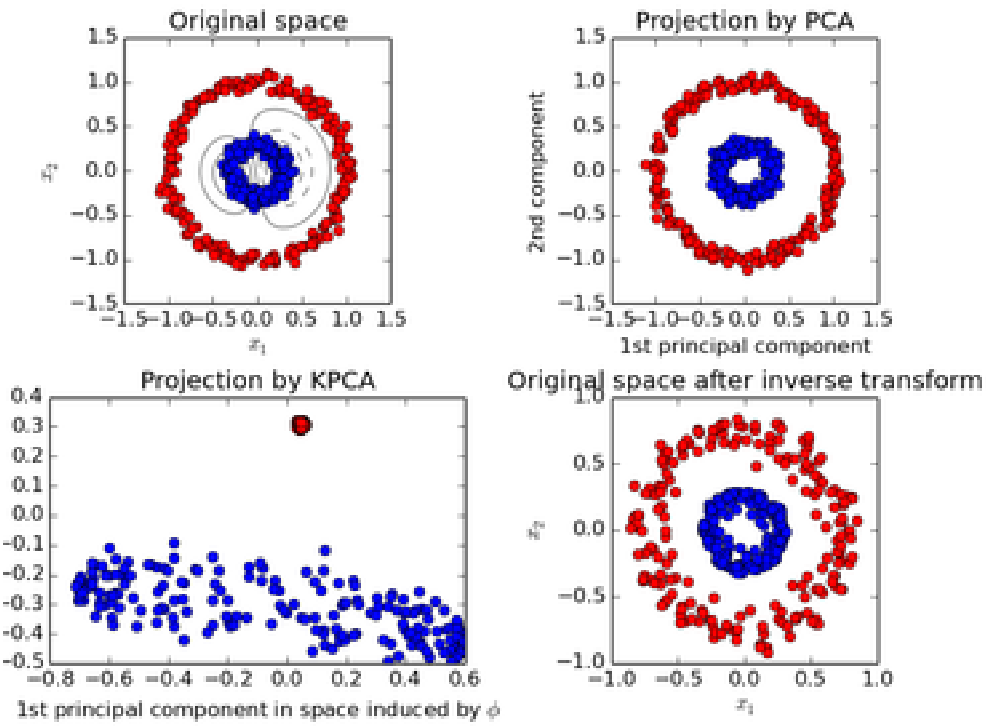


Figure 42.194: *Kernel PCA*

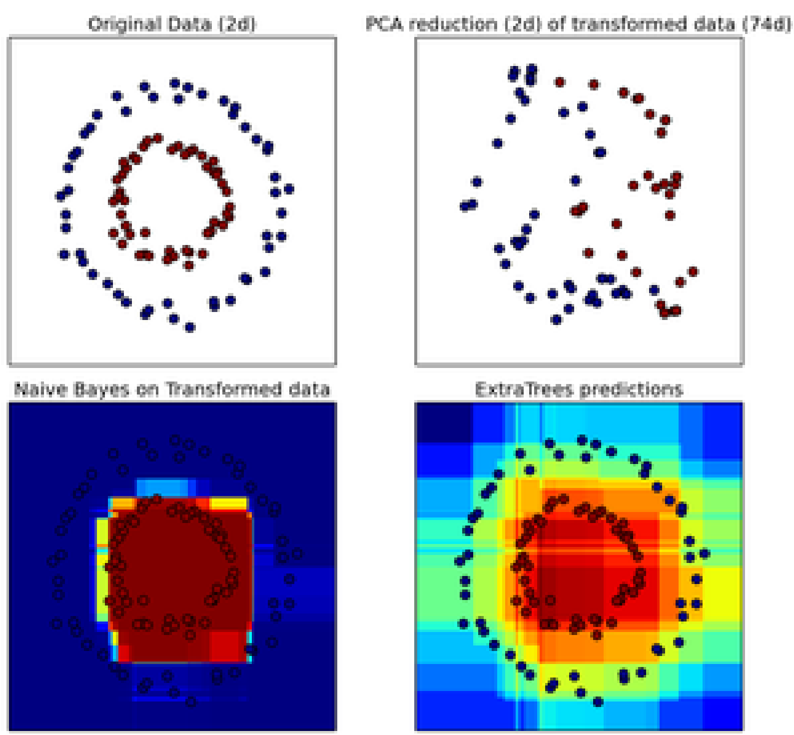


Figure 42.195: *Hashing feature transformation using Totally Random Trees*

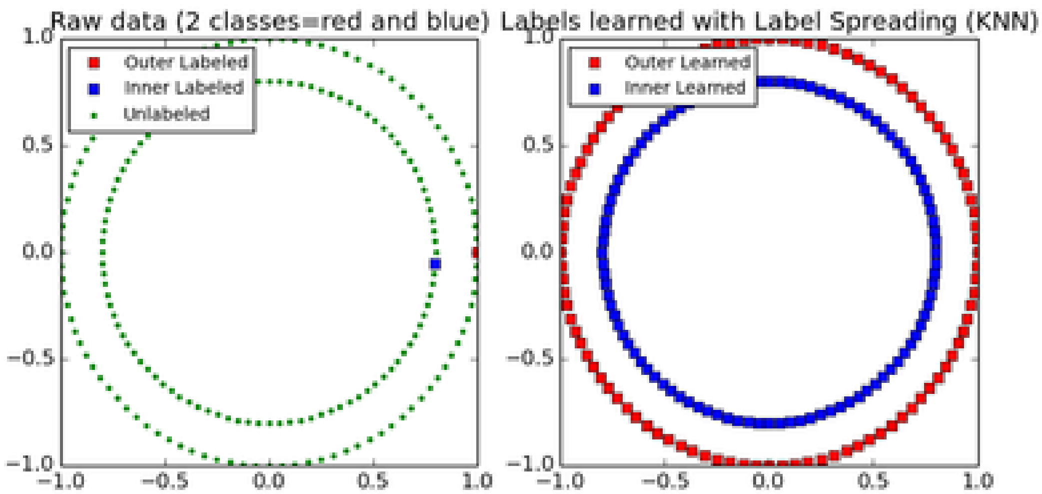


Figure 42.196: *Label Propagation learning a complex structure*

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, 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*.

ReturnsX : array of shape [n_samples, n_features]

The input samples.

y : array of shape [n_samples]

The output values.

References

[R110], [R111]

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
```

Parametersn_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, 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*.

ReturnsX : array of shape [n_samples, 4]

The input samples.

y : array of shape [n_samples]

The output values.

References

[R112], [R113]

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).
```

Parametersn_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, 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*.

ReturnsX : array of shape [n_samples, 4]

The input samples.

y : array of shape [n_samples]

The output values.

References

[R114], [R115]

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).

Parametersmean : 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, 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*.

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

[R116]

Examples using `sklearn.datasets.make_gaussian_quantiles`

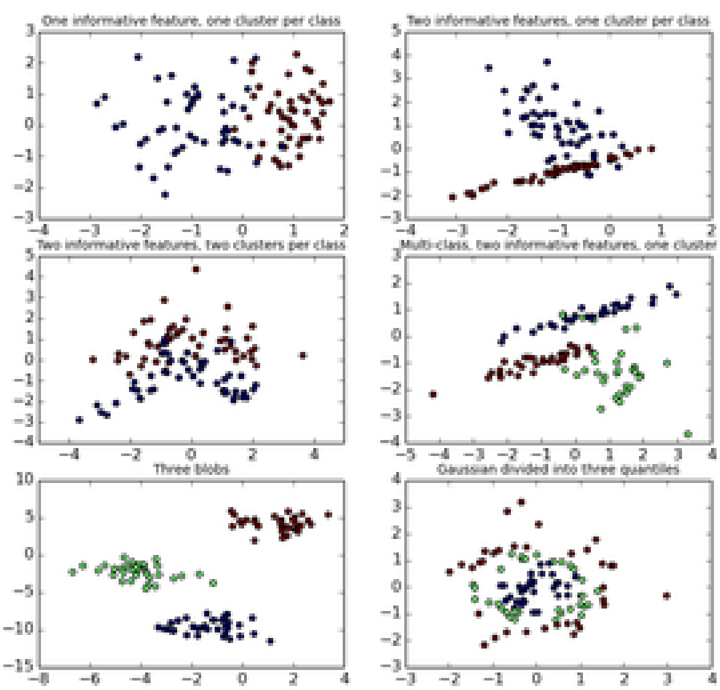


Figure 42.197: *Plot randomly generated classification dataset*

`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:

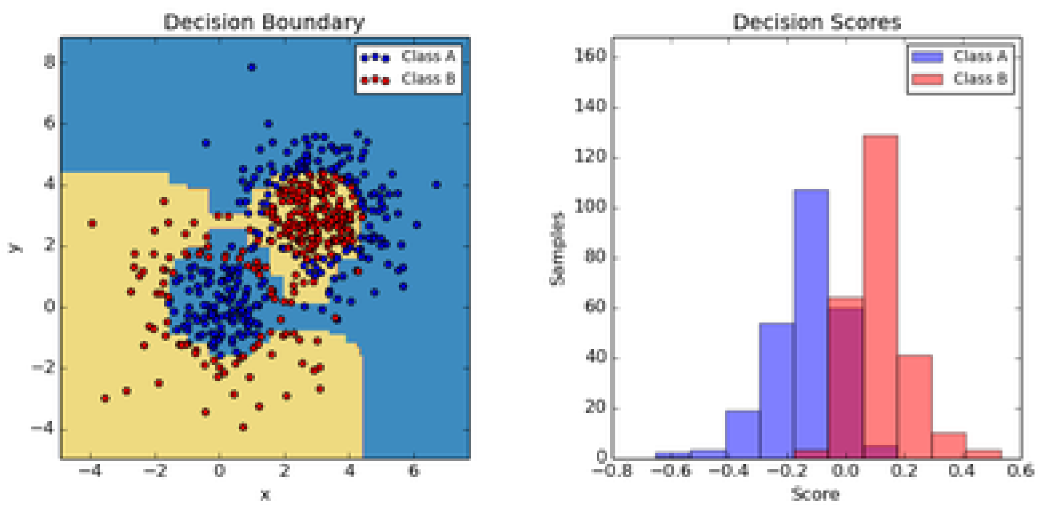


Figure 42.198: *Two-class AdaBoost*

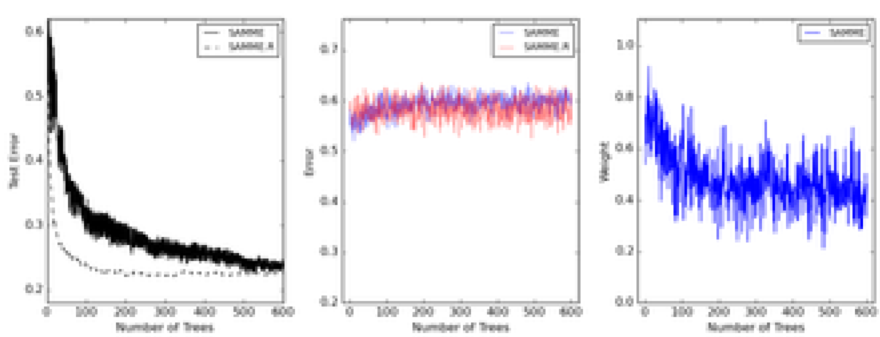


Figure 42.199: *Multi-class AdaBoosted Decision Trees*

```
y[i] = 1 if np.sum(X[i] ** 2) > 9.34 else -1
```

Parametersn_samples : int, optional (default=12000)

The number of samples.

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*.

ReturnsX : 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

[R117]

Examples using `sklearn.datasets.make_hastie_10_2`

`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

Parametersn_samples : int, optional (default=100)

The number of samples.

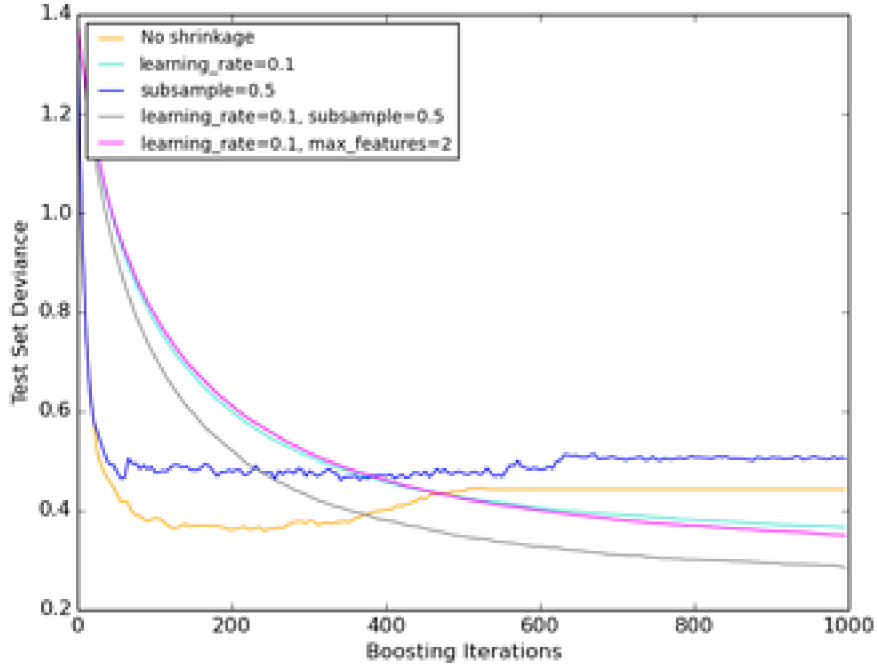


Figure 42.200: *Gradient Boosting regularization*

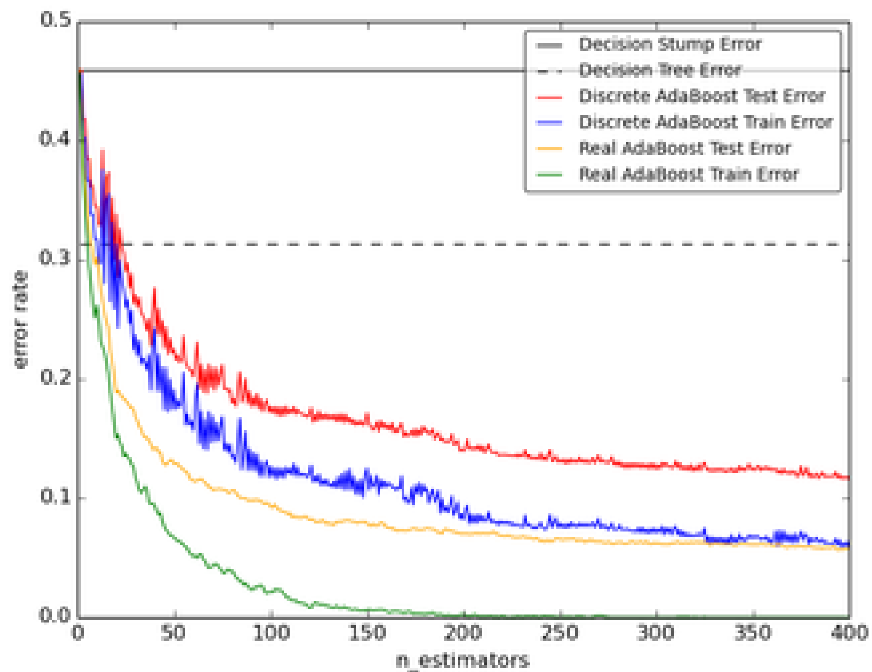


Figure 42.201: *Discrete versus Real AdaBoost*

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, 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*.

ReturnsX : 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.

Parametersn_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.

ReturnsX : 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

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=False*, *return_distributions=False*, *random_state=None*)

Generate a random multilabel classification problem.

For each sample, the generative process is:

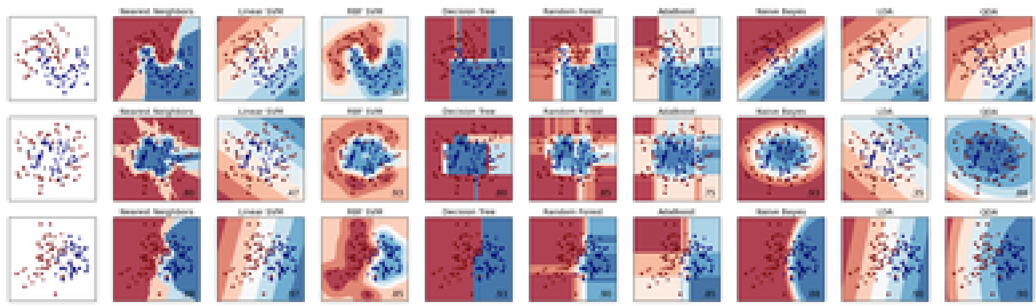


Figure 42.202: *Classifier comparison*

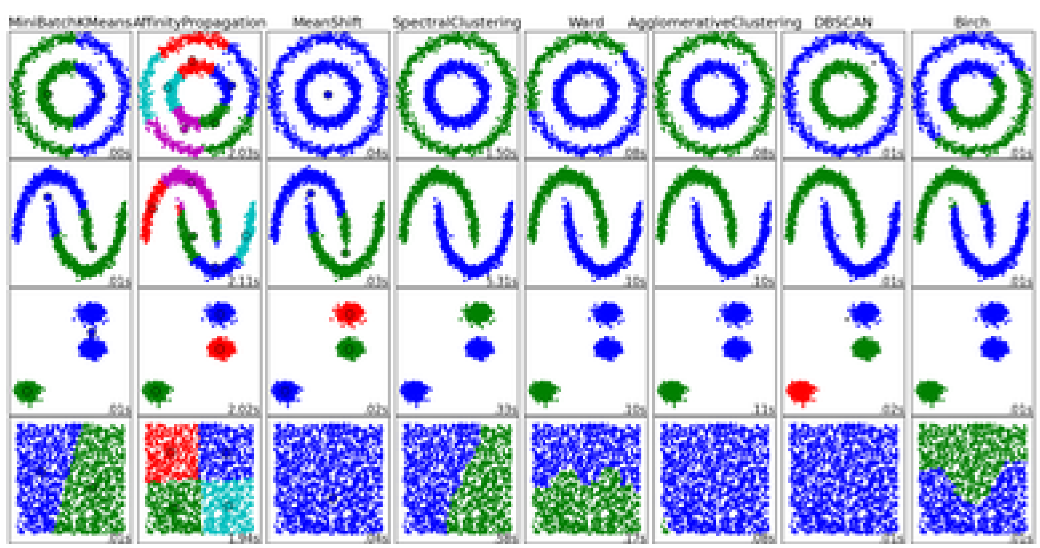


Figure 42.203: *Comparing different clustering algorithms on toy datasets*

- 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.

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

return_indicator : bool, optional (default=False),

If `True`, return `Y` in the binary indicator format, else return a tuple 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, 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*.

Returns

X : array or sparse CSR matrix of shape [n_samples, n_features]

The generated samples.

Y : tuple of lists or array 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`

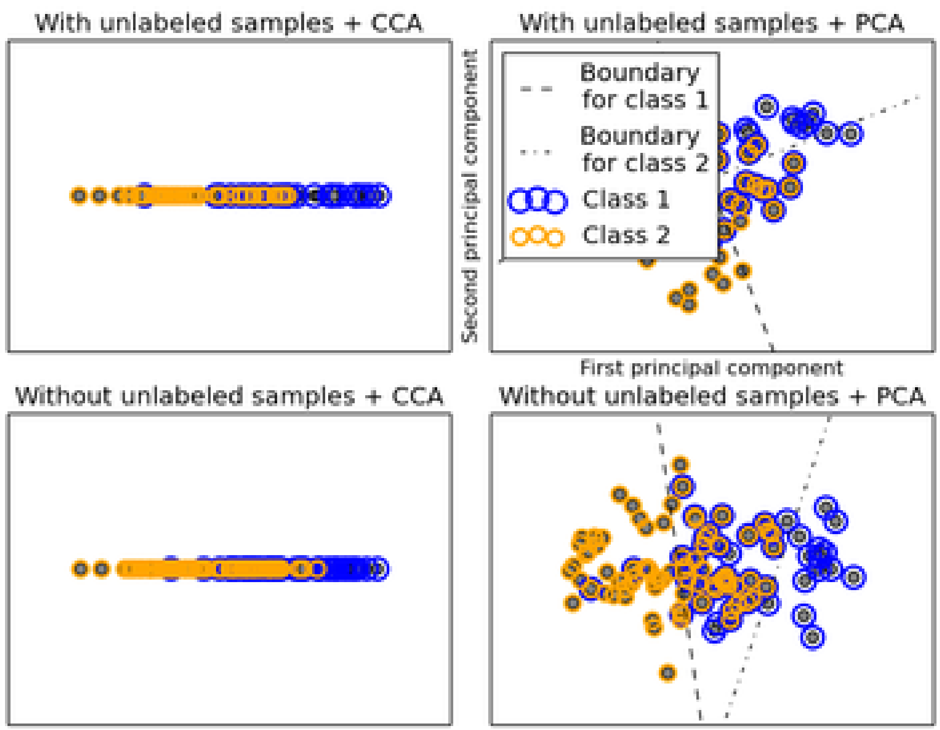


Figure 42.204: *Multilabel classification*

`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.

Parametersn_samples : int, optional (default=100)

The number of samples.

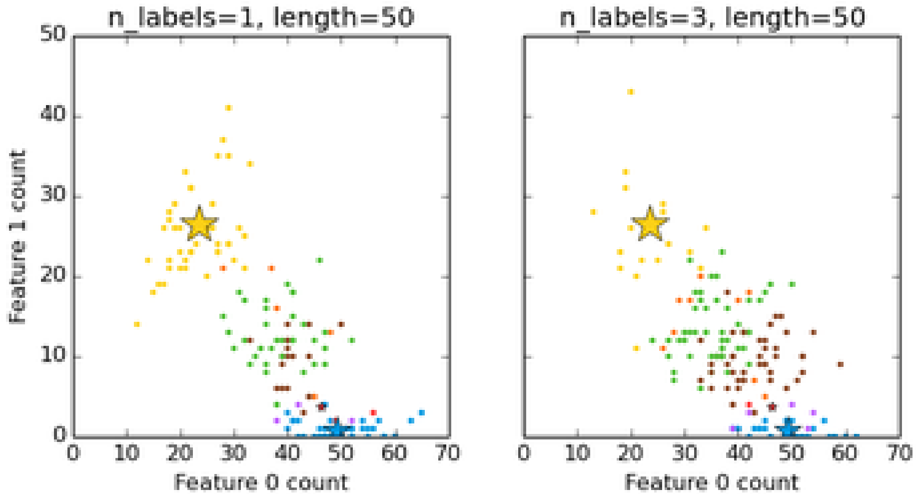


Figure 42.205: *Plot randomly generated multilabel dataset*

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, 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*.

ReturnsX : 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`

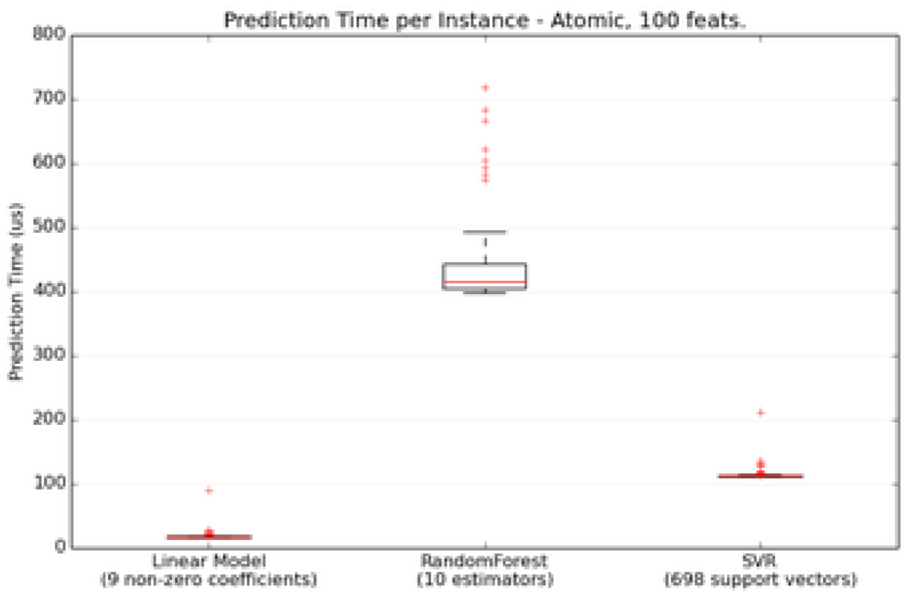


Figure 42.206: *Prediction Latency*

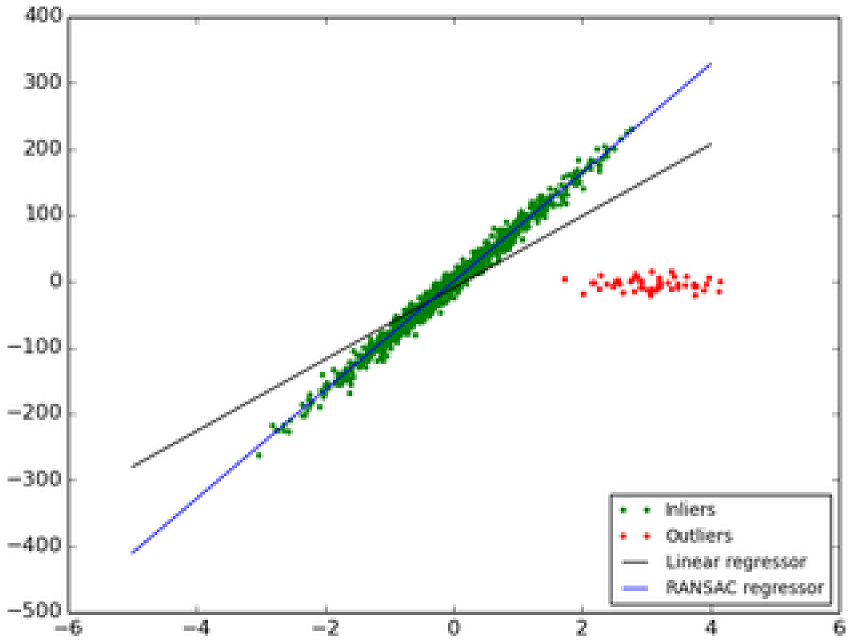


Figure 42.207: *Robust linear model estimation using RANSAC*

Figure 42.208: *Lasso on dense and sparse data*

sklearn.datasets.make_s_curve

sklearn.datasets.**make_s_curve**(*n_samples=100*, *noise=0.0*, *random_state=None*)

Generate an S curve dataset.

Parametersn_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, 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*.

ReturnsX : 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

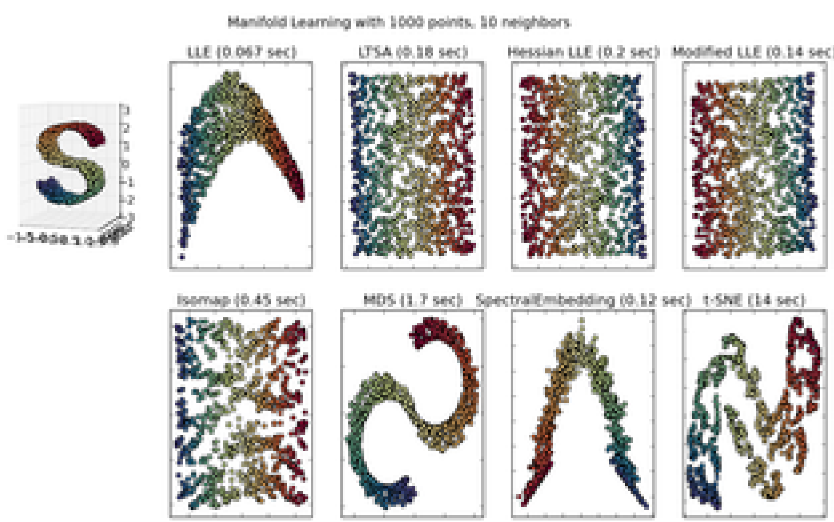


Figure 42.209: *Comparison of Manifold Learning methods*

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.

Parametersn_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 or RandomState instance, optional (default=None) :

seed used by the pseudo random number generator

Returnsdata: 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

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.

Parametersdim: 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 non zero (see notes).

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*.

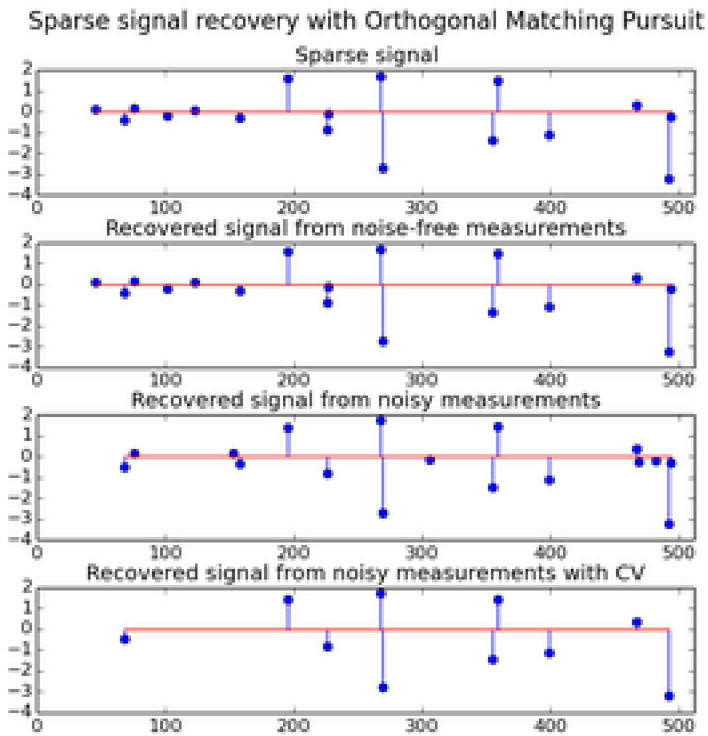


Figure 42.210: *Orthogonal Matching Pursuit*

largest_coef : float between 0 and 1, optional (default=0.9)

The value of the largest coefficient.

smallest_coef : float between 0 and 1, optional (default=0.1)

The value of the smallest coefficient.

norm_diag : boolean, optional (default=False)

Whether to normalize the output matrix to make the leading diagonal elements all 1

Returnsprec : 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.

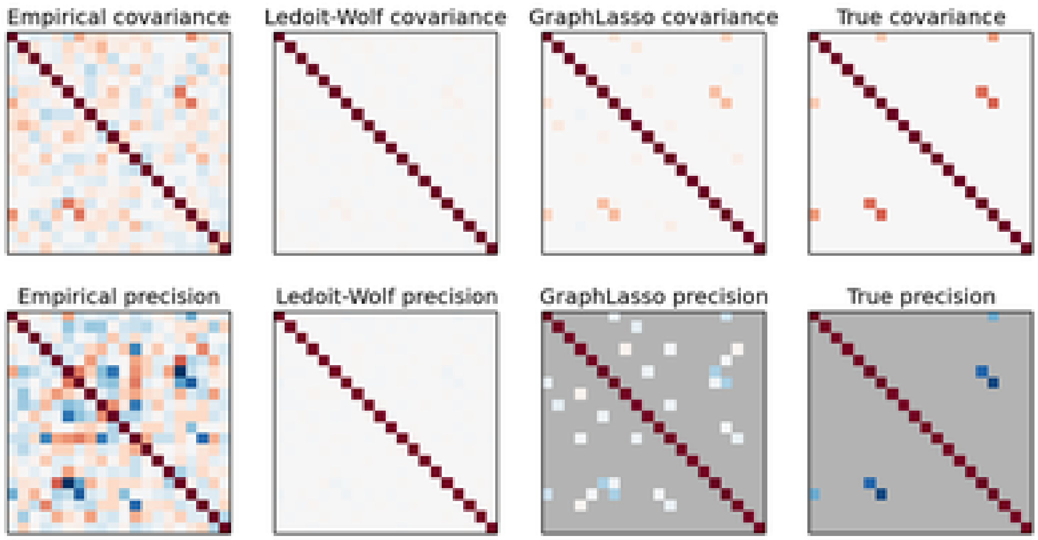


Figure 42.211: *Sparse inverse covariance estimation*

Examples using `sklearn.datasets.make_sparse_spd_matrix`

`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.

Parametersn_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, 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*.

ReturnsX : array of shape [n_samples, n_features]

The input samples.

y : array of shape [n_samples]

The output values.

References

[R118]

sklearn.datasets.make_spd_matrix

sklearn.datasets.**make_spd_matrix**(*n_dim*, *random_state=None*)

Generate a random symmetric, positive-definite matrix.

Parametersn_dim : int

The matrix dimension.

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*.

ReturnsX : 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.

Parametersn_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, 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*.

ReturnsX : 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

[R119]

Examples using `sklearn.datasets.make_swiss_roll`

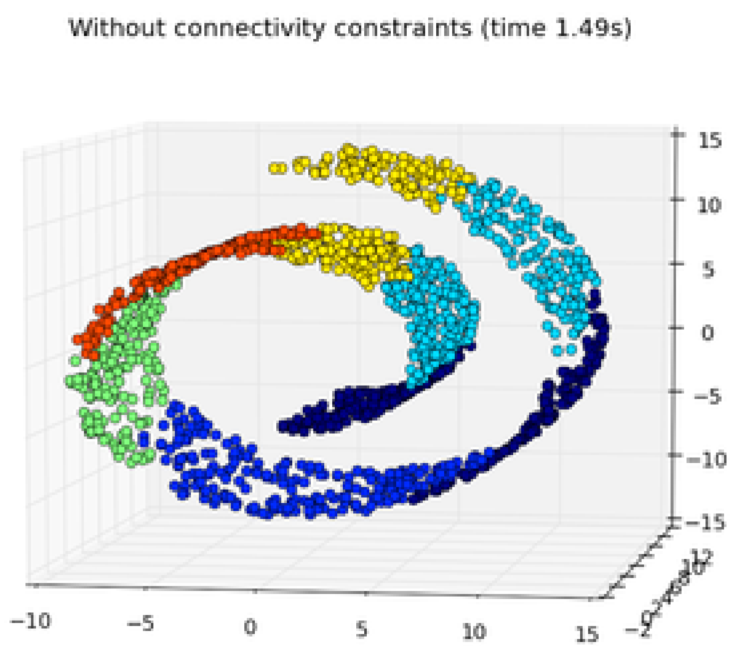


Figure 42.212: *Hierarchical clustering: structured vs unstructured ward*

`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.

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.

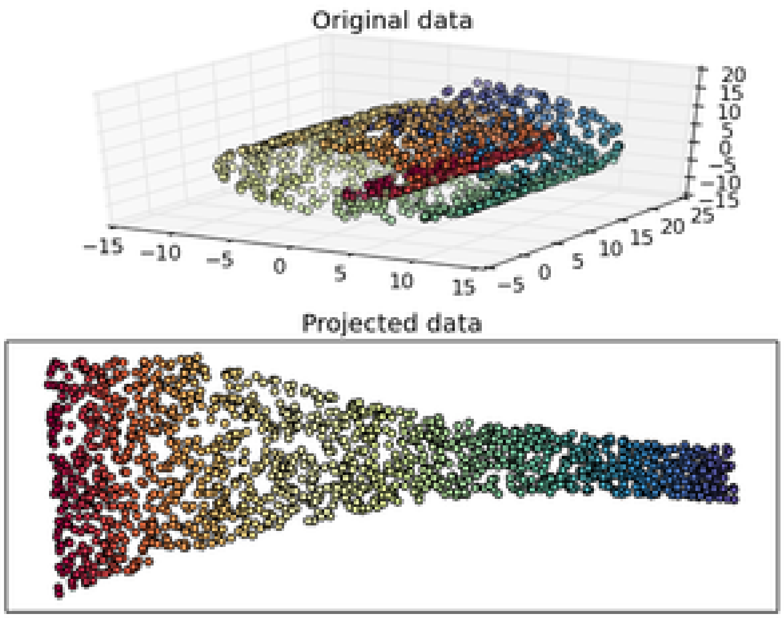


Figure 42.213: *Swiss Roll reduction with LLE*

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, 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*.

ReturnsX : 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

[R107]

Examples using `sklearn.datasets.make_biclusters`

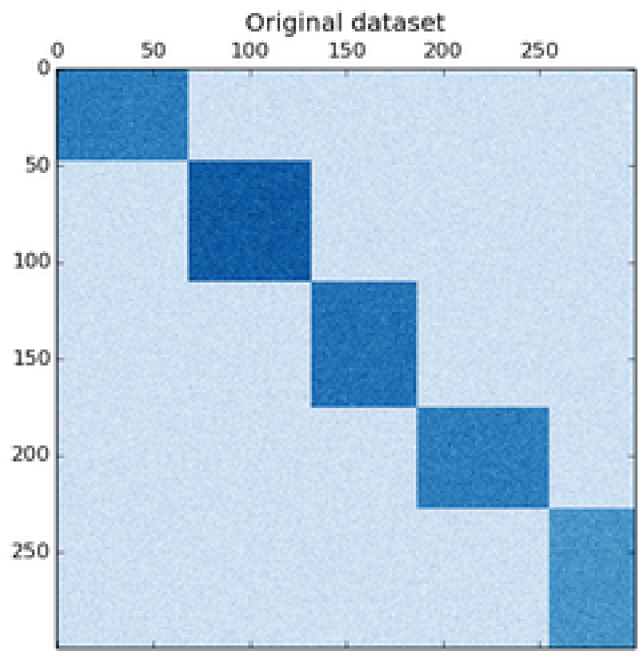


Figure 42.214: *A demo of the Spectral Co-Clustering algorithm*

`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.

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, 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*.

ReturnsX : 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

[R108]

Examples using `sklearn.datasets.make_checkerboard`

42.7 `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.PCA`([n_components, copy, whiten])	Principal component analysis (PCA)
`decomposition.IncrementalPCA`([n_components, ...])	Incremental principal components analysis (IPCA).
`decomposition.ProjectedGradientNMF`([...])	Non-Negative matrix factorization by Projected Gradient (NMF)
`decomposition.RandomizedPCA`([n_components, ...])	Principal component analysis (PCA) using randomized SVD
`decomposition.KernelPCA`([n_components, ...])	Kernel Principal component analysis (KPCA)
`decomposition.FactorAnalysis`([n_components, ...])	Factor Analysis (FA)
`decomposition.FastICA`([n_components, ...])	FastICA: a fast algorithm for Independent Component Analysis.
`decomposition.TruncatedSVD`([n_components, ...])	Dimensionality reduction using truncated SVD (aka LSA).
`decomposition.NMF`([n_components, init, ...])	Non-Negative matrix factorization by Projected Gradient (NMF)
`decomposition.SparsePCA`([n_components, ...])	Sparse Principal Components Analysis (SparsePCA)
`decomposition.MiniBatchSparsePCA`([...])	Mini-batch Sparse Principal Components Analysis
`decomposition.SparseCoder`(dictionary[, ...])	Sparse coding
`decomposition.DictionaryLearning`([...])	Dictionary learning
`decomposition.MiniBatchDictionaryLearning`([...])	Mini-batch dictionary learning

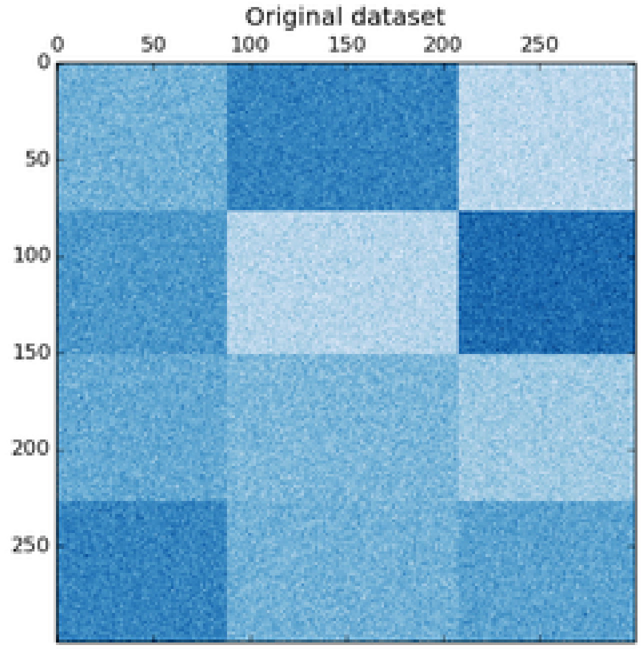


Figure 42.215: *A demo of the Spectral Biclustering algorithm*

42.7.1 sklearn.decomposition.PCA

class `sklearn.decomposition.`**PCA**(*n_components=None*, *copy=True*, *whiten=False*)

Principal component analysis (PCA)

Linear dimensionality reduction using Singular Value Decomposition of the data and keeping only the most significant singular vectors to project the data to a lower dimensional space.

This implementation uses the scipy.linalg implementation of the singular value decomposition. It only works for dense arrays and is not scalable to large dimensional data.

The time complexity of this implementation is `O(n ** 3)` assuming n ~ n_samples ~ n_features.

Parametersn_components : int, None or string

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', Minka's MLE is used to guess the dimension if `0 < n_components < 1`, 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

copy : bool

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

When True (False by default) the *components_* vectors are divided by n_samples times 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 there data respect some hard-wired assumptions.

Attributescomponents_ : array, [n_components, n_features]

Components with maximum variance.

explained_variance_ratio_ : array, [n_components]

Percentage of variance explained by each of the selected components. k is not set then all components are stored and the sum of explained variances is equal to 1.0

mean_ : array, [n_features]

Per-feature empirical mean, estimated from the training set.

n_components_ : int

The estimated number of components. Relevant when n_components is set to 'mle' or a number between 0 and 1 to select using explained variance.

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 computed the estimated data covariance and score samples.

See also:

`RandomizedPCA`, `KernelPCA`, `SparsePCA`, `TruncatedSVD`

Notes

For n_components='mle', this class uses the method of *Thomas P. Minka: Automatic Choice of Dimensionality for PCA. NIPS 2000: 598-604*

Implements the probabilistic PCA model from: M. Tipping and C. Bishop, Probabilistic Principal Component Analysis, Journal of the Royal Statistical Society, Series B, 61, Part 3, pp. 611-622 via the score and score_samples methods. See http://www.miketipping.com/papers/met-mppca.pdf

Due to implementation subtleties of the Singular Value Decomposition (SVD), which is used in this implementation, running fit twice on the same matrix can lead to principal components with signs flipped (change in direction). For this reason, it is important to always use the same estimator object to transform data in a consistent fashion.

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(copy=True, n_components=2, whiten=False)
>>> print(pca.explained_variance_ratio_)
[ 0.99244...  0.00755...]
```

Methods

`fit`(X[, y])	Fit the model with X.
`fit_transform`(X[, y])	Fit the model with X and apply the dimensionality reduction on X.
`get_covariance`()	Compute data covariance with the generative model.
`get_params`([deep])	Get parameters for this estimator.
`get_precision`()	Compute data precision matrix with the generative model.
`inverse_transform`(X)	Transform data back to its original space, i.e.,
`score`(X[, y])	Return the average log-likelihood of all samples
`score_samples`(X)	Return the log-likelihood of each sample
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Apply the dimensionality reduction on X.

static `__init__`(*n_components=None*, *copy=True*, *whiten=False*)

static `fit`(*X*, *y=None*)

Fit the model with X.

ParametersX: 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.

Returnsself : object

Returns the instance itself.

static `fit_transform`(*X*, *y=None*)

Fit the model with X and apply the dimensionality reduction on X.

ParametersX : 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.

ReturnsX_new : array-like, shape (n_samples, n_components)

static `get_covariance`()

Compute data covariance with the generative model.

`cov = components_.T * S**2 * components_ + sigma2 * eye(n_features)` where S**2 contains the explained variances.

Returnscov : array, shape=(n_features, n_features)

Estimated covariance of data.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static `get_precision`()

Compute data precision matrix with the generative model.

Equals the inverse of the covariance but computed with the matrix inversion lemma for efficiency.

Returnsprecision : array, shape=(n_features, n_features)

Estimated precision of data.

static inverse_transform(*X*)

Transform data back to its original space, i.e., return an input X_original whose transform would be X

ParametersX : 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.

ReturnsX_original array-like, shape (n_samples, n_features) :

static score(*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

ParametersX: array, shape(n_samples, n_features) :

The data.

Returnsll: float :

Average log-likelihood of the samples under the current model

static score_samples(*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

ParametersX: array, shape(n_samples, n_features) :

The data.

Returnsll: array, shape (n_samples,) :

Log-likelihood of each sample under the current model

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static transform(*X*)

Apply the dimensionality reduction on X.

X is projected on the first principal components previous extracted from a training set.

ParametersX : 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.

ReturnsX_new : array-like, shape (n_samples, n_components)

Examples using `sklearn.decomposition.PCA`

Figure 42.216: *Concatenating multiple feature extraction methods*

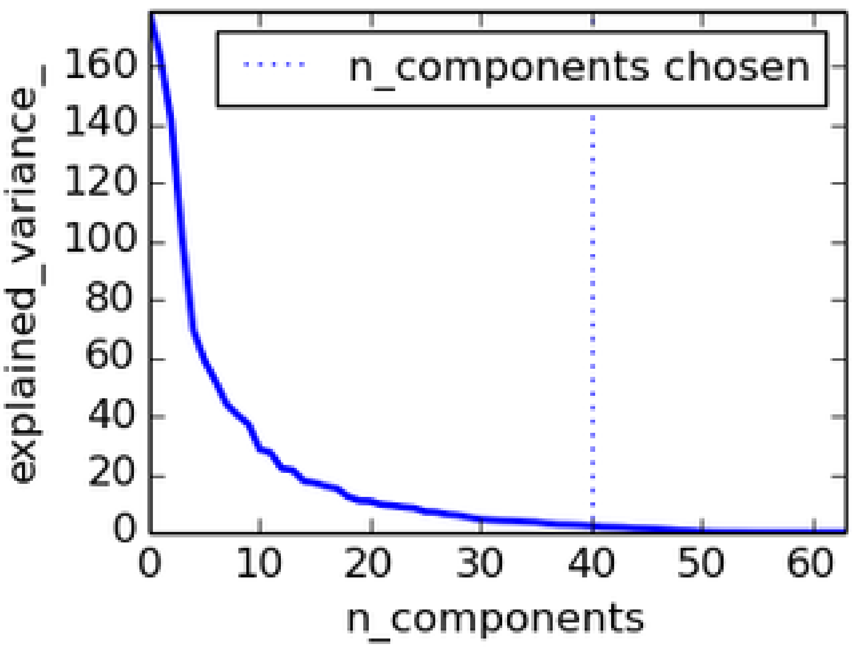


Figure 42.217: *Pipelining: chaining a PCA and a logistic regression*

42.7.2 `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 centered data, keeping only the most significant singular vectors to project the data to a lower dimensional space.

Depending on the size of the input data, this algorithm can be much more memory efficient than a PCA.

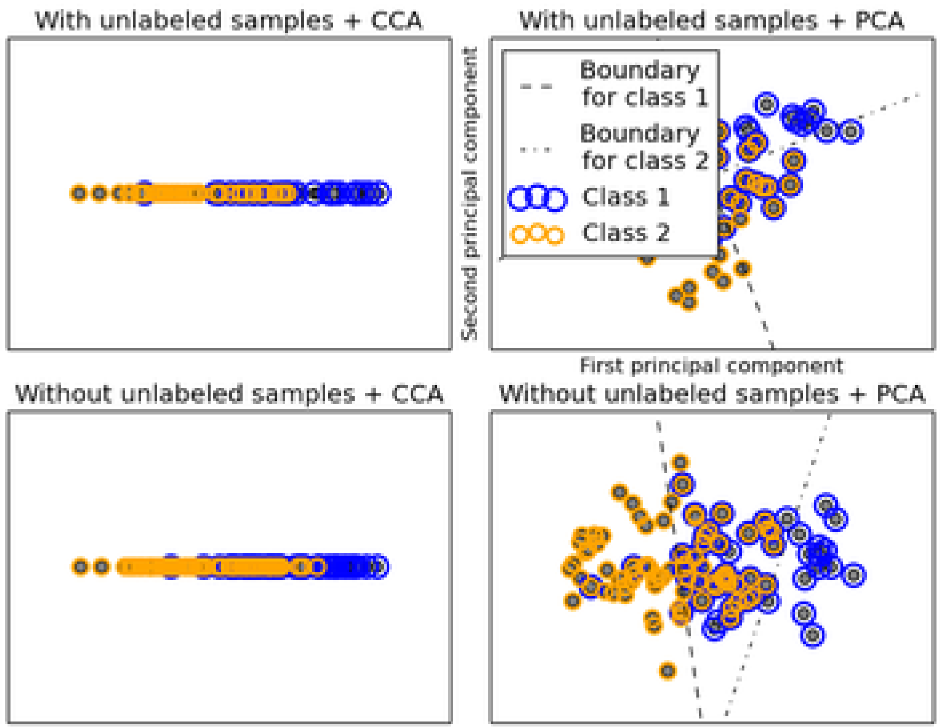


Figure 42.218: *Multilabel classification*

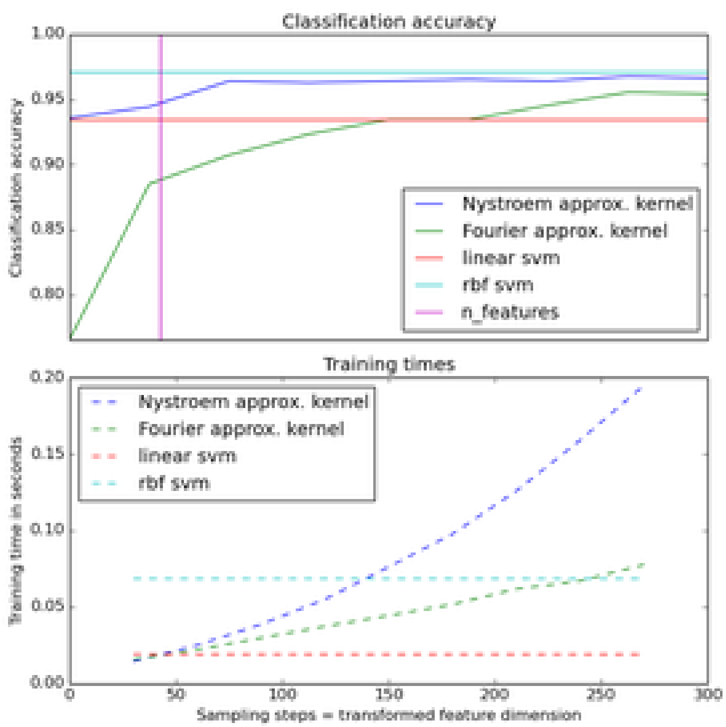


Figure 42.219: *Explicit feature map approximation for RBF kernels*

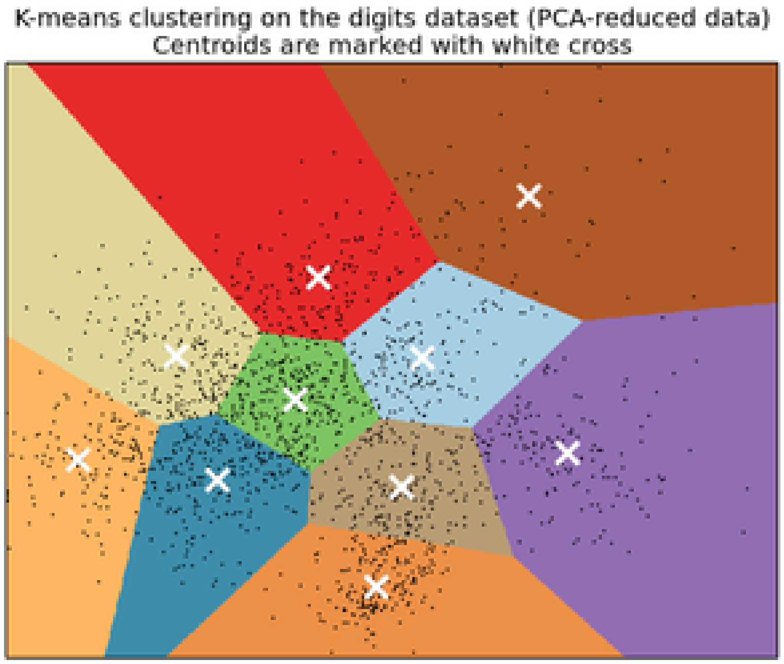


Figure 42.220: *A demo of K-Means clustering on the handwritten digits data*

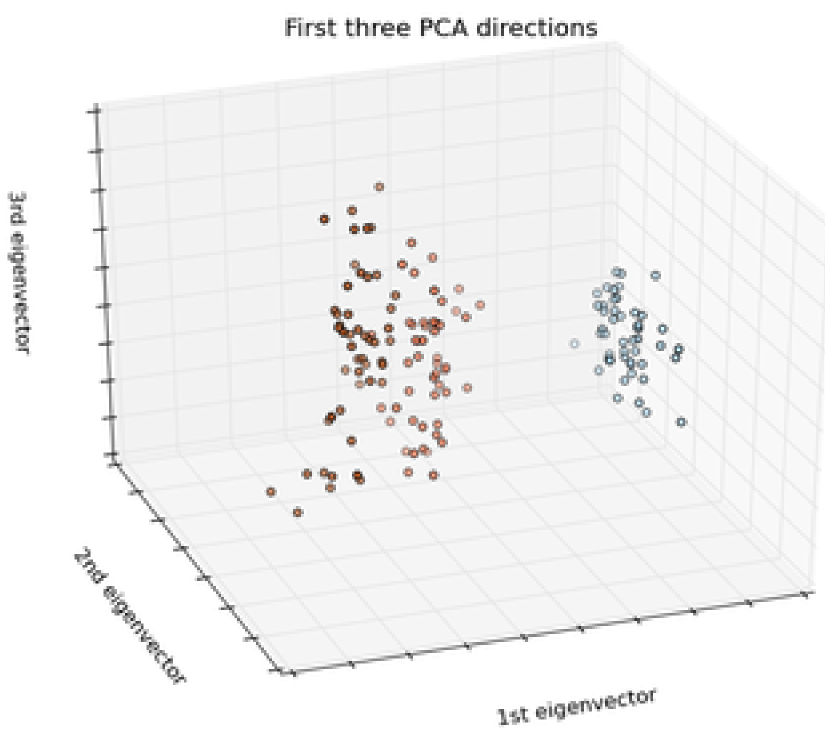


Figure 42.221: *The Iris Dataset*

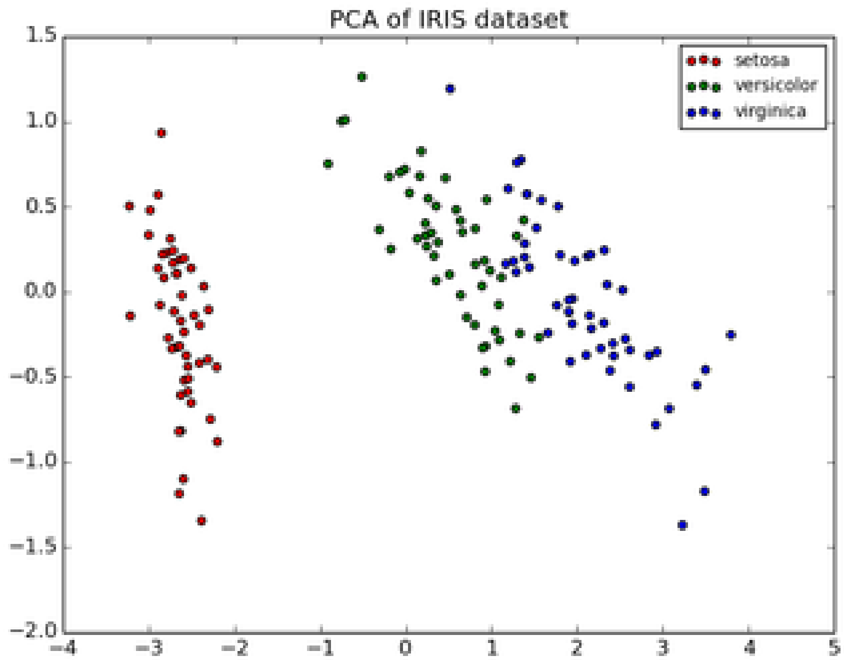


Figure 42.222: *Comparison of LDA and PCA 2D projection of Iris dataset*

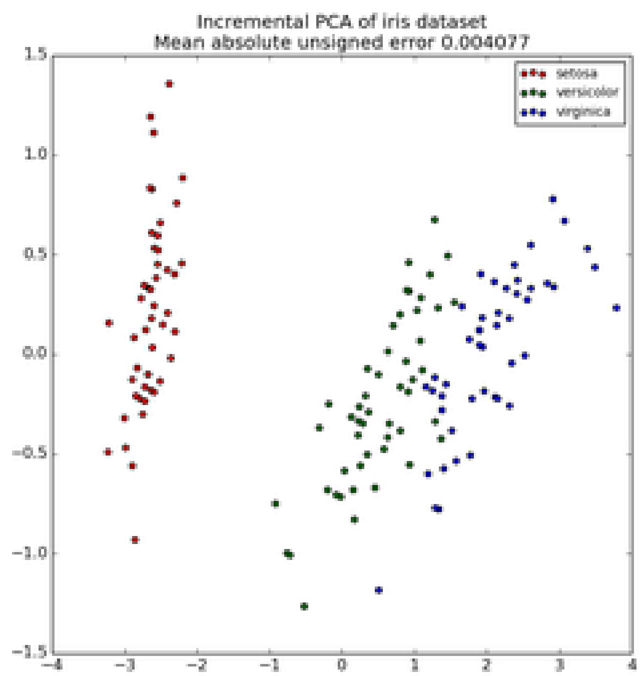


Figure 42.223: *Incremental PCA*

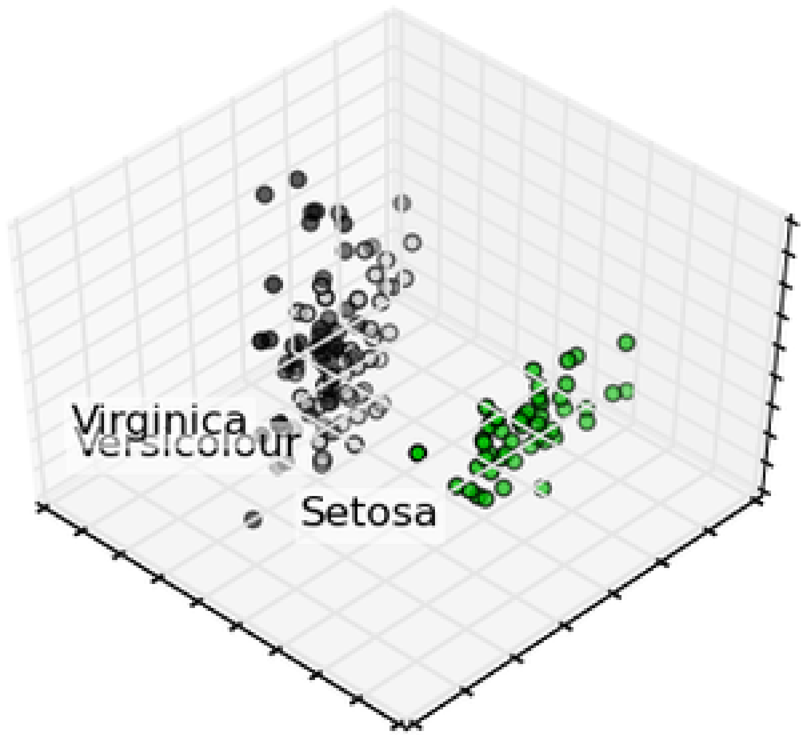


Figure 42.224: *PCA example with Iris Data-set*

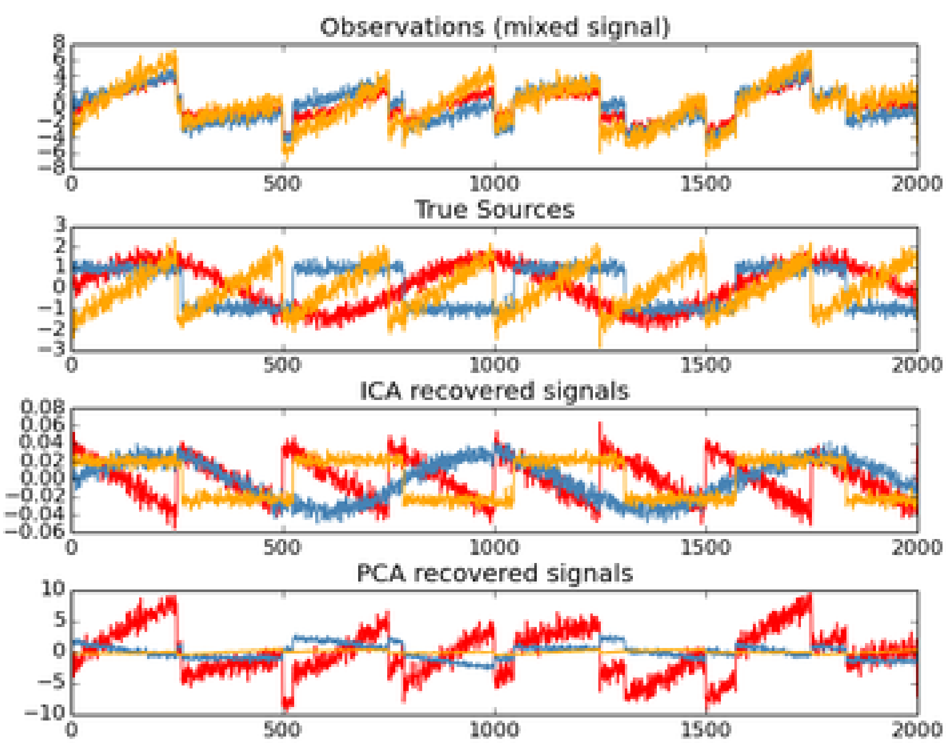


Figure 42.225: *Blind source separation using FastICA*

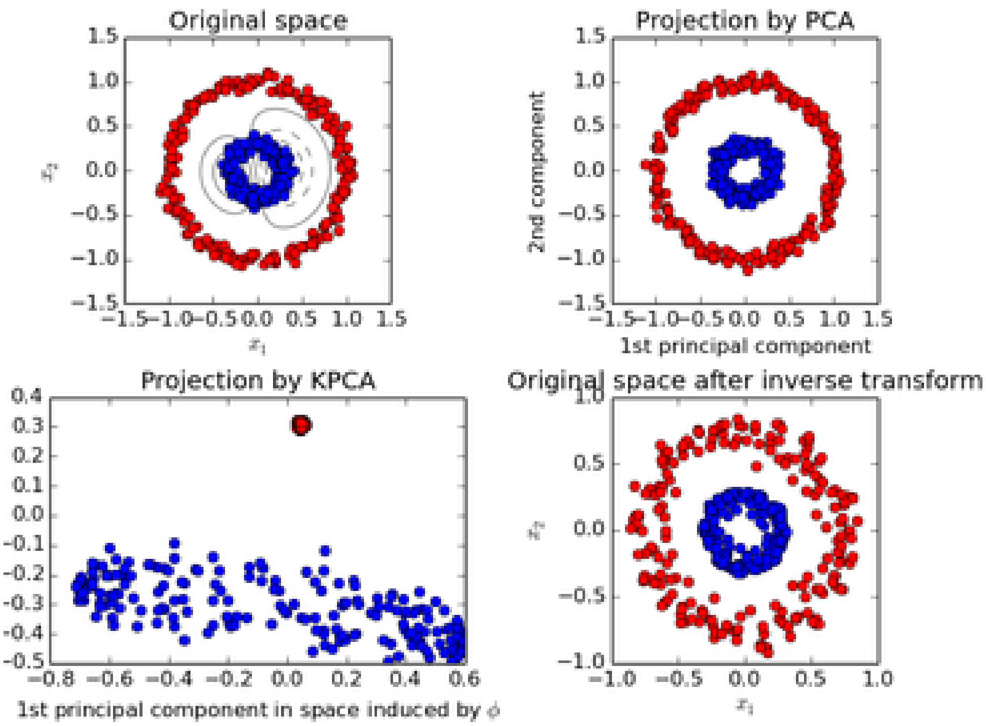


Figure 42.226: *Kernel PCA*

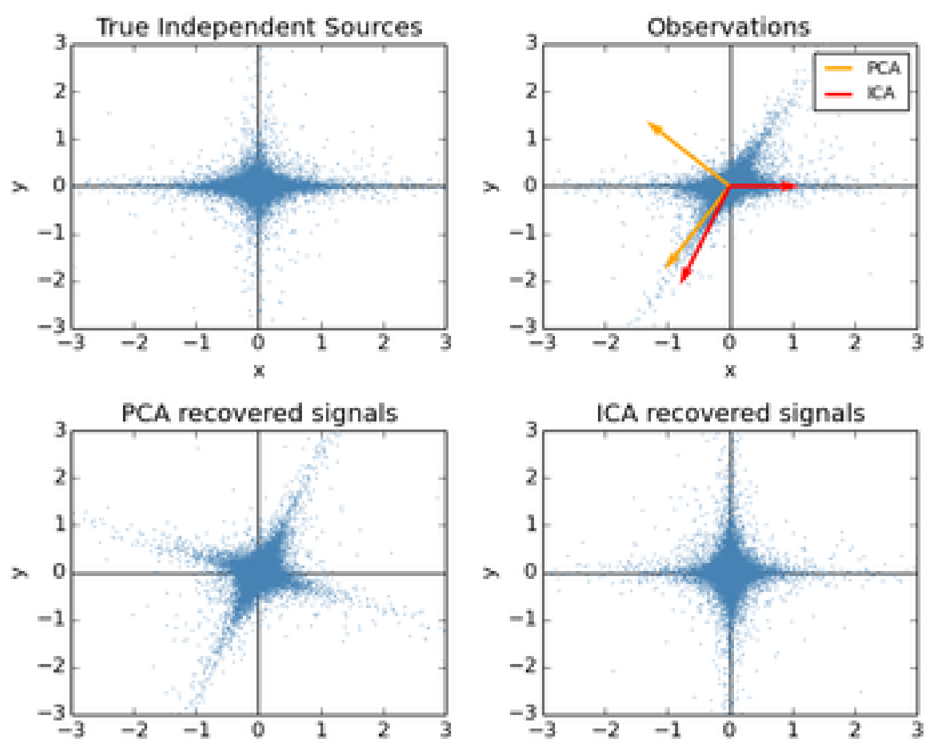


Figure 42.227: *FastICA on 2D point clouds*

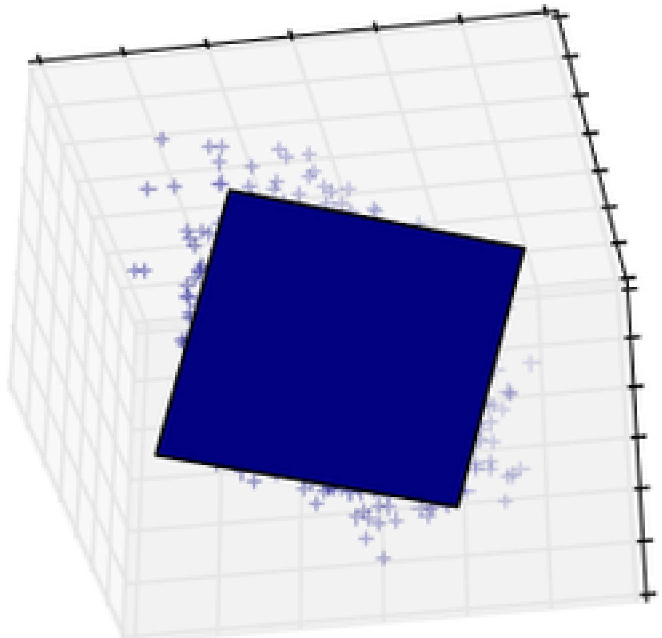


Figure 42.228: *Principal components analysis (PCA)*

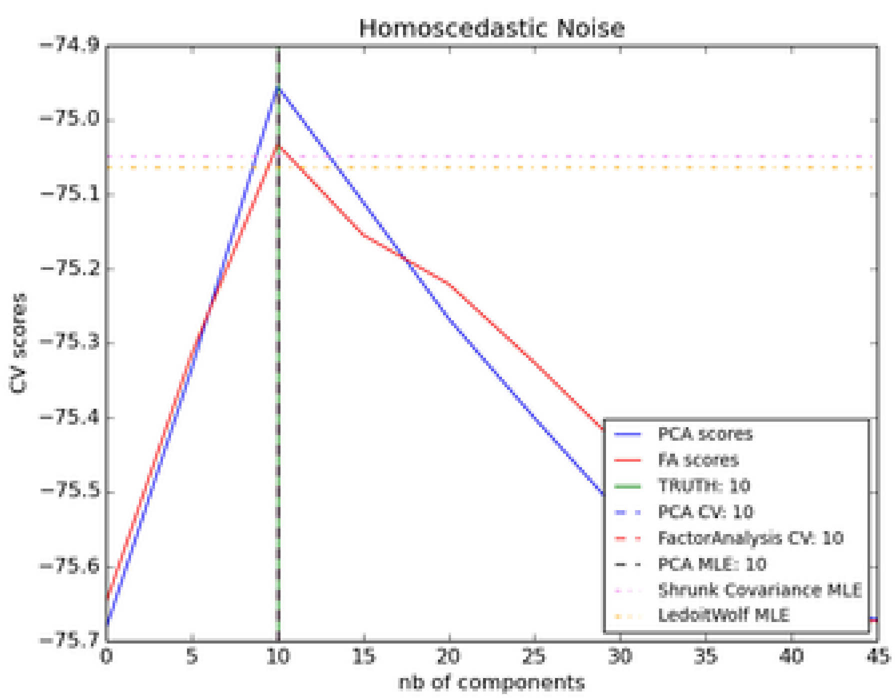


Figure 42.229: *Model selection with Probabilistic (PCA) and Factor Analysis (FA)*

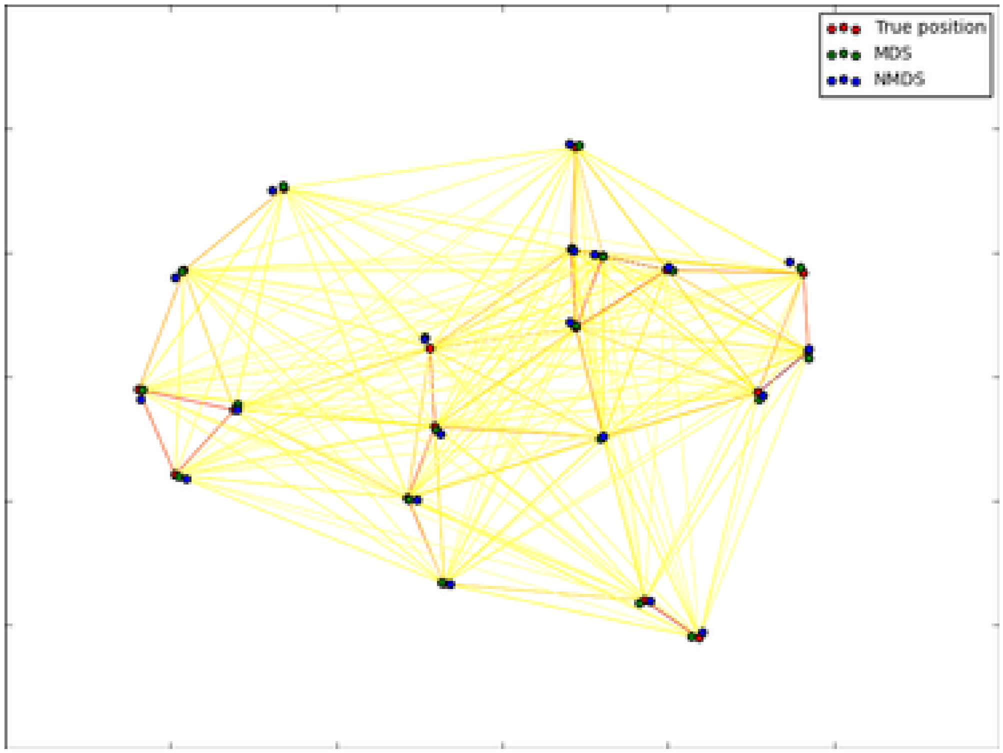


Figure 42.230: *Multi-dimensional scaling*

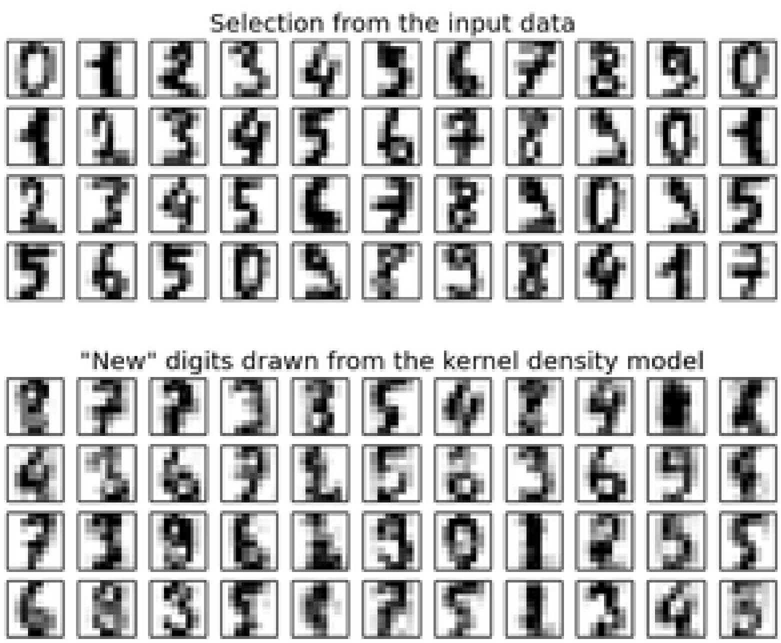


Figure 42.231: *Kernel Density Estimation*

This algorithm has constant memory complexity, on the order of `batch_size`, enabling use of np.memmap files without loading the entire file into memory.

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.

Parametersn_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)`.

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.

copy : bool, (default=True)

If False, X will be overwritten. `copy=False` can be used to save memory but is unsafe for general use.

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.

Attributescomponents_ : 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

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`, `RandomizedPCA`, `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 http://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 http://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

4. **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.

7. **Golub and C. Van Loan. Matrix Computations, Third Edition, Chapter 5,**Section 5.4.4, pp. 252-253.

Methods

`fit`(X[, y])	Fit the model with X, using minibatches of size batch_size.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_covariance`()	Compute data covariance with the generative model.
`get_params`([deep])	Get parameters for this estimator.
`get_precision`()	Compute data precision matrix with the generative model.
`inverse_transform`(X[, y])	Transform data back to its original space.
`partial_fit`(X[, y])	Incremental fit with X.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y])	Apply dimensionality reduction to X.

static `__init__`(*n_components=None*, *whiten=False*, *copy=True*, *batch_size=None*)

static `fit`(*X*, *y=None*)

Fit the model with X, using minibatches of size batch_size.

ParametersX: 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: Passthrough for ''Pipeline'' compatibility. :

Returnsself: object :

Returns the instance itself.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_covariance()

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.

Returnscov : array, shape=(n_features, n_features)

Estimated covariance of data.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_precision()

Compute data precision matrix with the generative model.

Equals the inverse of the covariance but computed with the matrix inversion lemma for efficiency.

Returnsprecision : array, shape=(n_features, n_features)

Estimated precision of data.

inverse_transform(*X*, *y=None*)

Transform data back to its original space.

In other words, return an input X_original whose transform would be X.

ParametersX : 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.

ReturnsX_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.

static partial_fit(*X*, *y=None*)

Incremental fit with X. All of X is processed as a single batch.

ParametersX: 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.

Returnsself: object :

Returns the instance itself.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*, *y=None*)

Apply dimensionality reduction to X.

X is projected on the first principal components previously extracted from a training set.

ParametersX : 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.

ReturnsX_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, copy=True, n_components=2, whiten=False)
>>> ipca.transform(X)
```

Examples using `sklearn.decomposition.IncrementalPCA`

42.7.3 `sklearn.decomposition`.ProjectedGradientNMF

class sklearn.decomposition.**ProjectedGradientNMF**(*n_components=None*, *init=None*, *sparseness=None*, *beta=1*, *eta=0.1*, *tol=0.0001*, *max_iter=200*, *nls_max_iter=2000*, *random_state=None*)

Non-Negative matrix factorization by Projected Gradient (NMF)

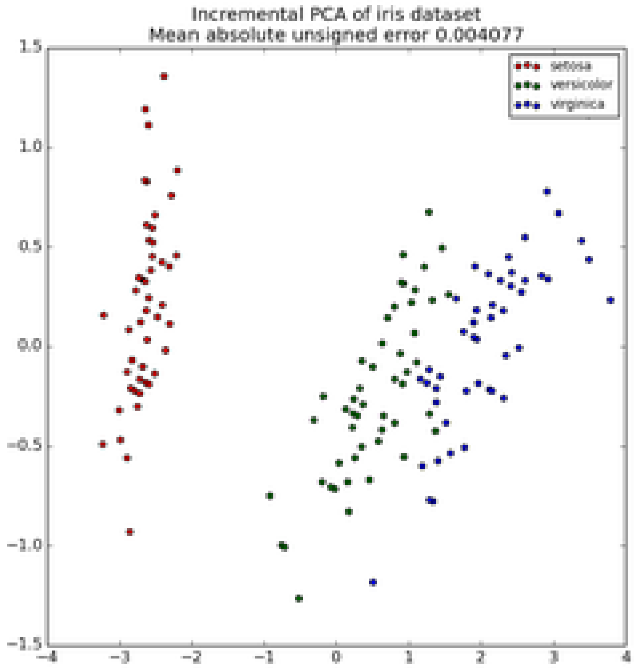


Figure 42.232: *Incremental PCA*

Parametersn_components : int or None

Number of components, if n_components is not set all components are kept

init : 'nndsvd' | 'nndsvda' | 'nndsvdar' | 'random'

Method used to initialize the procedure. Default: 'nndsvdar' if n_components < n_features, otherwise random. Valid options:

```
'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)
'random': non-negative random matrices
```

sparseness : 'data' | 'components' | None, default: None

Where to enforce sparsity in the model.

beta : double, default: 1

Degree of sparseness, if sparseness is not None. Larger values mean more sparseness.

eta : double, default: 0.1

Degree of correctness to maintain, if sparsity is not None. Smaller values mean larger error.

tol : double, default: 1e-4

Tolerance value used in stopping conditions.

max_iter : int, default: 200

Number of iterations to compute.

nls_max_iter : int, default: 2000

Number of iterations in NLS subproblem.

random_state : int or RandomState

Random number generator seed control.

Attributescomponents_ : array, [n_components, n_features]

Non-negative components of the data.

reconstruction_err_ : number

Frobenius norm of the matrix difference between the training data and the reconstructed data from the fit produced by the model. `|| X - WH ||_2`

n_iter_ : int

Number of iterations run.

References

This implements

C.-J. Lin. Projected gradient methods for non-negative matrix factorization. Neural Computation, 19(2007), 2756-2779. http://www.csie.ntu.edu.tw/~cjlin/nmf/

P. Hoyer. Non-negative Matrix Factorization with Sparseness Constraints. Journal of Machine Learning Research 2004.

NNDSVD is introduced in

C. Boutsidis, E. Gallopoulos: SVD based initialization: A head start for nonnegative matrix factorization - Pattern Recognition, 2008 http://tinyurl.com/nndsvd

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 ProjectedGradientNMF
>>> model = ProjectedGradientNMF(n_components=2, init='random',
...                              random_state=0)
>>> model.fit(X)
ProjectedGradientNMF(beta=1, eta=0.1, init='random', max_iter=200,
        n_components=2, nls_max_iter=2000, random_state=0, sparseness=None,
        tol=0.0001)
>>> model.components_
array([[ 0.77032744,  0.11118662],
       [ 0.38526873,  0.38228063]])
>>> model.reconstruction_err_
0.00746...
>>> model = ProjectedGradientNMF(n_components=2,
...              sparseness='components', init='random', random_state=0)
>>> model.fit(X)
```

```
ProjectedGradientNMF(beta=1, eta=0.1, init='random', max_iter=200,
            n_components=2, nls_max_iter=2000, random_state=0,
            sparseness='components', tol=0.0001)
>>> model.components_
array([[ 1.67481991,  0.29614922],
       [ 0.        ,  0.4681982 ]])
>>> model.reconstruction_err_
0.513...
```

Methods

`fit`(X[, y])	Learn a NMF model for the data X.
`fit_transform`(X[, y])	Learn a NMF model for the data X and returns the transformed data.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Transform the data X according to the fitted NMF model

static `__init__`(*n_components=None, init=None, sparseness=None, beta=1, eta=0.1, tol=0.0001, max_iter=200, nls_max_iter=2000, random_state=None*)

static `fit`(*X, y=None, **params*)

Learn a NMF model for the data X.

ParametersX: {array-like, sparse matrix}, shape = [n_samples, n_features] :

Data matrix to be decomposed

Returnsself :

static `fit_transform`(*X, y=None*)

Learn a NMF model for the data X and returns the transformed data.

This is more efficient than calling fit followed by transform.

ParametersX: {array-like, sparse matrix}, shape = [n_samples, n_features] :

Data matrix to be decomposed

Returnsdata: array, [n_samples, n_components] :

Transformed data

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static `transform`(*X*)

Transform the data X according to the fitted NMF model

ParametersX: {array-like, sparse matrix}, shape = [n_samples, n_features] :

Data matrix to be transformed by the model

Returnsdata: array, [n_samples, n_components] :

Transformed data

42.7.4 `sklearn.decomposition`.RandomizedPCA

class `sklearn.decomposition.`**`RandomizedPCA`**(*n_components=None*, *copy=True*, *iterated_power=3*, *whiten=False*, *random_state=None*)

Principal component analysis (PCA) using randomized SVD

Linear dimensionality reduction using approximated Singular Value Decomposition of the data and keeping only the most significant singular vectors to project the data to a lower dimensional space.

Parametersn_components : int, optional

Maximum number of components to keep. When not given or None, this is set to n_features (the second dimension of the training data).

copy : bool

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.

iterated_power : int, optional

Number of iterations for the power method. 3 by default.

whiten : bool, optional

When True (False by default) the *components_* vectors are 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.

random_state : int or RandomState instance or None (default)

Pseudo Random Number generator seed control. If None, use the numpy.random singleton.

Attributescomponents_ : array, [n_components, n_features]

Components with maximum variance.

explained_variance_ratio_ : array, [n_components]

Percentage of variance explained by each of the selected components. k is not set then all components are stored and the sum of explained variances is equal to 1.0

mean_ : array, [n_features]

Per-feature empirical mean, estimated from the training set.

See also:

PCA, TruncatedSVD

References

[Halko2009], [MRT]

Examples

```
>>> import numpy as np
>>> from sklearn.decomposition import RandomizedPCA
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> pca = RandomizedPCA(n_components=2)
>>> pca.fit(X)
RandomizedPCA(copy=True, iterated_power=3, n_components=2,
       random_state=None, whiten=False)
>>> print(pca.explained_variance_ratio_)
[ 0.99244...  0.00755...]
```

Methods

fit(X[, y])	Fit the model with X by extracting the first principal components.
fit_transform(X[, y])	Fit the model with X and apply the dimensionality reduction on X.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X[, y])	Transform data back to its original space.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Apply dimensionality reduction on X.

static `__init__`(*n_components=None*, *copy=True*, *iterated_power=3*, *whiten=False*, *random_state=None*)

static `fit`(*X*, *y=None*)

Fit the model with X by extracting the first principal components.

ParametersX: 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.

Returnsself : object

Returns the instance itself.

static `fit_transform`(*X*, *y=None*)

Fit the model with X and apply the dimensionality reduction on X.

ParametersX : 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.

ReturnsX_new : array-like, shape (n_samples, n_components)

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static inverse_transform(*X*, *y=None*)

Transform data back to its original space.

Returns an array X_original whose transform would be X.

ParametersX : array-like, shape (n_samples, n_components)

New data, where n_samples in the number of samples and n_components is the number of components.

ReturnsX_original array-like, shape (n_samples, n_features) :

Notes

If whitening is enabled, inverse_transform does not compute the exact inverse operation of transform.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static transform(*X*, *y=None*)

Apply dimensionality reduction on X.

X is projected on the first principal components previous extracted from a training set.

ParametersX : 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.

ReturnsX_new : array-like, shape (n_samples, n_components)

Examples using `sklearn.decomposition.RandomizedPCA`

42.7.5 `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*)

Kernel Principal component analysis (KPCA)

Non-linear dimensionality reduction through the use of kernels (see *Pairwise metrics, Affinities and Kernels*).

Parametersn_components: int or None :

Number of components. If None, all non-zero components are kept.

Figure 42.233: *Faces recognition example using eigenfaces and SVMs*

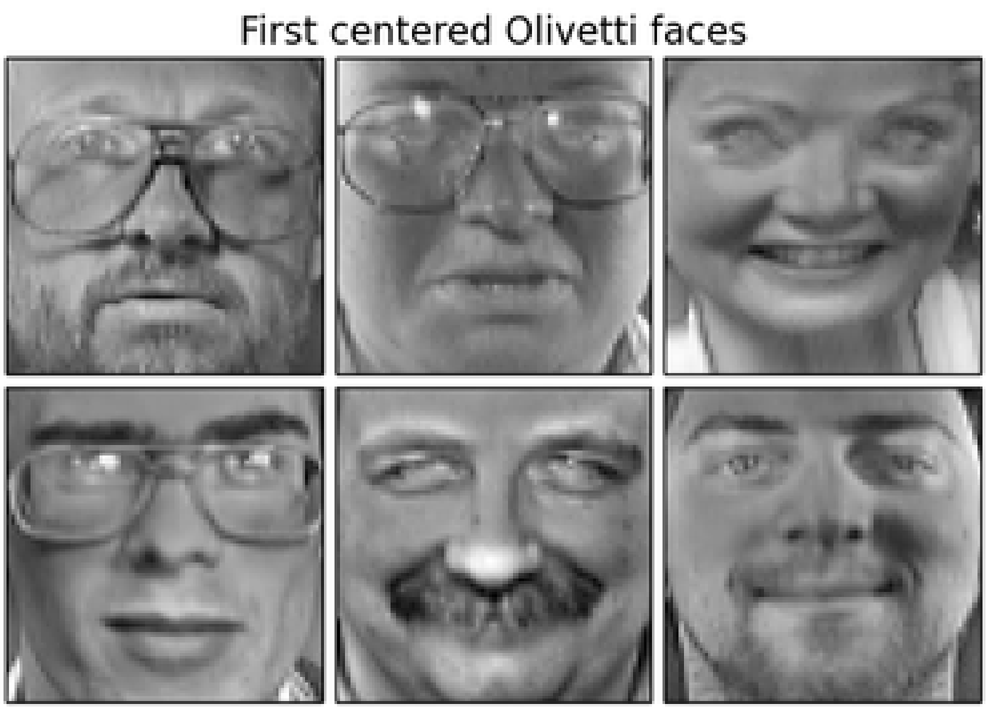


Figure 42.234: *Faces dataset decompositions*

kernel: "linear" | "poly" | "rbf" | "sigmoid" | "cosine" | "precomputed" :

Kernel. Default: "linear"

degree : int, default=3

Degree for poly kernels. Ignored by other kernels.

gamma : float, optional

Kernel coefficient for rbf and poly kernels. Default: 1/n_features. Ignored by other kernels.

coef0 : float, optional

Independent term in poly and sigmoid kernels. Ignored by other kernels.

kernel_params : mapping of string to any, optional

Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.

alpha: int :

Hyperparameter of the ridge regression that learns the inverse transform (when fit_inverse_transform=True). Default: 1.0

fit_inverse_transform: bool :

Learn the inverse transform for non-precomputed kernels. (i.e. learn to find the pre-image of a point) Default: False

eigen_solver: string ['auto'|'dense'|'arpack'] :

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 :

convergence tolerance for arpack. Default: 0 (optimal value will be chosen by arpack)

max_iter : int

maximum number of iterations for arpack Default: None (optimal value will be chosen by arpack)

remove_zero_eig : boolean, default=True

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.

Attributeslambdas_ : :

Eigenvalues of the centered kernel matrix

alphas_ : :

Eigenvectors of the centered kernel matrix

dual_coef_ : :

Inverse transform matrix

X_transformed_fit_ : :

Projection of the fitted data on the kernel principal components

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.

Methods

`fit`(X[, y])	Fit the model from data in X.
`fit_transform`(X[, y])	Fit the model from data in X and transform X.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(X)	Transform X back to original space.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Transform X.

static `__init__`(*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*)

static `fit`(*X*, *y=None*)

Fit the model from data in X.

ParametersX: 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.

Returnsself : object

Returns the instance itself.

static `fit_transform`(*X*, *y=None*, ***params*)

Fit the model from data in X and transform X.

ParametersX: 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.

ReturnsX_new: array-like, shape (n_samples, n_components) :

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static `inverse_transform`(*X*)

Transform X back to original space.

ParametersX: array-like, shape (n_samples, n_components) :

ReturnsX_new: array-like, shape (n_samples, n_features) :

References

"Learning to Find Pre-Images", G BakIr et al, 2004.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static **transform** (*X*)
Transform X.

ParametersX: array-like, shape (n_samples, n_features) :

ReturnsX_new: array-like, shape (n_samples, n_components) :

Examples using `sklearn.decomposition.KernelPCA`

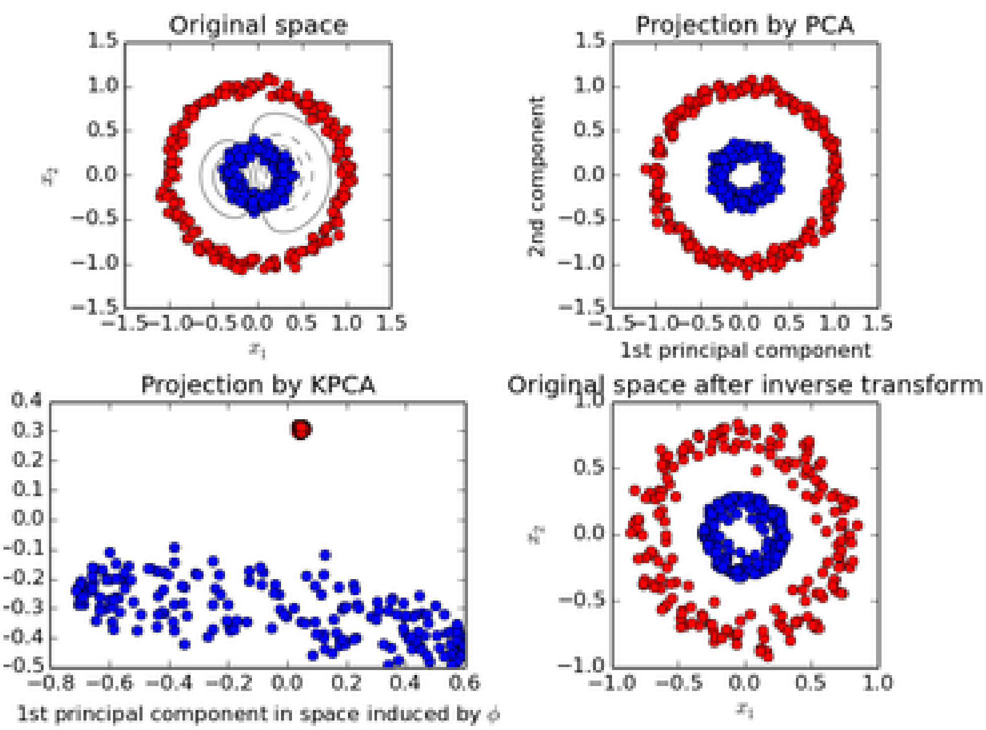


Figure 42.235: *Kernel PCA*

42.7.6 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 expectation-maximization (EM).

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 EM algorithm.

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 or RandomState

Pseudo number generator state used for random sampling. Only used if `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.

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

Methods

`fit`(X[, y])	Fit the FactorAnalysis model to X using EM
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_covariance`()	Compute data covariance with the FactorAnalysis model.
`get_params`([deep])	Get parameters for this estimator.
`get_precision`()	Compute data precision matrix with the FactorAnalysis model.
`score`(X[, y])	Compute the average log-likelihood of the samples
`score_samples`(X)	Compute the log-likelihood of each sample
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Apply dimensionality reduction to X using the model.

static `__init__`(*n_components=None, tol=0.01, copy=True, max_iter=1000, noise_variance_init=None, svd_method='randomized', iterated_power=3, random_state=0*)

static `fit`(*X, y=None*)

Fit the FactorAnalysis model to X using EM

ParametersX : array-like, shape (n_samples, n_features)

Training data.

Returnsself :

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

static **get_covariance**()

Compute data covariance with the FactorAnalysis model.

```
cov = components_.T * components_ + diag(noise_variance)
```

Returnscov : array, shape (n_features, n_features)

Estimated covariance of data.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **get_precision**()

Compute data precision matrix with the FactorAnalysis model.

Returnsprecision : array, shape (n_features, n_features)

Estimated precision of data.

static **score**(*X*, *y=None*)

Compute the average log-likelihood of the samples

ParametersX: array, shape (n_samples, n_features) :

The data

Returnsll: float :

Average log-likelihood of the samples under the current model

static **score_samples**(*X*)

Compute the log-likelihood of each sample

ParametersX: array, shape (n_samples, n_features) :

The data

Returnsll: array, shape (n_samples,) :

Log-likelihood of each sample under the current model

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **transform**(*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).

ParametersX : array-like, shape (n_samples, n_features)

Training data.

ReturnsX_new : array-like, shape (n_samples, n_components)

The latent variables of X.

Examples using `sklearn.decomposition.FactorAnalysis`

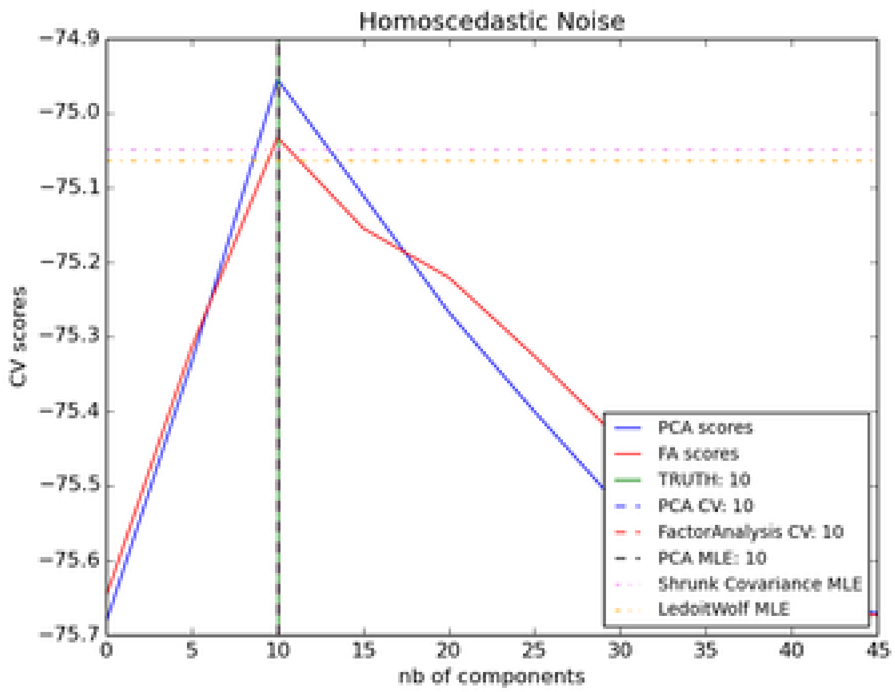


Figure 42.236: *Model selection with Probabilistic (PCA) and Factor Analysis (FA)*

42.7.7 `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.

Parametersn_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

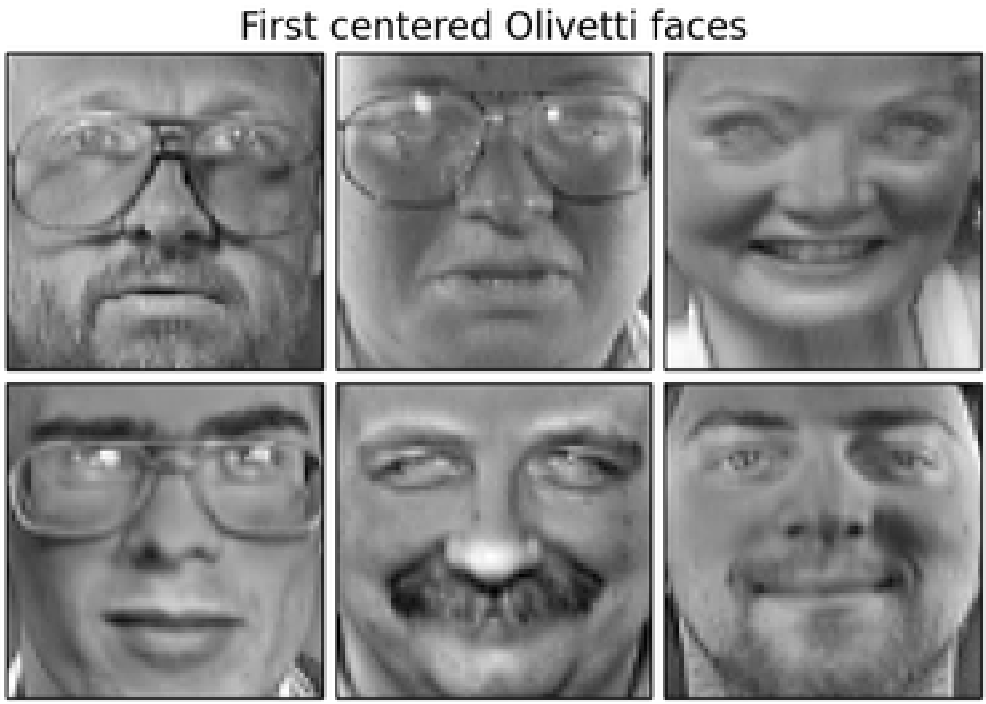


Figure 42.237: *Faces dataset decompositions*

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

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 or RandomState

Pseudo number generator state used for random sampling.

Attributescomponents_ : 2D array, shape (n_components, n_features)

The unmixing matrix.

mixing_ : array, shape (n_features, n_components)

The mixing matrix.

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.

Notes

Implementation based on *A. Hyvarinen and E. Oja, Independent Component Analysis: Algorithms and Applications, Neural Networks, 13(4-5), 2000, pp. 411-430*

Methods

`fit`(X[, y])	Fit the model to X.
`fit_transform`(X[, y])	Fit the model and recover the sources from X.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(X[, copy])	Transform the sources back to the mixed data (apply mixing matrix).
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y, copy])	Recover the sources from X (apply the unmixing matrix).

static **`__init__`**(*n_components=None*, *algorithm='parallel'*, *whiten=True*, *fun='logcosh'*, *fun_args=None*, *max_iter=200*, *tol=0.0001*, *w_init=None*, *random_state=None*)

static **`fit`**(*X*, *y=None*)
Fit the model to X.

ParametersX : 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.

Returnsself :

static **`fit_transform`**(*X*, *y=None*)
Fit the model and recover the sources from X.

ParametersX : 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.

ReturnsX_new : array-like, shape (n_samples, n_components)

`get_params`(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **`inverse_transform`**(*X*, *copy=True*)
Transform the sources back to the mixed data (apply mixing matrix).

ParametersX : 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.

ReturnsX_new : array-like, shape (n_samples, n_features)

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static `transform`(*X*, *y=None*, *copy=True*)

Recover the sources from X (apply the unmixing matrix).

ParametersX : 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.

ReturnsX_new : array-like, shape (n_samples, n_components)

Examples using `sklearn.decomposition.FastICA`

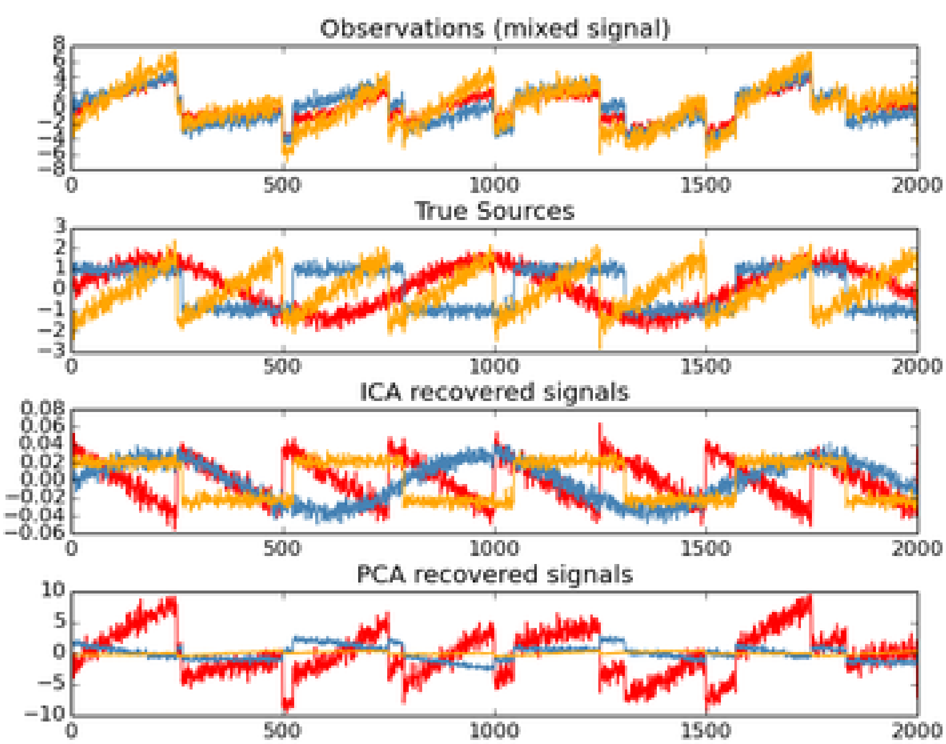


Figure 42.238: *Blind source separation using FastICA*

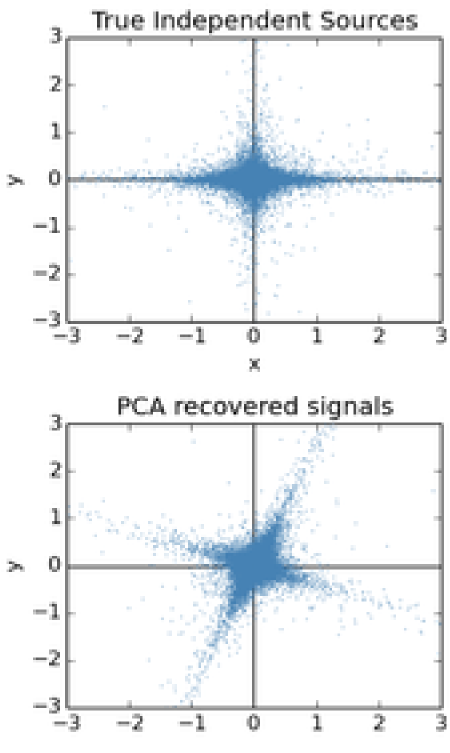


Figure 42.239: *FastICA on 2D point clouds*

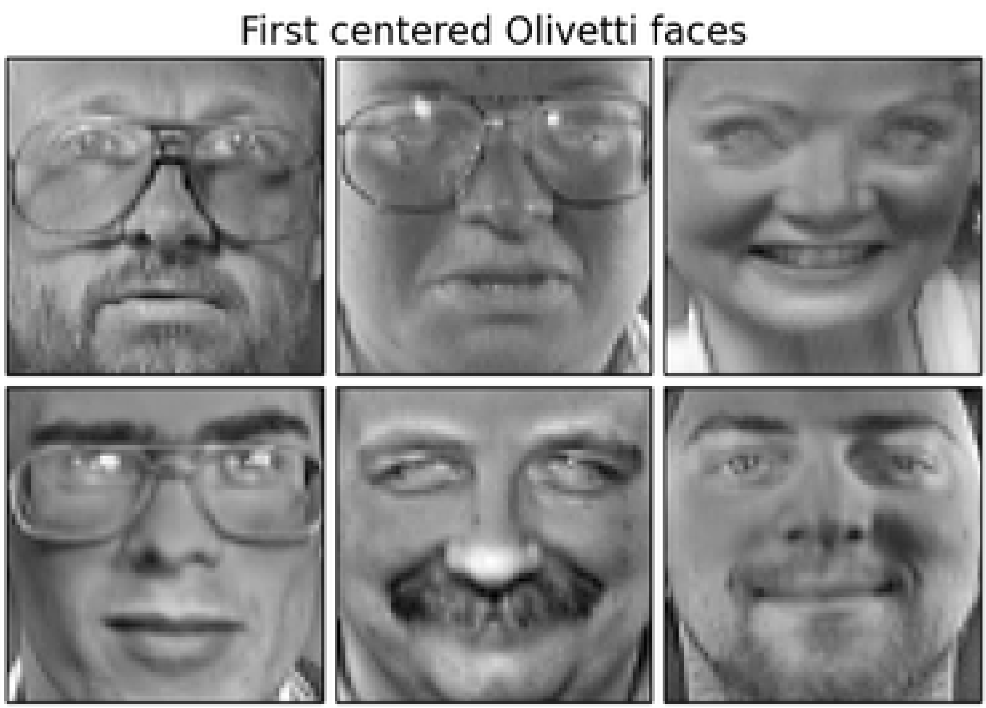


Figure 42.240: *Faces dataset decompositions*

42.7.8 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). It is very similar to PCA, but operates on sample vectors directly, instead of on a covariance matrix. 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 algorithm: 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.

Parametersn_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

Number of iterations for randomized SVD solver. Not used by ARPACK.

random_state : int or RandomState, optional

(Seed for) pseudo-random number generator. If not given, the numpy.random singleton is used.

tol : float, optional

Tolerance for ARPACK. 0 means machine precision. Ignored by randomized SVD solver.

Attributescomponents_ : array, shape (n_components, n_features)

explained_variance_ratio_ : array, [n_components]

Percentage of variance explained by each of the selected components.

explained_variance_ : array, [n_components]

The variance of the training samples transformed by a projection to each component.

See also:

PCA, RandomizedPCA

Notes

SVD suffers from a problem called "sign indeterminancy", 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) http://arxiv.org/pdf/0909.4061

Examples

```
>>> from sklearn.decomposition import TruncatedSVD
>>> from sklearn.random_projection import sparse_random_matrix
>>> X = sparse_random_matrix(100, 100, density=0.01, random_state=42)
>>> svd = TruncatedSVD(n_components=5, random_state=42)
>>> svd.fit(X)
TruncatedSVD(algorithm='randomized', n_components=5, n_iter=5,
        random_state=42, tol=0.0)
>>> print(svd.explained_variance_ratio_)
[ 0.07825... 0.05528... 0.05445... 0.04997... 0.04134...]
>>> print(svd.explained_variance_ratio_.sum())
0.27930...
```

Methods

`fit`(X[, y])	Fit LSI model on training data X.
`fit_transform`(X[, y])	Fit LSI model to X and perform dimensionality reduction on X.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(X)	Transform X back to its original space.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Perform dimensionality reduction on X.

static __init__(*n_components=2*, *algorithm='randomized'*, *n_iter=5*, *random_state=None*, *tol=0.0*)

static fit(*X*, *y=None*)

Fit LSI model on training data X.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

Training data.

Returnsself : object

Returns the transformer object.

static fit_transform(*X*, *y=None*)

Fit LSI model to X and perform dimensionality reduction on X.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

Training data.

ReturnsX_new : array, shape (n_samples, n_components)

Reduced version of X. This will always be a dense array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static inverse_transform (*X*)

Transform X back to its original space.

Returns an array X_original whose transform would be X.

ParametersX : array-like, shape (n_samples, n_components)

New data.

ReturnsX_original : array, shape (n_samples, n_features)

Note that this is always a dense array.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static transform (*X*)

Perform dimensionality reduction on X.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

New data.

ReturnsX_new : array, shape (n_samples, n_components)

Reduced version of X. This will always be a dense array.

Examples using `sklearn.decomposition.TruncatedSVD`

Figure 42.241: *Feature Union with Heterogeneous Data Sources*

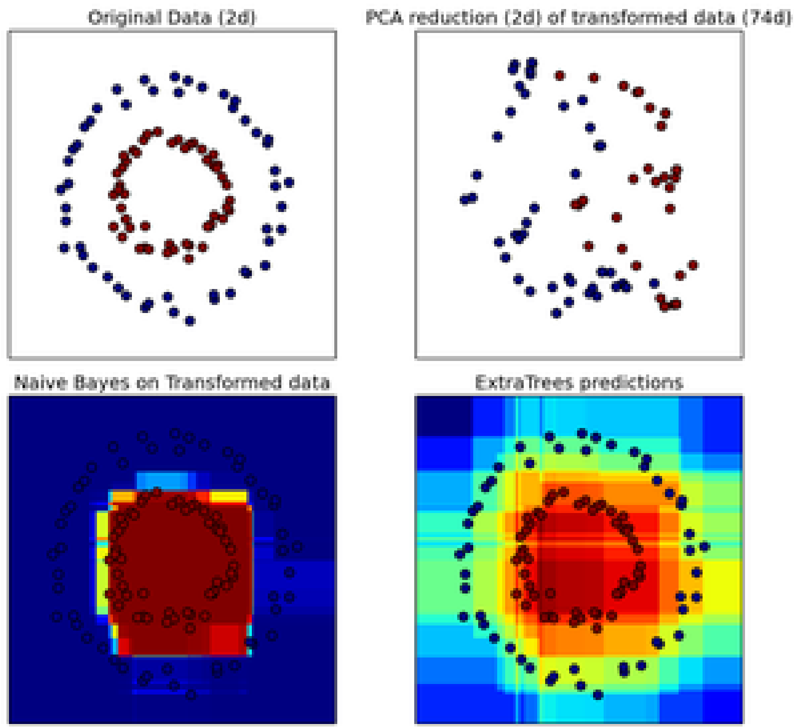


Figure 42.242: *Hashing feature transformation using Totally Random Trees*

Figure 42.243: *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

Figure 42.244: *Clustering text documents using k-means*

42.7.9 `sklearn.decomposition`.NMF

class `sklearn.decomposition.`**NMF**(*n_components=None*, *init=None*, *sparseness=None*, *beta=1*, *eta=0.1*, *tol=0.0001*, *max_iter=200*, *nls_max_iter=2000*, *random_state=None*)

Non-Negative matrix factorization by Projected Gradient (NMF)

Parametersn_components : int or None

Number of components, if n_components is not set all components are kept

init : 'nndsvd' | 'nndsvda' | 'nndsvdar' | 'random'

Method used to initialize the procedure. Default: 'nndsvdar' if n_components < n_features, otherwise random. Valid options:

```
'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)
'random': non-negative random matrices
```

sparseness : 'data' | 'components' | None, default: None

Where to enforce sparsity in the model.

beta : double, default: 1

Degree of sparseness, if sparseness is not None. Larger values mean more sparseness.

eta : double, default: 0.1

Degree of correctness to maintain, if sparsity is not None. Smaller values mean larger error.

tol : double, default: 1e-4

Tolerance value used in stopping conditions.

max_iter : int, default: 200

Number of iterations to compute.

nls_max_iter : int, default: 2000

Number of iterations in NLS subproblem.

random_state : int or RandomState

Random number generator seed control.

Attributescomponents_ : array, [n_components, n_features]

Non-negative components of the data.

reconstruction_err_ : number

Frobenius norm of the matrix difference between the training data and the reconstructed data from the fit produced by the model. `|| X - WH ||_2`

n_iter_ : int

Number of iterations run.

References

This implements

C.-J. Lin. Projected gradient methods for non-negative matrix factorization. Neural Computation, 19(2007), 2756-2779. http://www.csie.ntu.edu.tw/~cjlin/nmf/

P. Hoyer. Non-negative Matrix Factorization with Sparseness Constraints. Journal of Machine Learning Research 2004.

NNDSVD is introduced in

C. Boutsidis, E. Gallopoulos: SVD based initialization: A head start for nonnegative matrix factorization - Pattern Recognition, 2008 http://tinyurl.com/nndsvd

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 ProjectedGradientNMF
>>> model = ProjectedGradientNMF(n_components=2, init='random',
...                              random_state=0)
>>> model.fit(X)
ProjectedGradientNMF(beta=1, eta=0.1, init='random', max_iter=200,
        n_components=2, nls_max_iter=2000, random_state=0, sparseness=None,
        tol=0.0001)
>>> model.components_
array([[ 0.77032744,  0.11118662],
       [ 0.38526873,  0.38228063]])
>>> model.reconstruction_err_
0.00746...
>>> model = ProjectedGradientNMF(n_components=2,
...              sparseness='components', init='random', random_state=0)
>>> model.fit(X)
ProjectedGradientNMF(beta=1, eta=0.1, init='random', max_iter=200,
            n_components=2, nls_max_iter=2000, random_state=0,
            sparseness='components', tol=0.0001)
>>> model.components_
array([[ 1.67481991,  0.29614922],
```

```
       [ 0.        ,  0.4681982 ]])
>>> model.reconstruction_err_
0.513...
```

Methods

`fit`(X[, y])	Learn a NMF model for the data X.
`fit_transform`(X[, y])	Learn a NMF model for the data X and returns the transformed data.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Transform the data X according to the fitted NMF model

`__init__`(*n_components=None*, *init=None*, *sparseness=None*, *beta=1*, *eta=0.1*, *tol=0.0001*, *max_iter=200*, *nls_max_iter=2000*, *random_state=None*)

`fit`(*X*, *y=None*, ***params*)

Learn a NMF model for the data X.

ParametersX: {array-like, sparse matrix}, shape = [n_samples, n_features] :

Data matrix to be decomposed

Returnsself :

`fit_transform`(*X*, *y=None*)

Learn a NMF model for the data X and returns the transformed data.

This is more efficient than calling fit followed by transform.

ParametersX: {array-like, sparse matrix}, shape = [n_samples, n_features] :

Data matrix to be decomposed

Returnsdata: array, [n_samples, n_components] :

Transformed data

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

`transform`(*X*)

Transform the data X according to the fitted NMF model

ParametersX: {array-like, sparse matrix}, shape = [n_samples, n_features] :

Data matrix to be transformed by the model

Returnsdata: array, [n_samples, n_components] :

Transformed data

Examples using `sklearn.decomposition.NMF`

Figure 42.245: *Topics extraction with Non-Negative Matrix Factorization*

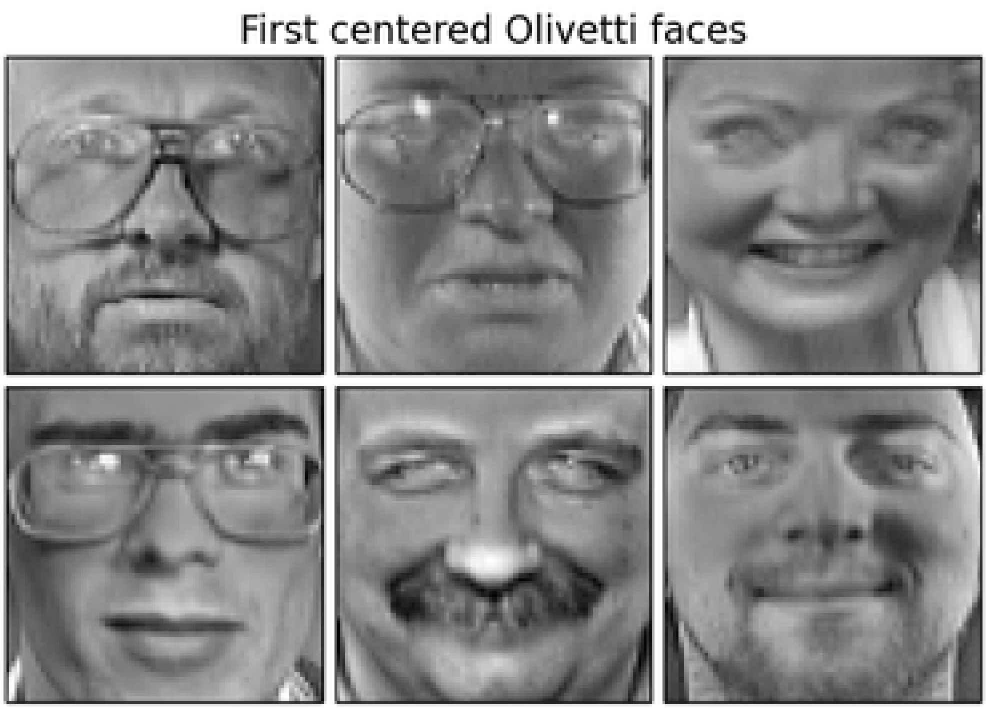


Figure 42.246: *Faces dataset decompositions*

42.7.10 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=1*, *U_init=None*, *V_init=None*, *verbose=False*, *random_state=None*)

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.

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,

Number of parallel jobs to run.

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 : :

Degree of verbosity of the printed output.

random_state : int or RandomState

Pseudo number generator state used for random sampling.

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.

See also:

PCA, MiniBatchSparsePCA, DictionaryLearning

Methods

fit(X[, y])	Fit the model from data in X.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.
transform(X[, ridge_alpha])	Least Squares projection of the data onto the sparse components.

static __init__(*n_components=None*, *alpha=1*, *ridge_alpha=0.01*, *max_iter=1000*, *tol=1e-08*, *method='lars'*, *n_jobs=1*, *U_init=None*, *V_init=None*, *verbose=False*, *random_state=None*)

static fit(*X*, *y=None*)

Fit the model from data in X.

ParametersX: 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.

Returnsself : object

Returns the instance itself.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static transform(*X*, *ridge_alpha=None*)

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.

ParametersX: 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.

ridge_alpha: float, default: 0.01 :

Amount of ridge shrinkage to apply in order to improve conditioning.

ReturnsX_new array, shape (n_samples, n_components) :

Transformed data.

42.7.11 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=1*, *method='lars'*, *random_state=None*)

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.

Parametersn_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,

callable that gets invoked every five iterations

batch_size : int,

the number of features to take in each mini batch

verbose : :

degree of output the procedure will print

shuffle : boolean,

whether to shuffle the data before splitting it in batches

n_jobs : int,

number of parallel jobs to run, or -1 to autodetect.

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 or RandomState

Pseudo number generator state used for random sampling.

Attributescomponents_ : 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.

See also:

`PCA`, `SparsePCA`, `DictionaryLearning`

Methods

`fit`(X[, y])	Fit the model from data in X.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, ridge_alpha])	Least Squares projection of the data onto the sparse components.

static `__init__`(*n_components=None*, *alpha=1*, *ridge_alpha=0.01*, *n_iter=100*, *callback=None*, *batch_size=3*, *verbose=False*, *shuffle=True*, *n_jobs=1*, *method='lars'*, *random_state=None*)

static `fit`(*X*, *y=None*)

Fit the model from data in X.

ParametersX: 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.

Returnsself : object

Returns the instance itself.

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*, *ridge_alpha=None*)

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.

ParametersX: 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.

ridge_alpha: float, default: 0.01 :

Amount of ridge shrinkage to apply in order to improve conditioning.

ReturnsX_new array, shape (n_samples, n_components) :

Transformed data.

Examples using `sklearn.decomposition.MiniBatchSparsePCA`

42.7.12 `sklearn.decomposition`.SparseCoder

class `sklearn.decomposition.`**`SparseCoder`** (*dictionary*, *transform_algorithm='omp'*, *transform_n_nonzero_coefs=None*, *transform_alpha=None*, *split_sign=False*, *n_jobs=1*)

Sparse coding

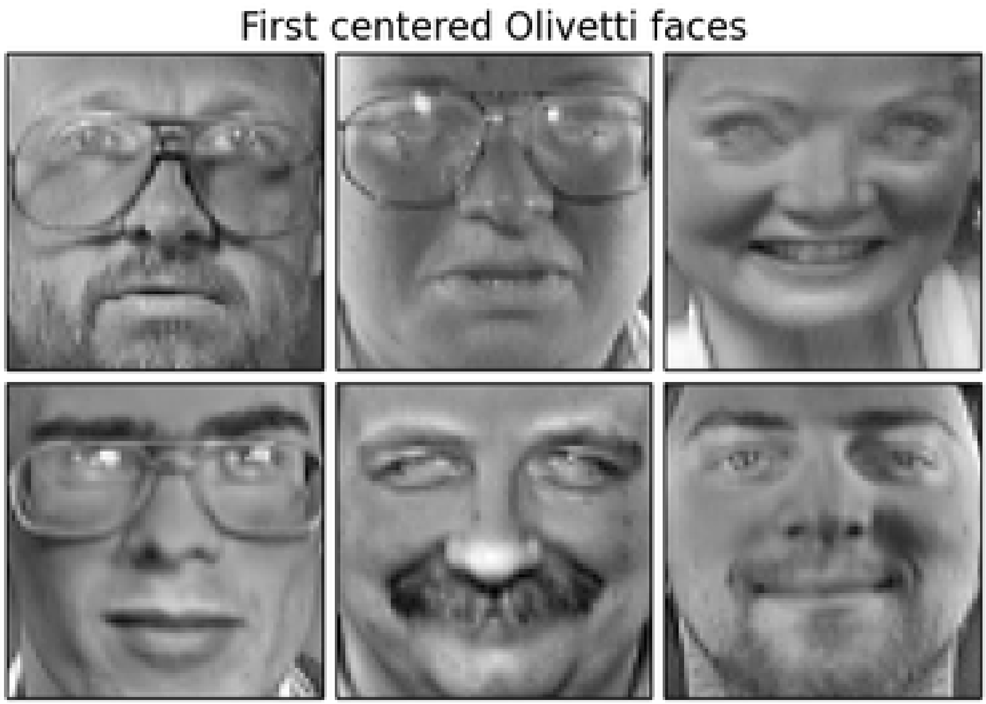


Figure 42.247: *Faces dataset decompositions*

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
```

Parametersdictionary : 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'}

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*.

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.

n_jobs : int,

number of parallel jobs to run

Attributescomponents_ : array, [n_components, n_features]

The unchanged dictionary atoms

See also:

`DictionaryLearning`, `MiniBatchDictionaryLearning`, `SparsePCA`, `MiniBatchSparsePCA`, `sparse_encode`

Methods

`fit`(X[, y])	Do nothing and return the estimator unchanged
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y])	Encode the data as a sparse combination of the dictionary atoms.

static `__init__`(*dictionary*, *transform_algorithm='omp'*, *transform_n_nonzero_coefs=None*, *transform_alpha=None*, *split_sign=False*, *n_jobs=1*)

static `fit`(*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.

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*, *y=None*)

Encode the data as a sparse combination of the dictionary atoms.

Coding method is determined by the object parameter *transform_algorithm*.

ParametersX : 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.

ReturnsX_new : array, shape (n_samples, n_components)

Transformed data

Examples using `sklearn.decomposition.SparseCoder`

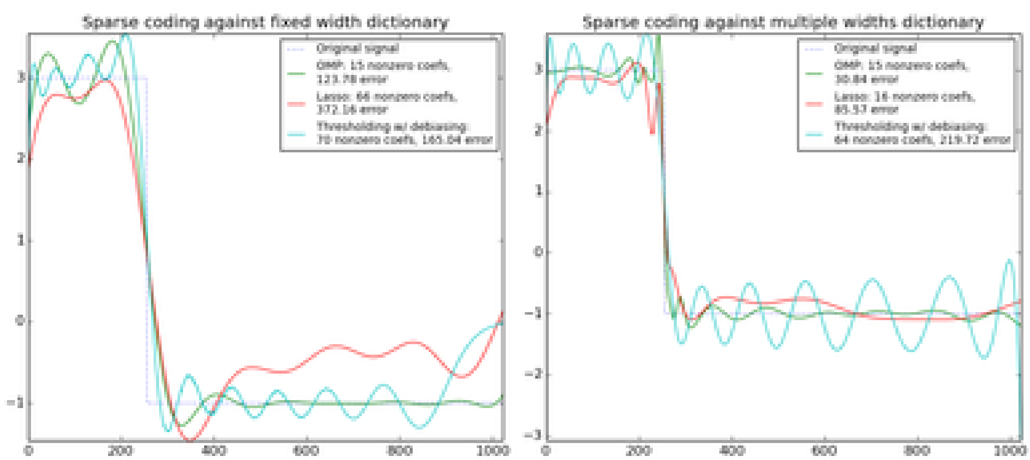


Figure 42.248: *Sparse coding with a precomputed dictionary*

42.7.13 `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=1*, *code_init=None*, *dict_init=None*, *verbose=False*, *split_sign=False*, *random_state=None*)

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
```

Parameters **n_components** : int,

number of dictionary elements to extract

alpha : float,

sparsity controlling parameter

max_iter : int,

maximum number of iterations to perform

tol : float,

tolerance for numerical error

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.

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*.

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.

n_jobs : int,

number of parallel jobs to run

code_init : array of shape (n_samples, n_components),

initial value for the code, for warm restart

dict_init : array of shape (n_components, n_features),

initial values for the dictionary, for warm restart

verbose : :

degree of verbosity of the printed output

random_state : int or RandomState

Pseudo number generator state used for random sampling.

Attributescomponents_ : 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 (http://www.di.ens.fr/sierra/pdfs/icml09.pdf)

Methods

`fit`(X[, y])	Fit the model from data in X.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y])	Encode the data as a sparse combination of the dictionary atoms.

static __init__(*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=1*, *code_init=None*, *dict_init=None*, *verbose=False*, *split_sign=False*, *random_state=None*)

static fit(*X*, *y=None*)

Fit the model from data in X.

ParametersX: 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.

Returnsself: object :

Returns the object itself

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*, *y=None*)

Encode the data as a sparse combination of the dictionary atoms.

Coding method is determined by the object parameter *transform_algorithm*.

ParametersX : 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.

ReturnsX_new : array, shape (n_samples, n_components)

Transformed data

42.7.14 `sklearn.decomposition`.MiniBatchDictionaryLearning

class `sklearn.decomposition.`**`MiniBatchDictionaryLearning`**(*n_components=None*, *alpha=1*, *n_iter=1000*, *fit_algorithm='lars'*, *n_jobs=1*, *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*)

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
```

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.

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*.

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.

n_jobs : int,

number of parallel jobs to run

dict_init : array of shape (n_components, n_features),

initial value of the dictionary for warm restart scenarios

verbose : :

degree of verbosity of the printed output

batch_size : int,

number of samples in each mini-batch

shuffle : bool,

whether to shuffle the samples before forming batches

random_state : int or RandomState

Pseudo number generator state used for random sampling.

Attributescomponents_ : 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 loosing 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.

See also:

`SparseCoder`, `DictionaryLearning`, `SparsePCA`, `MiniBatchSparsePCA`

Notes

References:

J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009: Online dictionary learning for sparse coding (http://www.di.ens.fr/sierra/pdfs/icml09.pdf)

Continued on next page

Table 42.51 – continued from previous page

Methods

fit(X[, y])	Fit the model from data in X.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
partial_fit(X[, y, iter_offset])	Updates the model using the data in X as a mini-batch.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Encode the data as a sparse combination of the dictionary atoms.

static `__init__` (*n_components=None, alpha=1, n_iter=1000, fit_algorithm='lars', n_jobs=1, 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*)

static `fit` (*X, y=None*)

Fit the model from data in X.

ParametersX: 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.

Returnsself : object

Returns the instance itself.

`fit_transform` (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

`get_params` (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static `partial_fit` (*X, y=None, iter_offset=None*)

Updates the model using the data in X as a mini-batch.

ParametersX: 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.

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.

Returnsself : object

Returns the instance itself.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*, *y=None*)

Encode the data as a sparse combination of the dictionary atoms.

Coding method is determined by the object parameter *transform_algorithm*.

ParametersX : 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.

ReturnsX_new : array, shape (n_samples, n_components)

Transformed data

Examples using `sklearn.decomposition.MiniBatchDictionaryLearning`

`decomposition.fastica`(X[, n_components, ...])	Perform Fast Independent Component Analysis.
`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.sparse_encode`(X, dictionary[, ...])	Sparse coding

42.7.15 `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.

ParametersX : 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

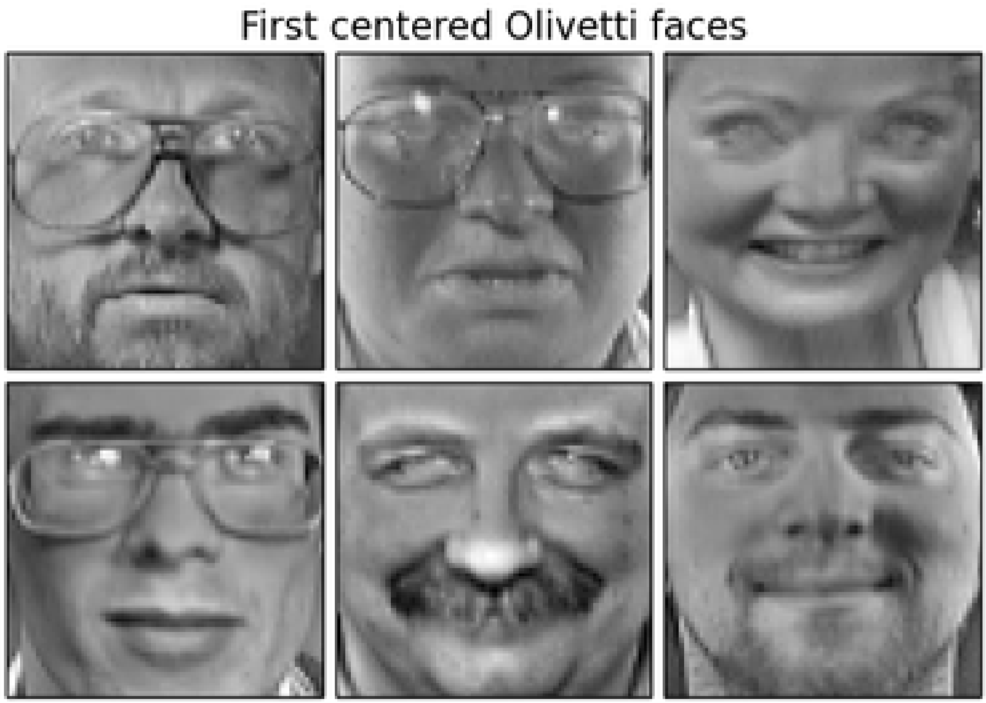


Figure 42.249: *Faces dataset decompositions*

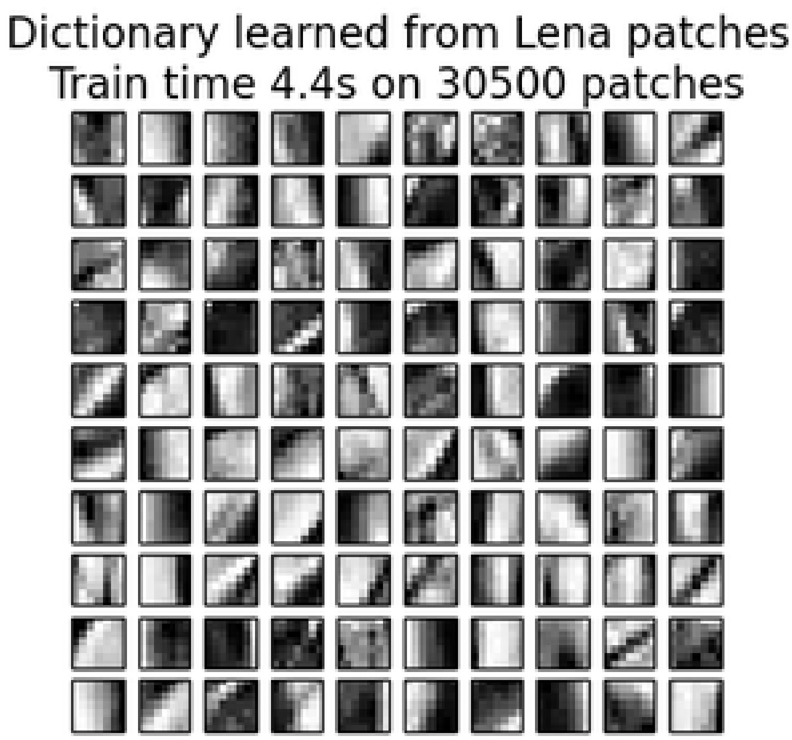


Figure 42.250: *Image denoising using dictionary learning*

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. Example:

def my_g(x):return x ** 3, 3 * x ** 2

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 or RandomState

Pseudo number generator state used for random sampling.

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)

Estimated un-mixing matrix. The mixing matrix can be obtained by:

```
w = np.dot(W, K.T)
A = w.T * (w * w.T).I
```

S : array, shape (n_components, n_samples) | 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.`*

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*

42.7.16 `sklearn.decomposition`.dict_learning

`sklearn.decomposition.`**`dict_learning`**(*X*, *n_components*, *alpha*, *max_iter=100*, *tol=1e-08*, *method='lars'*, *n_jobs=1*, *dict_init=None*, *code_init=None*, *callback=None*, *verbose=False*, *random_state=None*, *return_n_iter=False*)

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.

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, :

Number of parallel jobs to run, or -1 to autodetect.

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 that gets invoked every five iterations.

verbose: :

Degree of output the procedure will print.

random_state: int or RandomState :

Pseudo number generator state used for random sampling.

return_n_iter : bool

Whether or not to return the number of iterations.

Returnscode: 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`

42.7.17 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=1*, *method='lars'*, *iter_offset=0*, *random_state=None*, *return_inner_stats=False*, *inner_stats=None*, *return_n_iter=False*)

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.

ParametersX: 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 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 that gets invoked every five iterations.

batch_size : int,

The number of samples to take in each batch.

verbose : :

Degree of output the procedure will print.

shuffle : boolean,

Whether to shuffle the data before splitting it in batches.

n_jobs : int,

Number of parallel jobs to run, or -1 to autodetect.

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 or RandomState

Pseudo number generator state used for random sampling.

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 loosing 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.

Returnscode : 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`

42.7.18 `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=1*)

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
```

ParametersX: 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='threhold'*, *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*.

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'*.

copy_cov: boolean, optional :

Whether to copy the precomputed covariance matrix; if False, it may be overwritten.

n_jobs: int, optional :

Number of parallel jobs to run.

Returnscode: 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`

42.8 `sklearn.dummy`: Dummy estimators

User guide: See the *Model evaluation: 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.

42.8.1 `sklearn.dummy`.DummyClassifier

class `sklearn.dummy.DummyClassifier`(*strategy='stratified'*, *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.

Parametersstrategy : str

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.
- "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

random_state : int seed, RandomState instance, or None (default)

The seed of the pseudo random number generator to use.

constant : int or str or array of shape = [n_outputs]

The explicit constant as predicted by the "constant" strategy. This parameter is useful only for the "constant" strategy.

Attributesclasses_ : 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.

outputs_2d_ : bool,

True if the output at fit is 2d, else false.

'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.

Methods

`fit`(X, y[, sample_weight])	Fit the random classifier.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Perform classification on test vectors X.
`predict_log_proba`(X)	Return log probability estimates for the test vectors X.
`predict_proba`(X)	Return probability estimates for the test vectors X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*strategy='stratified'*, *random_state=None*, *constant=None*)

static `fit`(*X*, *y*, *sample_weight=None*)

Fit the random classifier.

ParametersX : {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] or [n_samples, n_outputs]

Target values.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

Returnsself : object

Returns self.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **predict** (*X*)

Perform classification on test vectors X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Input vectors, where n_samples is the number of samples and n_features is the number of features.

Returnsy : array, shape = [n_samples] or [n_samples, n_outputs]

Predicted target values for X.

static **predict_log_proba** (*X*)

Return log probability estimates for the test vectors X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Input vectors, where n_samples is the number of samples and n_features is the number of features.

ReturnsP : 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.

static **predict_proba** (*X*)

Return probability estimates for the test vectors X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Input vectors, where n_samples is the number of samples and n_features is the number of features.

ReturnsP : 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 (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

42.8.2 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.

Parametersstrategy : 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 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.

Attributesconstant_ : float or array of shape [n_outputs]

Mean or median or quantile of the training targets or constant value given by the user.

n_outputs_ : int,

Number of outputs.

outputs_2d_ : bool,

True if the output at fit is 2d, else false.

Methods

fit(X, y[, sample_weight])	Fit the random regressor.
get_params([deep])	Get parameters for this estimator.
predict(X)	Perform classification on test vectors X.
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*strategy='mean'*, *constant=None*, *quantile=None*)

static fit(*X*, *y*, *sample_weight=None*)

Fit the random regressor.

ParametersX : {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] or [n_samples, n_outputs]

Target values.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

Returnsself : object

Returns self.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static predict(*X*)

Perform classification on test vectors X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Input vectors, where n_samples is the number of samples and n_features is the number of features.

Returnsy : array, shape = [n_samples] or [n_samples, n_outputs]

Predicted target values for X.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

42.9 sklearn.ensemble: Ensemble Methods

The `sklearn.ensemble` module includes ensemble-based methods for classification and regression.

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.RandomForestClassifier([...])	A random forest classifier.
ensemble.RandomTreesEmbedding([...])	An ensemble of totally random trees.
ensemble.RandomForestRegressor([...])	A random forest regressor.

42.9.1 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].

Parametersbase_estimator : object, optional (default=DecisionTreeClassifier)

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.

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 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*.

Attributesestimators_ : 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_ : array of shape = [n_features]

The feature importances if supported by the `base_estimator`.

See also:

`AdaBoostRegressor`, `GradientBoostingClassifier`, `DecisionTreeClassifier`

References

[R120], [R121]

Methods

decision_function(X)	Compute the decision function of X.
fit(X, y[, sample_weight])	Build a boosted classifier from the training set (X, y).
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict classes for X.
predict_log_proba(X)	Predict class log-probabilities for X.
predict_proba(X)	Predict class probabilities for X.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.
staged_decision_function(X)	Compute decision function of X for each boosting iteration.
staged_predict(X)	Return staged predictions for X.
staged_predict_proba(X)	Predict class probabilities for X.
staged_score(X, y[, sample_weight])	Return staged scores for X, y.

static `__init__`(*base_estimator=None*, *n_estimators=50*, *learning_rate=1.0*, *algorithm='SAMME.R'*, *random_state=None*)

static `decision_function`(*X*)

Compute the decision function of `X`.

ParametersX : {array-like, sparse matrix} of shape = [n_samples, n_features]

The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. DOK and LIL are converted to CSR.

Returnsscore : 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.

static `fit`(*X*, *y*, *sample_weight=None*)

Build a boosted classifier from the training set (X, y).

ParametersX : {array-like, sparse matrix} of shape = [n_samples, n_features]

The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. 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], optional

Sample weights. If None, the sample weights are initialized to `1 / n_samples`.

Returnsself : object

Returns self.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **predict** (*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.

ParametersX : {array-like, sparse matrix} of shape = [n_samples, n_features]

The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. DOK and LIL are converted to CSR.

Returnsy : array of shape = [n_samples]

The predicted classes.

static **predict_log_proba** (*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.

ParametersX : {array-like, sparse matrix} of shape = [n_samples, n_features]

The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. DOK and LIL are converted to CSR.

Returnsp : array of shape = [n_samples]

The class probabilities of the input samples. The order of outputs is the same of that of the *classes_* attribute.

static **predict_proba** (*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.

ParametersX : {array-like, sparse matrix} of shape = [n_samples, n_features]

The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. DOK and LIL are converted to CSR.

Returnsp : array of shape = [n_samples]

The class probabilities of the input samples. The order of outputs is the same of that of the *classes_* attribute.

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **staged_decision_function**(*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.

ParametersX : {array-like, sparse matrix} of shape = [n_samples, n_features]

The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. DOK and LIL are converted to CSR.

Returnsscore : 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.

static **staged_predict**(*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.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsy : generator of array, shape = [n_samples]

The predicted classes.

static **staged_predict_proba**(*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.

ParametersX : {array-like, sparse matrix} of shape = [n_samples, n_features]

The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. DOK and LIL are converted to CSR.

Returnsp : 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 (*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.

ParametersX : {array-like, sparse matrix} of shape = [n_samples, n_features]

The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. DOK and LIL are converted to CSR.

y : array-like, shape = [n_samples]

Labels for X.

sample_weight : array-like, shape = [n_samples], optional

Sample weights.

Returnsz : float

Examples using `sklearn.ensemble.AdaBoostClassifier`

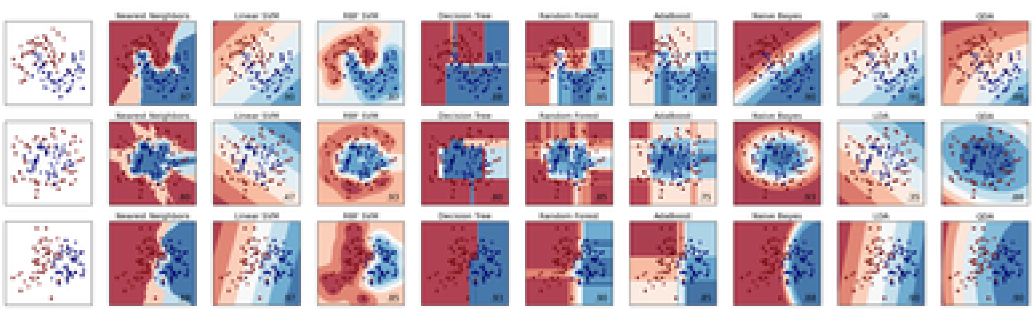


Figure 42.251: *Classifier comparison*

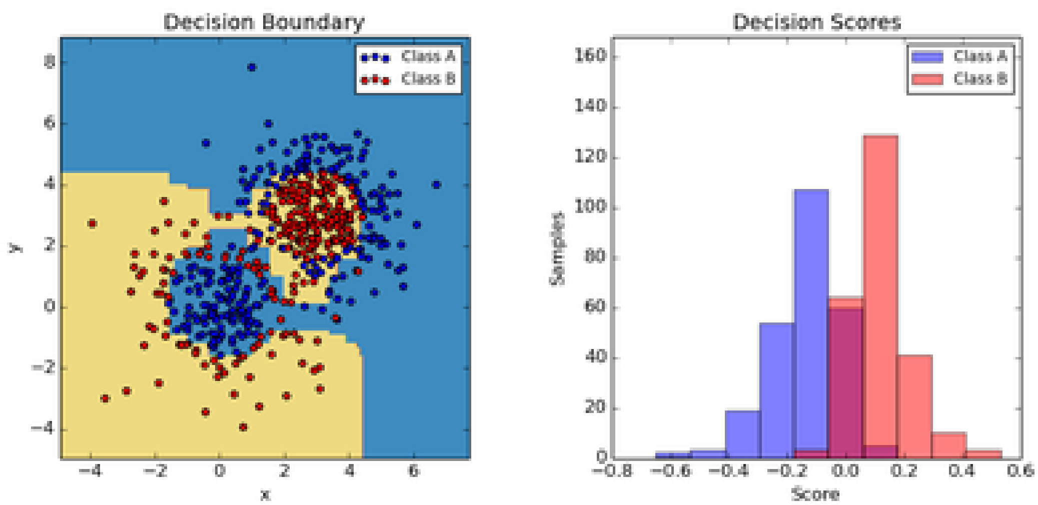


Figure 42.252: *Two-class AdaBoost*

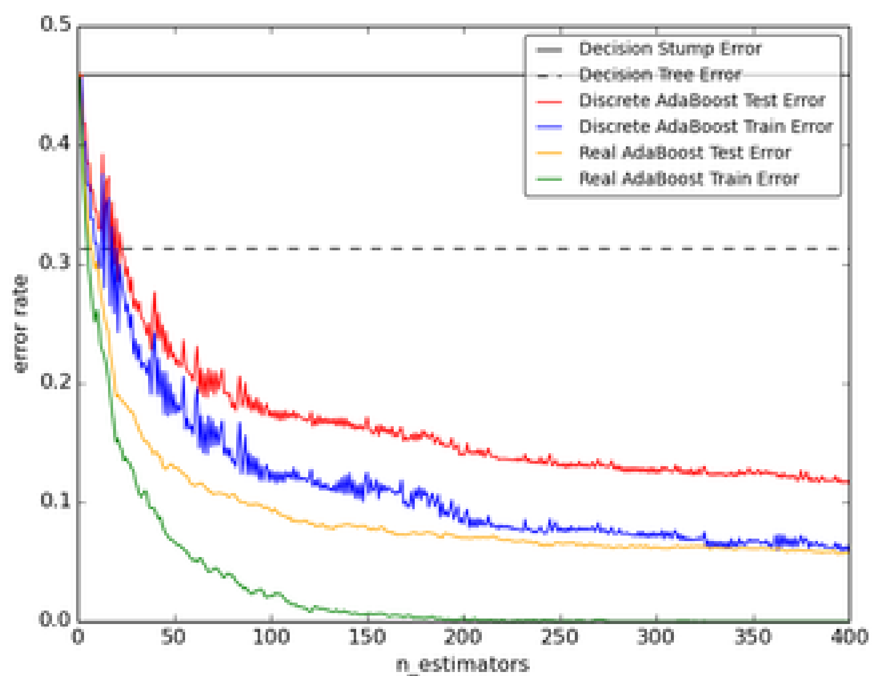


Figure 42.253: *Discrete versus Real AdaBoost*

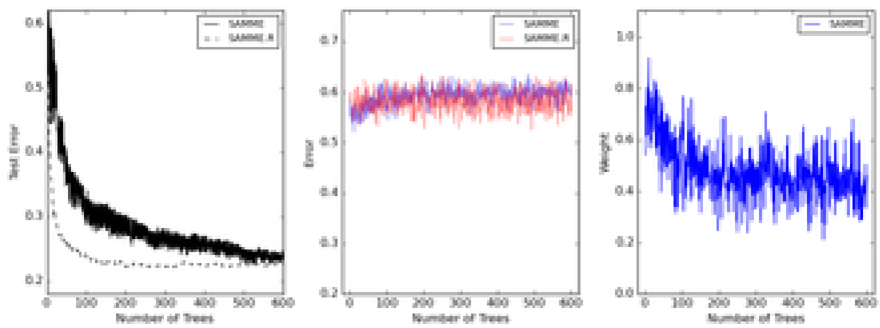


Figure 42.254: *Multi-class AdaBoosted Decision Trees*

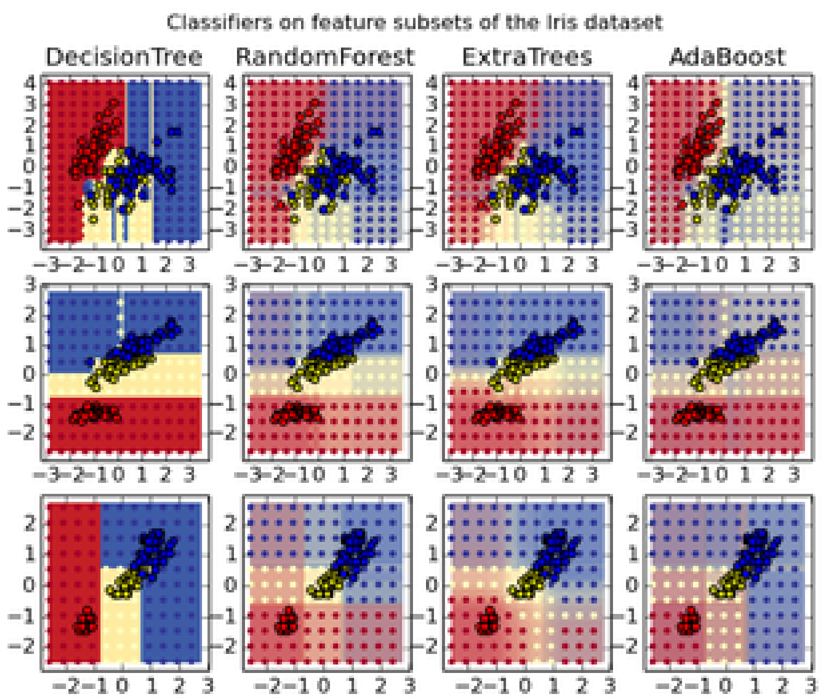


Figure 42.255: *Plot the decision surfaces of ensembles of trees on the iris dataset*

42.9.2 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].

Parameters **base_estimator** : object, optional (default=DecisionTreeRegressor)

The base estimator from which the boosted ensemble is built. Support for sample weighting is required.

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 **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_ : array of shape = [n_features]

The feature importances if supported by the `base_estimator`.

See also:

`AdaBoostClassifier`, `GradientBoostingRegressor`, `DecisionTreeRegressor`

References

[R122], [R123]

Methods

`fit`(X, y[, sample_weight])	Build a boosted regressor from the training set (X, y).
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict regression value for X.
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.
`staged_predict`(X)	Return staged predictions for X.
`staged_score`(X, y[, sample_weight])	Return staged scores for X, y.

static __init__ (*base_estimator=None, n_estimators=50, learning_rate=1.0, loss='linear', random_state=None*)

static fit (*X, y, sample_weight=None*)

Build a boosted regressor from the training set (X, y).

ParametersX : {array-like, sparse matrix} of shape = [n_samples, n_features]

The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. 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], optional

Sample weights. If None, the sample weights are initialized to 1 / n_samples.

Returnsself : object

Returns self.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static predict (*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.

ParametersX : {array-like, sparse matrix} of shape = [n_samples, n_features]

The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. DOK and LIL are converted to CSR.

Returnsy : array of shape = [n_samples]

The predicted regression values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

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.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns self :

static **staged_predict**(*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. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. DOK and LIL are converted to CSR.

Returns y : generator of array, shape = [n_samples]

The predicted regression values.

staged_score(*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. DOK and LIL are converted to CSR.

y : array-like, shape = [n_samples]

Labels for X.

sample_weight : array-like, shape = [n_samples], optional

Sample weights.

Returns z : float

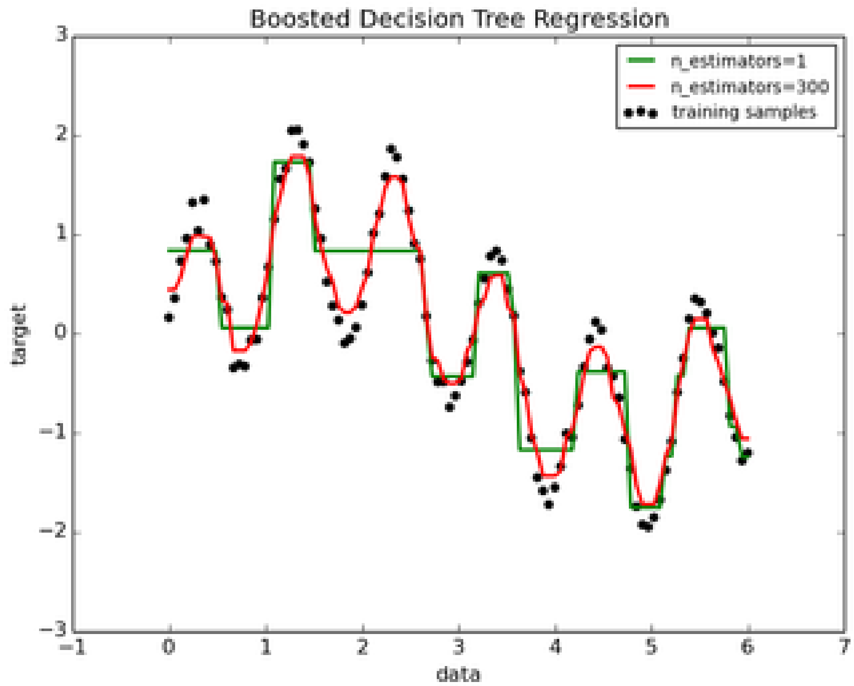


Figure 42.256: *Decision Tree Regression with AdaBoost*

Examples using `sklearn.ensemble.AdaBoostRegressor`

42.9.3 `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*, *n_jobs=1*, *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 [R124]. If samples are drawn with replacement, then the method is known as Bagging [R125]. When random subsets of the dataset are drawn as random subsets of the features, then the method is known as Random Subspaces [R126]. Finally, when base estimators are built on subsets of both samples and features, then the method is known as Random Patches [R127].

Parametersbase_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.

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.

n_jobs : int, optional (default=1)

The number of jobs to run in parallel for both *fit* and *predict*. If -1, then the number of jobs is set to the number of cores.

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 building process.

Attributesbase_estimator_ : list of estimators

The base estimator from which the ensemble is grown.

estimators_ : list of estimators

The collection of fitted base estimators.

estimators_samples_ : list of arrays

The subset of drawn samples (i.e., the in-bag 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.

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.

References

[R124], [R125], [R126], [R127]

Methods

`decision_function`	Average of the decision functions of the base classifiers.
`fit`(X, y[, sample_weight])	Build a Bagging ensemble of estimators from the training set (X, y).
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict class for X.
`predict_log_proba`(X)	Predict class log-probabilities for X.
`predict_proba`(X)	Predict class probabilities for X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.

static __init__(*base_estimator=None*, *n_estimators=10*, *max_samples=1.0*, *max_features=1.0*, *bootstrap=True*, *bootstrap_features=False*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*)

decision_function

Average of the decision functions of the base classifiers.

ParametersX : {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.

Returnsscore : 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`.

fit(*X*, *y*, *sample_weight=None*)

Build a Bagging ensemble of estimators from the trainingset (X, y).

ParametersX : {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, shape = [n_samples]

The target values (class labels in classification, real numbers in regression).

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 the base estimator supports sample weighting.

Returnsself : object

Returns self.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static predict (*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.

ParametersX : {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.

Returnsy : array of shape = [n_samples]

The predicted classes.

static predict_log_proba (*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.

ParametersX : {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.

Returnsp : 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_*.

static predict_proba (*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 a an input sample represents the proportion of estimators predicting each class.

ParametersX : {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.

Returnsp : 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 (*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.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returns **self** :

42.9.4 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*, *n_jobs=1*, *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 [R128]. If samples are drawn with replacement, then the method is known as Bagging [R129]. When random subsets of the dataset are drawn as random subsets of the features, then the method is known as Random Subspaces [R130]. Finally, when base estimators are built on subsets of both samples and features, then the method is known as Random Patches [R131].

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.

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.

n_jobs : int, optional (default=1)

The number of jobs to run in parallel for both *fit* and *predict*. If -1, then the number of jobs is set to the number of cores.

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 building process.

Attributesestimators_ : list of estimators

The collection of fitted sub-estimators.

estimators_samples_ : list of arrays

The subset of drawn samples (i.e., the in-bag 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.

oob_prediction_ : array 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.

References

[R128], [R129], [R130], [R131]

Methods

fit(X, y[, sample_weight])	Build a Bagging ensemble of estimators from the training set (X, y).
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict regression target for X.
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static `__init__`(*base_estimator=None*, *n_estimators=10*, *max_samples=1.0*, *max_features=1.0*, *bootstrap=True*, *bootstrap_features=False*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*)

`fit`(*X*, *y*, *sample_weight=None*)

Build a Bagging ensemble of estimators from the trainingset (X, y).

ParametersX : {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, shape = [n_samples]

The target values (class labels in classification, real numbers in regression).

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 the base estimator supports sample weighting.

Returnsself : object

Returns self.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static `predict`(*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.

ParametersX : {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.

Returnsy : array of shape = [n_samples]

The predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.ensemble.BaggingRegressor`

42.9.5 `sklearn.ensemble`.ExtraTreesClassifier

class `sklearn.ensemble.`**`ExtraTreesClassifier`**(*n_estimators=10*, *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*, *bootstrap=False*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*, *warm_start=False*, *class_weight=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 use averaging to improve the predictive accuracy and control over-fitting.

Parametersn_estimators : integer, optional (default=10)

The number of trees in the forest.

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_features : int, float, string or None, optional (default="auto")

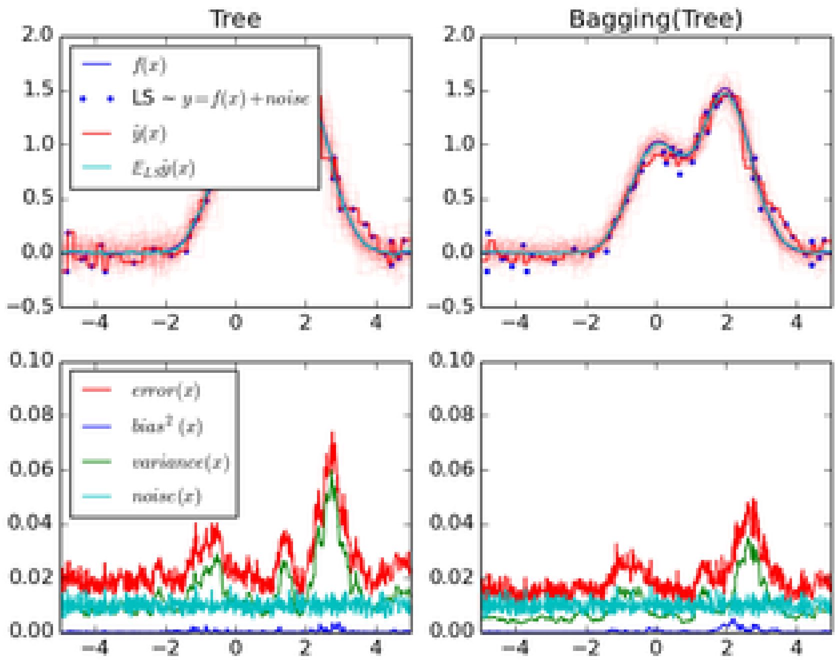


Figure 42.257: *Single estimator versus bagging: bias-variance decomposition*

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 percentage 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. 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. Ignored if `max_leaf_nodes` is not None. Note: this parameter is tree-specific.

min_samples_split : integer, optional (default=2)

The minimum number of samples required to split an internal node. Note: this parameter is tree-specific.

min_samples_leaf : integer, optional (default=1)

The minimum number of samples in newly created leaves. A split is discarded if after the split, one of the leaves would contain less then `min_samples_leaf` samples. Note: this parameter is tree-specific.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node. Note: this parameter is tree-specific.

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. If not None then `max_depth` will be ignored. Note: this parameter is tree-specific.

bootstrap : boolean, optional (default=False)

Whether bootstrap samples are used when building trees.

oob_score : bool

Whether to use out-of-bag samples to estimate the generalization error.

n_jobs : integer, optional (default=1)

The number of jobs to run in parallel for both *fit* and *predict*. If -1, then the number of jobs is set to the number of cores.

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.

class_weight : dict, list of dicts, "auto", "subsample" 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.

The "auto" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data.

The "subsample" mode is the same as "auto" 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.

Attributesestimators_ : 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_ : array of shape = [n_features]

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.

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.

See also:

`sklearn.tree.ExtraTreeClassifier`Base classifier for this ensemble.

`RandomForestClassifier`Ensemble Classifier based on trees with optimal splits.

References

[R132]

Methods

`apply`(X)	Apply trees in the forest to X, return leaf indices.
`fit`(X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict class for X.
`predict_log_proba`(X)	Predict class log-probabilities for X.
`predict_proba`(X)	Predict class probabilities for X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, threshold])	Reduce X to its most important features.

static `__init__`(*n_estimators=10, 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, bootstrap=False, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False, class_weight=None*)

`apply`(*X*)

Apply trees in the forest to X, return leaf indices.

ParametersX : 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`.

ReturnsX_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.

fit (*X*, *y*, *sample_weight=None*)

Build a forest of trees from the training set (X, y).

ParametersX : array-like or 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, shape = [n_samples] or [n_samples, n_outputs]

The target values (class labels in classification, real numbers in regression).

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.

Returnsself : object

Returns self.

fit_transform (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict class for X.

The predicted class of an input sample is computed as the majority prediction of the trees in the forest.

ParametersX : 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`.

Returnsy : array of shape = [n_samples] or [n_samples, n_outputs]

The predicted classes.

predict_log_proba(*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, 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], 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(*X*)

Predict class probabilities for X.

The predicted class probabilities of an input sample is 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, 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], 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(*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.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns self :

transform(*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Examples using `sklearn.ensemble.ExtraTreesClassifier`

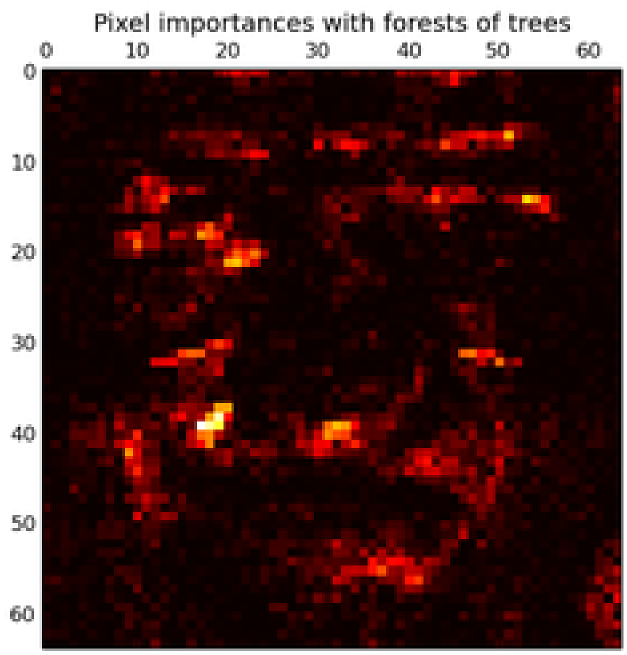


Figure 42.258: *Pixel importances with a parallel forest of trees*

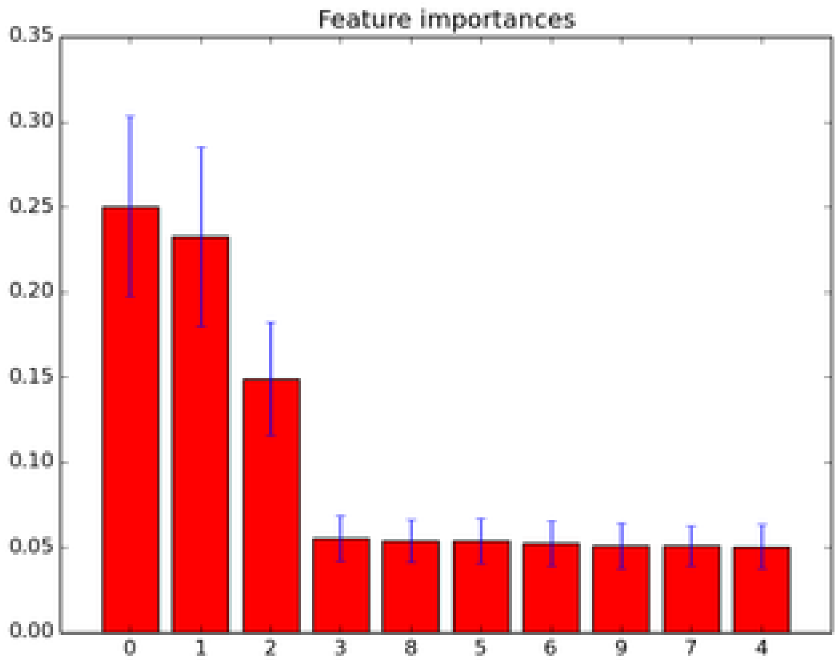


Figure 42.259: *Feature importances with forests of trees*

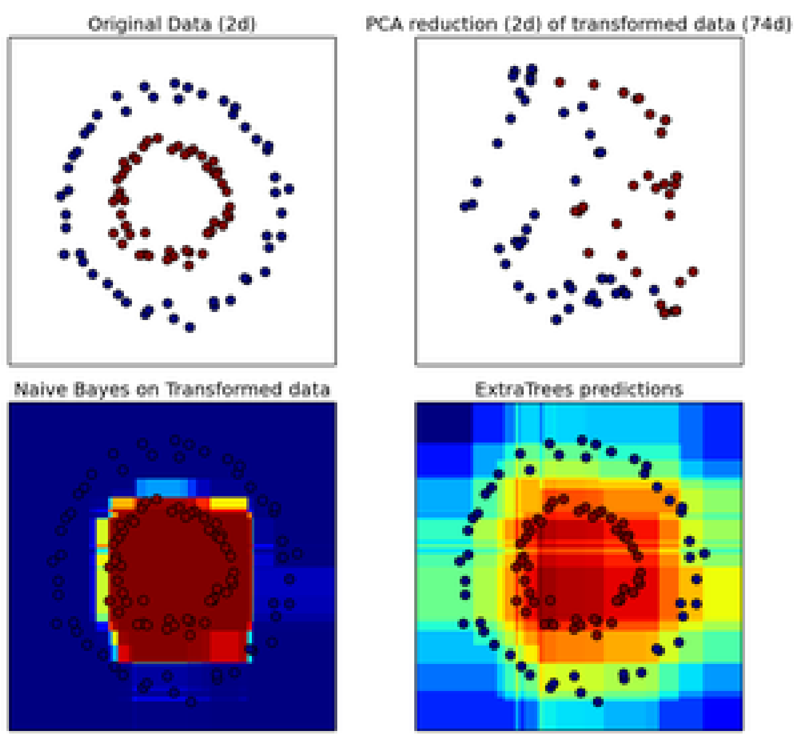


Figure 42.260: *Hashing feature transformation using Totally Random Trees*

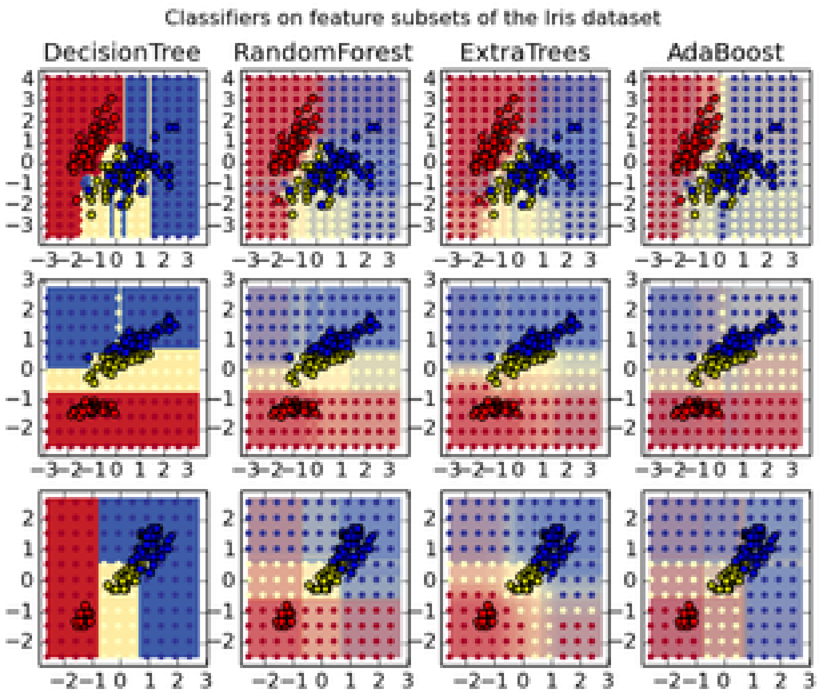


Figure 42.261: *Plot the decision surfaces of ensembles of trees on the iris dataset*

42.9.6 sklearn.ensemble.ExtraTreesRegressor

class sklearn.ensemble.**ExtraTreesRegressor**(*n_estimators=10*, *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*, *bootstrap=False*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*, *warm_start=False*)

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 use averaging to improve the predictive accuracy and control over-fitting.

Parametersn_estimators : integer, optional (default=10)

The number of trees in the forest.

criterion : string, optional (default="mse")

The function to measure the quality of a split. The only supported criterion is "mse" for the mean squared error. Note: this parameter is tree-specific.

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 percentage 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. 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. Ignored if `max_leaf_nodes` is not None. Note: this parameter is tree-specific.

min_samples_split : integer, optional (default=2)

The minimum number of samples required to split an internal node. Note: this parameter is tree-specific.

min_samples_leaf : integer, optional (default=1)

The minimum number of samples in newly created leaves. A split is discarded if after the split, one of the leaves would contain less then `min_samples_leaf` samples. Note: this parameter is tree-specific.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node. Note: this parameter is tree-specific.

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. If not None then `max_depth` will be ignored. Note: this parameter is tree-specific.

bootstrap : boolean, optional (default=False)

Whether bootstrap samples are used when building trees. Note: this parameter is tree-specific.

oob_score : bool

Whether to use out-of-bag samples to estimate the generalization error.

n_jobs : integer, optional (default=1)

The number of jobs to run in parallel for both *fit* and *predict*. If -1, then the number of jobs is set to the number of cores.

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.

Attributesestimators_ : list of DecisionTreeRegressor

The collection of fitted sub-estimators.

feature_importances_ : array of shape = [n_features]

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.

oob_prediction_ : array of shape = [n_samples]

Prediction computed with out-of-bag estimate on the training set.

See also:

`sklearn.tree.ExtraTreeRegressor`Base estimator for this ensemble.

`RandomForestRegressor`Ensemble regressor using trees with optimal splits.

References

[R133]

Methods

`apply`(X)	Apply trees in the forest to X, return leaf indices.
`fit`(X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict regression target for X.
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, threshold])	Reduce X to its most important features.

static `__init__`(*n_estimators=10, 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, bootstrap=False, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False*)

`apply`(*X*)

Apply trees in the forest to X, return leaf indices.

ParametersX : 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`.

ReturnsX_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.

fit (*X*, *y*, *sample_weight=None*)
Build a forest of trees from the training set (X, y).

ParametersX : array-like or 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, shape = [n_samples] or [n_samples, n_outputs]

The target values (class labels in classification, real numbers in regression).

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.

Returnsself : object

Returns self.

fit_transform (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*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.

ParametersX : 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`.

Returnsy : array of shape = [n_samples] or [n_samples, n_outputs]

The predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*, *threshold=None*)
Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Figure 42.262: *Face completion with a multi-output estimators*

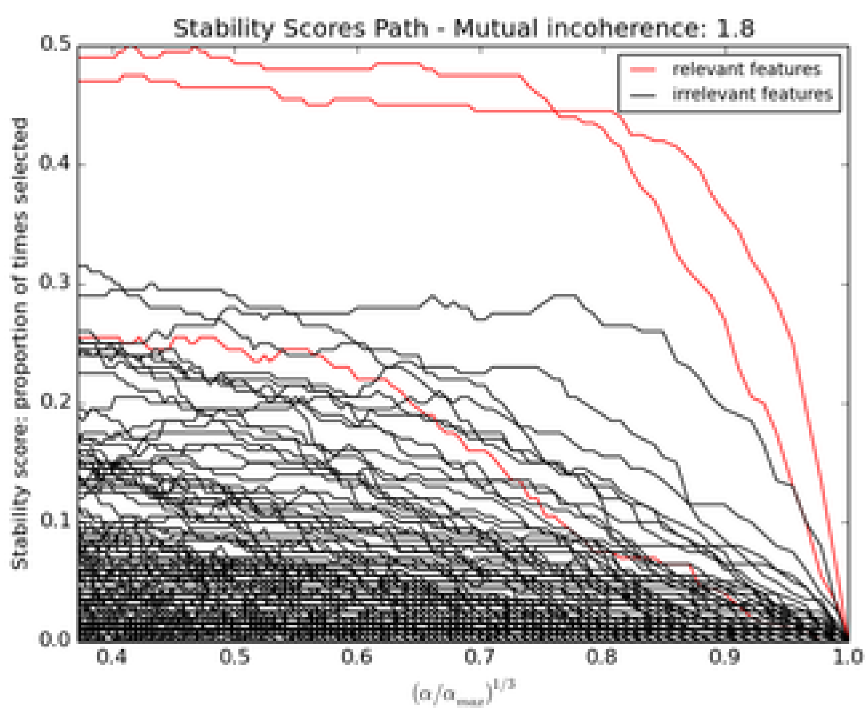


Figure 42.263: *Sparse recovery: feature selection for sparse linear models*

Examples using `sklearn.ensemble.ExtraTreesRegressor`

42.9.7 `sklearn.ensemble`.GradientBoostingClassifier

class `sklearn.ensemble.`**`GradientBoostingClassifier`**(*loss='deviance'*, *learning_rate=0.1*, *n_estimators=100*, *subsample=1.0*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_depth=3*, *init=None*, *random_state=None*, *max_features=None*, *verbose=0*, *max_leaf_nodes=None*, *warm_start=False*)

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.

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.

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. Ignored if `max_leaf_nodes` is not None.

min_samples_split : integer, optional (default=2)

The minimum number of samples required to split an internal node.

min_samples_leaf : integer, optional (default=1)

The minimum number of samples required to be at a leaf node.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node.

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.

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 percentage 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.

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. If not None then `max_depth` will be ignored.

init : BaseEstimator, None, optional (default=None)

An estimator object that is used to compute the initial predictions. `init` has to provide `fit` and `predict`. If None it uses `loss.init_estimator`.

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.

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.

Attributesfeature_importances_ : array, shape = [n_features]

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.

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 : BaseEstimator

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.

See also:

`sklearn.tree.DecisionTreeClassifier`, `RandomForestClassifier`, `AdaBoostClassifier`

References

J. Friedman, Greedy Function Approximation: A Gradient Boosting Machine, The Annals of Statistics, Vol. 29, No. 5, 2001.

10.Friedman, Stochastic Gradient Boosting, 1999

T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning Ed. 2, Springer, 2009.

Methods

`decision_function`(X)	Compute the decision function of `X`.
`fit`(X, y[, sample_weight, monitor])	Fit the gradient boosting model.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict class for X.
`predict_log_proba`(X)	Predict class log-probabilities for X.
`predict_proba`(X)	Predict class probabilities for X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.
`staged_decision_function`(X)	Compute decision function of `X` for each iteration.
`staged_predict`(X)	Predict class at each stage for X.
`staged_predict_proba`(X)	Predict class probabilities at each stage for X.
`transform`(X[, threshold])	Reduce X to its most important features.

static `__init__`(*loss='deviance'*, *learning_rate=0.1*, *n_estimators=100*, *subsample=1.0*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_depth=3*, *init=None*, *random_state=None*, *max_features=None*, *verbose=0*, *max_leaf_nodes=None*, *warm_start=False*)

`decision_function`(*X*)

Compute the decision function of `X`.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsscore : 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`(*X*, *y*, *sample_weight=None*, *monitor=None*)

Fit the gradient boosting model.

ParametersX : 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 (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.

Returnsself : object

Returns self.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **predict**(*X*)

Predict class for X.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsy: array of shape = ["n_samples] :

The predicted values.

static **predict_log_proba**(*X*)
Predict class log-probabilities for X.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsp : array of shape = [n_samples]

The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

RaisesAttributeError :

If the `loss` does not support probabilities.

static **predict_proba**(*X*)
Predict class probabilities for X.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsp : array of shape = [n_samples]

The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

RaisesAttributeError :

If the `loss` does not support probabilities.

score(*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

staged_decision_function(*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 of shape = [n_samples, n_features]

The input samples.

Returns score : generator of array, shape = [n_samples, k]

The decision function of the input samples. The order of the classes corresponds to that in the attribute *classes_*. Regression and binary classification are special cases with `k == 1`, otherwise `k==n_classes`.

static **staged_predict**(*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 of shape = [n_samples, n_features]

The input samples.

Returns y : generator of array of shape = [n_samples]

The predicted value of the input samples.

static **staged_predict_proba**(*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 of shape = [n_samples, n_features]

The input samples.

Returns y : generator of array of shape = [n_samples]

The predicted value of the input samples.

transform(*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

Parameters X : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

Returns X_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

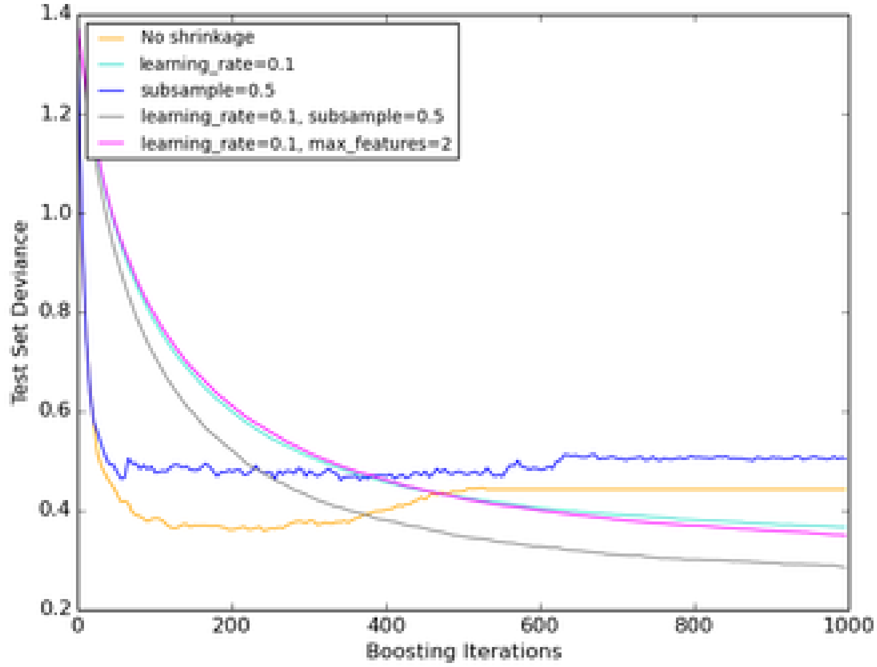


Figure 42.264: *Gradient Boosting regularization*

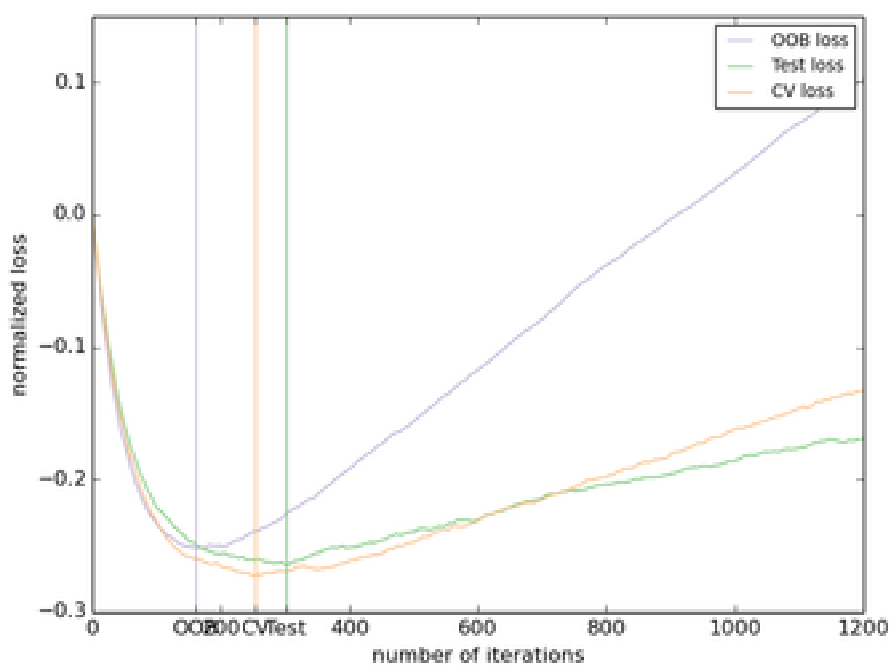


Figure 42.265: *Gradient Boosting Out-of-Bag estimates*

Examples using `sklearn.ensemble.GradientBoostingClassifier`

42.9.8 `sklearn.ensemble`.GradientBoostingRegressor

class `sklearn.ensemble.`**`GradientBoostingRegressor`**(*loss='ls'*, *learning_rate=0.1*, *n_estimators=100*, *subsample=1.0*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_depth=3*, *init=None*, *random_state=None*, *max_features=None*, *alpha=0.9*, *verbose=0*, *max_leaf_nodes=None*, *warm_start=False*)

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.

Parametersloss : {'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.

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. Ignored if `max_leaf_nodes` is not None.

min_samples_split : integer, optional (default=2)

The minimum number of samples required to split an internal node.

min_samples_leaf : integer, optional (default=1)

The minimum number of samples required to be at a leaf node.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node.

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.

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 percentage 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.

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.

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'`.

init : BaseEstimator, None, optional (default=None)

An estimator object that is used to compute the initial predictions. `init` has to provide `fit` and `predict`. If None it uses `loss.init_estimator`.

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.

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.

Attributesfeature_importances_ : array, shape = [n_features]

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.

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' : BaseEstimator

The estimator that provides the initial predictions. Set via the `init` argument or `loss.init_estimator`.

estimators_ : ndarray of DecisionTreeRegressor, shape = [n_estimators, 1]

The collection of fitted sub-estimators.

See also:

`DecisionTreeRegressor`, `RandomForestRegressor`

References

J. Friedman, Greedy Function Approximation: A Gradient Boosting Machine, The Annals of Statistics, Vol. 29, No. 5, 2001.

10.Friedman, Stochastic Gradient Boosting, 1999

T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning Ed. 2, Springer, 2009.

Methods

`decision_function`(X)	Compute the decision function of `X`.
`fit`(X, y[, sample_weight, monitor])	Fit the gradient boosting model.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict regression target for X.
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.
`staged_decision_function`(X)	Compute decision function of `X` for each iteration.
`staged_predict`(X)	Predict regression target at each stage for X.
`transform`(X[, threshold])	Reduce X to its most important features.

static `__init__`(*loss='ls'*, *learning_rate=0.1*, *n_estimators=100*, *subsample=1.0*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_depth=3*, *init=None*, *random_state=None*, *max_features=None*, *alpha=0.9*, *verbose=0*, *max_leaf_nodes=None*, *warm_start=False*)

`decision_function`(*X*)

Compute the decision function of `X`.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsscore : 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`(*X*, *y*, *sample_weight=None*, *monitor=None*)

Fit the gradient boosting model.

ParametersX : 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 (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.

Returnsself : object

Returns self.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static predict(*X*)

Predict regression target for X.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsy : array of shape = [n_samples]

The predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

staged_decision_function (*X*)
Compute decision function of `X` for each iteration.

This method allows monitoring (i.e. determine error on testing set) after each stage.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsscore : generator of array, shape = [n_samples, k]

The decision function of the input samples. The order of the classes corresponds to that in the attribute *classes_*. Regression and binary classification are special cases with `k == 1`, otherwise `k==n_classes`.

static **staged_predict** (*X*)
Predict regression target at each stage for X.

This method allows monitoring (i.e. determine error on testing set) after each stage.

ParametersX : array-like of shape = [n_samples, n_features]

The input samples.

Returnsy : generator of array of shape = [n_samples]

The predicted value of the input samples.

transform (*X*, *threshold=None*)
Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Examples using `sklearn.ensemble.GradientBoostingRegressor`

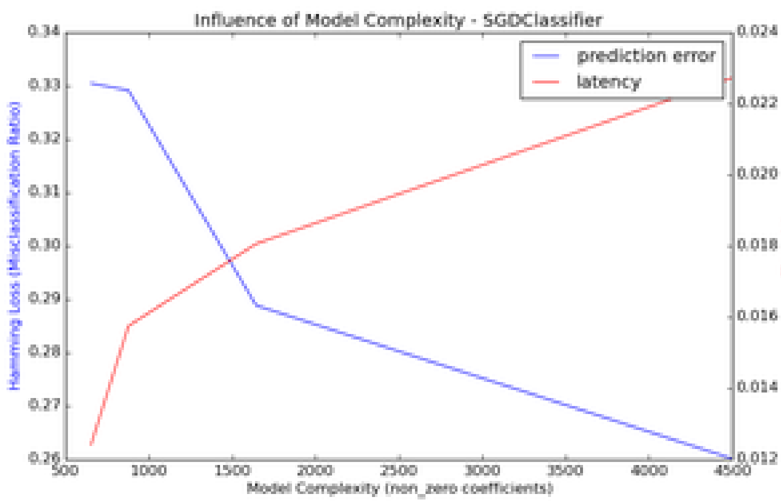


Figure 42.266: *Model Complexity Influence*

42.9.9 `sklearn.ensemble`.RandomForestClassifier

class `sklearn.ensemble.`**`RandomForestClassifier`**(*n_estimators=10*, *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*, *bootstrap=True*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*, *warm_start=False*, *class_weight=None*)

A random forest classifier.

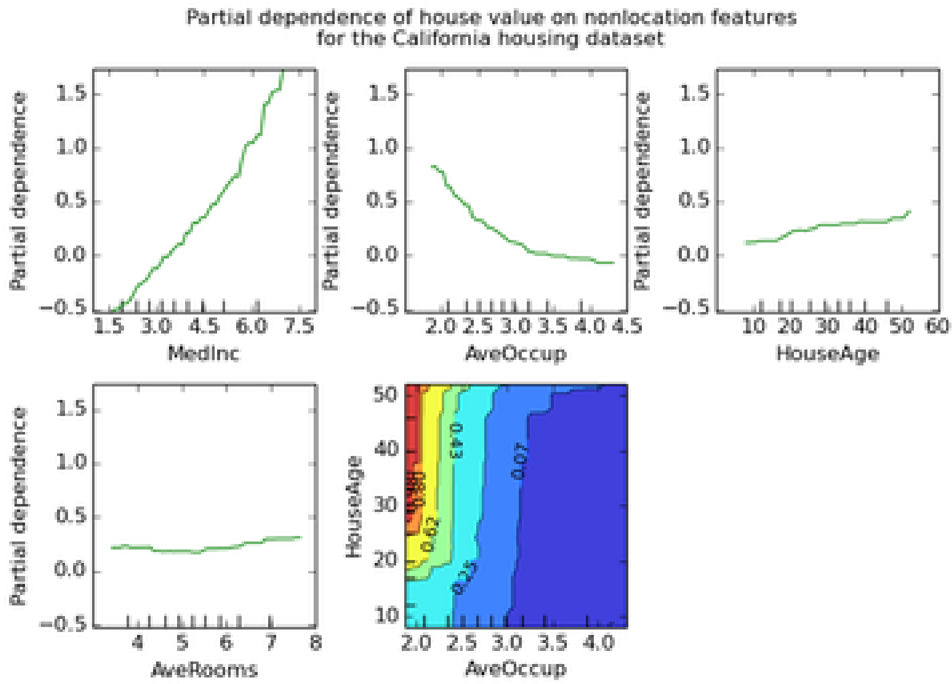


Figure 42.267: *Partial Dependence Plots*

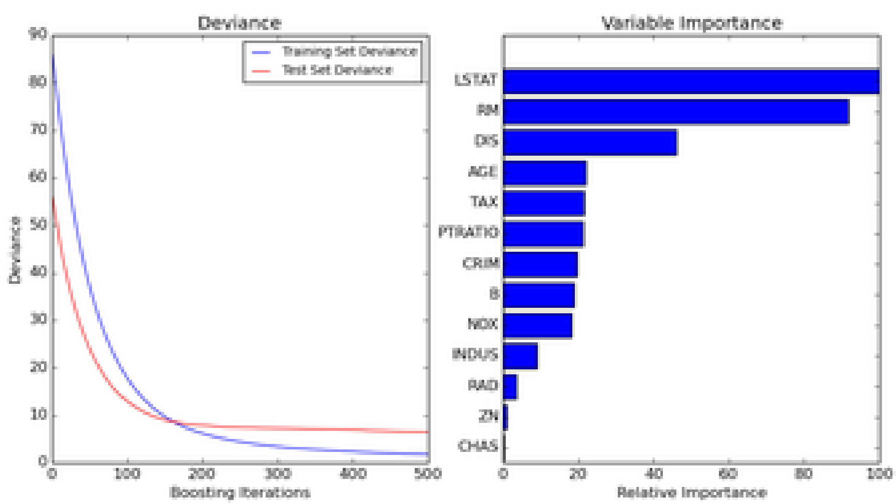


Figure 42.268: *Gradient Boosting regression*

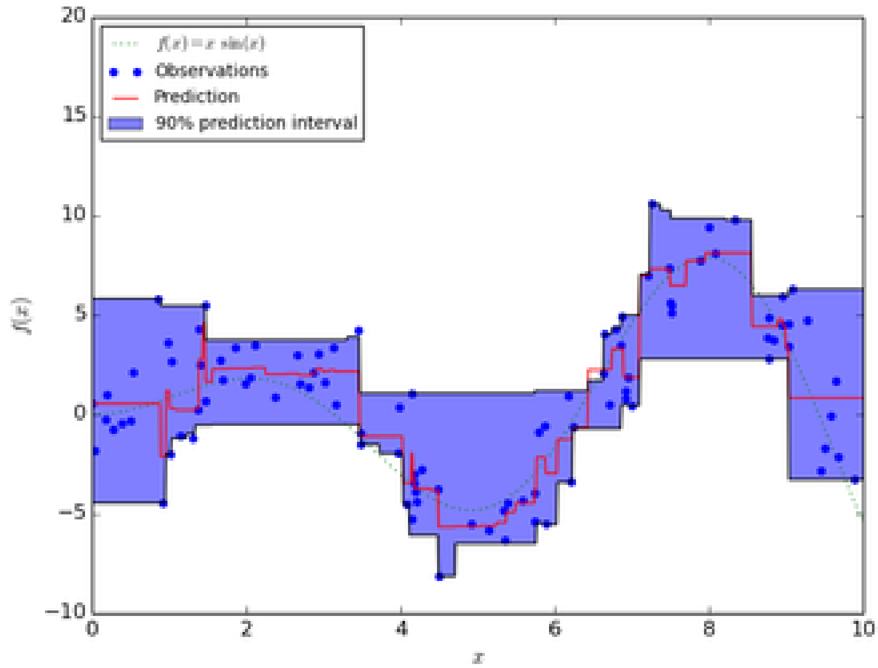


Figure 42.269: *Prediction Intervals for Gradient Boosting Regression*

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting.

Parametersn_estimators : integer, optional (default=10)

The number of trees in the forest.

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_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 percentage 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. 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. Ignored if `max_leaf_nodes` is not None. Note: this parameter is tree-specific.

min_samples_split : integer, optional (default=2)

The minimum number of samples required to split an internal node. Note: this parameter is tree-specific.

min_samples_leaf : integer, optional (default=1)

The minimum number of samples in newly created leaves. A split is discarded if after the split, one of the leaves would contain less then `min_samples_leaf` samples. Note: this parameter is tree-specific.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node. Note: this parameter is tree-specific.

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. If not None then `max_depth` will be ignored. Note: this parameter is tree-specific.

bootstrap : boolean, optional (default=True)

Whether bootstrap samples are used when building trees.

oob_score : bool

Whether to use out-of-bag samples to estimate the generalization error.

n_jobs : integer, optional (default=1)

The number of jobs to run in parallel for both *fit* and *predict*. If -1, then the number of jobs is set to the number of cores.

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.

class_weight : dict, list of dicts, “auto”, “subsample” 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.

The “auto” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data.

The “subsample” mode is the same as “auto” 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.

Attributesestimators_ : 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_ : array of shape = [n_features]

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.

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.

See also:

`DecisionTreeClassifier`, `ExtraTreesClassifier`

References

[R134]

Methods

`apply`(X)	Apply trees in the forest to X, return leaf indices.
`fit`(X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict class for X.
`predict_log_proba`(X)	Predict class log-probabilities for X.
`predict_proba`(X)	Predict class probabilities for X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, threshold])	Reduce X to its most important features.

static `__init__`(*n_estimators=10, 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, bootstrap=True, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False, class_weight=None*)

apply(*X*)
Apply trees in the forest to X, return leaf indices.

ParametersX : 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`.

ReturnsX_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.

fit(*X*, *y*, *sample_weight=None*)
Build a forest of trees from the training set (X, y).

ParametersX : array-like or 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, shape = [n_samples] or [n_samples, n_outputs]

The target values (class labels in classification, real numbers in regression).

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.

Returnsself : object

Returns self.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)
Predict class for X.

The predicted class of an input sample is computed as the majority prediction of the trees in the forest.

ParametersX : 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`.

Returnsy : array of shape = [n_samples] or [n_samples, n_outputs]

The predicted classes.

predict_log_proba (*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.

ParametersX : 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`.

Returnsp : 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 (*X*)
Predict class probabilities for X.

The predicted class probabilities of an input sample is 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.

ParametersX : 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`.

Returnsp : 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 (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Examples using `sklearn.ensemble.RandomForestClassifier`

42.9.10 `sklearn.ensemble`.RandomTreesEmbedding

class `sklearn.ensemble.`**RandomTreesEmbedding**(*n_estimators=10*, *max_depth=5*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_leaf_nodes=None*, *sparse_output=True*, *n_jobs=1*, *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`.

Parametersn_estimators : int

Number of trees in the forest.

max_depth : int

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. Ignored if `max_leaf_nodes` is not None.

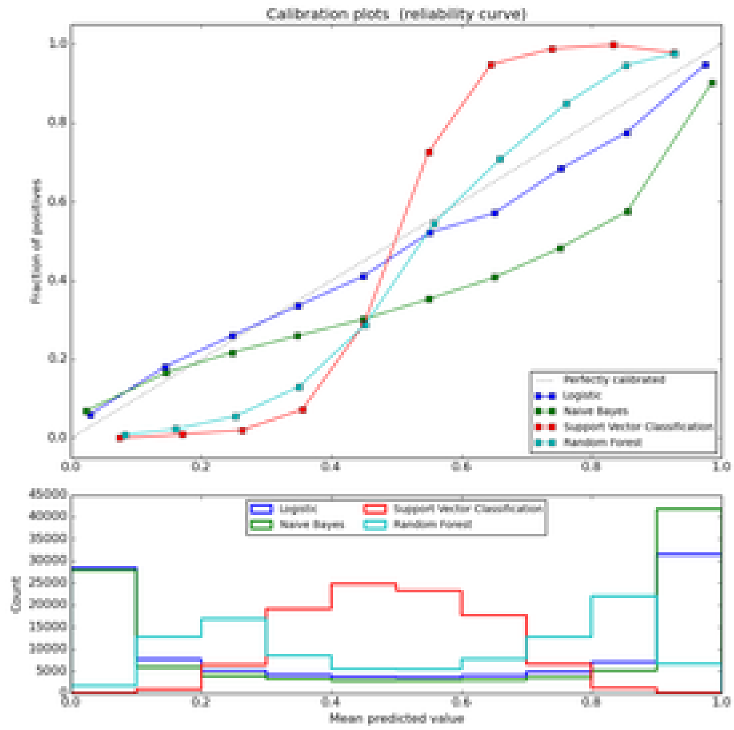


Figure 42.270: *Comparison of Calibration of Classifiers*

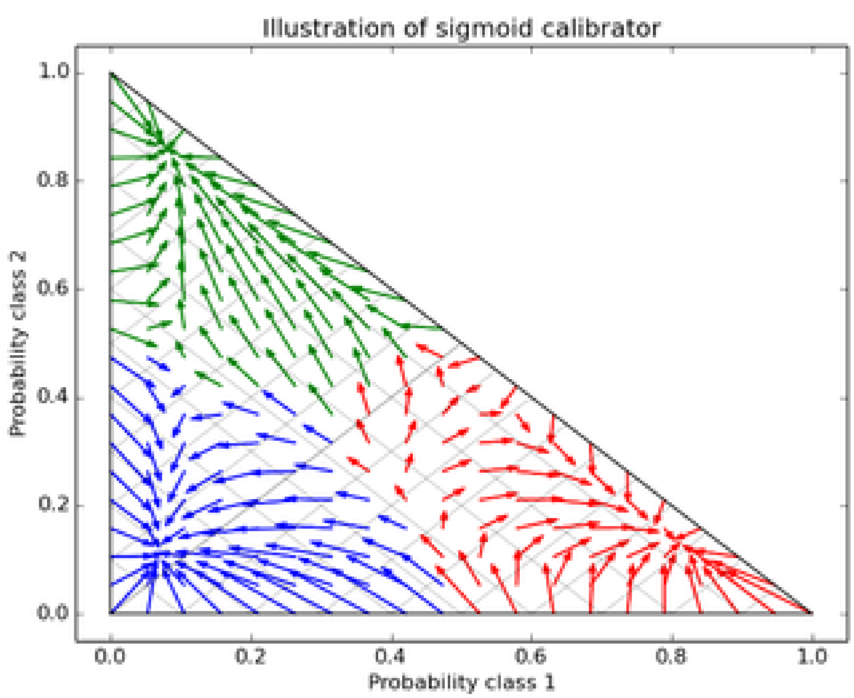


Figure 42.271: *Probability Calibration for 3-class classification*

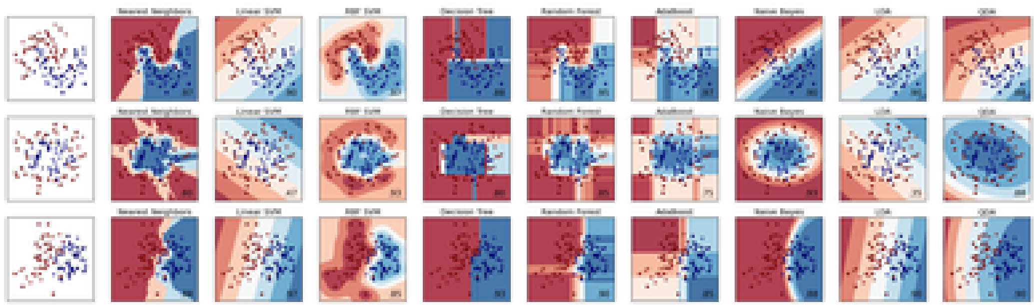


Figure 42.272: *Classifier comparison*

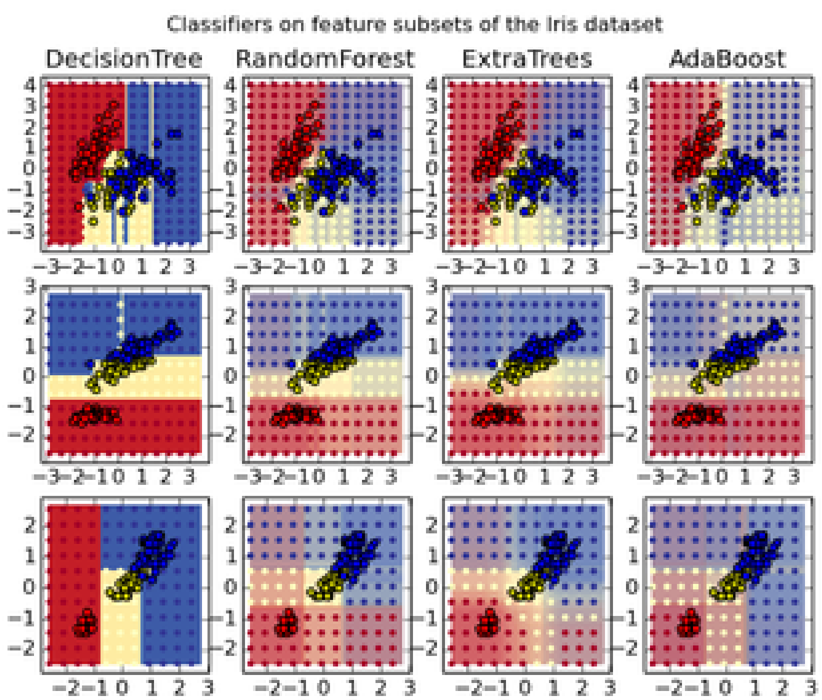


Figure 42.273: *Plot the decision surfaces of ensembles of trees on the iris dataset*

Figure 42.274: *Comparing randomized search and grid search for hyperparameter estimation*

Figure 42.275: *Classification of text documents using sparse features*

min_samples_split : integer, optional (default=2)

The minimum number of samples required to split an internal node.

min_samples_leaf : integer, optional (default=1)

The minimum number of samples in newly created leaves. A split is discarded if after the split, one of the leaves would contain less then `min_samples_leaf` samples.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node.

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. If not None then `max_depth` will be ignored.

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 : integer, optional (default=1)

The number of jobs to run in parallel for both *fit* and *predict*. If -1, then the number of jobs is set to the number of cores.

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.

Attributesestimators_ : list of DecisionTreeClassifier

The collection of fitted sub-estimators.

References

[R136], [R137]

Methods

`apply`(X)	Apply trees in the forest to X, return leaf indices.
`fit`(X[, y, sample_weight])	Fit estimator.
`fit_transform`(X[, y, sample_weight])	Fit estimator and transform dataset.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Transform dataset.

static __init__(*n_estimators=10*, *max_depth=5*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_leaf_nodes=None*, *sparse_output=True*, *n_jobs=1*, *random_state=None*, *verbose=0*, *warm_start=False*)

apply(*X*)
Apply trees in the forest to X, return leaf indices.

ParametersX : 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`.

ReturnsX_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.

static fit(*X*, *y=None*, *sample_weight=None*)
Fit estimator.

ParametersX : 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.

Returnsself : object

Returns self.

static fit_transform(*X*, *y=None*, *sample_weight=None*)
Fit estimator and transform dataset.

ParametersX : array-like or sparse matrix, shape=(n_samples, n_features)

Input data used to build forests. Use `dtype=np.float32` for maximum efficiency.

ReturnsX_transformed : sparse matrix, shape=(n_samples, n_out)

Transformed dataset.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static transform(*X*)
Transform dataset.

ParametersX : 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.

ReturnsX_transformed : sparse matrix, shape=(n_samples, n_out)

Transformed dataset.

Examples using `sklearn.ensemble.RandomTreesEmbedding`

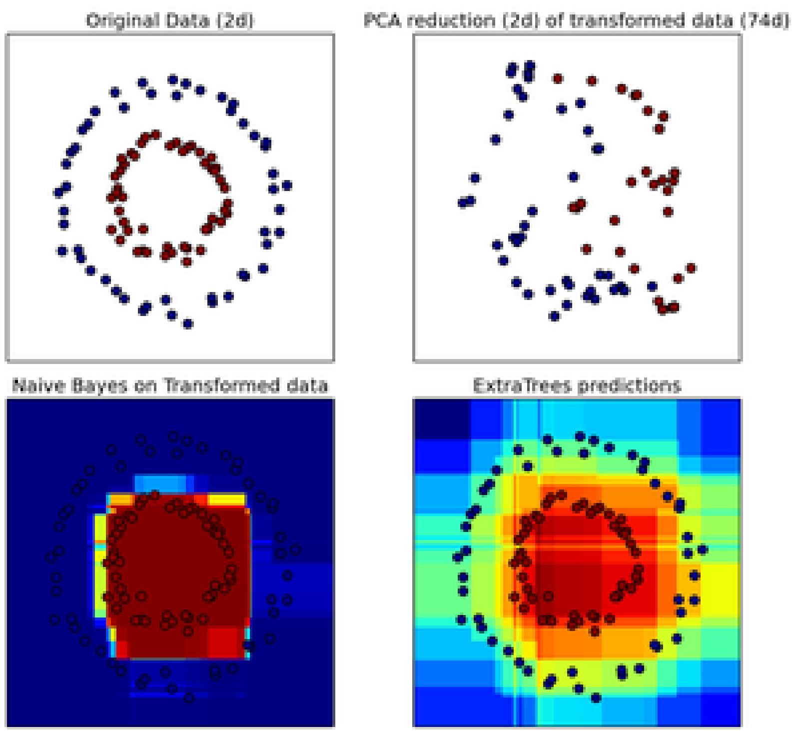


Figure 42.276: *Hashing feature transformation using Totally Random Trees*

42.9.11 `sklearn.ensemble`.RandomForestRegressor

class `sklearn.ensemble.`**`RandomForestRegressor`**(*n_estimators=10*, *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*, *bootstrap=True*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*, *warm_start=False*)

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 use averaging to improve the predictive accuracy and control over-fitting.

Parametersn_estimators : integer, optional (default=10)

The number of trees in the forest.

Figure 42.277: *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

criterion : string, optional (default="mse")

The function to measure the quality of a split. The only supported criterion is "mse" for the mean squared error. Note: this parameter is tree-specific.

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 percentage 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. 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. Ignored if `max_leaf_nodes` is not None. Note: this parameter is tree-specific.

min_samples_split : integer, optional (default=2)

The minimum number of samples required to split an internal node. Note: this parameter is tree-specific.

min_samples_leaf : integer, optional (default=1)

The minimum number of samples in newly created leaves. A split is discarded if after the split, one of the leaves would contain less then `min_samples_leaf` samples. Note: this parameter is tree-specific.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node. Note: this parameter is tree-specific.

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. If not None then `max_depth` will be ignored. Note: this parameter is tree-specific.

bootstrap : boolean, optional (default=True)

Whether bootstrap samples are used when building trees.

oob_score : bool

whether to use out-of-bag samples to estimate the generalization error.

n_jobs : integer, optional (default=1)

The number of jobs to run in parallel for both *fit* and *predict*. If -1, then the number of jobs is set to the number of cores.

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.

Attributesestimators_ : list of DecisionTreeRegressor

The collection of fitted sub-estimators.

feature_importances_ : array of shape = [n_features]

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.

oob_prediction_ : array of shape = [n_samples]

Prediction computed with out-of-bag estimate on the training set.

See also:

`DecisionTreeRegressor`, `ExtraTreesRegressor`

References

[R135]

Methods

apply(X)	Apply trees in the forest to X, return leaf indices.
fit(X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict regression target for X.
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.
transform(X[, threshold])	Reduce X to its most important features.

static __init__(*n_estimators=10, 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, bootstrap=True, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False*)

apply(*X*)

Apply trees in the forest to X, return leaf indices.

ParametersX : 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`.

ReturnsX_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.

fit(*X, y, sample_weight=None*)

Build a forest of trees from the training set (X, y).

ParametersX : array-like or 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, shape = [n_samples] or [n_samples, n_outputs]

The target values (class labels in classification, real numbers in regression).

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.

Returnsself : object

Returns self.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*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.

ParametersX : 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`.

Returnsy : array of shape = [n_samples] or [n_samples, n_outputs]

The predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Examples using `sklearn.ensemble.RandomForestRegressor`

Figure 42.278: *Imputing missing values before building an estimator*

42.9.12 partial dependence

Partial dependence plots for tree ensembles.

`ensemble.partial_dependence.partial_dependence`(...)	Partial dependence of `target_variables`.
`ensemble.partial_dependence.plot_partial_dependence`(...)	Partial dependence plots for `features`.

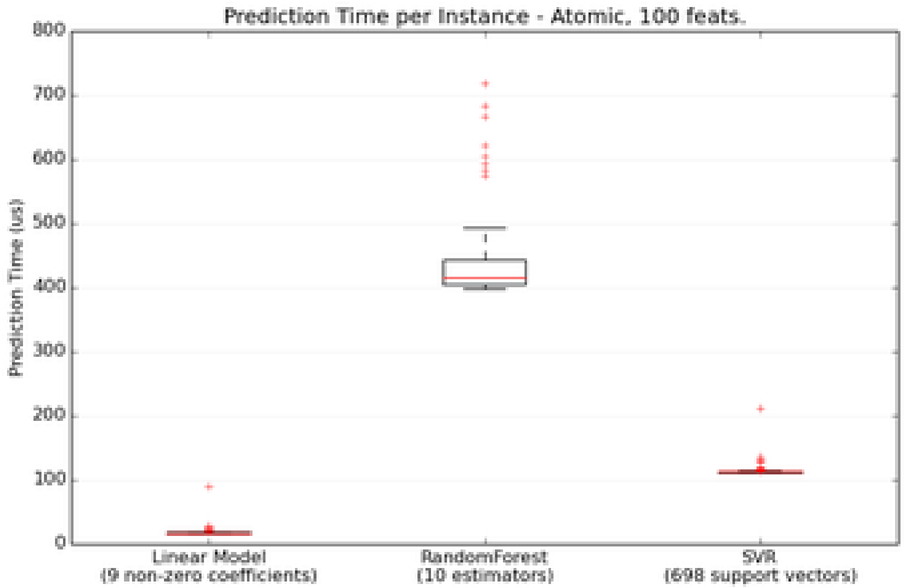


Figure 42.279: *Prediction Latency*

`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*)

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`.

Parametersgbrt : BaseGradientBoosting

A fitted gradient boosting model.

target_variables : array-like, dtype=int

The target features for which the partial dependecy should be computed (size should be smaller than 3 for visual renderings).

grid : array-like, shape=(n_points, len(target_variables))

The grid of `target_variables` values for which the partial dependecy should be evaluated (either `grid` or `X` must be specified).

X : array-like, 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`.

Returnspdp : 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)
(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=1, verbose=0, ax=None, line_kw=None, contour_kw=None, **fig_kw*)

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.

Parametersgbrt : BaseGradientBoosting

A fitted gradient boosting model.

X : array-like, shape=(n_samples, n_features)

The data on which `gbrt` was trained.

features : seq of tuples or ints

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.

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).

percentiles : (low, high), default=(0.05, 0.95)

The lower and upper percentile used create the extreme values for the PDP axes.

grid_resolution : int, default=100

The number of equally spaced points on the axes.

n_jobs : int

The number of CPUs to use to compute the PDs. -1 means 'all CPUs'. Defaults to 1.

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 `pylab.plot` call. For one-way partial dependence plots.

contour_kw : dict

Dict with keywords passed to the `pylab.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.

Returnsfig : 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)])
...
```

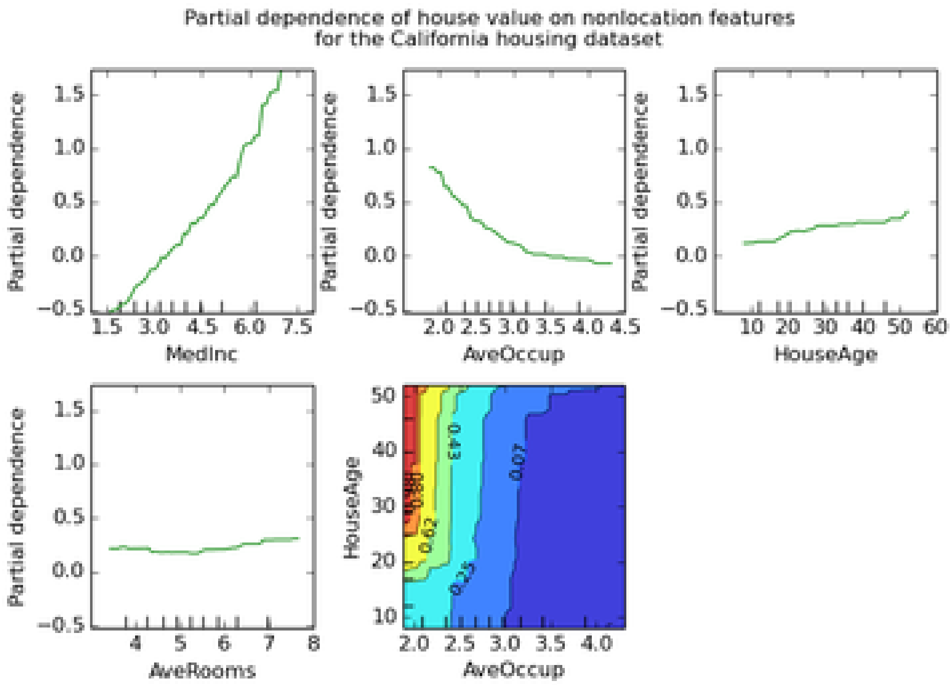


Figure 42.280: *Partial Dependence Plots*

Examples using `sklearn.ensemble.partial_dependence.plot_partial_dependence`

42.10 `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.

42.10.1 `sklearn.feature_extraction`.DictVectorizer

class `sklearn.feature_extraction.`**`DictVectorizer`**(*dtype=<type '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".

Features that do not occur in a sample (mapping) will have a zero value in the resulting array/matrix.

Parametersdtype : callable, optional

The type of feature values. Passed to Numpy array/scipy.sparse matrix constructors as the dtype argument.

separator: string, optional :

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`(X[, y])	Learn a list of feature name -> indices mappings.
`fit_transform`(X[, y])	Learn a list of feature name -> indices mappings and transform X.
`get_feature_names`()	Returns a list of feature names, ordered by their indices.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(X[, dict_type])	Transform array or sparse matrix X back to feature mappings.
`restrict`(support[, indices])	Restrict the features to those in support.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y])	Transform feature->value dicts to array or sparse matrix.

static `__init__`(*dtype=<type 'numpy.float64'>, separator='=', sparse=True, sort=True*)

static `fit`(*X, y=None*)

Learn a list of feature name -> indices mappings.

ParametersX : 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)

Returnsself :

static `fit_transform`(*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.

ParametersX : 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)

ReturnsXa : {array, sparse matrix}

Feature vectors; always 2-d.

static `get_feature_names`()

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(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **inverse_transform**(*X*, *dict_type=<type '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.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Sample matrix.

dict_type : callable, optional

Constructor for feature mappings. Must conform to the collections.Mapping API.

ReturnsD : list of dict_type objects, length = n_samples

Feature mappings for the samples in X.

static **restrict**(*support*, *indices=False*)
Restrict the features to those in support.

Parameterssupport : 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.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **transform**(*X*, *y=None*)
Transform feature->value dicts to array or sparse matrix.

Named features not encountered during fit or fit_transform will be silently ignored.

ParametersX : 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).

y : (ignored)

ReturnsXa : {array, sparse matrix}

Feature vectors; always 2-d.

Examples using `sklearn.feature_extraction.DictVectorizer`

Figure 42.281: *Feature Union with Heterogeneous Data Sources*

Figure 42.282: *FeatureHasher and DictVectorizer Comparison*

42.10.2 `sklearn.feature_extraction`.FeatureHasher

class `sklearn.feature_extraction.`**`FeatureHasher`**(*n_features=1048576*, *input_type='dict'*, *dtype=<type 'numpy.float64'>*, *non_negative=False*)

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.

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.

Parametersn_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.

dtype : numpy type, optional

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.

input_type : string, optional

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).

non_negative : boolean, optional, default np.float64

Whether output matrices should contain non-negative values only; effectively calls abs on the matrix prior to returning it. When True, output values can be interpreted as frequencies. When False, output values will have expected value zero.

See also:

`DictVectorizer`vectorizes string-valued features using a hash table.

`sklearn.preprocessing.OneHotEncoder`handles nominal/categorical features encoded as columns of integers.

Methods

`fit`([X, y])	No-op.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(raw_X[, y])	Transform a sequence of instances to a scipy.sparse matrix.

static `__init__`(*n_features=1048576*, *input_type='dict'*, *dtype=<type 'numpy.float64'>*, *non_negative=False*)

static `fit`(*X=None*, *y=None*)

No-op.

This method doesn't do anything. It exists purely for compatibility with the scikit-learn transformer API.

Returnsself : FeatureHasher

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **transform** (*raw_X*, *y=None*)

Transform a sequence of instances to a scipy.sparse matrix.

Parametersraw_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.

y : (ignored)

ReturnsX : scipy.sparse matrix, shape = (n_samples, self.n_features)

Feature matrix, for use with estimators or further transformers.

Examples using sklearn.feature_extraction.FeatureHasher

Figure 42.283: *FeatureHasher and DictVectorizer Comparison*

42.10.3 From images

The `sklearn.feature_extraction.image` submodule gathers utilities to extract features from images.

`feature_extraction.image.img_to_graph`(img[, ...])	Graph of the pixel-to-pixel gradient connections
`feature_extraction.image.grid_to_graph`(n_x, n_y)	Graph of the pixel-to-pixel connections
`feature_extraction.image.extract_patches_2d`(...)	Reshape a 2D image into a collection of patches
`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.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.

Parametersimg: 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 sklearn 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.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=<type 'int'>*)

Graph of the pixel-to-pixel connections

Edges exist if 2 voxels are connected.

Parametersn_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 sklearn 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.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.

Parametersimage: 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 or RandomState :

Pseudo number generator state used for random sampling to use if *max_patches* is not None.

Returnspatches: 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.feature_extraction import image
>>> one_image = np.arange(16).reshape((4, 4))
>>> one_image
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> patches = image.extract_patches_2d(one_image, (2, 2))
>>> print(patches.shape)
(9, 2, 2)
>>> patches[0]
array([[0, 1],
       [4, 5]])
>>> patches[1]
array([[1, 2],
       [5, 6]])
>>> patches[8]
array([[10, 11],
       [14, 15]])
```

Examples using `sklearn.feature_extraction.image.extract_patches_2d`

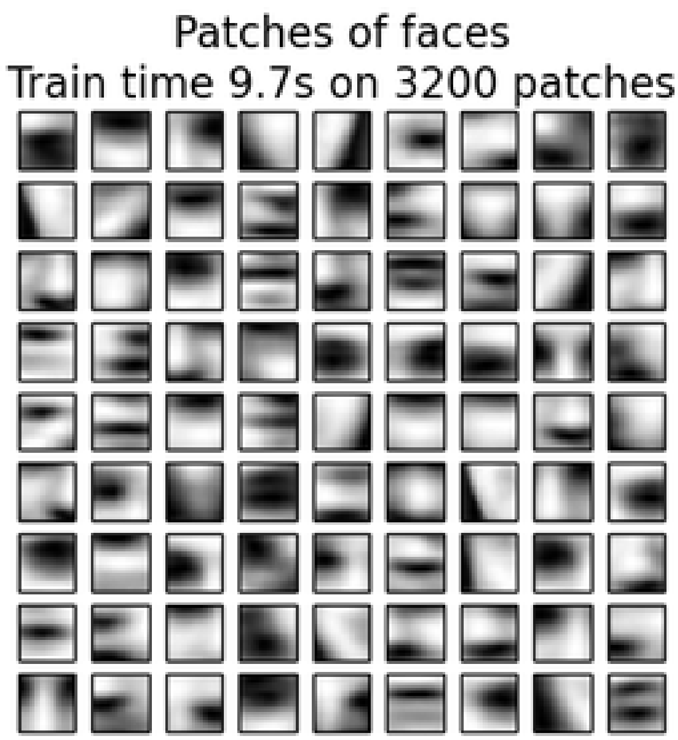


Figure 42.284: *Online learning of a dictionary of parts of faces*

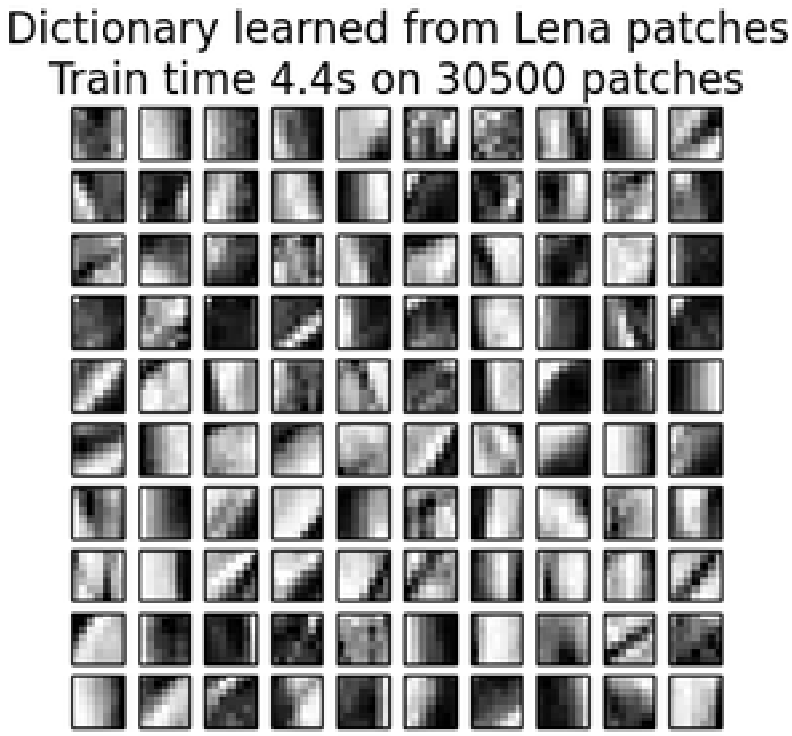


Figure 42.285: *Image denoising using dictionary learning*

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.

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

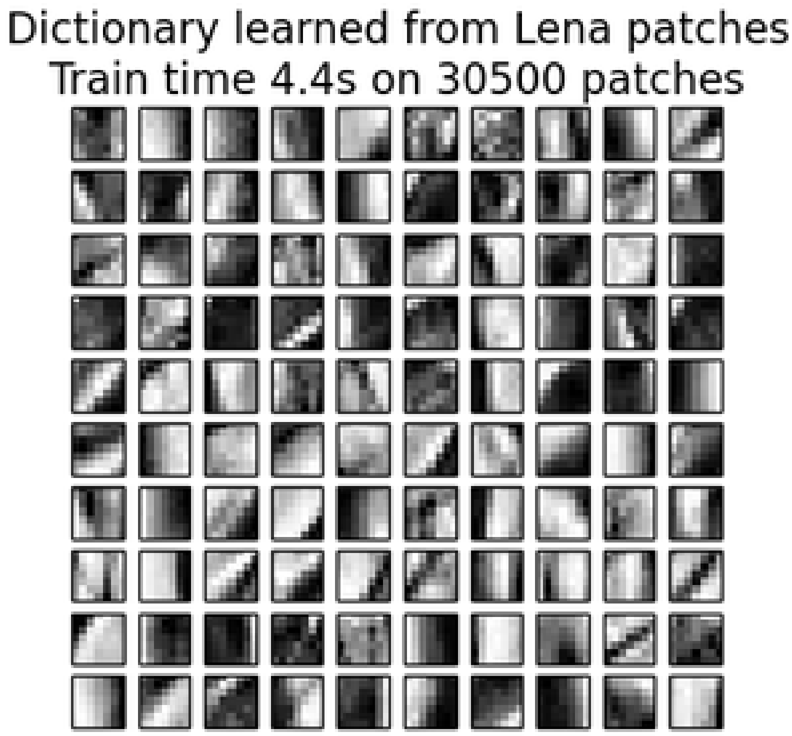


Figure 42.286: *Image denoising using dictionary learning*

Examples using `sklearn.feature_extraction.image.reconstruct_from_patches_2d`

`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

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 or RandomState :

Pseudo number generator state used for random sampling.

Methods

`fit`(X[, y])	Do nothing and return the estimator unchanged
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.

Continued on next page

Table 42.75 – continued from previous page

transform(X)	Transforms the image samples in X into a matrix of patch data.

static __init__(*patch_size=None*, *max_patches=None*, *random_state=None*)

static fit(*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.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static transform(*X*)

Transforms the image samples in X into a matrix of patch data.

ParametersX : 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*.

Returnspatches: 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.

42.10.4 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 coun
feature_extraction.text.HashingVectorizer([...])	Convert a collection of text documents to a matrix of token occu
feature_extraction.text.TfidfTransformer([...])	Transform a count matrix to a normalized tf or tf-idf representati
feature_extraction.text.TfidfVectorizer([...])	Convert a collection of raw documents to a matrix of TF-IDF fea

sklearn.feature_extraction.text.CountVectorizer

class sklearn.feature_extraction.text.**CountVectorizer**(*input=u'content', encoding=u'utf-8', decode_error=u'strict', strip_accents=None, lowercase=True, preprocessor=None, tokenizer=None, stop_words=None, token_pattern=u'(?u)\b\w\w+\b', ngram_range=(1, 1), analyzer=u'word', max_df=1.0, min_df=1, max_features=None, vocabulary=None, binary=False, dtype=<type '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.coo_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.

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 the sequence strings or bytes items are expected to be analyzed directly.

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 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.

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.

If a callable is passed it is used to extract the sequence of features out of the raw, unprocessed input.

preprocessor : callable or None (default)

Override the preprocessing (string transformation) stage while preserving the tokenizing and n-grams generation steps.

tokenizer : callable or None (default)

Override the string tokenization step while preserving the preprocessing and n-grams generation steps.

ngram_range : tuple (min_n, max_n)

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.

stop_words : string {'english'}, list, or None (default)

If 'english', a built-in stop word list for English is used.

If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens.

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.

lowercase : boolean, True by default

Convert all characters to lowercase before tokenizing.

token_pattern : string

Regular expression denoting what constitutes a "token", only used if *tokenize* == *'word'*. The default regexp select tokens of 2 or more alphanumeric characters (punctuation is completely ignored and always treated as a token separator).

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().

Attributesvocabulary_ : dict

A mapping of terms to feature indices.

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.

Methods

`build_analyzer`()	Return a callable that handles preprocessing and tokenization
`build_preprocessor`()	Return a function to preprocess the text before tokenization
`build_tokenizer`()	Return a function that splits a string into a sequence of tokens
`decode`(doc)	Decode the input into a string of unicode symbols
`fit`(raw_documents[, y])	Learn a vocabulary dictionary of all tokens in the raw documents.
`fit_transform`(raw_documents[, y])	Learn the vocabulary dictionary and return term-document matrix.
`get_feature_names`()	Array mapping from feature integer indices to feature name
`get_params`([deep])	Get parameters for this estimator.
`get_stop_words`()	Build or fetch the effective stop words list
`inverse_transform`(X)	Return terms per document with nonzero entries in X.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(raw_documents)	Transform documents to document-term matrix.

static `__init__`(*input=u'content', encoding=u'utf-8', decode_error=u'strict', strip_accents=None, lowercase=True, preprocessor=None, tokenizer=None, stop_words=None, token_pattern=u'(?u)\\b\\w\\w+\\b', ngram_range=(1, 1), analyzer=u'word', max_df=1.0, min_df=1, max_features=None, vocabulary=None, binary=False, dtype=<type 'numpy.int64'>*)

`build_analyzer`()

Return a callable that handles preprocessing and tokenization

`build_preprocessor`()

Return a function to preprocess the text before tokenization

`build_tokenizer`()

Return a function that splits a string into a sequence of tokens

`decode`(*doc*)

Decode the input into a string of unicode symbols

The decoding strategy depends on the vectorizer parameters.

static **fit** (*raw_documents*, *y=None*)
Learn a vocabulary dictionary of all tokens in the raw documents.

Parametersraw_documents : iterable

An iterable which yields either str, unicode or file objects.

Returnsself :

static **fit_transform** (*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.

Parametersraw_documents : iterable

An iterable which yields either str, unicode or file objects.

ReturnsX : array, [n_samples, n_features]

Document-term matrix.

static **get_feature_names** ()
Array mapping from feature integer indices to feature name

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_stop_words ()
Build or fetch the effective stop words list

static **inverse_transform** (*X*)
Return terms per document with nonzero entries in X.

ParametersX : {array, sparse matrix}, shape = [n_samples, n_features]

ReturnsX_inv : list of arrays, len = n_samples

List of arrays of terms.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static **transform** (*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.

Parametersraw_documents : iterable

An iterable which yields either str, unicode or file objects.

ReturnsX : sparse matrix, [n_samples, n_features]

Document-term matrix.

Examples using `sklearn.feature_extraction.text.CountVectorizer`

Figure 42.287: *Sample pipeline for text feature extraction and evaluation*

`sklearn.feature_extraction.text.HashingVectorizer`

class `sklearn.feature_extraction.text.`**`HashingVectorizer`**(*input=u'content', encoding=u'utf-8', decode_error=u'strict', strip_accents=None, lowercase=True, preprocessor=None, tokenizer=None, stop_words=None, token_pattern=u'(?u)\b\w\w+\b', ngram_range=(1, 1), analyzer=u'word', n_features=1048576, binary=False, norm=u'l2', non_negative=False, dtype=<type '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.

Parametersinput : 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 the sequence strings or bytes items are expected to be analyzed directly.

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 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.

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.

If a callable is passed it is used to extract the sequence of features out of the raw, unprocessed input.

preprocessor : callable or None (default)

Override the preprocessing (string transformation) stage while preserving the tokenizing and n-grams generation steps.

tokenizer : callable or None (default)

Override the string tokenization step while preserving the preprocessing and n-grams generation steps.

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.

stop_words : string {'english'}, list, or None (default)

If 'english', a built-in stop word list for English is used.

If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens.

lowercase : boolean, default=True

Convert all characters to lowercase before tokenizing.

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).

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.

norm : 'l1', 'l2' or None, optional

Norm used to normalize term vectors. None for no normalization.

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().

non_negative : boolean, default=False

Whether output matrices should contain non-negative values only; effectively calls abs on the matrix prior to returning it. When True, output values can be interpreted as frequencies. When False, output values will have expected value zero.

See also:

`CountVectorizer`, `TfidfVectorizer`

Methods

`build_analyzer`()	Return a callable that handles preprocessing and tokenization
`build_preprocessor`()	Return a function to preprocess the text before tokenization
`build_tokenizer`()	Return a function that splits a string into a sequence of tokens
`decode`(doc)	Decode the input into a string of unicode symbols
`fit`(X[, y])	Does nothing: this transformer is stateless.
`fit_transform`(X[, y])	Transform a sequence of documents to a document-term matrix.
`get_params`([deep])	Get parameters for this estimator.
`get_stop_words`()	Build or fetch the effective stop words list
`partial_fit`(X[, y])	Does nothing: this transformer is stateless.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y])	Transform a sequence of documents to a document-term matrix.

static __init__(*input=u'content'*, *encoding=u'utf-8'*, *decode_error=u'strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *stop_words=None*, *token_pattern=u'(?u)\\b\\w\\w+\\b'*, *ngram_range=(1, 1)*, *analyzer=u'word'*, *n_features=1048576*, *binary=False*, *norm=u'l2'*, *non_negative=False*, *dtype=<type 'numpy.float64'>*)

build_analyzer()
Return a callable that handles preprocessing and tokenization

build_preprocessor()
Return a function to preprocess the text before tokenization

build_tokenizer()
Return a function that splits a string into a sequence of tokens

decode(*doc*)
Decode the input into a string of unicode symbols

The decoding strategy depends on the vectorizer parameters.

static fit(*X*, *y=None*)
Does nothing: this transformer is stateless.

static fit_transform(*X*, *y=None*)
Transform a sequence of documents to a document-term matrix.

ParametersX : 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 : (ignored)

ReturnsX : scipy.sparse matrix, shape = (n_samples, self.n_features)

Document-term matrix.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_stop_words()
Build or fetch the effective stop words list

static partial_fit(*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.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static transform (*X*, *y=None*)

Transform a sequence of documents to a document-term matrix.

ParametersX : 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 : (ignored)

ReturnsX : scipy.sparse matrix, shape = (n_samples, self.n_features)

Document-term matrix.

Examples using `sklearn.feature_extraction.text.HashingVectorizer`

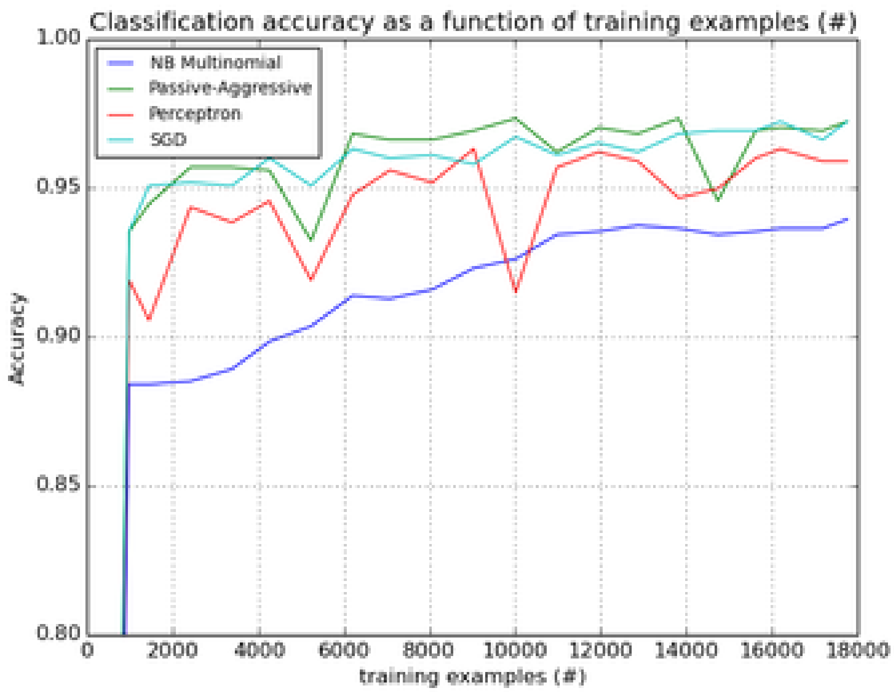


Figure 42.288: *Out-of-core classification of text documents*

`sklearn.feature_extraction.text`.TfidfTransformer

class `sklearn.feature_extraction.text.`**`TfidfTransformer`**(*norm=u'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.

Figure 42.289: *Clustering text documents using k-means*

Figure 42.290: *Classification of text documents using sparse features*

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 actual formula used for tf-idf is tf * (idf + 1) = tf + tf * idf, instead of tf * idf. The effect of this is that terms with zero idf, i.e. that occur in all documents of a training set, will not be entirely ignored. 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.

Parametersnorm : 'l1', 'l2' or None, optional

Norm used to normalize term vectors. None for no normalization.

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).

References

[Yates2011], [MRS2008]

Methods

`fit`(X[, y])	Learn the idf vector (global term weights)
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, copy])	Transform a count matrix to a tf or tf-idf representation

static `__init__`(*norm=u'l2'*, *use_idf=True*, *smooth_idf=True*, *sublinear_tf=False*)

static `fit`(*X*, *y=None*)

Learn the idf vector (global term weights)

ParametersX : sparse matrix, [n_samples, n_features]

a matrix of term/token counts

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static **transform** (*X*, *copy=True*)

Transform a count matrix to a tf or tf-idf representation

ParametersX : 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.

Returnsvectors : sparse matrix, [n_samples, n_features]

Examples using `sklearn.feature_extraction.text.TfidfTransformer`

Figure 42.291: *Sample pipeline for text feature extraction and evaluation*

Figure 42.292: *Clustering text documents using k-means*

sklearn.feature_extraction.text.TfidfVectorizer

class sklearn.feature_extraction.text.**TfidfVectorizer**(*input=u'content'*, *encoding=u'utf-8'*, *decode_error=u'strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *analyzer=u'word'*, *stop_words=None*, *token_pattern=u'(?u)\b\w\w+\b'*, *ngram_range=(1, 1)*, *max_df=1.0*, *min_df=1*, *max_features=None*, *vocabulary=None*, *binary=False*, *dtype=<type 'numpy.int64'>*, *norm=u'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.

Parametersinput : 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 the sequence strings or bytes items are expected to be analyzed directly.

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 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.

analyzer : string, {'word', 'char'} or callable

Whether the feature should be made of word or character n-grams.

If a callable is passed it is used to extract the sequence of features out of the raw, unprocessed input.

preprocessor : callable or None (default)

Override the preprocessing (string transformation) stage while preserving the tokenizing and n-grams generation steps.

tokenizer : callable or None (default)

Override the string tokenization step while preserving the preprocessing and n-grams generation steps.

ngram_range : tuple (min_n, max_n)

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.

stop_words : string {'english'}, list, or None (default)

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.

If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens.

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.

lowercase : boolean, default True

Convert all characters to lowercase before tokenizing.

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).

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.

binary : boolean, 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

Type of the matrix returned by fit_transform() or transform().

norm : 'l1', 'l2' or None, optional

Norm used to normalize term vectors. None for no normalization.

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).

Attributesidf_ : array, shape = [n_features], or None

The learned idf vector (global term weights) when `use_idf` is set to True, None otherwise.

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`Tokenize the documents and count the occurrences of token and return them as a sparse matrix

`TfidfTransformer`Apply Term Frequency Inverse Document Frequency normalization to a sparse matrix of occurrence 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.

Methods

`build_analyzer`()	Return a callable that handles preprocessing and tokenization
`build_preprocessor`()	Return a function to preprocess the text before tokenization
`build_tokenizer`()	Return a function that splits a string into a sequence of tokens
`decode`(doc)	Decode the input into a string of unicode symbols
`fit`(raw_documents[, y])	Learn vocabulary and idf from training set.
`fit_transform`(raw_documents[, y])	Learn vocabulary and idf, return term-document matrix.
`get_feature_names`()	Array mapping from feature integer indices to feature name
`get_params`([deep])	Get parameters for this estimator.
`get_stop_words`()	Build or fetch the effective stop words list
`inverse_transform`(X)	Return terms per document with nonzero entries in X.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(raw_documents[, copy])	Transform documents to document-term matrix.

static **`__init__`**(*input=u'content'*, *encoding=u'utf-8'*, *decode_error=u'strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *analyzer=u'word'*, *stop_words=None*, *token_pattern=u'(?u)\\b\\w\\w+\\b'*, *ngram_range=(1, 1)*, *max_df=1.0*, *min_df=1*, *max_features=None*, *vocabulary=None*, *binary=False*, *dtype=<type 'numpy.int64'>*, *norm=u'l2'*, *use_idf=True*, *smooth_idf=True*, *sublinear_tf=False*)

`build_analyzer`()
Return a callable that handles preprocessing and tokenization

`build_preprocessor`()
Return a function to preprocess the text before tokenization

`build_tokenizer`()
Return a function that splits a string into a sequence of tokens

`decode`(*doc*)
Decode the input into a string of unicode symbols

The decoding strategy depends on the vectorizer parameters.

static **`fit`**(*raw_documents*, *y=None*)
Learn vocabulary and idf from training set.

Parametersraw_documents : iterable

an iterable which yields either str, unicode or file objects

Returnsself : TfidfVectorizer

static **`fit_transform`**(*raw_documents*, *y=None*)
Learn vocabulary and idf, return term-document matrix.

This is equivalent to fit followed by transform, but more efficiently implemented.

Parametersraw_documents : iterable

an iterable which yields either str, unicode or file objects

ReturnsX : sparse matrix, [n_samples, n_features]

Tf-idf-weighted document-term matrix.

get_feature_names()

Array mapping from feature integer indices to feature name

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_stop_words()

Build or fetch the effective stop words list

inverse_transform(*X*)

Return terms per document with nonzero entries in X.

ParametersX : {array, sparse matrix}, shape = [n_samples, n_features]

ReturnsX_inv : list of arrays, len = n_samples

List of arrays of terms.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **transform**(*raw_documents*, *copy=True*)

Transform documents to document-term matrix.

Uses the vocabulary and document frequencies (df) learned by fit (or fit_transform).

Parametersraw_documents : iterable

an iterable which yields either str, unicode or file objects

copy : boolean, default True

Whether to copy X and operate on the copy or perform in-place operations.

ReturnsX : sparse matrix, [n_samples, n_features]

Tf-idf-weighted document-term matrix.

Examples using `sklearn.feature_extraction.text.TfidfVectorizer`

42.11 `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.

Figure 42.293: *Feature Union with Heterogeneous Data Sources*

Figure 42.294: *Topics extraction with Non-Negative Matrix Factorization*

Figure 42.295: *Biclustering documents with the Spectral Co-clustering algorithm*

Figure 42.296: *Classification of text documents: using a MLComp dataset*

Figure 42.297: *Clustering text documents using k-means*

Figure 42.298: *Classification of text documents using sparse features*

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.SelectFwe([score_func, alpha])`	Filter: Select the p-values corresponding to Family-wise error ra
`feature_selection.RFE(estimator[, ...])`	Feature ranking with recursive feature elimination.
`feature_selection.RFECV(estimator[, step, ...])`	Feature ranking with recursive feature elimination and cross-vali
`feature_selection.VarianceThreshold([threshold])`	Feature selector that removes all low-variance features.

42.11.1 `sklearn.feature_selection`.GenericUnivariateSelect

class `sklearn.feature_selection.`**`GenericUnivariateSelect`**(*score_func=<function f_classif at 0x2ad3d0872c80>, mode='percentile', param=1e-05*)

Univariate feature selector with configurable strategy.

Parameters **score_func** : callable

Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues).

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, shape=(n_features,)

Scores of features.

pvalues_ : array-like, shape=(n_features,)

p-values of feature scores.

See also:

`f_classif` ANOVA F-value between labe/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.

`SelectFwe` Select features based on family-wise error rate.

Methods

`fit`(X, y)	Run score function on (X, y) and get the appropriate features.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`get_support`([indices])	Get a mask, or integer index, of the features selected
`inverse_transform`(X)	Reverse the transformation operation
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Reduce X to the selected features.

static __init__(*score_func=<function f_classif at 0x2ad3d0872c80>, mode='percentile', param=1e-05*)

fit(*X, y*)

Run score function on (X, y) and get the appropriate features.

ParametersX : array-like, shape = [n_samples, n_features]

The training input samples.

y : array-like, shape = [n_samples]

The target values (class labels in classification, real numbers in regression).

Returnsself : object

Returns self.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_support(*indices=False*)

Get a mask, or integer index, of the features selected

Parametersindices : boolean (default False)

If True, the return value will be an array of integers, rather than a boolean mask.

Returnssupport : 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(*X*)

Reverse the transformation operation

ParametersX : array of shape [n_samples, n_selected_features]

The input samples.

ReturnsX_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(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*)

Reduce X to the selected features.

ParametersX : array of shape [n_samples, n_features]

The input samples.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

42.11.2 sklearn.feature_selection.SelectPercentile

class sklearn.feature_selection.**SelectPercentile**(*score_func=<function f_classif at 0x2ad3d0872c80>, percentile=10*)

Select features according to a percentile of the highest scores.

Parametersscore_func : callable

Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues).

percentile : int, optional, default=10

Percent of features to keep.

Attributesscores_ : array-like, shape=(n_features,)

Scores of features.

pvalues_ : array-like, shape=(n_features,)

p-values of feature scores.

See also:

f_classifANOVA F-value between labe/feature for classification tasks.

chi2 Chi-squared stats of non-negative features for classification tasks.

f_regressionF-value between label/feature for regression tasks.

SelectKBestSelect features based on the k highest scores.

SelectFprSelect features based on a false positive rate test.

SelectFdrSelect features based on an estimated false discovery rate.

SelectFweSelect features based on family-wise error rate.

GenericUnivariateSelectUnivariate feature selector with configurable mode.

Notes

Ties between features with equal scores will be broken in an unspecified way.

Methods

fit(X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
get_support([indices])	Get a mask, or integer index, of the features selected
inverse_transform(X)	Reverse the transformation operation
set_params(**params)	Set the parameters of this estimator.
transform(X)	Reduce X to the selected features.

static __init__ (*score_func=<function f_classif at 0x2ad3d0872c80>, percentile=10*)

fit (*X, y*)

Run score function on (X, y) and get the appropriate features.

ParametersX : array-like, shape = [n_samples, n_features]

The training input samples.

y : array-like, shape = [n_samples]

The target values (class labels in classification, real numbers in regression).

Returnsself : object

Returns self.

fit_transform (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_support(*indices=False*)
Get a mask, or integer index, of the features selected

Parametersindices : boolean (default False)

If True, the return value will be an array of integers, rather than a boolean mask.

Returnssupport : 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(*X*)
Reverse the transformation operation

ParametersX : array of shape [n_samples, n_selected_features]

The input samples.

ReturnsX_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(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*)
Reduce X to the selected features.

ParametersX : array of shape [n_samples, n_features]

The input samples.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

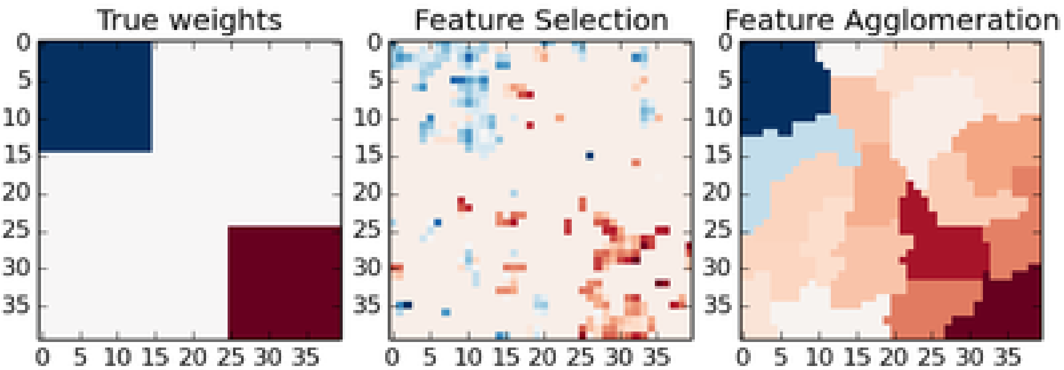


Figure 42.299: *Feature agglomeration vs. univariate selection*

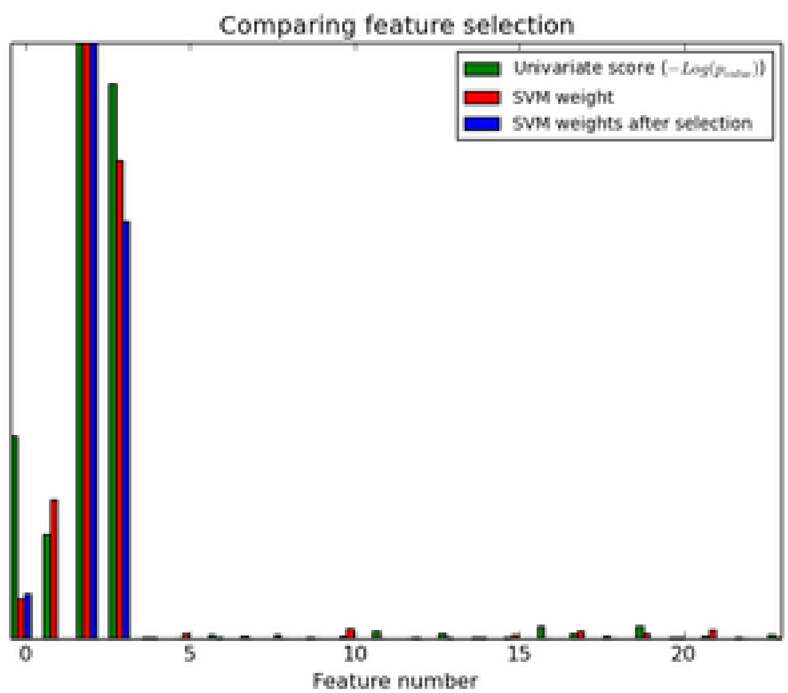


Figure 42.300: *Univariate Feature Selection*

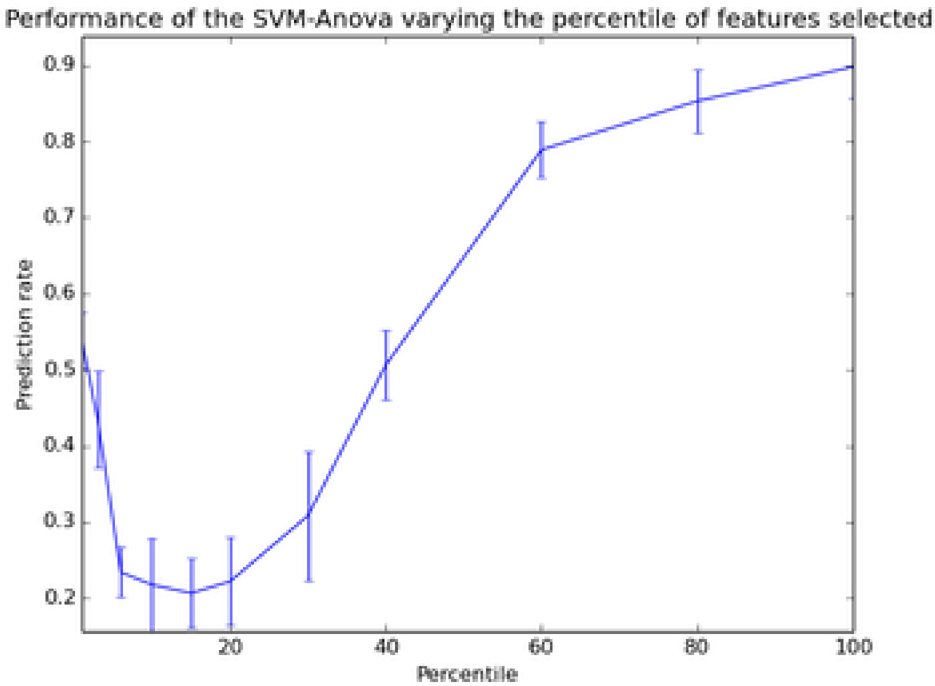


Figure 42.301: *SVM-Anova: SVM with univariate feature selection*

Examples using `sklearn.feature_selection.SelectPercentile`

42.11.3 `sklearn.feature_selection`.SelectKBest

class `sklearn.feature_selection.`**`SelectKBest`**(*score_func=<function f_classif at 0x2ad3d0872c80>, k=10*)

Select features according to the k highest scores.

Parameters score_func : callable

Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues).

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, shape=(n_features,)

Scores of features.

pvalues_ : array-like, shape=(n_features,)

p-values of feature scores.

See also:

`f_classif` ANOVA F-value between labe/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.

SelectPercentileSelect features based on percentile of the highest scores.

SelectFprSelect features based on a false positive rate test.

SelectFdrSelect features based on an estimated false discovery rate.

SelectFweSelect features based on family-wise error rate.

GenericUnivariateSelectUnivariate feature selector with configurable mode.

Notes

Ties between features with equal scores will be broken in an unspecified way.

Methods

fit(X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
get_support([indices])	Get a mask, or integer index, of the features selected
inverse_transform(X)	Reverse the transformation operation
set_params(**params)	Set the parameters of this estimator.
transform(X)	Reduce X to the selected features.

static __init__ (*score_func=<function f_classif at 0x2ad3d0872c80>, k=10*)

fit (*X, y*)
Run score function on (X, y) and get the appropriate features.

ParametersX : array-like, shape = [n_samples, n_features]

The training input samples.

y : array-like, shape = [n_samples]

The target values (class labels in classification, real numbers in regression).

Returnsself : object

Returns self.

fit_transform (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_support (*indices=False*)

Get a mask, or integer index, of the features selected

Parametersindices : boolean (default False)

If True, the return value will be an array of integers, rather than a boolean mask.

Returnssupport : 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 (*X*)

Reverse the transformation operation

ParametersX : array of shape [n_samples, n_selected_features]

The input samples.

ReturnsX_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 (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*)

Reduce X to the selected features.

ParametersX : array of shape [n_samples, n_features]

The input samples.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Examples using `sklearn.feature_selection.SelectKBest`

42.11.4 `sklearn.feature_selection`.SelectFpr

class `sklearn.feature_selection.`**SelectFpr** (*score_func=<function f_classif at 0x2ad3d0872c80>, 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.

Figure 42.302: *Concatenating multiple feature extraction methods*

Figure 42.303: *Pipeline Anova SVM*

Figure 42.304: *Classification of text documents using sparse features*

Parametersscore_func : callable

Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues).

alpha : float, optional

The highest p-value for features to be kept.

Attributesscores_ : array-like, shape=(n_features,)

Scores of features.

pvalues_ : array-like, shape=(n_features,)

p-values of feature scores.

See also:

`f_classif`ANOVA F-value between labe/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.

`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.

Methods

`fit`(X, y)	Run score function on (X, y) and get the appropriate features.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`get_support`([indices])	Get a mask, or integer index, of the features selected
`inverse_transform`(X)	Reverse the transformation operation
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Reduce X to the selected features.

static `__init__`(*score_func=<function f_classif at 0x2ad3d0872c80>, alpha=0.05*)

`fit`(*X, y*)

Run score function on (X, y) and get the appropriate features.

ParametersX : array-like, shape = [n_samples, n_features]

The training input samples.

y : array-like, shape = [n_samples]

The target values (class labels in classification, real numbers in regression).

Returnsself : object

Returns self.

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_support(*indices=False*)

Get a mask, or integer index, of the features selected

Parametersindices : boolean (default False)

If True, the return value will be an array of integers, rather than a boolean mask.

Returnssupport : 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(*X*)

Reverse the transformation operation

ParametersX : array of shape [n_samples, n_selected_features]

The input samples.

ReturnsX_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(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*)

Reduce X to the selected features.

ParametersX : 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.

42.11.5 sklearn.feature_selection.SelectFdr

class sklearn.feature_selection.**SelectFdr**(*score_func=<function f_classif at 0x2ad3d0872c80>, 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.

Parameters score_func : callable

Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues).

alpha : float, optional

The highest uncorrected p-value for features to keep.

Attributes scores_ : array-like, shape=(n_features,)

Scores of features.

pvalues_ : array-like, shape=(n_features,)

p-values of feature scores.

See also:

f_classif ANOVA F-value between labe/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.

SelectFwe Select features based on family-wise error rate.

GenericUnivariateSelect Univariate feature selector with configurable mode.

References

http://en.wikipedia.org/wiki/False_discovery_rate

Methods

fit(X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
get_support([indices])	Get a mask, or integer index, of the features selected
inverse_transform(X)	Reverse the transformation operation
set_params(**params)	Set the parameters of this estimator.
transform(X)	Reduce X to the selected features.

static __init__(*score_func=<function f_classif at 0x2ad3d0872c80>*, *alpha=0.05*)

fit(*X*, *y*)
Run score function on (X, y) and get the appropriate features.

ParametersX : array-like, shape = [n_samples, n_features]

The training input samples.

y : array-like, shape = [n_samples]

The target values (class labels in classification, real numbers in regression).

Returnsself : object

Returns self.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_support(*indices=False*)
Get a mask, or integer index, of the features selected

Parametersindices : boolean (default False)

If True, the return value will be an array of integers, rather than a boolean mask.

Returnssupport : 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(*X*)
Reverse the transformation operation

ParametersX : array of shape [n_samples, n_selected_features]

The input samples.

ReturnsX_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(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*)
Reduce X to the selected features.

ParametersX : array of shape [n_samples, n_features]

The input samples.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

42.11.6 sklearn.feature_selection.SelectFwe

class sklearn.feature_selection.**SelectFwe**(*score_func=<function f_classif at 0x2ad3d0872c80>*, *alpha=0.05*)
Filter: Select the p-values corresponding to Family-wise error rate

Parametersscore_func : callable

Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues).

alpha : float, optional

The highest uncorrected p-value for features to keep.

Attributesscores_ : array-like, shape=(n_features,)

Scores of features.

pvalues_ : array-like, shape=(n_features,)

p-values of feature scores.

See also:

f_classifANOVA F-value between labe/feature for classification tasks.

chi2 Chi-squared stats of non-negative features for classification tasks.

f_regressionF-value between label/feature for regression tasks.

SelectPercentileSelect features based on percentile of the highest scores.

SelectKBestSelect features based on the k highest scores.

SelectFprSelect features based on a false positive rate test.

SelectFdrSelect features based on an estimated false discovery rate.

GenericUnivariateSelectUnivariate feature selector with configurable mode.

Methods

fit(X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
get_support([indices])	Get a mask, or integer index, of the features selected
inverse_transform(X)	Reverse the transformation operation
set_params(**params)	Set the parameters of this estimator.
transform(X)	Reduce X to the selected features.

static __init__(*score_func=<function f_classif at 0x2ad3d0872c80>, alpha=0.05*)

fit(*X, y*)

Run score function on (X, y) and get the appropriate features.

ParametersX : array-like, shape = [n_samples, n_features]

The training input samples.

y : array-like, shape = [n_samples]

The target values (class labels in classification, real numbers in regression).

Returnsself : object

Returns self.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_support(*indices=False*)

Get a mask, or integer index, of the features selected

Parametersindices : boolean (default False)

If True, the return value will be an array of integers, rather than a boolean mask.

Returnssupport : 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(*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(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns self :

transform(*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.

42.11.7 sklearn.feature_selection.RFE

class sklearn.feature_selection.**RFE**(*estimator*, *n_features_to_select=None*, *step=1*, *estimator_params=None*, *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 weights are assigned to each one of them. Then, features whose absolute weights are the smallest are pruned from the current set features. That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.

Parameters estimator : object

A supervised learning estimator with a *fit* method that updates a *coef_* attribute that holds the fitted parameters. Important features must correspond to high absolute values in the *coef_* array.

For instance, this is the case for most supervised learning algorithms such as Support Vector Classifiers and Generalized Linear Models from the *svm* and *linear_model* modules.

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.

estimator_params : dict

Parameters for the external estimator. Useful for doing grid searches when an *RFE* object is passed as an argument to, e.g., a *sklearn.grid_search.GridSearchCV* object.

verbose : int, default=0

Controls verbosity of output.

Attributesn_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.

References

[R138]

Examples

The following example shows how to retrieve the 5 right 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], dtype=bool)
>>> selector.ranking_
array([1, 1, 1, 1, 1, 6, 4, 3, 2, 5])
```

Methods

`decision_function`	
`fit(X, y)`	Fit the RFE model and then the underlying estimator on the selected features.

Continued on next page

Table 42.88 – continued from previous page

fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
get_support([indices])	Get a mask, or integer index, of the features selected
inverse_transform(X)	Reverse the transformation operation
predict	Reduce X to the selected features and then predict using the underlying estimator.
predict_log_proba	
predict_proba	
score	Reduce X to the selected features and then return the score of the underlying estimator.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Reduce X to the selected features.

static __init__(*estimator*, *n_features_to_select=None*, *step=1*, *estimator_params=None*, *verbose=0*)

static fit(*X*, *y*)

Fit the RFE model and then the underlying estimator on the selectedfeatures.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

The training input samples.

y : array-like, shape = [n_samples]

The target values.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_support(*indices=False*)

Get a mask, or integer index, of the features selected

Parametersindices : boolean (default False)

If True, the return value will be an array of integers, rather than a boolean mask.

Returnssupport : 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(*X*)
Reverse the transformation operation

ParametersX : array of shape [n_samples, n_selected_features]

The input samples.

ReturnsX_r : array of shape [n_samples, n_original_features]

X with columns of zeros inserted where features would have been removed by *transform*.

predict

Reduce X to the selected features and then predict using theunderlying estimator.

ParametersX : array of shape [n_samples, n_features]

The input samples.

Returnsy : array of shape [n_samples]

The predicted target values.

score

Reduce X to the selected features and then return the score of theunderlying estimator.

ParametersX : array of shape [n_samples, n_features]

The input samples.

y : array of shape [n_samples]

The target values.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*)
Reduce X to the selected features.

ParametersX : array of shape [n_samples, n_features]

The input samples.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

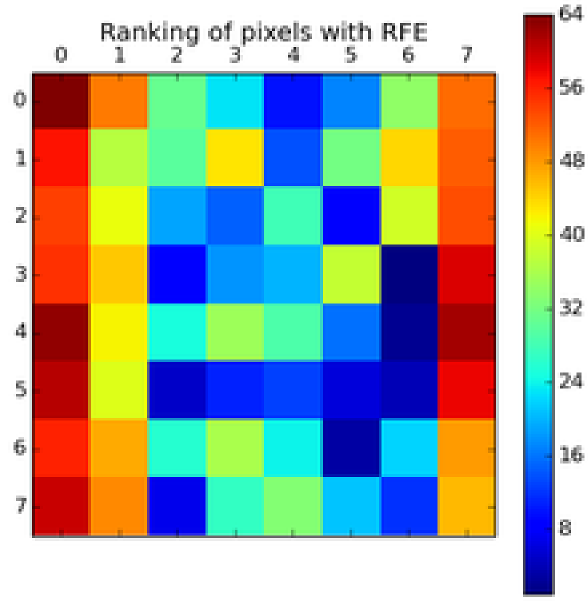


Figure 42.305: *Recursive feature elimination*

Examples using `sklearn.feature_selection.RFE`

42.11.8 `sklearn.feature_selection`.RFECV

class `sklearn.feature_selection.`**`RFECV`**(*estimator*, *step=1*, *cv=None*, *scoring=None*, *estimator_params=None*, *verbose=0*)

Feature ranking with recursive feature elimination and cross-validated selection of the best number of features.

Parametersestimator : object

A supervised learning estimator with a *fit* method that updates a *coef_* attribute that holds the fitted parameters. Important features must correspond to high absolute values in the *coef_* array.

For instance, this is the case for most supervised learning algorithms such as Support Vector Classifiers and Generalized Linear Models from the *svm* and *linear_model* modules.

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.

cv : int or cross-validation generator, optional (default=None)

If int, it is the number of folds. If None, 3-fold cross-validation is performed by default. Specific cross-validation objects can also be passed, see *sklearn.cross_validation module* for details.

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)`.

estimator_params : dict

Parameters for the external estimator. Useful for doing grid searches when an *RFE* object is passed as an argument to, e.g., a *sklearn.grid_search.GridSearchCV* object.

verbose : int, default=0

Controls verbosity of output.

Attributesn_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.

Notes

The size of `grid_scores_` is equal to (n_features + step - 2) // step + 1, where step is the number of features removed at each iteration.

References

[R139]

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], dtype=bool)
```

```
>>> selector.ranking_
array([1, 1, 1, 1, 1, 6, 4, 3, 2, 5])
```

Methods

decision_function(*args, **kwargs)	
fit(X, y)	Fit the RFE model and automatically tune the number of selected features.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
get_support([indices])	Get a mask, or integer index, of the features selected
inverse_transform(X)	Reverse the transformation operation
predict(*args, **kwargs)	Reduce X to the selected features and then predict using the underlying estimator.
predict_log_proba(*args, **kwargs)	
predict_proba(*args, **kwargs)	
score(*args, **kwargs)	Reduce X to the selected features and then return the score of the underlying estimator.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Reduce X to the selected features.

static `__init__` (*estimator*, *step=1*, *cv=None*, *scoring=None*, *estimator_params=None*, *verbose=0*)

static `fit` (*X*, *y*)

Fit the RFE model and automatically tune the number of selectedfeatures.

ParametersX : {array-like, sparse matrix}, 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, shape = [n_samples]

Target values (integers for classification, real numbers for regression).

`fit_transform` (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

`get_params` (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_support (*indices=False*)

Get a mask, or integer index, of the features selected

Parametersindices : boolean (default False)

If True, the return value will be an array of integers, rather than a boolean mask.

Returnssupport : 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 (*X*)

Reverse the transformation operation

ParametersX : array of shape [n_samples, n_selected_features]

The input samples.

ReturnsX_r : array of shape [n_samples, n_original_features]

X with columns of zeros inserted where features would have been removed by *transform*.

static **predict** (**args*, ***kwargs*)

Reduce X to the selected features and then predict using theunderlying estimator.

ParametersX : array of shape [n_samples, n_features]

The input samples.

Returnsy : array of shape [n_samples]

The predicted target values.

static **score** (**args*, ***kwargs*)

Reduce X to the selected features and then return the score of theunderlying estimator.

ParametersX : array of shape [n_samples, n_features]

The input samples.

y : array of shape [n_samples]

The target values.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*)

Reduce X to the selected features.

ParametersX : array of shape [n_samples, n_features]

The input samples.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Examples using `sklearn.feature_selection.RFECV`

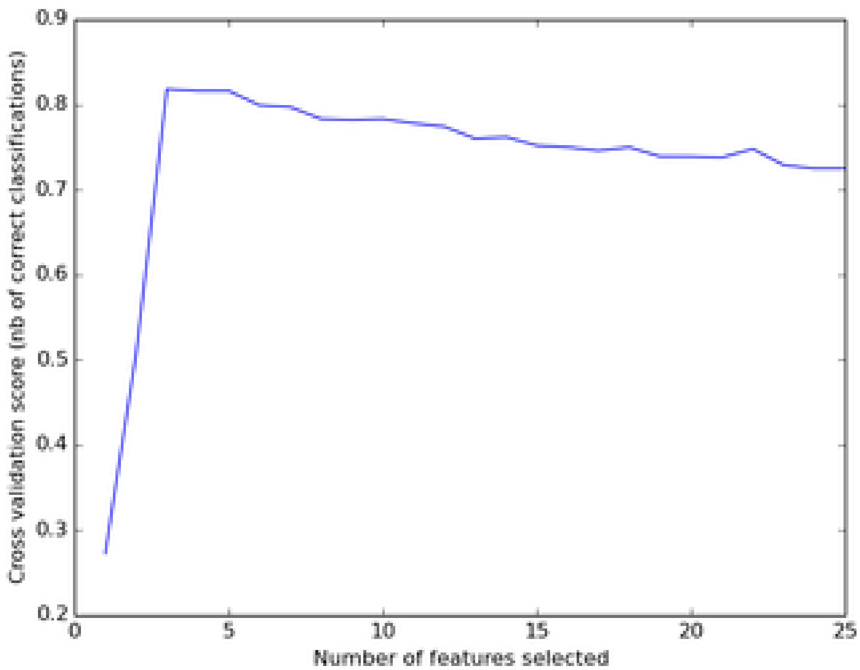


Figure 42.306: *Recursive feature elimination with cross-validation*

42.11.9 `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.

Parametersthreshold : 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.

Attributesvariances_ : array, shape (n_features,)

Variances of individual features.

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`(X[, y])	Learn empirical variances from X.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`get_support`([indices])	Get a mask, or integer index, of the features selected
`inverse_transform`(X)	Reverse the transformation operation
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Reduce X to the selected features.

static `__init__`(*threshold=0.0*)

static `fit`(*X*, *y=None*)

Learn empirical variances from X.

ParametersX : {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.

Returnsself :

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_support (*indices=False*)

Get a mask, or integer index, of the features selected

Parametersindices : boolean (default False)

If True, the return value will be an array of integers, rather than a boolean mask.

Returnssupport : 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 (*X*)

Reverse the transformation operation

ParametersX : array of shape [n_samples, n_selected_features]

The input samples.

ReturnsX_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 (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*)

Reduce X to the selected features.

ParametersX : array of shape [n_samples, n_features]

The input samples.

ReturnsX_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
`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.

42.11.10 `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.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features_in)

Sample vectors.

y : array-like, shape = (n_samples,)

Target vector (class labels).

Returnschi2 : 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 labe/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`

Figure 42.307: *Classification of text documents using sparse features*

42.11.11 `sklearn.feature_selection`.f_classif

`sklearn.feature_selection.f_classif`(*X*, *y*)

Compute the ANOVA F-value for the provided sample.

ParametersX : {array-like, sparse matrix} shape = [n_samples, n_features]

The set of regressors that will tested sequentially.

y : array of shape(n_samples)

The data matrix.

ReturnsF : 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`

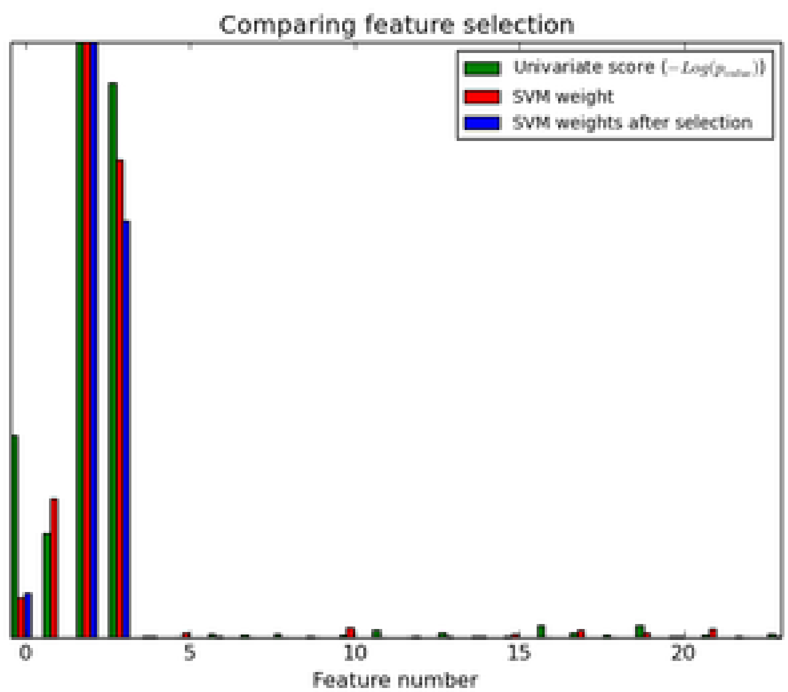


Figure 42.308: *Univariate Feature Selection*

42.11.12 `sklearn.feature_selection`.f_regression

`sklearn.feature_selection.`**`f_regression`**(*X*, *y*, *center=True*)

Univariate linear regression tests.

Quick linear model for testing the effect of a single regressor, sequentially for many regressors.

This is done in 3 steps:

1.The regressor of interest and the data are orthogonalized wrt constant regressors.

2.The cross correlation between data and regressors is computed.

3.It is converted to an F score then to a p-value.

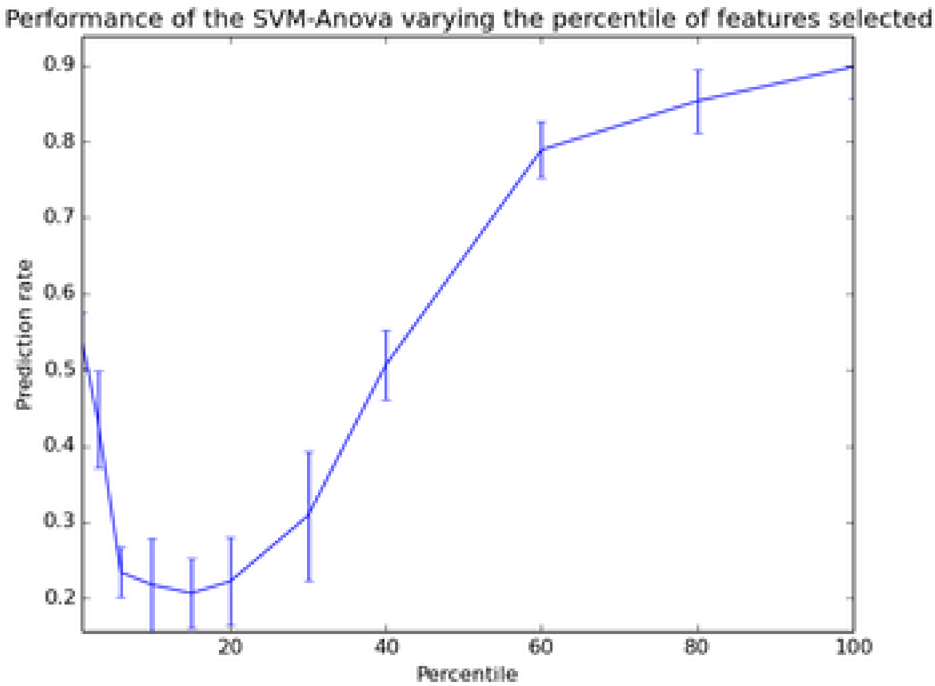


Figure 42.309: *SVM-Anova: SVM with univariate feature selection*

ParametersX : {array-like, sparse matrix} shape = (n_samples, n_features)

The set of regressors that will tested sequentially.

y : array of shape(n_samples).

The data matrix

center : True, bool,

If true, X and y will be centered.

ReturnsF : array, shape=(n_features,)

F values of features.

pval : array, shape=(n_features,)

p-values of F-scores.

See also:

`f_classif`ANOVA F-value between labe/feature for classification tasks.

`chi2`Chi-squared stats of non-negative features for classification tasks.

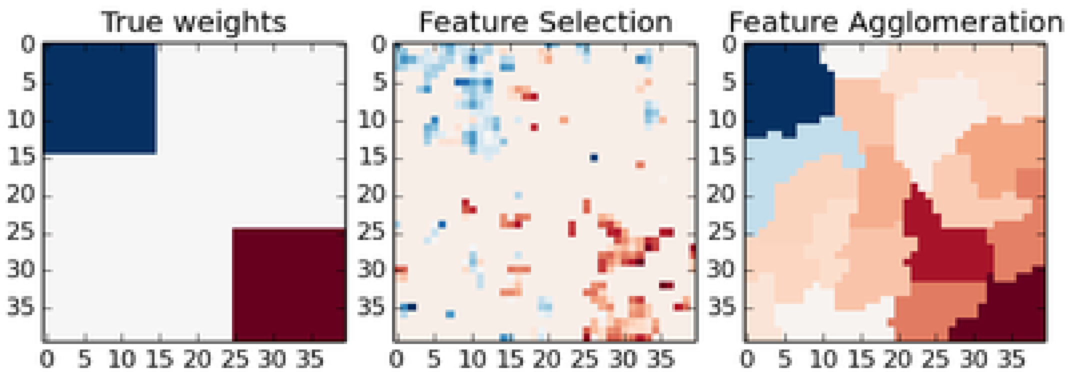


Figure 42.310: *Feature agglomeration vs. univariate selection*

Figure 42.311: *Pipeline Anova SVM*

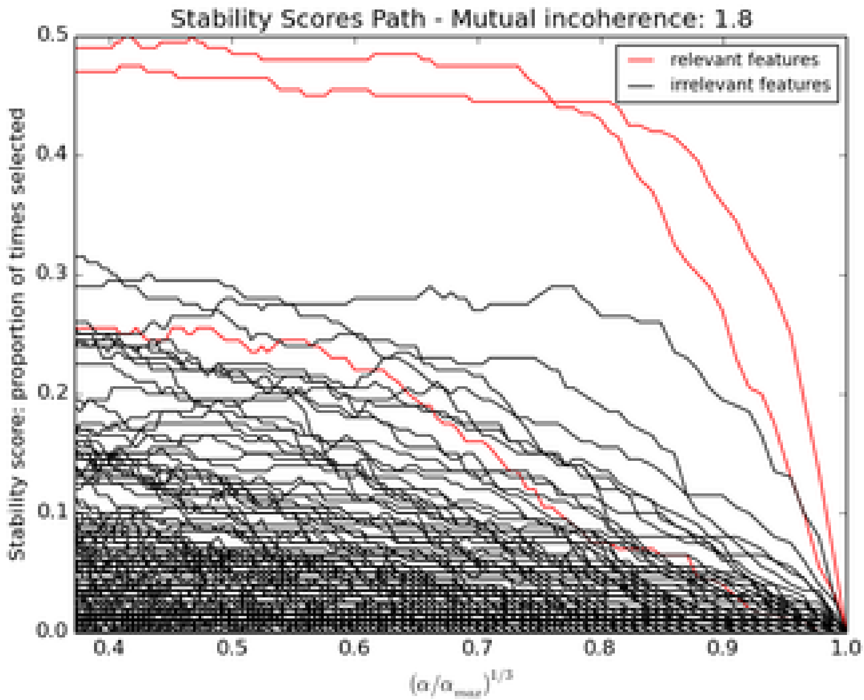


Figure 42.312: *Sparse recovery: feature selection for sparse linear models*

Examples using `sklearn.feature_selection.f_regression`

42.12 `sklearn.gaussian_process`: Gaussian Processes

The `sklearn.gaussian_process` module implements scalar Gaussian Process based predictions.

User guide: See the *Gaussian Processes* section for further details.

`gaussian_process.GaussianProcess`([regr, ...])	The Gaussian Process model class.

42.12.1 `sklearn.gaussian_process.GaussianProcess`

class `sklearn.gaussian_process.`**`GaussianProcess`**(*regr='constant', corr='squared_exponential', beta0=None, storage_mode='full', verbose=False, theta0=0.1, thetaL=None, thetaU=None, optimizer='fmin_cobyla', random_start=1, normalize=True, nugget=2.220446049250313e-15, random_state=None*)

The Gaussian Process model class.

Parametersregr : string or callable, optional

A regression function returning an array of outputs of the linear regression functional basis. The number of observations n_samples should be greater than the size p of this basis. Default assumes a simple constant regression trend. Available built-in regression models are:

`'constant', 'linear', 'quadratic'`

corr : string or callable, optional

A stationary autocorrelation function returning the autocorrelation between two points

A boolean specifying the verbose level. Default is verbose = False.

theta0 : double array_like, optional

An array with shape (n_features,) or (1,). The parameters in the autocorrelation model. If thetaL and thetaU are also specified, theta0 is considered as the starting point for the maximum likelihood estimation of the best set of parameters. Default assumes isotropic autocorrelation model with theta0 = 1e-1.

thetaL : double array_like, optional

An array with shape matching theta0's. Lower bound on the autocorrelation parameters for maximum likelihood estimation. Default is None, so that it skips maximum likelihood estimation and it uses theta0.

thetaU : double array_like, optional

An array with shape matching theta0's. Upper bound on the autocorrelation parameters for maximum likelihood estimation. Default is None, so that it skips maximum likelihood estimation and it uses theta0.

normalize : boolean, optional

Input X and observations y are centered and reduced wrt means and standard deviations estimated from the n_samples observations provided. Default is normalize = True so that data is normalized to ease maximum likelihood estimation.

nugget : double or ndarray, optional

Introduce a nugget effect to allow smooth predictions from noisy data. If nugget is an ndarray, it must be the same length as the number of data points used for the fit. The nugget is added to the diagonal of the assumed training covariance; in this way it acts as a Tikhonov regularization in the problem. In the special case of the squared exponential correlation function, the nugget mathematically represents the variance of the input values. Default assumes a nugget close to machine precision for the sake of robustness (nugget = 10. * MACHINE_EPSILON).

optimizer : string, optional

A string specifying the optimization algorithm to be used. Default uses 'fmin_cobyla' algorithm from scipy.optimize. Available optimizers are:

```
'fmin_cobyla', 'Welch'
```

'Welch' optimizer is dued to Welch et al., see reference [WBSWM1992]. It consists in iterating over several one-dimensional optimizations instead of running one single multi-dimensional optimization.

random_start : int, optional

The number of times the Maximum Likelihood Estimation should be performed from a random starting point. The first MLE always uses the specified starting point (theta0), the next starting points are picked at random according to an exponential distribution (log-uniform on [thetaL, thetaU]). Default does not use random starting point (random_start = 1).

random_state: integer or numpy.RandomState, optional :

The generator used to shuffle the sequence of coordinates of theta in the Welch optimizer. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

Attributestheta_ : array

Specified theta OR the best set of autocorrelation parameters (the sought maximizer of the reduced likelihood function).

reduced_likelihood_function_value_ : array

The optimal reduced likelihood function value.

Notes

The presentation implementation is based on a translation of the DACE Matlab toolbox, see reference [NLNS2002].

References

[NLNS2002], [WBSWM1992]

Examples

```
>>> import numpy as np
>>> from sklearn.gaussian_process import GaussianProcess
>>> X = np.array([[1., 3., 5., 6., 7., 8.]]).T
>>> y = (X * np.sin(X)).ravel()
>>> gp = GaussianProcess(theta0=0.1, thetaL=.001, thetaU=1.)
>>> gp.fit(X, y)
GaussianProcess(beta0=None...
        ...
```

Methods

fit(X, y)	The Gaussian Process model fitting method.
get_params([deep])	Get parameters for this estimator.
predict(X[, eval_MSE, batch_size])	This function evaluates the Gaussian Process model at x.
reduced_likelihood_function([theta])	This function determines the BLUP parameters and evaluates the reduced likeliho
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__ (*regr='constant', corr='squared_exponential', beta0=None, storage_mode='full', verbose=False, theta0=0.1, thetaL=None, thetaU=None, optimizer='fmin_cobyla', random_start=1, normalize=True, nugget=2.2204460492503131e-15, random_state=None*)

static fit (*X, y*)

The Gaussian Process model fitting method.

ParametersX : double array_like

An array with shape (n_samples, n_features) with the input at which observations were made.

y : double array_like

An array with shape (n_samples,) or shape (n_samples, n_targets) with the observations of the output to be predicted.

Returnsgp : self

A fitted Gaussian Process model object awaiting data to perform predictions.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **predict**(*X*, *eval_MSE=False*, *batch_size=None*)
This function evaluates the Gaussian Process model at x.

ParametersX : array_like

An array with shape (n_eval, n_features) giving the point(s) at which the prediction(s) should be made.

eval_MSE : boolean, optional

A boolean specifying whether the Mean Squared Error should be evaluated or not. Default assumes evalMSE = False and evaluates only the BLUP (mean prediction).

batch_size : integer, optional

An integer giving the maximum number of points that can be evaluated simultaneously (depending on the available memory). Default is None so that all given points are evaluated at the same time.

Returnsy : array_like, shape (n_samples,) or (n_samples, n_targets)

An array with shape (n_eval,) if the Gaussian Process was trained on an array of shape (n_samples,) or an array with shape (n_eval, n_targets) if the Gaussian Process was trained on an array of shape (n_samples, n_targets) with the Best Linear Unbiased Prediction at x.

MSE : array_like, optional (if eval_MSE == True)

An array with shape (n_eval,) or (n_eval, n_targets) as with y, with the Mean Squared Error at x.

static **reduced_likelihood_function**(*theta=None*)
This function determines the BLUP parameters and evaluates the reduced likelihood function for the given autocorrelation parameters theta.

Maximizing this function wrt the autocorrelation parameters theta is equivalent to maximizing the likelihood of the assumed joint Gaussian distribution of the observations y evaluated onto the design of experiments X.

Parameterstheta : array_like, optional

An array containing the autocorrelation parameters at which the Gaussian Process model parameters should be determined. Default uses the built-in autocorrelation parameters (ie `theta = self.theta_`).

Returnsreduced_likelihood_function_value : double

The value of the reduced likelihood function associated to the given autocorrelation parameters theta.

par : dict

A dictionary containing the requested Gaussian Process model parameters:

sigma2 Gaussian Process variance.

betaGeneralized least-squares regression weights for Universal Kriging or given beta0 for Ordinary Kriging.

gammaGaussian Process weights.

CCholesky decomposition of the correlation matrix [R].

FtSolution of the linear equation system : [R] x Ft = F

GQR decomposition of the matrix Ft.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.gaussian_process.GaussianProcess`

`gaussian_process.correlation_models.absolute_exponential`(...)	Absolute exponential autocorrelation m
`gaussian_process.correlation_models.squared_exponential`(...)	Squared exponential correlation model
`gaussian_process.correlation_models.generalized_exponential`(...)	Generalized exponential correlation mo
`gaussian_process.correlation_models.pure_nugget`(...)	Spatial independence correlation mode
`gaussian_process.correlation_models.cubic`(...)	Cubic correlation model:
`gaussian_process.correlation_models.linear`(...)	Linear correlation model:
`gaussian_process.regression_models.constant`(x)	Zero order polynomial (constant, p = 1
`gaussian_process.regression_models.linear`(x)	First order polynomial (linear, p = n+1

Table 42.94 – continued from previous page

gaussian_process.regression_models.quadratic(x)	Second order polynomial (quadratic, p

42.12.2 sklearn.gaussian_process.correlation_models.absolute_exponential

sklearn.gaussian_process.correlation_models.**absolute_exponential**(*theta*, *d*)

Absolute exponential autocorrelation model. (Ornstein-Uhlenbeck stochastic process):

```
                                          n
theta, d --> r(theta, d) = exp(  sum  - theta_i * |d_i| )
                                        i = 1
```

Parameterstheta : array_like

An array with shape 1 (isotropic) or n (anisotropic) giving the autocorrelation parameter(s).

d : array_like

An array with shape (n_eval, n_features) giving the componentwise distances between locations x and x' at which the correlation model should be evaluated.

Returnsr : array_like

An array with shape (n_eval,) containing the values of the autocorrelation model.

42.12.3 sklearn.gaussian_process.correlation_models.squared_exponential

sklearn.gaussian_process.correlation_models.**squared_exponential**(*theta*, *d*)

Squared exponential correlation model (Radial Basis Function). (Infinitely differentiable stochastic process, very smooth):

```
                                          n
theta, d --> r(theta, d) = exp(  sum  - theta_i * (d_i)^2 )
                                        i = 1
```

Parameterstheta : array_like

An array with shape 1 (isotropic) or n (anisotropic) giving the autocorrelation parameter(s).

d : array_like

An array with shape (n_eval, n_features) giving the componentwise distances between locations x and x' at which the correlation model should be evaluated.

Returnsr : array_like

An array with shape (n_eval,) containing the values of the autocorrelation model.

42.12.4 sklearn.gaussian_process.correlation_models.generalized_exponential

sklearn.gaussian_process.correlation_models.**generalized_exponential**(*theta*, *d*)

Generalized exponential correlation model. (Useful when one does not know the smoothness of the function to be predicted.):

Figure 42.313: *Gaussian Processes regression: goodness-of-fit on the 'diabetes' dataset*

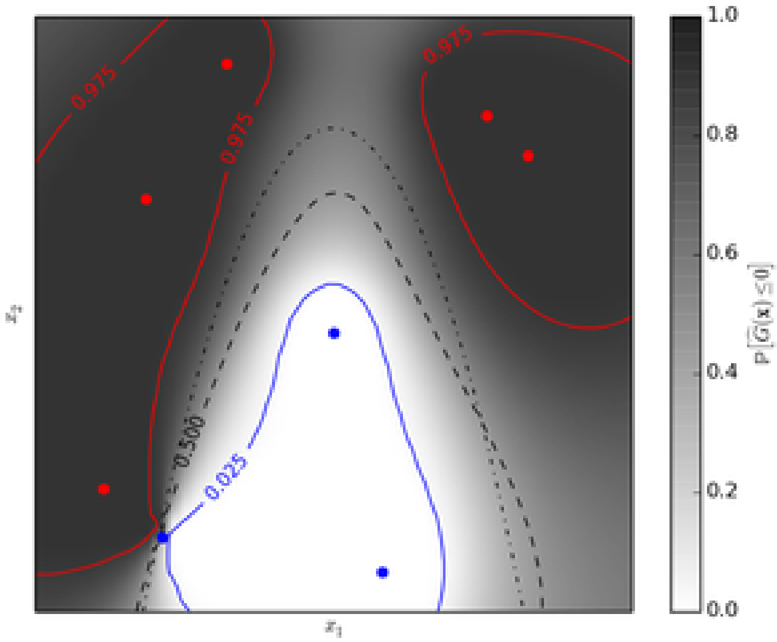


Figure 42.314: *Gaussian Processes classification example: exploiting the probabilistic output*

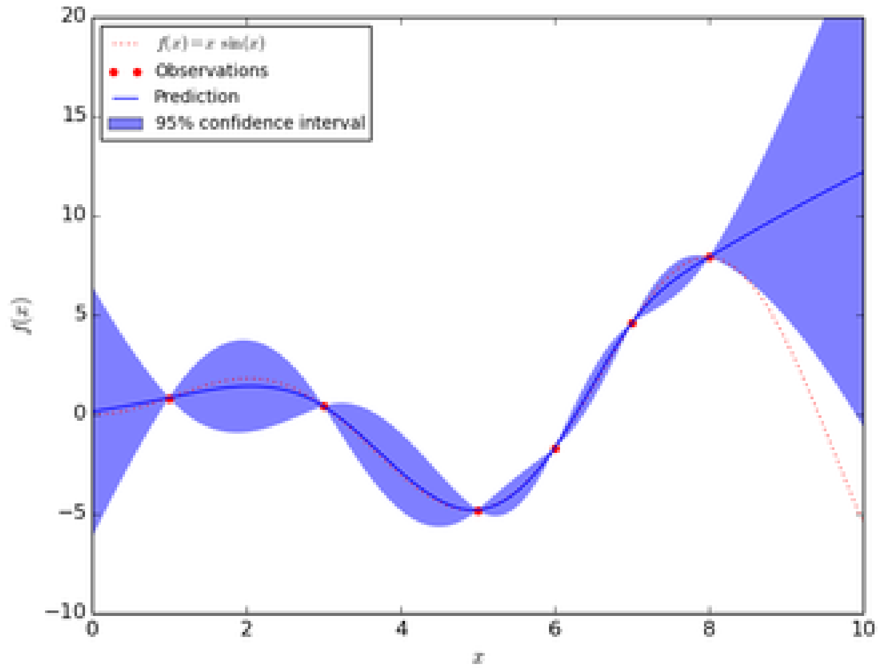


Figure 42.315: *Gaussian Processes regression: basic introductory example*

```
                                    n
theta, d --> r(theta, d) = exp(  sum  - theta_i * |d_i|^p )
                                  i = 1
```

Parameterstheta : array_like

An array with shape 1+1 (isotropic) or n+1 (anisotropic) giving the autocorrelation parameter(s) (theta, p).

d : array_like

An array with shape (n_eval, n_features) giving the componentwise distances between locations x and x' at which the correlation model should be evaluated.

Returnsr : array_like

An array with shape (n_eval,) with the values of the autocorrelation model.

42.12.5 sklearn.gaussian_process.correlation_models.pure_nugget

sklearn.gaussian_process.correlation_models.**pure_nugget**(*theta*, *d*)

Spatial independence correlation model (pure nugget). (Useful when one wants to solve an ordinary least squares problem!):

```
                                     n
theta, d --> r(theta, d) = 1 if   sum |d_i| == 0
                                   i = 1
                              0 otherwise
```

Parameterstheta : array_like

None.

d : array_like

An array with shape (n_eval, n_features) giving the componentwise distances between locations x and x' at which the correlation model should be evaluated.

Returnsr : array_like

An array with shape (n_eval,) with the values of the autocorrelation model.

42.12.6 `sklearn.gaussian_process.correlation_models`.cubic

`sklearn.gaussian_process.correlation_models.`**`cubic`**(*theta*, *d*)

Cubic correlation model:

```
theta, d --> r(theta, d) =
  n
 prod max(0, 1 - 3(theta_j*d_ij)^2 + 2(theta_j*d_ij)^3) ,  i = 1,...,m
j = 1
```

Parameterstheta : array_like

An array with shape 1 (isotropic) or n (anisotropic) giving the autocorrelation parameter(s).

d : array_like

An array with shape (n_eval, n_features) giving the componentwise distances between locations x and x' at which the correlation model should be evaluated.

Returnsr : array_like

An array with shape (n_eval,) with the values of the autocorrelation model.

42.12.7 `sklearn.gaussian_process.correlation_models`.linear

`sklearn.gaussian_process.correlation_models.`**`linear`**(*theta*, *d*)

Linear correlation model:

```
theta, d --> r(theta, d) =
      n
    prod max(0, 1 - theta_j*d_ij) ,  i = 1,...,m
    j = 1
```

Parameterstheta : array_like

An array with shape 1 (isotropic) or n (anisotropic) giving the autocorrelation parameter(s).

d : array_like

An array with shape (n_eval, n_features) giving the componentwise distances between locations x and x' at which the correlation model should be evaluated.

Returnsr : array_like

An array with shape (n_eval,) with the values of the autocorrelation model.

42.12.8 sklearn.gaussian_process.regression_models.constant

sklearn.gaussian_process.regression_models.**constant**(*x*)
Zero order polynomial (constant, p = 1) regression model.

x –> f(x) = 1

Parametersx : array_like

An array with shape (n_eval, n_features) giving the locations x at which the regression model should be evaluated.

Returnsf : array_like

An array with shape (n_eval, p) with the values of the regression model.

42.12.9 sklearn.gaussian_process.regression_models.linear

sklearn.gaussian_process.regression_models.**linear**(*x*)
First order polynomial (linear, p = n+1) regression model.

x –> f(x) = [1, x_1, ..., x_n].T

Parametersx : array_like

An array with shape (n_eval, n_features) giving the locations x at which the regression model should be evaluated.

Returnsf : array_like

An array with shape (n_eval, p) with the values of the regression model.

42.12.10 sklearn.gaussian_process.regression_models.quadratic

sklearn.gaussian_process.regression_models.**quadratic**(*x*)
Second order polynomial (quadratic, p = n*(n-1)/2+n+1) regression model.

x –> f(x) = [1, { x_i, i = 1,...,n }, { x_i * x_j, (i,j) = 1,...,n }].Ti > j

Parametersx : array_like

An array with shape (n_eval, n_features) giving the locations x at which the regression model should be evaluated.

Returnsf : array_like

An array with shape (n_eval, p) with the values of the regression model.

42.13 sklearn.grid_search: Grid Search

The sklearn.grid_search includes utilities to fine-tune the parameters of an estimator.

User guide: See the *Grid Search: Searching for estimator parameters* section for further details.

grid_search.GridSearchCV(estimator, param_grid)	Exhaustive search over specified parameter values for an estimator.
grid_search.ParameterGrid(param_grid)	Grid of parameters with a discrete number of values for each.
grid_search.ParameterSampler(...[, random_state])	Generator on parameters sampled from given distributions.

Continued on next page

Table 42.95 – continued from previous page

grid_search.RandomizedSearchCV(estimator, ...)	Randomized search on hyper parameters.

42.13.1 sklearn.grid_search.GridSearchCV

class sklearn.grid_search.**GridSearchCV**(*estimator*, *param_grid*, *scoring=None*, *loss_func=None*, *score_func=None*, *fit_params=None*, *n_jobs=1*, *iid=True*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *error_score='raise'*)

Exhaustive search over specified parameter values for an estimator.

Important members are fit, predict.

GridSearchCV implements a "fit" method and a "predict" method like any classifier except that the parameters of the classifier used to predict is optimized by cross-validation.

Parametersestimator : object type that implements the "fit" and "predict" methods

A object of that type is instantiated for each grid point.

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 or None, optional, default: None

A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`.

fit_params : dict, optional

Parameters to pass to the fit method.

n_jobs : int, default 1

Number of jobs to run in parallel.

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=True

If True, 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.

cv : integer or cross-validation generator, default=3

If an integer is passed, it is the number of folds. Specific cross-validation objects can be passed, see sklearn.cross_validation module for the list of possible objects

refit : boolean, default=True

Refit the best estimator with the entire dataset. If “False”, it is impossible to make predictions using this GridSearchCV instance after fitting.

verbose : integer

Controls the verbosity: the higher, the more messages.

error_score : ‘raise’ (default) 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.

Attributesgrid_scores_ : list of named tuples

Contains scores for all parameter combinations in param_grid. Each entry corresponds to one parameter setting. Each named tuple has the attributes:

- `parameters`, a dict of parameter settings
- `mean_validation_score`, the mean score over the cross-validation folds
- `cv_validation_scores`, the list of scores for each fold

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.

best_score_ : float

Score of best_estimator on the left out data.

best_params_ : dict

Parameter setting that gave the best results on the hold out data.

scorer_ : function

Scorer function used on the held out data to choose the best parameters for the model.

See also:

`ParameterGrid`generates all the combinations of a an hyperparameter grid.

`sklearn.cross_validation.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, grid_search, datasets
>>> iris = datasets.load_iris()
>>> parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
>>> svr = svm.SVC()
>>> clf = grid_search.GridSearchCV(svr, parameters)
>>> clf.fit(iris.data, iris.target)
...
GridSearchCV(cv=None, error_score=...,
       estimator=SVC(C=1.0, cache_size=..., class_weight=..., coef0=...,
                     degree=..., gamma=..., kernel='rbf', max_iter=-1,
                     probability=False, random_state=None, shrinking=True,
                     tol=..., verbose=False),
       fit_params={}, iid=..., n_jobs=1,
       param_grid=..., pre_dispatch=..., refit=...,
       scoring=..., verbose=...)
```

Methods

`decision_function`(*args, **kwargs)	Call decision_function on the estimator with the best found parameters.
`fit`(X[, y])	Run fit with all sets of parameters.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(*args, **kwargs)	Call inverse_transform on the estimator with the best found parameters.
`predict`(*args, **kwargs)	Call predict on the estimator with the best found parameters.
`predict_log_proba`(*args, **kwargs)	Call predict_log_proba on the estimator with the best found parameters.
`predict_proba`(*args, **kwargs)	Call predict_proba on the estimator with the best found parameters.
`score`(X[, y])	Returns the score on the given data, if the estimator has been refit
`set_params`(**params)	Set the parameters of this estimator.
`transform`(*args, **kwargs)	Call transform on the estimator with the best found parameters.

static `__init__`(*estimator*, *param_grid*, *scoring=None*, *loss_func=None*, *score_func=None*, *fit_params=None*, *n_jobs=1*, *iid=True*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *error_score='raise'*)

static `decision_function`(**args*, ***kwargs*)

Call decision_function on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `decision_function`.

ParametersX : indexable, length n_samples

Must fulfill the input assumptions of the underlying estimator.

static `fit`(*X*, *y=None*)

Run fit with all sets of parameters.

ParametersX : 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_output], optional

Target relative to X for classification or regression; None for unsupervised learning.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **inverse_transform**(**args*, ***kwargs*)
Call inverse_transform on the estimator with the best found parameters.

Only available if the underlying estimator implements `inverse_transform` and `refit=True`.

ParametersXt : indexable, length n_samples

Must fulfill the input assumptions of the underlying estimator.

static **predict**(**args*, ***kwargs*)
Call predict on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict`.

ParametersX : indexable, length n_samples

Must fulfill the input assumptions of the underlying estimator.

static **predict_log_proba**(**args*, ***kwargs*)
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`.

ParametersX : indexable, length n_samples

Must fulfill the input assumptions of the underlying estimator.

static **predict_proba**(**args*, ***kwargs*)
Call predict_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_proba`.

ParametersX : indexable, length n_samples

Must fulfill the input assumptions of the underlying estimator.

score(*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.

ParametersX : 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 : array-like, shape = [n_samples] or [n_samples, n_output], optional

Target relative to X for classification or regression; None for unsupervised learning.

Returnsscore : float

Notes

- The long-standing behavior of this method changed in version 0.16.
- It no longer uses the metric provided by `estimator.score` if the `scoring` parameter was set when fitting.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static **transform**(**args*, ***kwargs*)

Call transform on the estimator with the best found parameters.

Only available if the underlying estimator supports `transform` and `refit=True`.

ParametersX : indexable, length n_samples

Must fulfill the input assumptions of the underlying estimator.

Examples using `sklearn.grid_search.GridSearchCV`

Figure 42.316: *Concatenating multiple feature extraction methods*

42.13.2 `sklearn.grid_search`.ParameterGrid

class `sklearn.grid_search.`**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.

Parametersparam_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.

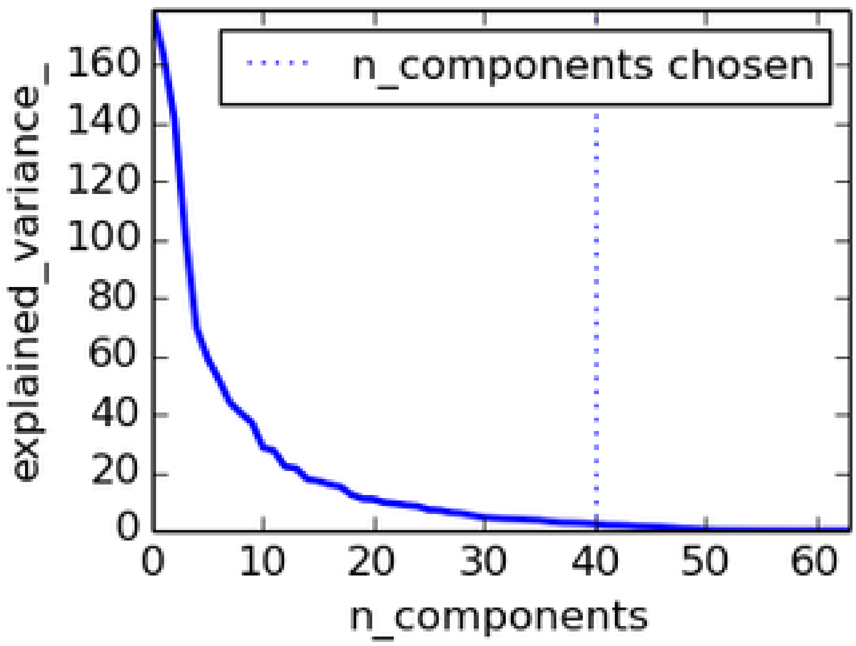


Figure 42.317: *Pipelining: chaining a PCA and a logistic regression*

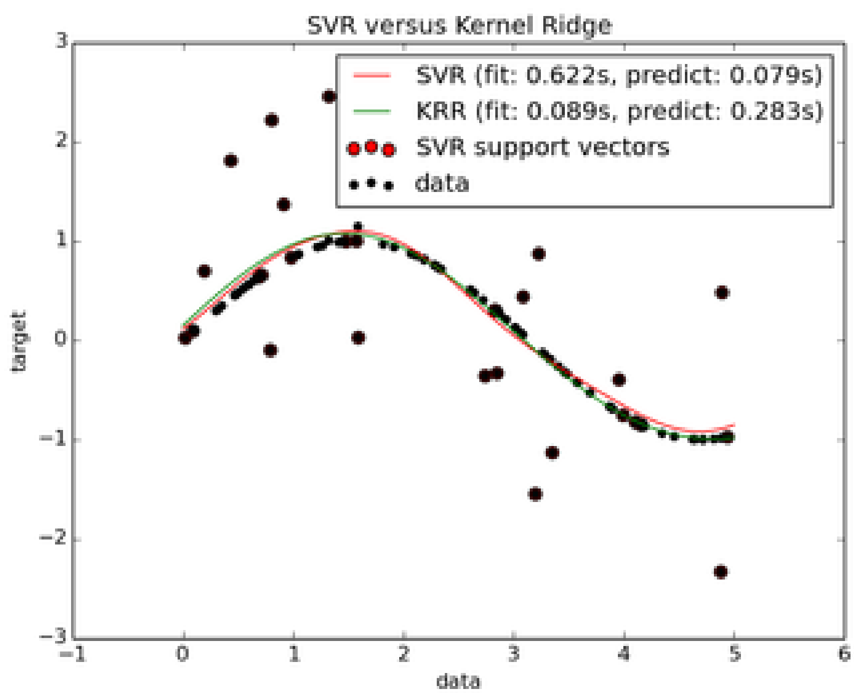


Figure 42.318: *Comparison of kernel ridge regression and SVR*

Figure 42.319: *Faces recognition example using eigenfaces and SVMs*

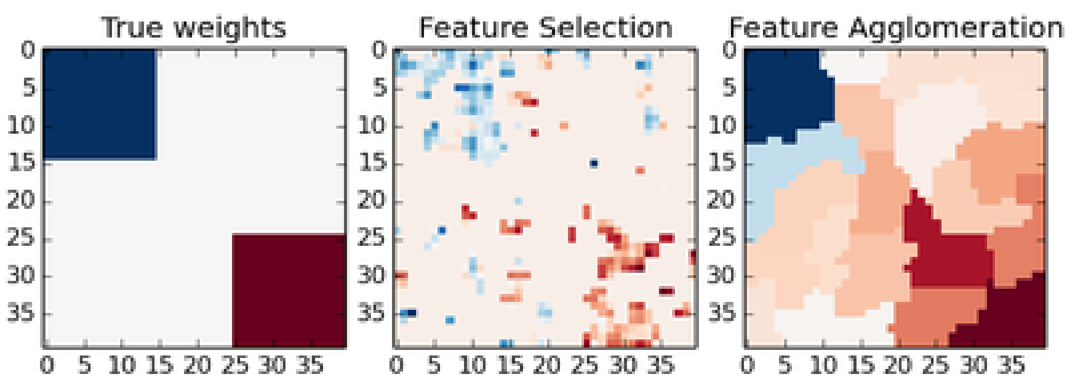


Figure 42.320: *Feature agglomeration vs. univariate selection*

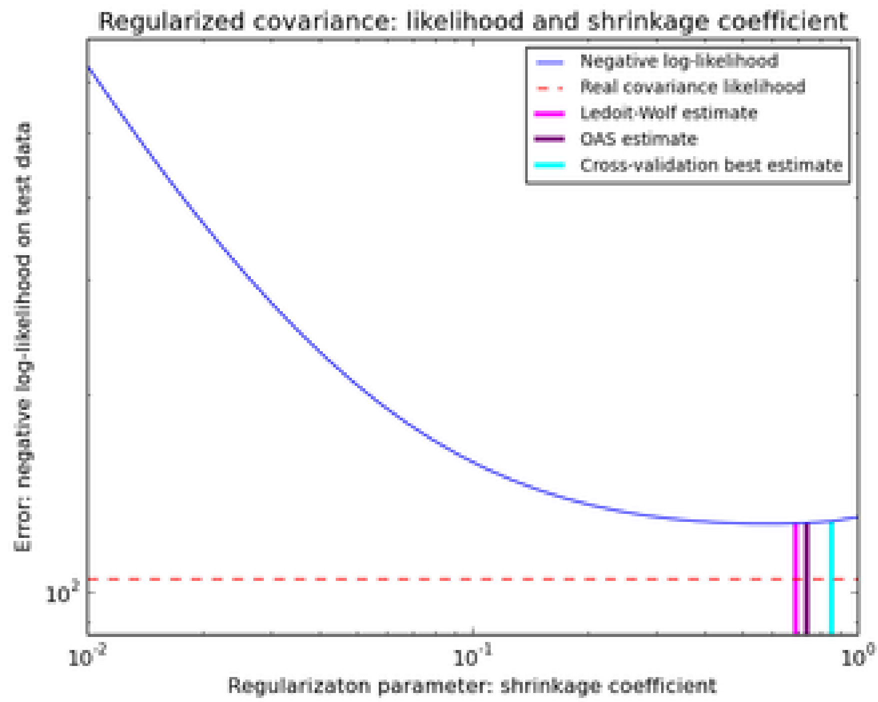


Figure 42.321: *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*

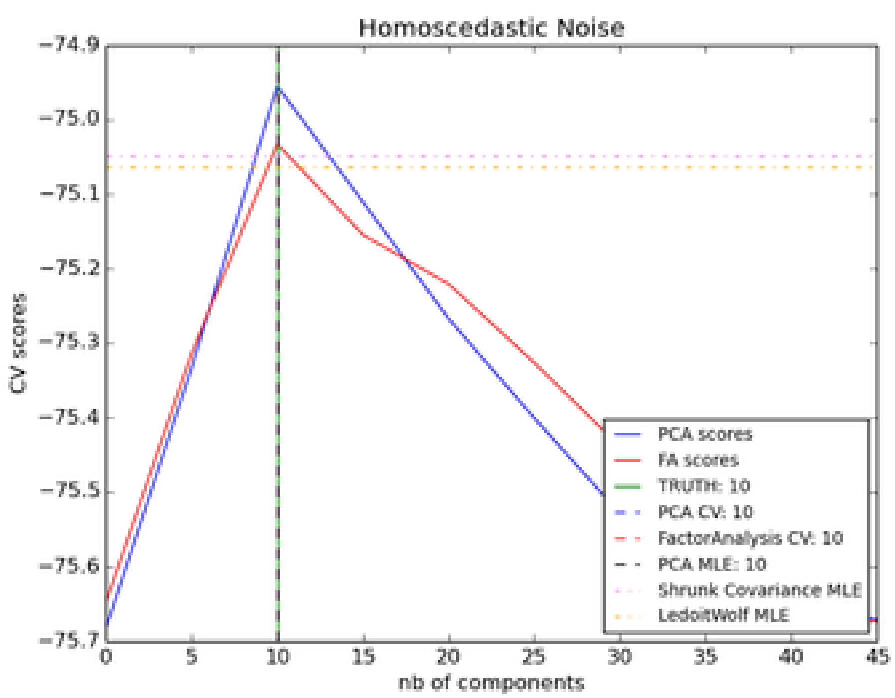


Figure 42.322: *Model selection with Probabilistic (PCA) and Factor Analysis (FA)*

Figure 42.323: *Parameter estimation using grid search with cross-validation*

Figure 42.324: *Comparing randomized search and grid search for hyperparameter estimation*

Figure 42.325: *Sample pipeline for text feature extraction and evaluation*

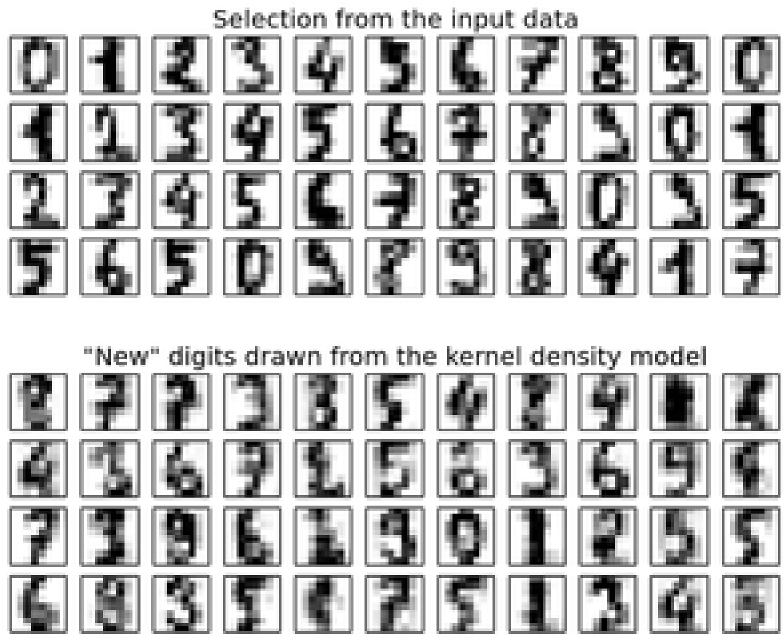


Figure 42.326: *Kernel Density Estimation*

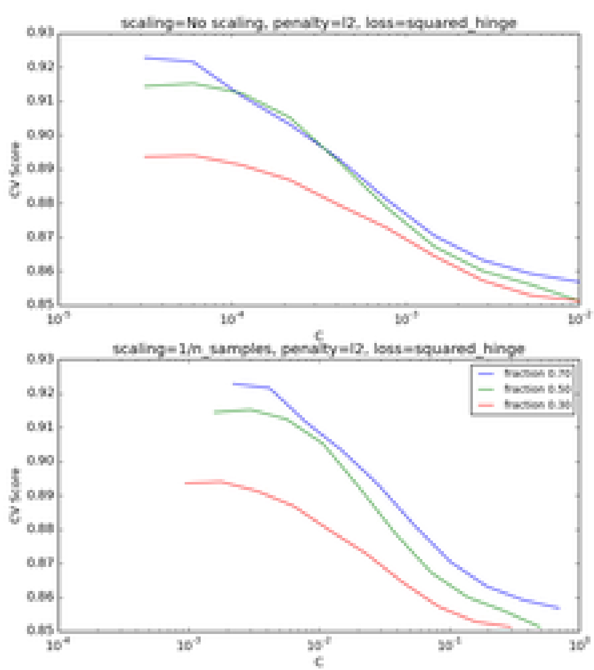


Figure 42.327: *Scaling the regularization parameter for SVCs*

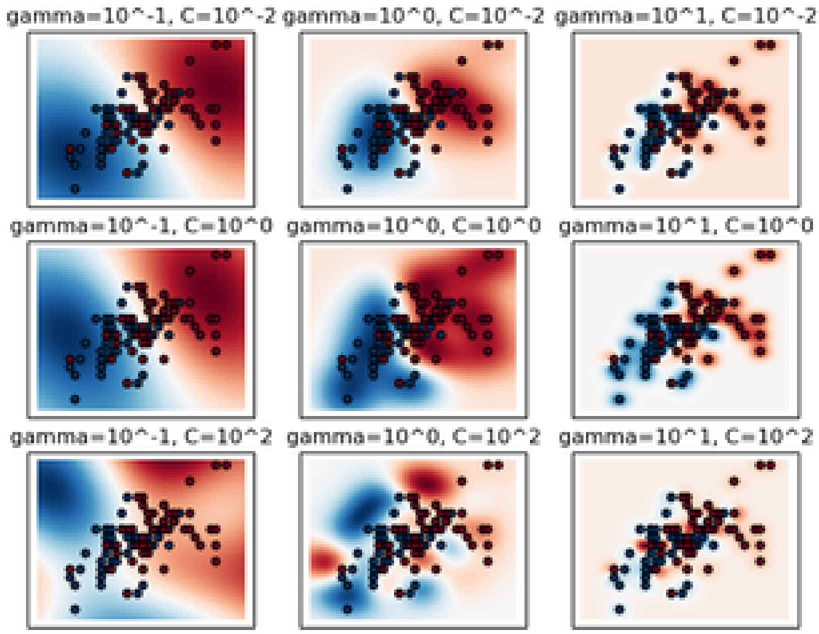


Figure 42.328: *RBF SVM parameters*

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:

GridSearchCVuses `ParameterGrid` to perform a full parallelized parameter search.

Examples

```
>>> from sklearn.grid_search 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
.. automethod:: __init__
```

42.13.3 `sklearn.grid_search.ParameterSampler`

class `sklearn.grid_search.`**`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.

Note that as of SciPy 0.12, the `scipy.stats.distributions` do not accept a custom RNG instance and always use the singleton RNG from `numpy.random`. Hence setting `random_state` will not guarantee a deterministic iteration whenever `scipy.stats` distributions are used to define the parameter search space.

Parametersparam_distributions : dict

Dictionary where the keys are parameters and values are distributions from which a parameter is to be sampled. Distributions either have to provide a `rvs` function to sample from them, or can be given as a list of values, where a uniform distribution is assumed.

n_iter : integer

Number of parameter settings that are produced.

random_state : int or RandomState

Pseudo random number generator state used for random uniform sampling from lists of possible values instead of scipy.stats distributions.

Returnsparams : dict of string to any

Yields dictionaries mapping each estimator parameter to as sampled value.

Examples

```
>>> from sklearn.grid_search import ParameterSampler
>>> from scipy.stats.distributions import expon
>>> import numpy as np
>>> np.random.seed(0)
>>> param_grid = {'a':[1, 2], 'b': expon()}
>>> param_list = list(ParameterSampler(param_grid, n_iter=4))
>>> 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
.. automethod:: __init__
```

42.13.4 `sklearn.grid_search`.RandomizedSearchCV

class `sklearn.grid_search.`**`RandomizedSearchCV`**(*estimator*, *param_distributions*, *n_iter=10*, *scoring=None*, *fit_params=None*, *n_jobs=1*, *iid=True*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *random_state=None*, *error_score='raise'*)

Randomized search on hyper parameters.

RandomizedSearchCV implements a "fit" method and a "predict" method like any classifier except that the parameters of the classifier used to predict is optimized by cross-validation.

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.

Parametersestimator : object type that implements the "fit" and "predict" methods

A object of that type is instantiated for each parameter setting.

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.

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 or None, optional, default: None

A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`.

fit_params : dict, optional

Parameters to pass to the fit method.

n_jobs : int, default=1

Number of jobs to run in parallel.

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=True

If True, 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.

cv : integer or cross-validation generator, optional

If an integer is passed, it is the number of folds (default 3). Specific cross-validation objects can be passed, see sklearn.cross_validation module for the list of possible objects

refit : boolean, default=True

Refit the best estimator with the entire dataset. If "False", it is impossible to make predictions using this RandomizedSearchCV instance after fitting.

verbose : integer

Controls the verbosity: the higher, the more messages.

error_score : 'raise' (default) 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.

Attributesgrid_scores_ : list of named tuples

Contains scores for all parameter combinations in param_grid. Each entry corresponds to one parameter setting. Each named tuple has the attributes:

- `parameters`, a dict of parameter settings
- `mean_validation_score`, the mean score over the cross-validation folds
- `cv_validation_scores`, the list of scores for each fold

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.

best_score_ : float

Score of best_estimator on the left out data.

best_params_ : dict

Parameter setting that gave the best results on the hold out data.

See also:

`GridSearchCV`Does exhaustive search over a grid of parameters.

`ParameterSampler`A generator over parameter settins, 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*.

Methods

decision_function(*args, **kwargs)	Call decision_function on the estimator with the best found parameters.
fit(X[, y])	Run fit on the estimator with randomly drawn parameters.
get_params([deep])	Get parameters for this estimator.
inverse_transform(*args, **kwargs)	Call inverse_transform on the estimator with the best found parameters.
predict(*args, **kwargs)	Call predict on the estimator with the best found parameters.
predict_log_proba(*args, **kwargs)	Call predict_log_proba on the estimator with the best found parameters.
predict_proba(*args, **kwargs)	Call predict_proba on the estimator with the best found parameters.
score(X[, y])	Returns the score on the given data, if the estimator has been refit
set_params(**params)	Set the parameters of this estimator.
transform(*args, **kwargs)	Call transform on the estimator with the best found parameters.

static `__init__`(*estimator*, *param_distributions*, *n_iter=10*, *scoring=None*, *fit_params=None*, *n_jobs=1*, *iid=True*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *random_state=None*, *error_score='raise'*)

static `decision_function`(**args*, ***kwargs*)

Call decision_function on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `decision_function`.

ParametersX : indexable, length n_samples

Must fulfill the input assumptions of the underlying estimator.

static `fit`(*X*, *y=None*)

Run fit on the estimator with randomly drawn parameters.

ParametersX : 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] or [n_samples, n_output], optional

Target relative to X for classification or regression; None for unsupervised learning.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static `inverse_transform`(**args*, ***kwargs*)

Call inverse_transform on the estimator with the best found parameters.

Only available if the underlying estimator implements `inverse_transform` and `refit=True`.

ParametersXt : indexable, length n_samples

Must fulfill the input assumptions of the underlying estimator.

static `predict`(**args*, ***kwargs*)

Call predict on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict`.

ParametersX : indexable, length n_samples

Must fulfill the input assumptions of the underlying estimator.

static **predict_log_proba**(**args*, ***kwargs*)

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`.

ParametersX : indexable, length n_samples

Must fulfill the input assumptions of the underlying estimator.

static **predict_proba**(**args*, ***kwargs*)

Call predict_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_proba`.

ParametersX : indexable, length n_samples

Must fulfill the input assumptions of the underlying estimator.

score(*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.

ParametersX : 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 : array-like, shape = [n_samples] or [n_samples, n_output], optional

Target relative to X for classification or regression; None for unsupervised learning.

Returnsscore : float

Notes

- The long-standing behavior of this method changed in version 0.16.
- It no longer uses the metric provided by `estimator.score` if the `scoring` parameter was set when fitting.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static **transform**(**args*, ***kwargs*)

Call transform on the estimator with the best found parameters.

Only available if the underlying estimator supports `transform` and `refit=True`.

ParametersX : indexable, length n_samples

Must fulfill the input assumptions of the underlying estimator.

Examples using `sklearn.grid_search.RandomizedSearchCV`

Figure 42.329: *Comparing randomized search and grid search for hyperparameter estimation*

42.14 `sklearn.isotonic`: Isotonic regression

User guide: See the *Isotonic regression* section for further details.

`isotonic.IsotonicRegression`([y_min, y_max, ...])	Isotonic regression model.

42.14.1 `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)

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.

AttributesX_ : ndarray (n_samples,)

A copy of the input X.

y_ : ndarray (n_samples,)

Isotonic fit of y.

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 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

Methods

`fit`(X, y[, sample_weight])	Fit the model using X, y as training data.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(T)	Predict new data by linear interpolation.
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(T)	Transform new data by linear interpolation

static `__init__`(*y_min=None*, *y_max=None*, *increasing=True*, *out_of_bounds='nan'*)

static **fit** (*X*, *y*, *sample_weight=None*)
Fit the model using X, y as training data.

ParametersX : array-like, shape=(n_samples,)

Training data.

y : array-like, shape=(n_samples,)

Training target.

sample_weight : array-like, shape=(n_samples,), optional, default: None

Weights. If set to None, all weights will be set to 1 (equal weights).

Returnsself : object

Returns an instance of self.

Notes

X is stored for future use, as *transform* needs X to interpolate new input data.

fit_transform (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **predict** (*T*)
Predict new data by linear interpolation.

ParametersT : array-like, shape=(n_samples,)

Data to transform.

ReturnsT_ : array, shape=(n_samples,)

Transformed data.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **transform**(*T*)

Transform new data by linear interpolation

ParametersT : array-like, shape=(n_samples,)

Data to transform.

ReturnsT_ : array, shape=(n_samples,)

The transformed data

Examples using sklearn.isotonic.IsotonicRegression

isotonic.isotonic_regression(y[, ...])	Solve the isotonic regression model:
isotonic.check_increasing(x, y)	Determine whether y is monotonically correlated with x.

42.14.2 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)

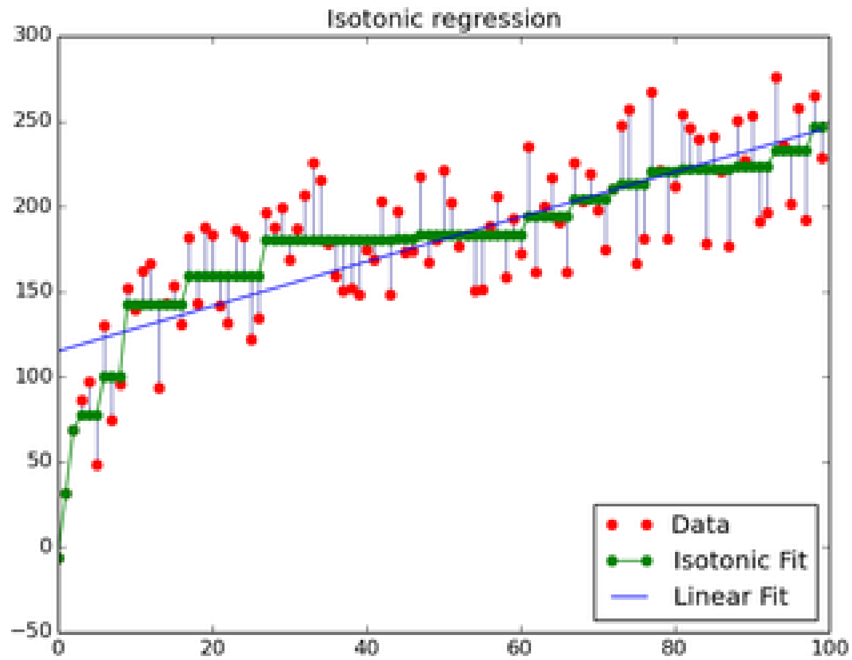


Figure 42.330: *Isotonic Regression*

Parametersy : iterable of floating-point values

The data.

sample_weight : iterable of floating-point values, 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)

Returnsy_ : list of floating-point values

Isotonic fit of y.

References

"Active set algorithms for isotonic regression; A unifying framework" by Michael J. Best and Nilotpal Chakravarti, section 3.

42.14.3 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.

Parametersx : array-like, shape=(n_samples,)

Training data.

y : array-like, 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. http://en.wikipedia.org/w/index.php?title=Fisher_transformation

42.15 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 appr
kernel_approximation.SkewedChi2Sampler([...])	Approximates feature map of the "skewed chi-squared" kernel by

42.15.1 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.

Parameterssample_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}.

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" Vedaldi, A. and Zisserman, A., Computer Vision and Pattern Recognition 2010

Methods

`fit`(X[, y])	Set parameters.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y])	Apply approximate feature map to X.

static `__init__`(*sample_steps=2*, *sample_interval=None*)

static `fit`(*X*, *y=None*)

Set parameters.

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **transform**(*X*, *y=None*)
Apply approximate feature map to X.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

ReturnsX_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.

42.15.2 sklearn.kernel_approximation.Nystroem

class sklearn.kernel_approximation.**Nystroem**(*kernel='rbf'*, *gamma=None*, *coef0=1*, *degree=3*, *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.

Parameterskernel : 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.

n_components : int

Number of features to construct. How many data points will be used to construct the mapping.

gamma : float, default=None

Gamma parameter for the RBF, 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.

random_state : {int, RandomState}, optional

If int, random_state is the seed used by the random number generator; if RandomState instance, random_state is the random number generator.

Attributescomponents_ : 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

Methods

`fit`(X[, y])	Fit estimator to data.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Apply feature map to X.

static `__init__` (*kernel='rbf', gamma=None, coef0=1, degree=3, kernel_params=None, n_components=100, random_state=None*)

static `fit` (*X, y=None*)

Fit estimator to data.

Samples a subset of training points, computes kernel on these and computes normalization matrix.

ParametersX : array-like, shape=(n_samples, n_feature)

Training data.

fit_transform (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static **transform** (*X*)
Apply feature map to X.

Computes an approximate feature map using the kernel between some training points and X.

ParametersX : array-like, shape=(n_samples, n_features)

Data to transform.

ReturnsX_transformed : array, shape=(n_samples, n_components)

Transformed data.

Examples using sklearn.kernel_approximation.Nystroem

42.15.3 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]

Parametersgamma : float

Parameter of RBF kernel: exp(-gamma * x^2)

n_components : int

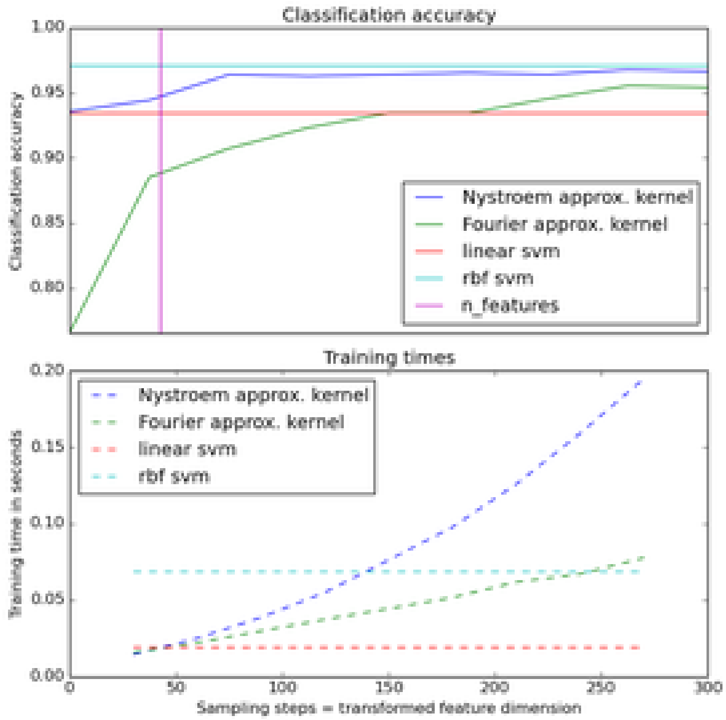


Figure 42.331: *Explicit feature map approximation for RBF kernels*

Number of Monte Carlo samples per original feature. Equals the dimensionality of the computed feature space.

random_state : {int, RandomState}, optional

If int, random_state is the seed used by the random number generator; if RandomState instance, random_state is the random number generator.

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. (http://www.eecs.berkeley.edu/~brecht/papers/08.rah.rec.nips.pdf)

Methods

`fit`(X[, y])	Fit the model with X.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y])	Apply the approximate feature map to X.

static `__init__`(*gamma=1.0*, *n_components=100*, *random_state=None*)

static `fit`(*X*, *y=None*)

Fit the model with X.

Samples random projection according to n_features.

ParametersX : {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.

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static **transform** (*X*, *y=None*)

Apply the approximate feature map to X.

ParametersX : {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.

ReturnsX_new : array-like, shape (n_samples, n_components)

Examples using `sklearn.kernel_approximation.RBFSampler`

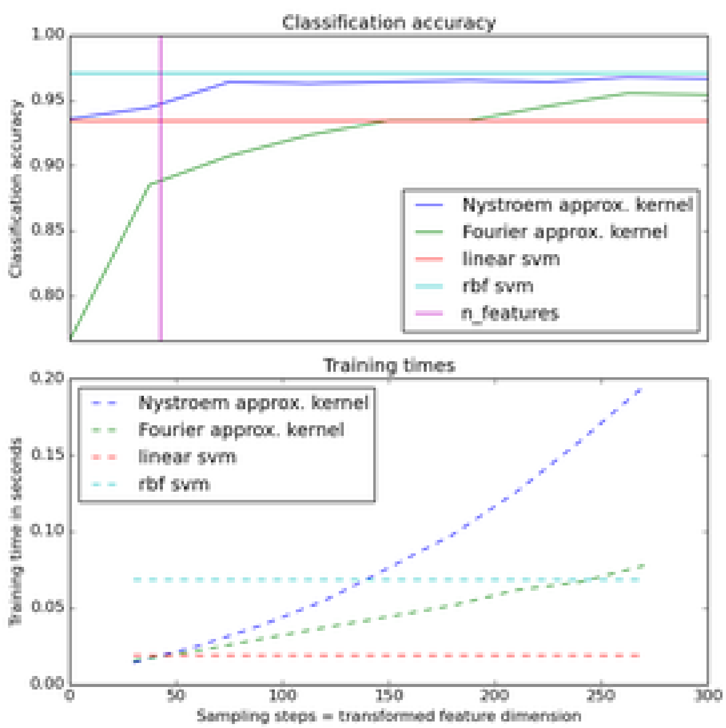


Figure 42.332: *Explicit feature map approximation for RBF kernels*

42.15.4 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.

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}, optional

If int, random_state is the seed used by the random number generator; if RandomState instance, random_state is the random number generator.

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.

Methods

fit(X[, y])	Fit the model with X.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Apply the approximate feature map to X.

static __init__(*skewedness=1.0*, *n_components=100*, *random_state=None*)

static fit(*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 (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static transform (*X*, *y=None*)
Apply the approximate feature map to X.

ParametersX : 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.

ReturnsX_new : array-like, shape (n_samples, n_components)

42.16 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, ...])	Kernel ridge regression.

42.16.1 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]).

Parametersalpha : {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.

gamma : float, default=None

> Gamma parameter for the RBF, 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.

Attributesdual_coef_ : array, shape = [n_features] or [n_targets, n_features]

> Weight vector(s) in kernel space

X_fit_ : {array-like, sparse matrix}, shape = [n_samples, n_features]

> Training data, which is also required for prediction

See also:

`Ridge`Linear ridge regression.

`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, coef0=1, degree=3, gamma=None, kernel='linear',
            kernel_params=None)
```

Methods

`fit`(X[, y, sample_weight])	Fit Kernel Ridge regression model
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict using the the kernel ridge model
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

static __init__(*alpha=1*, *kernel='linear'*, *gamma=None*, *degree=3*, *coef0=1*, *kernel_params=None*)

static fit(*X*, *y=None*, *sample_weight=None*)

Fit Kernel Ridge regression model

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Training data

y : array-like, shape = [n_samples] or [n_samples, n_targets]

Target values

sample_weight : float or numpy array of shape [n_samples]

Individual weights for each sample, ignored if None is passed.

Returnsself : returns an instance of self.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static predict(*X*)

Predict using the the kernel ridge model

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Samples.

ReturnsC : array, shape = [n_samples] or [n_samples, n_targets]

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.kernel_ridge.KernelRidge`

42.17 `sklearn.lda`: Linear Discriminant Analysis

Linear Discriminant Analysis (LDA)

User guide: See the *Linear and quadratic discriminant analysis* section for further details.

`lda.LDA`([solver, shrinkage, priors, ...])	Linear Discriminant Analysis (LDA).

42.17.1 `sklearn.lda`.LDA

class `sklearn.lda.`**LDA** (*solver='svd'*, *shrinkage=None*, *priors=None*, *n_components=None*, *store_covariance=False*, *tol=0.0001*)

Linear Discriminant Analysis (LDA).

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.

Parameterssolver : string, optional

Solver to use, possible values:

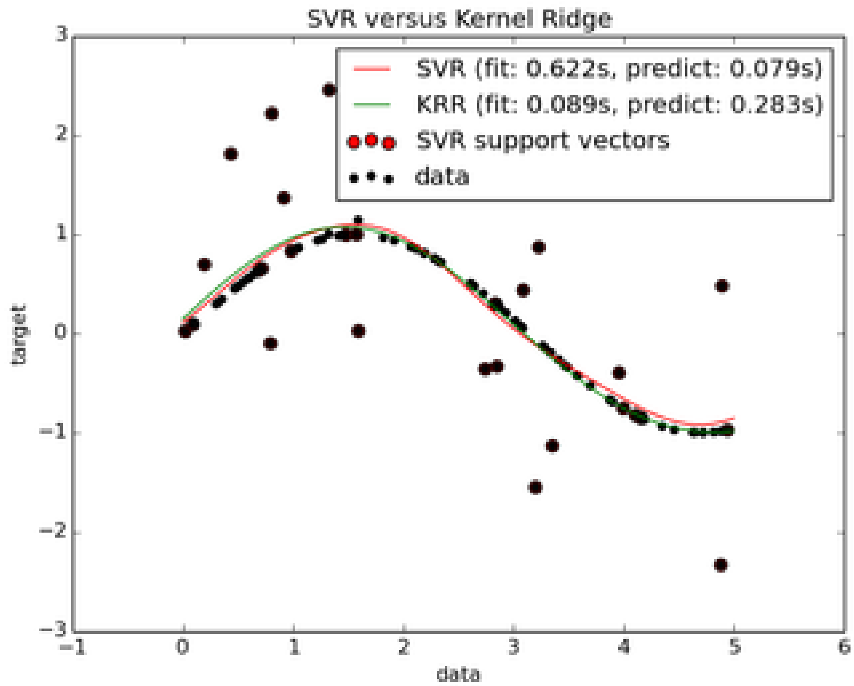


Figure 42.333: *Comparison of kernel ridge regression and SVR*

- **'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

Number of components (< n_classes - 1) for dimensionality reduction.

store_covariance : bool, optional

Additionally compute class covariance matrix (default False).

tol : float, optional

Threshold used for rank estimation in SVD solver.

Attributescoef_ : array, shape (n_features,) or (n_classes, n_features)

Weight vector(s).

intercept_ : array, shape (n_features,)

Intercept term.

covariance_ : array-like, shape (n_features, n_features)

Covariance matrix (shared by all classes).

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.qda.QDA`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.lda import LDA
>>> 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 = LDA()
>>> clf.fit(X, y)
LDA(n_components=None, priors=None, shrinkage=None, solver='svd',
  store_covariance=False, tol=0.0001)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

decision_function(X)	Predict confidence scores for samples.
fit(X, y[, store_covariance, tol])	Fit LDA model according to the given training data and parameters.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict class labels for samples in X.
predict_log_proba(X)	Estimate log probability.
predict_proba(X)	Estimate probability.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Project data to maximize class separation.

static __init__(*solver='svd'*, *shrinkage=None*, *priors=None*, *n_components=None*, *store_covariance=False*, *tol=0.0001*)

decision_function(*X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returnsarray, 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.

static fit(*X*, *y*, *store_covariance=False*, *tol=0.0001*)

Fit LDA model according to the given training data and parameters.

ParametersX : array-like, shape (n_samples, n_features)

Training data.

y : array, shape (n_samples,)

Target values.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict class labels for samples in X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Samples.

ReturnsC : array, shape = [n_samples]

Predicted class label per sample.

static **predict_log_proba** (*X*)

Estimate log probability.

ParametersX : array-like, shape (n_samples, n_features)

Input data.

ReturnsC : array, shape (n_samples, n_classes)

Estimated log probabilities.

static **predict_proba** (*X*)

Estimate probability.

ParametersX : array-like, shape (n_samples, n_features)

Input data.

ReturnsC : array, shape (n_samples, n_classes)

Estimated probabilities.

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static transform (*X*)

Project data to maximize class separation.

ParametersX : array-like, shape (n_samples, n_features)

Input data.

ReturnsX_new : array, shape (n_samples, n_components)

Transformed data.

Examples using `sklearn.lda.LDA`

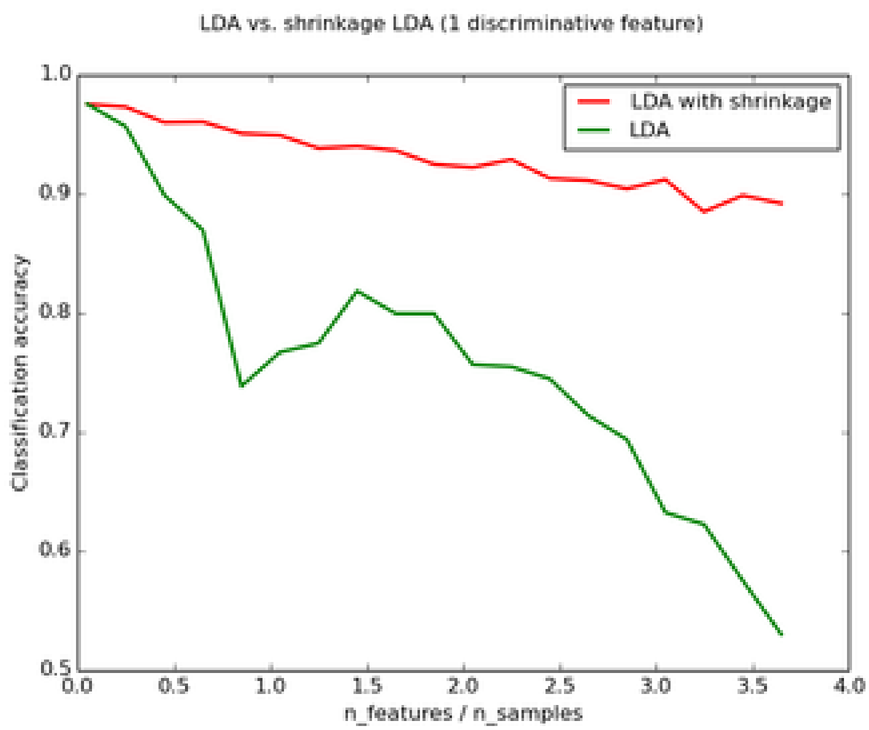


Figure 42.334: *Normal and Shrinkage Linear Discriminant Analysis for classification*

42.18 `sklearn.learning_curve` Learning curve evaluation

Utilities to evaluate models with respect to a variable

`learning_curve.learning_curve`(estimator, X, y)	Learning curve.
`learning_curve.validation_curve`(estimator, ...)	Validation curve.

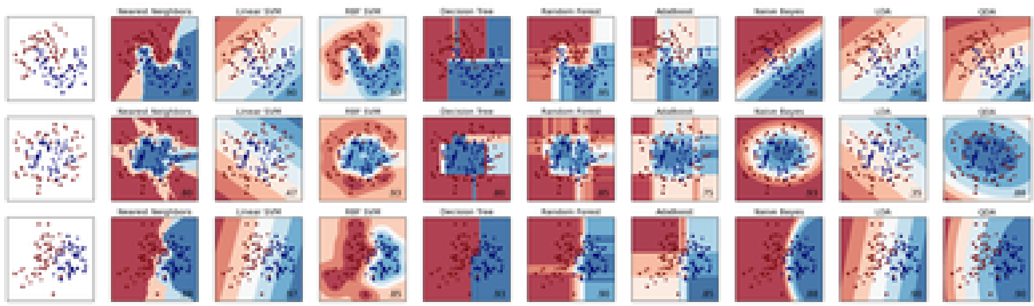


Figure 42.335: *Classifier comparison*

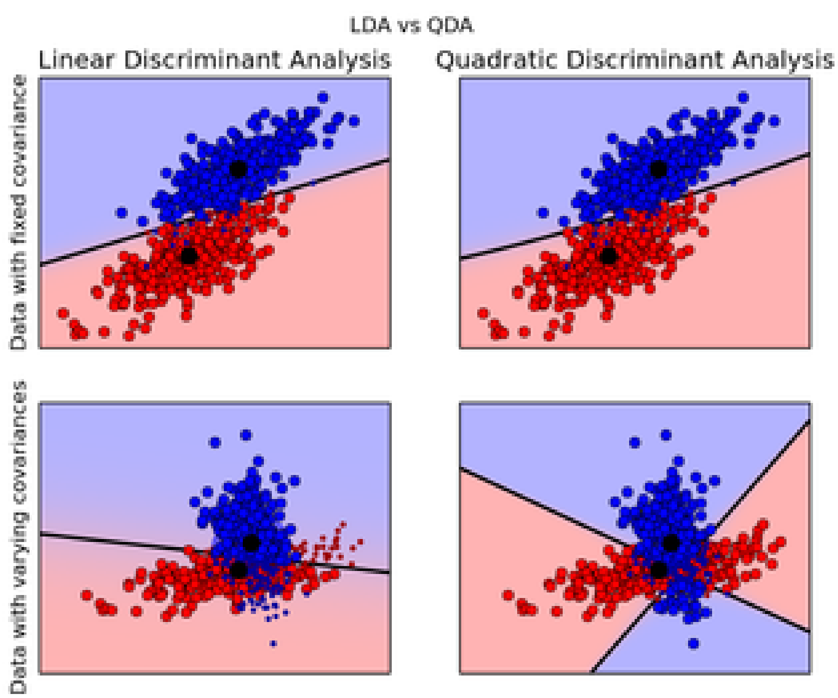


Figure 42.336: *Linear and Quadratic Discriminant Analysis with confidence ellipsoid*

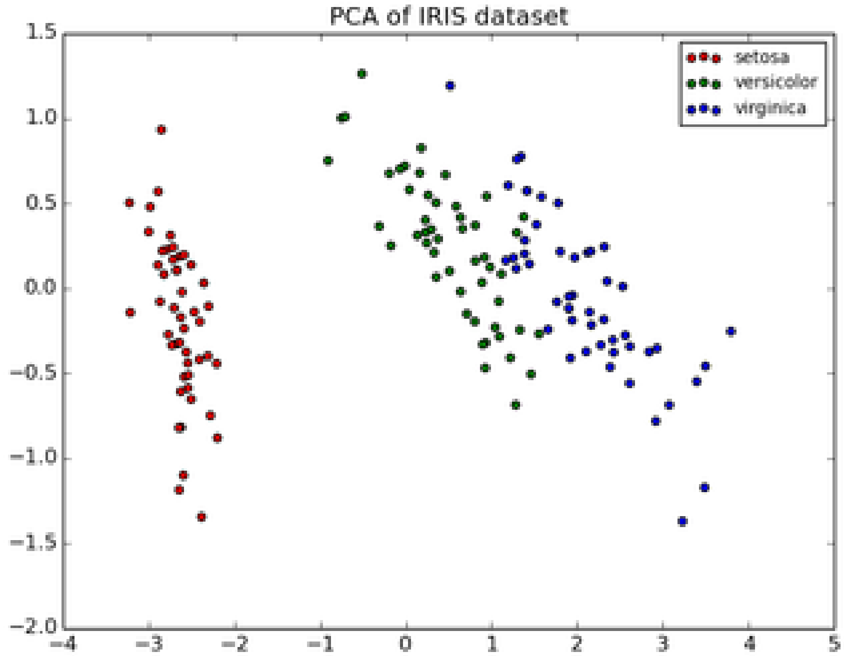


Figure 42.337: *Comparison of LDA and PCA 2D projection of Iris dataset*

Figure 42.338: *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

42.18.1 sklearn.learning_curve.learning_curve

sklearn.learning_curve.**learning_curve**(*estimator*, *X*, *y*, *train_sizes=array([0.1, 0.325, 0.55, 0.775, 1.])*, *cv=None*, *scoring=None*, *exploit_incremental_learning=False*, *n_jobs=1*, *pre_dispatch='all'*, *verbose=0*)

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.

Parametersestimator : 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.

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 : integer, cross-validation generator, optional

If an integer is passed, it is the number of folds (defaults to 3). Specific cross-validation objects can be passed, see sklearn.cross_validation module for the list of possible objects

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 : integer, optional

Number of jobs to run in parallel (default 1).

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.

Returnstrain_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.

Notes

See *examples/model_selection/plot_learning_curve.py*

42.18.2 sklearn.learning_curve.validation_curve

`sklearn.learning_curve.validation_curve`(*estimator*, *X*, *y*, *param_name*, *param_range*, *cv=None*, *scoring=None*, *n_jobs=1*, *pre_dispatch='all'*, *verbose=0*)

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.

Parametersestimator : 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.

cv : integer, cross-validation generator, optional

If an integer is passed, it is the number of folds (defaults to 3). Specific cross-validation objects can be passed, see sklearn.cross_validation module for the list of possible objects

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 : integer, optional

Number of jobs to run in parallel (default 1).

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.

Returnstrain_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 *examples/model_selection/plot_validation_curve.py*

Examples using `sklearn.learning_curve.validation_curve`

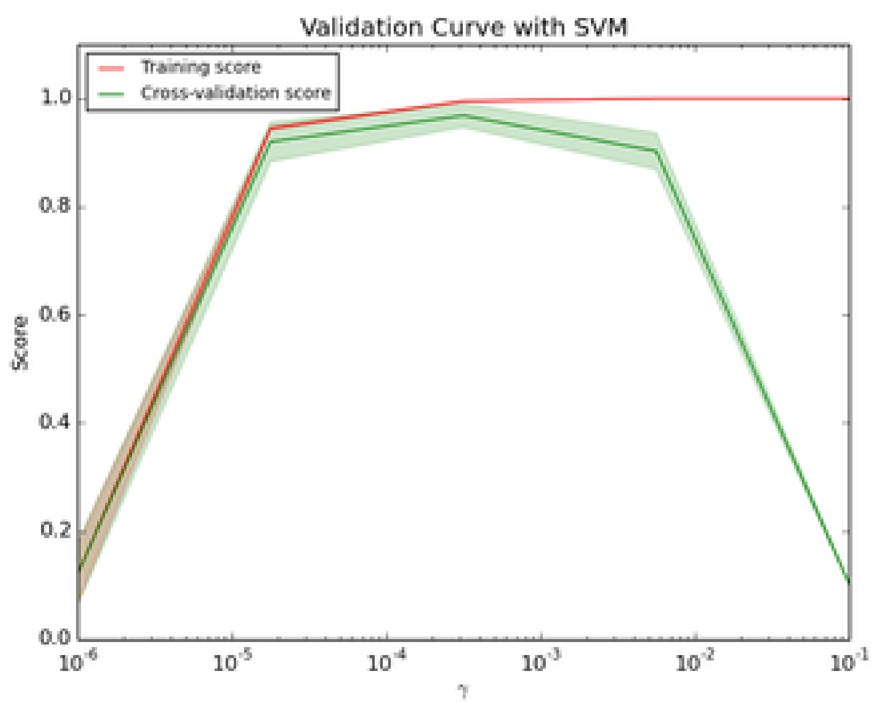


Figure 42.339: *Plotting Validation Curves*

42.19 `sklearn.linear_model`: Generalized Linear Models

The `sklearn.linear_model` module implements generalized linear models. It includes Ridge regression, Bayesian Regression, Lasso and Elastic Net estimators computed with Least Angle Regression and coordinate descent. It also implements Stochastic Gradient Descent related algorithms.

User guide: See the *Generalized Linear Models* section for further details.

`linear_model.ARDRegression`([n_iter, tol, ...])	Bayesian ARD regression.
`linear_model.BayesianRidge`([n_iter, tol, ...])	Bayesian ridge regression
`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 pat
`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 pa
`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.LinearRegression`([...])	Ordinary least squares Linear Regression.
`linear_model.LogisticRegression`([penalty, ...])	Logistic Regression (aka logit, MaxEnt) classifier.
`linear_model.LogisticRegressionCV`([Cs, ...])	Logistic Regression CV (aka logit, MaxEnt) classifier.
`linear_model.MultiTaskLasso`([alpha, ...])	Multi-task Lasso model trained with L1/L2 mixed-norm as regul
`linear_model.MultiTaskElasticNet`([alpha, ...])	Multi-task ElasticNet model trained with L1/L2 mixed-norm as
`linear_model.MultiTaskLassoCV`([eps, ...])	Multi-task L1/L2 Lasso with built-in cross-validation.
`linear_model.MultiTaskElasticNetCV`([...])	Multi-task L1/L2 ElasticNet with built-in cross-validation.
`linear_model.OrthogonalMatchingPursuit`([...])	Orthogonal Matching Pursuit model (OMP)
`linear_model.OrthogonalMatchingPursuitCV`([...])	Cross-validated Orthogonal Matching Pursuit model (OMP)
`linear_model.PassiveAggressiveClassifier`([...])	Passive Aggressive Classifier
`linear_model.PassiveAggressiveRegressor`([C, ...])	Passive Aggressive Regressor
`linear_model.Perceptron`([penalty, alpha, ...])	Perceptron
`linear_model.RandomizedLasso`([alpha, ...])	Randomized Lasso.
`linear_model.RandomizedLogisticRegression`([...])	Randomized Logistic Regression
`linear_model.RANSACRegressor`([...])	RANSAC (RANdom SAmple Consensus) algorithm.
`linear_model.Ridge`([alpha, fit_intercept, ...])	Linear least squares with l2 regularization.
`linear_model.RidgeClassifier`([alpha, ...])	Classifier using Ridge regression.
`linear_model.RidgeClassifierCV`([alphas, ...])	Ridge classifier with built-in cross-validation.
`linear_model.RidgeCV`([alphas, ...])	Ridge regression with built-in cross-validation.
`linear_model.SGDClassifier`([loss, penalty, ...])	Linear classifiers (SVM, logistic regression, a.o.) with SGD trai
`linear_model.SGDRegressor`([loss, penalty, ...])	Linear model fitted by minimizing a regularized empirical loss w
`linear_model.TheilSenRegressor`([...])	Theil-Sen Estimator: robust multivariate regression model.

42.19.1 `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)

Parametersn_iter : int, optional

Maximum number of iterations. Default is 300

tol : float, optional

Stop the algorithm if w has converged. Default is 1.e-3.

alpha_1 : float, optional

Hyper-parameter : shape parameter for the Gamma distribution prior over the alpha parameter. Default is 1.e-6.

alpha_2 : float, optional

Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the alpha parameter. Default is 1.e-6.

lambda_1 : float, optional

Hyper-parameter : shape parameter for the Gamma distribution prior over the lambda parameter. Default is 1.e-6.

lambda_2 : float, optional

Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the lambda parameter. Default is 1.e-6.

compute_score : boolean, optional

If True, compute the objective function at each step of the model. Default is False.

threshold_lambda : float, optional

threshold for removing (pruning) weights with high precision from the computation. Default is 1.e+4.

fit_intercept : boolean, optional

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). Default is True.

normalize : boolean, optional, default False

If True, the regressors X will be normalized before regression.

copy_X : boolean, optional, default True.

If True, X will be copied; else, it may be overwritten.

verbose : boolean, optional, default False

Verbose mode when fitting the model.

Attributescoef_ : array, shape = (n_features)

Coefficients of the regression model (mean of distribution)

alpha_ : float

estimated precision of the noise.

lambda_ : array, shape = (n_features)

estimated precisions of the weights.

sigma_ : array, shape = (n_features, n_features)

estimated variance-covariance matrix of the weights

scores_ : float

if computed, value of the objective function (to be maximized)

Notes

See examples/linear_model/plot_ard.py for an example.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.ARDRegression()
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
...
ARDRegression(alpha_1=1e-06, alpha_2=1e-06, compute_score=False,
        copy_X=True, fit_intercept=True, lambda_1=1e-06, lambda_2=1e-06,
        n_iter=300, normalize=False, threshold_lambda=10000.0, tol=0.001,
        verbose=False)
>>> clf.predict([[1, 1]])
array([ 1.])
```

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y)	Fit the ARDRegression model according to the given training data and parameters.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*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*)

decision_function(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

static fit(*X*, *y*)

Fit the ARDRegression model according to the given training data and parameters.

Iterative procedure to maximize the evidence

ParametersX : 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, shape = [n_samples]

Target values (integers)

Returnsself : returns an instance of self.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score(*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

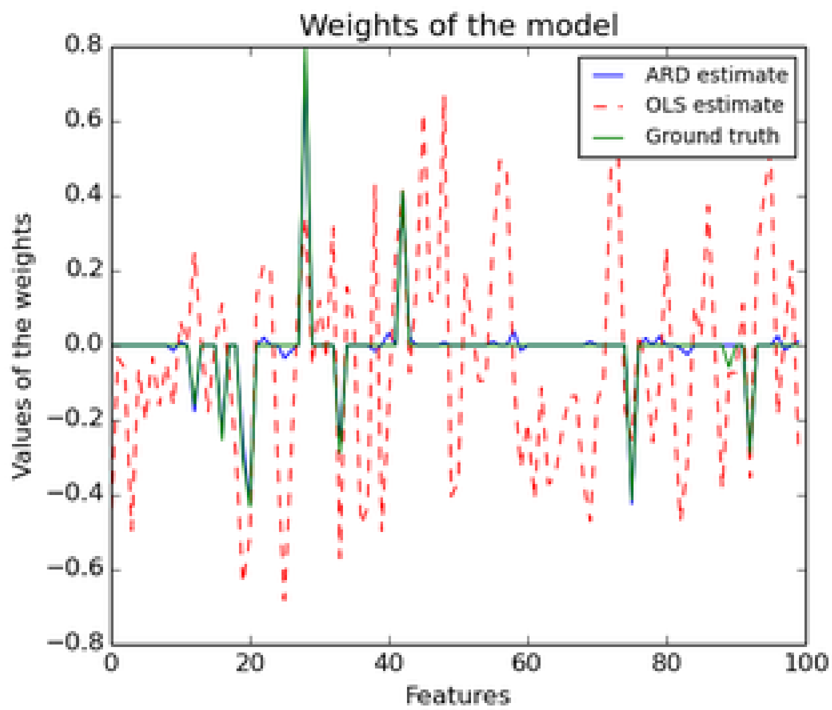


Figure 42.340: *Automatic Relevance Determination Regression (ARD)*

Examples using `sklearn.linear_model.ARDRegression`

42.19.2 `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*, *compute_score=False*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *verbose=False*)

Bayesian ridge regression

Fit a Bayesian ridge model and optimize the regularization parameters lambda (precision of the weights) and alpha (precision of the noise).

Parametersn_iter : int, optional

Maximum number of iterations. Default is 300.

tol : float, optional

Stop the algorithm if w has converged. Default is 1.e-3.

alpha_1 : float, optional

Hyper-parameter : shape parameter for the Gamma distribution prior over the alpha parameter. Default is 1.e-6

alpha_2 : float, optional

Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the alpha parameter. Default is 1.e-6.

lambda_1 : float, optional

Hyper-parameter : shape parameter for the Gamma distribution prior over the lambda parameter. Default is 1.e-6.

lambda_2 : float, optional

Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the lambda parameter. Default is 1.e-6

compute_score : boolean, optional

If True, compute the objective function at each step of the model. Default is False

fit_intercept : boolean, optional

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). Default is True.

normalize : boolean, optional, default False

If True, the regressors X will be normalized before regression.

copy_X : boolean, optional, default True

If True, X will be copied; else, it may be overwritten.

verbose : boolean, optional, default False

Verbose mode when fitting the model.

Attributescoef_ : array, shape = (n_features)

Coefficients of the regression model (mean of distribution)

alpha_ : float

estimated precision of the noise.

lambda_ : array, shape = (n_features)

estimated precisions of the weights.

scores_ : float

if computed, value of the objective function (to be maximized)

Notes

See examples/linear_model/plot_bayesian_ridge.py for an example.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.BayesianRidge()
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
...
BayesianRidge(alpha_1=1e-06, alpha_2=1e-06, compute_score=False,
        copy_X=True, fit_intercept=True, lambda_1=1e-06, lambda_2=1e-06,
        n_iter=300, normalize=False, tol=0.001, verbose=False)
>>> clf.predict([[1, 1]])
array([ 1.])
```

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y)	Fit the model
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*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*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *verbose=False*)

decision_function(*X*)
Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

static fit(*X*, *y*)
Fit the model

ParametersX : numpy array of shape [n_samples,n_features]

Training data

y : numpy array of shape [n_samples]

Target values

Returnsself : returns an instance of self.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*X*)
Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score(*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.BayesianRidge`

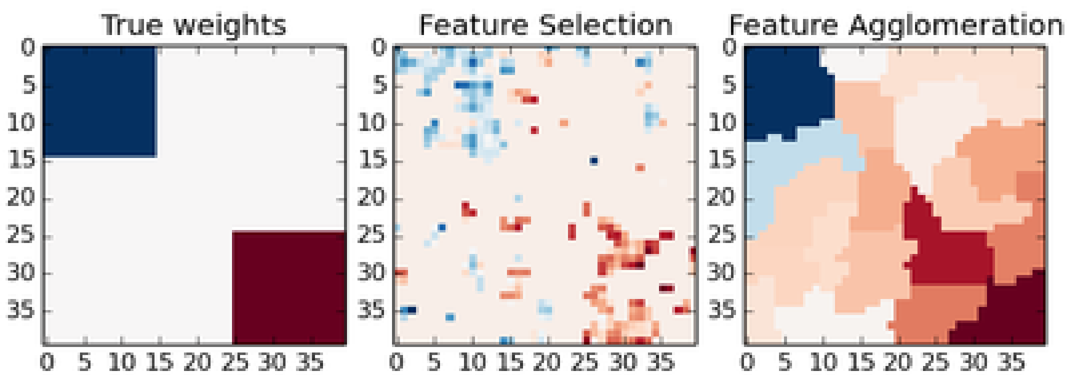


Figure 42.341: *Feature agglomeration vs. univariate selection*

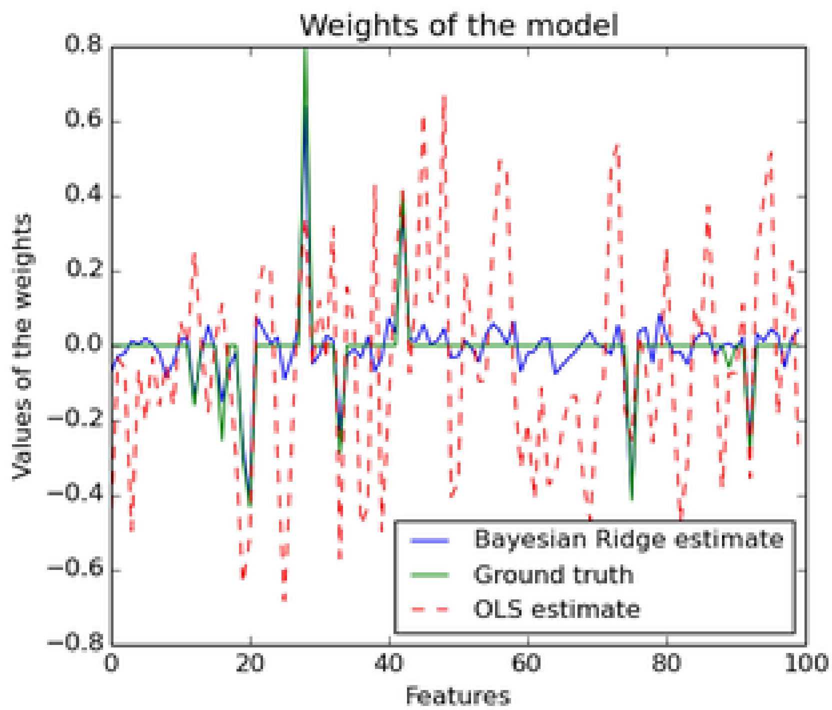


Figure 42.342: *Bayesian Ridge Regression*

42.19.3 `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.

Parametersalpha : float

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 and you should prefer 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

If `True`, the regressors X will be normalized before regression.

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. For sparse input this option is always `True` to preserve sparsity. WARNING : The `'auto'` option is deprecated and will be removed in 0.18.

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.

positive : bool, optional

When set to `True`, forces the coefficients to be positive.

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.

random_state : int, RandomState instance, or None (default)

The seed of the pseudo random number generator that selects a random feature to update. Useful only when selection is set to 'random'.

Attributescoef_ : 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_coef_` is a readonly property derived from `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:

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.

Methods

`decision_function`(X)	Decision function of the linear model
`fit`(X, y)	Fit model with coordinate descent.
`get_params`([deep])	Get parameters for this estimator.
`path`	
`predict`(X)	Predict using the linear model
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*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'*)

static `decision_function`(*X*)

Decision function of the linear model

ParametersX : numpy array or scipy.sparse matrix of shape (n_samples, n_features)

ReturnsT : array, shape = (n_samples,)

The predicted decision function

static `fit`(*X, y*)

Fit model with coordinate descent.

ParametersX : ndarray or scipy.sparse matrix, (n_samples, n_features)

Data

y : ndarray, shape = (n_samples,) or (n_samples, n_targets)

Target

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 (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)
Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

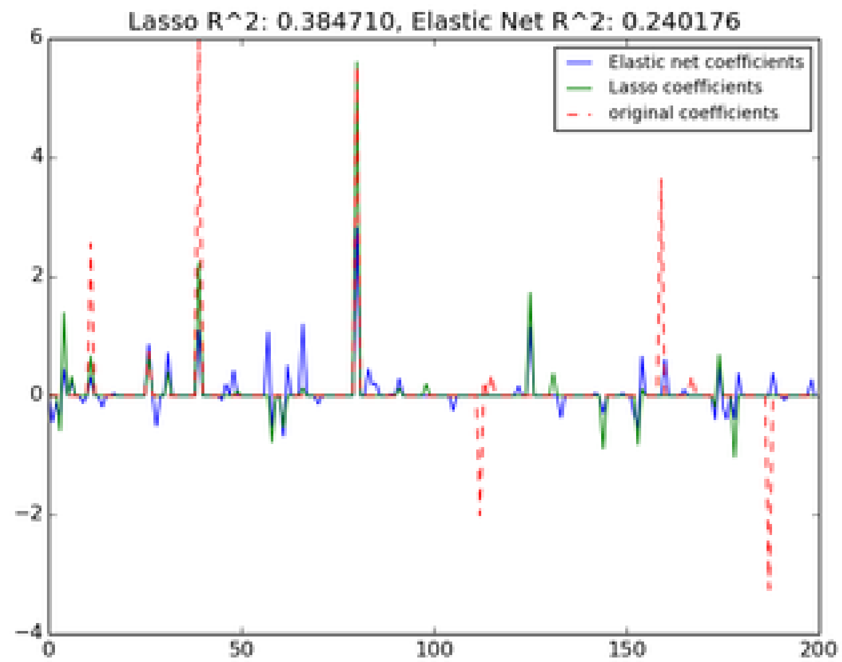


Figure 42.343: *Lasso and Elastic Net for Sparse Signals*

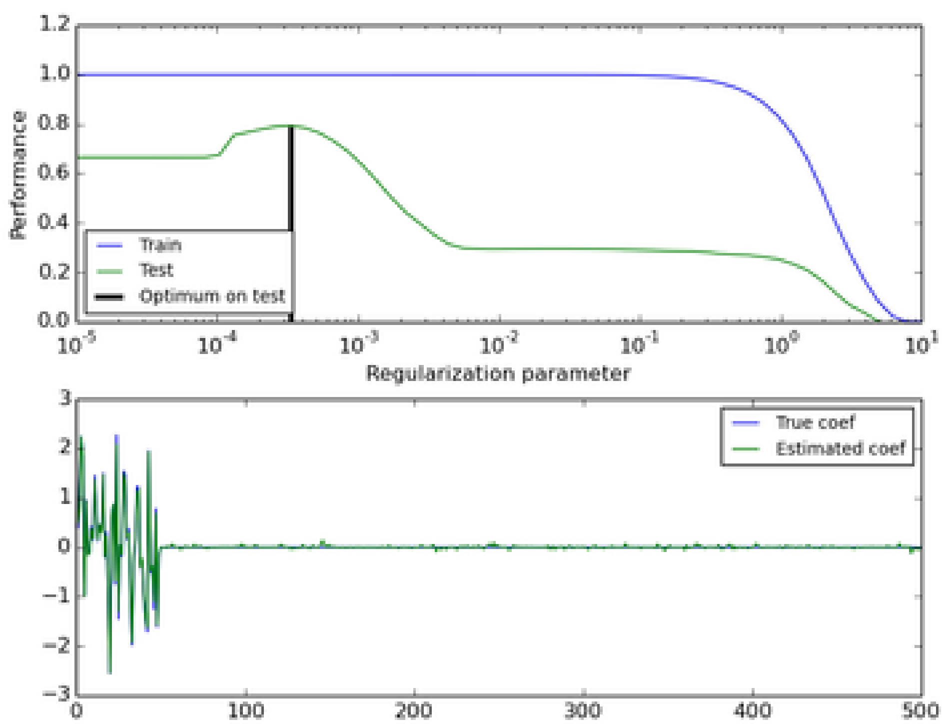


Figure 42.344: *Train error vs Test error*

Examples using `sklearn.linear_model.ElasticNet`

42.19.4 `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=1*, *positive=False*, *random_state=None*, *selection='cyclic'*)

Elastic Net model with iterative fitting along a regularization path

The best model is selected by cross-validation.

Parameters **l1_ratio** : float, 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

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 : integer or cross-validation generator, optional

If an integer is passed, it is the number of fold (default 3). Specific cross-validation objects can be passed, see the `sklearn.cross_validation` module for the list of possible objects.

verbose : bool or integer

Amount of verbosity.

n_jobs : integer, optional

Number of CPUs to use during the cross validation. If `-1`, use all the CPUs.

positive : bool, optional

When set to `True`, forces the coefficients to be positive.

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.

random_state : int, RandomState instance, or None (default)

The seed of the pseudo random number generator that selects a random feature to update. Useful only when selection is set to 'random'.

fit_intercept : boolean

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 : boolean, optional, default False

If `True`, the regressors X will be normalized before regression.

copy_X : boolean, optional, default True

If `True`, X will be copied; else, it may be overwritten.

Attributesalpha_ : 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

See examples/linear_model/lasso_path_with_crossvalidation.py for an example.

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).
```

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y)	Fit linear model with coordinate descent
get_params([deep])	Get parameters for this estimator.
path	
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*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=1*, *positive=False*, *random_state=None*, *selection='cyclic'*)

decision_function(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

fit(*X*, *y*)

Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

ParametersX : {array-like}, shape (n_samples, n_features)

Training data. Pass directly as float64, 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(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

42.19.5 sklearn.linear_model.Lars

class sklearn.linear_model.**Lars** (*fit_intercept=True*, *verbose=False*, *normalize=True*, *precompute='auto'*, *n_nonzero_coefs=500*, *eps=2.2204460492503131e-16*, *copy_X=True*, *fit_path=True*)

Least Angle Regression model a.k.a. LAR

Parametersn_nonzero_coefs : int, optional

Target number of non-zero coefficients. Use `np.inf` for no limit.

fit_intercept : boolean

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).

verbose : boolean or integer, optional

Sets the verbosity amount

normalize : boolean, optional, default False

If `True`, the regressors X will be normalized before regression.

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.

copy_X : boolean, optional, default True

If `True`, X will be copied; else, it may be overwritten.

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.

fit_path : boolean

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.

Attributesalphas_ : array, 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, 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, shape (n_features,) or (n_targets, n_features)

Parameter vector (w in the formulation formula).

intercept_ : float | array, 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
>>> clf = linear_model.Lars(n_nonzero_coefs=1)
>>> clf.fit([[-1, 1], [0, 0], [1, 1]], [-1.1111, 0, -1.1111])
...
Lars(copy_X=True, eps=..., fit_intercept=True, fit_path=True,
   n_nonzero_coefs=1, normalize=True, precompute='auto', verbose=False)
>>> print(clf.coef_)
[ 0. -1.11...]
```

Methods

`decision_function`(X)	Decision function of the linear model.
`fit`(X, y[, Xy])	Fit the model using X, y as training data.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict using the linear model
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*fit_intercept=True, verbose=False, normalize=True, precompute='auto', n_nonzero_coefs=500, eps=2.2204460492503131e-16, copy_X=True, fit_path=True*)

`decision_function`(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

static `fit`(*X, y, Xy=None*)

Fit the model using X, y as training data.

ParametersX : array-like, shape (n_samples, n_features)

Training data.

y : array-like, shape (n_samples,) or (n_samples, n_targets)

Target values.

Xy : array-like, shape (n_samples,) or (n_samples, n_targets), optional

Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

Returnsself : object

returns an instance of self.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

42.19.6 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=1*, *eps=2.2204460492503131e-16*, *copy_X=True*)

Cross-validated Least Angle Regression model

Parametersfit_intercept : boolean

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).

verbose : boolean or integer, optional

Sets the verbosity amount

normalize : boolean, optional, default False

If True, the regressors X will be normalized before regression.

copy_X : boolean, optional, default True

If `True`, X will be copied; else, it may be overwritten.

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: integer, optional :

Maximum number of iterations to perform.

cv : cross-validation generator, optional

see `sklearn.cross_validation`. If `None` is passed, default to a 5-fold strategy

max_n_alphas : integer, optional

The maximum number of points on the path used to compute the residuals in the cross-validation

n_jobs : integer, optional

Number of CPUs to use during the cross validation. If `-1`, use all the CPUs

eps : float, optional

The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.

Attributescoef_ : array, shape (n_features,)

parameter vector (w in the formulation formula)

intercept_ : float

independent term in decision function

coef_path_ : array, shape (n_features, n_alphas)

the varying values of the coefficients along the path

alpha_ : float

the estimated regularization parameter alpha

alphas_ : array, shape (n_alphas,)

the different values of alpha along the path

cv_alphas_ : array, shape (n_cv_alphas,)

all the values of alpha along the path for the different folds

cv_mse_path_ : array, 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`

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y)	Fit the model using X, y as training data.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__ (*fit_intercept=True, verbose=False, max_iter=500, normalize=True, precompute='auto', cv=None, max_n_alphas=1000, n_jobs=1, eps=2.2204460492503131e-16, copy_X=True*)

decision_function (*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

static fit (*X, y*)

Fit the model using X, y as training data.

ParametersX : array-like, shape (n_samples, n_features)

Training data.

y : array-like, shape (n_samples,)

Target values.

Returnsself : object

returns an instance of self.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

42.19.7 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).

Parametersalpha : 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` is with the Lasso object is not advised and you should prefer the LinearRegression object.

fit_intercept : boolean

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 : boolean, optional, default False

If `True`, the regressors X will be normalized before regression.

copy_X : boolean, optional, default True

If True, X will be copied; else, it may be overwritten.

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. For sparse input this option is always True to preserve sparsity. WARNING : The 'auto' option is deprecated and will be removed in 0.18.

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.

positive : bool, optional

When set to True, forces the coefficients to be positive.

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.

random_state : int, RandomState instance, or None (default)

The seed of the pseudo random number generator that selects a random feature to update. Useful only when selection is set to 'random'.

Attributescoef_ : 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_coef_ is a readonly property derived from 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, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=None,
   selection='cyclic', tol=0.0001, warm_start=False)
>>> print(clf.coef_)
[ 0.85  0.  ]
>>> print(clf.intercept_)
0.15
```

Methods

decision_function(X)	Decision function of the linear model
fit(X, y)	Fit model with coordinate descent.
get_params([deep])	Get parameters for this estimator.
path	
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*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'*)

decision_function(*X*)

Decision function of the linear model

ParametersX : numpy array or scipy.sparse matrix of shape (n_samples, n_features)

ReturnsT : array, shape = (n_samples,)

The predicted decision function

fit(*X, y*)

Fit model with coordinate descent.

ParametersX : ndarray or scipy.sparse matrix, (n_samples, n_features)

Data

y : ndarray, shape = (n_samples,) or (n_samples, n_targets)

Target

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(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.Lasso`

42.19.8 `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=1*, *positive=False*, *random_state=None*, *selection='cyclic'*)

Lasso linear model with iterative fitting along a regularization path

The best model is selected by cross-validation.

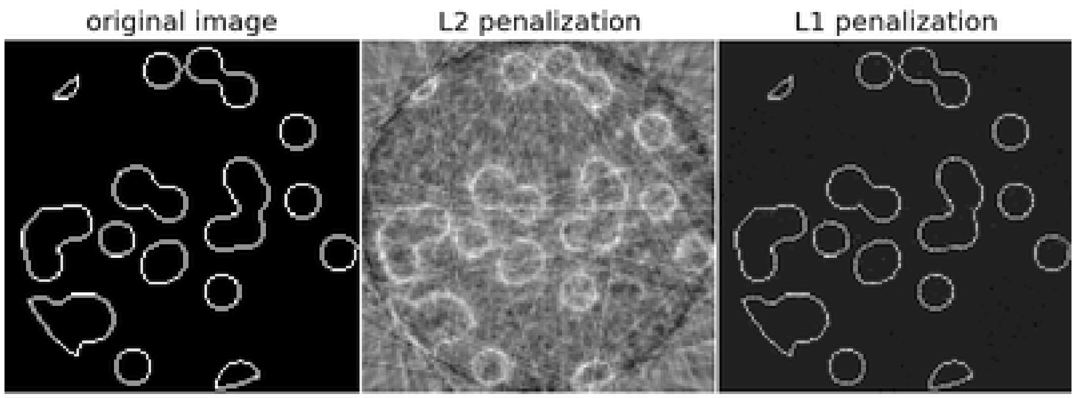


Figure 42.345: *Compressive sensing: tomography reconstruction with L1 prior (Lasso)*

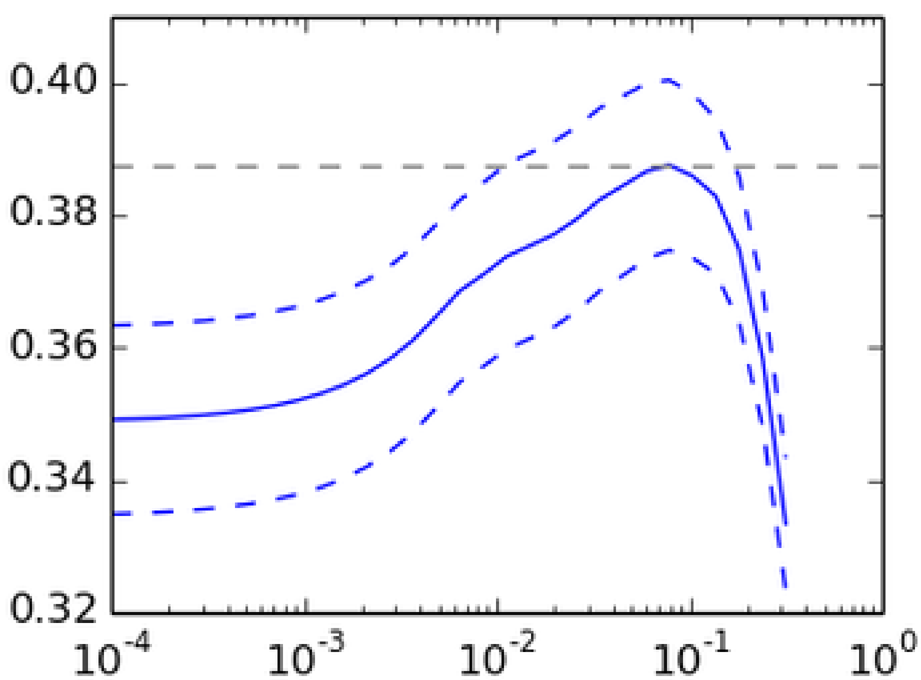


Figure 42.346: *Cross-validation on diabetes Dataset Exercise*

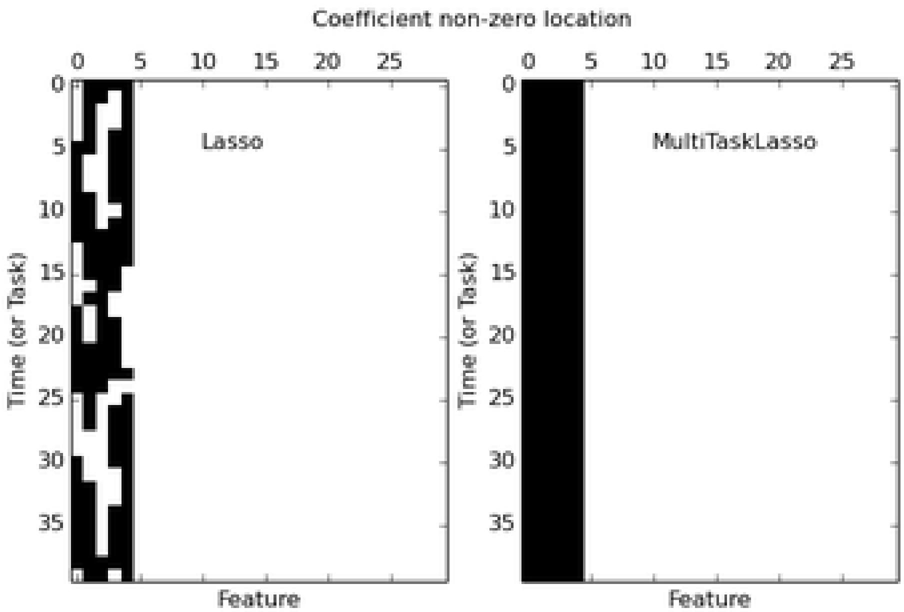


Figure 42.347: *Joint feature selection with multi-task Lasso*

Figure 42.348: *Lasso on dense and sparse data*

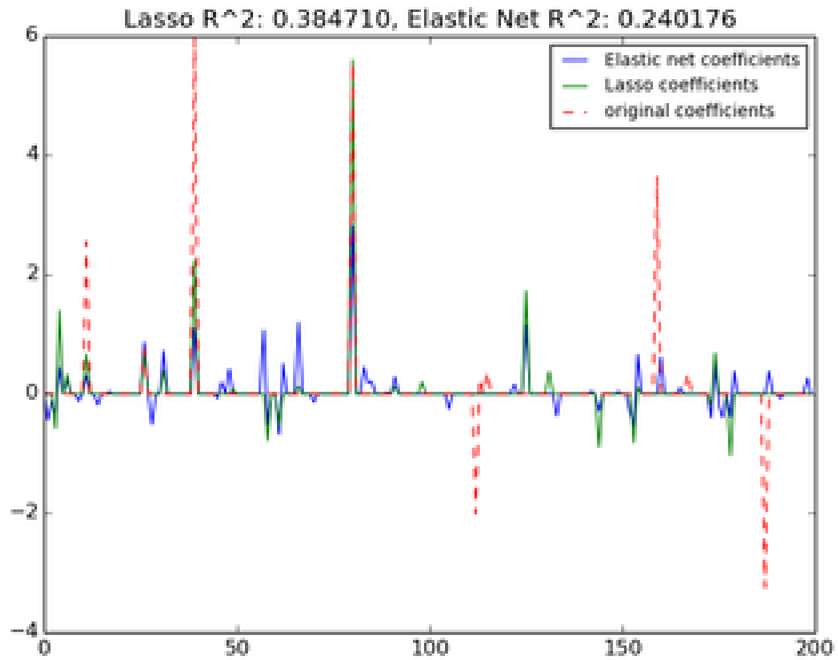


Figure 42.349: *Lasso and Elastic Net for Sparse Signals*

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Parameterseps : 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

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 : integer or cross-validation generator, optional

If an integer is passed, it is the number of fold (default 3). Specific cross-validation objects can be passed, see the `sklearn.cross_validation` module for the list of possible objects.

verbose : bool or integer

Amount of verbosity.

n_jobs : integer, optional

Number of CPUs to use during the cross validation. If `-1`, use all the CPUs.

positive : bool, optional

If positive, restrict regression coefficients to be positive

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.

random_state : int, RandomState instance, or None (default)

The seed of the pseudo random number generator that selects a random feature to update. Useful only when selection is set to 'random'.

fit_intercept : boolean, 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 : boolean, optional, default False

If `True`, the regressors X will be normalized before regression.

copy_X : boolean, optional, default True

If `True`, X will be copied; else, it may be overwritten.

Attributesalpha_ : 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

See examples/linear_model/lasso_path_with_crossvalidation.py for an example.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y)	Fit linear model with coordinate descent
get_params([deep])	Get parameters for this estimator.
path	
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*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=1, positive=False, random_state=None, selection='cyclic'*)

decision_function(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

fit(*X, y*)

Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

ParametersX : {array-like}, shape (n_samples, n_features)

Training data. Pass directly as float64, 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(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

Parameters X : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returns C : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

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.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns self :

Examples using `sklearn.linear_model.LassoCV`

42.19.9 `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=2.2204460492503131e-16*, *copy_X=True*, *fit_path=True*)

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
```

Parameters alpha : float

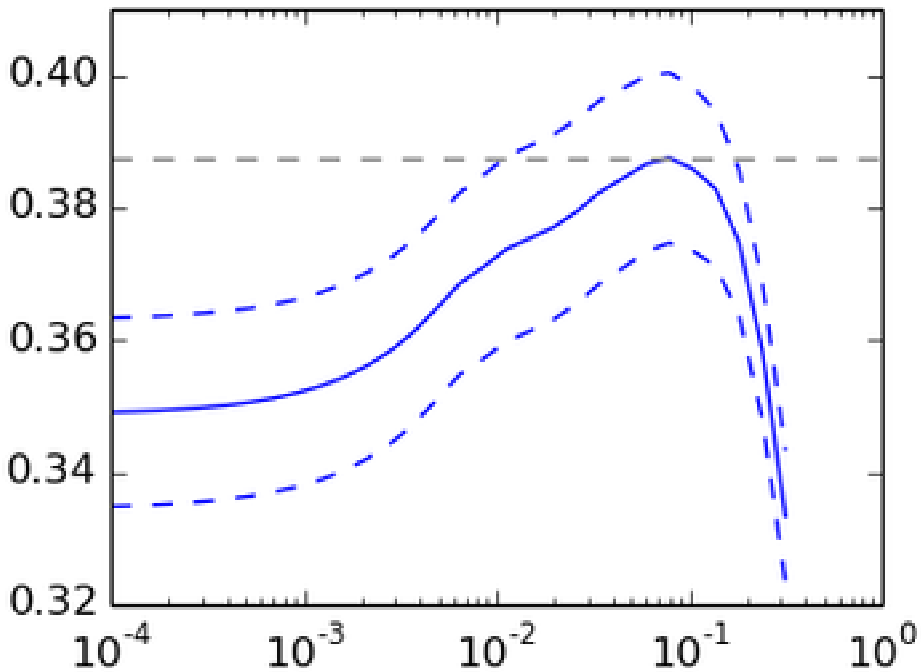


Figure 42.350: *Cross-validation on diabetes Dataset Exercise*

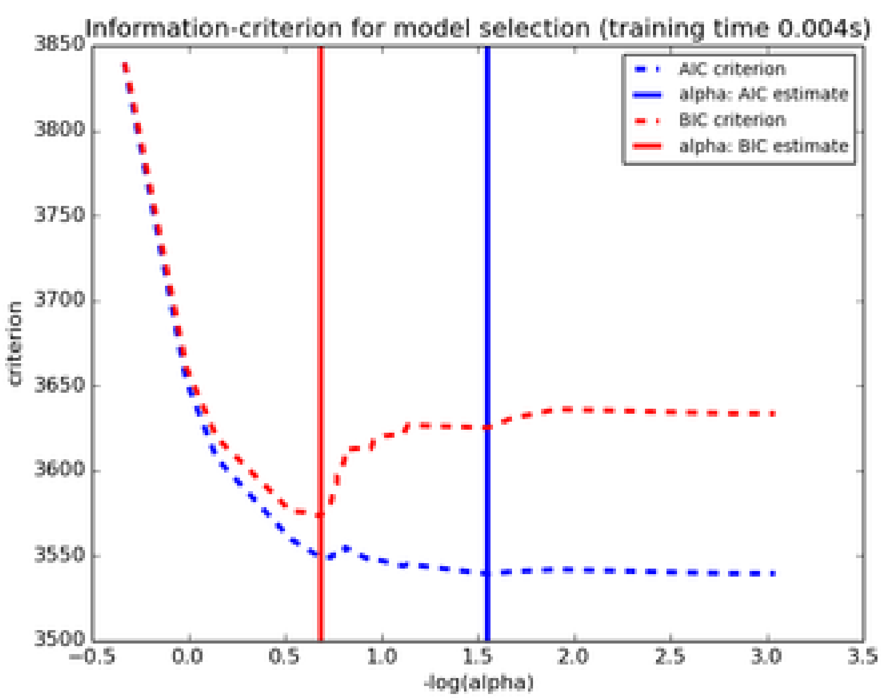


Figure 42.351: *Lasso model selection: Cross-Validation / AIC / BIC*

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 : boolean

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).

verbose : boolean or integer, optional

Sets the verbosity amount

normalize : boolean, optional, default False

If True, the regressors X will be normalized before regression.

copy_X : boolean, optional, default True

If True, X will be copied; else, it may be overwritten.

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 : integer, optional

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.

fit_path : boolean

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.

Attributesalphas_ : array, 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, 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, shape (n_features,) or (n_targets, n_features)

Parameter vector (w in the formulation formula).

intercept_ : float | array, 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, sklearn.decomposition.sparse_encode

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.LassoLars(alpha=0.01)
>>> clf.fit([[-1, 1], [0, 0], [1, 1]], [-1, 0, -1])
...
LassoLars(alpha=0.01, copy_X=True, eps=..., fit_intercept=True,
     fit_path=True, max_iter=500, normalize=True, precompute='auto',
     verbose=False)
>>> print(clf.coef_)
[ 0.         -0.963257...]
```

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y[, Xy])	Fit the model using X, y as training data.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*alpha=1.0, fit_intercept=True, verbose=False, normalize=True, precompute='auto', max_iter=500, eps=2.2204460492503131e-16, copy_X=True, fit_path=True*)

decision_function(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

fit(*X, y, Xy=None*)

Fit the model using X, y as training data.

ParametersX : array-like, shape (n_samples, n_features)

Training data.

y : array-like, shape (n_samples,) or (n_samples, n_targets)

Target values.

Xy : array-like, shape (n_samples,) or (n_samples, n_targets), optional

Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

Returnsself : object

returns an instance of self.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

42.19.10 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=1*, *eps=2.2204460492503131e-16*, *copy_X=True*)

Cross-validated Lasso, using the LARS algorithm

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Parametersfit_intercept : boolean

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).

verbose : boolean or integer, optional

Sets the verbosity amount

normalize : boolean, optional, default False

If True, the regressors X will be normalized before regression.

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 : integer, optional

Maximum number of iterations to perform.

cv : cross-validation generator, optional

see sklearn.cross_validation module. If None is passed, default to a 5-fold strategy

max_n_alphas : integer, optional

The maximum number of points on the path used to compute the residuals in the cross-validation

n_jobs : integer, optional

Number of CPUs to use during the cross validation. If `-1`, use all the CPUs

eps : float, optional

The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.

copy_X : boolean, optional, default True

If True, X will be copied; else, it may be overwritten.

Attributescoef_ : array, shape (n_features,)

parameter vector (w in the formulation formula)

intercept_ : float

independent term in decision function.

coef_path_ : array, shape (n_features, n_alphas)

the varying values of the coefficients along the path

alpha_ : float

the estimated regularization parameter alpha

alphas_ : array, shape (n_alphas,)

the different values of alpha along the path

cv_alphas_ : array, shape (n_cv_alphas,)

all the values of alpha along the path for the different folds

cv_mse_path_ : array, 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.

Methods

`decision_function`(X)	Decision function of the linear model.
`fit`(X, y)	Fit the model using X, y as training data.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict using the linear model
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

`__init__`(*fit_intercept=True*, *verbose=False*, *max_iter=500*, *normalize=True*, *precompute='auto'*, *cv=None*, *max_n_alphas=1000*, *n_jobs=1*, *eps=2.2204460492503131e-16*, *copy_X=True*)

`decision_function`(*X*)

Decision function of the linear model.

Parameters X : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returns C : array, shape = (n_samples,)

Returns predicted values.

`fit`(*X*, *y*)

Fit the model using X, y as training data.

ParametersX : array-like, shape (n_samples, n_features)

Training data.

y : array-like, shape (n_samples,)

Target values.

Returnsself : object

returns an instance of self.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.LassoLarsCV`

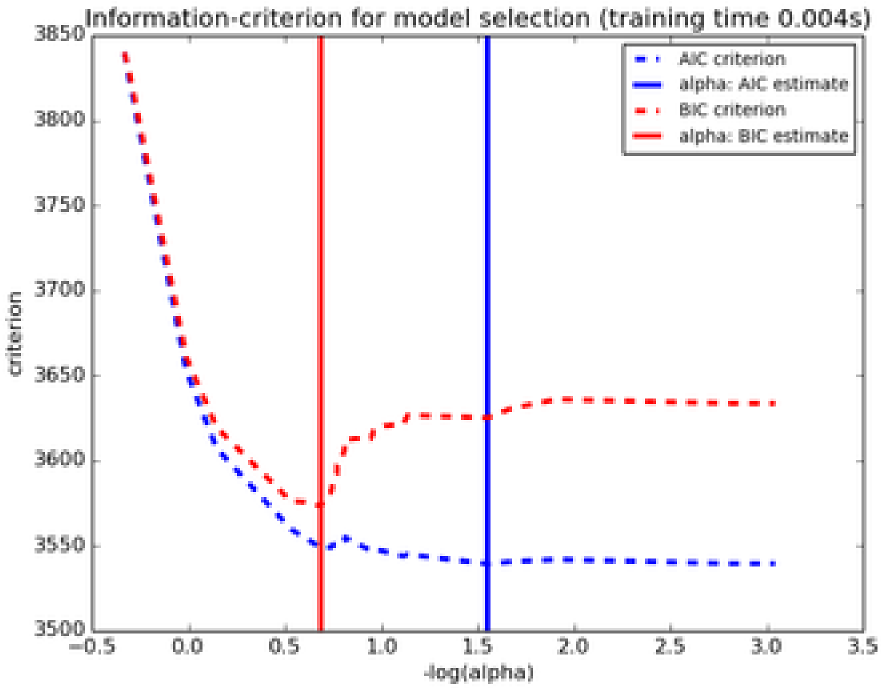


Figure 42.352: *Lasso model selection: Cross-Validation / AIC / BIC*

42.19.11 `sklearn.linear_model.LassoLarsIC`

class `sklearn.linear_model.`**`LassoLarsIC`**(*criterion='aic'*, *fit_intercept=True*, *verbose=False*, *normalize=True*, *precompute='auto'*, *max_iter=500*, *eps=2.2204460492503131e-16*, *copy_X=True*)

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.

Parameterscriterion : 'bic' | 'aic'

The type of criterion to use.

fit_intercept : boolean

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).

verbose : boolean or integer, optional

Sets the verbosity amount

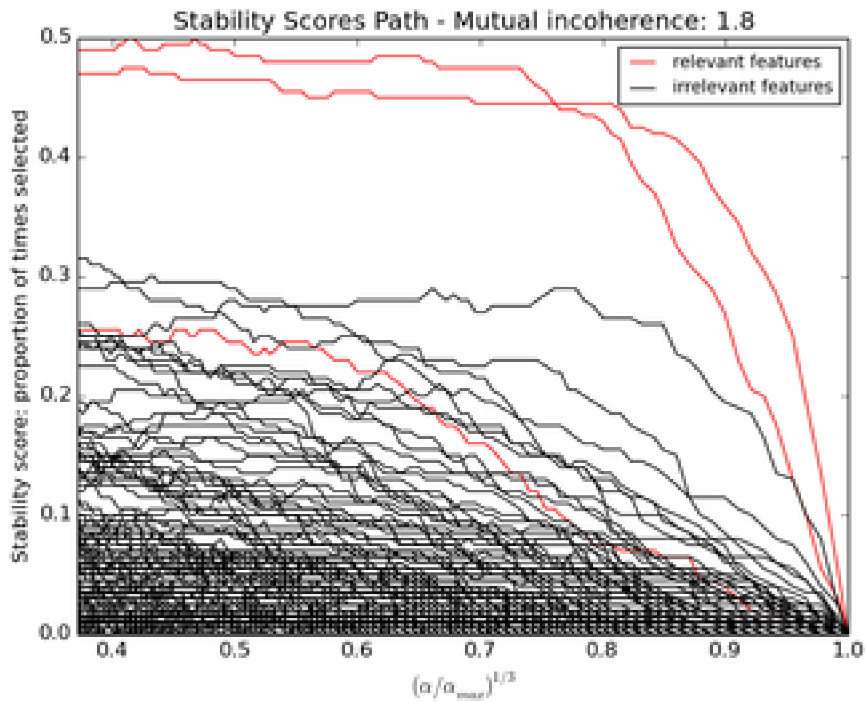


Figure 42.353: *Sparse recovery: feature selection for sparse linear models*

normalize : boolean, optional, default False

If True, the regressors X will be normalized before regression.

copy_X : boolean, optional, default True

If True, X will be copied; else, it may be overwritten.

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 : integer, optional

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.

Attributescoef_ : array, 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, shape (n_alphas,)

The value of the information criteria ('aic', 'bic') across all alphas. The alpha which has the smallest information criteria is chosen.

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.

http://en.wikipedia.org/wiki/Akaike_information_criterion http://en.wikipedia.org/wiki/Bayesian_information_criterion

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.LassoLarsIC(criterion='bic')
>>> clf.fit([[-1, 1], [0, 0], [1, 1]], [-1.1111, 0, -1.1111])
...
LassoLarsIC(copy_X=True, criterion='bic', eps=..., fit_intercept=True,
      max_iter=500, normalize=True, precompute='auto',
      verbose=False)
>>> print(clf.coef_)
[ 0.  -1.11...]
```

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y[, copy_X])	Fit the model using X, y as training data.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*criterion='aic', fit_intercept=True, verbose=False, normalize=True, precompute='auto', max_iter=500, eps=2.2204460492503131e-16, copy_X=True*)

decision_function(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

static fit (*X*, *y*, *copy_X=True*)

Fit the model using X, y as training data.

ParametersX : array-like, shape (n_samples, n_features)

training data.

y : array-like, shape (n_samples,)

target values.

copy_X : boolean, optional, default True

If `True`, X will be copied; else, it may be overwritten.

Returnsself : object

returns an instance of self.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.LassoLarsIC`

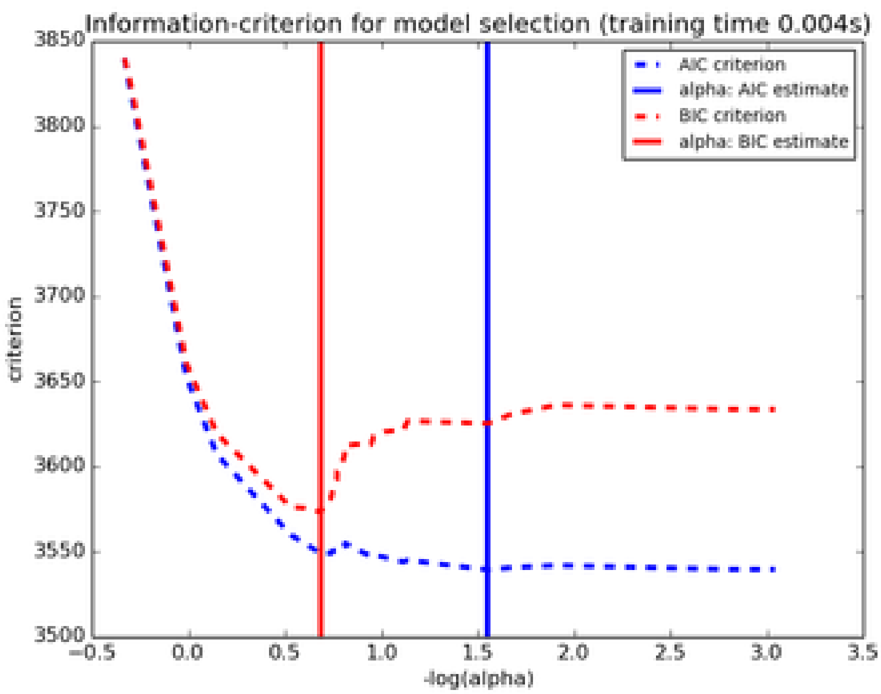


Figure 42.354: *Lasso model selection: Cross-Validation / AIC / BIC*

42.19.12 `sklearn.linear_model`.LinearRegression

class `sklearn.linear_model.`**`LinearRegression`** (*fit_intercept=True*, *normalize=False*, *copy_X=True*, *n_jobs=1*)
Ordinary least squares Linear Regression.

Parametersfit_intercept : boolean, optional

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 : boolean, optional, default False

If True, the regressors X will be normalized before regression.

copy_X : boolean, optional, default True

If True, X will be copied; else, it may be overwritten.

n_jobs : int, optional, default 1

The number of jobs to use for the computation. If -1 all CPUs are used. This will only provide speedup for n_targets > 1 and sufficient large problems.

Attributescoef_ : array, 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.

intercept_ : array

Independent term in the linear model.

Notes

From the implementation point of view, this is just plain Ordinary Least Squares (scipy.linalg.lstsq) wrapped as a predictor object.

Methods

`decision_function`(X)	Decision function of the linear model.
`fit`(X, y[, n_jobs])	Fit linear model.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict using the linear model
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__` (*fit_intercept=True*, *normalize=False*, *copy_X=True*, *n_jobs=1*)

`decision_function` (*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

static `fit` (*X*, *y*, *n_jobs=1*)

Fit linear model.

ParametersX : numpy array or sparse matrix of shape [n_samples,n_features]

Training data

y : numpy array of shape [n_samples, n_targets]

Target values

Returnsself : returns an instance of self.

`get_params` (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)
Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.LinearRegression`

42.19.13 `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='liblinear', max_iter=100, multi_class='ovr', verbose=0*)

Logistic Regression (aka logit, MaxEnt) classifier.

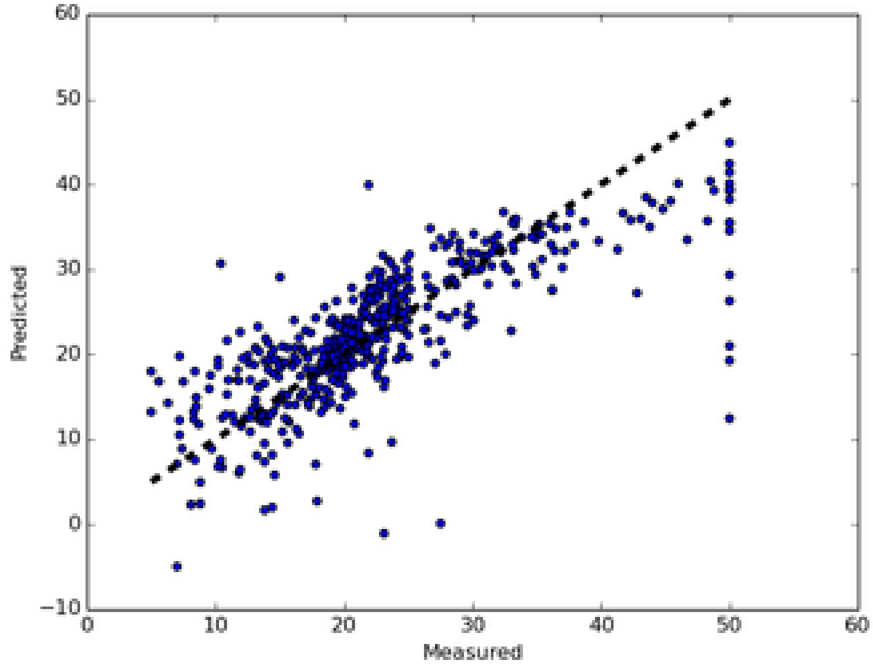


Figure 42.355: *Plotting Cross-Validated Predictions*

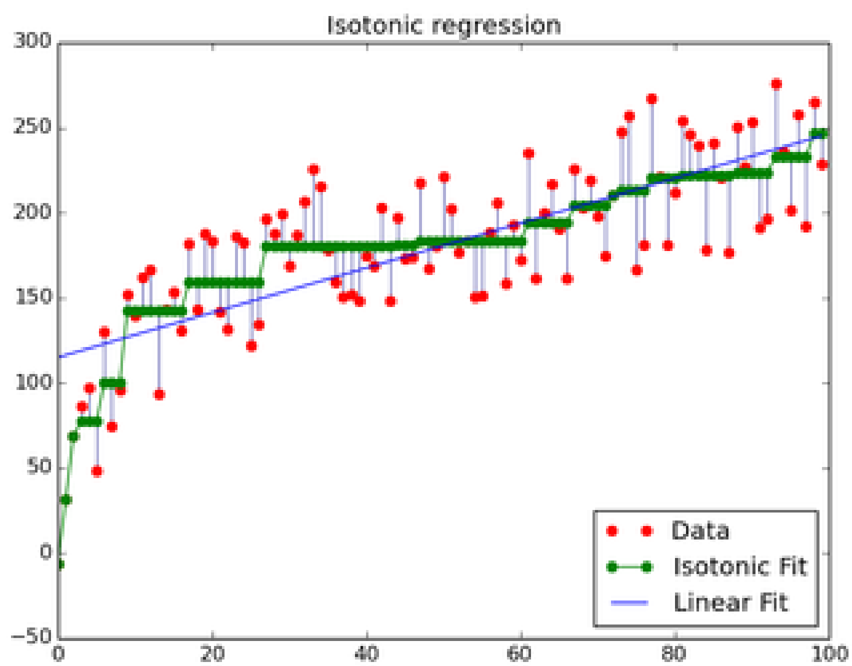


Figure 42.356: *Isotonic Regression*

Figure 42.357: *Face completion with a multi-output estimators*

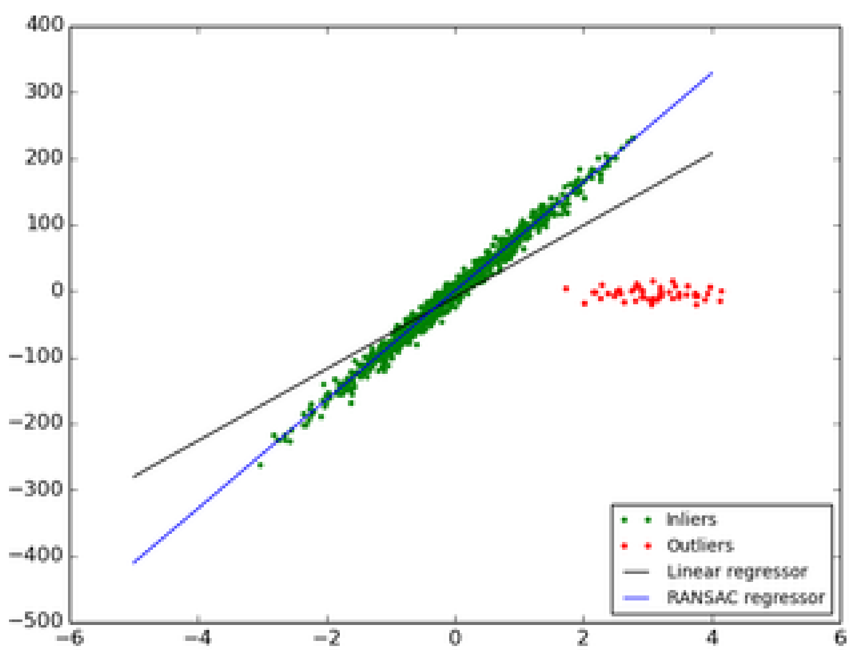


Figure 42.358: *Robust linear model estimation using RANSAC*

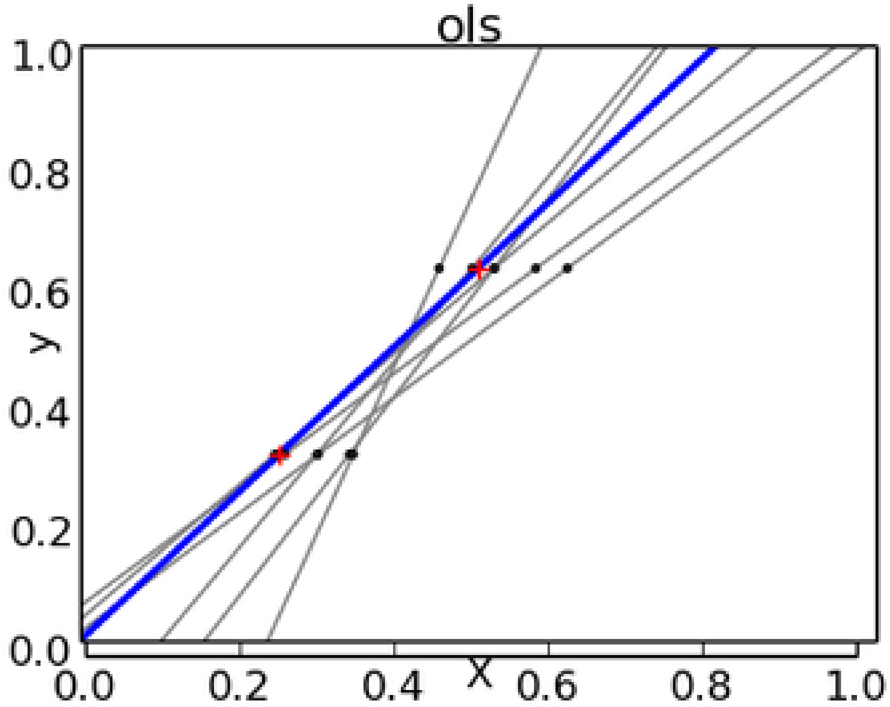


Figure 42.359: *Ordinary Least Squares and Ridge Regression Variance*

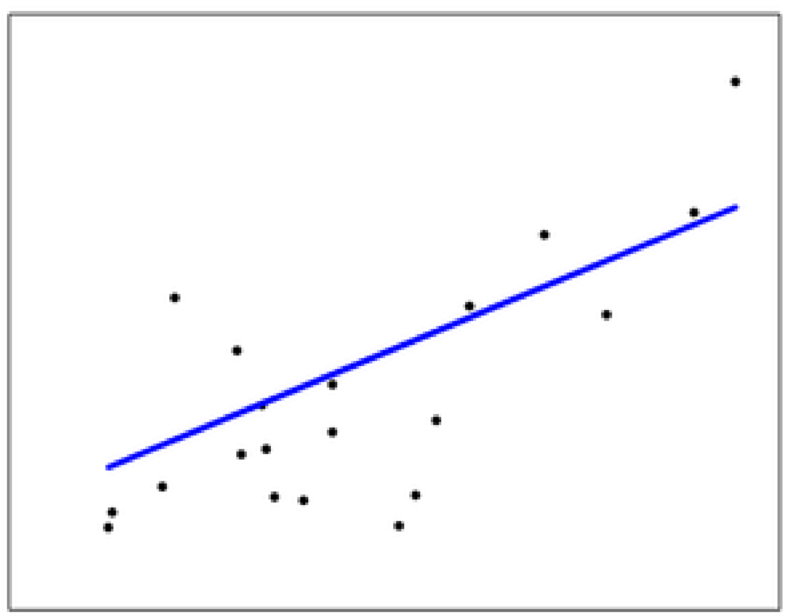


Figure 42.360: *Linear Regression Example*

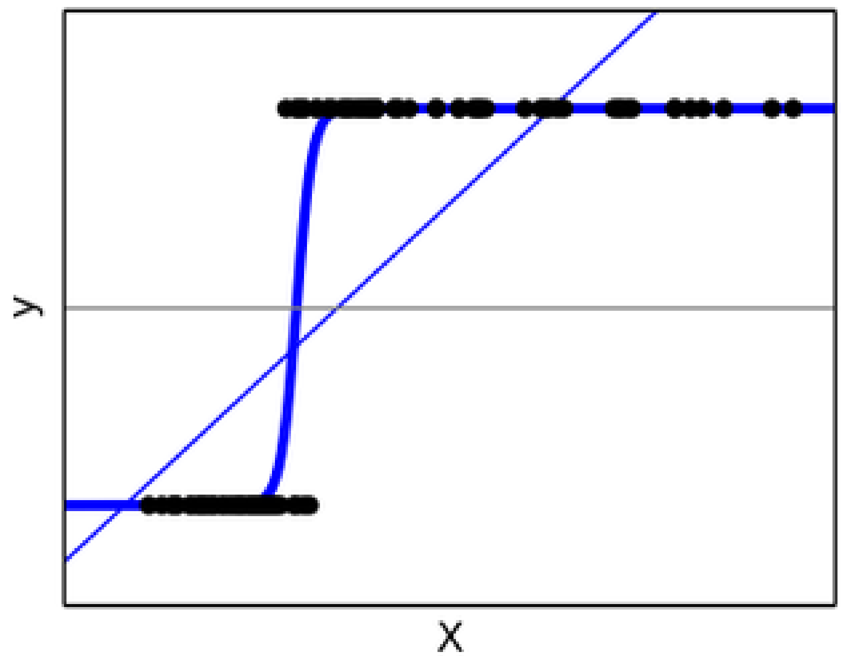


Figure 42.361: *Logit function*

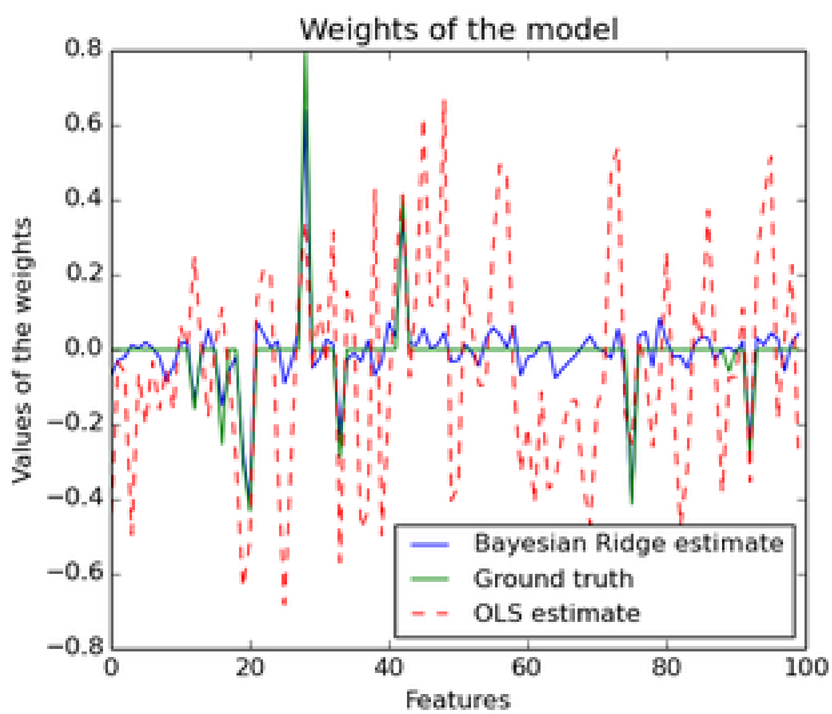


Figure 42.362: *Bayesian Ridge Regression*

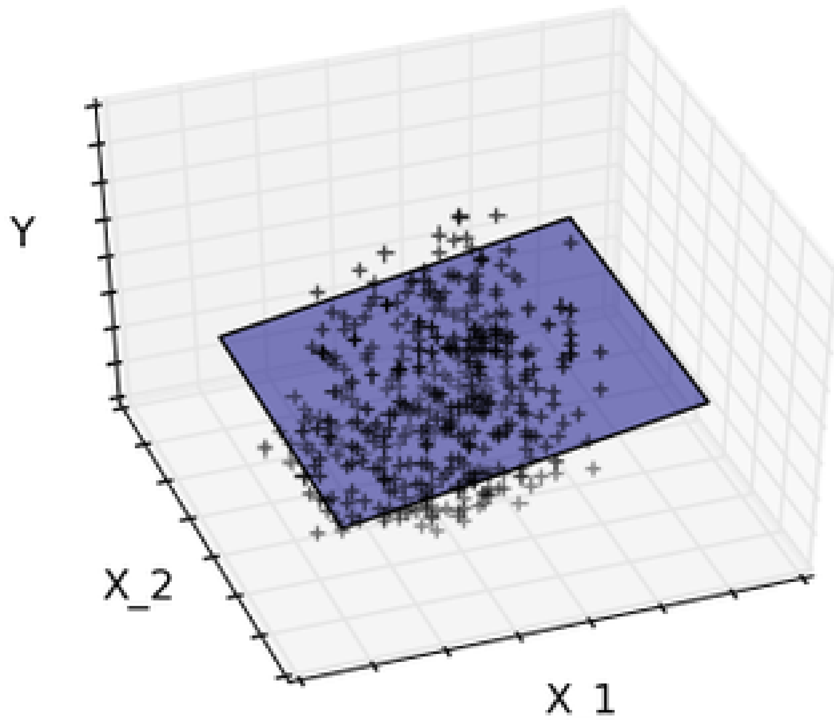


Figure 42.363: *Sparsity Example: Fitting only features 1 and 2*

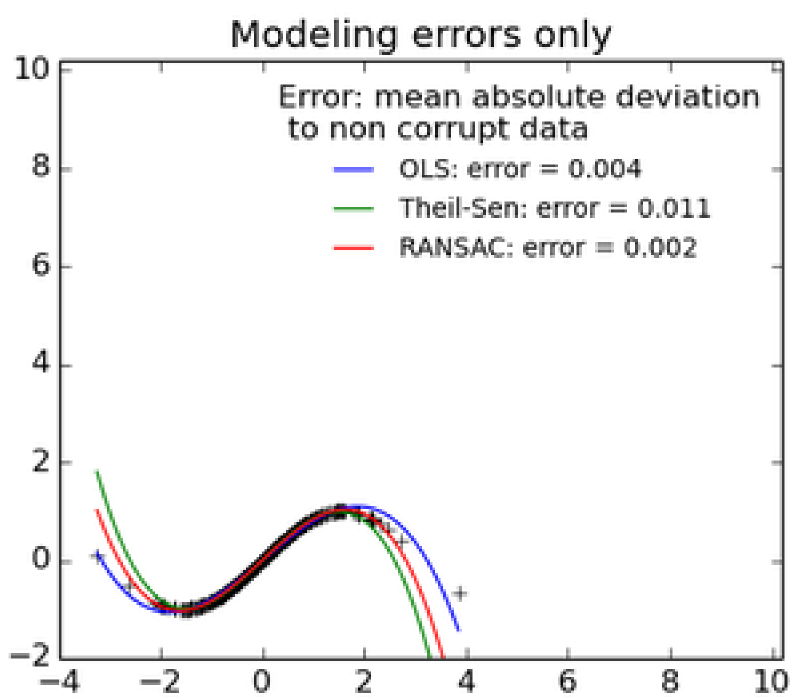


Figure 42.364: *Robust linear estimator fitting*

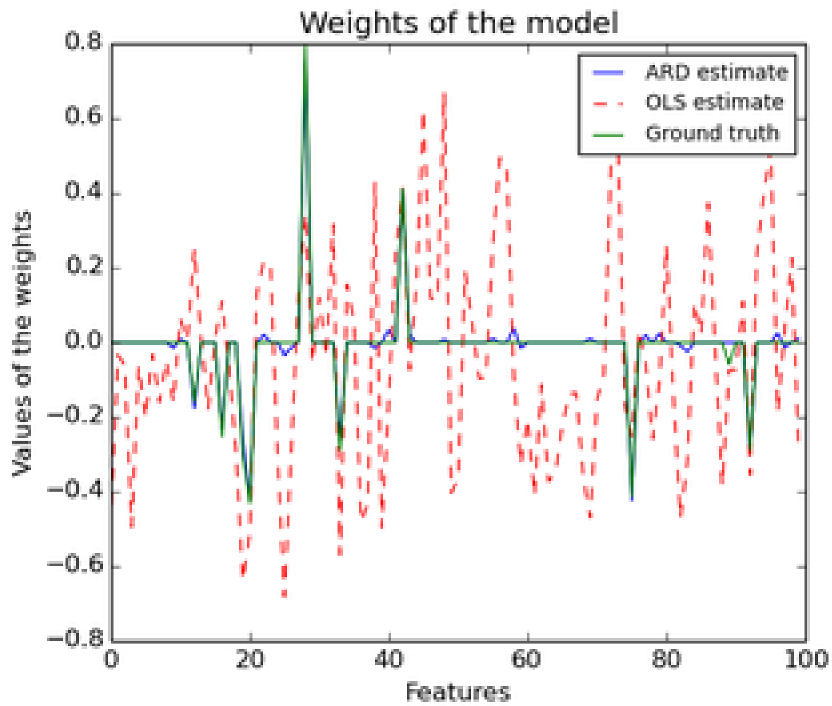


Figure 42.365: *Automatic Relevance Determination Regression (ARD)*

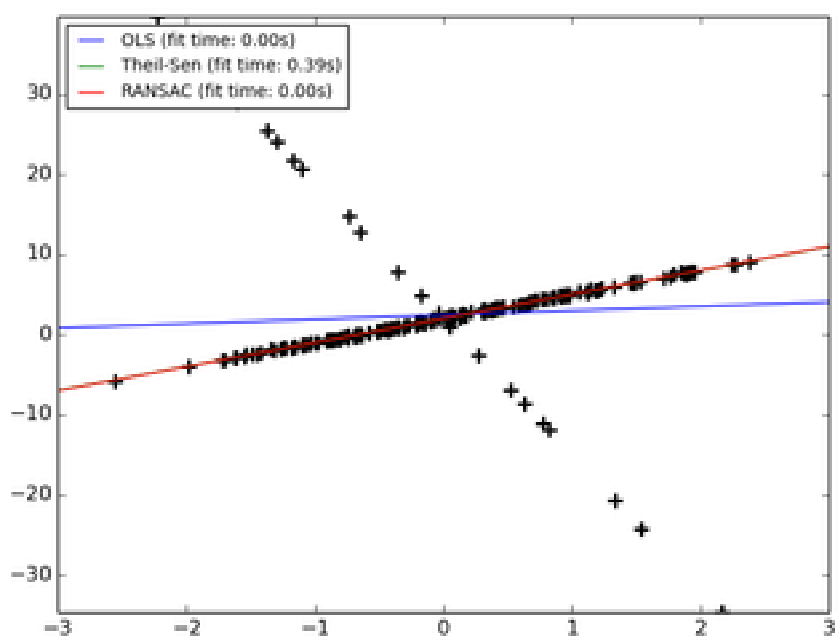


Figure 42.366: *Theil-Sen Regression*

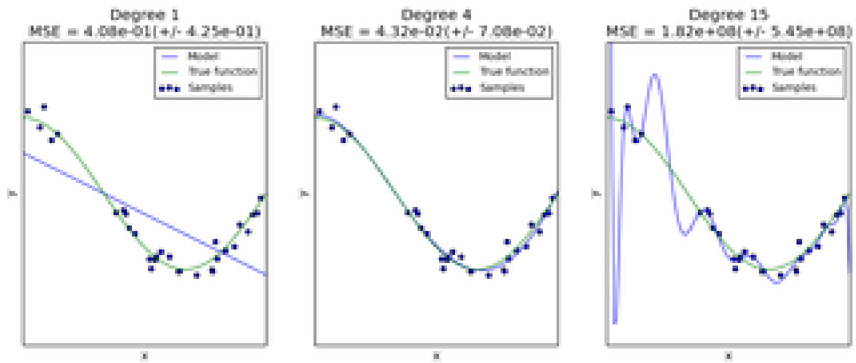


Figure 42.367: *Underfitting vs. Overfitting*

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' and 'newton-cg' solvers.)

This class implements regularized logistic regression using the *liblinear* library, newton-cg and lbfgs solvers. 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 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.

Parameterspenalty : str, 'l1' or 'l2'

Used to specify the norm used in the penalization. The newton-cg and lbfgs solvers support only l2 penalties.

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.

C : float, optional (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 the decision function.

intercept_scaling : float, default: 1

Useful only if solver is liblinear. 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, 'auto'}, optional

Over-/undersamples the samples of each class according to the given weights. If not given, all classes are supposed to have weight one. The 'auto' mode selects weights inversely proportional to class frequencies in the training set.

max_iter : int

Useful only for the newton-cg and lbfgs solvers. Maximum number of iterations taken for the solvers to converge.

random_state : int seed, RandomState instance, or None (default)

The seed of the pseudo random number generator to use when shuffling the data.

solver : {'newton-cg', 'lbfgs', 'liblinear'}

Algorithm to use in the optimization problem.

tol : float, optional

Tolerance for stopping criteria.

multi_class : str, {'ovr', 'multinomial'}

Multiclass option can be either 'ovr' or 'multinomial'. If the option chosen is 'ovr', then a binary problem is fit for each label. Else the loss minimised is the multinomial loss fit across the entire probability distribution. Works only for the 'lbfgs' solver.

verbose : int

For the liblinear and lbfgs solvers set verbose to any positive number for verbosity.

Attributescoef_ : array, shape (n_classes, n_features)

Coefficient of the features in the decision function.

intercept_ : array, shape (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.

n_iter_ : int

Maximum of the actual number of iterations across all classes. Valid only for the liblinear solver.

See also:

`sklearn.linear_model.SGDClassifier`

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

LIBLINEAR – A Library for Large Linear Classificationhttp://www.csie.ntu.edu.tw/~cjlin/liblinear/

Hsiang-Fu Yu, Fang-Lan Huang, Chih-Jen Lin (2011). Dual coordinate descentmethods for logistic regression and maximum entropy models. Machine Learning 85(1-2):41-75. http://www.csie.ntu.edu.tw/~cjlin/papers/maxent_dual.pdf

Methods

`decision_function`(X)	Predict confidence scores for samples.
`densify`()	Convert coefficient matrix to dense array format.
`fit`(X, y)	Fit the model according to the given training data.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict class labels for samples in X.
`predict_log_proba`(X)	Log of probability estimates.
`predict_proba`(X)	Probability estimates.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.
`sparsify`()	Convert coefficient matrix to sparse format.
`transform`(X[, threshold])	Reduce X to its most important features.

static `__init__`(*penalty='l2'*, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *random_state=None*, *solver='liblinear'*, *max_iter=100*, *multi_class='ovr'*, *verbose=0*)

`decision_function`(*X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returnsarray, 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`()

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.

Returnsself: estimator :

static `fit`(*X*, *y*)

Fit the model according to the given training data.

ParametersX : {array-like, sparse matrix}, 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.

Returnsself : object

Returns self.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*X*)

Predict class labels for samples in X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Samples.

ReturnsC : array, shape = [n_samples]

Predicted class label per sample.

static **predict_log_proba**(*X*)

Log of probability estimates.

The returned estimates for all classes are ordered by the label of classes.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsT : 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_`.

static **predict_proba**(*X*)

Probability estimates.

The returned estimates for all classes are ordered by the label of classes.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsT : array-like, 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 (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

sparsify ()
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.

Returnsself: 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.

transform (*X*, *threshold=None*)
Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Examples using `sklearn.linear_model.LogisticRegression`

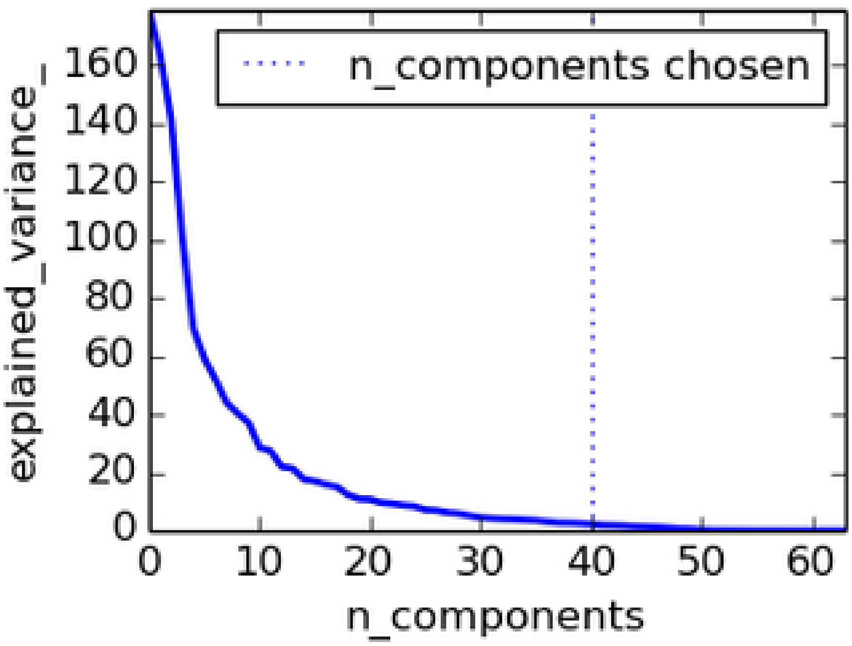


Figure 42.368: *Pipelining: chaining a PCA and a logistic regression*

42.19.14 `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=1, verbose=0, refit=True, intercept_scaling=1.0, multi_class='ovr'*)

Logistic Regression CV (aka logit, MaxEnt) classifier.

This class implements logistic regression using liblinear, newton-cg or LBFGS optimizer. The newton-cg 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.

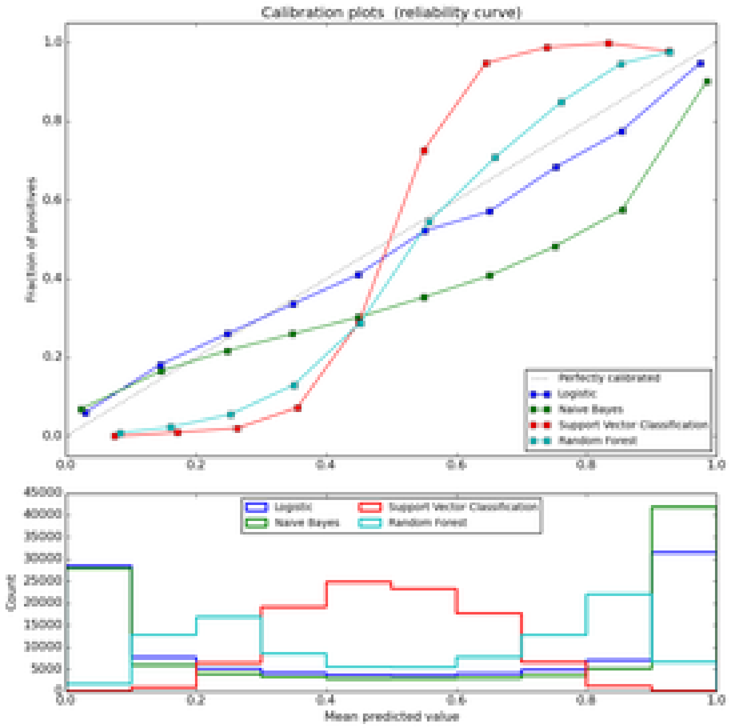


Figure 42.369: *Comparison of Calibration of Classifiers*

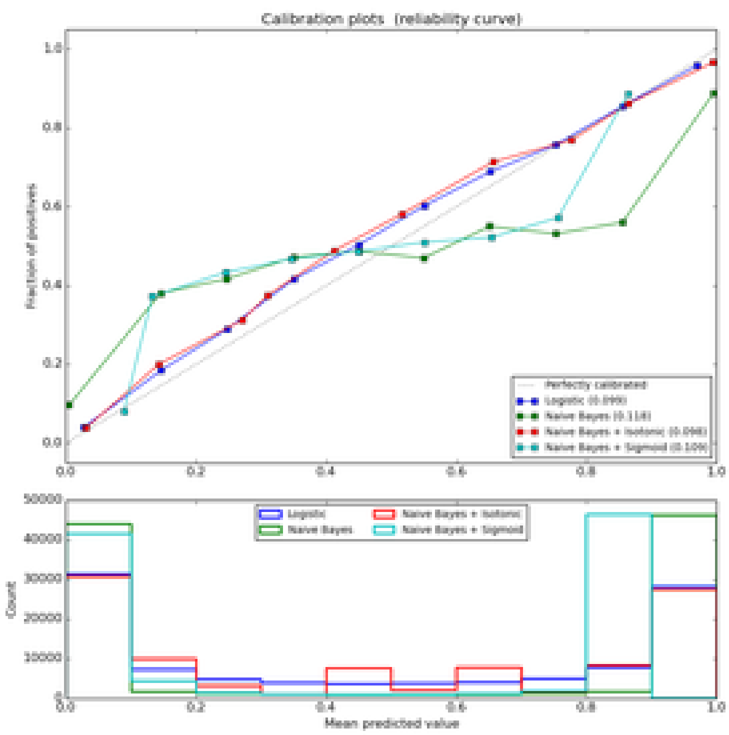


Figure 42.370: *Probability Calibration curves*

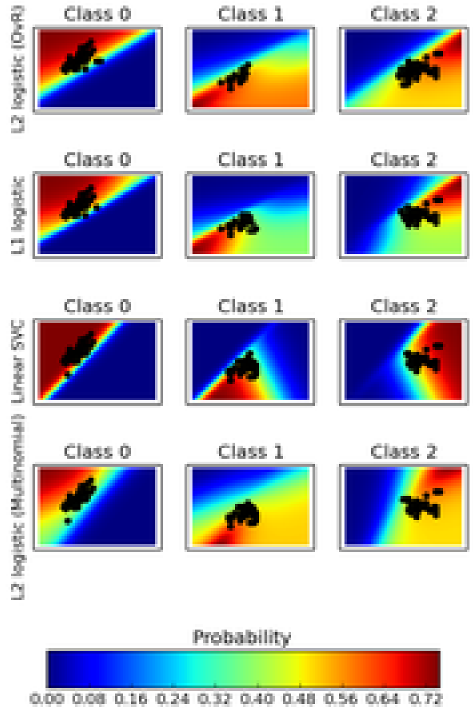


Figure 42.371: *Plot classification probability*

Figure 42.372: *Digits Classification Exercise*

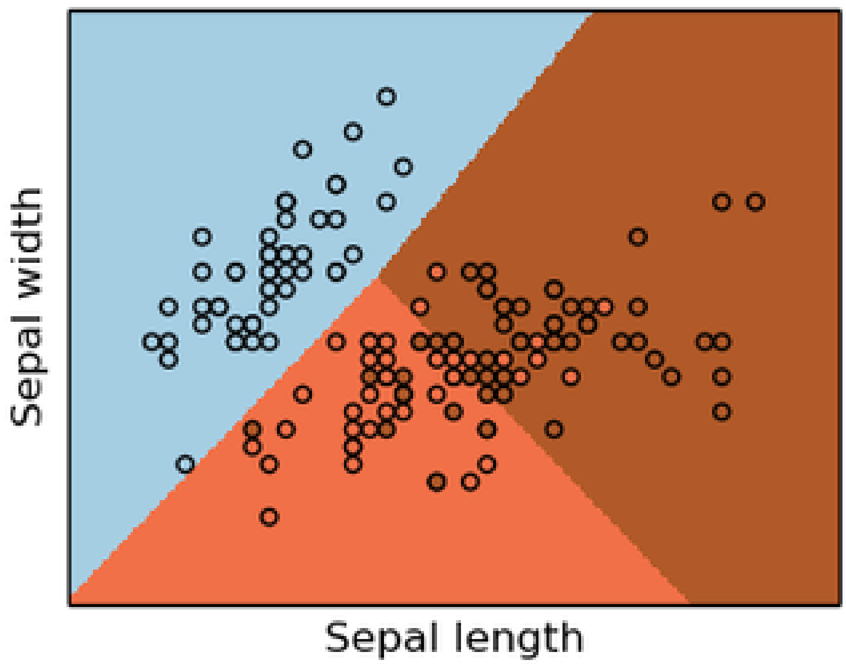


Figure 42.373: *Logistic Regression 3-class Classifier*

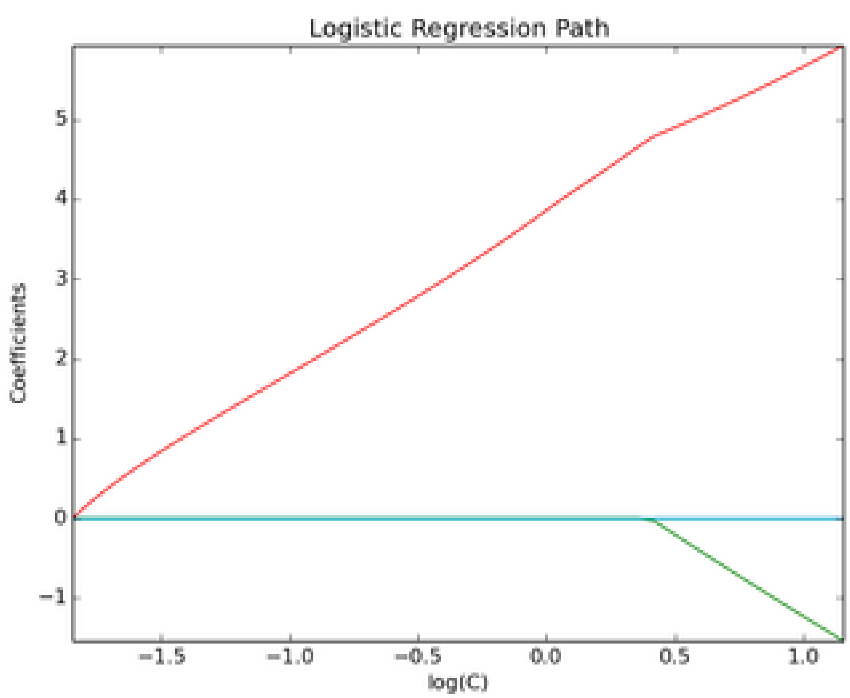


Figure 42.374: *Path with L1- Logistic Regression*

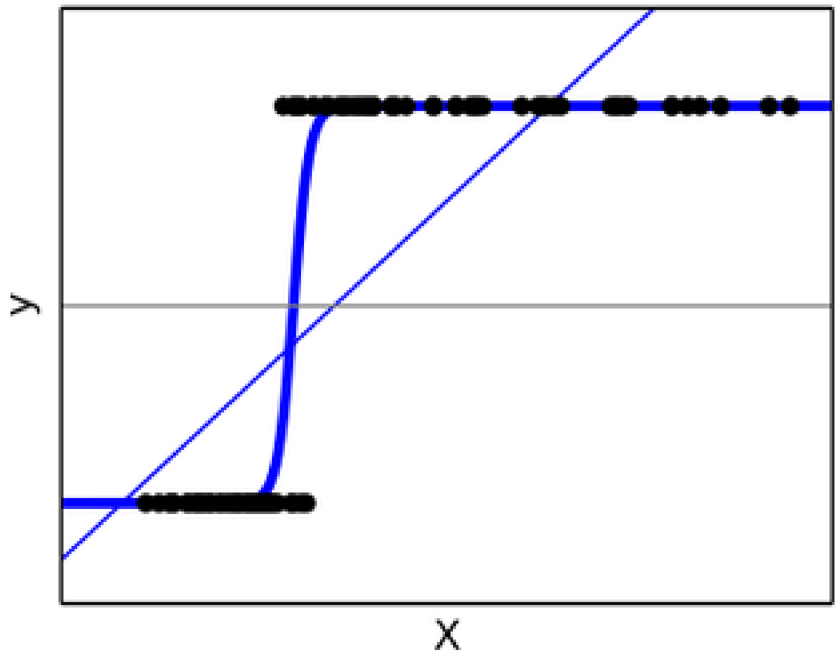


Figure 42.375: *Logit function*

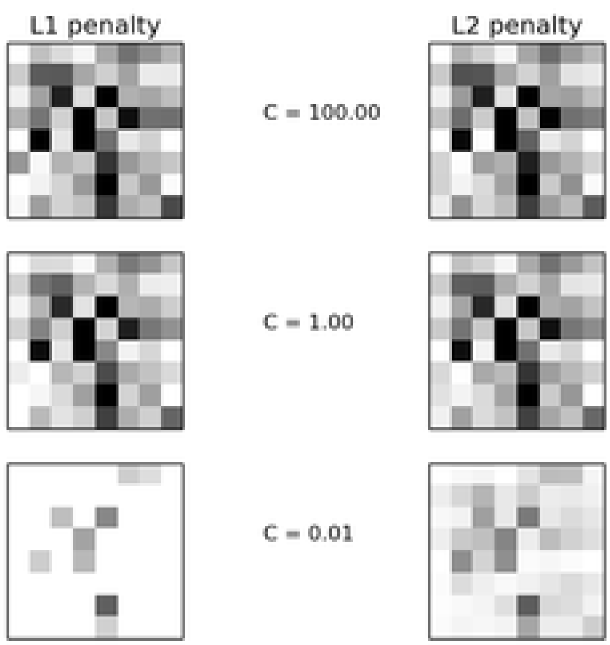


Figure 42.376: *L1 Penalty and Sparsity in Logistic Regression*

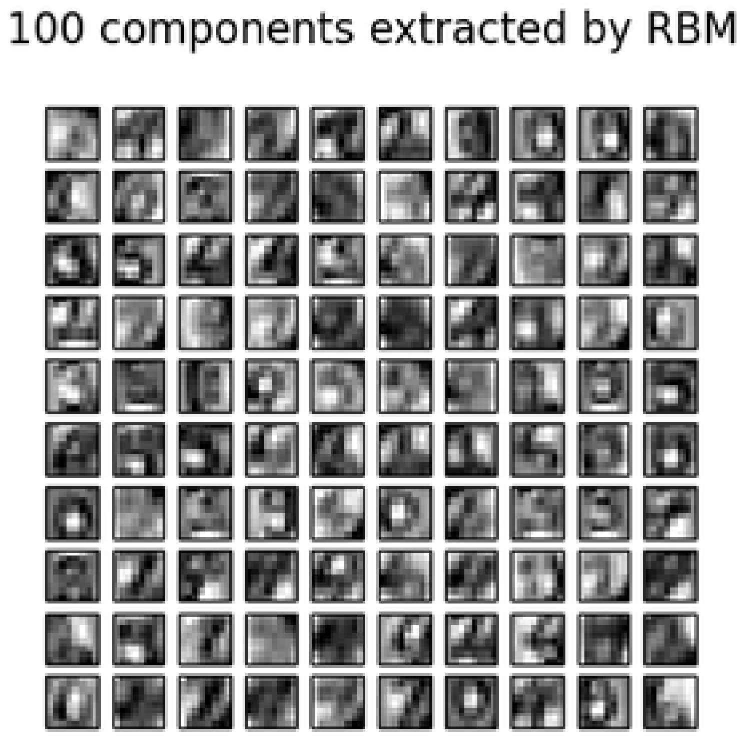


Figure 42.377: *Restricted Boltzmann Machine features for digit classification*

For the grid of Cs values (that are set by default to be ten values in a logarithmic scale between 1e-4 and 1e4), the best hyperparameter is selected by the cross-validator StratifiedKFold, but it can be changed using the cv parameter. In the case of newton-cg and lbfgs solvers, we warm start along the path i.e guess the initial coefficients of the present fit to be the coefficients got after convergence in the previous fit, so in general it is supposed to be faster.

For a multiclass problem, the hyperparameters for each class are computed using the best scores got by doing a one-vs-rest in parallel across all folds and classes. Hence this is not the true multinomial loss.

ParametersCs : list of floats | int

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 the decision function.

class_weight : {dict, 'auto'}, optional

Over-/undersamples the samples of each class according to the given weights. If not given, all classes are supposed to have weight one. The 'auto' mode selects weights inversely proportional to class frequencies in the training set.

cv : integer or cross-validation generator

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.cross_validation` module for the list of possible cross-validation objects.

penalty : str, '11' or 'l2'

Used to specify the norm used in the penalization. The newton-cg and lbfgs solvers support only l2 penalties.

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.

scoring : callabale

Scoring function to use as cross-validation criteria. For a list of scoring functions that can be used, look at `sklearn.metrics`. The default scoring option used is accuracy_score.

solver : {'newton-cg', 'lbfgs', 'liblinear'}

Algorithm to use in the optimization problem.

tol : float, optional

Tolerance for stopping criteria.

max_iter : int, optional

Maximum number of iterations of the optimization algorithm.

class_weight : {dict, 'auto'}, optional

Over-/undersamples the samples of each class according to the given weights. If not given, all classes are supposed to have weight one. The 'auto' mode selects weights inversely proportional to class frequencies in the training set.

n_jobs : int, optional

Number of CPU cores used during the cross-validation loop. If given a value of -1, all cores are used.

verbose : int

For the liblinear and lbfgs solvers set verbose to any positive number for verbosity.

refit : bool

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.

multi_class : str, {'ovr', 'multinomial'}

Multiclass option can be either 'ovr' or 'multinomial'. If the option chosen is 'ovr', then a binary problem is fit for each label. Else the loss minimised is the multinomial loss fit across the entire probability distribution. Works only for the 'lbfgs' solver.

intercept_scaling : float, default 1.

Useful only if solver is liblinear. This parameter is 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 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.

Attributescoef_ : array, 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. *coef_* is readonly property derived from *raw_coef_* that follows the internal memory layout of liblinear.

intercept_ : array, shape (1,) or (n_classes,)

Intercept (a.k.a. bias) added to the decision function. It is available only when parameter intercept is set to True and is of shape(1,) when the problem is binary.

Cs_ : array

Array of C i.e. inverse of regularization parameter values used for cross-validation.

coefs_paths_ : array, shape `(n_folds, len(Cs_), n_features)` or `(n_folds, len(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, len(Cs_), n_features)` or `(n_folds, len(Cs_), n_features + 1)` depending on whether the intercept is fit or not.

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, len(Cs))

C_ : array, 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.

See also:

`LogisticRegression`

Methods

`decision_function`(X)	Predict confidence scores for samples.
`densify`()	Convert coefficient matrix to dense array format.
`fit`(X, y)	Fit the model according to the given training data.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict class labels for samples in X.
`predict_log_proba`(X)	Log of probability estimates.
`predict_proba`(X)	Probability estimates.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.
`sparsify`()	Convert coefficient matrix to sparse format.

Continued on next page

Table 42.125 – continued from previous page

transform(X[, threshold])	Reduce X to its most important features.

static __init__ (*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=1*, *verbose=0*, *refit=True*, *intercept_scaling=1.0*, *multi_class='ovr'*)

decision_function (*X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returnsarray, 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 ()

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.

Returnsself: estimator :

static fit (*X*, *y*)

Fit the model according to the given training data.

ParametersX : {array-like, sparse matrix}, 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.

Returnsself : object

Returns self.

fit_transform (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict class labels for samples in X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Samples.

ReturnsC : array, shape = [n_samples]

Predicted class label per sample.

predict_log_proba (*X*)

Log of probability estimates.

The returned estimates for all classes are ordered by the label of classes.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsT : 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_`.

predict_proba (*X*)

Probability estimates.

The returned estimates for all classes are ordered by the label of classes.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsT : array-like, 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 (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

sparsify()

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.

Returnsself: 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.

transform(*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

42.19.15 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 earch row.

Parametersalpha : 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 (e.g. data is expected to be already centered).

normalize : boolean, optional, default False

If `True`, the regressors X will be normalized before regression.

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.

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

random_state : int, RandomState instance, or None (default)

The seed of the pseudo random number generator that selects a random feature to update. Useful only when selection is set to 'random'.

Attributescoef_ : array, shape = (n_tasks, n_features)

parameter vector (W in the cost function formula)

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:

`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, copy_X=True, fit_intercept=True, max_iter=1000,
        normalize=False, random_state=None, selection='cyclic', tol=0.0001,
        warm_start=False)
>>> print(clf.coef_)
[[ 0.89393398  0.        ]
 [ 0.89393398  0.        ]]
>>> print(clf.intercept_)
[ 0.10606602  0.10606602]
```

Methods

decision_function(X)	Decision function of the linear model
fit(X, y)	Fit MultiTaskLasso model with coordinate descent
get_params([deep])	Get parameters for this estimator.
path(X, y[, l1_ratio, eps, n_alphas, ...])	Compute elastic net path with coordinate descent
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*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'*)

decision_function(*X*)

Decision function of the linear model

ParametersX : numpy array or scipy.sparse matrix of shape (n_samples, n_features)

ReturnsT : array, shape = (n_samples,)

The predicted decision function

fit(*X*, *y*)

Fit MultiTaskLasso model with coordinate descent

ParametersX : ndarray, shape = (n_samples, n_features)

Data

y : ndarray, shape = (n_samples, n_tasks)

Target

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 (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : 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*, ***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.

ParametersX : {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

float 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 : 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.

params : kwargs

keyword arguments passed to the coordinate descent solver.

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.

Returnsalphas : 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

See examples/plot_lasso_coordinate_descent_path.py for an example.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.MultiTaskLasso`

42.19.16 `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.

Parametersalpha : float, optional

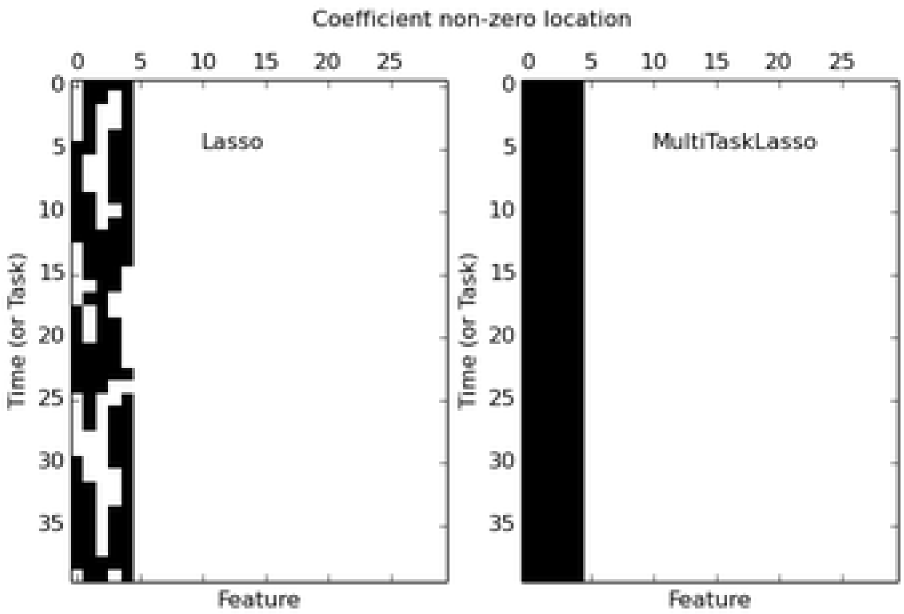


Figure 42.378: *Joint feature selection with multi-task Lasso*

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 = 0 the penalty is an L1/L2 penalty. For l1_ratio = 1 it is an L1 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 (e.g. data is expected to be already centered).

normalize : boolean, optional, default False

If `True`, the regressors X will be normalized before regression.

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.

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.

random_state : int, RandomState instance, or None (default)

The seed of the pseudo random number generator that selects a random feature to update. Useful only when selection is set to 'random'.

Attributesintercept_ : 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

n_iter_ : int

number of iterations run by the coordinate descent solver to reach the specified tolerance.

See also:

`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, copy_X=True, fit_intercept=True,
        l1_ratio=0.5, max_iter=1000, normalize=False, random_state=None,
        selection='cyclic', tol=0.0001, warm_start=False)
>>> print(clf.coef_)
[[ 0.45663524  0.45612256]
 [ 0.45663524  0.45612256]]
>>> print(clf.intercept_)
[ 0.0872422  0.0872422]
```

Methods

`decision_function`(X)	Decision function of the linear model
`fit`(X, y)	Fit MultiTaskLasso model with coordinate descent
`get_params`([deep])	Get parameters for this estimator.

Continued on next page

Table 42.127 – continued from previous page

path(X, y[, l1_ratio, eps, n_alphas, ...])	Compute elastic net path with coordinate descent
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static `__init__`(*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'*)

`decision_function`(*X*)

Decision function of the linear model

ParametersX : numpy array or scipy.sparse matrix of shape (n_samples, n_features)

ReturnsT : array, shape = (n_samples,)

The predicted decision function

static `fit`(*X, y*)

Fit MultiTaskLasso model with coordinate descent

ParametersX : ndarray, shape = (n_samples, n_features)

Data

y : ndarray, shape = (n_samples, n_tasks)

Target

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`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : 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, **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.

ParametersX : {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

float 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 : 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.

params : kwargs

keyword arguments passed to the coordinate descent solver.

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.

Returnsalphas : 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

See examples/plot_lasso_coordinate_descent_path.py for an example.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

42.19.17 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=1*, *random_state=None*, *selection='cyclic'*)

Multi-task L1/L2 Lasso with built-in cross-validation.

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.

Parameterseps : float, optional

Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

alphas : array-like, optional

List of alphas where to compute the models. If not provided, set automaticlly.

n_alphas : int, optional

Number of alphas along the regularization path

fit_intercept : boolean

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 : boolean, optional, default False

If `True`, the regressors X will be normalized before regression.

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`.

cv : integer or cross-validation generator, optional

If an integer is passed, it is the number of fold (default 3). Specific cross-validation objects can be passed, see the `sklearn.cross_validation` module for the list of possible objects.

verbose : bool or integer

Amount of verbosity.

n_jobs : integer, optional

Number of CPUs to use during the cross validation. If -1, use all the CPUs. Note that this is used only if multiple values for l1_ratio are given.

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.

random_state : int, RandomState instance, or None (default)

The seed of the pseudo random number generator that selects a random feature to update. Useful only when selection is set to 'random'.

Attributesintercept_ : array, shape (n_tasks,)

Independent term in decision function.

coef_ : array, shape (n_tasks, n_features)

Parameter vector (W in the cost function formula).

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.

Methods

`decision_function`(X)	Decision function of the linear model.
`fit`(X, y)	Fit linear model with coordinate descent
`get_params`([deep])	Get parameters for this estimator.
`path`	

Continued on next page

Table 42.128 – continued from previous page

predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__ (*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=1*, *random_state=None*, *selection='cyclic'*)

decision_function (*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

fit (*X*, *y*)

Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

ParametersX : {array-like}, shape (n_samples, n_features)

Training data. Pass directly as float64, 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 (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

42.19.18 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=1*, *random_state=None*, *selection='cyclic'*)

Multi-task L1/L2 ElasticNet with built-in cross-validation.

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.

Parameterseps : float, optional

Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

alphas : array-like, optional

List of alphas where to compute the models. If not provided, set automatically.

n_alphas : int, optional

Number of alphas along the regularization path

l1_ratio : float or array of floats

The ElasticNet mixing parameter, with 0 < l1_ratio <= 1. For l1_ratio = 0 the penalty is an L1/L2 penalty. For l1_ratio = 1 it is an L1 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 (e.g. data is expected to be already centered).

normalize : boolean, optional, default False

If `True`, the regressors X will be normalized before regression.

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`.

cv : integer or cross-validation generator, optional

If an integer is passed, it is the number of fold (default 3). Specific cross-validation objects can be passed, see the `sklearn.cross_validation` module for the list of possible objects.

verbose : bool or integer

Amount of verbosity.

n_jobs : integer, optional

Number of CPUs to use during the cross validation. If `-1`, use all the CPUs. Note that this is used only if multiple values for l1_ratio are given.

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.

random_state : int, RandomState instance, or None (default)

The seed of the pseudo random number generator that selects a random feature to update. Useful only when selection is set to 'random'.

Attributesintercept_ : array, shape (n_tasks,)

Independent term in decision function.

coef_ : array, shape (n_tasks, n_features)

Parameter vector (W in the cost function formula).

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()
>>> clf.fit([[0,0], [1, 1], [2, 2]],
...         [[0, 0], [1, 1], [2, 2]])
...
MultiTaskElasticNetCV(alphas=None, copy_X=True, cv=None, eps=0.001,
       fit_intercept=True, l1_ratio=0.5, max_iter=1000, n_alphas=100,
       n_jobs=1, normalize=False, random_state=None, selection='cyclic',
       tol=0.0001, verbose=0)
>>> print(clf.coef_)
[[ 0.52875032  0.46958558]
 [ 0.52875032  0.46958558]]
>>> print(clf.intercept_)
[ 0.00166409  0.00166409]
```

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y)	Fit linear model with coordinate descent
get_params([deep])	Get parameters for this estimator.
path	
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*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=1*, *random_state=None*, *selection='cyclic'*)

decision_function(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

fit (*X*, *y*)

Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

ParametersX : {array-like}, shape (n_samples, n_features)

Training data. Pass directly as float64, 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 (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

42.19.19 `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)

Parametersn_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 (e.g. data is expected to be already centered).

normalize : boolean, optional

If False, the regressors X are assumed to be already normalized.

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.

Attributescoef_ : array, shape (n_features,) or (n_features, n_targets)

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`

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. http://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y)	Fit the model using X, y as training data.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*n_nonzero_coefs=None*, *tol=None*, *fit_intercept=True*, *normalize=True*, *precompute='auto'*)

decision_function(*X*)
Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

static fit(*X*, *y*)
Fit the model using X, y as training data.

ParametersX : array-like, shape (n_samples, n_features)

Training data.

y : array-like, shape (n_samples,) or (n_samples, n_targets)

Target values.

Returnsself : object

returns an instance of self.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*X*)
Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score(*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.OrthogonalMatchingPursuit`

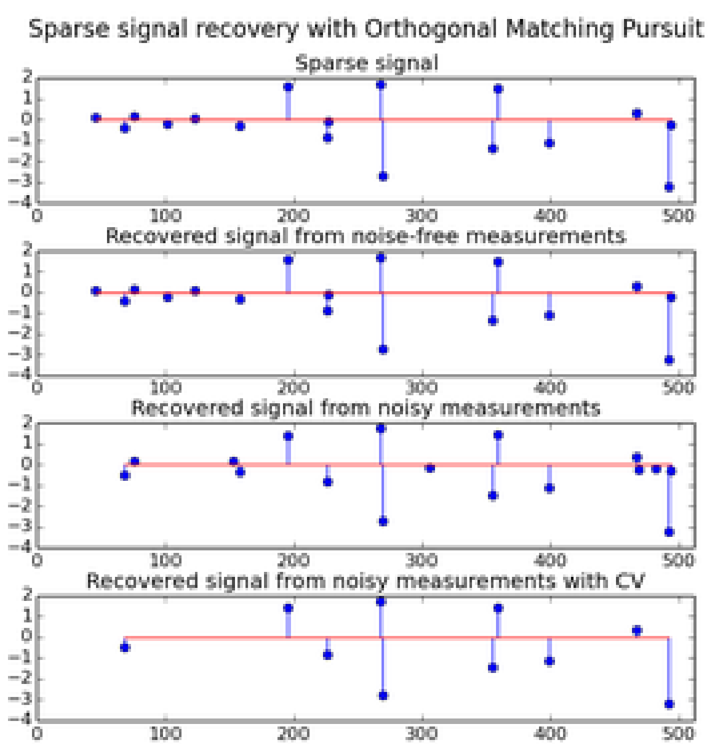


Figure 42.379: *Orthogonal Matching Pursuit*

42.19.20 sklearn.linear_model.OrthogonalMatchingPursuitCV

class sklearn.linear_model.**OrthogonalMatchingPursuitCV**(*copy=True*, *fit_intercept=True*, *normalize=True*, *max_iter=None*, *cv=None*, *n_jobs=1*, *verbose=False*)

Cross-validated Orthogonal Matching Pursuit model (OMP)

Parameterscopy : 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 (e.g. data is expected to be already centered).

normalize : boolean, optional

If False, the regressors X are assumed to be already normalized.

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 : cross-validation generator, optional

see `sklearn.cross_validation`. If `None` is passed, default to a 5-fold strategy

n_jobs : integer, optional

Number of CPUs to use during the cross validation. If `-1`, use all the CPUs

verbose : boolean or integer, optional

Sets the verbosity amount

Attributesintercept_ : float or array, shape (n_targets,)

Independent term in decision function.

coef_ : array, shape (n_features,) or (n_features, n_targets)

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`

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y)	Fit the model using X, y as training data.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static `__init__`(*copy=True*, *fit_intercept=True*, *normalize=True*, *max_iter=None*, *cv=None*, *n_jobs=1*, *verbose=False*)

`decision_function`(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

static `fit`(*X*, *y*)

Fit the model using X, y as training data.

ParametersX : array-like, shape [n_samples, n_features]

Training data.

y : array-like, shape [n_samples]

Target values.

Returnsself : object

returns an instance of self.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

`predict`(*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

`score`(*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.OrthogonalMatchingPursuitCV`

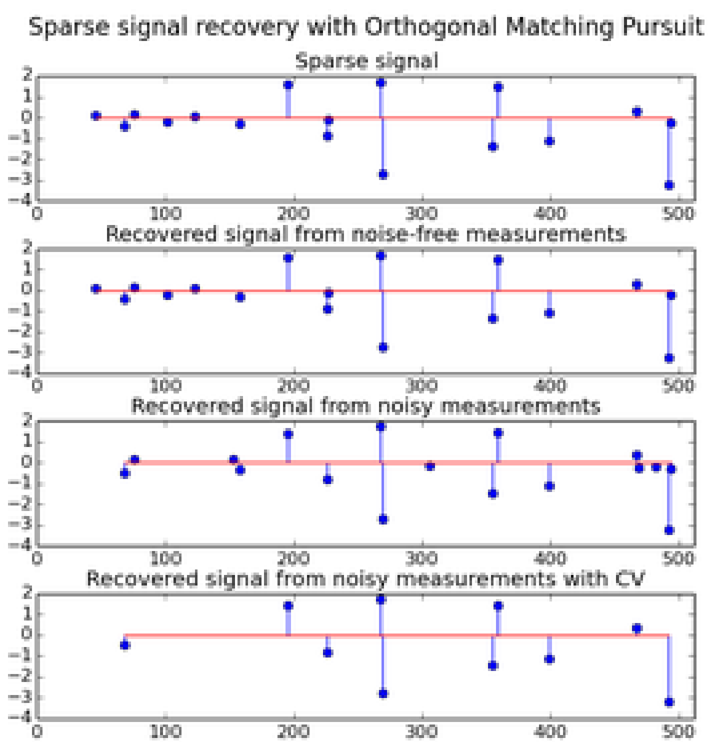


Figure 42.380: *Orthogonal Matching Pursuit*

42.19.21 sklearn.linear_model.PassiveAggressiveClassifier

class sklearn.linear_model.**PassiveAggressiveClassifier**(*C=1.0*, *fit_intercept=True*, *n_iter=5*, *shuffle=True*, *verbose=0*, *loss='hinge'*, *n_jobs=1*, *random_state=None*, *warm_start=False*)

Passive Aggressive Classifier

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.

n_iter : int, optional

The number of passes over the training data (aka epochs). Defaults to 5.

shuffle : bool, default=True

Whether or not the training data should be shuffled after each epoch.

random_state : int seed, RandomState instance, or None (default)

The seed of the pseudo random number generator to use when shuffling the data.

verbose : integer, optional

The verbosity level

n_jobs : integer, optional

The number of CPUs to use to do the OVA (One Versus All, for multi-class problems) computation. -1 means 'all CPUs'. Defaults to 1.

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.

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.

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.

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)

Methods

decision_function(X)	Predict confidence scores for samples.
densify()	Convert coefficient matrix to dense array format.
fit(X, y[, coef_init, intercept_init])	Fit linear model with Passive Aggressive algorithm.
get_params([deep])	Get parameters for this estimator.
partial_fit(X, y[, classes])	Fit linear model with Passive Aggressive algorithm.
predict(X)	Predict class labels for samples in X.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(*args, **kwargs)	
sparsify()	Convert coefficient matrix to sparse format.

static __init__(*C=1.0*, *fit_intercept=True*, *n_iter=5*, *shuffle=True*, *verbose=0*, *loss='hinge'*, *n_jobs=1*, *random_state=None*, *warm_start=False*)

decision_function(*X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returnsarray, 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()

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.

Returnsself: estimator :

static fit(*X*, *y*, *coef_init=None*, *intercept_init=None*)

Fit linear model with Passive Aggressive algorithm.

ParametersX : {array-like, sparse matrix}, 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.

Returnsself : returns an instance of self.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **partial_fit** (*X*, *y*, *classes=None*)

Fit linear model with Passive Aggressive algorithm.

ParametersX : {array-like, sparse matrix}, 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*.

Returnsself : returns an instance of self.

predict (*X*)

Predict class labels for samples in X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Samples.

ReturnsC : array, shape = [n_samples]

Predicted class label per sample.

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

sparsify ()

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.

Returnsself: 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`

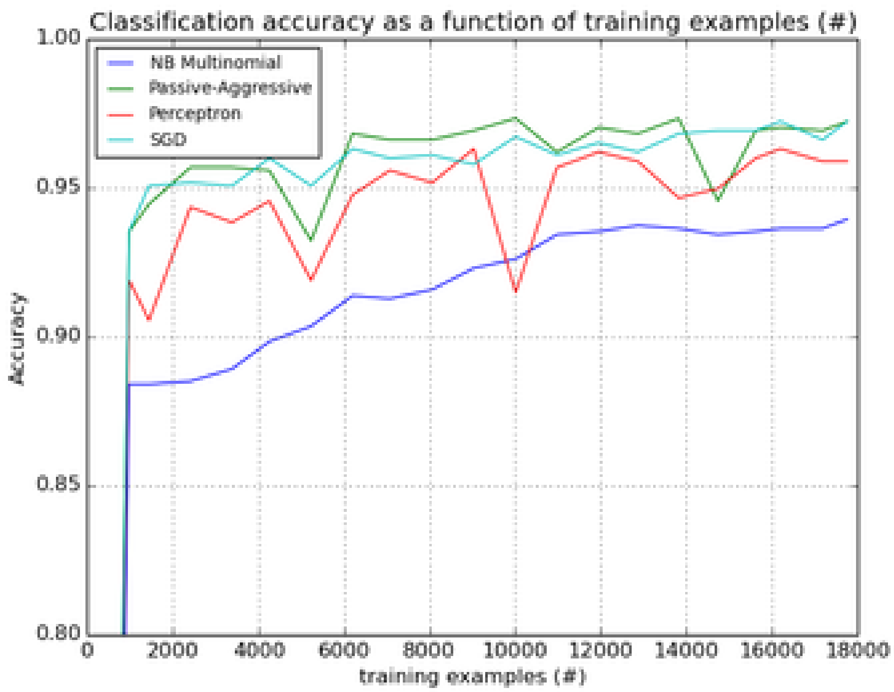


Figure 42.381: *Out-of-core classification of text documents*

42.19.22 `sklearn.linear_model`.PassiveAggressiveRegressor

class `sklearn.linear_model.`**`PassiveAggressiveRegressor`**(*C=1.0*, *fit_intercept=True*, *n_iter=5*, *shuffle=True*, *verbose=0*, *loss='epsilon_insensitive'*, *epsilon=0.1*, *random_state=None*, *class_weight=None*, *warm_start=False*)

Passive Aggressive Regressor

ParametersC : float

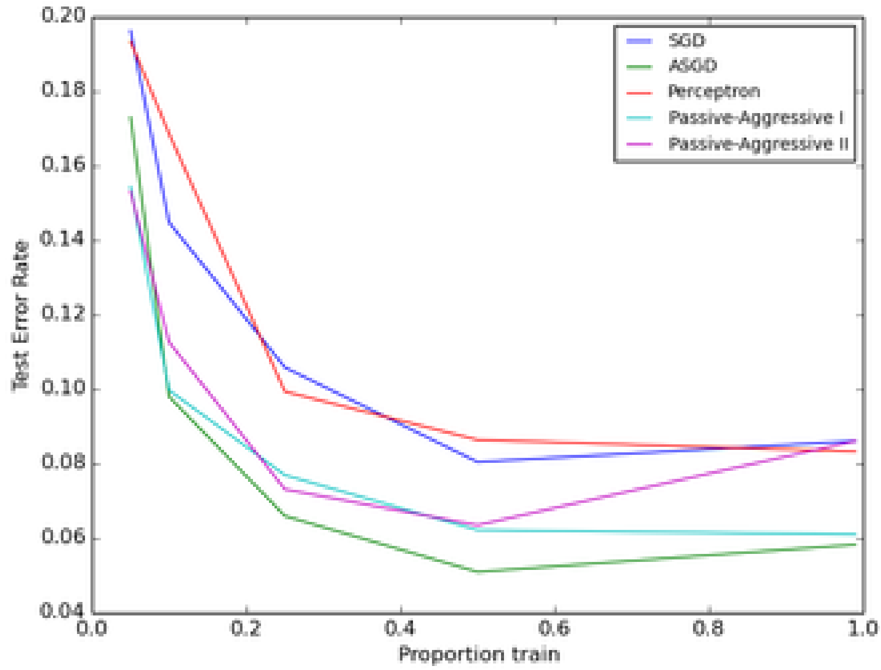


Figure 42.382: *Comparing various online solvers*

Figure 42.383: *Classification of text documents using sparse features*

Maximum step size (regularization). Defaults to 1.0.

epsilon : float

If the difference between the current prediction and the correct label is below this threshold, the model is not updated.

fit_intercept : bool

Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

n_iter : int, optional

The number of passes over the training data (aka epochs). Defaults to 5.

shuffle : bool, default=True

Whether or not the training data should be shuffled after each epoch.

random_state : int seed, RandomState instance, or None (default)

The seed of the pseudo random number generator to use when shuffling the data.

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.

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.

Attributescoef_ : 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.

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)

Methods

`decision_function`(X)	Predict using the linear model
`densify`()	Convert coefficient matrix to dense array format.
`fit`(X, y[, coef_init, intercept_init])	Fit linear model with Passive Aggressive algorithm.
`get_params`([deep])	Get parameters for this estimator.
`partial_fit`(X, y)	Fit linear model with Passive Aggressive algorithm.

Continued on next page

Table 42.133 – continued from previous page

predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(*args, **kwargs)	
sparsify()	Convert coefficient matrix to sparse format.

static `__init__`(*C=1.0, fit_intercept=True, n_iter=5, shuffle=True, verbose=0, loss='epsilon_insensitive', epsilon=0.1, random_state=None, class_weight=None, warm_start=False*)

`decision_function`(*X*)
Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

Returnsarray, shape (n_samples,) :

Predicted target values per element in X.

`densify`()
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.

Returnsself: estimator :

static `fit`(*X, y, coef_init=None, intercept_init=None*)
Fit linear model with Passive Aggressive algorithm.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Training data

y : numpy array of shape [n_samples]

Target values

coef_init : array, shape = [n_features]

The initial coefficients to warm-start the optimization.

intercept_init : array, shape = [1]

The initial intercept to warm-start the optimization.

Returnsself : returns an instance of self.

`get_params`(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static `partial_fit`(*X, y*)
Fit linear model with Passive Aggressive algorithm.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Subset of training data

y : numpy array of shape [n_samples]

Subset of target values

Returnsself : returns an instance of self.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

Returnsarray, shape (n_samples,) :

Predicted target values per element in X.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

sparsify ()

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.

Returnsself: 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.

42.19.23 sklearn.linear_model.Perceptron

class sklearn.linear_model.**Perceptron**(*penalty=None*, *alpha=0.0001*, *fit_intercept=True*, *n_iter=5*, *shuffle=True*, *verbose=0*, *eta0=1.0*, *n_jobs=1*, *random_state=0*, *class_weight=None*, *warm_start=False*)

Perceptron

Parameterspenalty : None, 'l2' or 'l1' or 'elasticnet'

The penalty (aka regularization term) to be used. Defaults to None.

alpha : float

Constant that multiplies the regularization term if regularization is used. Defaults to 0.0001

fit_intercept : bool

Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

n_iter : int, optional

The number of passes over the training data (aka epochs). Defaults to 5.

shuffle : bool, optional, default True

Whether or not the training data should be shuffled after each epoch.

random_state : int seed, RandomState instance, or None (default)

The seed of the pseudo random number generator to use when shuffling the data.

verbose : integer, optional

The verbosity level

n_jobs : integer, optional

The number of CPUs to use to do the OVA (One Versus All, for multi-class problems) computation. -1 means 'all CPUs'. Defaults to 1.

eta0 : double

Constant by which the updates are multiplied. Defaults to 1.

class_weight : dict, {class_label: weight} or "auto" 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 "auto" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

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.

Attributescoef_ : 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.

See also:

`SGDClassifier`

Notes

Perceptron and *SGDClassifier* share the same underlying implementation. In fact, *Perceptron()* is equivalent to *SGDClassifier(loss="perceptron", eta0=1, learning_rate="constant", penalty=None)*.

References

http://en.wikipedia.org/wiki/Perceptron and references therein.

Methods

`decision_function`(X)	Predict confidence scores for samples.
`densify`()	Convert coefficient matrix to dense array format.
`fit`(X, y[, coef_init, intercept_init, ...])	Fit linear model with Stochastic Gradient Descent.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`partial_fit`(X, y[, classes, sample_weight])	Fit linear model with Stochastic Gradient Descent.
`predict`(X)	Predict class labels for samples in X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(*args, **kwargs)	
`sparsify`()	Convert coefficient matrix to sparse format.
`transform`(X[, threshold])	Reduce X to its most important features.

static `__init__`(*penalty=None*, *alpha=0.0001*, *fit_intercept=True*, *n_iter=5*, *shuffle=True*, *verbose=0*, *eta0=1.0*, *n_jobs=1*, *random_state=0*, *class_weight=None*, *warm_start=False*)

`decision_function`(*X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returnsarray, 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`()

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.

Returnsself: estimator :

fit (*X*, *y*, *coef_init=None*, *intercept_init=None*, *class_weight=None*, *sample_weight=None*)
Fit linear model with Stochastic Gradient Descent.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

Training data

y : numpy array, 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.

sample_weight : array-like, shape (n_samples,), optional

Weights applied to individual samples. If not provided, uniform weights are assumed. These weights will be multiplied with class_weight (passed through the contructor) if class_weight is specified

Returnsself : returns an instance of self.

fit_transform (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

partial_fit (*X*, *y*, *classes=None*, *sample_weight=None*)
Fit linear model with Stochastic Gradient Descent.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

Subset of the training data

y : numpy array, 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*.

sample_weight : array-like, shape (n_samples,), optional

Weights applied to individual samples. If not provided, uniform weights are assumed.

Returnsself : returns an instance of self.

predict (*X*)

Predict class labels for samples in X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Samples.

ReturnsC : array, shape = [n_samples]

Predicted class label per sample.

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

sparsify ()

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.

Returnsself: 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.

transform (*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Examples using `sklearn.linear_model.Perceptron`

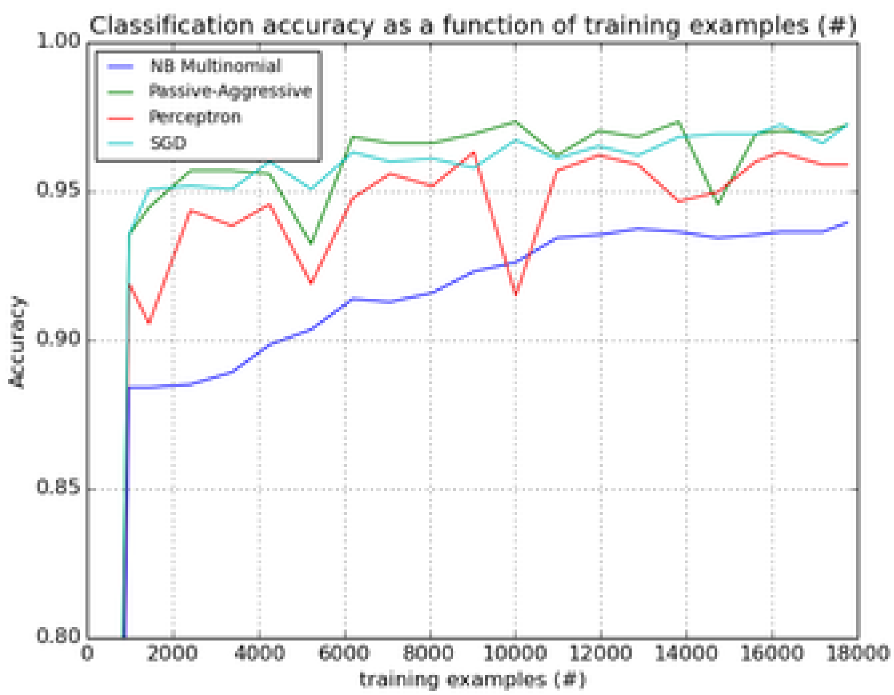


Figure 42.384: *Out-of-core classification of text documents*

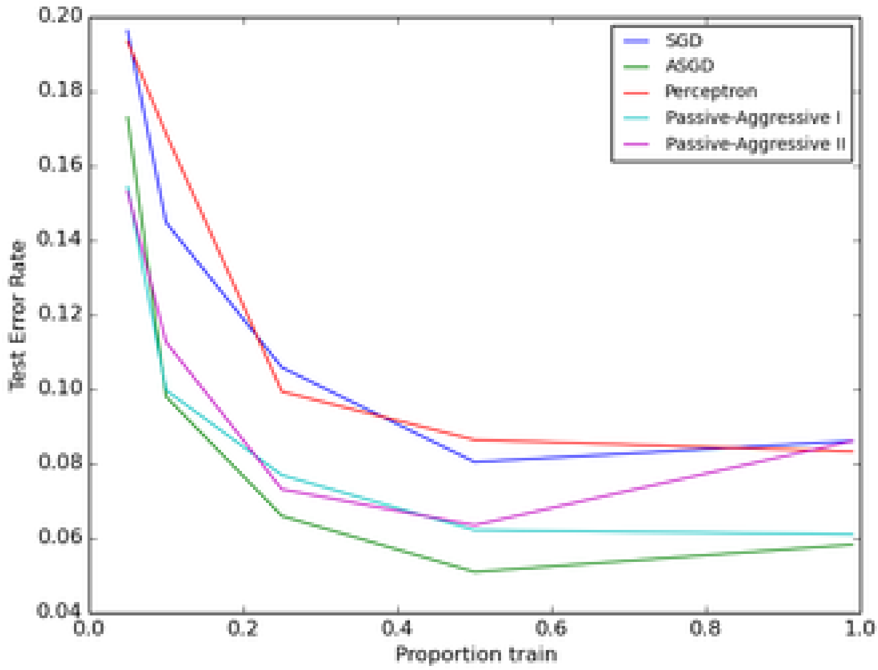


Figure 42.385: *Comparing various online solvers*

Figure 42.386: *Classification of text documents using sparse features*

42.19.24 sklearn.linear_model.RandomizedLasso

class `sklearn.linear_model.`**`RandomizedLasso`**(*alpha='aic'*, *scaling=0.5*, *sample_fraction=0.75*, *n_resampling=200*, *selection_threshold=0.25*, *fit_intercept=True*, *verbose=False*, *normalize=True*, *precompute='auto'*, *max_iter=500*, *eps=2.220446049250313e-16*, *random_state=None*, *n_jobs=1*, *pre_dispatch='3*n_jobs'*, *memory=Memory(cachedir=None)*)

Randomized Lasso.

Randomized Lasso works by resampling the train data and computing a Lasso on each resampling. In short, the features selected more often are good features. It is also known as stability selection.

Parametersalpha : float, 'aic', or 'bic', optional

The regularization parameter alpha parameter in the Lasso. Warning: this is not the alpha parameter in the stability selection article which is scaling.

scaling : float, optional

The alpha parameter in the stability selection article used to randomly scale the features. Should be between 0 and 1.

sample_fraction : float, optional

The fraction of samples to be used in each randomized design. Should be between 0 and 1. If 1, all samples are used.

n_resampling : int, optional

Number of randomized models.

selection_threshold: float, optional :

The score above which features should be selected.

fit_intercept : boolean, optional

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).

verbose : boolean or integer, optional

Sets the verbosity amount

normalize : boolean, optional, default True

If True, the regressors X will be normalized before regression.

precompute : True | False | '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 : integer, optional

Maximum number of iterations to perform in the Lars algorithm.

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.

n_jobs : integer, optional

Number of CPUs to use during the resampling. If '-1', use all the CPUs

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*.

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'

memory : Instance of joblib.Memory or string

Used for internal caching. By default, no caching is done. If a string is given, it is the path to the caching directory.

Attributesscores_ : array, shape = [n_features]

Feature scores between 0 and 1.

all_scores_ : array, shape = [n_features, n_reg_parameter]

Feature scores between 0 and 1 for all values of the regularization parameter. The reference article suggests `scores_` is the max of `all_scores_`.

See also:

`RandomizedLogisticRegression`, `LogisticRegression`

Notes

See examples/linear_model/plot_sparse_recovery.py for an example.

References

Stability selection Nicolai Meinshausen, Peter Buhlmann Journal of the Royal Statistical Society: Series B Volume 72, Issue 4, pages 417-473, September 2010 DOI: 10.1111/j.1467-9868.2010.00740.x

Examples

```
>>> from sklearn.linear_model import RandomizedLasso
>>> randomized_lasso = RandomizedLasso()
```

Methods

`fit`(X, y)	Fit the model using X, y as training data.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`get_support`([indices])	Return a mask, or list, of the features/indices selected.
`inverse_transform`(X)	Transform a new matrix using the selected features
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Transform a new matrix using the selected features

static `__init__`(*alpha='aic', scaling=0.5, sample_fraction=0.75, n_resampling=200, selection_threshold=0.25, fit_intercept=True, verbose=False, normalize=True, precompute='auto', max_iter=500, eps=2.2204460492503131e-16, random_state=None, n_jobs=1, pre_dispatch='3*n_jobs', memory=Memory(cachedir=None)*)

`fit`(*X, y*)

Fit the model using X, y as training data.

ParametersX : array-like, sparse matrix shape = [n_samples, n_features]

Training data.

y : array-like, shape = [n_samples]

Target values.

Returnsself : object

Returns an instance of self.

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

`get_support`(*indices=False*)

Return a mask, or list, of the features/indices selected.

`inverse_transform`(*X*)

Transform a new matrix using the selected features

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*)
Transform a new matrix using the selected features

Examples using `sklearn.linear_model.RandomizedLasso`

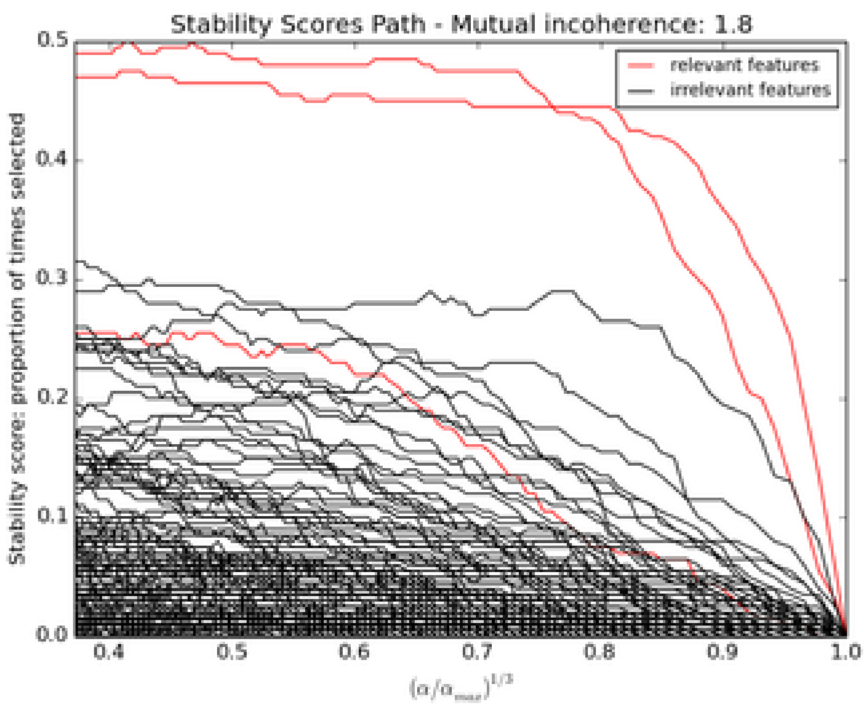


Figure 42.387: *Sparse recovery: feature selection for sparse linear models*

42.19.25 sklearn.linear_model.RandomizedLogisticRegression

class `sklearn.linear_model.`**`RandomizedLogisticRegression`**(*C=1*, *scaling=0.5*, *sample_fraction=0.75*, *n_resampling=200*, *selection_threshold=0.25*, *tol=0.001*, *fit_intercept=True*, *verbose=False*, *normalize=True*, *random_state=None*, *n_jobs=1*, *pre_dispatch='3*n_jobs'*, *memory=Memory(cachedir=None)*)

Randomized Logistic Regression

Randomized Regression works by resampling the train data and computing a LogisticRegression on each resampling. In short, the features selected more often are good features. It is also known as stability selection.

Parameters

C : float, optional, default=1

The regularization parameter C in the LogisticRegression.

scaling : float, optional, default=0.5

The alpha parameter in the stability selection article used to randomly scale the features. Should be between 0 and 1.

sample_fraction : float, optional, default=0.75

The fraction of samples to be used in each randomized design. Should be between 0 and 1. If 1, all samples are used.

n_resampling : int, optional, default=200

Number of randomized models.

selection_threshold : float, optional, default=0.25

The score above which features should be selected.

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 (e.g. data is expected to be already centered).

verbose : boolean or integer, optional

Sets the verbosity amount

normalize : boolean, optional, default=True

If True, the regressors X will be normalized before regression.

tol : float, optional, default=1e-3

tolerance for stopping criteria of LogisticRegression

n_jobs : integer, optional

Number of CPUs to use during the resampling. If '-1', use all the CPUs

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*.

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'

memory : Instance of joblib.Memory or string

Used for internal caching. By default, no caching is done. If a string is given, it is the path to the caching directory.

Attributesscores_ : array, shape = [n_features]

Feature scores between 0 and 1.

all_scores_ : array, shape = [n_features, n_reg_parameter]

Feature scores between 0 and 1 for all values of the regularization parameter. The reference article suggests `scores_` is the max of `all_scores_`.

See also:

`RandomizedLasso`, `Lasso`, `ElasticNet`

Notes

See examples/linear_model/plot_sparse_recovery.py for an example.

References

Stability selection Nicolai Meinshausen, Peter Buhlmann Journal of the Royal Statistical Society: Series B Volume 72, Issue 4, pages 417-473, September 2010 DOI: 10.1111/j.1467-9868.2010.00740.x

Examples

```
>>> from sklearn.linear_model import RandomizedLogisticRegression
>>> randomized_logistic = RandomizedLogisticRegression()
```

Methods

`fit`(X, y)	Fit the model using X, y as training data.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`get_support`([indices])	Return a mask, or list, of the features/indices selected.
`inverse_transform`(X)	Transform a new matrix using the selected features
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Transform a new matrix using the selected features

static __init__(*C=1*, *scaling=0.5*, *sample_fraction=0.75*, *n_resampling=200*, *selection_threshold=0.25*, *tol=0.001*, *fit_intercept=True*, *verbose=False*, *normalize=True*, *random_state=None*, *n_jobs=1*, *pre_dispatch='3*n_jobs'*, *memory=Memory(cachedir=None)*)

fit(*X*, *y*)
Fit the model using X, y as training data.

ParametersX : array-like, sparse matrix shape = [n_samples, n_features]

Training data.

y : array-like, shape = [n_samples]

Target values.

Returnsself : object

Returns an instance of self.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

get_support(*indices=False*)
Return a mask, or list, of the features/indices selected.

inverse_transform(*X*)
Transform a new matrix using the selected features

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*)
Transform a new matrix using the selected features

42.19.26 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*, *stop_n_inliers=inf*, *stop_score=inf*, *stop_probability=0.99*, *residual_metric=None*, *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. More information can be found in the general documentation of linear models.

A detailed description of the algorithm can be found in the documentation of the `linear_model` sub-package.

Parametersbase_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.

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.

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.

residual_metric : callable, optional

Metric to reduce the dimensionality of the residuals to 1 for multi-dimensional target values `y.shape[1] > 1`. By default the sum of absolute differences is used:

```
lambda dy: np.sum(np.abs(dy), axis=1)
```

random_state : integer or numpy.RandomState, optional

The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

Attributesestimator_ : 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`.

References

[R140], [R141], [R142]

Methods

`fit`(X, y)	Fit estimator using RANSAC algorithm.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict using the estimated model.
`score`(X, y)	Returns the score of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*base_estimator=None*, *min_samples=None*, *residual_threshold=None*, *is_data_valid=None*, *is_model_valid=None*, *max_trials=100*, *stop_n_inliers=inf*, *stop_score=inf*, *stop_probability=0.99*, *residual_metric=None*, *random_state=None*)

static `fit`(*X*, *y*)

Fit estimator using RANSAC algorithm.

ParametersX : array-like or sparse matrix, shape [n_samples, n_features]

Training data.

y : array-like, shape = [n_samples] or [n_samples, n_targets]

Target values.

RaisesValueError :

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 (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static predict (*X*)

Predict using the estimated model.

This is a wrapper for *estimator_.predict(X)*.

ParametersX : numpy array of shape [n_samples, n_features]

Returnsy : array, shape = [n_samples] or [n_samples, n_targets]

Returns predicted values.

static score (*X, y*)

Returns the score of the prediction.

This is a wrapper for *estimator_.score(X, y)*.

ParametersX : 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.

Returnsz : float

Score of the prediction.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

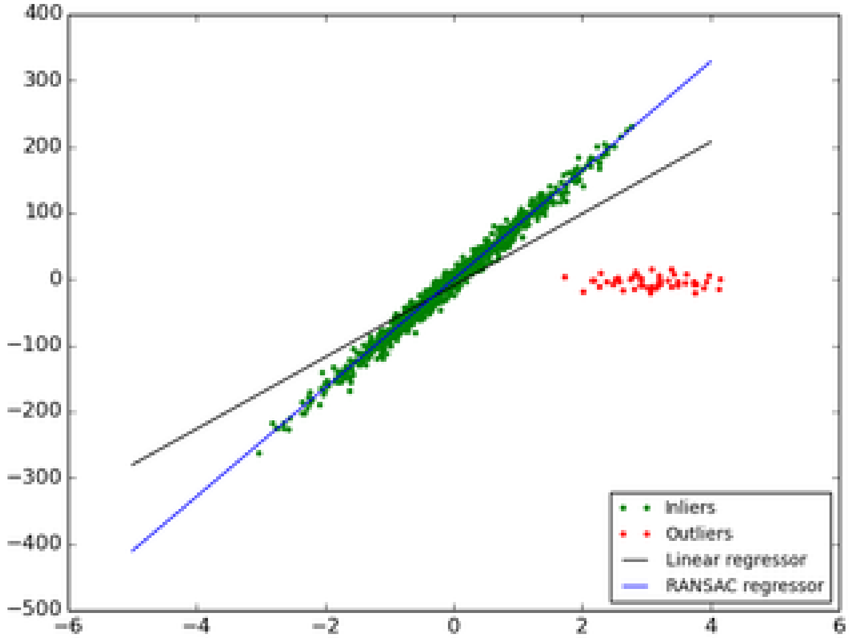


Figure 42.388: *Robust linear model estimation using RANSAC*

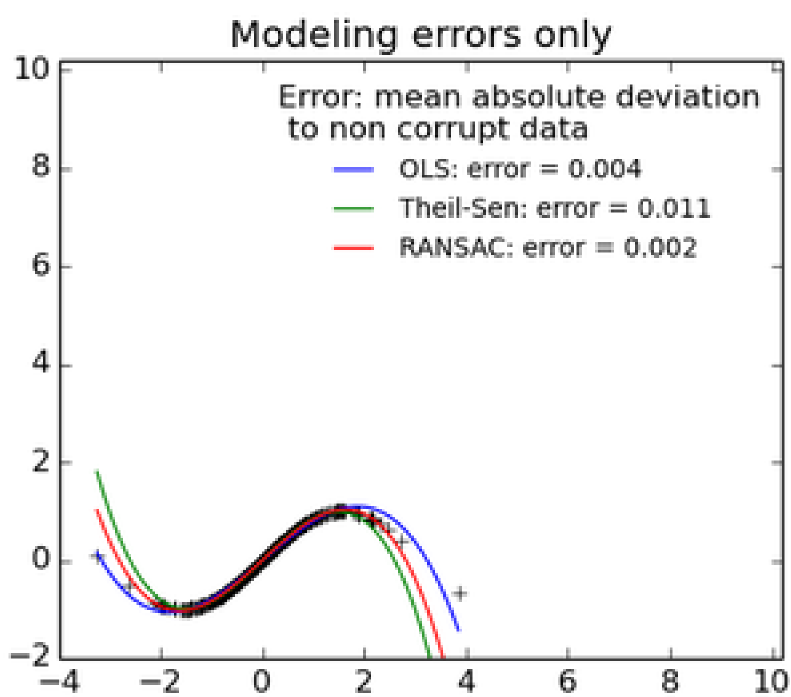


Figure 42.389: *Robust linear estimator fitting*

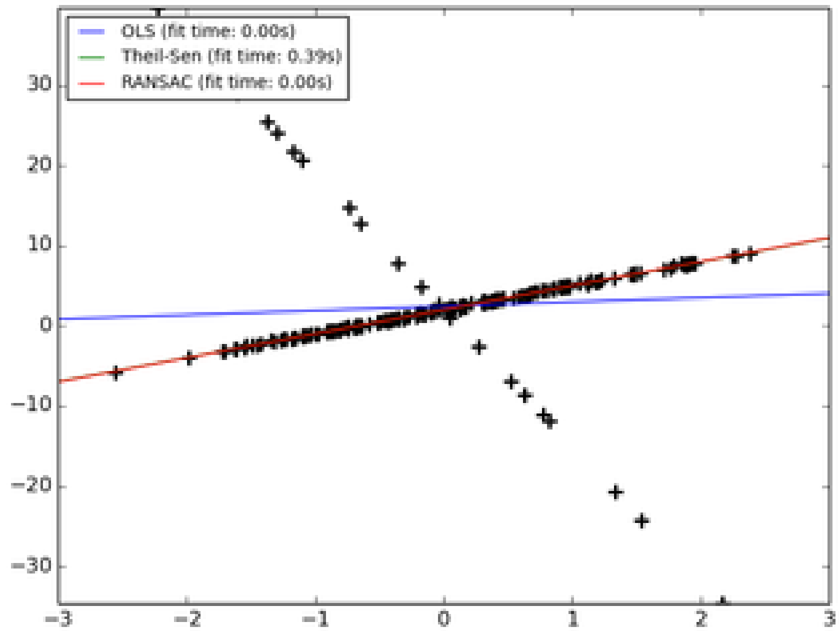


Figure 42.390: *Theil-Sen Regression*

Examples using `sklearn.linear_model.RANSACRegressor`

42.19.27 `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'*)

Linear least squares with l2 regularization.

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]).

Parametersalpha : {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.

copy_X : boolean, optional, default True

If True, X will be copied; else, it may be overwritten.

fit_intercept : boolean

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).

max_iter : int, optional

Maximum number of iterations for conjugate gradient solver. The default value is determined by scipy.sparse.linalg.

normalize : boolean, optional, default False

If True, the regressors X will be normalized before regression.

solver : {'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg'}

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 fatest but may not be available in old scipy versions. It also uses an iterative procedure.

All three solvers support both dense and sparse data.

tol : float

Precision of the solution.

Attributescoef_ : array, shape = [n_features] or [n_targets, n_features]

Weight vector(s).

See also:

RidgeClassifier, RidgeCV, KernelRidge

Examples

```
>>> from sklearn.linear_model import Ridge
>>> 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 = Ridge(alpha=1.0)
>>> clf.fit(X, y)
Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
      normalize=False, solver='auto', tol=0.001)
```

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y[, sample_weight])	Fit Ridge regression model
get_params([deep])	Get parameters for this estimator.

Continued on next page

Table 42.138 – continued from previous page

predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *solver='auto'*)

decision_function(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

static fit(*X*, *y*, *sample_weight=None*)

Fit Ridge regression model

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Training data

y : array-like, shape = [n_samples] or [n_samples, n_targets]

Target values

sample_weight : float or numpy array of shape [n_samples]

Individual weights for each sample

Returnsself : returns an instance of self.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score(*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.Ridge`

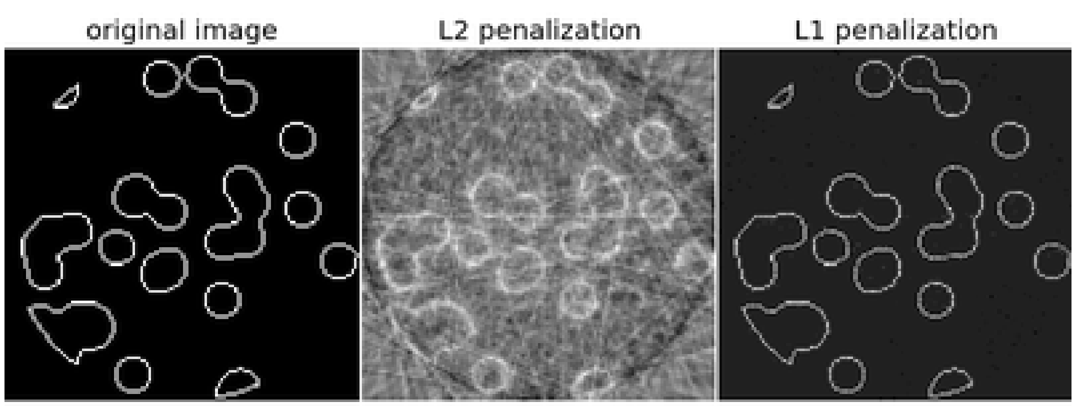


Figure 42.391: *Compressive sensing: tomography reconstruction with L1 prior (Lasso)*

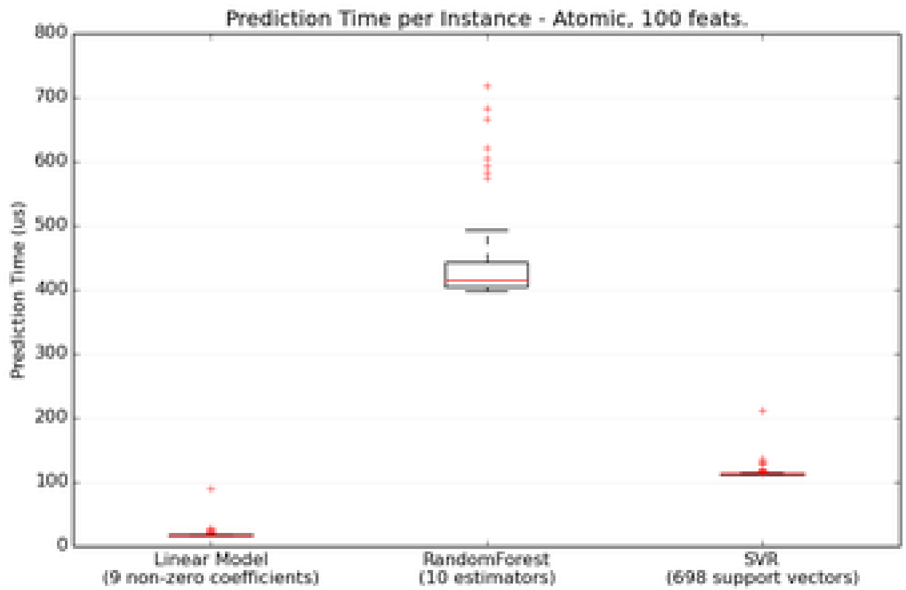


Figure 42.392: *Prediction Latency*

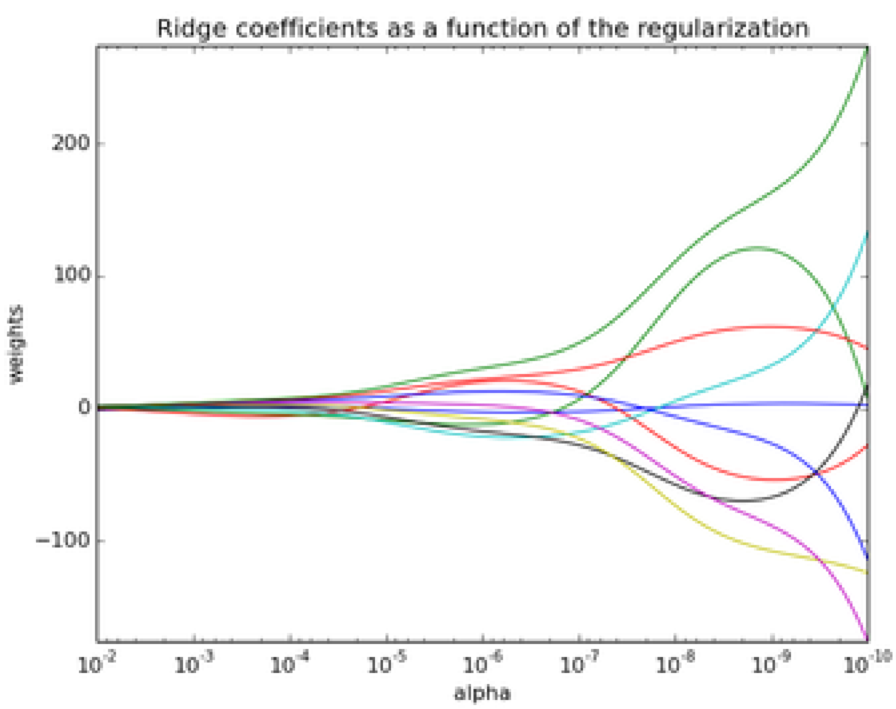


Figure 42.393: *Plot Ridge coefficients as a function of the regularization*

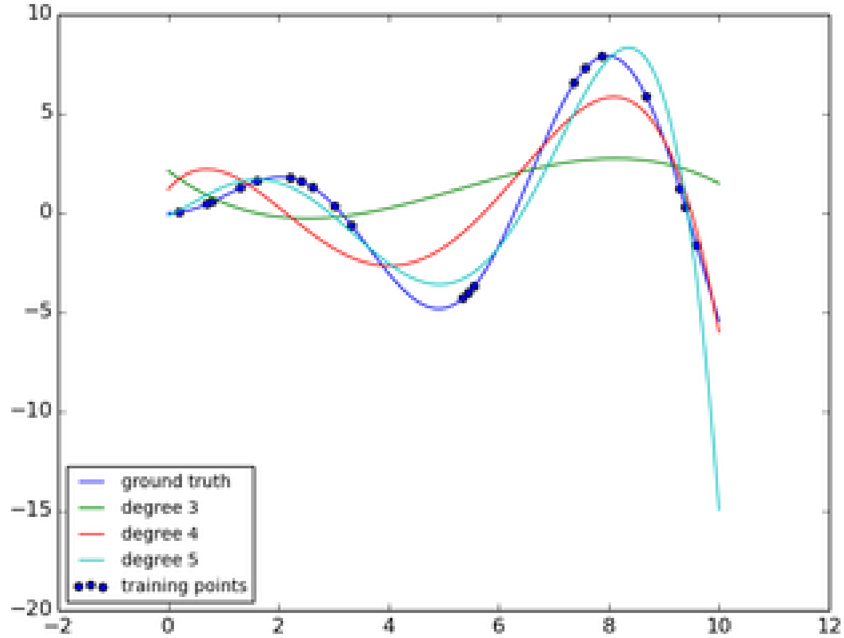


Figure 42.394: *Polynomial interpolation*

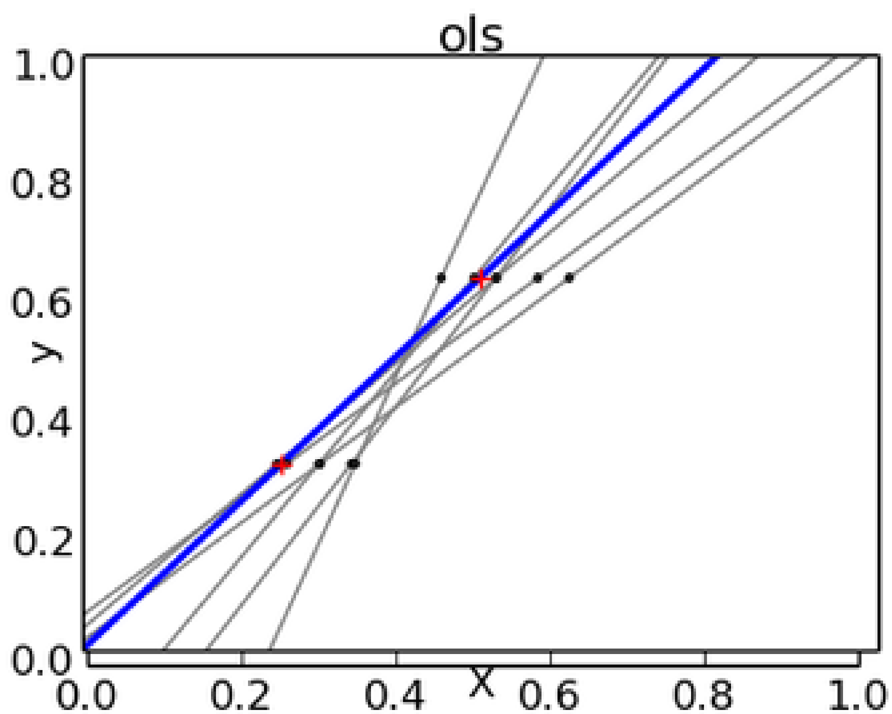


Figure 42.395: *Ordinary Least Squares and Ridge Regression Variance*

42.19.28 `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'*)

Classifier using Ridge regression.

Parametersalpha : float

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.

class_weight : dict, optional

Weights associated with classes in the form `{class_label : weight}`. If not given, all classes are supposed to have weight one.

copy_X : boolean, optional, default True

If True, X will be copied; else, it may be overwritten.

fit_intercept : boolean

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).

max_iter : int, optional

Maximum number of iterations for conjugate gradient solver. The default value is determined by scipy.sparse.linalg.

normalize : boolean, optional, default False

If True, the regressors X will be normalized before regression.

solver : {'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg'}

Solver to use in the computational routines. 'svd' will use a Singular value decomposition to obtain the solution, 'cholesky' will use the standard scipy.linalg.solve function, 'sparse_cg' will use the conjugate gradient solver as found in scipy.sparse.linalg.cg while 'auto' will chose the most appropriate depending on the matrix X. 'lsqr' uses a direct regularized least-squares routine provided by scipy.

tol : float

Precision of the solution.

Attributescoef_ : array, shape = [n_features] or [n_classes, n_features]

Weight vector(s).

See also:

`Ridge`, `RidgeClassifierCV`

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.

Methods

decision_function(X)	Predict confidence scores for samples.
fit(X, y)	Fit Ridge regression model.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict class labels for samples in X.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

static `__init__`(*alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *class_weight=None*, *solver='auto'*)

`decision_function`(*X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returnsarray, 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.

static `fit`(*X*, *y*)

Fit Ridge regression model.

ParametersX : {array-like, sparse matrix}, shape = [n_samples,n_features]

Training data

y : array-like, shape = [n_samples]

Target values

Returnsself : returns an instance of self.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

`predict`(*X*)

Predict class labels for samples in X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Samples.

ReturnsC : array, shape = [n_samples]

Predicted class label per sample.

`score`(*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.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returns self :

Examples using `sklearn.linear_model.RidgeClassifier`

Figure 42.396: *Classification of text documents using sparse features*

42.19.29 `sklearn.linear_model`.RidgeClassifierCV

class `sklearn.linear_model.`**RidgeClassifierCV**(*alphas=array([0.1, 1., 10.]), fit_intercept=True, normalize=False, scoring=None, cv=None, class_weight=None*)

Ridge classifier with built-in cross-validation.

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.

Parameters alphas : numpy array of shape [n_alphas]

Array of alpha values to try. 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.

fit_intercept : boolean

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 : boolean, optional, default False

If True, the regressors X will be normalized before regression.

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)`.

cv : cross-validation generator, optional

If None, Generalized Cross-Validation (efficient Leave-One-Out) will be used.

class_weight : dict, optional

Weights associated with classes in the form `{class_label : weight}`. If not given, all classes are supposed to have weight one.

Attributescv_values_ : array, shape = [n_samples, n_alphas] or shape = [n_samples, n_responses, 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_ : array, shape = [n_features] or [n_targets, n_features]

Weight vector(s).

alpha_ : float

Estimated regularization parameter

See also:

RidgeRidge regression

RidgeClassifierRidge classifier

RidgeCVRidge 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.

Methods

`decision_function`(X)	Predict confidence scores for samples.
`fit`(X, y[, sample_weight])	Fit the ridge classifier.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict class labels for samples in X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.

static __init__(*alphas=array([0.1, 1., 10.]), fit_intercept=True, normalize=False, scoring=None, cv=None, class_weight=None*)

decision_function(*X*)
Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returnsarray, 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.

static **fit**(*X, y, sample_weight=None*)
Fit the ridge classifier.

ParametersX : 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 : float or numpy array of shape (n_samples,)

Sample weight.

Returnsself : object

Returns self.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*X*)
Predict class labels for samples in X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Samples.

ReturnsC : array, shape = [n_samples]

Predicted class label per sample.

score(*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

42.19.30 sklearn.linear_model.RidgeCV

class sklearn.linear_model.**RidgeCV**(*alphas=array([0.1, 1., 10.]), fit_intercept=True, normalize=False, scoring=None, cv=None, gcv_mode=None, store_cv_values=False*)

Ridge regression with built-in cross-validation.

By default, it performs Generalized Cross-Validation, which is a form of efficient Leave-One-Out cross-validation.

Parametersalphas : numpy array of shape [n_alphas]

Array of alpha values to try. 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.

fit_intercept : boolean

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 : boolean, optional, default False

If True, the regressors X will be normalized before regression.

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)`.

cv : integer or cross-validation generator, optional

If None, Generalized Cross-Validation (efficient Leave-One-Out) will be used. If an integer is passed, it is the number of folds for KFold cross validation. Specific cross-validation objects can be passed, see sklearn.cross_validation module for the list of possible objects

gcv_mode : {None, 'auto', 'svd', eigen'}, optional

Flag indicating which strategy to use when performing Generalized Cross-Validation. Options are:

```
'auto' : use svd if n_samples > n_features or when X is a sparse
         matrix, otherwise use eigen
'svd' : force computation via singular value decomposition of X
        (does not work for sparse matrices)
'eigen' : force computation via eigendecomposition of X^T X
```

The 'auto' mode is the default and is intended to pick the cheaper option of the two depending upon the shape and format of the training data.

store_cv_values : boolean, 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).

Attributescv_values_ : array, 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_ : array, shape = [n_features] or [n_targets, n_features]

Weight vector(s).

alpha_ : float

Estimated regularization parameter.

intercept_ : float | array, shape = (n_targets,)

Independent term in decision function. Set to 0.0 if `fit_intercept = False`.

See also:

RidgeRidge regression

RidgeClassifierRidge classifier

RidgeClassifierCVRidge classifier with built-in cross validation

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y[, sample_weight])	Fit Ridge regression model
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

__init__(*alphas=array([0.1, 1., 10.]), fit_intercept=True, normalize=False, scoring=None, cv=None, gcv_mode=None, store_cv_values=False*)

decision_function(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

fit (*X*, *y*, *sample_weight=None*)
Fit Ridge regression model

ParametersX : array-like, shape = [n_samples, n_features]

Training data

y : array-like, shape = [n_samples] or [n_samples, n_targets]

Target values

sample_weight : float or array-like of shape [n_samples]

Sample weight

Returnsself : Returns self.

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)
Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.RidgeCV`

Figure 42.397: *Face completion with a multi-output estimators*

42.19.31 `sklearn.linear_model`.SGDClassifier

class `sklearn.linear_model.`**`SGDClassifier`**(*loss='hinge'*, *penalty='l2'*, *alpha=0.0001*, *l1_ratio=0.15*, *fit_intercept=True*, *n_iter=5*, *shuffle=True*, *verbose=0*, *epsilon=0.1*, *n_jobs=1*, *random_state=None*, *learning_rate='optimal'*, *eta0=0.0*, *power_t=0.5*, *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.

Parametersloss : str, 'hinge', 'log', 'modified_huber', 'squared_hinge', 'perceptron', or a regression loss: 'squared_loss', 'huber', 'epsilon_insensitive', or 'squared_epsilon_insensitive'

The loss function to be used. Defaults to 'hinge', which gives a linear SVM. 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 : str, 'none', 'l2', 'l1', or 'elasticnet'

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

Constant that multiplies the regularization term. Defaults to 0.0001

l1_ratio : float

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

Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

n_iter : int, optional

The number of passes over the training data (aka epochs). The number of iterations is set to 1 if using partial_fit. Defaults to 5.

shuffle : bool, optional

Whether or not the training data should be shuffled after each epoch. Defaults to True.

random_state : int seed, RandomState instance, or None (default)

The seed of the pseudo random number generator to use when shuffling the data.

verbose : integer, optional

The verbosity level

epsilon : float

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 : integer, optional

The number of CPUs to use to do the OVA (One Versus All, for multi-class problems) computation. -1 means 'all CPUs'. Defaults to 1.

learning_rate : string, optional

The learning rate schedule: constant: eta = eta0 optimal: eta = 1.0 / (t + t0) [default] invscaling: eta = eta0 / pow(t, power_t) where t0 is chosen by a heuristic proposed by Leon Bottou.

eta0 : double

The initial learning rate for the 'constant' or 'invscaling' schedules. The default value is 0.0 as eta0 is not used by the default schedule 'optimal'.

power_t : double

The exponent for inverse scaling learning rate [default 0.5].

class_weight : dict, {class_label: weight} or "auto" 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 "auto" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

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.

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.

Attributescoef_ : 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.

See also:

`LinearSVC`, `LogisticRegression`, `Perceptron`

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()
>>> clf.fit(X, Y)
...
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
        eta0=0.0, fit_intercept=True, l1_ratio=0.15,
```

```
        learning_rate='optimal', loss='hinge', n_iter=5, n_jobs=1,
        penalty='l2', power_t=0.5, random_state=None, shuffle=True,
        verbose=0, warm_start=False)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

decision_function(X)	Predict confidence scores for samples.
densify()	Convert coefficient matrix to dense array format.
fit(X, y[, coef_init, intercept_init, ...])	Fit linear model with Stochastic Gradient Descent.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
partial_fit(X, y[, classes, sample_weight])	Fit linear model with Stochastic Gradient Descent.
predict(X)	Predict class labels for samples in X.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(*args, **kwargs)	
sparsify()	Convert coefficient matrix to sparse format.
transform(X[, threshold])	Reduce X to its most important features.

static __init__(*loss='hinge'*, *penalty='l2'*, *alpha=0.0001*, *l1_ratio=0.15*, *fit_intercept=True*, *n_iter=5*, *shuffle=True*, *verbose=0*, *epsilon=0.1*, *n_jobs=1*, *random_state=None*, *learning_rate='optimal'*, *eta0=0.0*, *power_t=0.5*, *class_weight=None*, *warm_start=False*, *average=False*)

decision_function(*X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returnsarray, 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()

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.

Returnsself: estimator :

fit(*X*, *y*, *coef_init=None*, *intercept_init=None*, *class_weight=None*, *sample_weight=None*)

Fit linear model with Stochastic Gradient Descent.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

Training data

y : numpy array, 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.

sample_weight : array-like, shape (n_samples,), optional

Weights applied to individual samples. If not provided, uniform weights are assumed. These weights will be multiplied with class_weight (passed through the contructor) if class_weight is specified

Returnsself : returns an instance of self.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

partial_fit(*X*, *y*, *classes=None*, *sample_weight=None*)

Fit linear model with Stochastic Gradient Descent.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

Subset of the training data

y : numpy array, 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*.

sample_weight : array-like, shape (n_samples,), optional

Weights applied to individual samples. If not provided, uniform weights are assumed.

Returnsself : returns an instance of self.

predict (*X*)

Predict class labels for samples in X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Samples.

ReturnsC : array, shape = [n_samples]

Predicted class label per sample.

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

sparsify ()

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.

Returnsself: 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.

transform (*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Examples using `sklearn.linear_model.SGDClassifier`

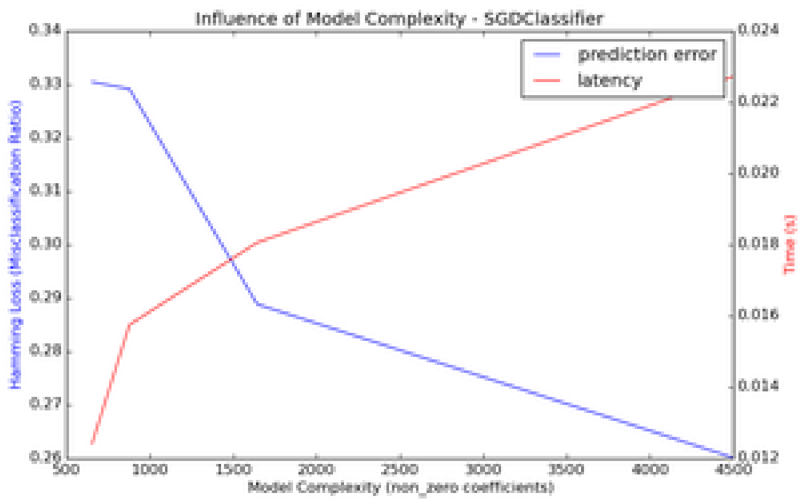


Figure 42.398: *Model Complexity Influence*

42.19.32 `sklearn.linear_model`.SGDRegressor

class `sklearn.linear_model.`**`SGDRegressor`**(*loss='squared_loss'*, *penalty='l2'*, *alpha=0.0001*, *l1_ratio=0.15*, *fit_intercept=True*, *n_iter=5*, *shuffle=True*, *verbose=0*, *epsilon=0.1*, *random_state=None*, *learning_rate='invscaling'*, *eta0=0.01*, *power_t=0.25*, *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

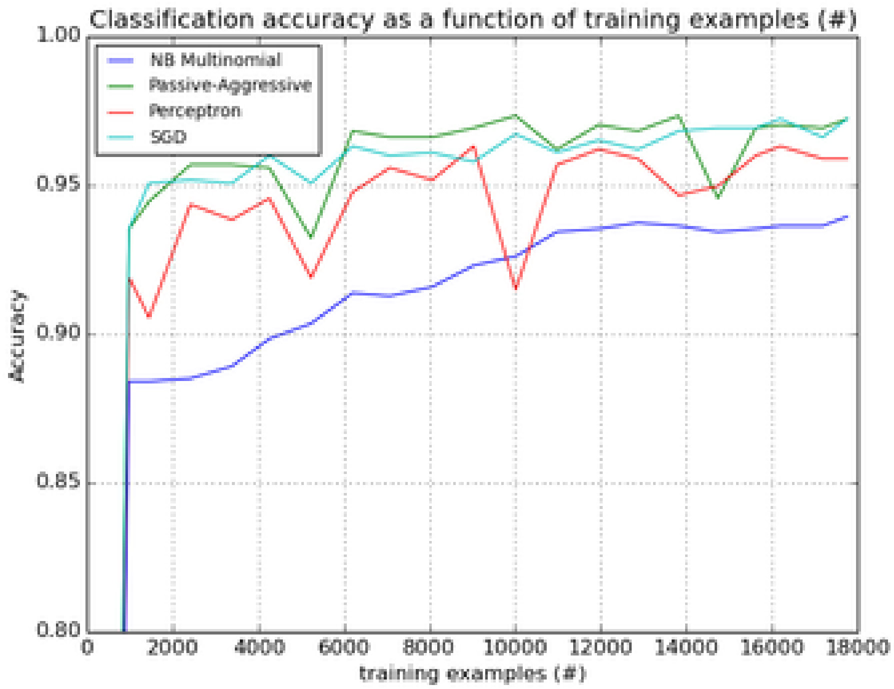


Figure 42.399: *Out-of-core classification of text documents*

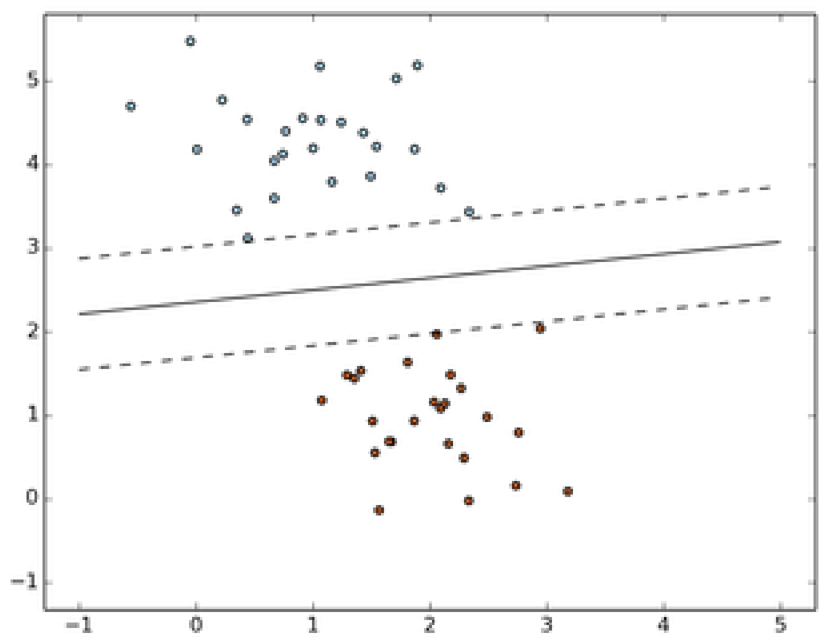


Figure 42.400: *SGD: Maximum margin separating hyperplane*

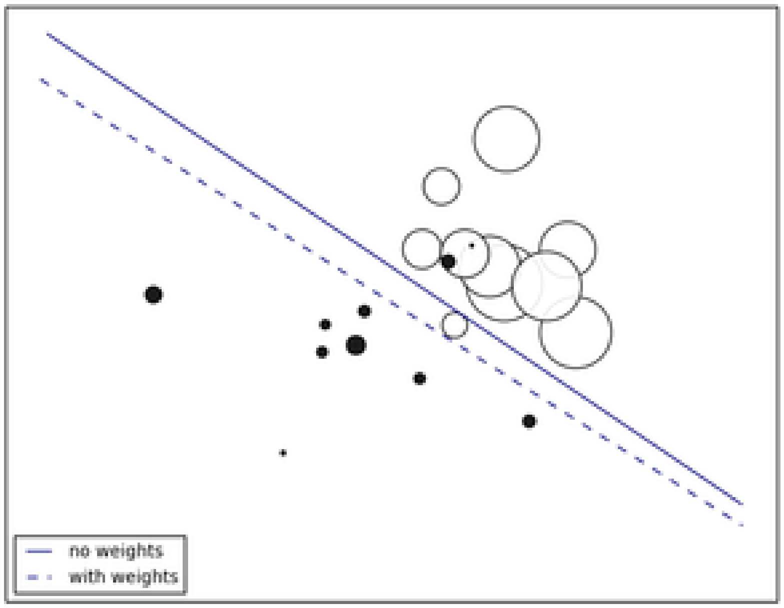


Figure 42.401: *SGD: Weighted samples*

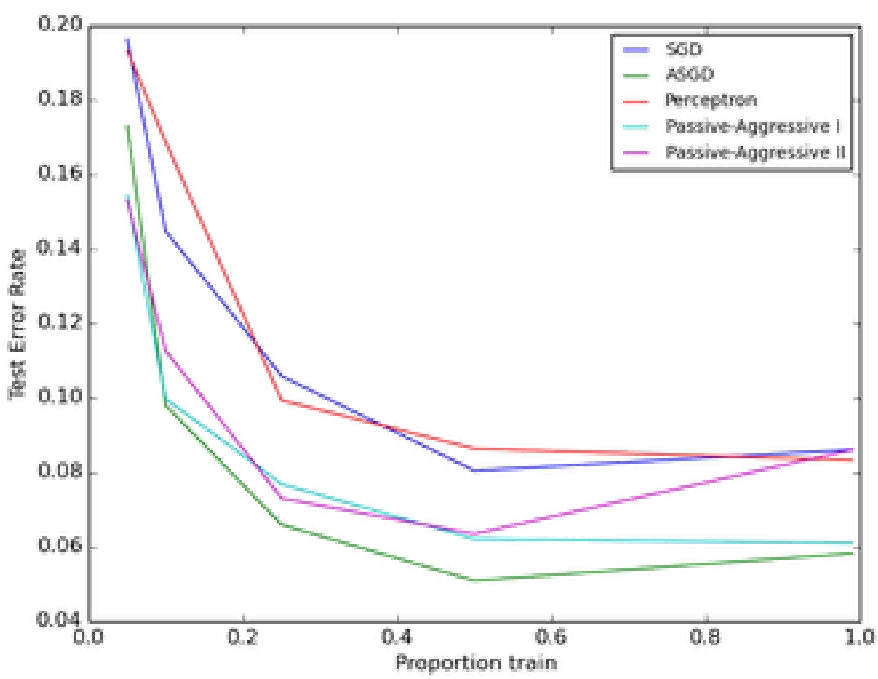


Figure 42.402: *Comparing various online solvers*

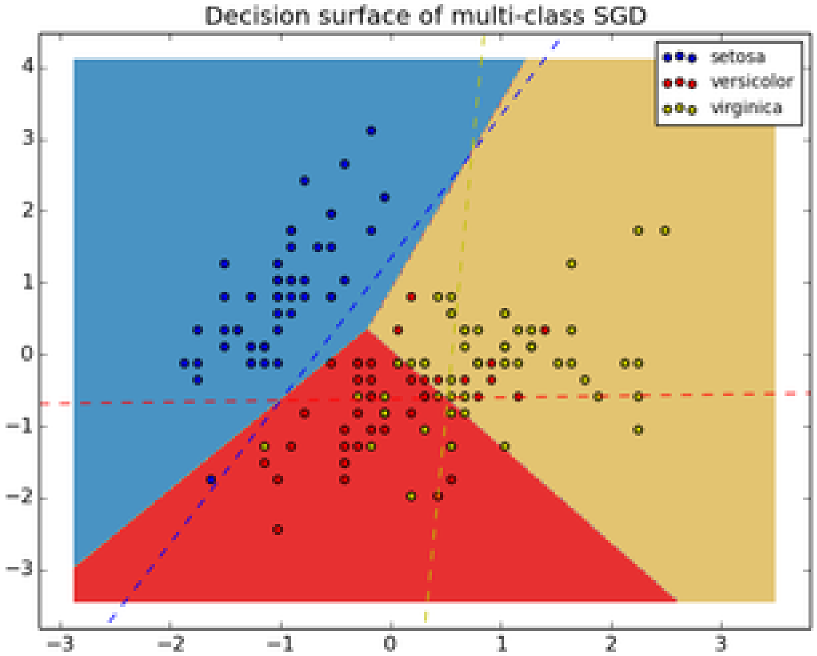


Figure 42.403: *Plot multi-class SGD on the iris dataset*

Figure 42.404: *Sample pipeline for text feature extraction and evaluation*

Figure 42.405: *Classification of text documents: using a MLComp dataset*

Figure 42.406: *Classification of text documents using sparse features*

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.

Parametersloss : str, 'squared_loss', 'huber', 'epsilon_insensitive', or 'squared_epsilon_insensitive'

The loss function to be used. Defaults to 'squared_loss' which 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 : str, 'none', 'l2', 'l1', or 'elasticnet'

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

Constant that multiplies the regularization term. Defaults to 0.0001

l1_ratio : float

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

Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

n_iter : int, optional

The number of passes over the training data (aka epochs). The number of iterations is set to 1 if using partial_fit. Defaults to 5.

shuffle : bool, optional

Whether or not the training data should be shuffled after each epoch. Defaults to True.

random_state : int seed, RandomState instance, or None (default)

The seed of the pseudo random number generator to use when shuffling the data.

verbose : integer, optional

The verbosity level.

epsilon : float

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.

learning_rate : string, optional

The learning rate: constant: eta = eta0 optimal: eta = 1.0/(alpha * t) invscaling: eta = eta0 / pow(t, power_t) [default]

eta0 : double, optional

The initial learning rate [default 0.01].

power_t : double, optional

The exponent for inverse scaling learning rate [default 0.25].

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.

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.

Attributescoef_ : array, shape (n_features,)

Weights assigned to the features.

intercept_ : array, shape (1,)

The intercept term.

'average_coef_' : array, shape (n_features,)

Averaged weights assigned to the features.

'average_intercept_' : array, shape (1,)

The averaged intercept term.

See also:

`Ridge`, `ElasticNet`, `Lasso`, SVR

Examples

```
>>> import numpy as np
>>> from sklearn import linear_model
>>> 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 = linear_model.SGDRegressor()
>>> clf.fit(X, y)
...
SGDRegressor(alpha=0.0001, average=False, epsilon=0.1, eta0=0.01,
             fit_intercept=True, l1_ratio=0.15, learning_rate='invscaling',
             loss='squared_loss', n_iter=5, penalty='l2', power_t=0.25,
             random_state=None, shuffle=True, verbose=0, warm_start=False)
```

Methods

`decision_function`(X)	Predict using the linear model
`densify`()	Convert coefficient matrix to dense array format.
`fit`(X, y[, coef_init, intercept_init, ...])	Fit linear model with Stochastic Gradient Descent.

Continued on next page

Table 42.143 – continued from previous page

fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
partial_fit(X, y[, sample_weight])	Fit linear model with Stochastic Gradient Descent.
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(*args, **kwargs)	
sparsify()	Convert coefficient matrix to sparse format.
transform(X[, threshold])	Reduce X to its most important features.

static __init__(*loss='squared_loss'*, *penalty='l2'*, *alpha=0.0001*, *l1_ratio=0.15*, *fit_intercept=True*, *n_iter=5*, *shuffle=True*, *verbose=0*, *epsilon=0.1*, *random_state=None*, *learning_rate='invscaling'*, *eta0=0.01*, *power_t=0.25*, *warm_start=False*, *average=False*)

decision_function(*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

Returnsarray, shape (n_samples,) :

Predicted target values per element in X.

densify()

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.

Returnsself: estimator :

fit(*X*, *y*, *coef_init=None*, *intercept_init=None*, *sample_weight=None*)

Fit linear model with Stochastic Gradient Descent.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

Training data

y : numpy array, shape (n_samples,)

Target values

coef_init : array, shape (n_features,)

The initial coefficients to warm-start the optimization.

intercept_init : array, shape (1,)

The initial intercept to warm-start the optimization.

sample_weight : array-like, shape (n_samples,), optional

Weights applied to individual samples (1. for unweighted).

Returnsself : returns an instance of self.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

partial_fit (*X*, *y*, *sample_weight=None*)

Fit linear model with Stochastic Gradient Descent.

ParametersX : {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,), optional

Weights applied to individual samples. If not provided, uniform weights are assumed.

Returnsself : returns an instance of self.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

Returnsarray, shape (n_samples,) :

Predicted target values per element in X.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

sparsify()

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.

Returnsself: 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.

transform(*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Examples using `sklearn.linear_model.SGDRegressor`

42.19.33 `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=1*, *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.

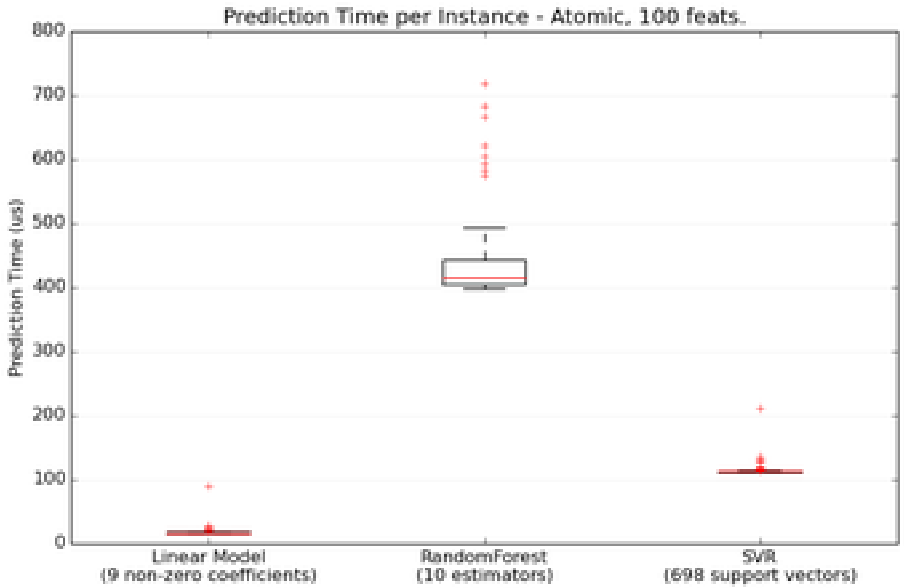


Figure 42.407: *Prediction Latency*

Parametersfit_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 : RandomState or an int seed, optional, default None

A random number generator instance to define the state of the random permutations generator.

n_jobs : integer, optional, default 1

Number of CPUs to use during the cross validation. If `-1`, use all the CPUs.

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://www.math.iupui.edu/~hpeng/MTSE_0908.pdf

Methods

`decision_function`(X)	Decision function of the linear model.
`fit`(X, y)	Fit linear model.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict using the linear model
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*fit_intercept=True*, *copy_X=True*, *max_subpopulation=10000.0*, *n_subsamples=None*, *max_iter=300*, *tol=0.001*, *random_state=None*, *n_jobs=1*, *verbose=False*)

`decision_function`(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

static fit (*X*, *y*)

Fit linear model.

ParametersX : numpy array of shape [n_samples, n_features]

Training data

y : numpy array of shape [n_samples]

Target values

Returnsself : returns an instance of self.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.linear_model.TheilSenRegressor`

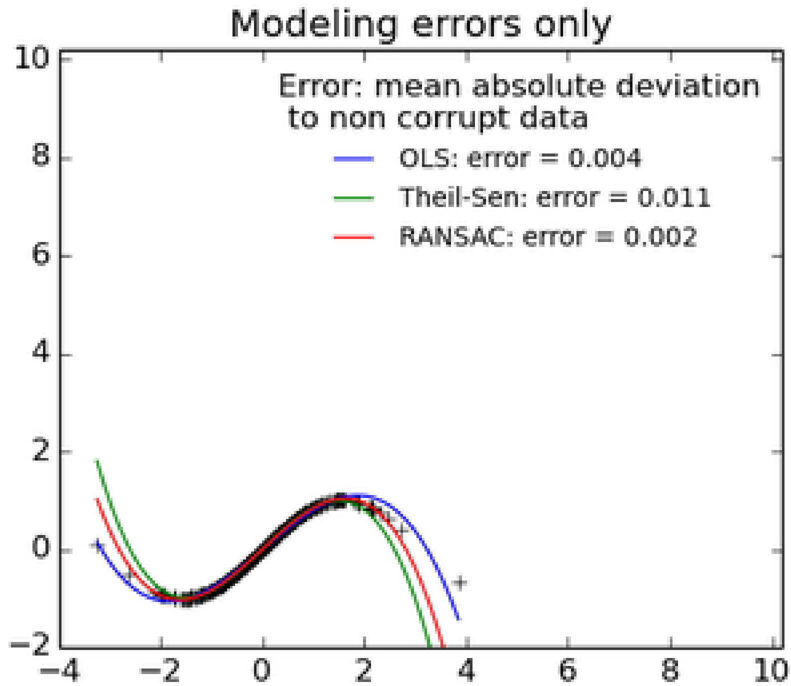


Figure 42.408: *Robust linear estimator fitting*

`linear_model.lars_path`(X, y[, Xy, Gram, ...])	Compute Least Angle Regression or Lasso path using LARS algorit
`linear_model.lasso_path`(X, y[, eps, ...])	Compute Lasso path with coordinate descent
`linear_model.lasso_stability_path`(X, y[, ...])	Stabiliy path based on randomized Lasso estimates
`linear_model.orthogonal_mp`(X, y[, ...])	Orthogonal Matching Pursuit (OMP)
`linear_model.orthogonal_mp_gram`(Gram, Xy[, ...])	Gram Orthogonal Matching Pursuit (OMP)

42.19.34 `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*)

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
```

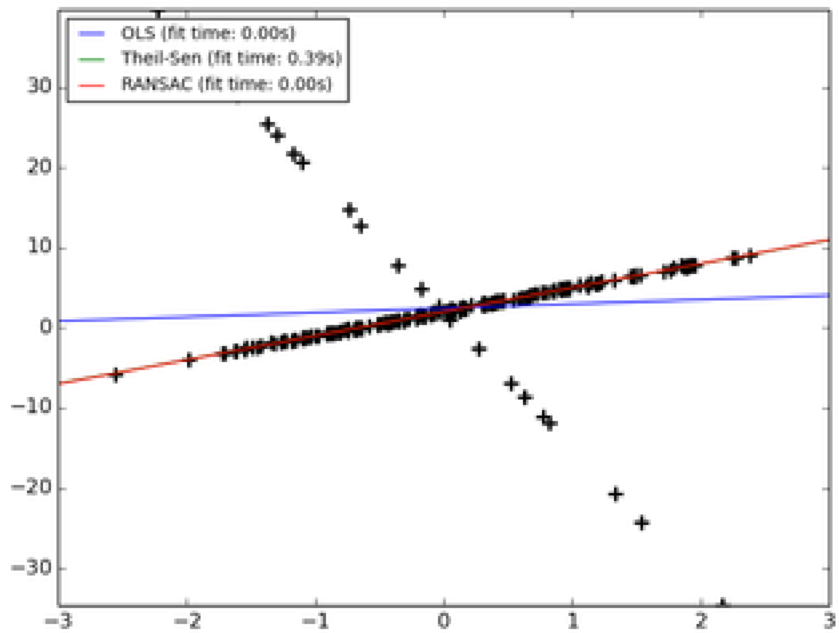


Figure 42.409: *Theil-Sen Regression*

in the case of method='lars', the objective function is only known in the form of an implicit equation (see discussion in [1])

ParametersX : array, shape: (n_samples, n_features)

Input data.

y : array, shape: (n_samples)

Input targets.

max_iter : integer, optional (default=500)

Maximum number of iterations to perform, set to infinity for no limit.

Gram : None, 'auto', array, shape: (n_features, n_features), optional

Precomputed Gram matrix (X' * X), if `'auto'`, the Gram matrix is precomputed from the given X, if there are more samples than features.

alpha_min : float, optional (default=0)

Minimum correlation along the path. It corresponds to the regularization parameter alpha parameter in the Lasso.

method : {'lar', 'lasso'}, optional (default='lar')

Specifies the returned model. Select `'lar'` for Least Angle Regression, `'lasso'` for the Lasso.

eps : float, optional (default=''np.finfo(np.float).eps'')

The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.

copy_X : bool, optional (default=True)

If `False`, `X` is overwritten.

copy_Gram : bool, optional (default=True)

If `False`, `Gram` is overwritten.

verbose : int (default=0)

Controls output verbosity.

return_path : bool, optional (default=True)

If `return_path==True` returns the entire path, else returns only the last point of the path.

return_n_iter : bool, optional (default=False)

Whether to return the number of iterations.

Returnsalphas : array, 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, shape [n_alphas]

Indices of active variables at the end of the path.

coefs : array, 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:

`lasso_path`, `LassoLars`, `Lars`, `LassoLarsCV`, `LarsCV`, `sklearn.decomposition.sparse_encode`

References

[R143], [R144], [R145]

Examples using `sklearn.linear_model.lars_path`

42.19.35 `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:

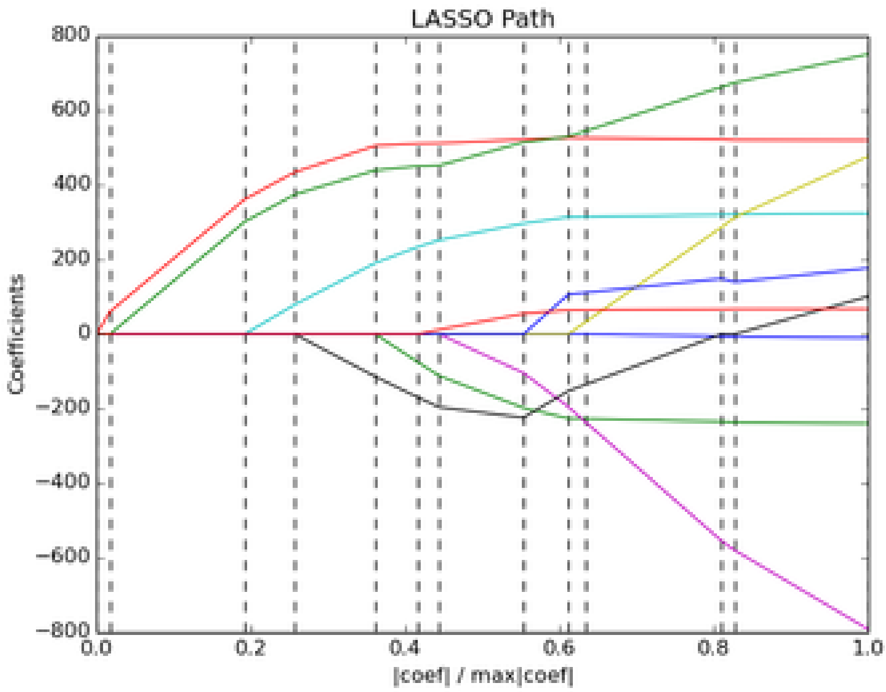


Figure 42.410: *Lasso path using LARS*

```
(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.

ParametersX : {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.

params : kwargs

keyword arguments passed to the coordinate descent solver.

positive : bool, default False

If set to True, forces coefficients to be positive.

return_n_iter : bool

whether to return the number of iterations or not.

Returnsalphas : 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

See examples/linear_model/plot_lasso_coordinate_descent_path.py for an example.

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`

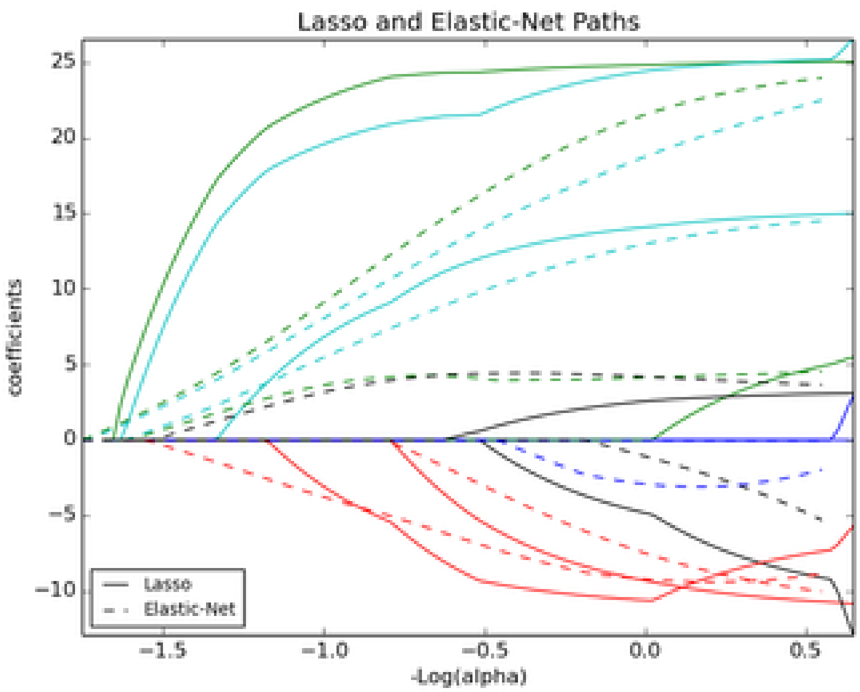


Figure 42.411: *Lasso and Elastic Net*

42.19.36 `sklearn.linear_model`.lasso_stability_path

`sklearn.linear_model.`**`lasso_stability_path`**(*X*, *y*, *scaling=0.5*, *random_state=None*, *n_resampling=200*, *n_grid=100*, *sample_fraction=0.75*, *eps=8.8817841970012523e-16*, *n_jobs=1*, *verbose=False*)

Stabiliy path based on randomized Lasso estimates

ParametersX : array-like, shape = [n_samples, n_features]

training data.

y : array-like, shape = [n_samples]

target values.

scaling : float, optional, default=0.5

The alpha parameter in the stability selection article used to randomly scale the features. Should be between 0 and 1.

random_state : integer or numpy.random.RandomState, optional

The generator used to randomize the design.

n_resampling : int, optional, default=200

Number of randomized models.

n_grid : int, optional, default=100

Number of grid points. The path is linearly reinterpolated on a grid between 0 and 1 before computing the scores.

sample_fraction : float, optional, default=0.75

The fraction of samples to be used in each randomized design. Should be between 0 and 1. If 1, all samples are used.

eps : float, optional

Smallest value of alpha / alpha_max considered

n_jobs : integer, optional

Number of CPUs to use during the resampling. If '-1', use all the CPUs

verbose : boolean or integer, optional

Sets the verbosity amount

Returnsalphas_grid : array, shape ~ [n_grid]

The grid points between 0 and 1: alpha/alpha_max

scores_path : array, shape = [n_features, n_grid]

The scores for each feature along the path.

Notes

See examples/linear_model/plot_sparse_recovery.py for an example.

Examples using `sklearn.linear_model.lasso_stability_path`

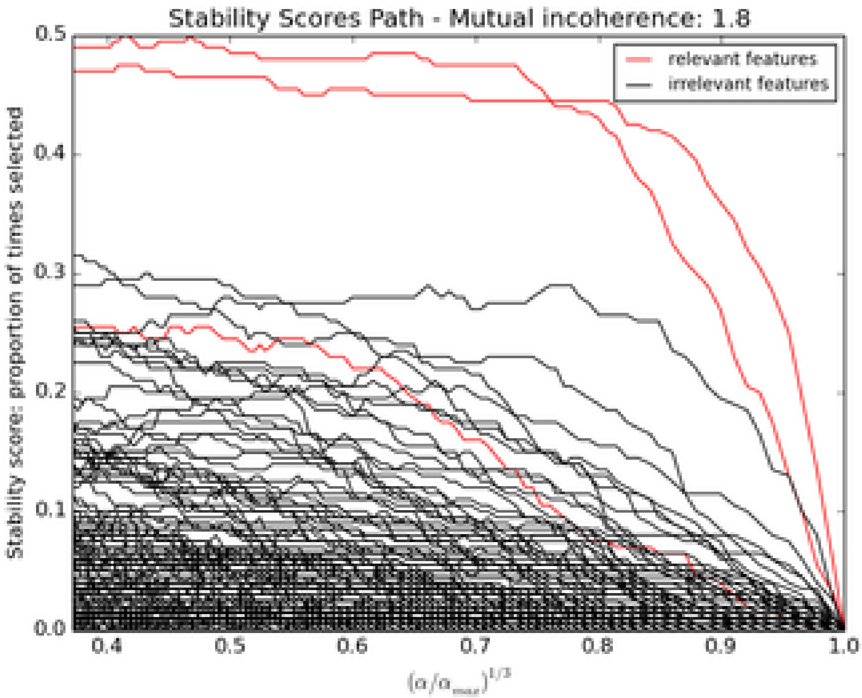


Figure 42.412: *Sparse recovery: feature selection for sparse linear models*

42.19.37 `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

ParametersX : 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.

Returnscoef : 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 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. http://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf

42.19.38 `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.

ParametersGram : 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.

Returnscoef : 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. http://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf

42.20 `sklearn.manifold`: Manifold Learning

The `sklearn.manifold` module implements data embedding techniques.

User guide: See the *Manifold learning* section for further details.

`manifold.LocallyLinearEmbedding`([...])	Locally Linear Embedding
`manifold.Isomap`([n_neighbors, n_components, ...])	Isomap 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.

42.20.1 `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*)

Locally Linear Embedding

Parametersn_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: numpy.RandomState or int, optional :

The generator or seed used to determine the starting vector for arpack iterations. Defaults to numpy.random.

Attributesembedding_vectors_ : array-like, shape [n_components, n_samples]

Stores the embedding vectors

reconstruction_error_ : float

Reconstruction error associated with *embedding_vectors_*

nbrs_ : NearestNeighbors object

Stores nearest neighbors instance, including BallTree or KDtree if applicable.

References

[R147], [R148], [R149], [R150]

Methods

`fit`(X[, y])	Compute the embedding vectors for data X
`fit_transform`(X[, y])	Compute the embedding vectors for data X and transform X.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Transform new points into embedding space.

static `__init__`(*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*)

static `fit`(*X*, *y=None*)

Compute the embedding vectors for data X

ParametersX : array-like of shape [n_samples, n_features]

training set.

Returnsself : returns an instance of self.

static `fit_transform`(*X*, *y=None*)
Compute the embedding vectors for data X and transform X.

ParametersX : array-like of shape [n_samples, n_features]

training set.

ReturnsX_new: array-like, shape (n_samples, n_components) :

`get_params`(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

`set_params`(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static `transform`(*X*)
Transform new points into embedding space.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsX_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`

42.20.2 `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'*)
Isomap Embedding

Non-linear dimensionality reduction through Isometric Mapping

Parametersn_neighbors : integer

number of neighbors to consider for each point.

n_components : integer

number of coordinates for the manifold

eigen_solver : ['auto'|'arpack'|'dense']

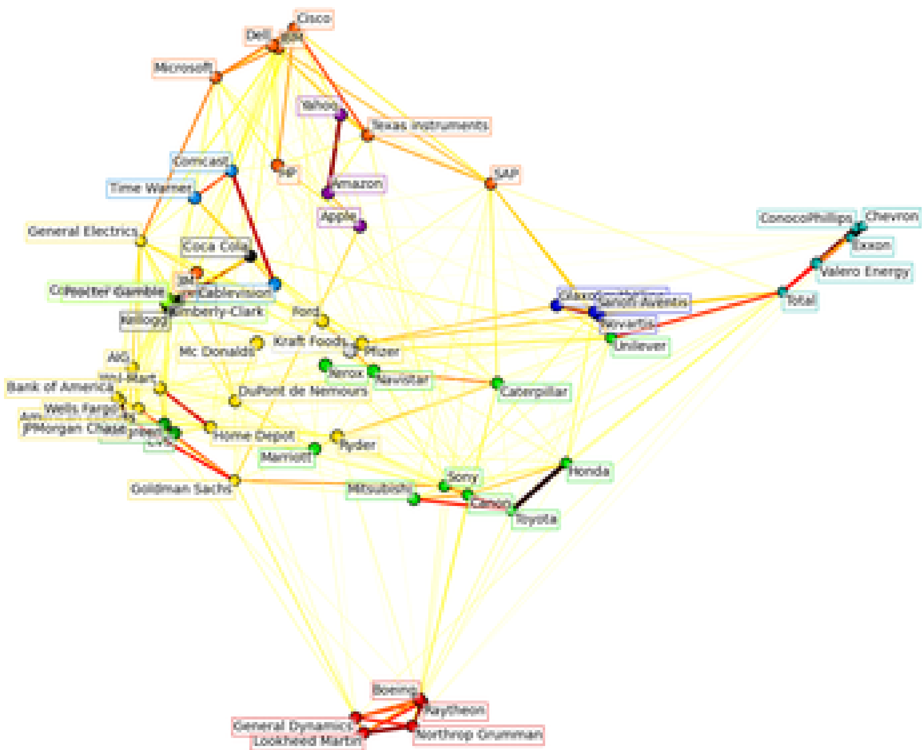


Figure 42.413: *Visualizing the stock market structure*

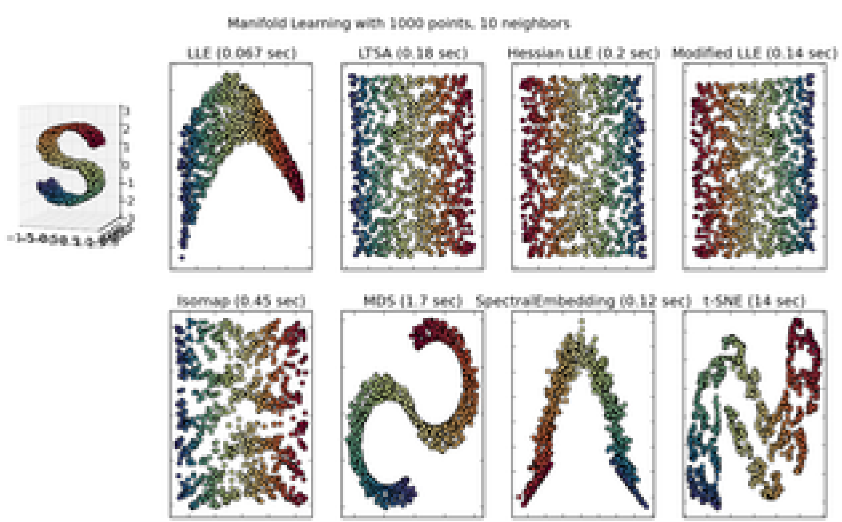


Figure 42.414: *Comparison of Manifold Learning methods*

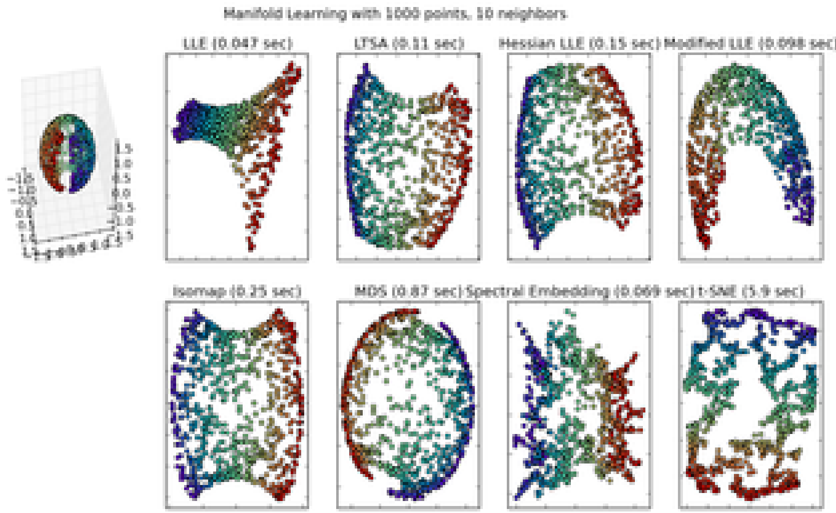


Figure 42.415: *Manifold Learning methods on a severed sphere*

Figure 42.416: *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

'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.

Attributesembedding_ : array-like, shape (n_samples, n_components)

Stores the embedding vectors.

kernel_pca_ : object

KernelPCA object used to implement the embedding.

training_data_ : array-like, shape (n_samples, n_features)

Stores the training data.

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

[R146]

Methods

`fit`(X[, y])	Compute the embedding vectors for data X
`fit_transform`(X[, y])	Fit the model from data in X and transform X.
`get_params`([deep])	Get parameters for this estimator.
`reconstruction_error`()	Compute the reconstruction error for the embedding.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Transform X.

static __init__(*n_neighbors=5, n_components=2, eigen_solver='auto', tol=0, max_iter=None, path_method='auto', neighbors_algorithm='auto'*)

static fit(*X, y=None*)
Compute the embedding vectors for data X

ParametersX : {array-like, sparse matrix, BallTree, KDTree, NearestNeighbors}

Sample data, shape = (n_samples, n_features), in the form of a numpy array, precomputed tree, or NearestNeighbors object.

Returnsself : returns an instance of self.

static fit_transform(*X, y=None*)
Fit the model from data in X and transform X.

ParametersX: {array-like, sparse matrix, BallTree, KDTree} :

Training vector, where n_samples in the number of samples and n_features is the number of features.

ReturnsX_new: array-like, shape (n_samples, n_components) :

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static reconstruction_error()
Compute the reconstruction error for the embedding.

Returnsreconstruction_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(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static transform(*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.

ParametersX: array-like, shape (n_samples, n_features) :

ReturnsX_new: array-like, shape (n_samples, n_components) :

Examples using `sklearn.manifold.Isomap`

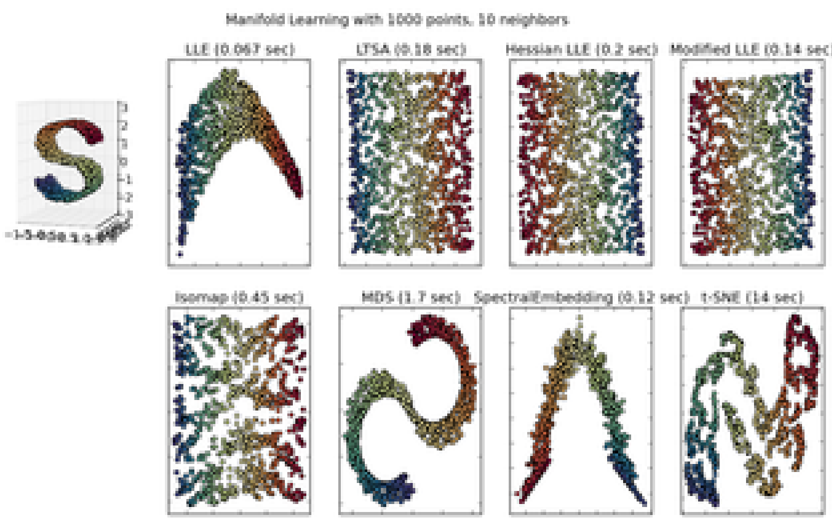


Figure 42.417: *Comparison of Manifold Learning methods*

42.20.3 `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=1*, *random_state=None*, *dissimilarity='euclidean'*)

Multidimensional scaling

Parametersmetric : boolean, optional, default: True

compute metric or nonmetric SMACOF (Scaling by Majorizing a Complicated Function) algorithm

n_components : int, optional, default: 2

number of dimension in which to immerse the similarities overridden if initial array is provided.

n_init : int, optional, default: 4

Number of time the smacof algorithm will be run with different initialisation. The final results will be the best output of the n_init consecutive runs in terms of stress.

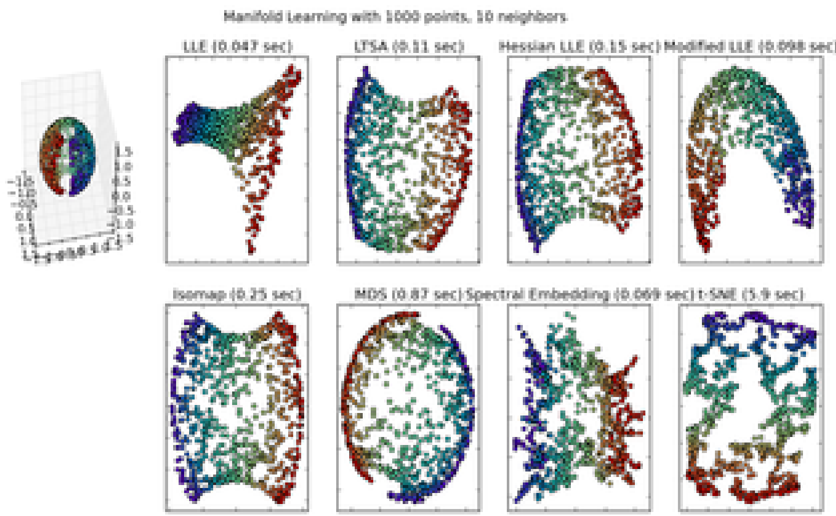


Figure 42.418: *Manifold Learning methods on a severed sphere*

Figure 42.419: *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

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-6

relative tolerance w.r.t stress to declare converge

n_jobs : int, optional, default: 1

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.

If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

random_state : integer or numpy.RandomState, optional

The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

dissimilarity : string

Which dissimilarity measure to use. Supported are 'euclidean' and 'precomputed'.

Attributesembedding_ : array-like, shape [n_components, n_samples]

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)

Methods

`fit`(X[, y, init])	Computes the position of the points in the embedding space
`fit_transform`(X[, y, init])	Fit the data from X, and returns the embedded coordinates
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*n_components=2*, *metric=True*, *n_init=4*, *max_iter=300*, *verbose=0*, *eps=0.001*, *n_jobs=1*, *random_state=None*, *dissimilarity='euclidean'*)

static `fit`(*X*, *y=None*, *init=None*)

Computes the position of the points in the embedding space

ParametersX : array, shape=[n_samples, n_features], or [n_samples, n_samples] if dissimilarity='precomputed'

Input data.

init : {None or ndarray, shape (n_samples,)}, optional

If None, randomly chooses the initial configuration if ndarray, initialize the SMACOF algorithm with this array.

static **fit_transform**(*X*, *y=None*, *init=None*)

Fit the data from X, and returns the embedded coordinates

ParametersX : array, shape=[n_samples, n_features], or [n_samples, n_samples] if dissimilarity='precomputed'

Input data.

init : {None or ndarray, shape (n_samples,)}, optional

If None, randomly chooses the initial configuration if ndarray, initialize the SMACOF algorithm with this array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.manifold.MDS`

42.20.4 `sklearn.manifold`.SpectralEmbedding

class `sklearn.manifold.`**SpectralEmbedding**(*n_components=2*, *affinity='nearest_neighbors'*, *gamma=None*, *random_state=None*, *eigen_solver=None*, *n_neighbors=None*)

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.

Parametersn_components : integer, default: 2

The dimension of the projected 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.

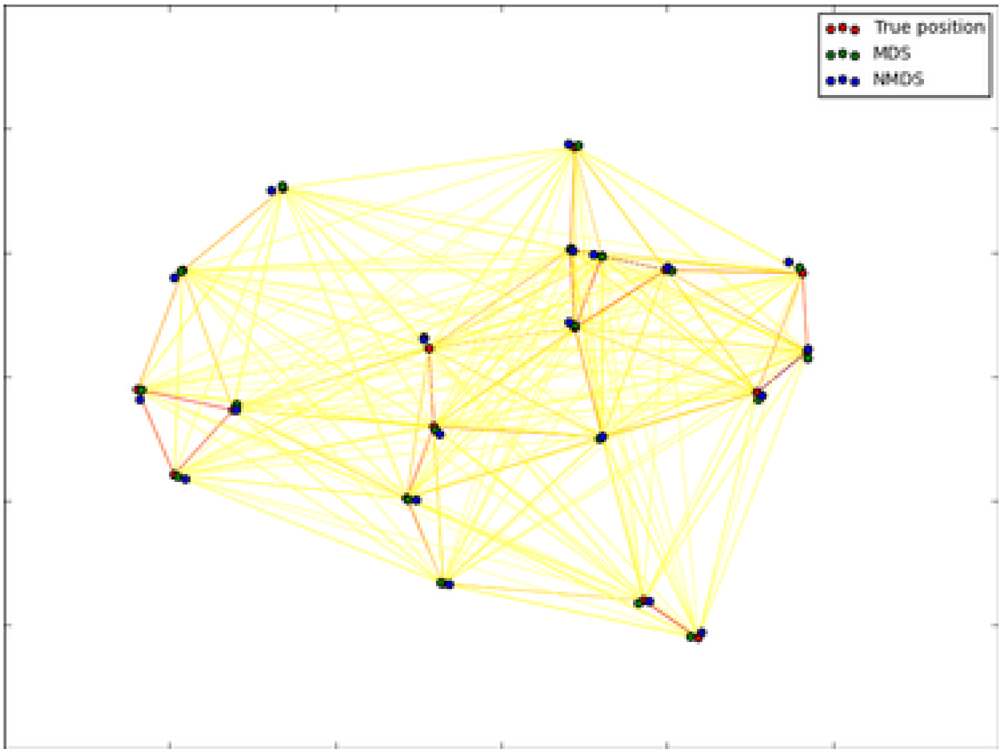


Figure 42.420: *Multi-dimensional scaling*

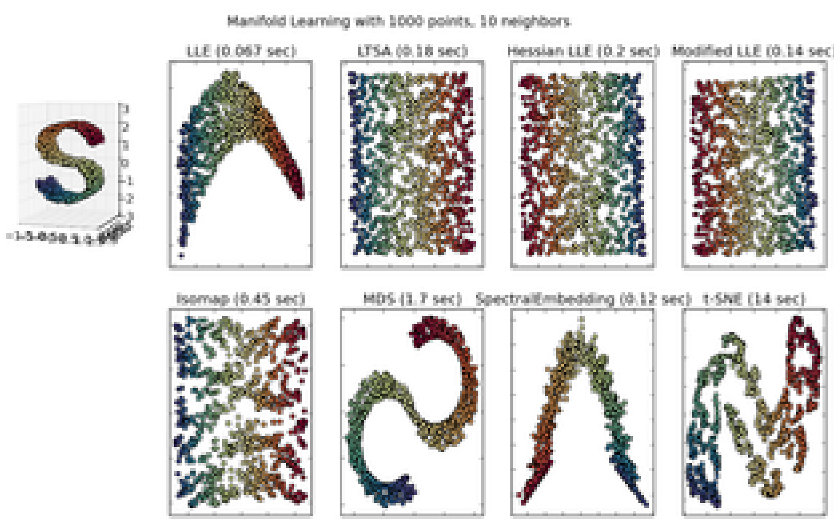


Figure 42.421: *Comparison of Manifold Learning methods*

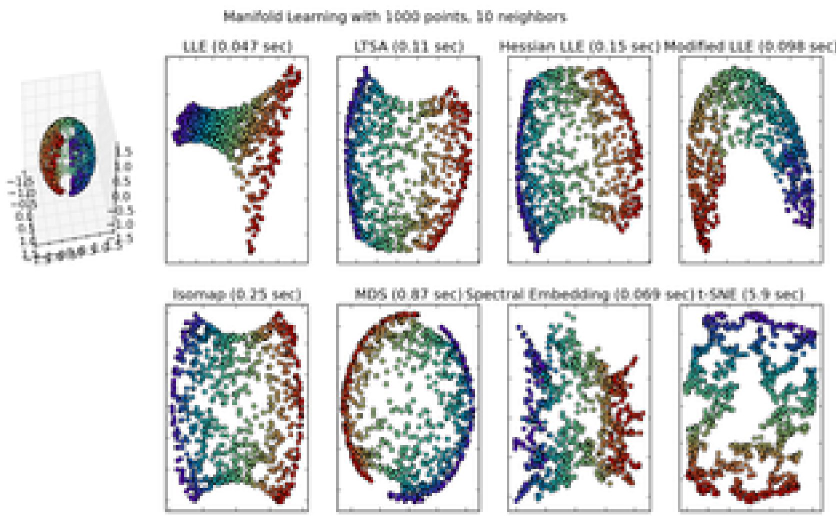


Figure 42.422: *Manifold Learning methods on a severed sphere*

Figure 42.423: *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

random_state : int seed, RandomState instance, or None, default

A pseudo random number generator used for the initialization of the lobpcg eigen vectors decomposition when eigen_solver == 'amg'.

affinity : string or callable, default

How to construct the affinity matrix.

- 'nearest_neighbors' : construct affinity matrix by knn graph
- 'rbf' : construct affinity matrix by rbf kernel
- 'precomputed' : interpret X as precomputed affinity matrix
- 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.

n_neighbors : int, default

Number of nearest neighbors for nearest_neighbors graph building.

Attributesembedding_ : 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.

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, 2011 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

Methods

`fit`(X[, y])	Fit the model from data in X.
`fit_transform`(X[, y])	Fit the model from data in X and transform X.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*n_components=2, affinity='nearest_neighbors', gamma=None, random_state=None, eigen_solver=None, n_neighbors=None*)

static `fit`(*X, y=None*)

Fit the model from data in X.

ParametersX : 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.

If affinity is "precomputed" X : array-like, shape (n_samples, n_samples), Interpret X as precomputed adjacency graph computed from samples.

Returnsself : object

Returns the instance itself.

static fit_transform(*X*, *y=None*)

Fit the model from data in X and transform X.

ParametersX: 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.

If affinity is "precomputed" X : array-like, shape (n_samples, n_samples), Interpret X as precomputed adjacency graph computed from samples.

ReturnsX_new: array-like, shape (n_samples, n_components) :

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.manifold.SpectralEmbedding`

42.20.5 `sklearn.manifold`.TSNE

class sklearn.manifold.**TSNE**(*n_components=2*, *perplexity=30.0*, *early_exaggeration=4.0*, *learning_rate=1000.0*, *n_iter=1000*, *metric='euclidean'*, *init='random'*, *verbose=0*, *random_state=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].

Parametersn_components : int, optional (default: 2)

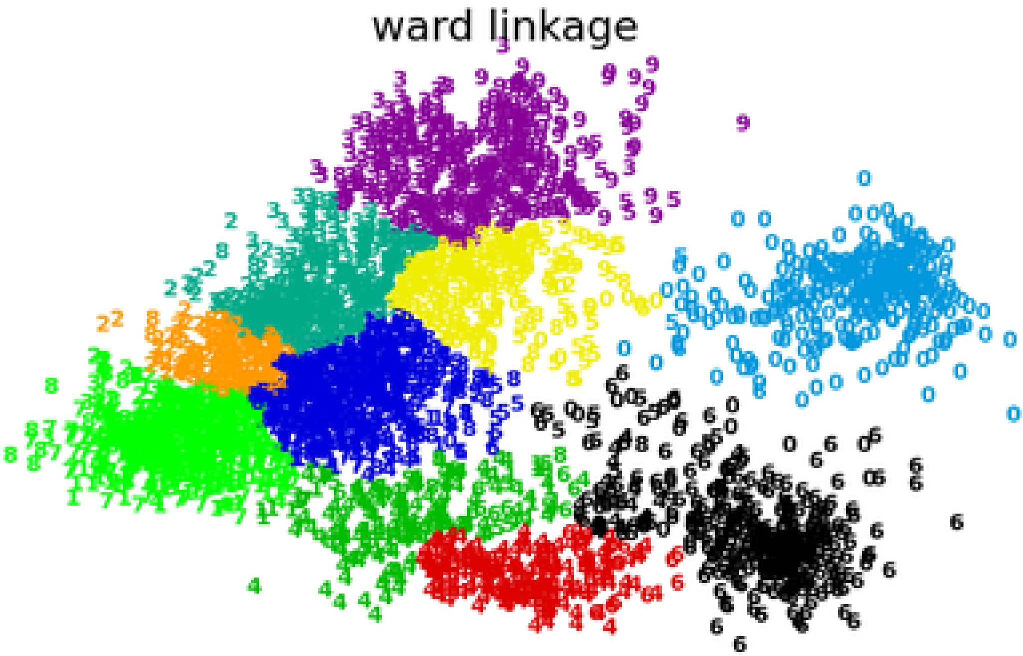


Figure 42.424: *Various Agglomerative Clustering on a 2D embedding of digits*

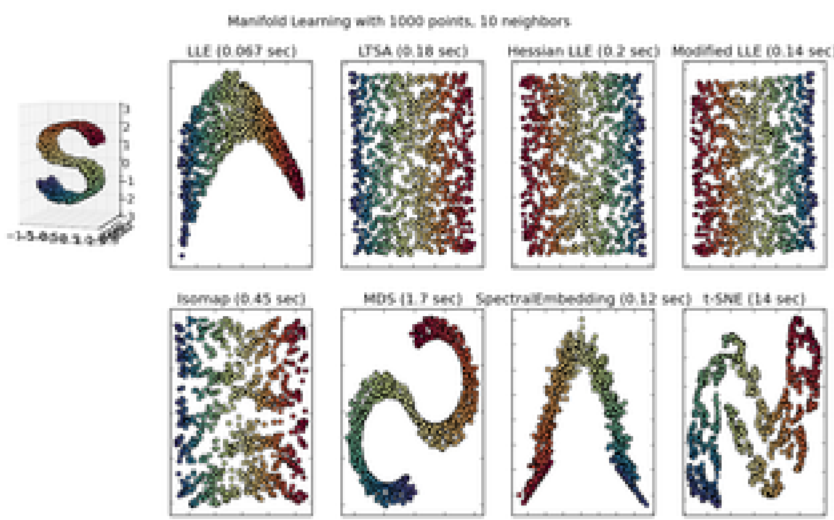


Figure 42.425: *Comparison of Manifold Learning methods*

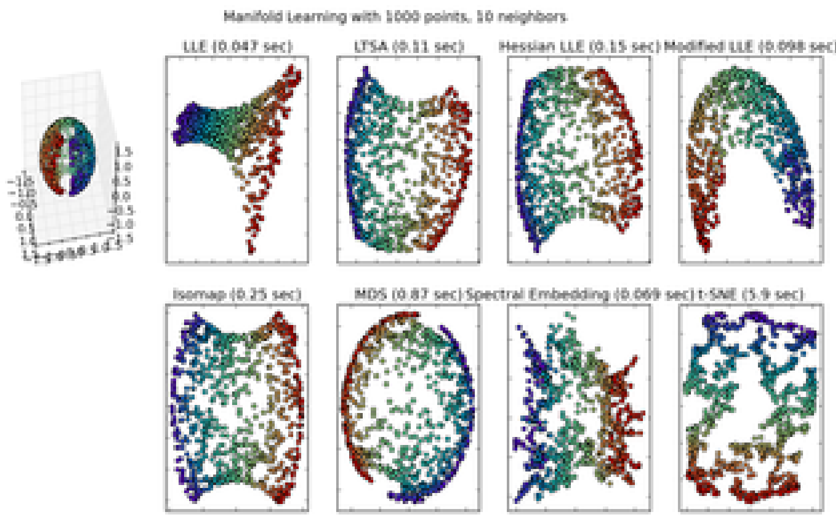


Figure 42.426: *Manifold Learning methods on a severed sphere*

Figure 42.427: *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

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 selcting a value between 5 and 50. The choice is not extremely critical since t-SNE is quite insensitive to this parameter.

early_exaggeration : float, optional (default: 4.0)

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: 1000)

The learning rate can be a critical parameter. It should be between 100 and 1000. If the cost function increases during initial optimization, the early exaggeration factor or the learning rate might be too high. If the cost function gets stuck in a bad local minimum increasing the learning rate helps sometimes.

n_iter : int, optional (default: 1000)

Maximum number of iterations for the optimization. Should be at least 200.

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, optional (default: "random")

Initialization of embedding. Possible options are 'random' and 'pca'. 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 or RandomState instance or None (default)

Pseudo Random Number generator seed control. If None, use the numpy.random singleton. Note that different initializations might result in different local minima of the cost function.

Attributesembedding_ : array-like, shape (n_samples, n_components)

Stores the embedding vectors.

training_data_ : array-like, shape (n_samples, n_features)

Stores the training data.

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**http://homepage.tudelft.nl/19j49/t-SNE.html

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]])
>>> model = TSNE(n_components=2, random_state=0)
>>> model.fit_transform(X)
array([[  887.28...,   238.61...],
       [ -714.79...,  3243.34...],
       [  957.30..., -2505.78...],
       [-1130.28...,  -974.78...]])
```

Methods

`fit`(X[, y])	Fit the model using X as training data.
`fit_transform`(X[, y])	Transform X to the embedded space.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*n_components=2*, *perplexity=30.0*, *early_exaggeration=4.0*, *learning_rate=1000.0*, *n_iter=1000*, *metric='euclidean'*, *init='random'*, *verbose=0*, *random_state=None*)

static `fit`(*X*, *y=None*)

Fit the model using X as training data.

ParametersX : 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.

static `fit_transform`(*X*, *y=None*)

Transform X to the embedded space.

ParametersX : 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.

ReturnsX_new : array, shape (n_samples, n_components)

Embedding of the training data in low-dimensional space.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.manifold.TSNE`

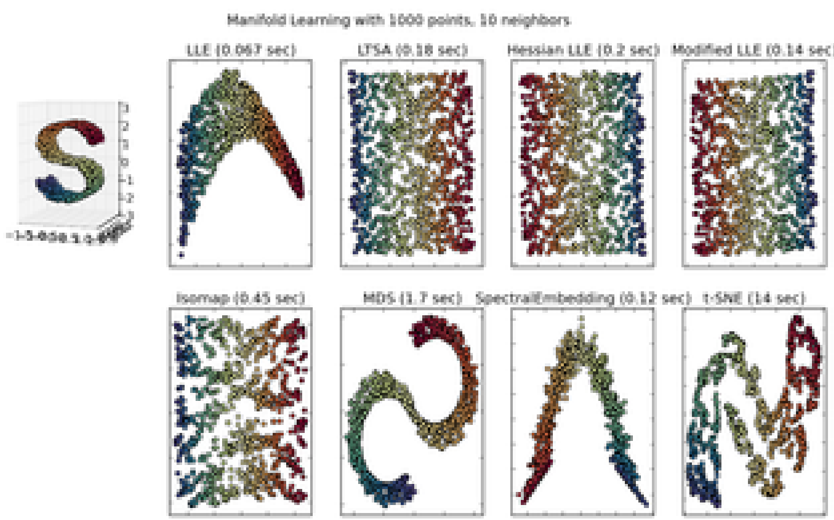


Figure 42.428: *Comparison of Manifold Learning methods*

`manifold.locally_linear_embedding`(X, ...[, ...])	Perform a Locally Linear Embedding analysis on the data.
`manifold.spectral_embedding`(adjacency[, ...])	Project the sample on the first eigen vectors of the graph Laplacian.

42.20.6 `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*)

Perform a Locally Linear Embedding analysis on the data.

ParametersX : {array-like, sparse matrix, BallTree, KDTree, NearestNeighbors}

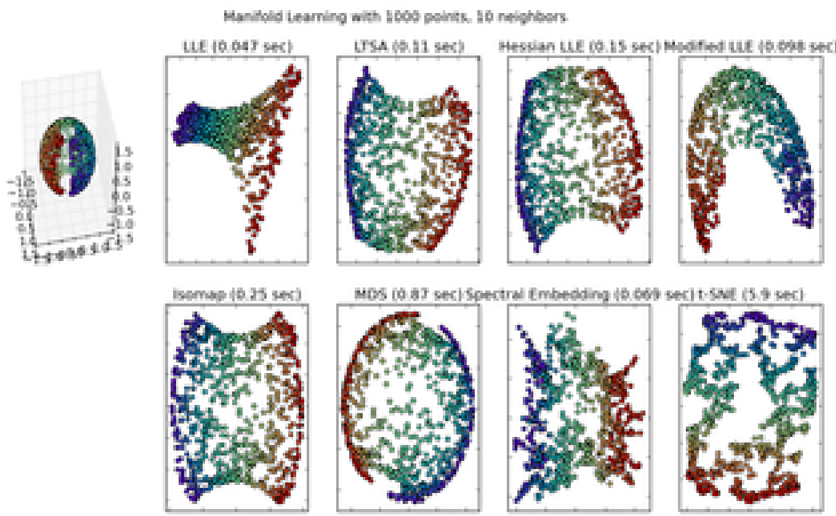


Figure 42.429: *Manifold Learning methods on a severed sphere*

Figure 42.430: *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

Sample data, shape = (n_samples, n_features), in the form of a numpy array, sparse array, precomputed tree, or 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 [R151]

hessian[use the Hessian eigenmap method. This method requires] n_neighbors > n_components * (1 + (n_components + 1) / 2. see reference [R152]

modified[use the modified locally linear embedding algorithm.] see reference [R153]

ltsa[use local tangent space alignment algorithm] see reference [R154]

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: numpy.RandomState or int, optional :

The generator or seed used to determine the starting vector for arpack iterations. Defaults to numpy.random.

ReturnsY : 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

[R151], [R152], [R153], [R154]

Examples using `sklearn.manifold.locally_linear_embedding`

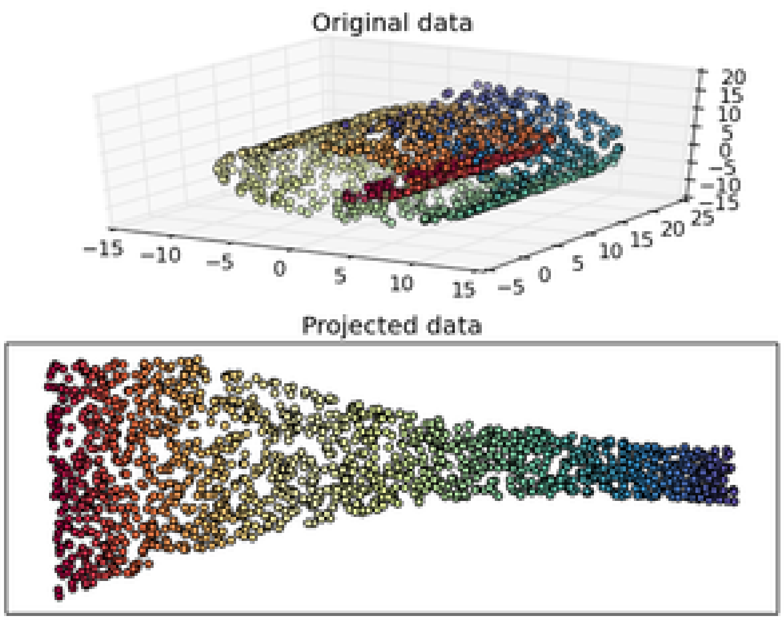


Figure 42.431: *Swiss Roll reduction with LLE*

42.20.7 `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 eigen vectors of the graph Laplacian.

The adjacency matrix is used to compute a normalized graph Laplacian whose spectrum (especially the eigen vectors associated to the smallest eigen values) 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 eigen vector decomposition works as expected.

Parametersadjacency : array-like or sparse matrix, 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 seed, RandomState instance, or None (default)

A pseudo random number generator used for the initialization of the lobpcg eigen vectors decomposition when eigen_solver == 'amg'. By default, arpack is used.

eigen_tol : float, optional, default=0.0

Stopping criterion for eigendecomposition of the Laplacian matrix when using arpack eigen_solver.

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.

norm_laplacian : bool, optional, default=True

If True, then compute normalized Laplacian.

Returnsembedding : array, shape=(n_samples, n_components)

The reduced samples.

Notes

Spectral embedding 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

•http://en.wikipedia.org/wiki/LOBPCG

•Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method Andrew V. Knyazev http://dx.doi.org/10.1137%2FS1064827500366124

42.21 `sklearn.metrics`: Metrics

See the *Model evaluation: 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.

42.21.1 Model Selection Interface

See the *The scoring parameter: defining model evaluation rules* section of the user guide for further details.

`metrics.make_scorer`(score_func[, ...])	Make a scorer from a performance metric or loss function.

Continued on next page

Table 42.153 – continued from previous page

metrics.get_scorer(scoring)

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.

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.

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.

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.

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.grid_search import GridSearchCV
>>> from sklearn.svm import LinearSVC
>>> grid = GridSearchCV(LinearSVC(), param_grid={'C': [1, 10]},
...                     scoring=ftwo_scorer)
```

sklearn.metrics.get_scorer

sklearn.metrics.**get_scorer**(*scoring*)

42.21.2 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[, reorder])	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.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.confusion_matrix(y_true, y_pred[, ...])	Compute confusion matrix to evaluate the accuracy of a classificati
metrics.f1_score(y_true, y_pred[, labels, ...])	Compute the F1 score, also known as balanced F-score or F-measu
metrics.fbeta_score(y_true, y_pred, beta[, ...])	Compute the F-beta score
metrics.hamming_loss(y_true, y_pred[, classes])	Compute the average Hamming loss.
metrics.hinge_loss(y_true, pred_decision[, ...])	Average hinge loss (non-regularized)
metrics.jaccard_similarity_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) for binary cl
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 Curve (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.
metrics.brier_score_loss(y_true, y_prob[, ...])	Compute the Brier score.

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.

Parametersy_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], optional

Sample weights.

Returnsscore : float

If `normalize == True`, return the correctly classified samples (float), else it 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_similarity_score`, `hamming_loss`, `zero_one_loss`

Notes

In binary and multiclass classification, this function is equal to the `jaccard_similarity_score` function.

Examples

```
>>> 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

Examples using `sklearn.metrics.accuracy_score`

`sklearn.metrics`.auc

`sklearn.metrics.`**`auc`**(*x*, *y*, *reorder=False*)

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`.

Parametersx : array, shape = [n]

x coordinates.

y : array, shape = [n]

y coordinates.

reorder : boolean, optional (default=False)

If True, assume that the curve is ascending in the case of ties, as for an ROC curve. If the curve is non-ascending, the result will be wrong.

Returnsauc : float

See also:

`roc_auc_score`Computes the area under the ROC curve

`precision_recall_curve`Compute precision-recall pairs for different probability thresholds

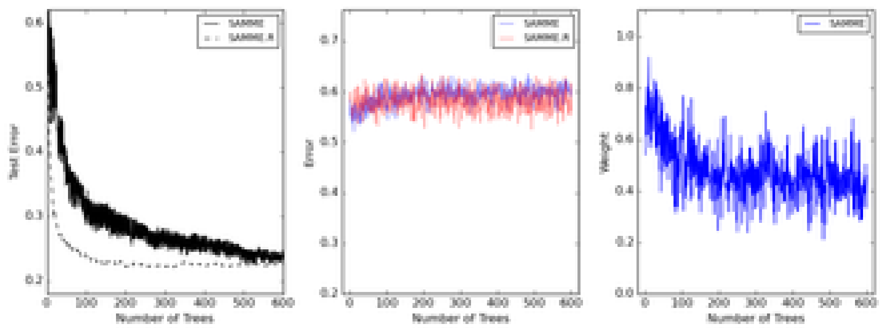


Figure 42.432: *Multi-class AdaBoosted Decision Trees*

Figure 42.433: *Classification of text documents using sparse features*

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`

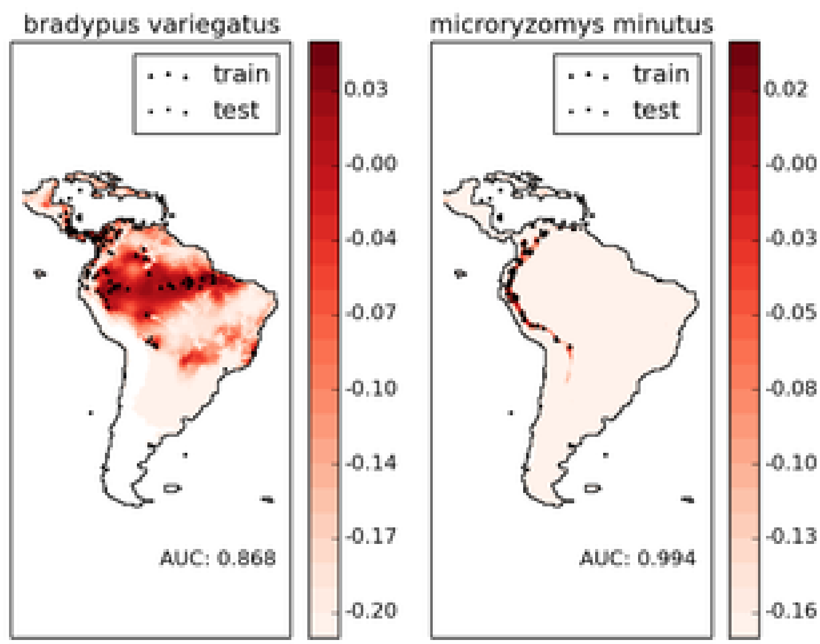


Figure 42.434: *Species distribution modeling*

`sklearn.metrics`.average_precision_score

`sklearn.metrics.`**`average_precision_score`**(*y_true*, *y_score*, *average='macro'*, *sample_weight=None*)

Compute average precision (AP) from prediction scores

This score corresponds to the area under the precision-recall curve.

Note: this implementation is restricted to the binary classification task or multilabel classification task.

Parametersy_true : array, shape = [n_samples] or [n_samples, n_classes]

True binary labels in binary label indicators.

y_score : array, shape = [n_samples] or [n_samples, n_classes]

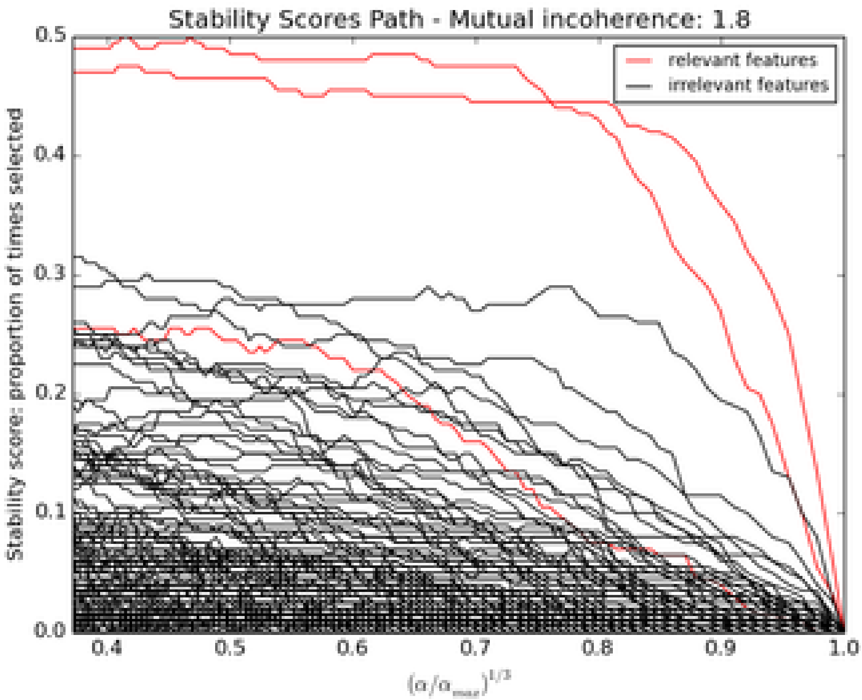


Figure 42.435: *Sparse recovery: feature selection for sparse linear models*

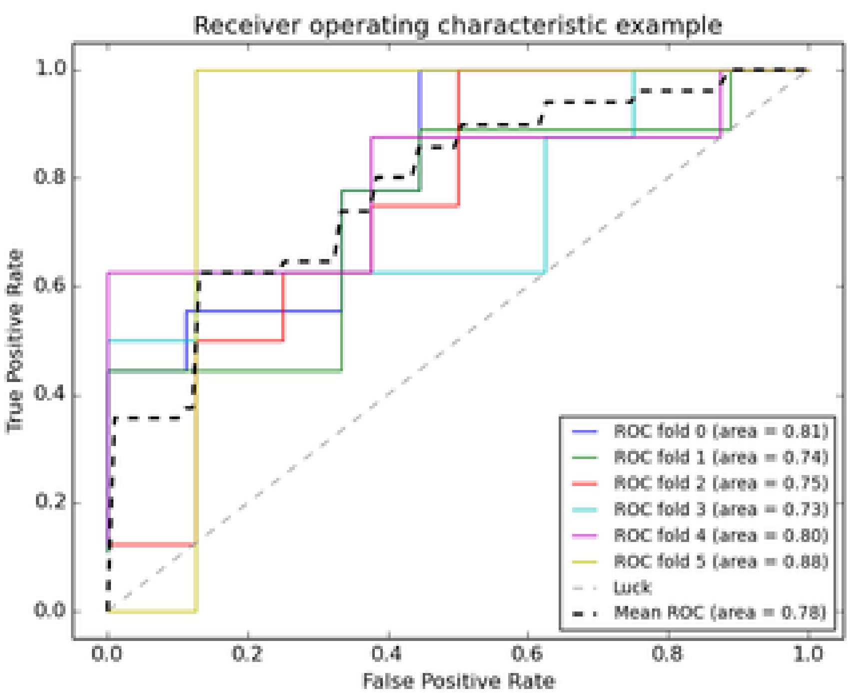


Figure 42.436: *Receiver Operating Characteristic (ROC) with cross validation*

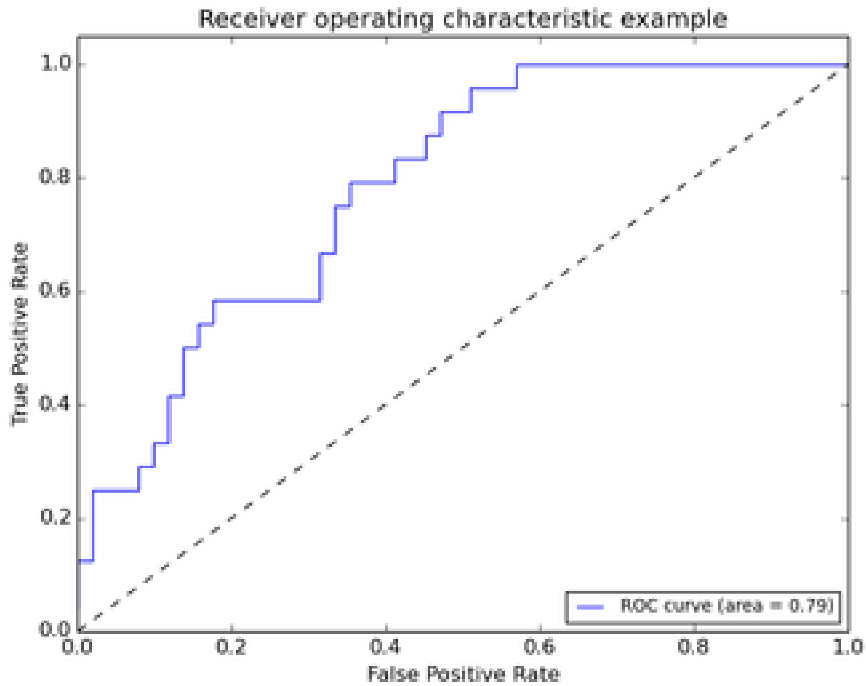


Figure 42.437: *Receiver Operating Characteristic (ROC)*

Target scores, can either be probability estimates of the positive class, confidence values, or binary decisions.

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.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

Returnsaverage_precision : float

See also:

`roc_auc_score`Area under the ROC curve

`precision_recall_curve`Compute precision-recall pairs for different probability thresholds

References

[R157]

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)
0.79...
```

Examples using `sklearn.metrics.average_precision_score`

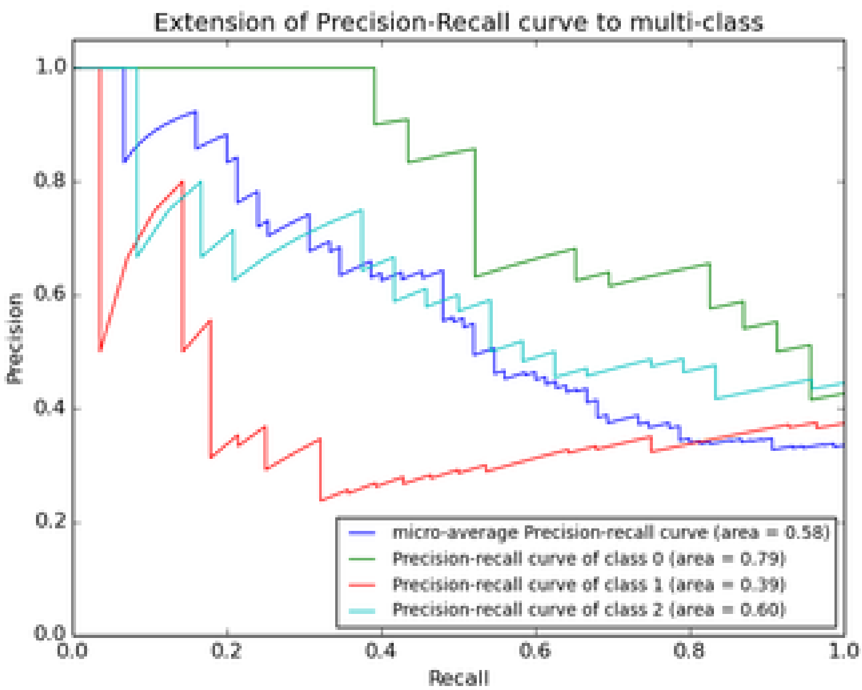


Figure 42.438: *Precision-Recall*

`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 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.

Parametersy_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], optional

Sample weights.

pos_label : int (default: None)

Label of the positive class. If None, the maximum label is used as positive class

Returnsscore : float

Brier score

References

http://en.wikipedia.org/wiki/Brier_score

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`

`sklearn.metrics`.classification_report

`sklearn.metrics.`**`classification_report`**(*y_true*, *y_pred*, *labels=None*, *target_names=None*, *sample_weight=None*, *digits=2*)

Build a text report showing the main classification metrics

Parametersy_true : 1d array-like, or label indicator array / sparse matrix

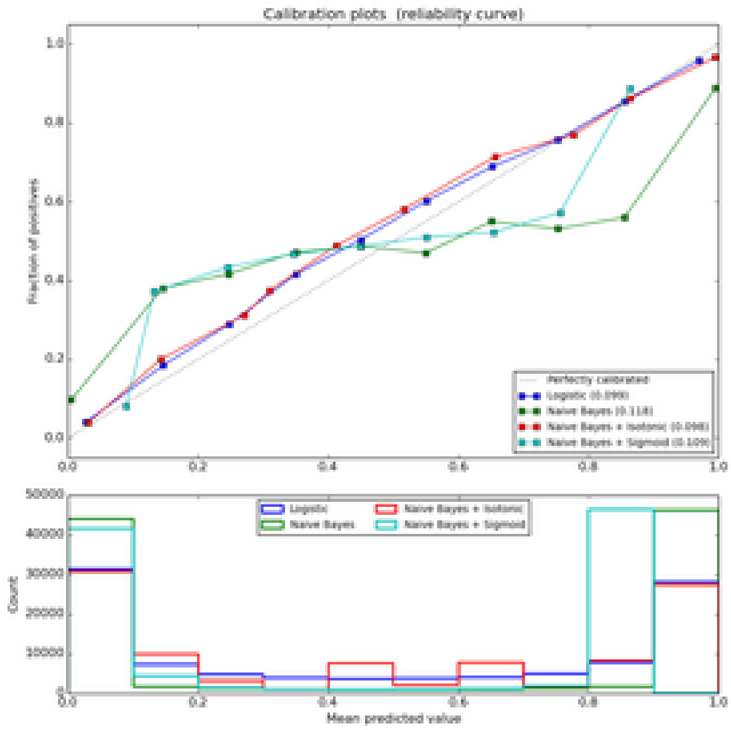


Figure 42.439: *Probability Calibration curves*

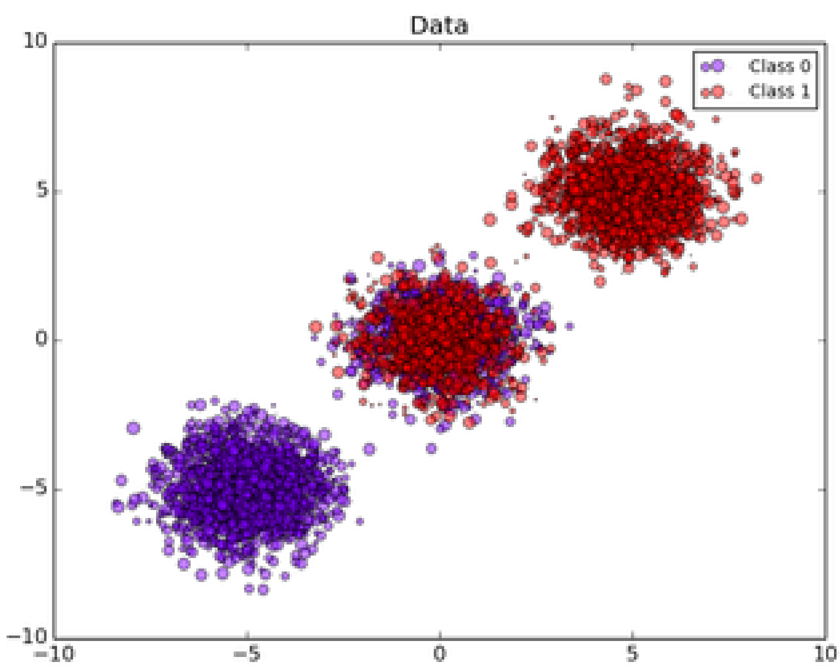


Figure 42.440: *Probability calibration of classifiers*

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], optional

Sample weights.

digits : int

Number of digits for formatting output floating point values

Returnsreport : string

Text summary of the precision, recall, F1 score for each class.

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))
             precision    recall  f1-score   support

    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

avg / total       0.70      0.60      0.61         5
```

Examples using `sklearn.metrics.classification_report`

Figure 42.441: *Feature Union with Heterogeneous Data Sources*

Figure 42.442: *Faces recognition example using eigenfaces and SVMs*

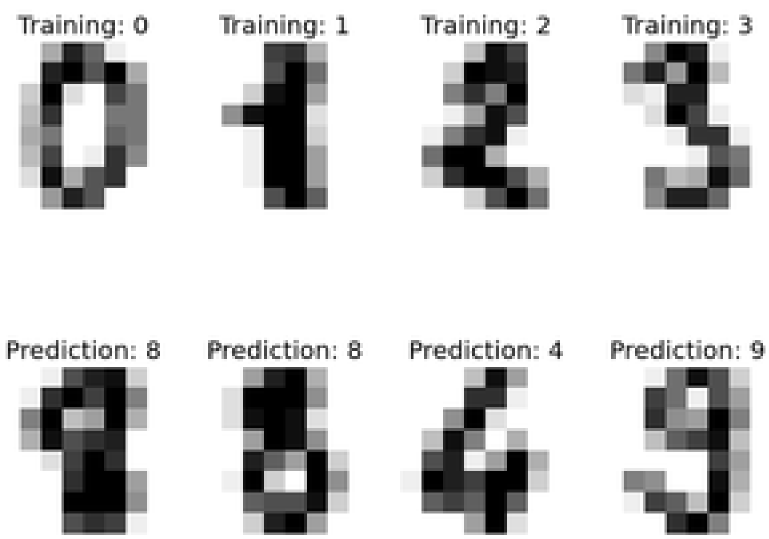


Figure 42.443: *Recognizing hand-written digits*

Figure 42.444: *Parameter estimation using grid search with cross-validation*

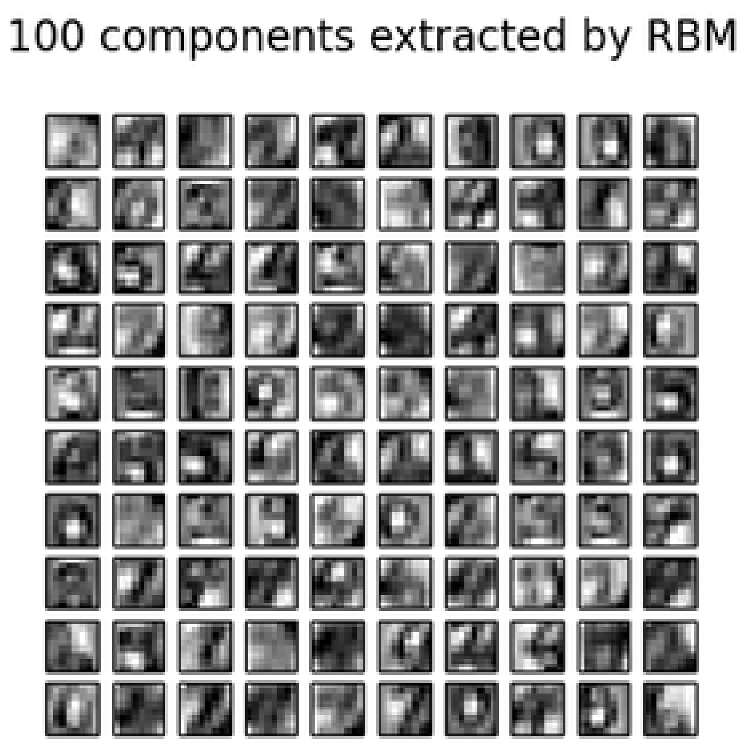


Figure 42.445: *Restricted Boltzmann Machine features for digit classification*

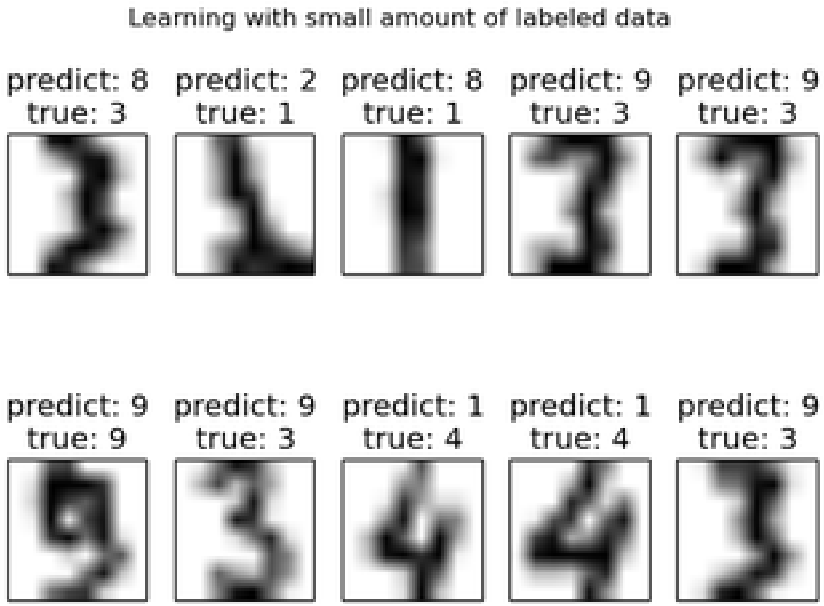


Figure 42.446: *Label Propagation digits: Demonstrating performance*

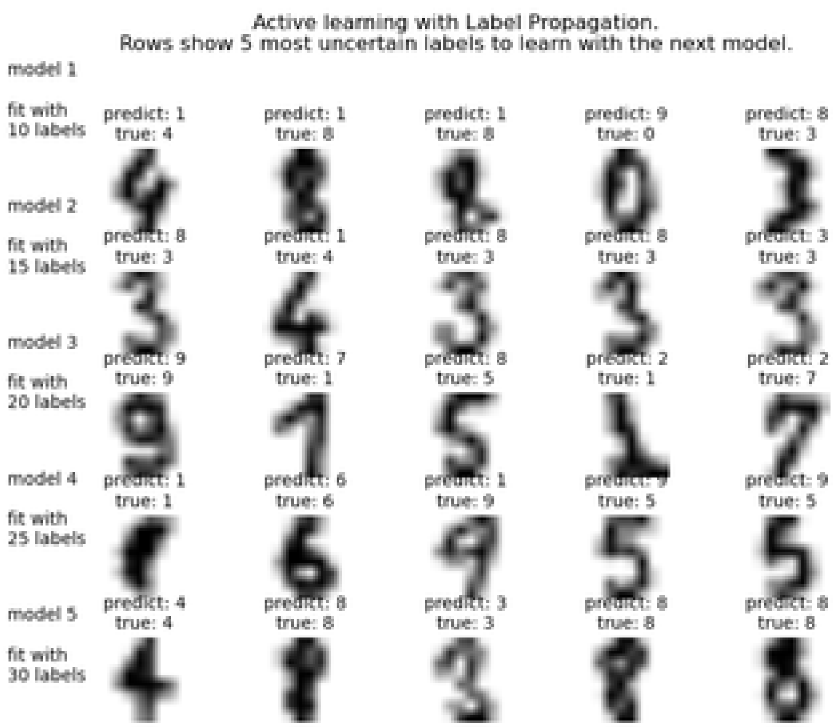


Figure 42.447: *Label Propagation digits active learning*

Figure 42.448: *Classification of text documents: using a MLComp dataset*

Figure 42.449: *Classification of text documents using sparse features*

sklearn.metrics.confusion_matrix

sklearn.metrics.**confusion_matrix**(*y_true*, *y_pred*, *labels=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 but predicted to be in group j.

Parametersy_true : array, shape = [n_samples]

Ground truth (correct) target values.

y_pred : array, shape = [n_samples]

Estimated targets as returned by a classifier.

labels : array, shape = [n_classes], optional

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.

ReturnsC : array, shape = [n_classes, n_classes]

Confusion matrix

References

[R159]

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]])
```

Examples using sklearn.metrics.confusion_matrix

sklearn.metrics.f1_score

sklearn.metrics.**f1_score**(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*)

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 weighted average of the F1 score of each class.

Parametersy_true : 1d array-like, or label indicator array / sparse matrix

Figure 42.450: *Faces recognition example using eigenfaces and SVMs*

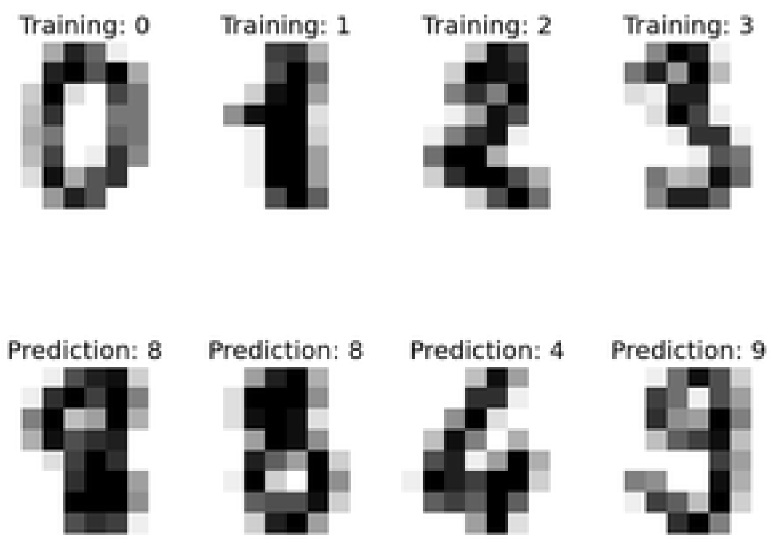


Figure 42.451: *Recognizing hand-written digits*

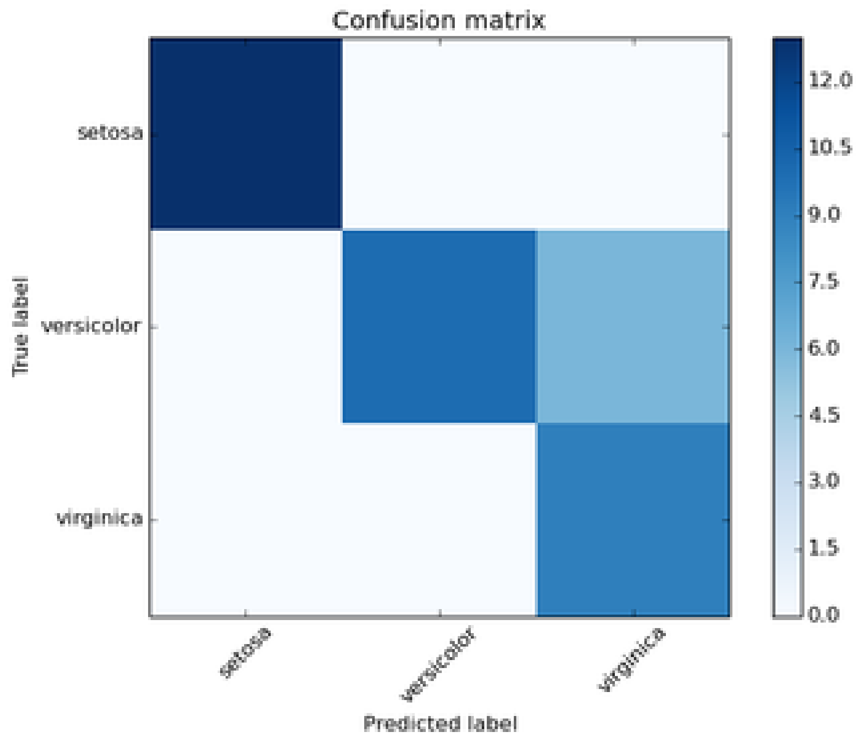


Figure 42.452: *Confusion matrix*

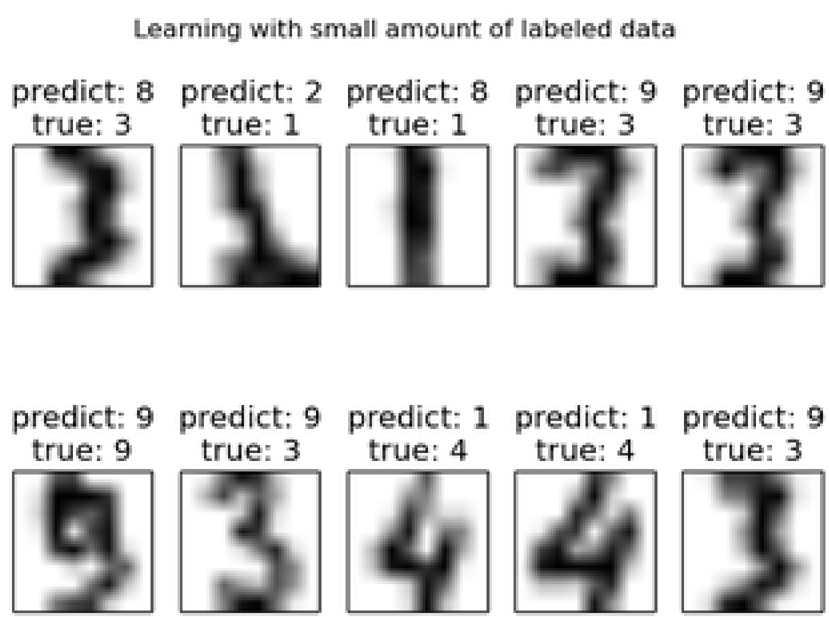


Figure 42.453: *Label Propagation digits: Demonstrating performance*

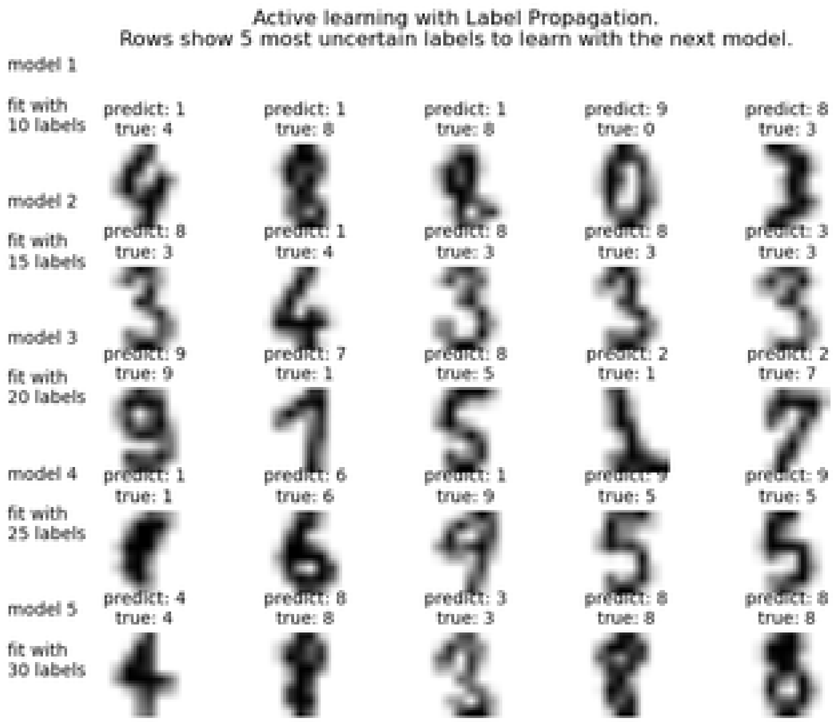


Figure 42.454: *Label Propagation digits active learning*

Figure 42.455: *Classification of text documents: using a MLComp dataset*

Figure 42.456: *Classification of text documents using sparse features*

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

Integer array of labels.

pos_label : str or int, 1 by default

The class to report if `average='binary'`. Until version 0.18 it is necessary to set `pos_label=None` if seeking to use another averaging method over binary targets.

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`).

Note that if `pos_label` is given in binary classification with *average != 'binary'*, only that positive class is reported. This behavior is deprecated and will change in version 0.18.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

Returnsf1_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.

References

[R161]

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. ])
```

Examples using `sklearn.metrics.f1_score`

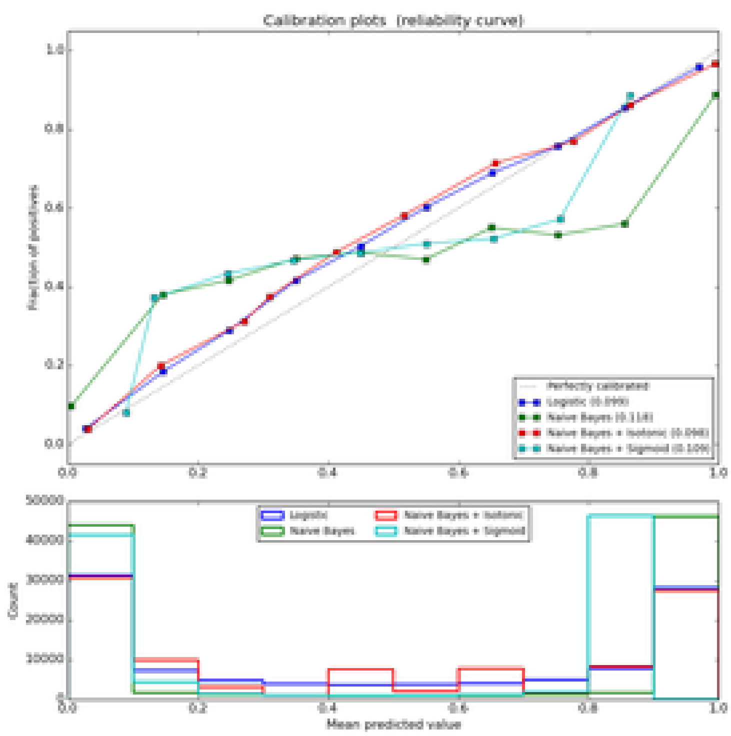


Figure 42.457: *Probability Calibration curves*

sklearn.metrics.fbeta_score

`sklearn.metrics.fbeta_score`(*y_true*, *y_pred*, *beta*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*)

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 precision 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).

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 :

Weight of precision in harmonic mean.

labels : array

Integer array of labels.

pos_label : str or int, 1 by default

The class to report if `average='binary'`. Until version 0.18 it is necessary to set `pos_label=None` if seeking to use another averaging method over binary targets.

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`).

Note that if `pos_label` is given in binary classification with *average != 'binary'*, only that positive class is reported. This behavior is deprecated and will change in version 0.18.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

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.

References

[R162], [R163]

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*, *classes=None*)

Compute the average Hamming loss.

The Hamming loss is the fraction of labels that are incorrectly predicted.

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.

classes : array, shape = [n_labels], optional

Integer array of labels.

Returns loss : float or int,

Return the average Hamming loss between element of `y_true` and `y_pred`.

See also:

`accuracy_score`, `jaccard_similarity_score`, `zero_one_loss`

Notes

In multiclass classification, the Hamming loss correspond to the Hamming distance between `y_true` and `y_pred` which is equivalent to the subset `zero_one_loss` function.

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 entirely match the true set of labels. Hamming loss is more forgiving in that it penalizes the individual labels.

The Hamming loss is upperbounded by the subset zero-one loss. When normalized over samples, the Hamming loss is always between 0 and 1.

References

[R164], [R165]

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:

```
>>> hamming_loss(np.array([[0, 1], [1, 1]]), np.zeros((2, 2)))
0.75
```

Examples using `sklearn.metrics.hamming_loss`

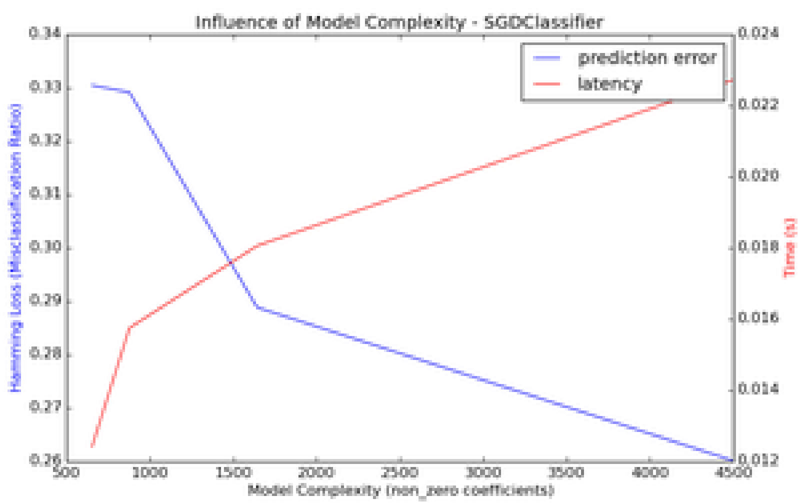


Figure 42.458: *Model Complexity Influence*

`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.

Parametersy_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], optional

Sample weights.

Returnsloss : float

References

[R166], [R167], [R168]

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(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=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:

```
>>> 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(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
>>> 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_similarity_score

sklearn.metrics.**jaccard_similarity_score**(*y_true*, *y_pred*, *normalize=True*, *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`.

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], optional

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

[R170]

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import jaccard_similarity_score
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]
>>> jaccard_similarity_score(y_true, y_pred)
0.5
>>> jaccard_similarity_score(y_true, y_pred, normalize=False)
2
```

In the multilabel case with binary label indicators:

```
>>> jaccard_similarity_score(np.array([[0, 1], [1, 1]]),        np.ones((2, 2)))
0.75
```

sklearn.metrics.log_loss

sklearn.metrics.**log_loss**(*y_true*, *y_pred*, *eps=1e-15*, *normalize=True*, *sample_weight=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. For a single sample with true label yt in {0,1} and estimated probability yp that yt = 1, the log loss is

-log P(yt|yp) = -(yt log(yp) + (1 - yt) log(1 - yp))

Parametersy_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)

Predicted probabilities, as returned by a classifier's predict_proba method.

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], optional

Sample weights.

Returnsloss : 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

```
>>> log_loss(["spam", "ham", "ham", "spam"],
...          [[.1, .9], [.9, .1], [.8, .2], [.35, .65]])
0.21616...
```

Examples using `sklearn.metrics.log_loss`

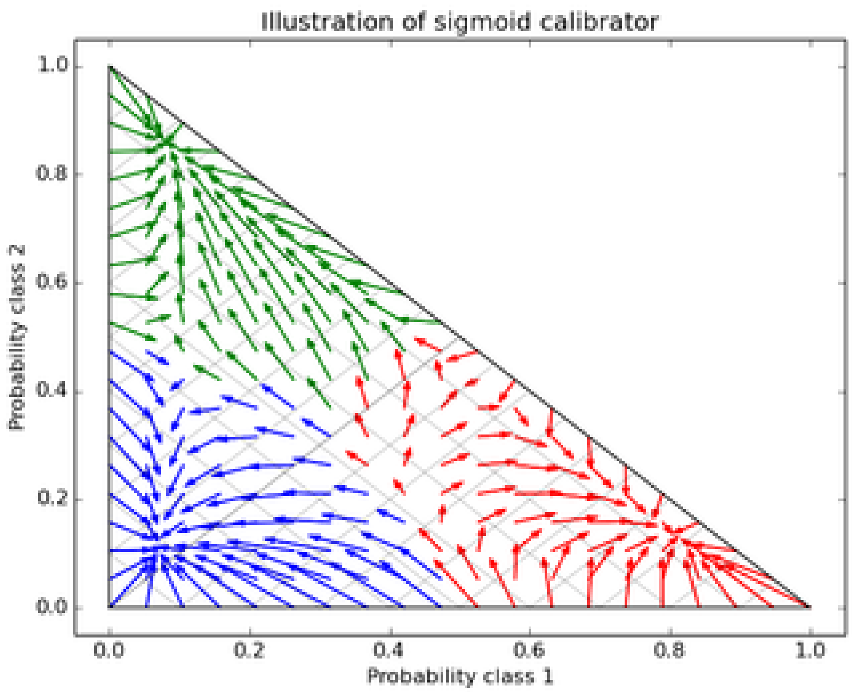


Figure 42.459: *Probability Calibration for 3-class classification*

`sklearn.metrics`.matthews_corrcoef

`sklearn.metrics.matthews_corrcoef(y_true, y_pred)`

Compute the Matthews correlation coefficient (MCC) for binary classes

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. [source: Wikipedia]

Only in the binary case does this relate to information about true and false positives and negatives. See references below.

Parametersy_true : array, shape = [n_samples]

Ground truth (correct) target values.

y_pred : array, shape = [n_samples]

Estimated targets as returned by a classifier.

Returnsmcc : float

The Matthews correlation coefficient (+1 represents a perfect prediction, 0 an average random prediction and -1 and inverse prediction).

References

[R171], [R172]

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.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 x axis.

Parametersy_true : array, shape = [n_samples]

True targets of binary classification in range {-1, 1} or {0, 1}.

probas_pred : array, shape = [n_samples]

Estimated probabilities or decision function.

pos_label : int, optional (default=None)

The label of the positive class

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

Returnsprecision : 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.

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([ 0.66...,  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`

`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*)

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'`.

Parametersy_true : 1d array-like, or label indicator array / sparse matrix

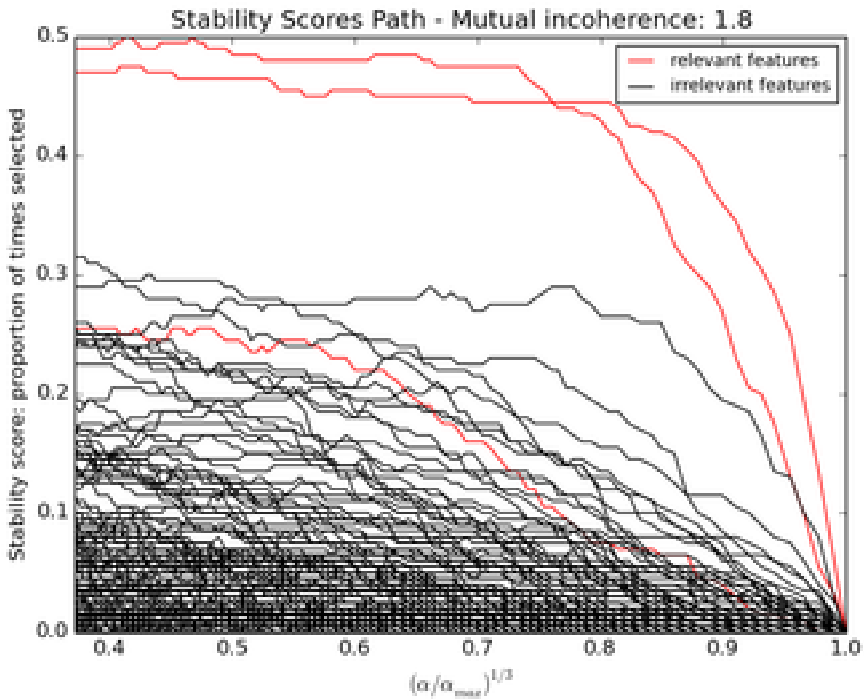


Figure 42.460: *Sparse recovery: feature selection for sparse linear models*

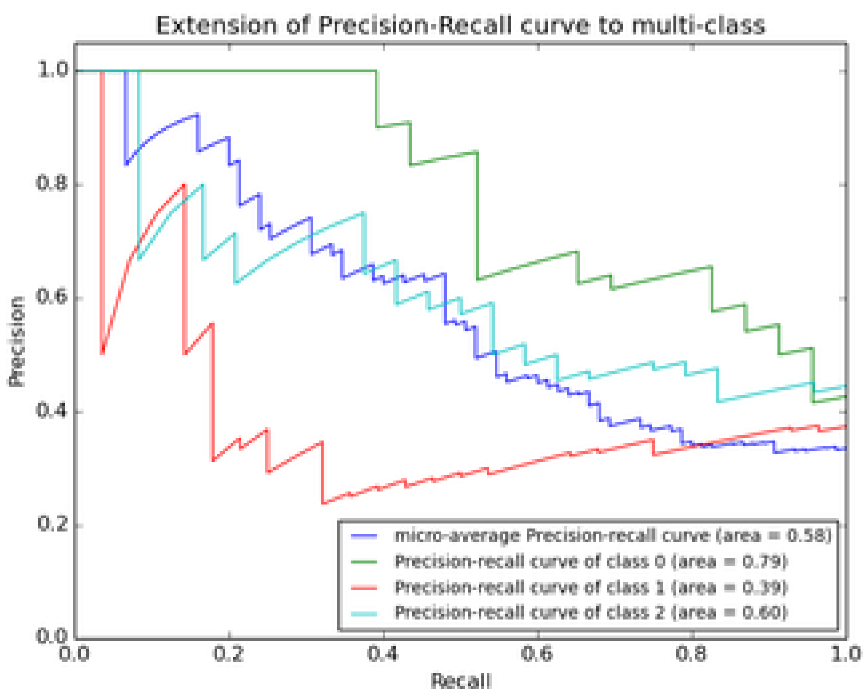


Figure 42.461: *Precision-Recall*

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 : array

Integer array of labels.

pos_label : str or int, 1 by default

The class to report if `average='binary'`. Until version 0.18 it is necessary to set `pos_label=None` if seeking to use another averaging method over binary targets.

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`).

Note that if `pos_label` is given in binary classification with *average != 'binary'*, only that positive class is reported. This behavior is deprecated and will change in version 0.18.

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], optional

Sample weights.

Returnsprecision: 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: int (if average is not None) or array of int, shape = [n_unique_labels] :

The number of occurrences of each label in `y_true`.

References

[R173], [R174], [R175]

Examples

```
>>> from sklearn.metrics import precision_recall_fscore_support
>>> y_true = np.array([0, 1, 2, 0, 1, 2])
>>> y_pred = np.array([0, 2, 1, 0, 0, 1])
>>> 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)
```

sklearn.metrics.precision_score

sklearn.metrics.**precision_score**(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*)

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.

Parametersy_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

Integer array of labels.

pos_label : str or int, 1 by default

The class to report if `average='binary'`. Until version 0.18 it is necessary to set `pos_label=None` if seeking to use another averaging method over binary targets.

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`).

Note that if `pos_label` is given in binary classification with *average != 'binary'*, only that positive class is reported. This behavior is deprecated and will change in version 0.18.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

Returnsprecision : 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.

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.        ])
```

Examples using `sklearn.metrics.precision_score`

`sklearn.metrics`.recall_score

`sklearn.metrics.`**`recall_score`**(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*)

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.

Parametersy_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

Integer array of labels.

pos_label : str or int, 1 by default

The class to report if `average='binary'`. Until version 0.18 it is necessary to set `pos_label=None` if seeking to use another averaging method over binary targets.

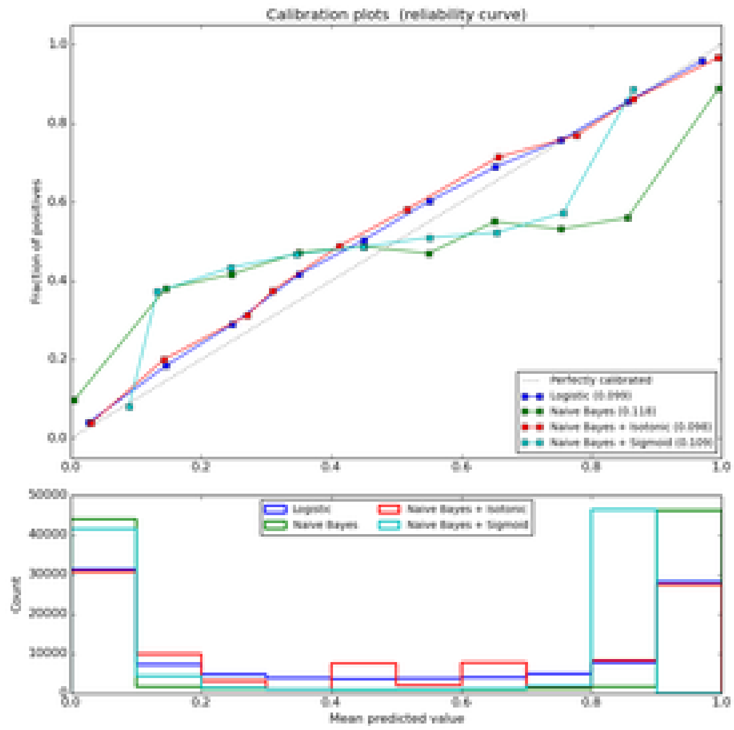


Figure 42.462: *Probability Calibration curves*

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`).

Note that if `pos_label` is given in binary classification with *average != 'binary'*, only that positive class is reported. This behavior is deprecated and will change in version 0.18.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

Returnsrecall : 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.

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.])
```

Examples using `sklearn.metrics.recall_score`

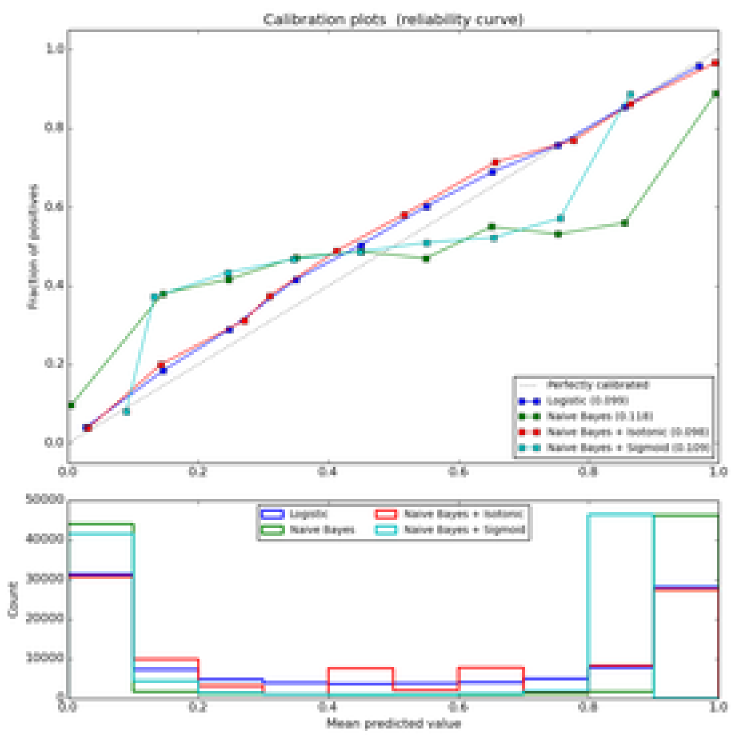


Figure 42.463: *Probability Calibration curves*

`sklearn.metrics`.roc_auc_score

`sklearn.metrics.`**`roc_auc_score`**(*y_true*, *y_score*, *average='macro'*, *sample_weight=None*)
Compute Area Under the Curve (AUC) from prediction scores

Note: this implementation is restricted to the binary classification task or multilabel classification task in label indicator format.

Parametersy_true : array, shape = [n_samples] or [n_samples, n_classes]

True binary labels in 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 binary decisions.

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.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

Returnsauc : float

See also:

`average_precision_score`Area under the precision-recall curve

`roc_curve`Compute Receiver operating characteristic (ROC)

References

[R177]

Examples

```
>>> 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
```

sklearn.metrics.roc_curve

sklearn.metrics.**roc_curve**(*y_true*, *y_score*, *pos_label=None*, *sample_weight=None*)

Compute Receiver operating characteristic (ROC)

Note: this implementation is restricted to the binary classification task.

Parametersy_true : array, shape = [n_samples]

True binary labels in range {0, 1} or {-1, 1}. If labels are not binary, pos_label should be explicitly given.

y_score : array, shape = [n_samples]

Target scores, can either be probability estimates of the positive class or confidence values.

pos_label : int

Label considered as positive and others are considered negative.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

Returnsfpr : 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 Area Under the Curve (AUC) from prediction scores

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

[R178]

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. ,  0.5,  0.5,  1. ])
>>> tpr
array([ 0.5,  0.5,  1. ,  1. ])
```

```
>>> thresholds
array([ 0.8 ,  0.4 ,  0.35,  0.1 ])
```

Examples using `sklearn.metrics.roc_curve`

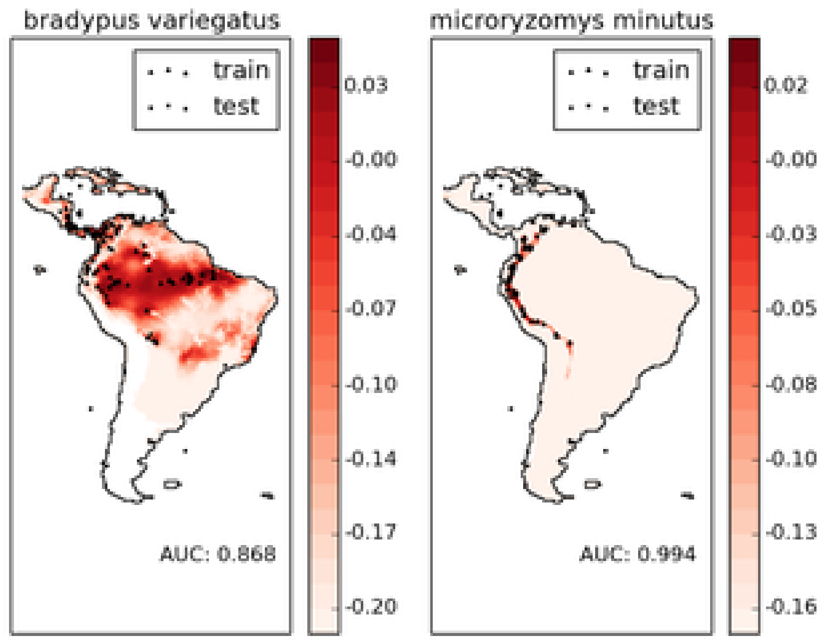


Figure 42.464: *Species distribution modeling*

`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.

Parametersy_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], optional

Sample weights.

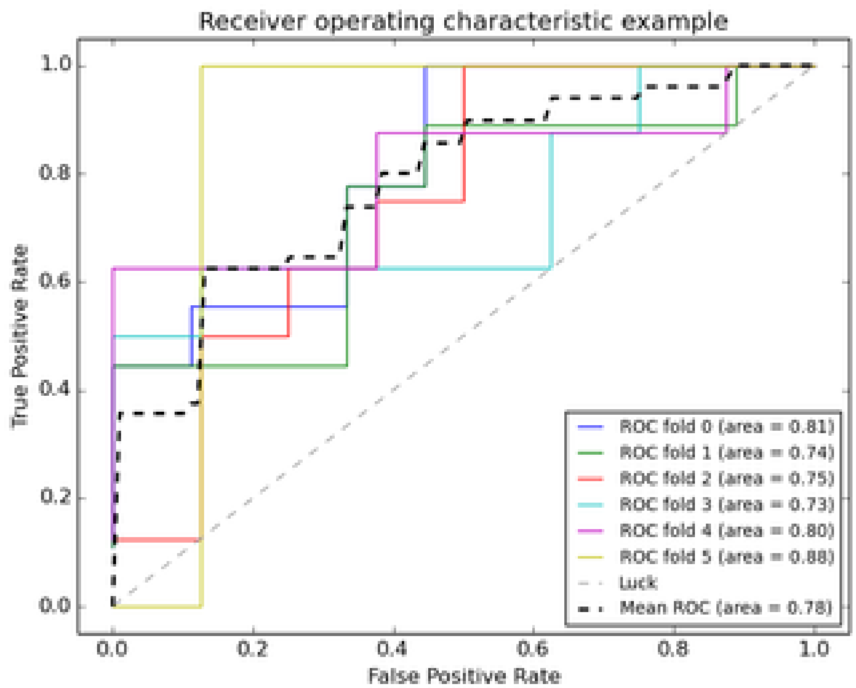


Figure 42.465: *Receiver Operating Characteristic (ROC) with cross validation*

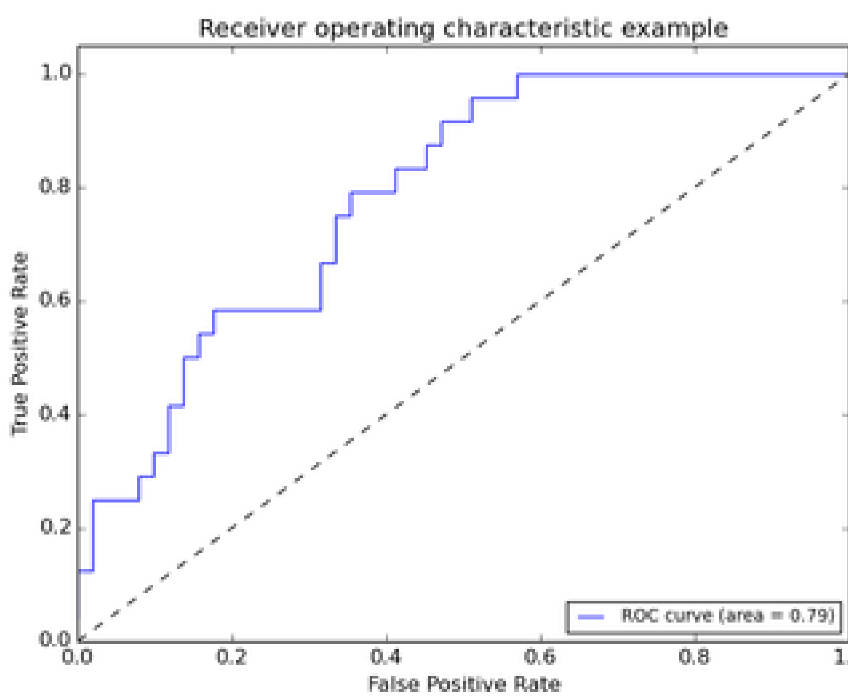


Figure 42.466: *Receiver Operating Characteristic (ROC)*

Returnsloss : 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_similarity_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:

```
>>> zero_one_loss(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5
```

Examples using `sklearn.metrics.zero_one_loss`

`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 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.

Parametersy_true : array, shape (n_samples,)

True targets.

y_prob : array, shape (n_samples,)

Probabilities of the positive class.

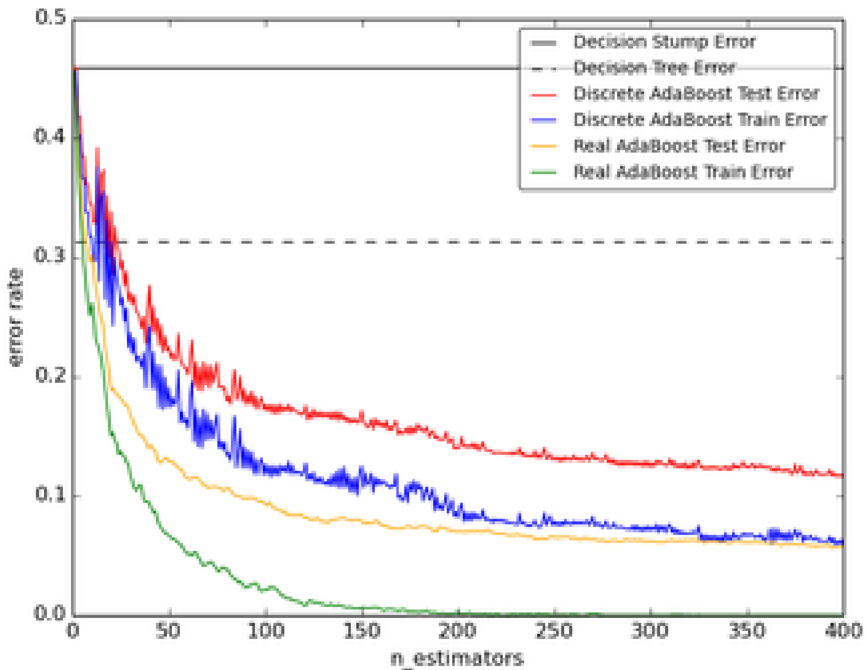


Figure 42.467: *Discrete versus Real AdaBoost*

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

pos_label : int (default: None)

Label of the positive class. If None, the maximum label is used as positive class

Returnsscore : float

Brier score

References

http://en.wikipedia.org/wiki/Brier_score

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`

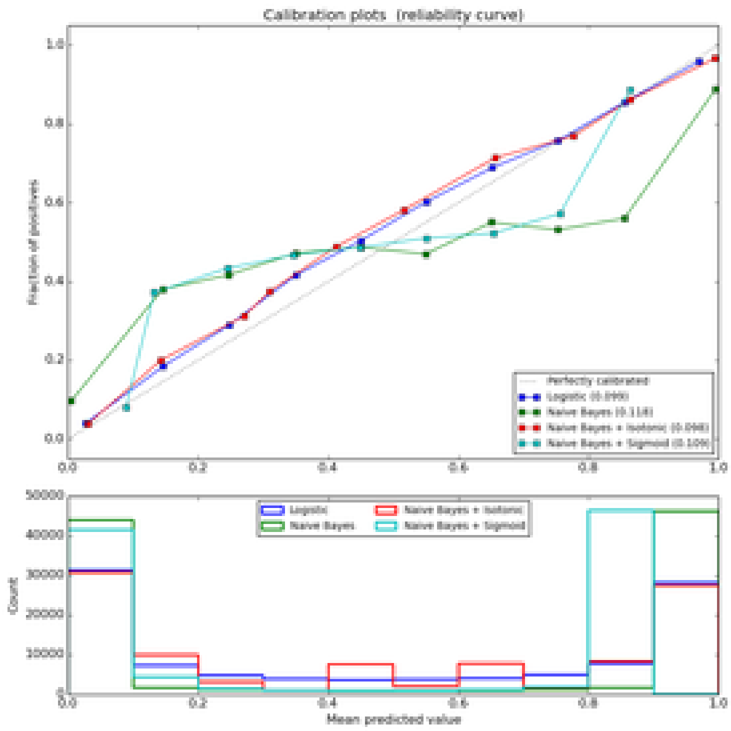


Figure 42.468: *Probability Calibration curves*

42.21.3 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.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.median_absolute_error`(y_true, y_pred)	Median absolute error regression loss
`metrics.r2_score`(y_true, y_pred[, sample_weight])	R^2 (coefficient of determination) regression score function.

`sklearn.metrics`.explained_variance_score

sklearn.metrics.**explained_variance_score**(*y_true*, *y_pred*, *sample_weight=None*)

Explained variance regression score function

Best possible score is 1.0, lower values are worse.

Parametersy_true : array-like

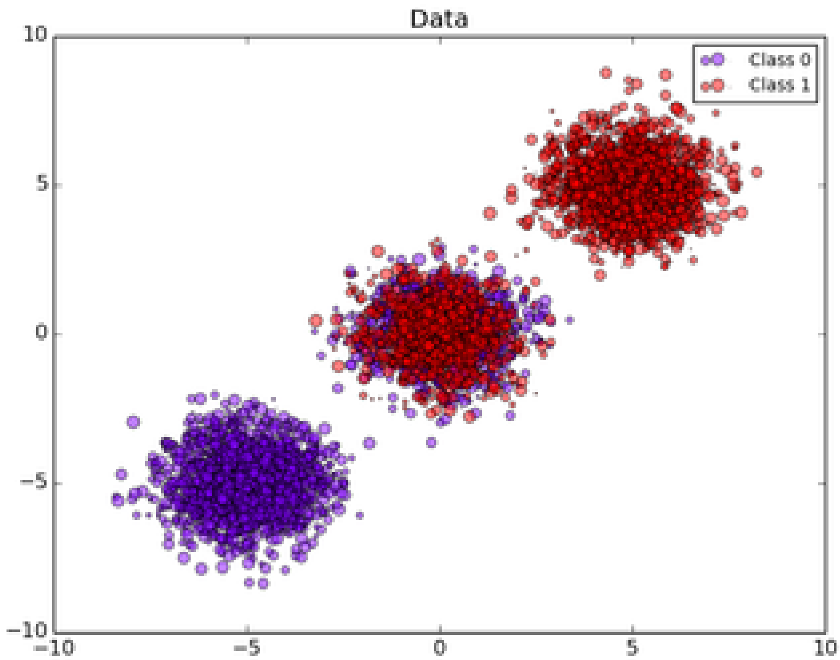


Figure 42.469: *Probability calibration of classifiers*

Ground truth (correct) target values.

y_pred : array-like

Estimated target values.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

Returnsscore : float

The explained variance.

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...
```

sklearn.metrics.mean_absolute_error

sklearn.metrics.**mean_absolute_error**(*y_true*, *y_pred*, *sample_weight=None*)

Mean absolute error regression loss

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.

Returns

loss : float

A positive floating point value (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
```

sklearn.metrics.mean_squared_error

sklearn.metrics.**mean_squared_error**(*y_true*, *y_pred*, *sample_weight=None*)

Mean squared error regression loss

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.

Returns

loss : float

A positive floating point value (the best value is 0.0).

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 = [[0.5, 1],[-1, 1],[7, -6]]
>>> y_pred = [[0, 2],[-1, 2],[8, -5]]
>>> mean_squared_error(y_true, y_pred)
0.708...
```

Examples using `sklearn.metrics.mean_squared_error`

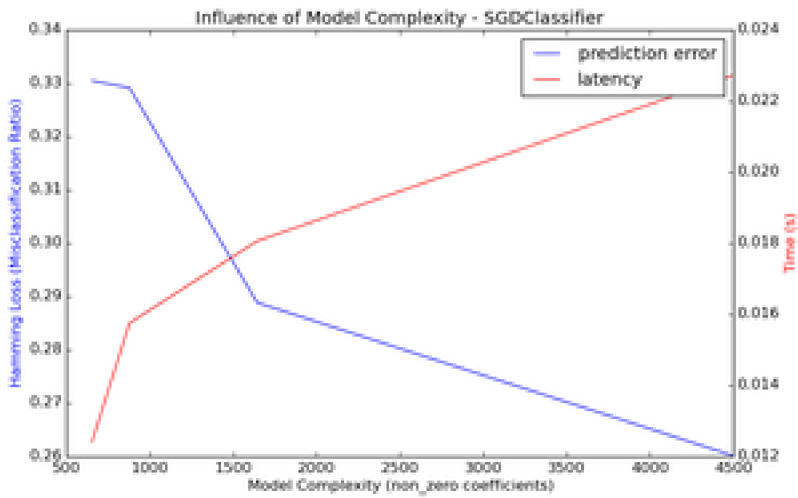


Figure 42.470: *Model Complexity Influence*

`sklearn.metrics`.median_absolute_error

`sklearn.metrics.median_absolute_error`(*y_true*, *y_pred*)

Median absolute error regression loss

Parametersy_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.

Returnsloss : float

A positive floating point value (the best value is 0.0).

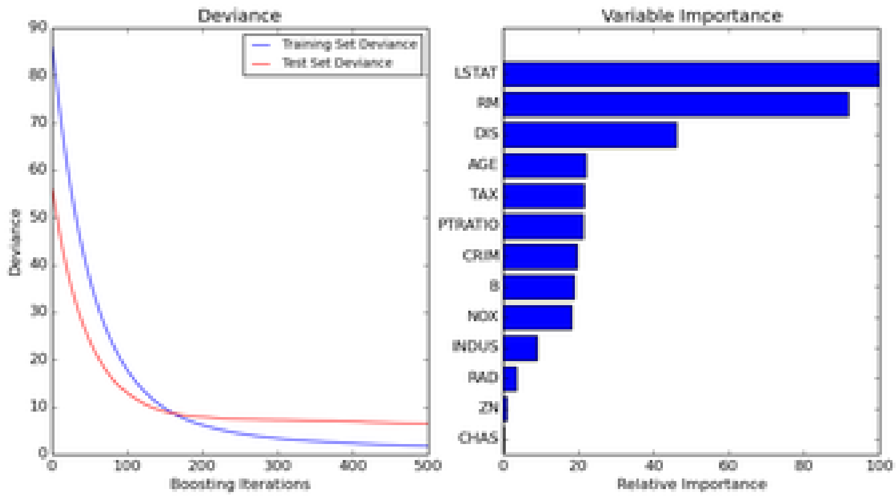


Figure 42.471: *Gradient Boosting regression*

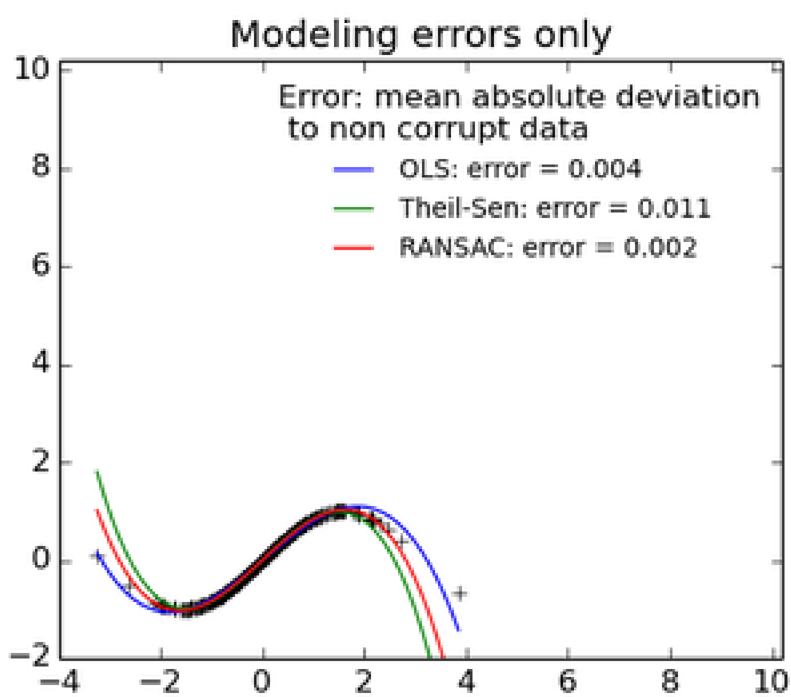


Figure 42.472: *Robust linear estimator fitting*

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
```

sklearn.metrics.r2_score

sklearn.metrics.**r2_score**(*y_true*, *y_pred*, *sample_weight=None*)

R^2 (coefficient of determination) regression score function.

Best possible score is 1.0, lower values are worse.

Parametersy_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.

Returnsz : float

The R^2 score.

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).

References

[R176]

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)
0.938...
```

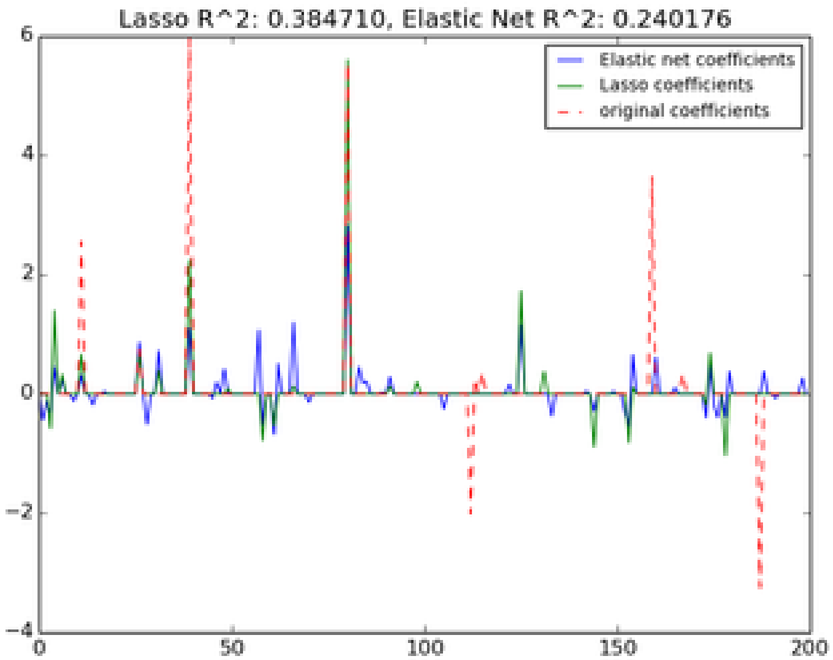


Figure 42.473: *Lasso and Elastic Net for Sparse Signals*

Examples using `sklearn.metrics.r2_score`

42.21.4 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`(...)	Compute ranking-based average precision

`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.

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 binary decisions.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

Returns **coverage_error** : float

References

Returnsscore : 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...
```

42.21.5 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.completeness_score`(labels_true, ...)	Completeness metric of a cluster labeling given a ground truth
`metrics.homogeneity_completeness_v_measure`(...)	Compute the homogeneity and completeness and V-Measure sco
`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
`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*)

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))] / [max(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.

Parameterslabels_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.

Returnsami: 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_information_score` Mutual Information (not adjusted for chance)

References

[R155], [R156]

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`

`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:

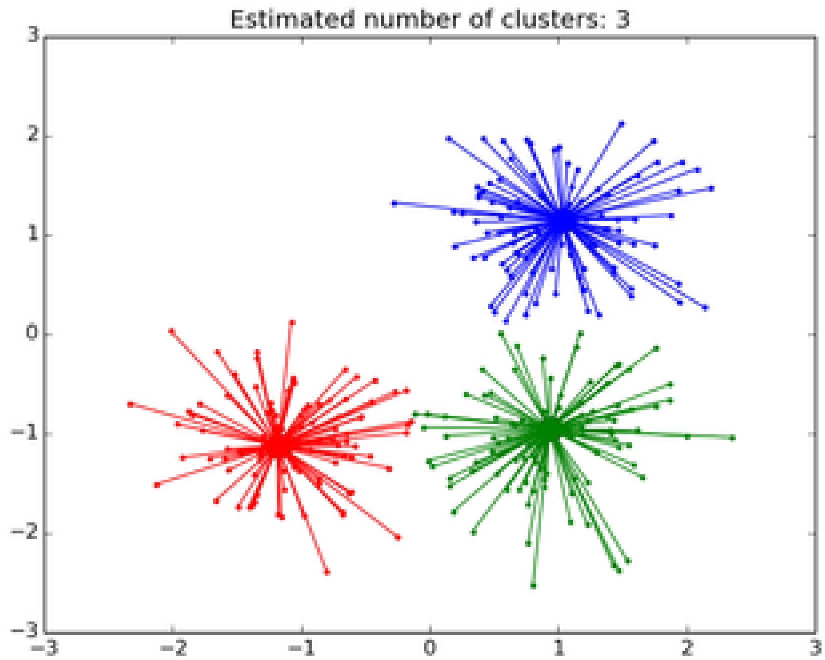


Figure 42.474: *Demo of affinity propagation clustering algorithm*

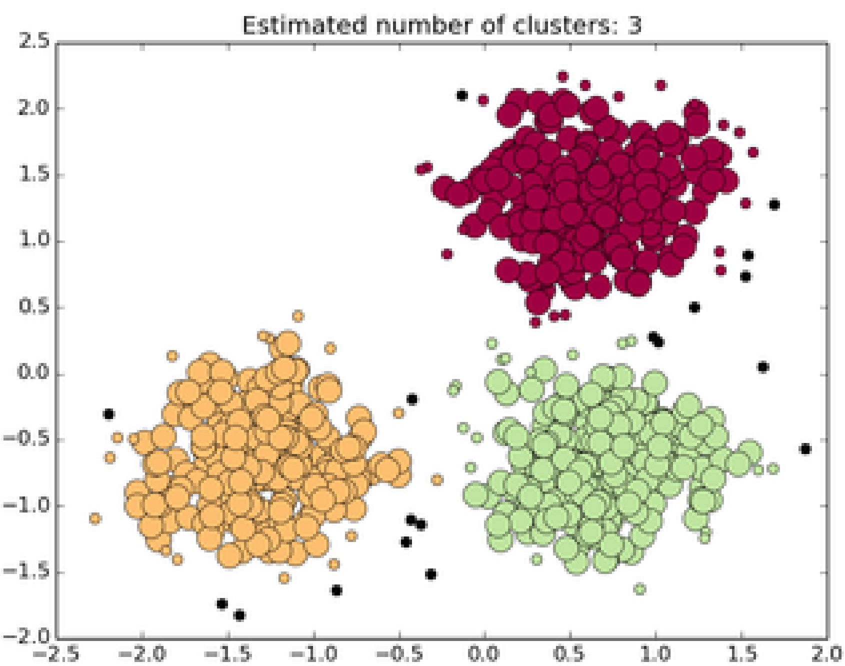


Figure 42.475: *Demo of DBSCAN clustering algorithm*

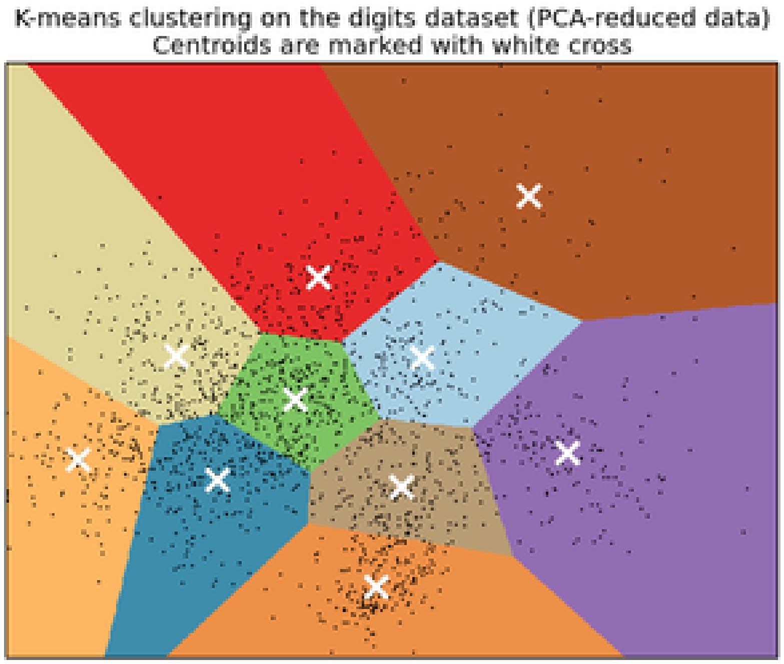


Figure 42.476: *A demo of K-Means clustering on the handwritten digits data*

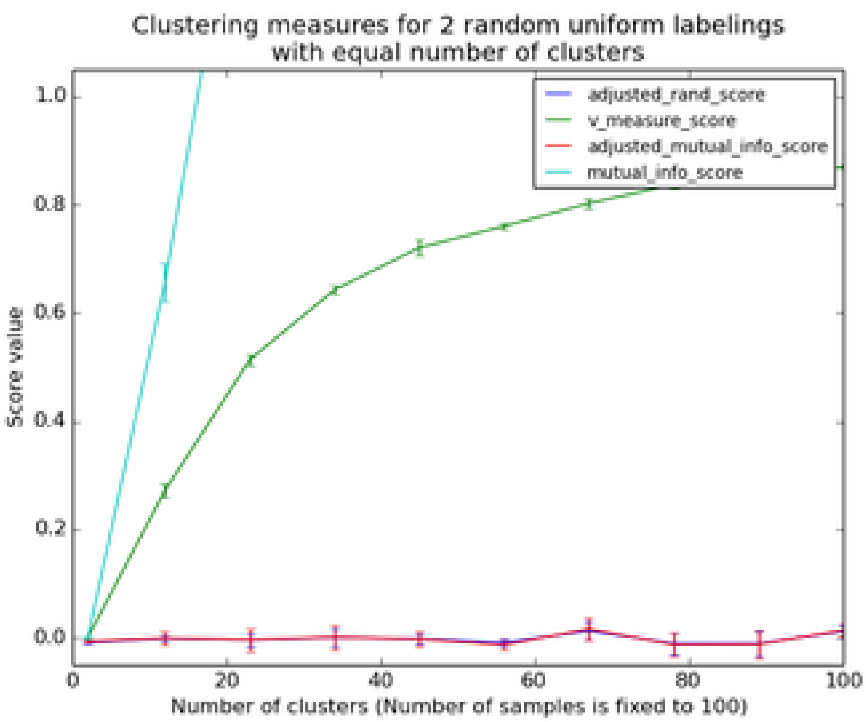


Figure 42.477: *Adjustment for chance in clustering performance evaluation*

```
adjusted_rand_score(a, b) == adjusted_rand_score(b, a)
```

Parameterslabels_true : int array, shape = [n_samples]

Ground truth class labels to be used as a reference

labels_pred : array, shape = [n_samples]

Cluster labels to evaluate

Returnsari: 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 maching 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

sklearn.metrics.completeness_score

sklearn.metrics.**completeness_score**(*labels_true*, *labels_pred*)

Completeness metric of a cluster labeling given a ground truth

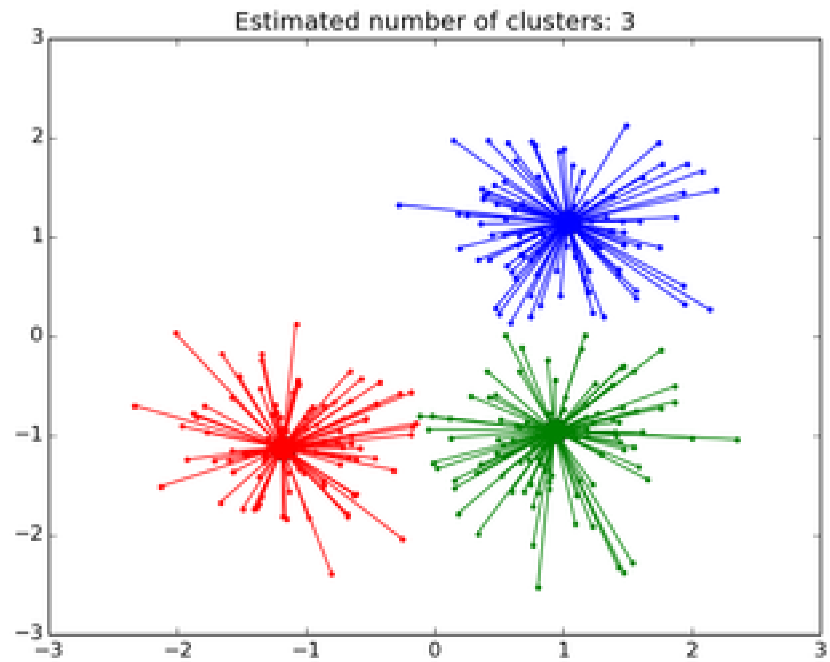


Figure 42.478: *Demo of affinity propagation clustering algorithm*

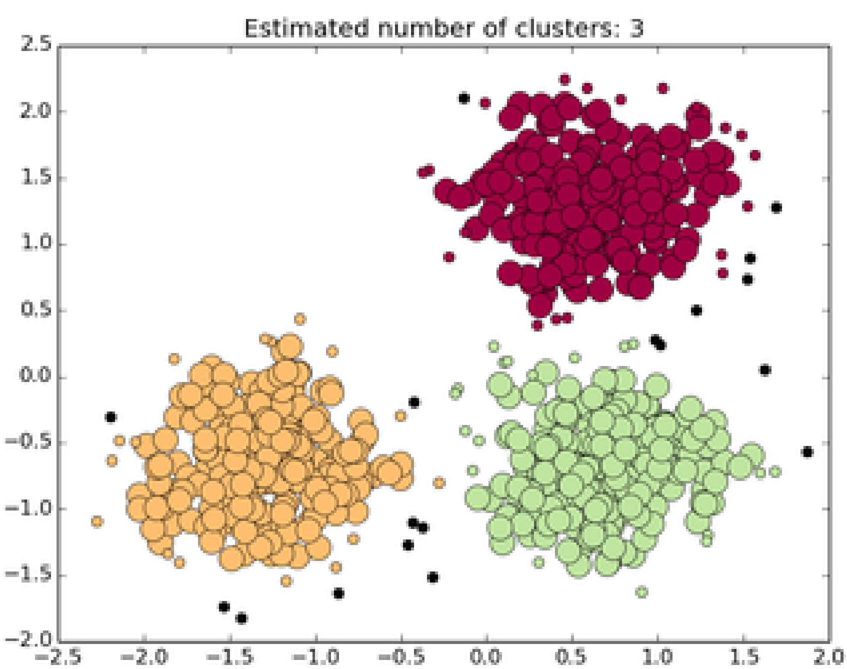


Figure 42.479: *Demo of DBSCAN clustering algorithm*

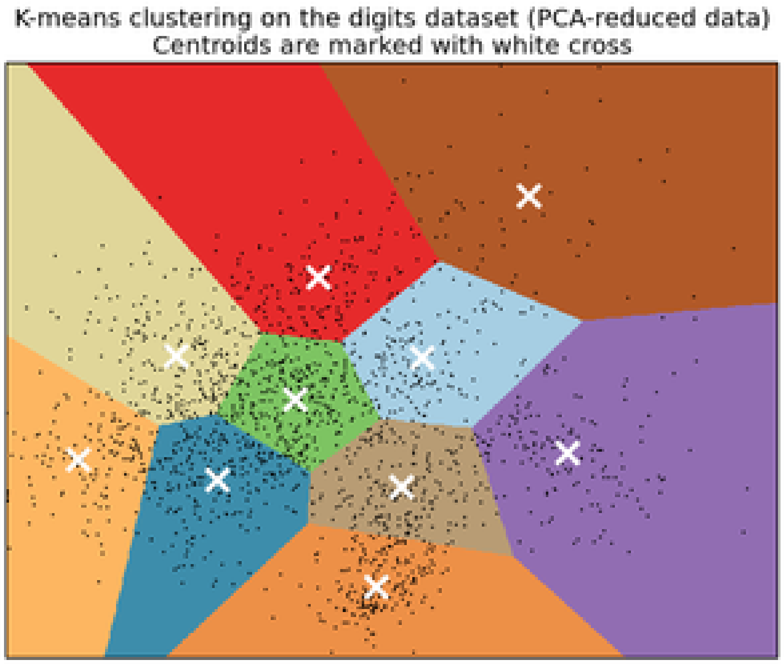


Figure 42.480: *A demo of K-Means clustering on the handwritten digits data*

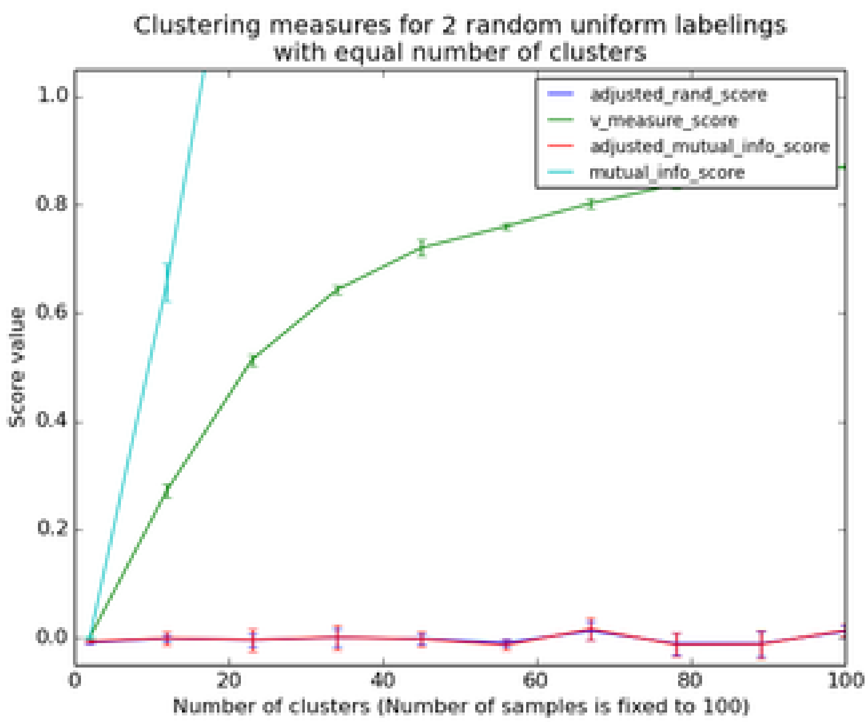


Figure 42.481: *Adjustment for chance in clustering performance evaluation*

Figure 42.482: *Clustering text documents using k-means*

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.

Parameterslabels_true : int array, shape = [n_samples]

ground truth class labels to be used as a reference

labels_pred : array, shape = [n_samples]

cluster labels to evaluate

Returnscompleteness: float :

score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

See also:

`homogeneity_score`, `v_measure_score`

References

[R158]

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]))
1.0
```

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`

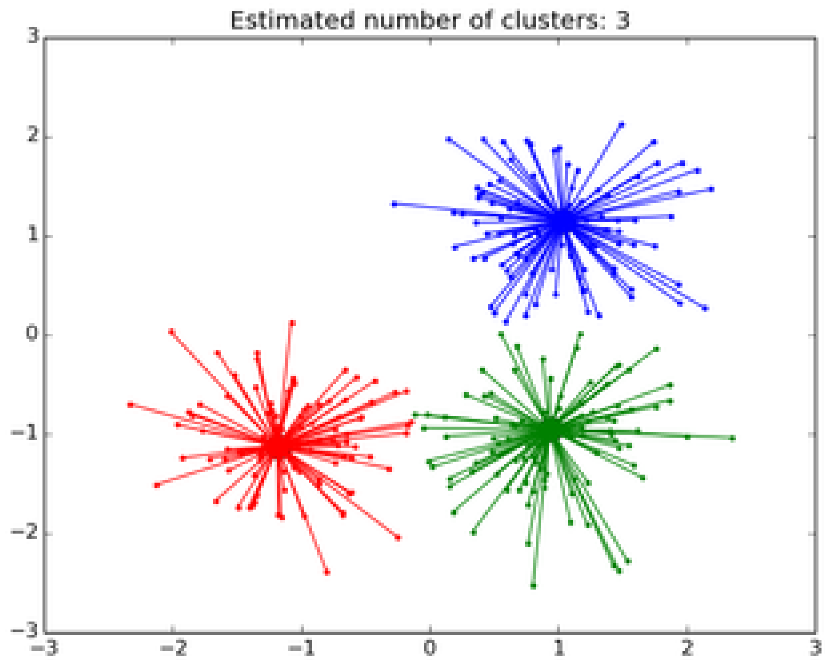


Figure 42.483: *Demo of affinity propagation clustering algorithm*

`sklearn.metrics`.homogeneity_completeness_v_measure

sklearn.metrics.**homogeneity_completeness_v_measure**(*labels_true*, *labels_pred*)

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.

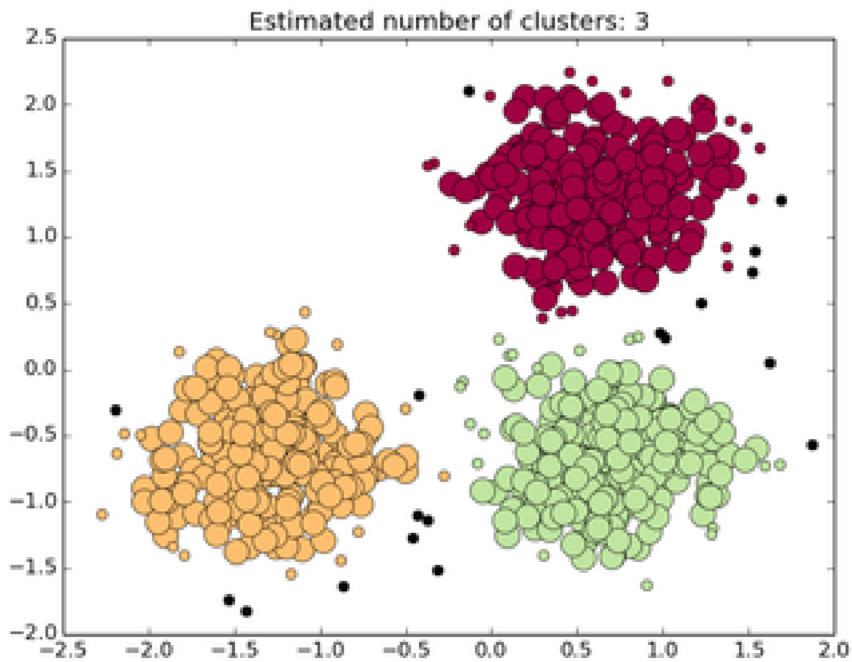


Figure 42.484: *Demo of DBSCAN clustering algorithm*

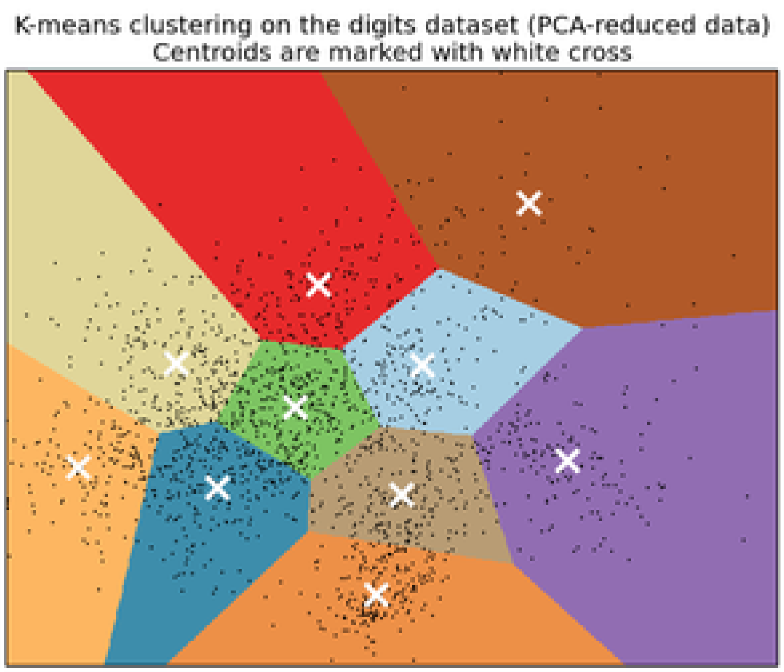


Figure 42.485: *A demo of K-Means clustering on the handwritten digits data*

Figure 42.486: *Clustering text documents using k-means*

V-Measure is furthermore symmetric: swapping `labels_true` and `label_pred` will give the same score. This does not hold for homogeneity and completeness.

Parameterslabels_true : int array, shape = [n_samples]

ground truth class labels to be used as a reference

labels_pred : array, shape = [n_samples]

cluster labels to evaluate

Returnshomogeneity: 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.

Parameterslabels_true : int array, shape = [n_samples]

ground truth class labels to be used as a reference

labels_pred : array, shape = [n_samples] cluster labels to evaluate

Returnshomogeneity: float :

score between 0.0 and 1.0. 1.0 stands for perfectly homogeneous labeling

See also:

completeness_score, v_measure_score

References

[R169]

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]))
...
1.0...
>>> print("%.6f" % homogeneity_score([0, 0, 1, 1], [0, 1, 2, 3]))
...
1.0...
```

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`

`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 $P(i)$ is the probability of a random sample occurring in cluster U_i and $P'(j)$ is the probability of a random sample occurring in cluster V_j, the Mutual Information between clusterings U and V is given as:

$$MI(U,V) = \sum_{i=1}^{R} \sum_{j=1}^{C} P(i,j) \log \frac{P(i,j)}{P(i)P'(j)}$$

This is equal to the Kullback-Leibler divergence of the joint distribution with the product distribution of the marginals.

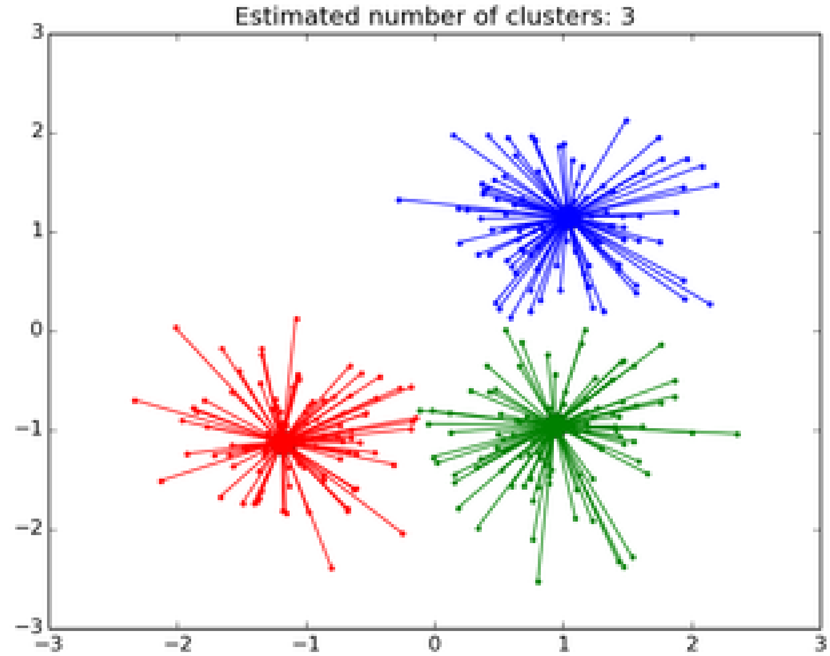


Figure 42.487: *Demo of affinity propagation clustering algorithm*

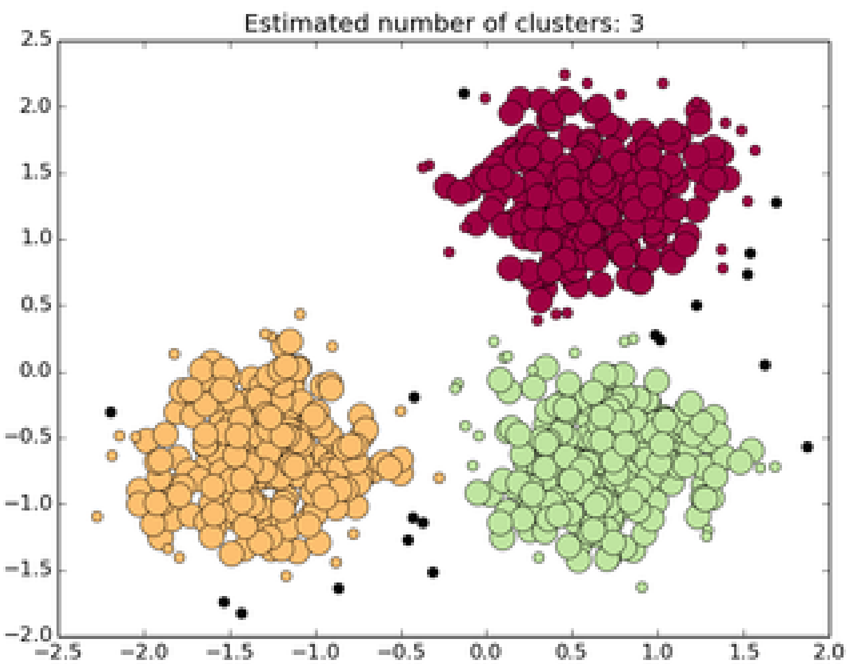


Figure 42.488: *Demo of DBSCAN clustering algorithm*

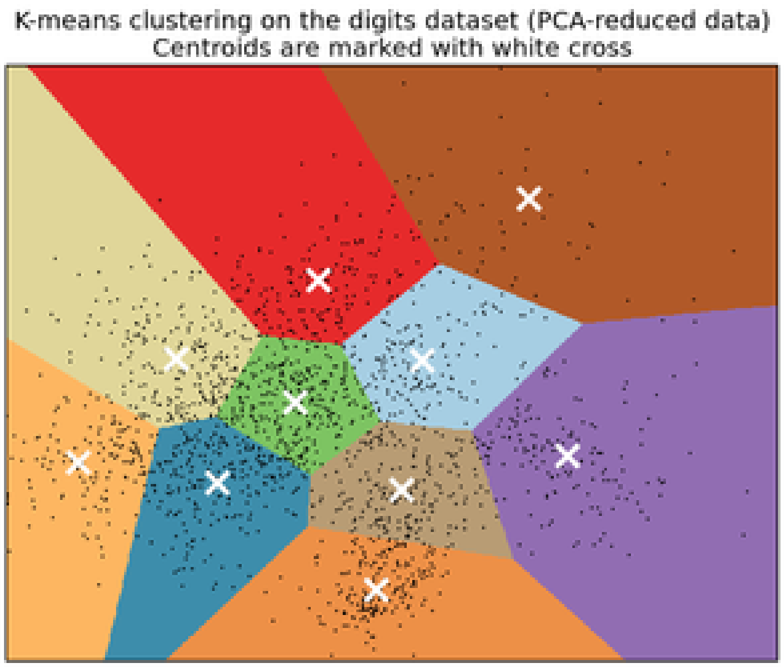


Figure 42.489: *A demo of K-Means clustering on the handwritten digits data*

Figure 42.490: *Clustering text documents using k-means*

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.

Parameterslabels_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.

contingency: None or array, 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.

Returnsmi: 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

Examples using `sklearn.metrics.mutual_info_score`

`sklearn.metrics`.normalized_mutual_info_score

`sklearn.metrics.`**`normalized_mutual_info_score`**(*labels_true*, *labels_pred*)

Normalized Mutual Information between two clusterings

Normalized Mutual Information (NMI) is an 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 `sqrt(H(labels_true) * H(labels_pred))`

This measure is not adjusted for chance. Therefore `adjusted_mustual_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.

Parameterslabels_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.

Returnsnmi: float :

score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

See also:

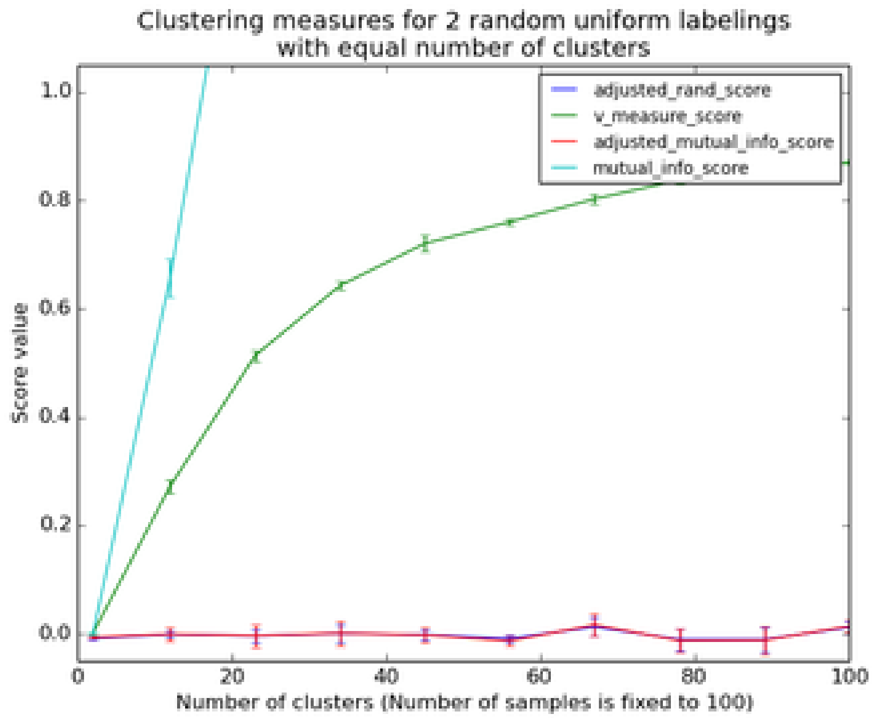


Figure 42.491: *Adjustment for chance in clustering performance evaluation*

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 Coefficent 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.

ParametersX : 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. If `sample_size is None`, no sampling is used.

random_state : integer or numpy.RandomState, optional

The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

'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.

Returnssilhouette : float

Mean Silhouette Coefficient for all samples.

References

[R181], [R182]

Examples using `sklearn.metrics.silhouette_score`

`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.

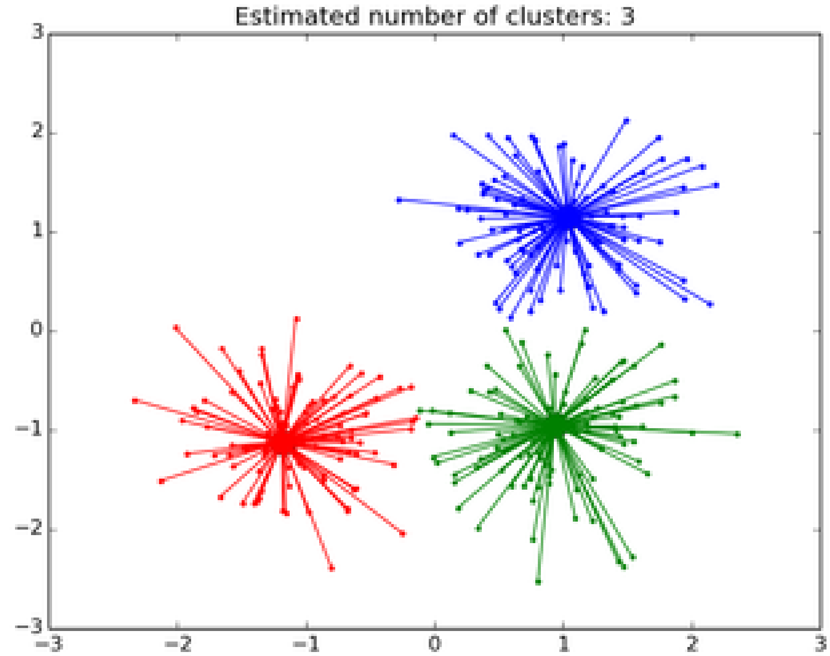


Figure 42.492: *Demo of affinity propagation clustering algorithm*

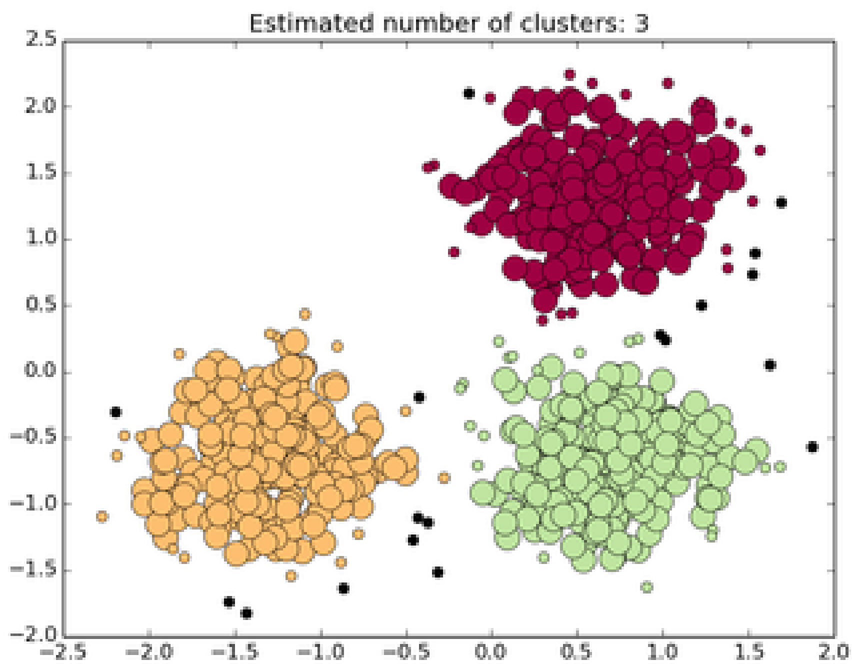


Figure 42.493: *Demo of DBSCAN clustering algorithm*

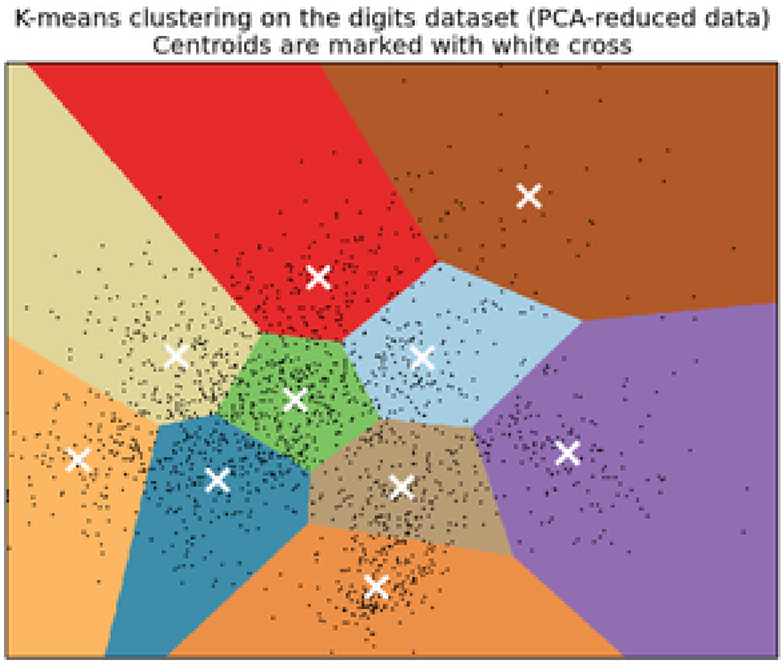


Figure 42.494: *A demo of K-Means clustering on the handwritten digits data*

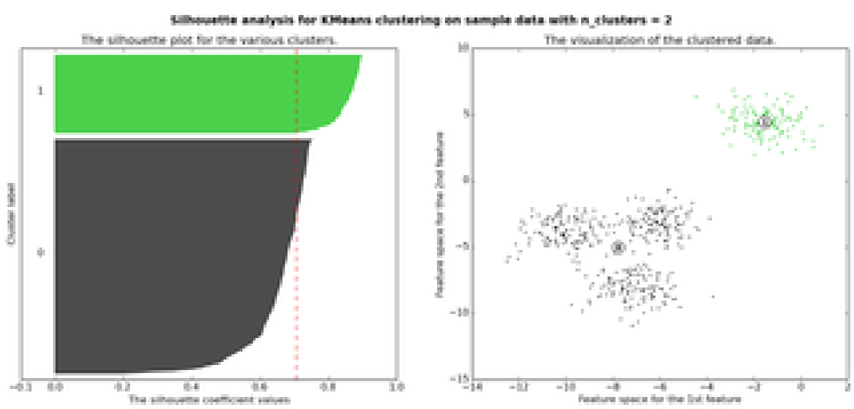


Figure 42.495: *Selecting the number of clusters with silhouette analysis on KMeans clustering*

Figure 42.496: *Clustering text documents using k-means*

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 Coefficent 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.

ParametersX : 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.

'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.

Returnssilhouette : array, shape = [n_samples]

Silhouette Coefficient for each samples.

References

[R179], [R180]

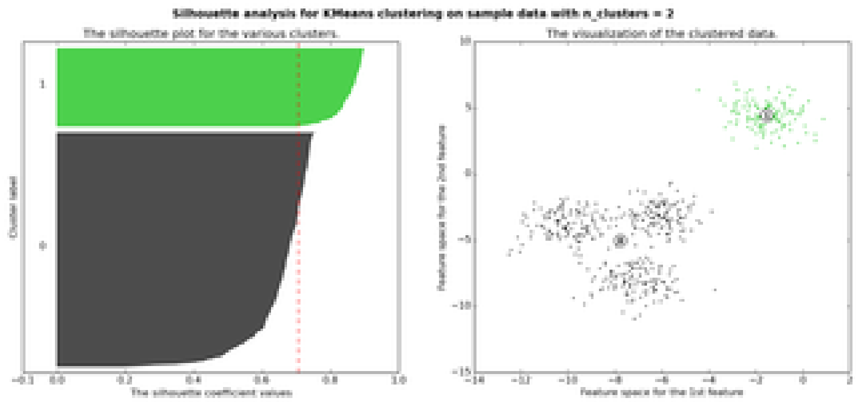


Figure 42.497: *Selecting the number of clusters with silhouette analysis on KMeans clustering*

Examples using `sklearn.metrics.silhouette_samples`

`sklearn.metrics`.v_measure_score

`sklearn.metrics.`**`v_measure_score`**(*labels_true*, *labels_pred*)

V-measure cluster labeling given a ground truth.

This score is identical to `normalized_mutual_info_score`.

The V-measure is the harmonic mean between homogeneity and completeness:

```
v = 2 * (homogeneity * completeness) / (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.

Parameterslabels_true : int array, shape = [n_samples]

ground truth class labels to be used as a reference

labels_pred : array, shape = [n_samples]

cluster labels to evaluate

Returnsv_measure: float :

score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

See also:

homogeneity_score, completeness_score

References

[R183]

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...
```

Figure 42.498: *Biclustering documents with the Spectral Co-clustering algorithm*

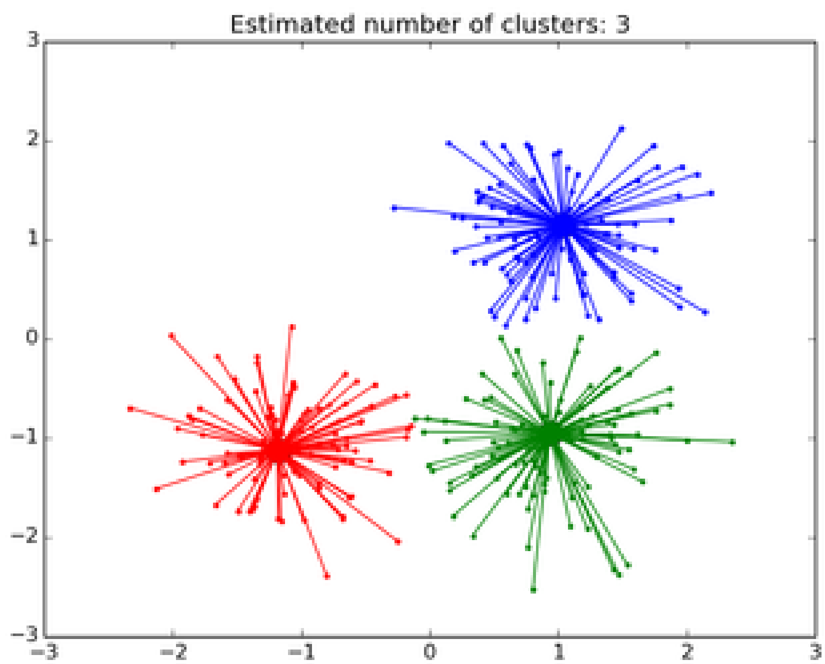


Figure 42.499: *Demo of affinity propagation clustering algorithm*

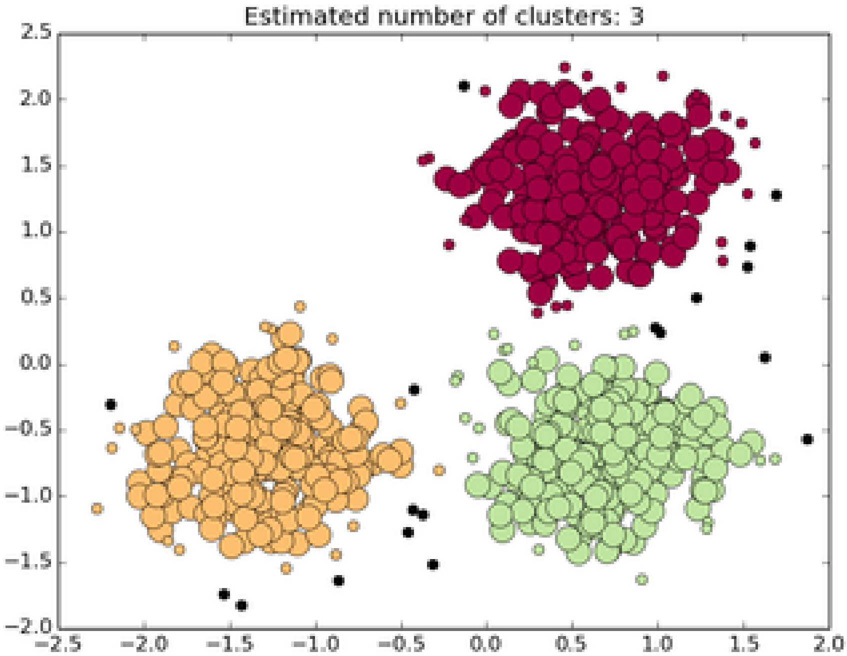


Figure 42.500: *Demo of DBSCAN clustering algorithm*

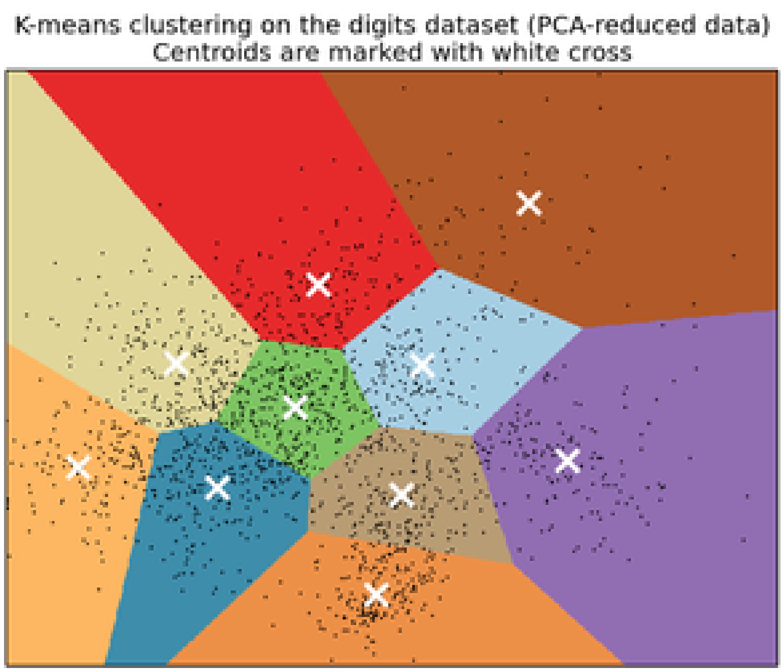


Figure 42.501: *A demo of K-Means clustering on the handwritten digits data*

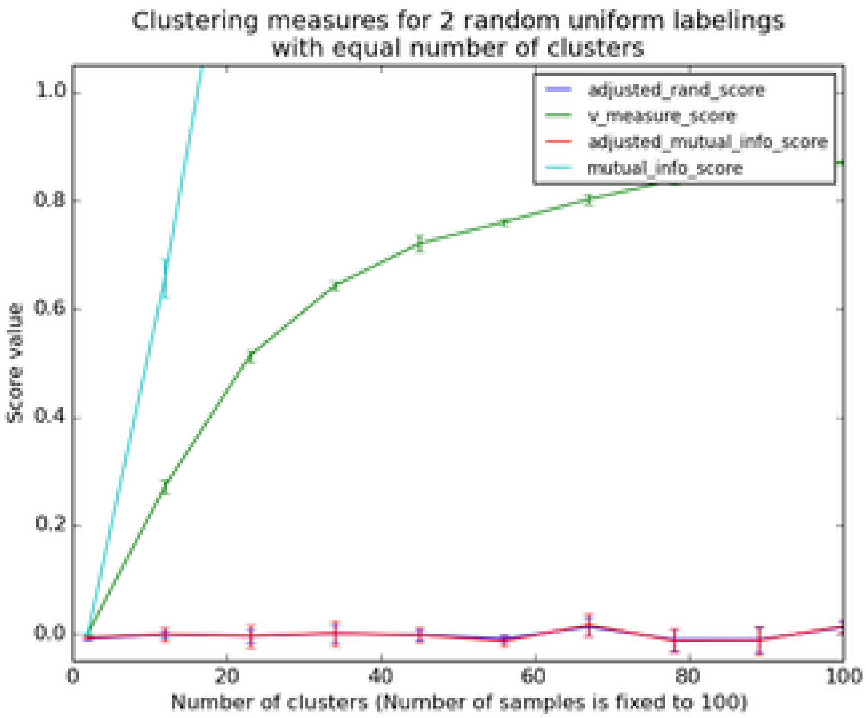


Figure 42.502: *Adjustment for chance in clustering performance evaluation*

Figure 42.503: *Clustering text documents using k-means*

Examples using `sklearn.metrics.v_measure_score`

42.21.6 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.

Parametersa : (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`

42.21.7 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
`metrics.pairwise.chi2_kernel`(X[, Y, gamma])	Computes the exponential chi-squared kernel 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 dist
`metrics.pairwise.kernel_metrics`()	Valid metrics for pairwise_kernels
`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.pairwise_distances`(X[, Y, ...])	Compute the distance matrix from a vector array X and optional Y
`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_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 point

Table 42.159 – continued from previous page

metrics.pairwise_distances_argmin_min(X, Y)	Compute minimum distances between one point and a set of point

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.

ParametersX : array-like of shape (n_samples_X, n_features)

Y : array of shape (n_samples_Y, n_features)

Returnskernel_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.AdditiveChi2SamplerA 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 http://eprints.pascal-network.org/archive/00002309/01/Zhang06-IJCV.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.

ParametersX : array-like of shape (n_samples_X, n_features)

Y : array of shape (n_samples_Y, n_features)

gamma : float, default=1.

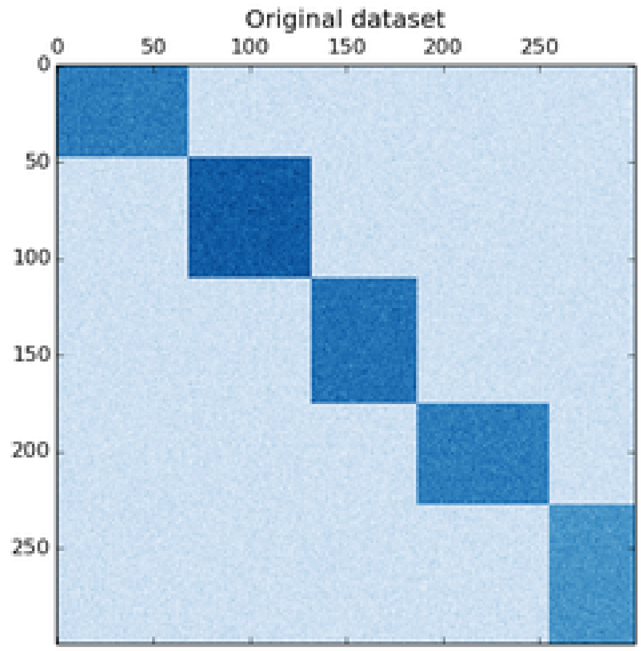


Figure 42.504: *A demo of the Spectral Co-Clustering algorithm*

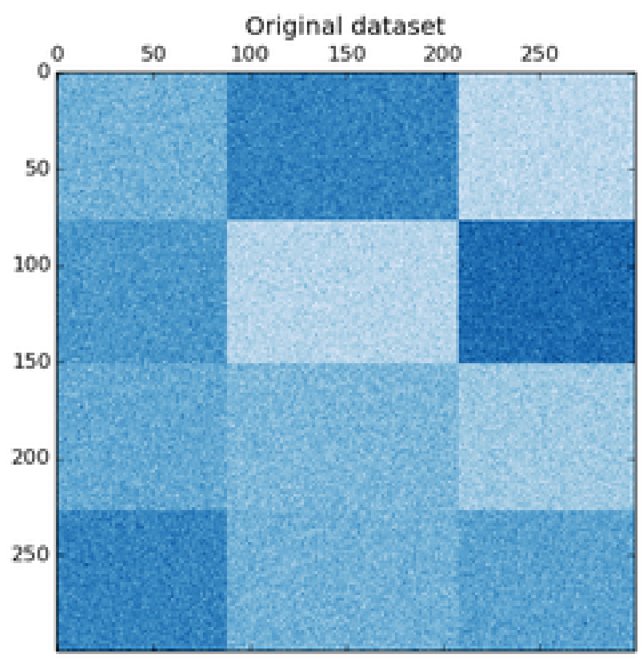


Figure 42.505: *A demo of the Spectral Biclustering algorithm*

Scaling parameter of the chi2 kernel.

Returnskernel_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 http://eprints.pascal-network.org/archive/00002309/01/Zhang06-IJCV.pdf

`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
'l1'	metrics.pairwise.manhattan_distances
'l2'	metrics.pairwise.euclidean_distances
'manhattan'	metrics.pairwise.manhattan_distances

`sklearn.metrics.pairwise`.euclidean_distances

`sklearn.metrics.pairwise.euclidean_distances(`*X, Y=None, Y_norm_squared=None, squared=False*`)`

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 x varies but y remains unchanged, then the right-most dot product *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.

ParametersX : {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)`)

squared : boolean, optional

Return squared Euclidean distances.

Returnsdistances : {array, sparse matrix}, shape (n_samples_1, n_samples_2)

See also:

`paired_distances`distances betweens pairs of elements of X and Y.

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`.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
'sigmoid'	sklearn.pairwise.sigmoid_kernel
'cosine'	sklearn.pairwise.cosine_similarity

`sklearn.metrics.pairwise`.linear_kernel

`sklearn.metrics.pairwise.`**`linear_kernel`**`(`*X, Y=None*`)`

Compute the linear kernel between X and Y.

ParametersX : array of shape (n_samples_1, n_features)

Y : array of shape (n_samples_2, n_features)

ReturnsGram 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*, *size_threshold=500000000.0*)

Compute the L1 distances between the vectors in X and Y.

With sum_over_features equal to False it returns the componentwise distances.

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.

size_threshold : int, default=5e8

Unused parameter.

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.

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 = 2 * np.ones((2, 2))
>>> manhattan_distances(X, y, sum_over_features=False)
array([[ 1.,  1.],
       [ 1.,  1.]]...)
```

sklearn.metrics.pairwise.pairwise_distances

sklearn.metrics.pairwise.**pairwise_distances**(*X*, *Y=None*, *metric='euclidean'*, *n_jobs=1*, ***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.
- From scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching', '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.

ParametersX : 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]

A second feature array only if X has shape [n_samples_a, n_features].

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

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.

If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are 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.

ReturnsD : 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.

sklearn.metrics.pairwise.pairwise_kernels

sklearn.metrics.pairwise.**pairwise_kernels**(*X*, *Y=None*, *metric='linear'*, *filter_params=False*, *n_jobs=1*, ***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::['rbf', 'sigmoid', 'polynomial', 'poly', 'linear', 'cosine']

ParametersX : 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 arrays from X as input and return a value indicating the distance between them.

n_jobs : int

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.

If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

filter_params: boolean :

Whether to filter invalid parameters or not.

'kwds'** : optional keyword parameters

Any further parameters are passed directly to the kernel function.

ReturnsK : 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
```

ParametersX : ndarray of shape (n_samples_1, n_features)

Y : ndarray of shape (n_samples_2, n_features)

coef0 : int, default 1

degree : int, default 3

ReturnsGram 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.

ParametersX : array of shape (n_samples_X, n_features)

Y : array of shape (n_samples_Y, n_features)

gamma : float

Returnskernel_matrix : array of shape (n_samples_X, n_samples_Y)

sklearn.metrics.pairwise_distances

sklearn.metrics.**pairwise_distances**(*X*, *Y=None*, *metric='euclidean'*, *n_jobs=1*, ***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.
- From scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching', '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.

ParametersX : 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]

A second feature array only if X has shape [n_samples_a, n_features].

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

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.

If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are 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.

ReturnsD : 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.

Examples using `sklearn.metrics.pairwise_distances`

`sklearn.metrics`.pairwise_distances_argmin

`sklearn.metrics.`**`pairwise_distances_argmin`**(*X*, *Y*, *axis=1*, *metric='euclidean'*, *batch_size=500*, *metric_kwargs={}*)

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)

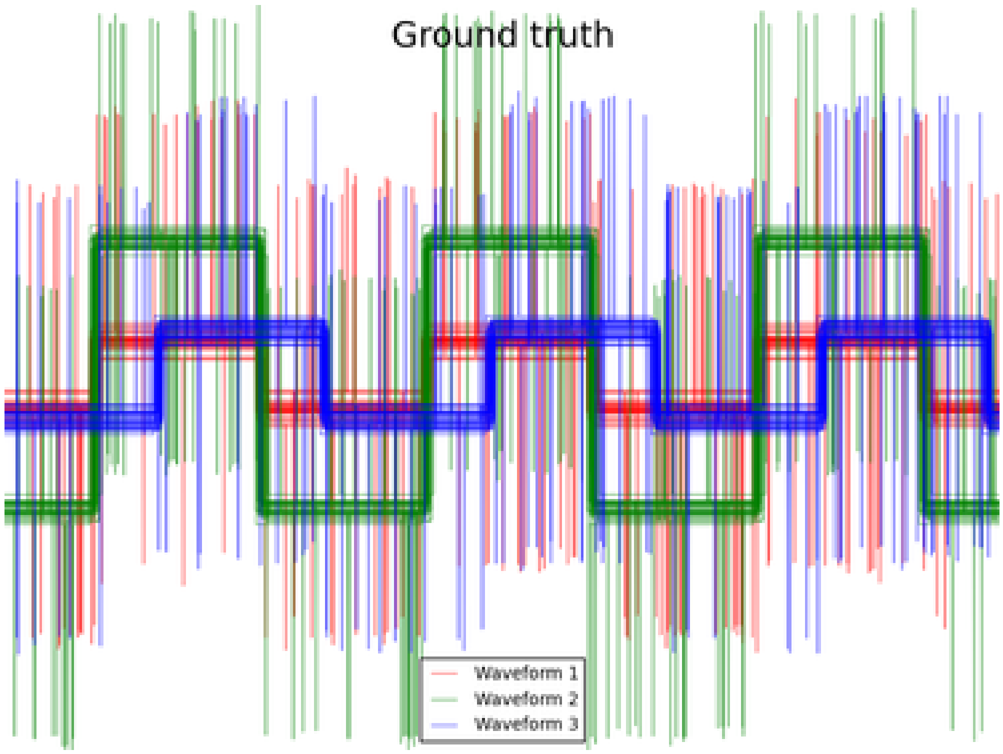


Figure 42.506: *Agglomerative clustering with different metrics*

but uses much less memory, and is faster for large arrays.

This function works with dense 2D arrays only.

ParametersX : 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)

batch_size : integer

To reduce memory consumption over the naive solution, data are processed in batches, comprising batch_size rows of X and batch_size rows of Y. The default value is quite conservative, but can be changed for fine-tuning. The larger the number, the larger the memory usage.

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', 'matching', '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.

axis : int, optional, default 1

Axis along which the argmin and distances are to be computed.

Returnsargmin : 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`

Figure 42.507: *Color Quantization using K-Means*

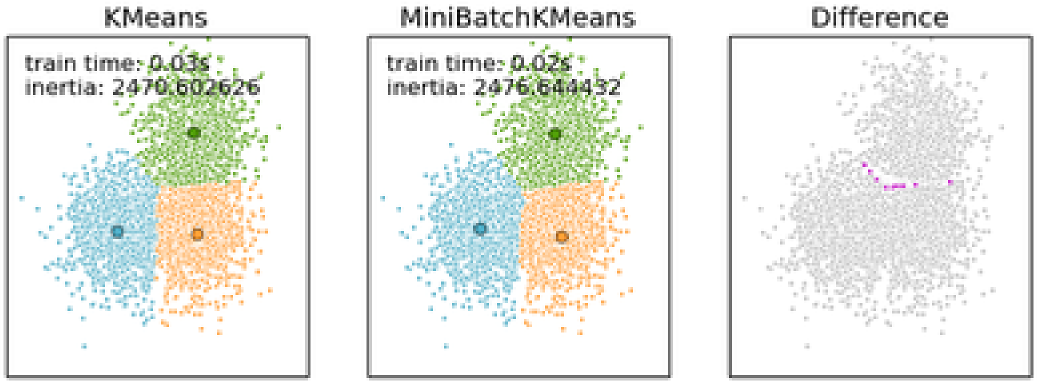


Figure 42.508: *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*

`sklearn.metrics`.pairwise_distances_argmin_min

`sklearn.metrics.pairwise_distances_argmin_min`(*X*, *Y*, *axis=1*, *metric='euclidean'*, *batch_size=500*, *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.

ParametersX, Y : {array-like, sparse matrix}

Arrays containing points. Respective shapes (n_samples1, n_features) and (n_samples2, n_features)

batch_size : integer

To reduce memory consumption over the naive solution, data are processed in batches, comprising batch_size rows of X and batch_size rows of Y. The default value is quite conservative, but can be changed for fine-tuning. The larger the number, the larger the memory usage.

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', 'matching', '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.

axis : int, optional, default 1

Axis along which the argmin and distances are to be computed.

Returnsargmin : 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`

42.22 `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.GMM`([n_components, covariance_type, ...])	Gaussian Mixture Model
`mixture.DPGMM`([n_components, ...])	Variational Inference for the Infinite Gaussian Mixture Model.
`mixture.VBGMM`([n_components, ...])	Variational Inference for the Gaussian Mixture Model

42.22.1 `sklearn.mixture.GMM`

class `sklearn.mixture.GMM`(*n_components=1*, *covariance_type='diag'*, *random_state=None*, *thresh=None*, *tol=0.001*, *min_covar=0.001*, *n_iter=100*, *n_init=1*, *params='wmc'*, *init_params='wmc'*)

Gaussian Mixture Model

Representation of a Gaussian mixture model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a GMM distribution.

Initializes parameters such that every mixture component has zero mean and identity covariance.

Parametersn_components : int, optional

Number of mixture components. Defaults to 1.

covariance_type : string, optional

String describing the type of covariance parameters to use. Must be one of 'spherical', 'tied', 'diag', 'full'. Defaults to 'diag'.

random_state: RandomState or an int seed (None by default) :

A random number generator instance

min_covar : float, optional

Floor on the diagonal of the covariance matrix to prevent overfitting. Defaults to 1e-3.

tol : float, optional

Convergence threshold. EM iterations will stop when average gain in log-likelihood is below this threshold. Defaults to 1e-3.

n_iter : int, optional

Number of EM iterations to perform.

n_init : int, optional

Number of initializations to perform. the best results is kept

params : string, optional

Controls which parameters are updated in the training process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars. Defaults to 'wmc'.

init_params : string, optional

Controls which parameters are updated in the initialization process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars. Defaults to 'wmc'.

Attributesweights_ : array, shape (*n_components*,)

This attribute stores the mixing weights for each mixture component.

means_ : array, shape (*n_components*, *n_features*)

Mean parameters for each mixture component.

covars_ : array

Covariance parameters for each mixture component. The shape depends on *covariance_type*:

```
(n_components, n_features)             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.

See also:

`DPGMM`Infinite gaussian mixture model, using the dirichlet process, fit with a variational algorithm

`VBGMM`Finite gaussian mixture model fit with a variational algorithm, better for situations where there might be too little data to get a good estimate of the covariance matrix.

Examples

```
>>> import numpy as np
>>> from sklearn import mixture
>>> np.random.seed(1)
>>> g = mixture.GMM(n_components=2)
>>> # Generate random observations with two modes centered on 0
>>> # and 10 to use for training.
>>> obs = np.concatenate((np.random.randn(100, 1),
...                       10 + np.random.randn(300, 1)))
>>> g.fit(obs)
GMM(covariance_type='diag', init_params='wmc', min_covar=0.001,
        n_components=2, n_init=1, n_iter=100, params='wmc',
        random_state=None, thresh=None, tol=0.001)
>>> np.round(g.weights_, 2)
array([ 0.75,  0.25])
>>> np.round(g.means_, 2)
array([[ 10.05],
       [  0.06]])
>>> np.round(g.covars_, 2)
array([[[ 1.02]],
       [[ 0.96]]])
>>> g.predict([[0], [2], [9], [10]])
array([1, 1, 0, 0]...)
>>> np.round(g.score([[0], [2], [9], [10]]), 2)
array([-2.19, -4.58, -1.75, -1.21])
>>> # Refit the model on new data (initial parameters remain the
>>> # same), this time with an even split between the two modes.
>>> g.fit(20 * [[0]] +  20 * [[10]])
GMM(covariance_type='diag', init_params='wmc', min_covar=0.001,
        n_components=2, n_init=1, n_iter=100, params='wmc',
        random_state=None, thresh=None, tol=0.001)
>>> np.round(g.weights_, 2)
array([ 0.5,  0.5])
```

Methods

`aic`(X)	Akaike information criterion for the current model fit
`bic`(X)	Bayesian information criterion for the current model fit
`fit`(X[, y])	Estimate model parameters with the expectation-maximization algorithm.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict label for data.
`predict_proba`(X)	Predict posterior probability of data under each Gaussian in the model.
`sample`([n_samples, random_state])	Generate random samples from the model.
`score`(X[, y])	Compute the log probability under the model.
`score_samples`(X)	Return the per-sample likelihood of the data under the model.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*n_components=1, covariance_type='diag', random_state=None, thresh=None, tol=0.001, min_covar=0.001, n_iter=100, n_init=1, params='wmc', init_params='wmc'*)

static `aic`(*X*)

Akaike information criterion for the current model fit and the proposed data

ParametersX : array of shape(n_samples, n_dimensions)

Returnsaic: float (the lower the better) :

static bic (*X*)
Bayesian information criterion for the current model fit and the proposed data

ParametersX : array of shape(n_samples, n_dimensions)

Returnsbic: float (the lower the better) :

static fit (*X*, *y=None*)
Estimate model parameters with the expectation-maximization algorithm.

A initialization step is performed before entering the em algorithm. If you want to avoid this step, set the keyword argument init_params to the empty string '' when creating the GMM object. Likewise, if you would like just to do an initialization, set n_iter=0.

ParametersX : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static predict (*X*)
Predict label for data.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsC : array, shape = (n_samples,)

static predict_proba (*X*)
Predict posterior probability of data under each Gaussian in the model.

ParametersX : array-like, shape = [n_samples, n_features]

Returnsresponsibilities : array-like, shape = (n_samples, n_components)

Returns the probability of the sample for each Gaussian (state) in the model.

static sample (*n_samples=1*, *random_state=None*)
Generate random samples from the model.

Parametersn_samples : int, optional

Number of samples to generate. Defaults to 1.

ReturnsX : array_like, shape (n_samples, n_features)

List of samples

static score (*X*, *y=None*)
Compute the log probability under the model.

ParametersX : array_like, shape (n_samples, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

Returnslogprob : array_like, shape (n_samples,)

Log probabilities of each data point in X

static **`score_samples`**(*X*)

Return the per-sample likelihood of the data under the model.

Compute the log probability of X under the model and return the posterior distribution (responsibilities) of each mixture component for each element of X.

ParametersX: array_like, shape (n_samples, n_features) :

List of n_features-dimensional data points. Each row corresponds to a single data point.

Returnslogprob : array_like, shape (n_samples,)

Log probabilities of each data point in X.

responsibilities : array_like, shape (n_samples, n_components)

Posterior probabilities of each mixture component for each observation

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.mixture.GMM`

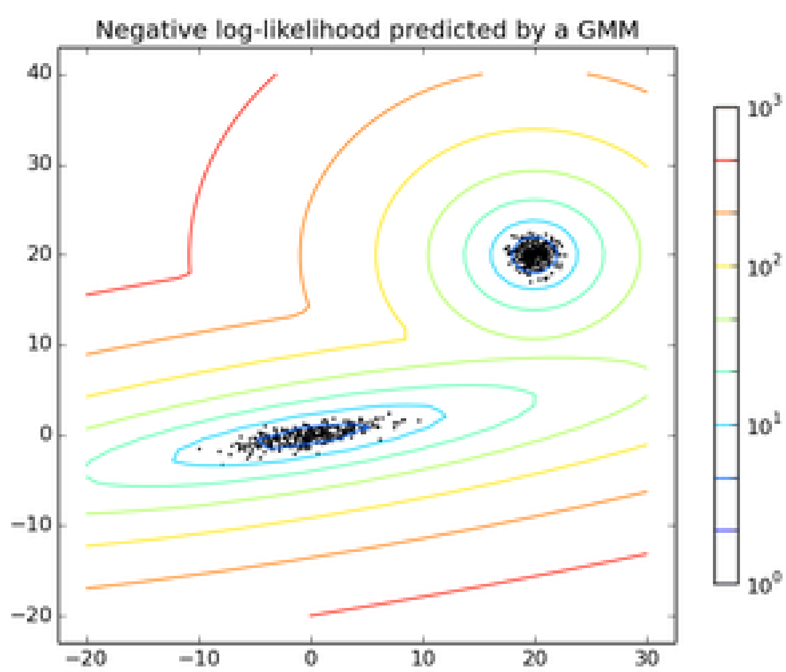


Figure 42.509: *Density Estimation for a mixture of Gaussians*

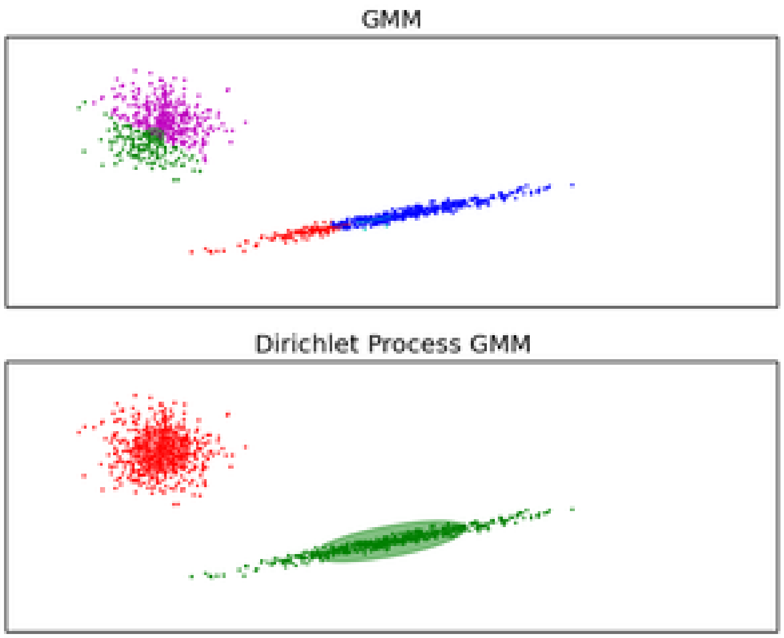


Figure 42.510: *Gaussian Mixture Model Ellipsoids*

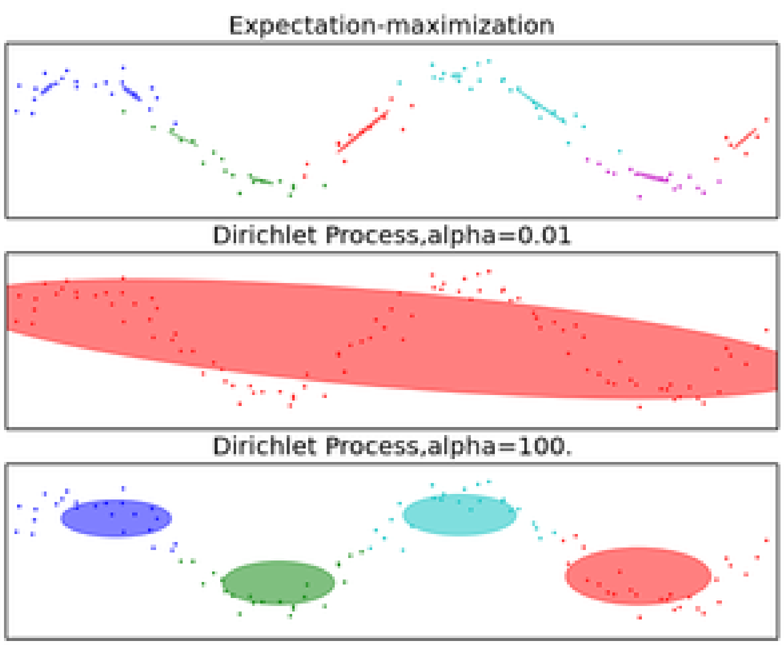


Figure 42.511: *Gaussian Mixture Model Sine Curve*

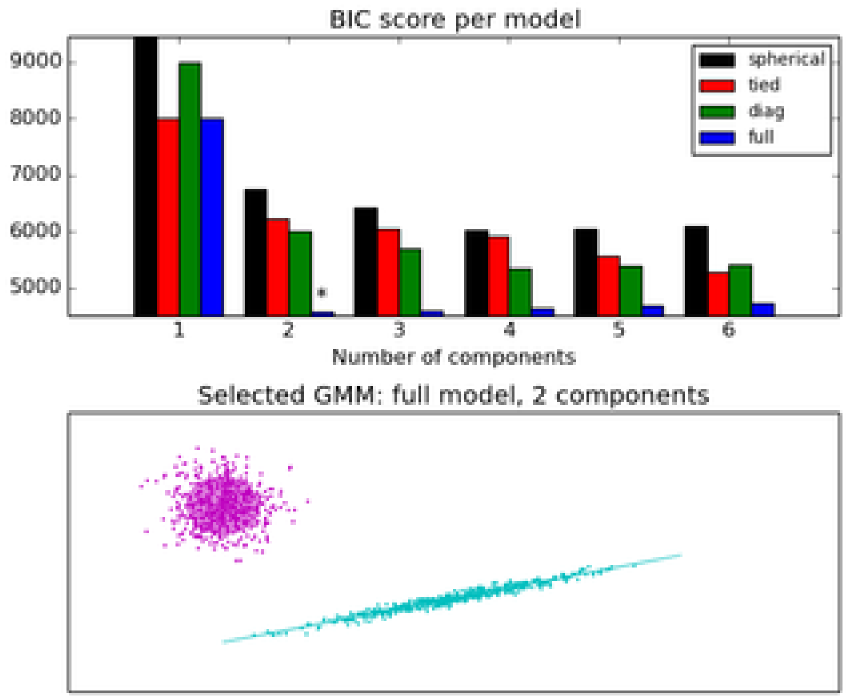


Figure 42.512: *Gaussian Mixture Model Selection*

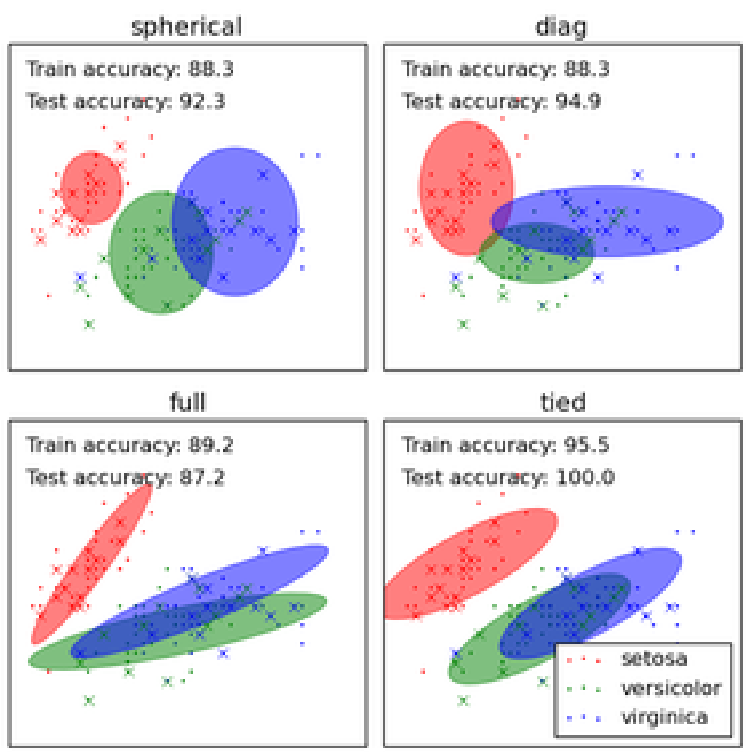


Figure 42.513: *GMM classification*

42.22.2 sklearn.mixture.DPGMM

class sklearn.mixture.**DPGMM**(*n_components=1*, *covariance_type='diag'*, *alpha=1.0*, *random_state=None*, *thresh=None*, *tol=0.001*, *verbose=False*, *min_covar=None*, *n_iter=10*, *params='wmc'*, *init_params='wmc'*)

Variational Inference for the Infinite Gaussian Mixture Model.

DPGMM stands for Dirichlet Process Gaussian Mixture Model, and it is an infinite mixture model with the Dirichlet Process as a prior distribution on the number of clusters. In practice the approximate inference algorithm uses a truncated distribution with a fixed maximum number of components, but almost always the number of components actually used depends on the data.

Stick-breaking Representation of a Gaussian mixture model probability distribution. This class allows for easy and efficient inference of an approximate posterior distribution over the parameters of a Gaussian mixture model with a variable number of components (smaller than the truncation parameter n_components).

Initialization is with normally-distributed means and identity covariance, for proper convergence.

Parametersn_components: int, default 1 :

Number of mixture components.

covariance_type: string, default 'diag' :

String describing the type of covariance parameters to use. Must be one of 'spherical', 'tied', 'diag', 'full'.

alpha: float, default 1 :

Real number representing the concentration parameter of the dirichlet process. Intuitively, the Dirichlet Process is as likely to start a new cluster for a point as it is to add that point to a cluster with alpha elements. A higher alpha means more clusters, as the expected number of clusters is `alpha*log(N)`.

tol : float, default 1e-3

Convergence threshold.

n_iter : int, default 10

Maximum number of iterations to perform before convergence.

params : string, default 'wmc'

Controls which parameters are updated in the training process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars.

init_params : string, default 'wmc'

Controls which parameters are updated in the initialization process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars. Defaults to 'wmc'.

verbose : boolean, default False

Controls output verbosity.

Attributescovariance_type : string

String describing the type of covariance parameters used by the DP-GMM. Must be one of 'spherical', 'tied', 'diag', 'full'.

n_components : int

Number of mixture components.

weights_ : array, shape (*n_components*,) Mixing weights for each mixture component.

means_ : array, shape (*n_components*, *n_features*)

Mean parameters for each mixture component.

precs_ : array

Precision (inverse covariance) parameters for each mixture component. The shape depends on *covariance_type*:

```
(`n_components`, 'n_features')                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.

See also:

GMMFinite Gaussian mixture model fit with EM

VBGMMFinite Gaussian mixture model fit with a variational algorithm, better for situations where there might be too little data to get a good estimate of the covariance matrix.

Methods

aic(X)	Akaike information criterion for the current model fit
bic(X)	Bayesian information criterion for the current model fit
fit(X[, y])	Estimate model parameters with the variational algorithm.
get_params([deep])	Get parameters for this estimator.
lower_bound(X, z)	returns a lower bound on model evidence based on X and membership
predict(X)	Predict label for data.
predict_proba(X)	Predict posterior probability of data under each Gaussian in the model.
sample([n_samples, random_state])	Generate random samples from the model.
score(X[, y])	Compute the log probability under the model.
score_samples(X)	Return the likelihood of the data under the model.
set_params(**params)	Set the parameters of this estimator.

static __init__(*n_components=1*, *covariance_type='diag'*, *alpha=1.0*, *random_state=None*, *thresh=None*, *tol=0.001*, *verbose=False*, *min_covar=None*, *n_iter=10*, *params='wmc'*, *init_params='wmc'*)

aic(*X*)

Akaike information criterion for the current model fit and the proposed data

ParametersX : array of shape(n_samples, n_dimensions)

Returnsaic: float (the lower the better) :

bic(*X*)

Bayesian information criterion for the current model fit and the proposed data

ParametersX : array of shape(n_samples, n_dimensions)

Returnsbic: float (the lower the better) :

static fit (*X*, *y=None*)

Estimate model parameters with the variational algorithm.

For a full derivation and description of the algorithm see doc/modules/dp-derivation.rst or http://scikit-learn.org/stable/modules/dp-derivation.html

A initialization step is performed before entering the em algorithm. If you want to avoid this step, set the keyword argument init_params to the empty string '' when when creating the object. Likewise, if you would like just to do an initialization, set n_iter=0.

ParametersX : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static lower_bound (*X*, *z*)

returns a lower bound on model evidence based on X and membership

predict (*X*)

Predict label for data.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsC : array, shape = (n_samples,)

predict_proba (*X*)

Predict posterior probability of data under each Gaussian in the model.

ParametersX : array-like, shape = [n_samples, n_features]

Returnsresponsibilities : array-like, shape = (n_samples, n_components)

Returns the probability of the sample for each Gaussian (state) in the model.

sample (*n_samples=1*, *random_state=None*)

Generate random samples from the model.

Parametersn_samples : int, optional

Number of samples to generate. Defaults to 1.

ReturnsX : array_like, shape (n_samples, n_features)

List of samples

score (*X*, *y=None*)

Compute the log probability under the model.

ParametersX : array_like, shape (n_samples, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

Returnslogprob : array_like, shape (n_samples,)

Log probabilities of each data point in X

static **score_samples** (*X*)
Return the likelihood of the data under the model.

Compute the bound on log probability of X under the model and return the posterior distribution (responsibilities) of each mixture component for each element of X.

This is done by computing the parameters for the mean-field of z for each observation.

ParametersX : array_like, shape (n_samples, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

Returnslogprob : array_like, shape (n_samples,)

Log probabilities of each data point in X

responsibilities: array_like, shape (n_samples, n_components) :

Posterior probabilities of each mixture component for each observation

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using sklearn.mixture.DPGMM

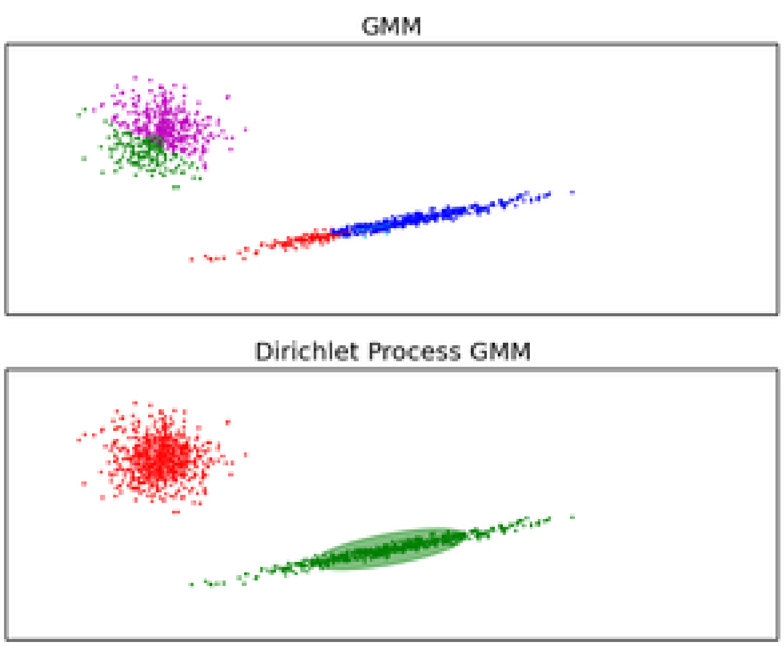


Figure 42.514: *Gaussian Mixture Model Ellipsoids*

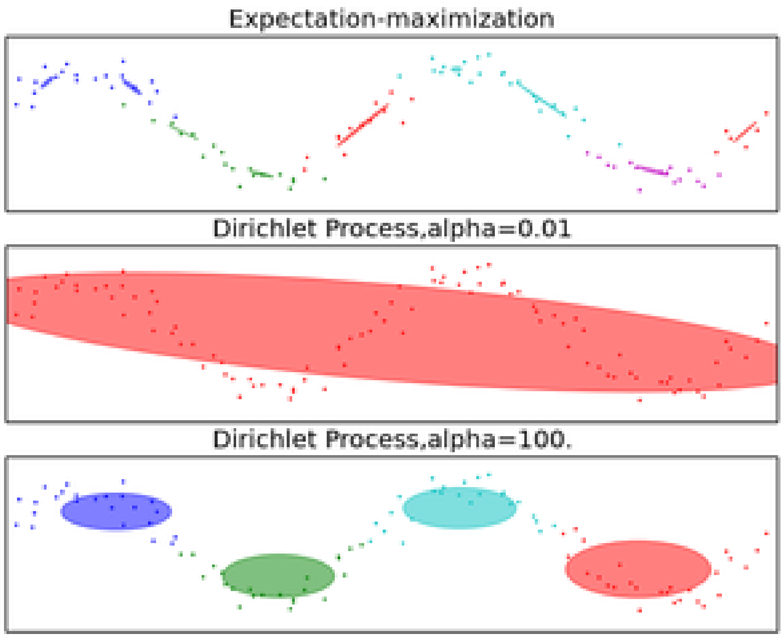


Figure 42.515: *Gaussian Mixture Model Sine Curve*

42.22.3 `sklearn.mixture.VBGMM`

class `sklearn.mixture.`**`VBGMM`**(*n_components=1*, *covariance_type='diag'*, *alpha=1.0*, *random_state=None*, *thresh=None*, *tol=0.001*, *verbose=False*, *min_covar=None*, *n_iter=10*, *params='wmc'*, *init_params='wmc'*)

Variational Inference for the Gaussian Mixture Model

Variational inference for a Gaussian mixture model probability distribution. This class allows for easy and efficient inference of an approximate posterior distribution over the parameters of a Gaussian mixture model with a fixed number of components.

Initialization is with normally-distributed means and identity covariance, for proper convergence.

Parametersn_components: int, default 1 :

Number of mixture components.

covariance_type: string, default 'diag' :

String describing the type of covariance parameters to use. Must be one of 'spherical', 'tied', 'diag', 'full'.

alpha: float, default 1 :

Real number representing the concentration parameter of the dirichlet distribution. Intuitively, the higher the value of alpha the more likely the variational mixture of Gaussians model will use all components it can.

tol : float, default 1e-3

Convergence threshold.

n_iter : int, default 10

Maximum number of iterations to perform before convergence.

params : string, default 'wmc'

Controls which parameters are updated in the training process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars.

init_params : string, default 'wmc'

Controls which parameters are updated in the initialization process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars. Defaults to 'wmc'.

verbose : boolean, default False

Controls output verbosity.

Attributescovariance_type : string

String describing the type of covariance parameters used by the DP-GMM. Must be one of 'spherical', 'tied', 'diag', 'full'.

n_features : int

Dimensionality of the Gaussians.

n_components : int (read-only)

Number of mixture components.

weights_ : array, shape (*n_components*,)

Mixing weights for each mixture component.

means_ : array, shape (*n_components*, *n_features*)

Mean parameters for each mixture component.

precs_ : array

Precision (inverse covariance) parameters for each mixture component. The shape depends on *covariance_type*:

```
(`n_components`, 'n_features')                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.

See also:

GMMFinite Gaussian mixture model fit with EM

DPGMMIninite Gaussian mixture model, using the dirichlet process, fit with a variational algorithm

Methods

aic(X)	Akaike information criterion for the current model fit
bic(X)	Bayesian information criterion for the current model fit

Continued on next page

Table 42.163 – continued from previous page

fit(X[, y])	Estimate model parameters with the variational algorithm.
get_params([deep])	Get parameters for this estimator.
lower_bound(X, z)	returns a lower bound on model evidence based on X and membership
predict(X)	Predict label for data.
predict_proba(X)	Predict posterior probability of data under each Gaussian in the model.
sample([n_samples, random_state])	Generate random samples from the model.
score(X[, y])	Compute the log probability under the model.
score_samples(X)	Return the likelihood of the data under the model.
set_params(**params)	Set the parameters of this estimator.

static __init__(*n_components=1*, *covariance_type='diag'*, *alpha=1.0*, *random_state=None*, *thresh=None*, *tol=0.001*, *verbose=False*, *min_covar=None*, *n_iter=10*, *params='wmc'*, *init_params='wmc'*)

aic(*X*)

Akaike information criterion for the current model fit and the proposed data

ParametersX : array of shape(n_samples, n_dimensions)

Returnsaic: float (the lower the better) :

bic(*X*)

Bayesian information criterion for the current model fit and the proposed data

ParametersX : array of shape(n_samples, n_dimensions)

Returnsbic: float (the lower the better) :

fit(*X*, *y=None*)

Estimate model parameters with the variational algorithm.

For a full derivation and description of the algorithm see doc/modules/dp-derivation.rst or http://scikit-learn.org/stable/modules/dp-derivation.html

A initialization step is performed before entering the em algorithm. If you want to avoid this step, set the keyword argument init_params to the empty string '' when when creating the object. Likewise, if you would like just to do an initialization, set n_iter=0.

ParametersX : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

lower_bound(*X*, *z*)

returns a lower bound on model evidence based on X and membership

predict(*X*)

Predict label for data.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsC : array, shape = (n_samples,)

predict_proba (*X*)
Predict posterior probability of data under each Gaussian in the model.

ParametersX : array-like, shape = [n_samples, n_features]

Returnsresponsibilities : array-like, shape = (n_samples, n_components)

Returns the probability of the sample for each Gaussian (state) in the model.

sample (*n_samples=1*, *random_state=None*)
Generate random samples from the model.

Parametersn_samples : int, optional

Number of samples to generate. Defaults to 1.

ReturnsX : array_like, shape (n_samples, n_features)

List of samples

score (*X*, *y=None*)
Compute the log probability under the model.

ParametersX : array_like, shape (n_samples, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

Returnslogprob : array_like, shape (n_samples,)

Log probabilities of each data point in X

static score_samples (*X*)
Return the likelihood of the data under the model.

Compute the bound on log probability of X under the model and return the posterior distribution (responsibilities) of each mixture component for each element of X.

This is done by computing the parameters for the mean-field of z for each observation.

ParametersX : array_like, shape (n_samples, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

Returnslogprob : array_like, shape (n_samples,)

Log probabilities of each data point in X

responsibilities: array_like, shape (n_samples, n_components) :

Posterior probabilities of each mixture component for each observation

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

42.23 `sklearn.multiclass`: Multiclass and multilabel classification

42.23.1 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

42.23.2 `sklearn.multiclass`.OneVsRestClassifier

class `sklearn.multiclass.`**`OneVsRestClassifier`**(*estimator*, *n_jobs=1*)

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.

Parametersestimator : estimator object

An estimator object implementing *fit* and one of *decision_function* or *predict_proba*.

n_jobs : int, optional, default: 1

The number of jobs to use for the computation. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

Attributesestimators_ : list of *n_classes* estimators

Estimators used for predictions.

classes_ : array, shape = [*n_classes*]

Class labels.

label_binarizer_ : LabelBinarizer object

Object used to transform multiclass labels to binary labels and vice-versa.

multilabel_ : boolean

Whether a OneVsRestClassifier is a multilabel classifier.

Methods

decision_function(X)	Returns the distance of each sample from the decision boundary for each class.
fit(X, y)	Fit underlying estimators.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict multi-class targets using underlying estimators.
predict_proba(X)	Probability estimates.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

static **`__init__`**(*estimator*, *n_jobs=1*)

static **`decision_function`**(*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.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsT : array-like, shape = [n_samples, n_classes]

static **`fit`**(*X*, *y*)

Fit underlying estimators.

ParametersX : (sparse) array-like, shape = [n_samples, n_features]

Data.

y : (sparse) array-like, shape = [n_samples] or [n_samples, n_classes]

Multi-class targets. An indicator matrix turns on multilabel classification.

Returnsself :

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **`predict`**(*X*)

Predict multi-class targets using underlying estimators.

ParametersX : (sparse) array-like, shape = [n_samples, n_features]

Data.

Returnsy : (sparse) array-like, shape = [n_samples] or [n_samples, n_classes].

Predicted multi-class targets.

static **predict_proba**(*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.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsT : (sparse) array-like, 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(*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.multiclass.OneVsRestClassifier`

42.23.3 `sklearn.multiclass`.OneVsOneClassifier

class `sklearn.multiclass.`**OneVsOneClassifier**(*estimator*, *n_jobs=1*)
One-vs-one multiclass strategy

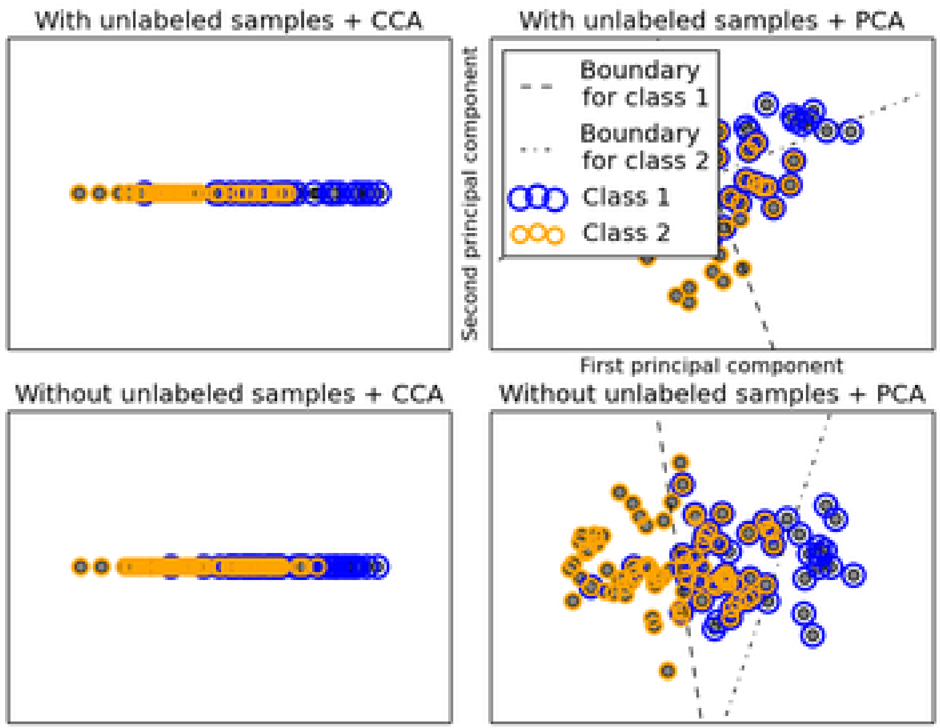


Figure 42.516: *Multilabel classification*

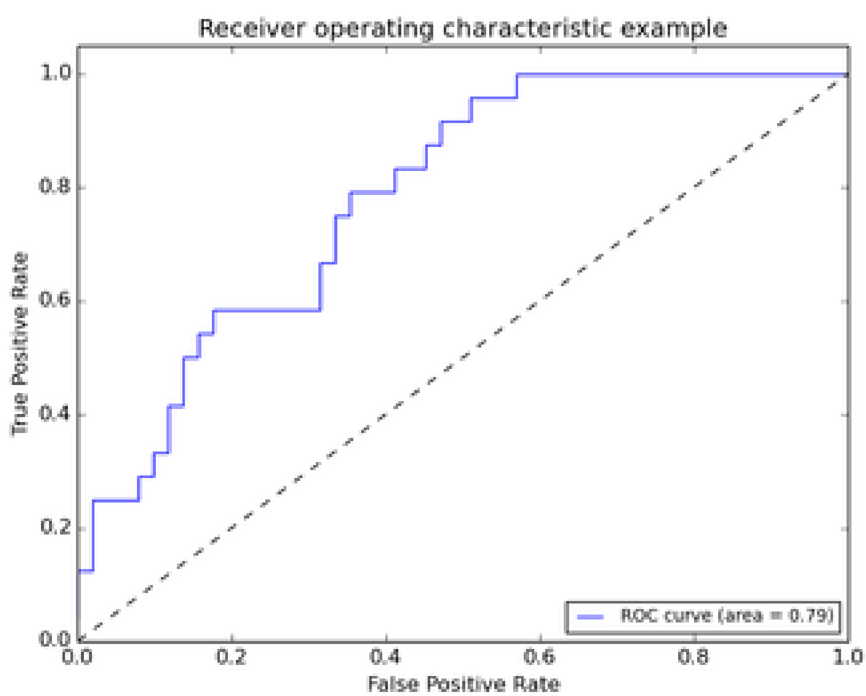


Figure 42.517: *Receiver Operating Characteristic (ROC)*

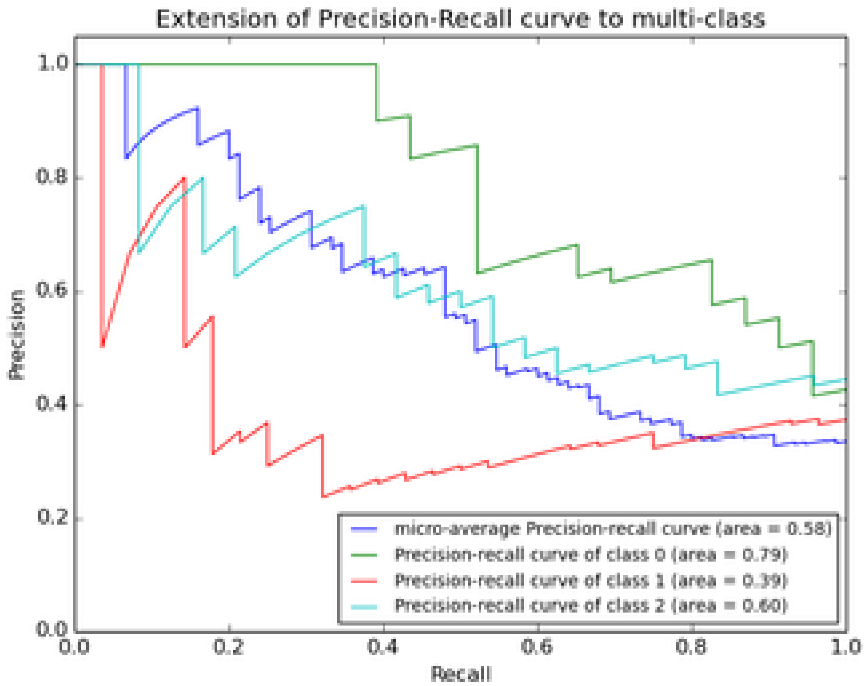


Figure 42.518: *Precision-Recall*

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.

Parametersestimator : estimator object

An estimator object implementing *fit* and one of *decision_function* or *predict_proba*.

n_jobs : int, optional, default: 1

The number of jobs to use for the computation. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

Attributesestimators_ : list of *n_classes * (n_classes - 1) / 2* estimators

Estimators used for predictions.

classes_ : numpy array of shape [n_classes]

Array containing labels.

Methods

`decision_function(X)`	Decision function for the OneVsOneClassifier.

Continued on next page

Table 42.166 – continued from previous page

fit(X, y)	Fit underlying estimators.
get_params([deep])	Get parameters for this estimator.
predict(X)	Estimate the best class label for each sample in X.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

static `__init__`(*estimator*, *n_jobs=1*)

static `decision_function`(*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.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsY : array-like, shape = [n_samples, n_classes]

static `fit`(*X*, *y*)

Fit underlying estimators.

ParametersX : (sparse) array-like, shape = [n_samples, n_features]

Data.

y : array-like, shape = [n_samples]

Multi-class targets.

Returnsself :

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static `predict`(*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.

ParametersX : (sparse) array-like, shape = [n_samples, n_features]

Data.

Returnsy : numpy array of shape [n_samples]

Predicted multi-class targets.

`score`(*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

42.23.4 sklearn.multiclass.OutputCodeClassifier

class sklearn.multiclass.**OutputCodeClassifier**(*estimator*, *code_size=1.5*, *random_state=None*, *n_jobs=1*)

(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.

Parametersestimator : 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 : numpy.RandomState, optional

The generator used to initialize the codebook. Defaults to numpy.random.

n_jobs : int, optional, default: 1

The number of jobs to use for the computation. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

Attributesestimators_ : 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

[R184], [R185], [R186]

Methods

fit(X, y)	Fit underlying estimators.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict multi-class targets using underlying estimators.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

static __init__ (*estimator*, *code_size=1.5*, *random_state=None*, *n_jobs=1*)

static fit (*X*, *y*)

Fit underlying estimators.

ParametersX : (sparse) array-like, shape = [n_samples, n_features]

Data.

y : numpy array of shape [n_samples]

Multi-class targets.

Returnsself :

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static predict (*X*)

Predict multi-class targets using underlying estimators.

ParametersX : (sparse) array-like, shape = [n_samples, n_features]

Data.

Returnsy : numpy array of shape [n_samples]

Predicted multi-class targets.

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(**params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

42.24 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.GaussianNB`	Gaussian Naive Bayes (GaussianNB)
`naive_bayes.MultinomialNB`([alpha, ...])	Naive Bayes classifier for multinomial models
`naive_bayes.BernoulliNB`([alpha, binarize, ...])	Naive Bayes classifier for multivariate Bernoulli models.

42.24.1 sklearn.naive_bayes.GaussianNB

class sklearn.naive_bayes.**GaussianNB**

Gaussian Naive Bayes (GaussianNB)

Can perform online updates to model parameters via *partial_fit* method. 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

Attributesclass_prior_ : array, shape (n_classes,)

probability of each class.

class_count_ : array, shape (n_classes,)

number of training samples observed in each class.

theta_ : array, shape (n_classes, n_features)

mean of each feature per class

sigma_ : array, shape (n_classes, n_features)

variance 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`(X, y)	Fit Gaussian Naive Bayes according to X, y
`get_params`([deep])	Get parameters for this estimator.
`partial_fit`(X, y[, classes])	Incremental fit on a batch of samples.
`predict`(X)	Perform classification on an array of test vectors X.
`predict_log_proba`(X)	Return log-probability estimates for the test vector X.
`predict_proba`(X)	Return probability estimates for the test vector X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.

`__init__`()

static **`fit`**(*X*, *y*)

Fit Gaussian Naive Bayes according to X, y

ParametersX : 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.

Returnsself : object

Returns self.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **partial_fit** (*X*, *y*, *classes=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.

ParametersX : 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,)

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.

Returnsself : object

Returns self.

predict (*X*)
Perform classification on an array of test vectors X.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsC : array, shape = [n_samples]

Predicted target values for X

predict_log_proba (*X*)
Return log-probability estimates for the test vector X.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsC : array-like, 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 (*X*)
Return probability estimates for the test vector X.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsC : array-like, 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 (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.naive_bayes.GaussianNB`

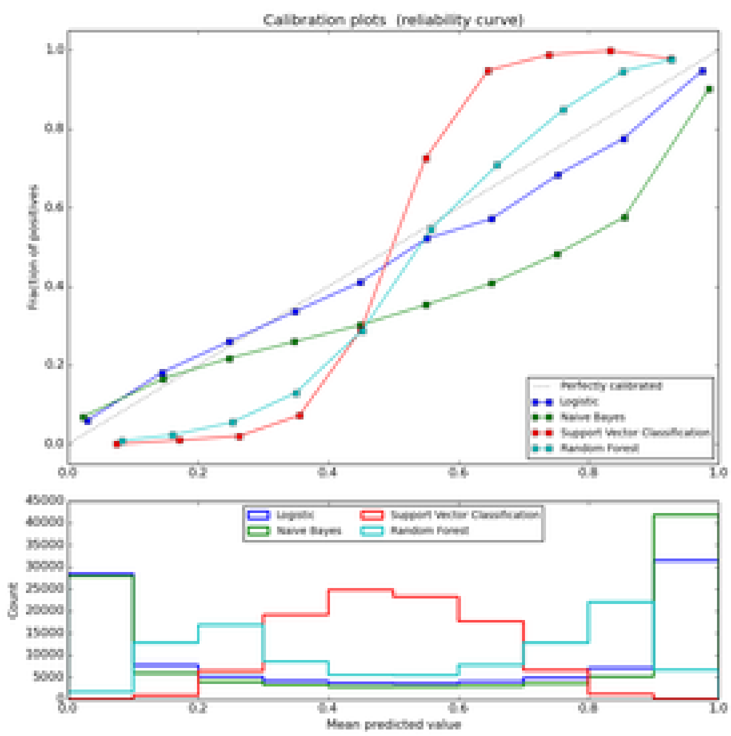


Figure 42.519: *Comparison of Calibration of Classifiers*

42.24.2 `sklearn.naive_bayes`.MultinomialNB

class `sklearn.naive_bayes.`**`MultinomialNB`**(*alpha=1.0*, *fit_prior=True*, *class_prior=None*)

Naive Bayes classifier for multinomial models

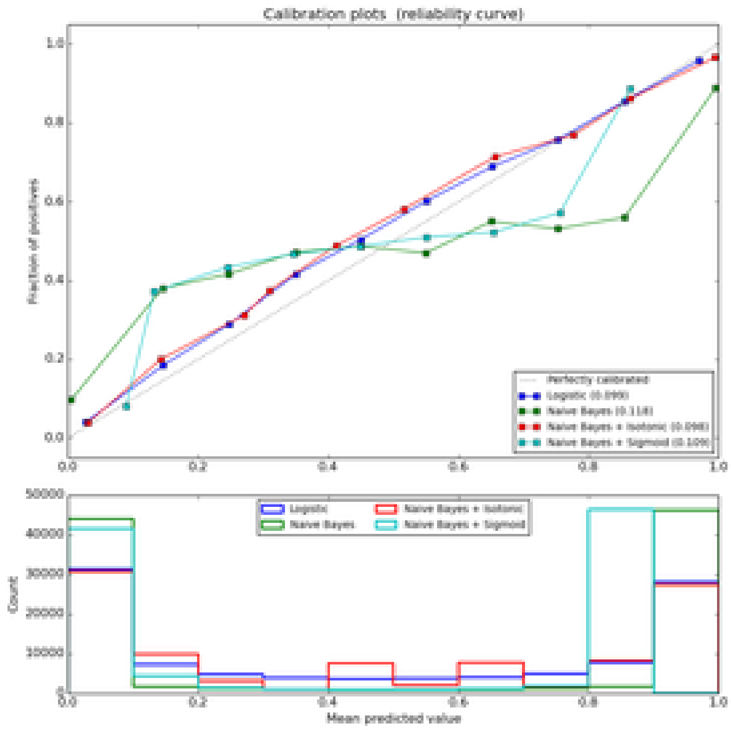


Figure 42.520: *Probability Calibration curves*

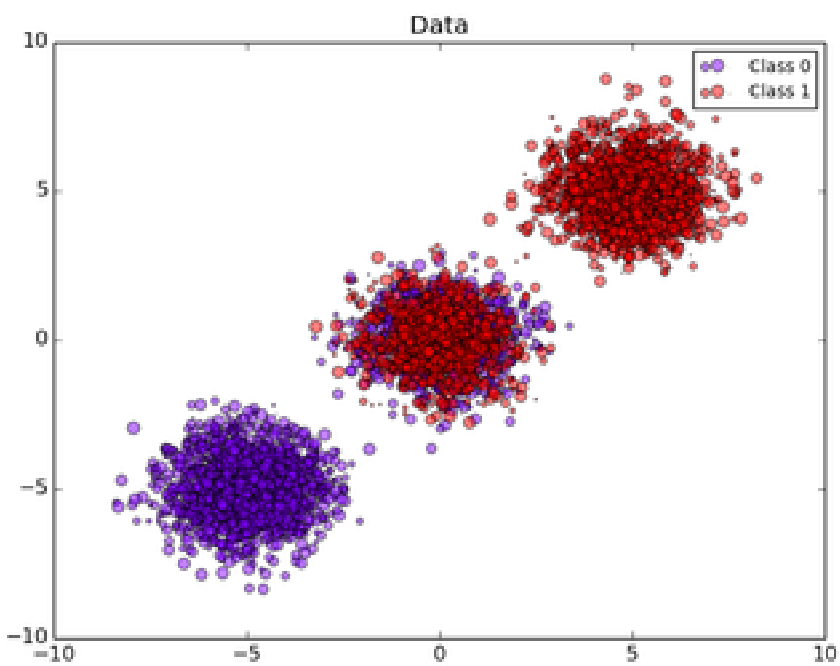


Figure 42.521: *Probability calibration of classifiers*

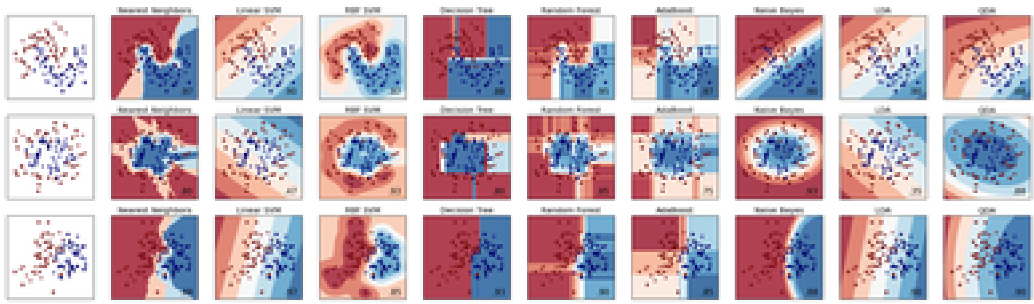


Figure 42.522: *Classifier comparison*

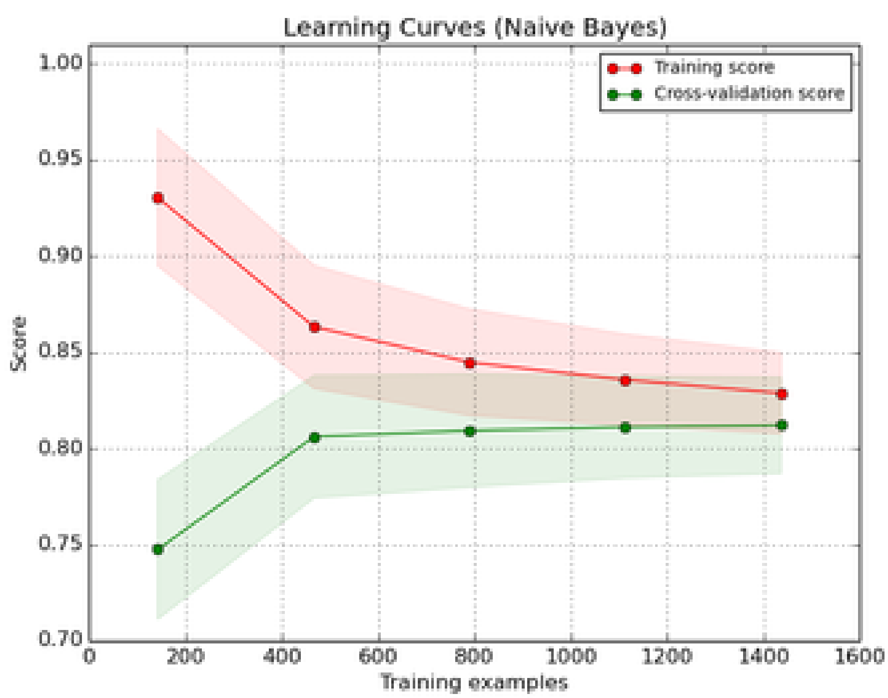


Figure 42.523: *Plotting Learning Curves*

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.

Parametersalpha : float, optional (default=1.0)

Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

fit_prior : boolean

Whether to learn class prior probabilities or not. If false, a uniform prior will be used.

class_prior : array-like, size (n_classes,)

Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

Attributesclass_log_prior_ : array, shape (n_classes,)

Smoothed empirical log probability for each class.

intercept_ : property

Mirrors `class_log_prior_` for interpreting MultinomialNB as a linear model.

feature_log_prob_ : array, shape (n_classes, n_features)

Empirical log probability of features given a class, `P(x_i|y)`.

coef_ : property

Mirrors `feature_log_prob_` for interpreting MultinomialNB as a linear model.

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.

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.

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. http://nlp.stanford.edu/IR-book/html/htmledition/naive-bayes-text-classification-1.html

Examples

```
>>> import numpy as np
>>> X = np.random.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(alpha=1.0, class_prior=None, fit_prior=True)
>>> print(clf.predict(X[2]))
[3]
```

Methods

fit(X, y[, sample_weight])	Fit Naive Bayes classifier according to X, y
get_params([deep])	Get parameters for this estimator.
partial_fit(X, y[, classes, sample_weight])	Incremental fit on a batch of samples.
predict(X)	Perform classification on an array of test vectors X.
predict_log_proba(X)	Return log-probability estimates for the test vector X.
predict_proba(X)	Return probability estimates for the test vector X.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

static __init__(*alpha=1.0*, *fit_prior=True*, *class_prior=None*)

fit(*X*, *y*, *sample_weight=None*)

Fit Naive Bayes classifier according to X, y

ParametersX : {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], optional

Weights applied to individual samples (1. for unweighted).

Returnsself : object

Returns self.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

partial_fit(*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.

ParametersX : {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.

classes : array-like, shape = [n_classes]

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

Weights applied to individual samples (1. for unweighted).

Returnsself : object

Returns self.

predict (*X*)

Perform classification on an array of test vectors X.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsC : array, shape = [n_samples]

Predicted target values for X

predict_log_proba (*X*)

Return log-probability estimates for the test vector X.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsC : array-like, 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 (*X*)

Return probability estimates for the test vector X.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsC : array-like, 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 (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.naive_bayes.MultinomialNB`

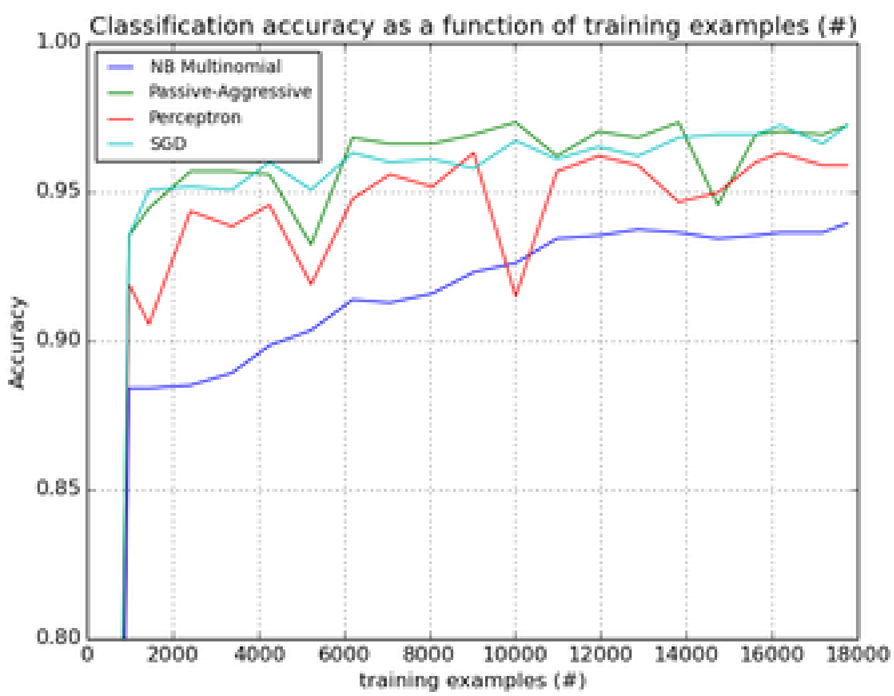


Figure 42.524: *Out-of-core classification of text documents*

42.24.3 `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.

Parametersalpha : float, optional (default=1.0)

Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

binarize : float or None, optional

Figure 42.525: *Classification of text documents: using a MLComp dataset*

Figure 42.526: *Classification of text documents using sparse features*

Threshold for binarizing (mapping to booleans) of sample features. If None, input is presumed to already consist of binary vectors.

fit_prior : boolean

Whether to learn class prior probabilities or not. If false, a uniform prior will be used.

class_prior : array-like, size=[n_classes,]

Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

Attributesclass_log_prior_ : array, shape = [n_classes]

Log probability of each class (smoothed).

feature_log_prob_ : array, shape = [n_classes, n_features]

Empirical log probability of features given a class, P(x_ily).

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.

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.

References

C.D. Manning, P. Raghavan and H. Schuetze (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 234-265. http://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
>>> X = np.random.randint(2, 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(alpha=1.0, binarize=0.0, class_prior=None, fit_prior=True)
>>> print(clf.predict(X[2]))
[3]
```

Methods

fit(X, y[, sample_weight])	Fit Naive Bayes classifier according to X, y
get_params([deep])	Get parameters for this estimator.

Continued on next page

Table 42.171 – continued from previous page

partial_fit(X, y[, classes, sample_weight])	Incremental fit on a batch of samples.
predict(X)	Perform classification on an array of test vectors X.
predict_log_proba(X)	Return log-probability estimates for the test vector X.
predict_proba(X)	Return probability estimates for the test vector X.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

static `__init__` (*alpha=1.0, binarize=0.0, fit_prior=True, class_prior=None*)

`fit` (*X, y, sample_weight=None*)

Fit Naive Bayes classifier according to X, y

ParametersX : {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], optional

Weights applied to individual samples (1. for unweighted).

Returnsself : object

Returns self.

`get_params` (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

`partial_fit` (*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.

ParametersX : {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.

classes : array-like, shape = [n_classes]

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

Weights applied to individual samples (1. for unweighted).

Returnsself : object

Returns self.

predict (*X*)

Perform classification on an array of test vectors X.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsC : array, shape = [n_samples]

Predicted target values for X

predict_log_proba (*X*)

Return log-probability estimates for the test vector X.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsC : array-like, 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 (*X*)

Return probability estimates for the test vector X.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsC : array-like, 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 (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.naive_bayes.BernoulliNB`

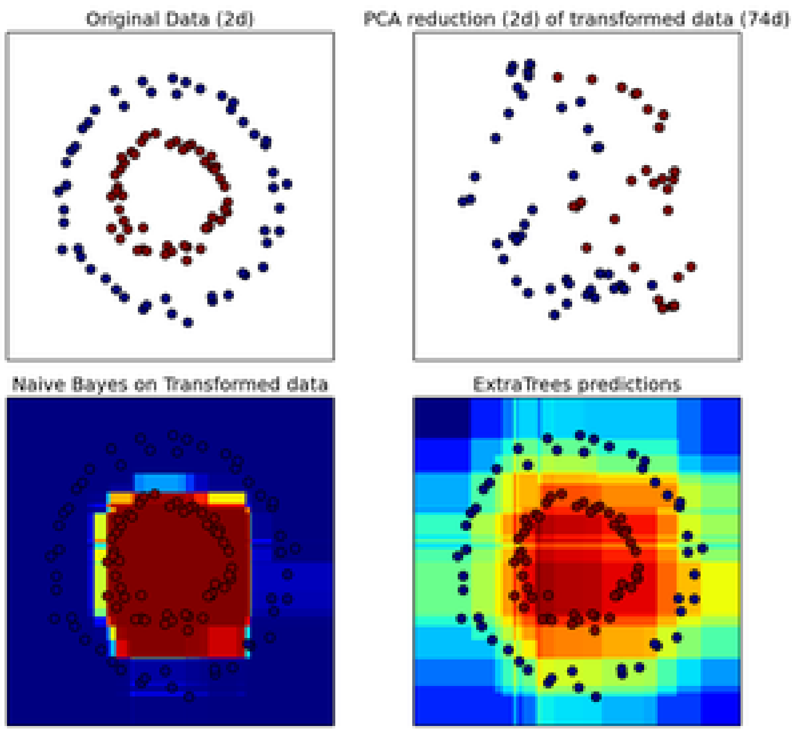


Figure 42.527: *Hashing feature transformation using Totally Random Trees*

Figure 42.528: *Classification of text documents using sparse features*

42.25 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.NearestNeighbors([n_neighbors, ...])	Unsupervised learner for implementing neighbor searches.
neighbors.KNeighborsClassifier([...])	Classifier implementing the k-nearest neighbors vote.
neighbors.RadiusNeighborsClassifier([...])	Classifier implementing a vote among neighbors within a given radi
neighbors.KNeighborsRegressor([n_neighbors, ...])	Regression based on k-nearest neighbors.
neighbors.RadiusNeighborsRegressor([radius, ...])	Regression based on neighbors within a fixed radius.
neighbors.NearestCentroid([metric, ...])	Nearest centroid classifier.
neighbors.BallTree	BallTree for fast generalized N-point problems
neighbors.KDTree	KDTree for fast generalized N-point problems
neighbors.LSHForest([n_estimators, radius, ...])	Performs approximate nearest neighbor search using LSH forest.
neighbors.DistanceMetric	DistanceMetric class
neighbors.KernelDensity([bandwidth, ...])	Kernel Density Estimation

42.25.1 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*, ***kwargs*)

Unsupervised learner for implementing neighbor searches.

Parametersn_neighbors : int, optional (default = 5)

Number of neighbors to use by default for k_neighbors queries.

radius : float, optional (default = 1.0)

Range of parameter space to use by default for :meth'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.

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 : 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', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule']

See the documentation for scipy.spatial.distance for details on these metrics.

metric_params: dict, optional (default = None) :

additional keyword arguments for the metric function.

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`.

http://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]]...)

>>> rng = neigh.radius_neighbors([0, 0, 1.3], 0.4, return_distance=False)
>>> np.asarray(rng[0][0])
array(2)
```

Methods

fit(X[, y])	Fit the model using X as training data
get_params([deep])	Get parameters for this estimator.
kneighbors([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
kneighbors_graph([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
radius_neighbors([X, radius, return_distance])	Finds the neighbors within a given radius of a point or points.
radius_neighbors_graph([X, radius, mode])	Computes the (weighted) graph of Neighbors for points in X
set_params(**params)	Set the parameters of this estimator.

static __init__(*n_neighbors=5, radius=1.0, algorithm='auto', leaf_size=30, metric='minkowski', p=2, metric_params=None, **kwargs*)

fit(*X, y=None*)

Fit the model using X as training data

ParametersX : {array-like, sparse matrix, BallTree, KDTree}

Training data. If array or matrix, shape = [n_samples, n_features]

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

kneighbors(*X=None, n_neighbors=None, return_distance=True*)

Finds the K-neighbors of a point.

Returns distance

ParametersX : array-like, last dimension same as that of fit data, 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.

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

Returnsdist : array

Array representing the lengths to points, only present if return_distance=True

ind : array

Indices of the nearest points in the population matrix.

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]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> 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 (*X=None*, *n_neighbors=None*, *mode='connectivity'*)

Computes the (weighted) graph of k-Neighbors for points in X

ParametersX : array-like, last dimension same as that of fit data, 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.

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.

ReturnsA : sparse matrix in CSR format, shape = [n_samples, 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(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[ 1.,  0.,  1.],
       [ 0.,  1.,  1.],
       [ 1.,  0.,  1.]])
```

radius_neighbors (*X=None*, *radius=None*, *return_distance=True*)

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.

ParametersX : 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

Returnsdist : 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.

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(algorithm='auto', leaf_size=30, ...)
>>> 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(*X=None*, *radius=None*, *mode='connectivity'*)

Computes the (weighted) graph of Neighbors for points in X

Neighborhoods are restricted the points at a distance lower than radius.

ParametersX : array-like, shape = [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

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.

ReturnsA : sparse matrix 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 NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.5)
>>> neigh.fit(X)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.radius_neighbors_graph(X)
>>> A.toarray()
array([[ 1.,  0.,  1.],
       [ 0.,  1.,  0.],
       [ 1.,  0.,  1.]])
```

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.neighbors.NearestNeighbors`

42.25.2 `sklearn.neighbors`.KNeighborsClassifier

class `sklearn.neighbors.`**`KNeighborsClassifier`**(*n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, **kwargs*)

Classifier implementing the k-nearest neighbors vote.

Parametersn_neighbors : int, optional (default = 5)

Number of neighbors to use by default for `k_neighbors` queries.

weights : str or callable

weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.

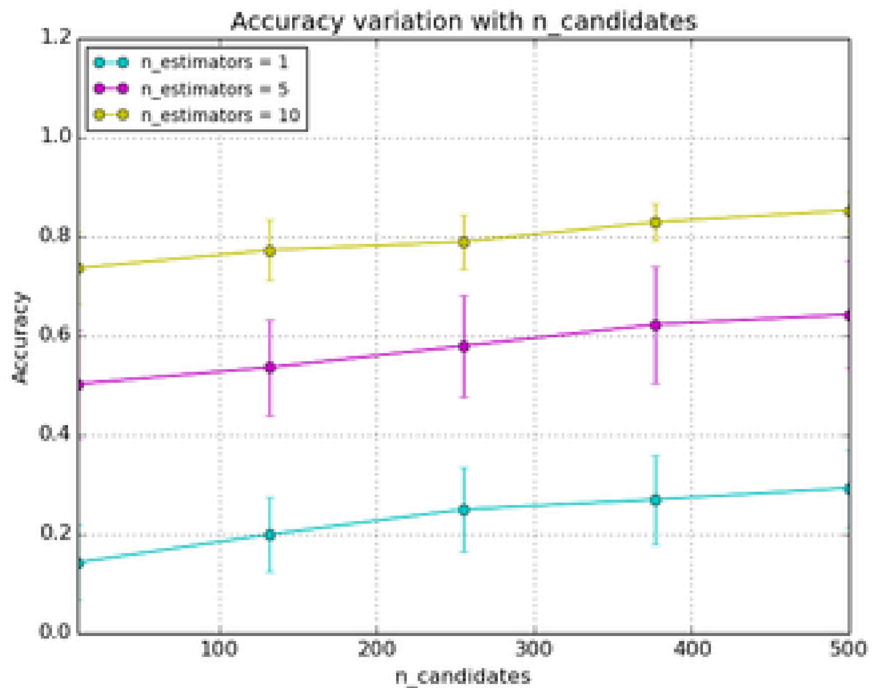


Figure 42.529: *Hyper-parameters of Approximate Nearest Neighbors*

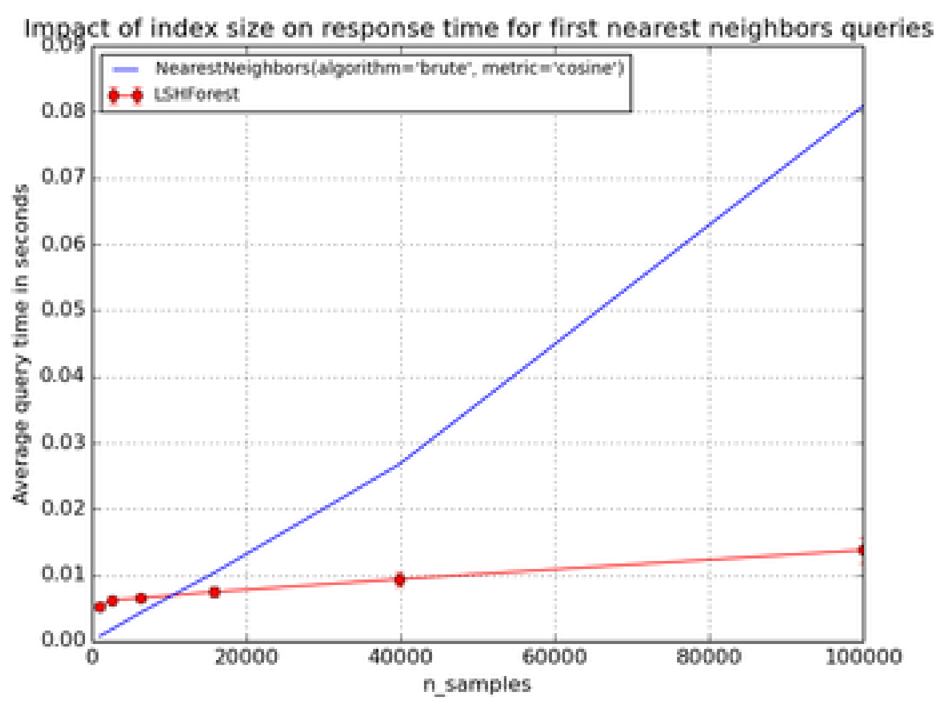


Figure 42.530: *Scalability of Approximate Nearest Neighbors*

- '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.

metric : string or DistanceMetric object (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.

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_params: dict, optional (default = None) :

additional keyword arguments for the metric function.

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 but different labels, the results will depend on the ordering of the training data.

http://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`(X, y)	Fit the model using X as training data and y as target values
`get_params`([deep])	Get parameters for this estimator.
`kneighbors`([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
`kneighbors_graph`([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
`predict`(X)	Predict the class labels for the provided data
`predict_proba`(X)	Return probability estimates for the test data X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.

static __init__(*n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, **kwargs*)

fit(*X, y*)

Fit the model using X as training data and y as target values

ParametersX : {array-like, sparse matrix, BallTree, KDTree}

Training data. If array or matrix, shape = [n_samples, n_features]

y : {array-like, sparse matrix}

Target values of shape = [n_samples] or [n_samples, n_outputs]

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

kneighbors(*X=None, n_neighbors=None, return_distance=True*)

Finds the K-neighbors of a point.

Returns distance

ParametersX : array-like, last dimension same as that of fit data, 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.

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

Returnsdist : array

Array representing the lengths to points, only present if return_distance=True

ind : array

Indices of the nearest points in the population matrix.

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]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> 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(*X=None*, *n_neighbors=None*, *mode='connectivity'*)

Computes the (weighted) graph of k-Neighbors for points in X

ParametersX : array-like, last dimension same as that of fit data, 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.

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.

ReturnsA : sparse matrix in CSR format, shape = [n_samples, 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(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[ 1.,  0.,  1.],
       [ 0.,  1.,  1.],
       [ 1.,  0.,  1.]])
```

static predict (*X*)

Predict the class labels for the provided data

ParametersX : array of shape [n_samples, n_features]

A 2-D array representing the test points.

Returnsy : array of shape [n_samples] or [n_samples, n_outputs]

Class labels for each data sample.

static predict_proba (*X*)

Return probability estimates for the test data X.

ParametersX : array, shape = (n_samples, n_features)

A 2-D array representing the test points.

Returnsp : array of shape = [n_samples, 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 (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.neighbors.KNeighborsClassifier`

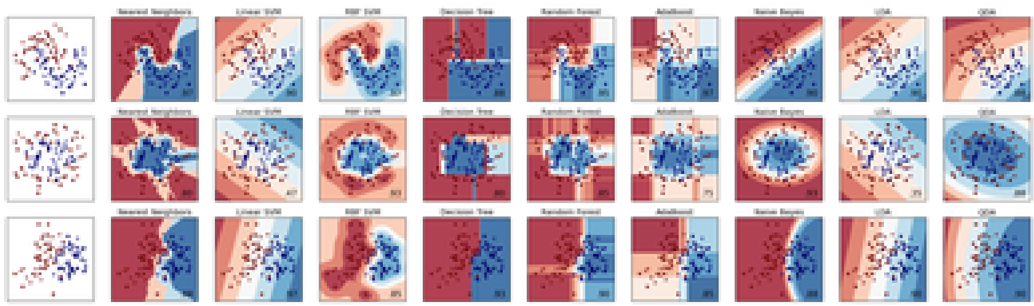


Figure 42.531: *Classifier comparison*

Figure 42.532: *Digits Classification Exercise*

42.25.3 `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*, ***kwargs*)

Classifier implementing a vote among neighbors within a given radius

Figure 42.533: *Nearest Neighbors Classification*

Figure 42.534: *Classification of text documents using sparse features*

Parametersradius : float, optional (default = 1.0)

Range of parameter space to use by default for :meth‘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.

metric : string or DistanceMetric object (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.

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.

outlier_label : int, optional (default = None)

Label, which is given for outlier samples (samples with no neighbors on given radius). If set to None, ValueError is raised, when outlier is detected.

metric_params: dict, optional (default = None) :

additional keyword arguments for the metric function.

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`.

http://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(...)
>>> print(neigh.predict([[1.5]]))
[0]
```

Methods

`fit`(X, y)	Fit the model using X as training data and y as target values
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict the class labels for the provided data
`radius_neighbors`([X, radius, return_distance])	Finds the neighbors within a given radius of a point or points.
`radius_neighbors_graph`([X, radius, mode])	Computes the (weighted) graph of Neighbors for points in X
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', outlier_label=None, metric_params=None, **kwargs*)

`fit`(*X, y*)
Fit the model using X as training data and y as target values

ParametersX : {array-like, sparse matrix, BallTree, KDTree}

Training data. If array or matrix, shape = [n_samples, n_features]

y : {array-like, sparse matrix}

Target values of shape = [n_samples] or [n_samples, n_outputs]

`get_params`(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static `predict`(*X*)
Predict the class labels for the provided data

ParametersX : array of shape [n_samples, n_features]

A 2-D array representing the test points.

Returnsy : array of shape [n_samples] or [n_samples, n_outputs]

Class labels for each data sample.

radius_neighbors (*X=None*, *radius=None*, *return_distance=True*)

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.

ParametersX : 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

Returnsdist : 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.

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(algorithm='auto', leaf_size=30, ...)
>>> 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 (*X=None*, *radius=None*, *mode='connectivity'*)
Computes the (weighted) graph of Neighbors for points in X

Neighborhoods are restricted the points at a distance lower than radius.

ParametersX : array-like, shape = [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

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.

ReturnsA : sparse matrix 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 NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.5)
>>> neigh.fit(X)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.radius_neighbors_graph(X)
>>> A.toarray()
array([[ 1.,  0.,  1.],
       [ 0.,  1.,  0.],
       [ 1.,  0.,  1.]])
```

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

42.25.4 sklearn.neighbors.KNeighborsRegressor

class sklearn.neighbors.**KNeighborsRegressor**(*n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=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.

Parametersn_neighbors : int, optional (default = 5)

Number of neighbors to use by default for `k_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.

metric : string or DistanceMetric object (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.

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_params: dict, optional (default = None) :

additional keyword arguments for the metric function.

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 but different labels, the results will depend on the ordering of the training data.

http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import KNeighborsRegressor
>>> neigh = KNeighborsRegressor(n_neighbors=2)
>>> neigh.fit(X, y)
KNeighborsRegressor(...)
>>> print(neigh.predict([[1.5]]))
[ 0.5]
```

Methods

`fit`(X, y)	Fit the model using X as training data and y as target values
`get_params`([deep])	Get parameters for this estimator.
`kneighbors`([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
`kneighbors_graph`([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
`predict`(X)	Predict the target for the provided data
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, ***kwargs*)

`fit`(*X*, *y*)

Fit the model using X as training data and y as target values

ParametersX : {array-like, sparse matrix, BallTree, KDTree}

Training data. If array or matrix, shape = [n_samples, n_features]

y : {array-like, sparse matrix}

Target values, array of float values, shape = [n_samples]or [n_samples, n_outputs]

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

kneighbors(*X=None*, *n_neighbors=None*, *return_distance=True*)

Finds the K-neighbors of a point.

Returns distance

ParametersX : array-like, last dimension same as that of fit data, 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.

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

Returnsdist : array

Array representing the lengths to points, only present if return_distance=True

ind : array

Indices of the nearest points in the population matrix.

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]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> 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 (*X=None*, *n_neighbors=None*, *mode='connectivity'*)
Computes the (weighted) graph of k-Neighbors for points in X

ParametersX : array-like, last dimension same as that of fit data, 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.

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.

ReturnsA : sparse matrix in CSR format, shape = [n_samples, 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(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[ 1.,  0.,  1.],
       [ 0.,  1.,  1.],
       [ 1.,  0.,  1.]])
```

static **predict** (*X*)
Predict the target for the provided data

ParametersX : array or matrix, shape = [n_samples, n_features]

Returnsy : array of int, shape = [n_samples] or [n_samples, n_outputs]

Target values

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.neighbors.KNeighborsRegressor`

Figure 42.535: *Face completion with a multi-output estimators*

42.25.5 `sklearn.neighbors`.RadiusNeighborsRegressor

class `sklearn.neighbors.`**`RadiusNeighborsRegressor`**(*radius=1.0*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=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.

Parametersradius : float, optional (default = 1.0)

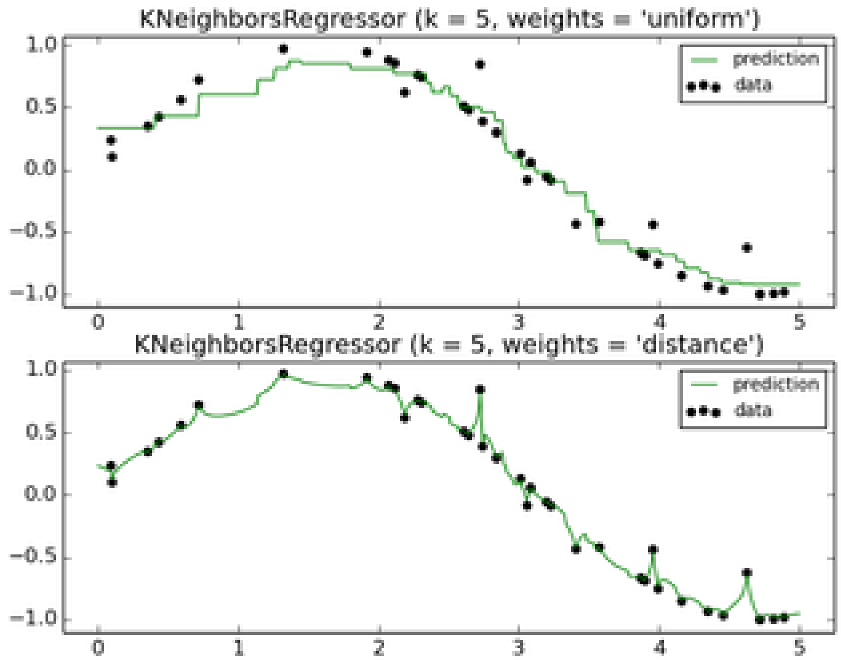


Figure 42.536: *Nearest Neighbors regression*

Range of parameter space to use by default for :meth'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.

metric : string or DistanceMetric object (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.

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_params: dict, optional (default = None) :

additional keyword arguments for the metric function.

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`.

http://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`(X, y)	Fit the model using X as training data and y as target values
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict the target for the provided data
`radius_neighbors`([X, radius, return_distance])	Finds the neighbors within a given radius of a point or points.
`radius_neighbors_graph`([X, radius, mode])	Computes the (weighted) graph of Neighbors for points in X
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*radius=1.0*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, ***kwargs*)

fit (*X*, *y*)
Fit the model using X as training data and y as target values

ParametersX : {array-like, sparse matrix, BallTree, KDTree}

Training data. If array or matrix, shape = [n_samples, n_features]

y : {array-like, sparse matrix}

Target values, array of float values, shape = [n_samples]or [n_samples, n_outputs]

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static predict (*X*)
Predict the target for the provided data

ParametersX : array or matrix, shape = [n_samples, n_features]

Returnsy : array of int, shape = [n_samples] or [n_samples, n_outputs]

Target values

radius_neighbors (*X=None*, *radius=None*, *return_distance=True*)
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.

ParametersX : 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

Returnsdist : 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.

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(algorithm='auto', leaf_size=30, ...)
>>> 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(*X=None*, *radius=None*, *mode='connectivity'*)

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, shape = [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

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.

Returns A : sparse matrix 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 NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.5)
>>> neigh.fit(X)
```

```
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.radius_neighbors_graph(X)
>>> A.toarray()
array([[ 1.,  0.,  1.],
       [ 0.,  1.,  0.],
       [ 1.,  0.,  1.]])
```

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

42.25.6 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.

Parametersmetric: 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.

Attributescentroids_ : array-like, shape = [n_classes, n_features]

Centroid of each class

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.nearest_centroid 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(metric='euclidean', shrink_threshold=None)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

`fit`(X, y)	Fit the NearestCentroid model according to the given training data.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Perform classification on an array of test vectors X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*metric='euclidean'*, *shrink_threshold=None*)

static `fit`(*X*, *y*)

Fit the NearestCentroid model according to the given training data.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Training vector, where n_samples in 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`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **predict** (*X*)

Perform classification on an array of test vectors X.

The predicted class C for each sample in X is returned.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsC : array, 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 (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.neighbors.NearestCentroid`

42.25.7 `sklearn.neighbors`.BallTree

class `sklearn.neighbors.`**`BallTree`**

BallTree for fast generalized N-point problems

BallTree(X, leaf_size=40, metric='minkowski', **kwargs)

ParametersX : array-like, shape = [n_samples, n_features]

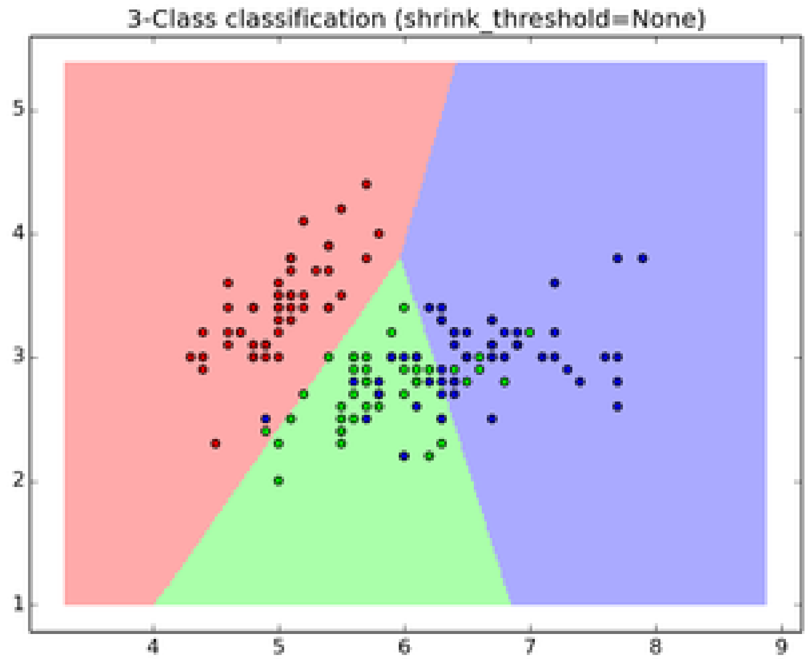


Figure 42.537: *Nearest Centroid Classification*

Figure 42.538: *Classification of text documents using sparse 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 = 20)

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. :

Attributesdata : np.ndarray

The training data

Examples

Query for k-nearest neighbors

```
>>> import numpy as np

>>> np.random.seed(0)
>>> X = np.random.random((10, 3))  # 10 points in 3 dimensions
>>> tree = BallTree(X, leaf_size=2)
>>> dist, ind = tree.query(X[0], k=3)
>>> 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
>>> np.random.seed(0)
>>> X = np.random.random((10, 3))  # 10 points in 3 dimensions
>>> tree = BallTree(X, leaf_size=2)
>>> s = pickle.dumps(tree)
>>> tree_copy = pickle.loads(s)
>>> dist, ind = tree_copy.query(X[0], k=3)
>>> 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
>>> np.random.seed(0)
>>> X = np.random.random((10, 3))  # 10 points in 3 dimensions
>>> tree = BallTree(X, leaf_size=2)
>>> print tree.query_radius(X[0], r=0.3, count_only=True)
3
>>> ind = tree.query_radius(X[0], r=0.3)
>>> print ind  # indices of neighbors within distance 0.3
[3 0 1]
```

Compute a gaussian kernel density estimate:

```
>>> import numpy as np
>>> np.random.seed(1)
>>> X = np.random.random((100, 3))
>>> tree = BallTree(X)
>>> 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
>>> np.random.seed(0)
>>> X = np.random.random((30, 3))
>>> r = np.linspace(0, 1, 5)
>>> tree = BallTree(X)
>>> tree.two_point_correlation(X, r)
array([ 30,  62, 278, 580, 820])
```

Methods

`get_arrays`
`get_n_calls`
`get_tree_stats`
`kernel_density`(self, X, h[, kernel, atol, ...])
`query`(X[, k, return_distance, dualtree, ...])
`query_radius`
`reset_n_calls`
`two_point_correlation`

`__init__`()

42.25.8 `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, 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 = 20)

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. :

Attributesdata : np.ndarray

The training data

Examples

Query for k-nearest neighbors

```
>>> import numpy as np

>>> np.random.seed(0)
>>> X = np.random.random((10, 3))  # 10 points in 3 dimensions
>>> tree = KDTree(X, leaf_size=2)
>>> dist, ind = tree.query(X[0], k=3)
>>> 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
>>> np.random.seed(0)
>>> X = np.random.random((10, 3))  # 10 points in 3 dimensions
>>> tree = KDTree(X, leaf_size=2)
>>> s = pickle.dumps(tree)
>>> tree_copy = pickle.loads(s)
>>> dist, ind = tree_copy.query(X[0], k=3)
>>> 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
>>> np.random.seed(0)
>>> X = np.random.random((10, 3))  # 10 points in 3 dimensions
>>> tree = KDTree(X, leaf_size=2)
>>> print tree.query_radius(X[0], r=0.3, count_only=True)
```

```
3
>>> ind = tree.query_radius(X[0], r=0.3)
>>> print ind  # indices of neighbors within distance 0.3
[3 0 1]
```

Compute a gaussian kernel density estimate:

```
>>> import numpy as np
>>> np.random.seed(1)
>>> X = np.random.random((100, 3))
>>> tree = KDTree(X)
>>> 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
>>> np.random.seed(0)
>>> X = np.random.random((30, 3))
>>> r = np.linspace(0, 1, 5)
>>> tree = KDTree(X)
>>> tree.two_point_correlation(X, r)
array([ 30,  62, 278, 580, 820])
```

Methods

`get_arrays`
`get_n_calls`
`get_tree_stats`
`kernel_density`(self, X, h[, kernel, atol, ...])
`query`(X[, k, return_distance, dualtree, ...])
`query_radius`
`reset_n_calls`
`two_point_correlation`

`__init__`()

42.25.9 `sklearn.neighbors`.LSHForest

class `sklearn.neighbors.`**`LSHForest`**(*n_estimators=10*, *radius=1.0*, *n_candidates=50*, *n_neighbors=5*, *min_hash_match=4*, *radius_cutoff_ratio=0.9*, *random_state=None*)

Performs approximate nearest neighbor search using LSH forest.

LSH Forest: Locality Sensitive Hashing forest [1] is an alternative method for vanilla approximate nearest neighbor search methods. LSH forest data structure has been implemented using sorted arrays and binary search and 32 bit fixed-length hashes. Random projection is used as the hash family which approximates cosine distance.

The cosine distance is defined as `1 - cosine_similarity`: the lowest value is 0 (identical point) but it is bounded above by 2 for the farthest points. Its value does not depend on the norm of the vector points but only on their relative angles.

Parametersn_estimators : int (default = 10)

Number of trees in the LSH Forest.

min_hash_match : int (default = 4)

lowest hash length to be searched when candidate selection is performed for nearest neighbors.

n_candidates : int (default = 10)

Minimum number of candidates evaluated per estimator, assuming enough items meet the *min_hash_match* constraint.

n_neighbors : int (default = 5)

Number of neighbors to be returned from query function when it is not provided to the `kneighbors` method.

radius : float, optinal (default = 1.0)

Radius from the data point to its neighbors. This is the parameter space to use by default for the :meth'radius_neighbors' queries.

radius_cutoff_ratio : float, optional (default = 0.9)

A value ranges from 0 to 1. Radius neighbors will be searched until the ratio between total neighbors within the radius and the total candidates becomes less than this value unless it is terminated by hash length reaching *min_hash_match*.

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*.

Attributeshash_functions_ : list of GaussianRandomProjectionHash objects

Hash function g(p,x) for a tree is an array of 32 randomly generated float arrays with the same dimenstion as the data set. This array is stored in GaussianRandomProjectionHash object and can be obtained from `components_` attribute.

trees_ : array, shape (n_estimators, n_samples)

Each tree (corresponding to a hash function) contains an array of sorted hashed values. The array representation may change in future versions.

original_indices_ : array, shape (n_estimators, n_samples)

Original indices of sorted hashed values in the fitted index.

References

[R187]

Examples

```
>>> from sklearn.neighbors import LSHForest

>>> X_train = [[5, 5, 2], [21, 5, 5], [1, 1, 1], [8, 9, 1], [6, 10, 2]]
>>> X_test = [[9, 1, 6], [3, 1, 10], [7, 10, 3]]
>>> lshf = LSHForest()
>>> lshf.fit(X_train)
```

```
LSHForest(min_hash_match=4, n_candidates=50, n_estimators=10,
          n_neighbors=5, radius=1.0, radius_cutoff_ratio=0.9,
          random_state=None)
>>> distances, indices = lshf.kneighbors(X_test, n_neighbors=2)
>>> distances
array([[ 0.069...,  0.149...],
       [ 0.229...,  0.481...],
       [ 0.004...,  0.014...]])
>>> indices
array([[1, 2],
       [2, 0],
       [4, 0]])
```

Methods

fit(X[, y])	Fit the LSH forest on the data.
get_params([deep])	Get parameters for this estimator.
kneighbors(X[, n_neighbors, return_distance])	Returns n_neighbors of approximate nearest neighbors.
kneighbors_graph([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
partial_fit(X[, y])	Inserts new data into the already fitted LSH Forest.
radius_neighbors(X[, radius, return_distance])	Finds the neighbors within a given radius of a point or points.
radius_neighbors_graph([X, radius, mode])	Computes the (weighted) graph of Neighbors for points in X
set_params(**params)	Set the parameters of this estimator.

static `__init__`(*n_estimators=10*, *radius=1.0*, *n_candidates=50*, *n_neighbors=5*, *min_hash_match=4*, *radius_cutoff_ratio=0.9*, *random_state=None*)

static `fit`(*X*, *y=None*)

Fit the LSH forest on the data.

This creates binary hashes of input data points by getting the dot product of input points and hash_function then transforming the projection into a binary string array based on the sign (positive/negative) of the projection. A sorted array of binary hashes is created.

ParametersX : array_like or sparse (CSR) matrix, shape (n_samples, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

Returnsself : object

Returns self.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static `kneighbors`(*X*, *n_neighbors=None*, *return_distance=True*)

Returns n_neighbors of approximate nearest neighbors.

ParametersX : array_like or sparse (CSR) matrix, shape (n_samples, n_features)

List of n_features-dimensional data points. Each row corresponds to a single query.

n_neighbors : int, opitonal (default = None)

Number of neighbors required. If not provided, this will return the number specified at the initialization.

return_distance : boolean, optional (default = False)

Returns the distances of neighbors if set to True.

Returnsdist : array, shape (n_samples, n_neighbors)

Array representing the cosine distances to each point, only present if return_distance=True.

ind : array, shape (n_samples, n_neighbors)

Indices of the approximate nearest points in the population matrix.

kneighbors_graph (*X=None*, *n_neighbors=None*, *mode='connectivity'*)

Computes the (weighted) graph of k-Neighbors for points in X

ParametersX : array-like, last dimension same as that of fit data, 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.

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.

ReturnsA : sparse matrix in CSR format, shape = [n_samples, 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(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[ 1.,  0.,  1.],
       [ 0.,  1.,  1.],
       [ 1.,  0.,  1.]])
```

static **partial_fit** (*X*, *y=None*)

Inserts new data into the already fitted LSH Forest. Cost is proportional to new total size, so additions should be batched.

ParametersX : array_like or sparse (CSR) matrix, shape (n_samples, n_features)

New data point to be inserted into the LSH Forest.

static radius_neighbors (*X*, *radius=None*, *return_distance=True*)

Finds the neighbors within a given radius of a point or points.

Return the indices and distances of some points 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.

LSH Forest being an approximate method, some true neighbors from the indexed dataset might be missing from the results.

ParametersX : array_like or sparse (CSR) matrix, shape (n_samples, n_features)

List of n_features-dimensional data points. Each row corresponds to a single query.

radius : float

Limiting distance of neighbors to return. (default is the value passed to the constructor).

return_distance : boolean, optional (default = False)

Returns the distances of neighbors if set to True.

Returnsdist : array, shape (n_samples,) of arrays

Each element is an array representing the cosine distances to some points found within `radius` of the respective query. Only present if `return_distance=True`.

ind : array, shape (n_samples,) of arrays

Each element is an array of indices for neighbors within `radius` of the respective query.

radius_neighbors_graph (*X=None*, *radius=None*, *mode='connectivity'*)

Computes the (weighted) graph of Neighbors for points in X

Neighborhoods are restricted the points at a distance lower than radius.

ParametersX : array-like, shape = [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

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.

ReturnsA : sparse matrix 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 NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.5)
>>> neigh.fit(X)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.radius_neighbors_graph(X)
>>> A.toarray()
array([[ 1.,  0.,  1.],
       [ 0.,  1.,  0.],
       [ 1.,  0.,  1.]])
```

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.neighbors.LSHForest`

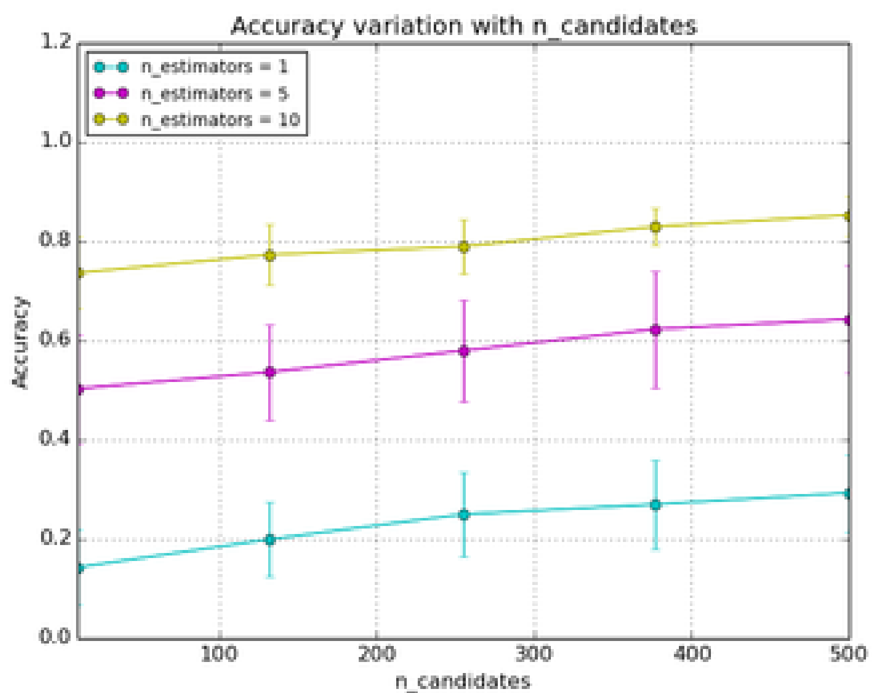


Figure 42.539: *Hyper-parameters of Approximate Nearest Neighbors*

42.25.10 `sklearn.neighbors`.DistanceMetric

class `sklearn.neighbors.`**`DistanceMetric`**

DistanceMetric class

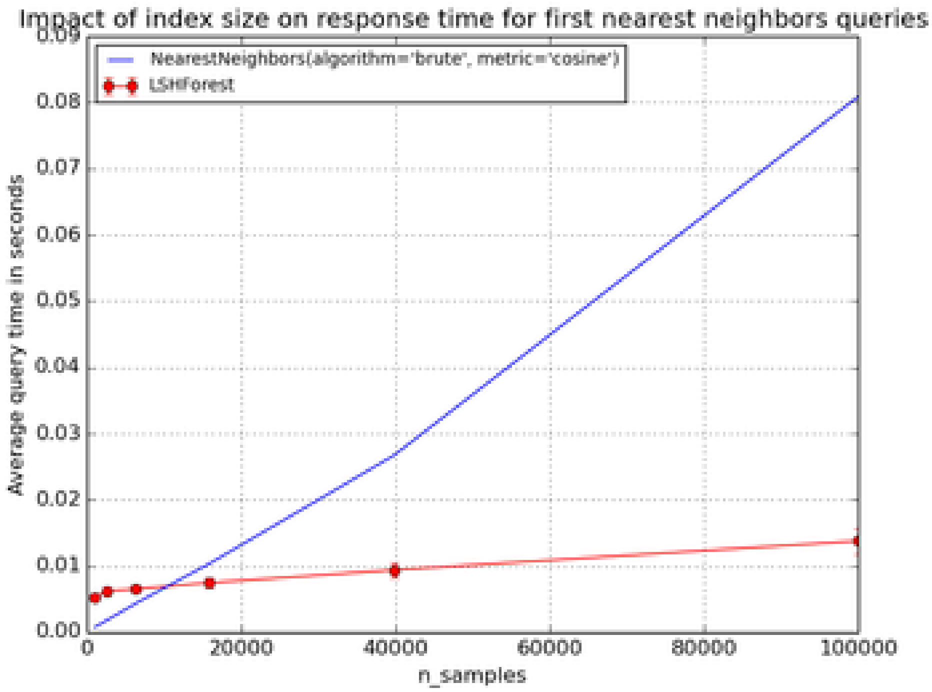


Figure 42.540: *Scalability of Approximate Nearest Neighbors*

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	•	`sum(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:

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
“maching”	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`
`get_metric`
`pairwise`
`rdist_to_dist`

`__init__`()

42.25.11 `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

Parametersbandwidth : float

The bandwidth of the kernel.

algorithm : string

The tree algorithm to use. Valid options are ['kd_tree'|'ball_tree'|'auto']. Default is 'auto'.

kernel : string

The kernel to use. Valid kernels are ['gaussian'|'tophat'|'epanechnikov'|'exponential'|'linear'|'cosine'] Default is 'gaussian'.

metric : string

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 : boolean

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`.

Methods

`fit`(X[, y])	Fit the Kernel Density model on the data.
`get_params`([deep])	Get parameters for this estimator.
`sample`([n_samples, random_state])	Generate random samples from the model.
`score`(X[, y])	Compute the total log probability under the model.
`score_samples`(X)	Evaluate the density model on the data.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*bandwidth=1.0, algorithm='auto', kernel='gaussian', metric='euclidean', atol=0, rtol=0, breadth_first=True, leaf_size=40, metric_params=None*)

static `fit`(*X, y=None*)

Fit the Kernel Density model on the data.

ParametersX : array_like, shape (n_samples, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static sample (*n_samples=1*, *random_state=None*)

Generate random samples from the model.

Currently, this is implemented only for gaussian and tophat kernels.

Parametersn_samples : int, optional

Number of samples to generate. Defaults to 1.

random_state : RandomState or an int seed (0 by default)

A random number generator instance.

ReturnsX : array_like, shape (n_samples, n_features)

List of samples.

static score (*X*, *y=None*)

Compute the total log probability under the model.

ParametersX : array_like, shape (n_samples, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

Returnslogprob : float

Total log-likelihood of the data in X.

static score_samples (*X*)

Evaluate the density model on the data.

ParametersX : array_like, shape (n_samples, n_features)

An array of points to query. Last dimension should match dimension of training data (n_features).

Returnsdensity : ndarray, shape (n_samples,)

The array of log(density) evaluations.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.neighbors.KernelDensity`

`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

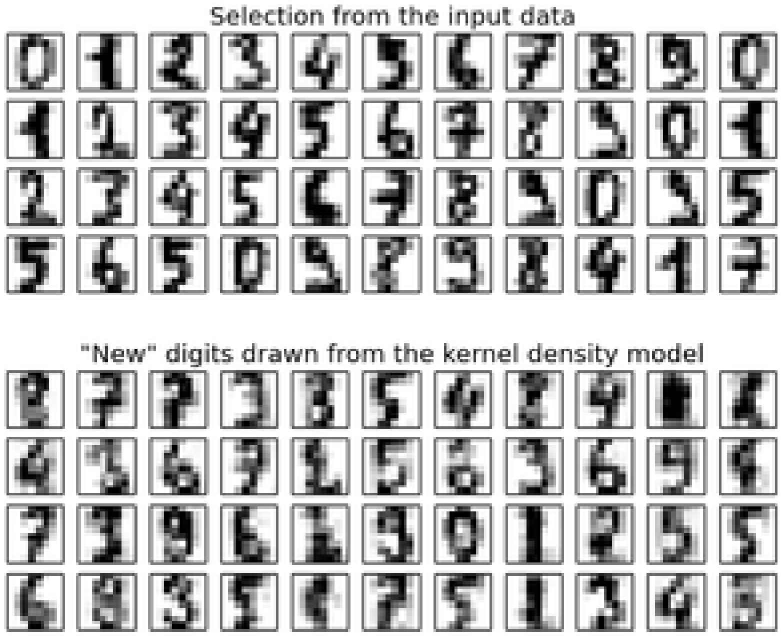


Figure 42.541: *Kernel Density Estimation*

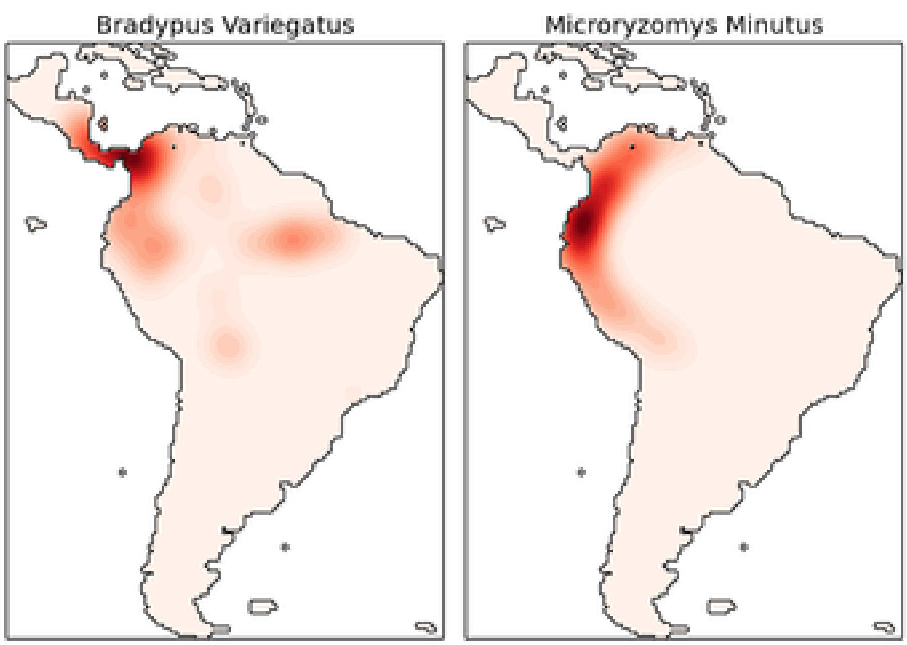


Figure 42.542: *Kernel Density Estimate of Species Distributions*

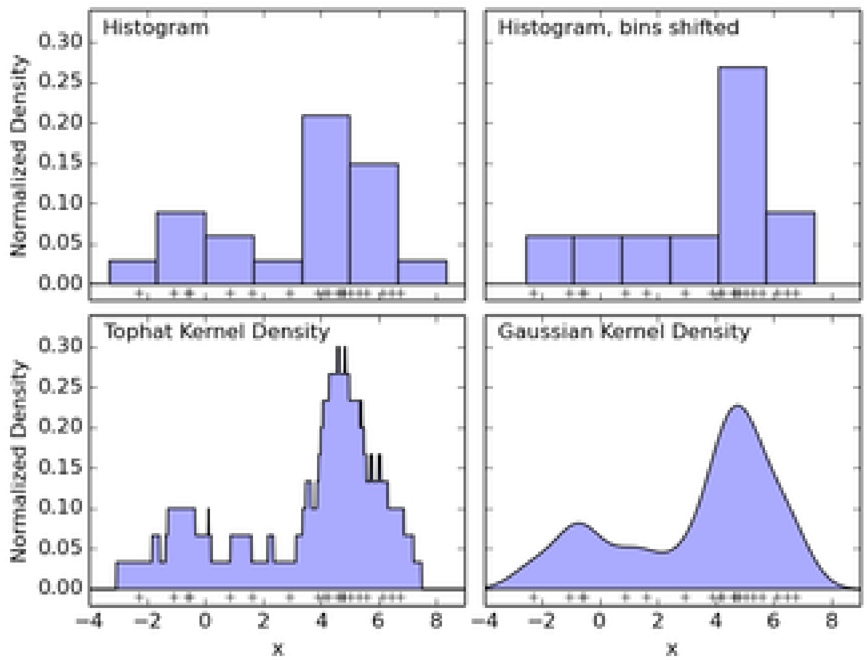


Figure 42.543: *Simple 1D Kernel Density Estimation*

42.25.12 `sklearn.neighbors`.kneighbors_graph

`sklearn.neighbors.`**`kneighbors_graph`**(*X*, *n_neighbors*, *mode='connectivity'*, *metric='minkowski'*, *p=2*, *metric_params=None*, *include_self=None*)

Computes the (weighted) graph of k-Neighbors for points in X

Parameters X : array-like or BallTree, shape = [n_samples, n_features]

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, in 'distance' the edges are Euclidean distance between points.

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.)

include_self: bool, default backward-compatible. :

Whether or not to mark each sample as the first nearest neighbor to itself. If *None*, then True is used for mode='connectivity' and False for mode='distance' as this will preserve backwards compatibilty. From version 0.18, the default value will be False, irrespective of the value of *mode*.

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.

ReturnsA : sparse matrix 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)
>>> A.toarray()
array([[ 1.,  0.,  1.],
       [ 0.,  1.,  1.],
       [ 1.,  0.,  1.]])
```

Examples using `sklearn.neighbors.kneighbors_graph`

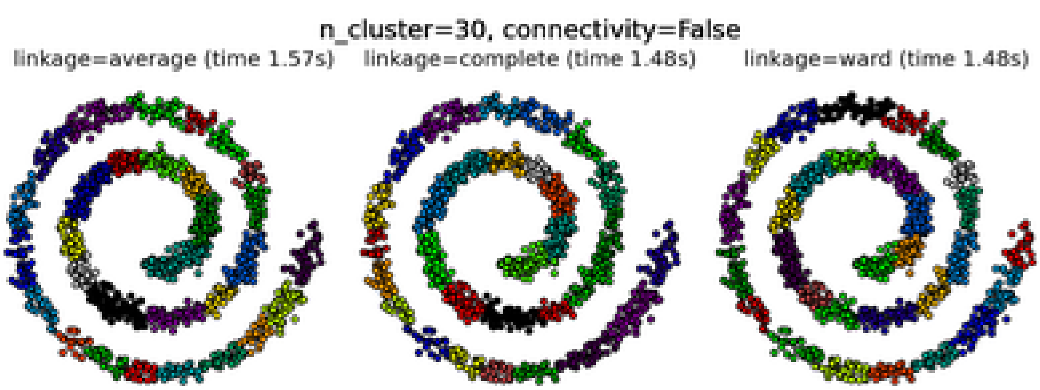


Figure 42.544: *Agglomerative clustering with and without structure*

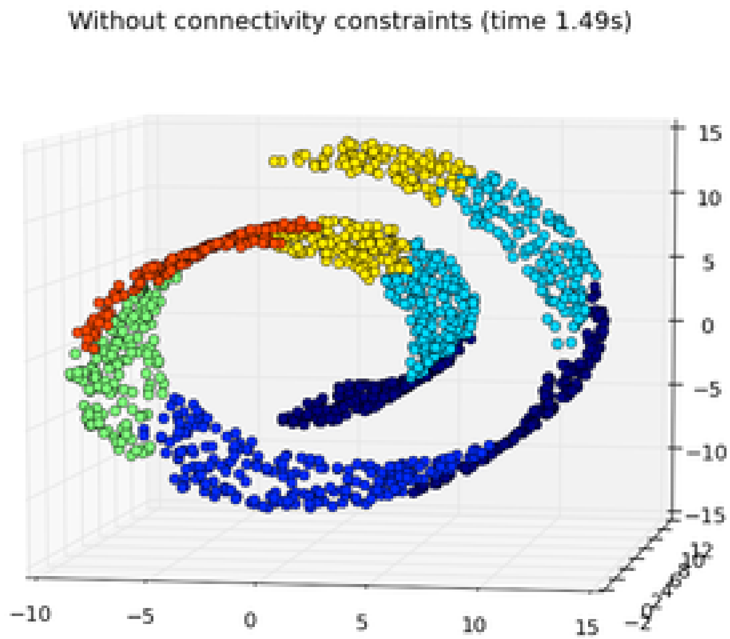


Figure 42.545: *Hierarchical clustering: structured vs unstructured ward*

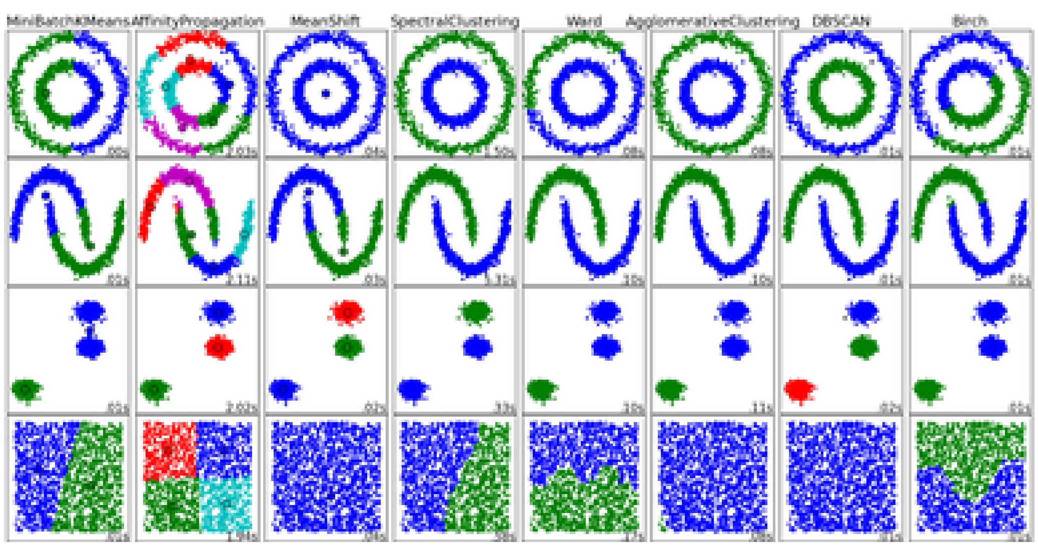


Figure 42.546: *Comparing different clustering algorithms on toy datasets*

42.25.13 sklearn.neighbors.radius_neighbors_graph

sklearn.neighbors.**radius_neighbors_graph**(*X*, *radius*, *mode='connectivity'*, *metric='minkowski'*, *p=2*, *metric_params=None*, *include_self=None*)

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 or BallTree, shape = [n_samples, n_features]

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, in 'distance' the edges are Euclidean distance between points.

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.)

include_self: bool, default None :

Whether or not to mark each sample as the first nearest neighbor to itself. If *None*, then True is used for mode='connectivity' and False for mode='distance' as this will preserve backwards compatibilty. From version 0.18, the default value will be False, irrespective of the value of *mode*.

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.

Returns **A** : sparse matrix 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)
>>> A.toarray()
array([[ 1.,  0.,  1.],
       [ 0.,  1.,  0.],
       [ 1.,  0.,  1.]])
```

42.26 `sklearn.neural_network`: Neural network models

The `sklearn.neural_network` module includes models based on neural networks.

User guide: See the *Neural network models (unsupervised)* section for further details.

`neural_network.BernoulliRBM`([n_components, ...])	Bernoulli Restricted Boltzmann Machine (RBM).

42.26.1 `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 hiddens. 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.

Parametersn_components : int, optional

Number of binary hidden units.

learning_rate : float, optional

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, optional

Number of examples per minibatch.

n_iter : int, optional

Number of iterations/sweeps over the training dataset to perform during training.

verbose : int, optional

The verbosity level. The default, zero, means silent mode.

random_state : integer or numpy.RandomState, optional

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.

Attributesintercept_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.

References

[1] Hinton, G. E., Osindero, S. and Teh, Y. A fast learning algorithm fordeep belief nets. Neural Computation 18, pp 1527-1554. http://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(batch_size=10, learning_rate=0.1, n_components=2, n_iter=10,
       random_state=None, verbose=0)
```

Methods

`fit`(X[, y])	Fit the model to the data X.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`gibbs`(v)	Perform one Gibbs sampling step.
`partial_fit`(X[, y])	Fit the model to the data X which should contain a partial segment of the data.
`score_samples`(X)	Compute the pseudo-likelihood of X.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Compute the hidden layer activation probabilities, P(h=1|v=X).

static `__init__`(*n_components=256*, *learning_rate=0.1*, *batch_size=10*, *n_iter=10*, *verbose=0*, *random_state=None*)

static `fit`(*X*, *y=None*)

Fit the model to the data X.

ParametersX : {array-like, sparse matrix} shape (n_samples, n_features)

Training data.

Returnsself : BernoulliRBM

The fitted model.

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **gibbs** (*v*)

Perform one Gibbs sampling step.

Parametersv : array-like, shape (n_samples, n_features)

Values of the visible layer to start from.

Returnsv_new : array-like, shape (n_samples, n_features)

Values of the visible layer after one Gibbs step.

static **partial_fit** (*X*, *y=None*)

Fit the model to the data X which should contain a partial segment of the data.

ParametersX : array-like, shape (n_samples, n_features)

Training data.

Returnsself : BernoulliRBM

The fitted model.

static **score_samples** (*X*)

Compute the pseudo-likelihood of X.

ParametersX : {array-like, sparse matrix} shape (n_samples, n_features)

Values of the visible layer. Must be all-boolean (not checked).

Returnspseudo_likelihood : array-like, 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 (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **transform** (*X*)

Compute the hidden layer activation probabilities, P(h=1|v=X).

ParametersX : {array-like, sparse matrix} shape (n_samples, n_features)

The data to be transformed.

Returnsh : array, shape (n_samples, n_components)

Latent representations of the data.

Examples using `sklearn.neural_network.BernoulliRBM`

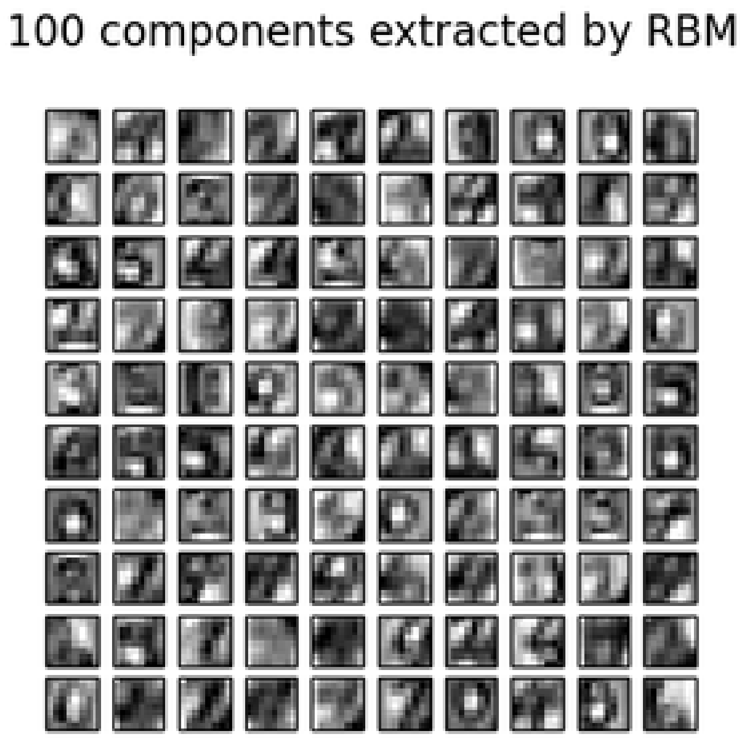


Figure 42.547: *Restricted Boltzmann Machine features for digit classification*

42.27 `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.

42.27.1 `sklearn.calibration`.CalibratedClassifierCV

class `sklearn.calibration.`**`CalibratedClassifierCV`**(*base_estimator=None*, *method='sigmoid'*, *cv=3*)

Probability calibration with isotonic regression or sigmoid.

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 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 disjpint.

Parametersbase_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' | 'isotonic'

The method to use for calibration. Can be 'sigmoid' which corresponds to Platt's method or 'isotonic' which is a non-parameteric 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 or cross-validation generator or "prefit", optional

If an integer is passed, it is the number of folds (default 3). Specific cross-validation objects can be passed, see sklearn.cross_validation module for the list of possible objects. If "prefit" is passed, it is assumed that base_estimator has been fitted already and all data is used for calibration.

Attributesclasses_ : 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

[R103], [R104], [R105], [R106]

Methods

`fit`(X, y[, sample_weight])	Fit the calibrated model
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict the target of new samples.
`predict_proba`(X)	Posterior probabilities of classification
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*base_estimator=None*, *method='sigmoid'*, *cv=3*)

static `fit`(*X*, *y*, *sample_weight=None*)

Fit the calibrated model

ParametersX : array-like, shape (n_samples, n_features)

Training data.

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.

Returnsself : object

Returns an instance of self.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **predict** (*X*)

Predict the target of new samples. Can be different from the prediction of the uncalibrated classifier.

ParametersX : array-like, shape (n_samples, n_features)

The samples.

ReturnsC : array, shape (n_samples,)

The predicted class.

static **predict_proba** (*X*)

Posterior probabilities of classification

This function returns posterior probabilities of classification according to each class on an array of test vectors X.

ParametersX : array-like, shape (n_samples, n_features)

The samples.

ReturnsC : array, shape (n_samples, n_classes)

The predicted probas.

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.calibration.CalibratedClassifierCV`

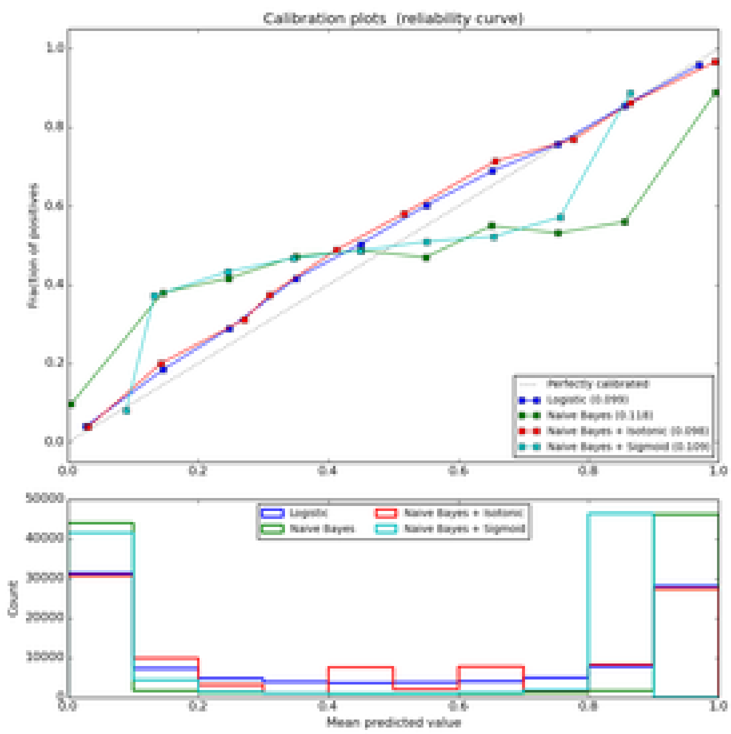


Figure 42.548: *Probability Calibration curves*

`calibration.calibration_curve`(y_true, y_prob)	Compute true and predicted probabilities for a calibration curve.

42.27.2 `sklearn.calibration`.calibration_curve

`sklearn.calibration.`**`calibration_curve`**(*y_true*, *y_prob*, *normalize=False*, *n_bins=5*)

Compute true and predicted probabilities for a calibration curve.

Parametersy_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.

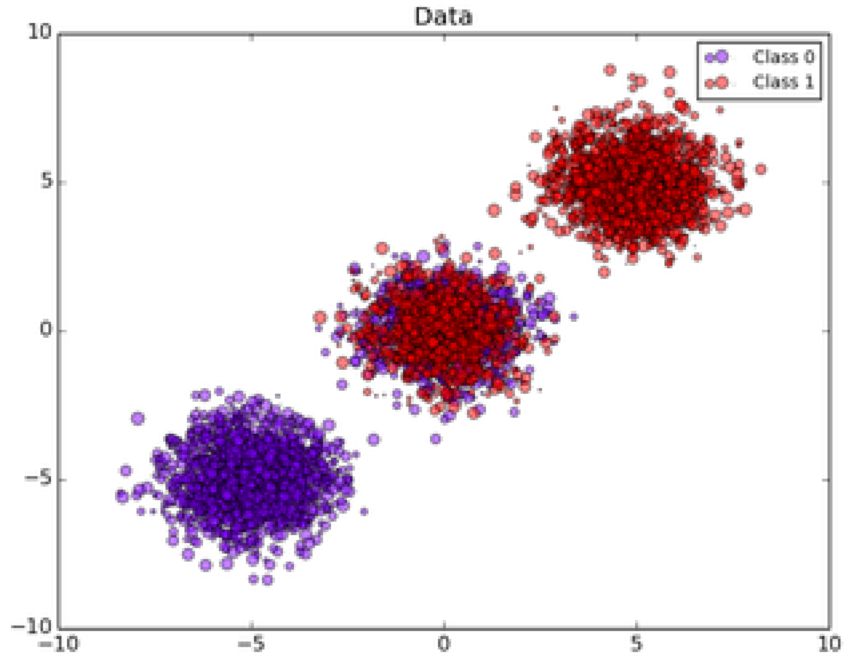


Figure 42.549: *Probability calibration of classifiers*

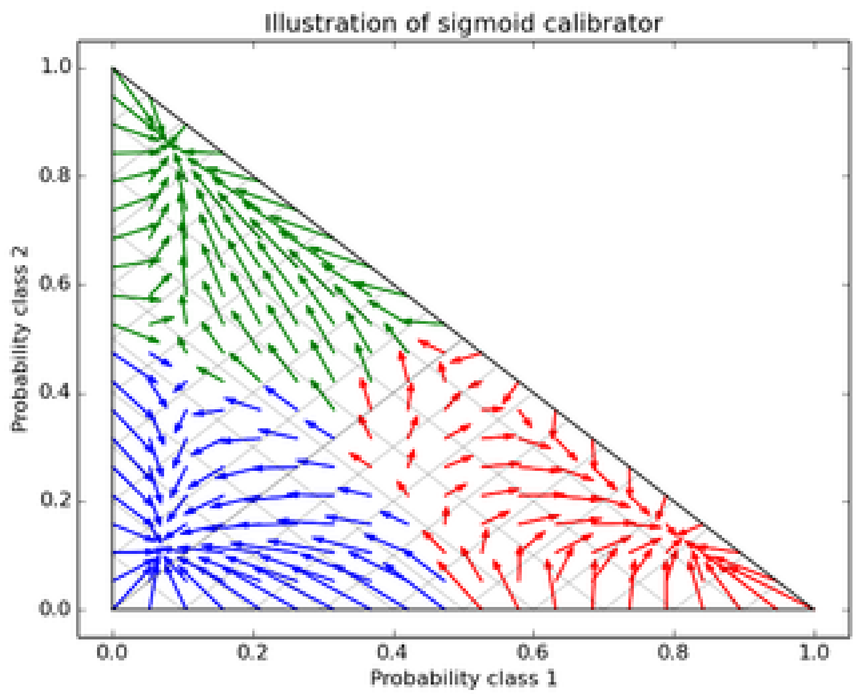


Figure 42.550: *Probability Calibration for 3-class classification*

n_bins : int

Number of bins. A bigger number requires more data.

Returnsprob_true : array, shape (n_bins,)

The true probability in each bin (fraction of positives).

prob_pred : array, shape (n_bins,)

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`

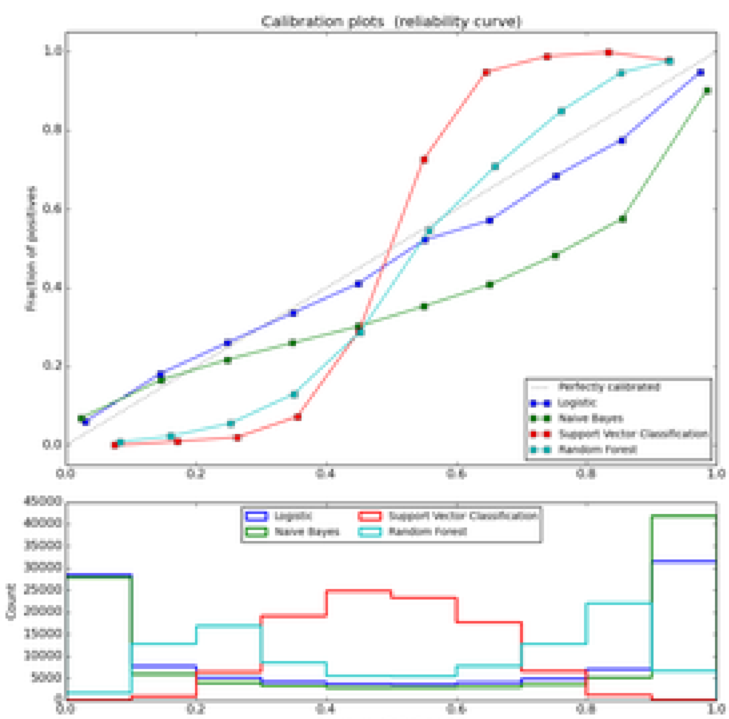


Figure 42.551: *Comparison of Calibration of Classifiers*

42.28 `sklearn.cross_decomposition`: Cross decomposition

User guide: See the *Cross decomposition* section for further details.

`cross_decomposition.PLSRegression([...])`	PLS regression

Table 42.190 – continued from previous page

cross_decomposition.PLSCanonical([...])	PLSCanonical implements the 2 blocks canonical PLS of the original W
cross_decomposition.CCA([n_components, ...])	CCA Canonical Correlation Analysis.
cross_decomposition.PLSSVD([n_components, ...])	Partial Least Square SVD

42.28.1 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".

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]

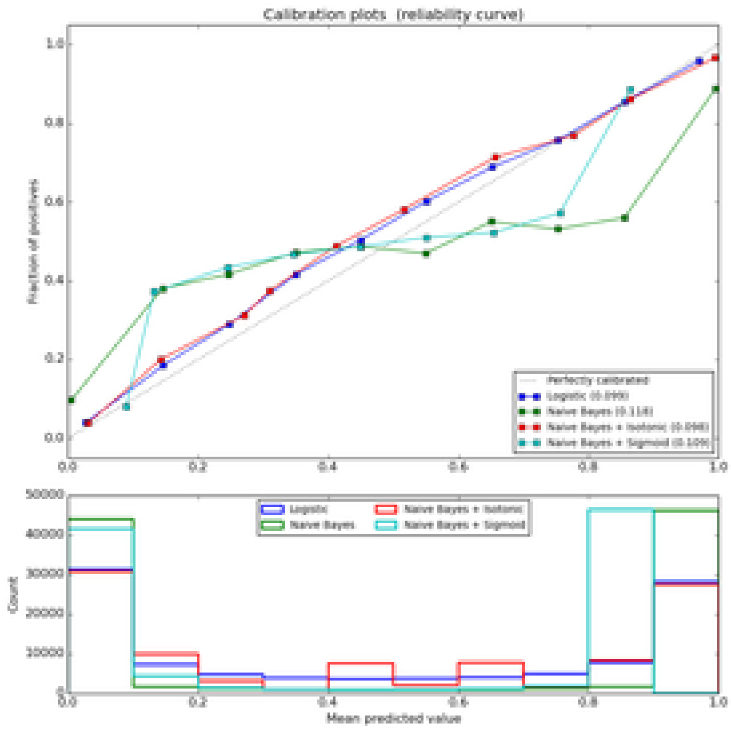


Figure 42.552: *Probability Calibration curves*

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

For each component k, find weights u, v that optimizes: `max corr(Xk u, Yk v) * var(Xk u) var(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(copy=True, max_iter=500, n_components=2, scale=True,
        tol=1e-06)
>>> Y_pred = pls2.predict(X)
```

Methods

fit(X, Y)	Fit model to data.
fit_transform(X[, y])	Learn and apply the dimension reduction on the train data.
get_params([deep])	Get parameters for this estimator.
predict(X[, copy])	Apply the dimension reduction learned on the train data.
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.
transform(X[, Y, copy])	Apply the dimension reduction learned on the train data.

static __init__(*n_components=2*, *scale=True*, *max_iter=500*, *tol=1e-06*, *copy=True*)

fit(*X*, *Y*)

Fit model to data.

ParametersX : array-like, shape = [n_samples, n_features]

Training vectors, where n_samples in the number of samples and n_features is the number of predictors.

Y : array-like of response, shape = [n_samples, n_targets]

Target vectors, where n_samples in the number of samples and n_targets is the number of response variables.

fit_transform(*X*, *y=None*, ***fit_params*)

Learn and apply the dimension reduction on the train data.

ParametersX : array-like of predictors, shape = [n_samples, p]

Training vectors, where n_samples in the number of samples and p is the number of predictors.

Y : array-like of response, shape = [n_samples, q], optional

Training vectors, where n_samples in the number of samples and q is the number of response variables.

copy : boolean, default True

Whether to copy X and Y, or perform in-place normalization.

Returnsx_scores if Y is not given, (x_scores, y_scores) otherwise. :

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*, *copy=True*)

Apply the dimension reduction learned on the train data.

ParametersX : array-like of predictors, shape = [n_samples, p]

Training vectors, where n_samples in the number of samples and p 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 (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*, *Y=None*, *copy=True*)
Apply the dimension reduction learned on the train data.

ParametersX : array-like of predictors, shape = [n_samples, p]

Training vectors, where n_samples in the number of samples and p is the number of predictors.

Y : array-like of response, shape = [n_samples, q], optional

Training vectors, where n_samples in the number of samples and q is the number of response variables.

copy : boolean, default True

Whether to copy X and Y, or perform in-place normalization.

Returnsx_scores if Y is not given, (x_scores, y_scores) otherwise. :

Examples using `sklearn.cross_decomposition.PLSRegression`

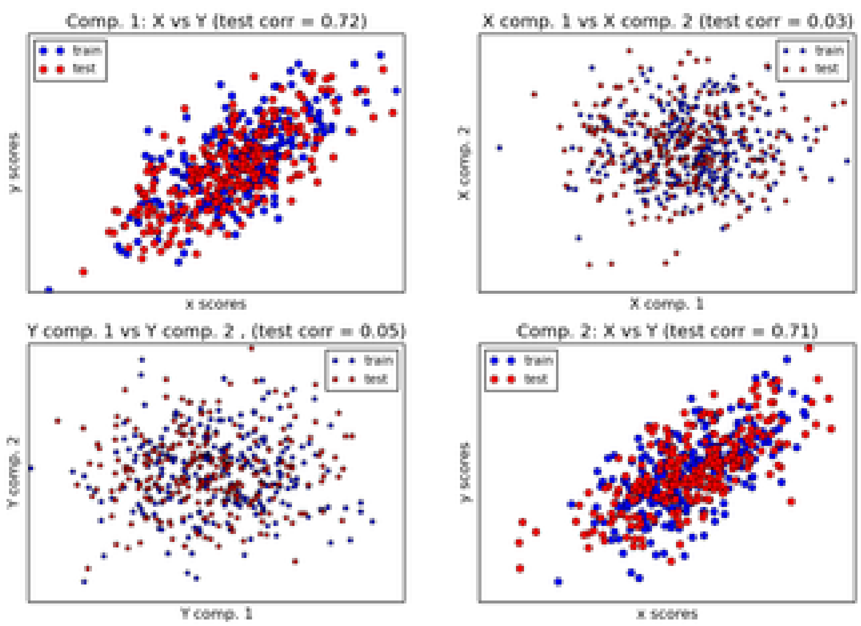


Figure 42.553: *Compare cross decomposition methods*

42.28.2 `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.

Parametersscale : boolean, scale data? (default True)

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

n_components : int, number of components to keep. (default 2).

Attributesx_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

For each component k, find weights u, v that optimize:: max corr(Xk u, Yk v) * var(Xk u) var(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(algorithm='nipals', copy=True, max_iter=500, n_components=2,
             scale=True, tol=1e-06)
>>> X_c, Y_c = plsca.transform(X, Y)
```

Methods

`fit`(X, Y)	Fit model to data.
`fit_transform`(X[, y])	Learn and apply the dimension reduction on the train data.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X[, copy])	Apply the dimension reduction learned on the train data.
`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, Y, copy])	Apply the dimension reduction learned on the train data.

static `__init__`(*n_components=2*, *scale=True*, *algorithm='nipals'*, *max_iter=500*, *tol=1e-06*, *copy=True*)

`fit`(*X*, *Y*)

Fit model to data.

ParametersX : array-like, shape = [n_samples, n_features]

Training vectors, where n_samples in the number of samples and n_features is the number of predictors.

Y : array-like of response, shape = [n_samples, n_targets]

Target vectors, where n_samples in the number of samples and n_targets is the number of response variables.

fit_transform (*X*, *y=None*, ***fit_params*)

Learn and apply the dimension reduction on the train data.

ParametersX : array-like of predictors, shape = [n_samples, p]

Training vectors, where n_samples in the number of samples and p is the number of predictors.

Y : array-like of response, shape = [n_samples, q], optional

Training vectors, where n_samples in the number of samples and q is the number of response variables.

copy : boolean, default True

Whether to copy X and Y, or perform in-place normalization.

Returnsx_scores if Y is not given, (x_scores, y_scores) otherwise. :

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*, *copy=True*)

Apply the dimension reduction learned on the train data.

ParametersX : array-like of predictors, shape = [n_samples, p]

Training vectors, where n_samples in the number of samples and p 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 (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*, *Y=None*, *copy=True*)

Apply the dimension reduction learned on the train data.

ParametersX : array-like of predictors, shape = [n_samples, p]

Training vectors, where n_samples in the number of samples and p is the number of predictors.

Y : array-like of response, shape = [n_samples, q], optional

Training vectors, where n_samples in the number of samples and q is the number of response variables.

copy : boolean, default True

Whether to copy X and Y, or perform in-place normalization.

Returnsx_scores if Y is not given, (x_scores, y_scores) otherwise. :

Examples using `sklearn.cross_decomposition.PLSCanonical`

42.28.3 `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".

Parametersn_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

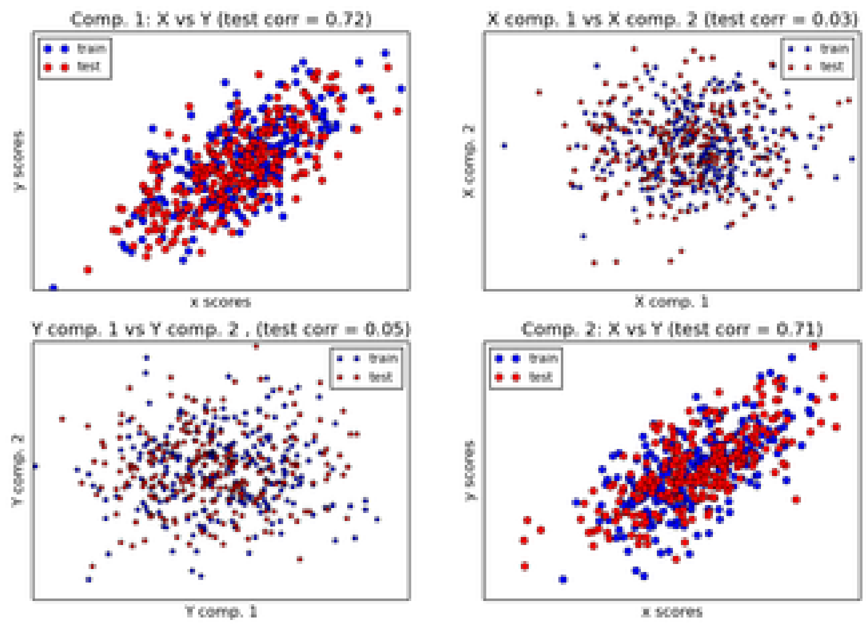


Figure 42.554: *Compare cross decomposition methods*

copy : boolean

Whether the deflation be done on a copy. Let the default value to True unless you don't care about side effects

Attributesx_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(copy=True, max_iter=500, n_components=1, scale=True, tol=1e-06)
>>> X_c, Y_c = cca.transform(X, Y)
```

Methods

fit(X, Y)	Fit model to data.
fit_transform(X[, y])	Learn and apply the dimension reduction on the train data.
get_params([deep])	Get parameters for this estimator.
predict(X[, copy])	Apply the dimension reduction learned on the train data.
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.
transform(X[, Y, copy])	Apply the dimension reduction learned on the train data.

static `__init__`(*n_components=2*, *scale=True*, *max_iter=500*, *tol=1e-06*, *copy=True*)

`fit`(*X*, *Y*)
Fit model to data.

ParametersX : array-like, shape = [n_samples, n_features]

Training vectors, where n_samples in the number of samples and n_features is the number of predictors.

Y : array-like of response, shape = [n_samples, n_targets]

Target vectors, where n_samples in the number of samples and n_targets is the number of response variables.

fit_transform (*X*, *y=None*, ***fit_params*)

Learn and apply the dimension reduction on the train data.

ParametersX : array-like of predictors, shape = [n_samples, p]

Training vectors, where n_samples in the number of samples and p is the number of predictors.

Y : array-like of response, shape = [n_samples, q], optional

Training vectors, where n_samples in the number of samples and q is the number of response variables.

copy : boolean, default True

Whether to copy X and Y, or perform in-place normalization.

Returnsx_scores if Y is not given, (x_scores, y_scores) otherwise. :

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*, *copy=True*)

Apply the dimension reduction learned on the train data.

ParametersX : array-like of predictors, shape = [n_samples, p]

Training vectors, where n_samples in the number of samples and p 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 (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*, *Y=None*, *copy=True*)

Apply the dimension reduction learned on the train data.

ParametersX : array-like of predictors, shape = [n_samples, p]

Training vectors, where n_samples in the number of samples and p is the number of predictors.

Y : array-like of response, shape = [n_samples, q], optional

Training vectors, where n_samples in the number of samples and q is the number of response variables.

copy : boolean, default True

Whether to copy X and Y, or perform in-place normalization.

Returnsx_scores if Y is not given, (x_scores, y_scores) otherwise. :

Examples using `sklearn.cross_decomposition.CCA`

42.28.4 `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.

Parametersn_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.

Attributesx_weights_ : array, [p, n_components]

X block weights vectors.

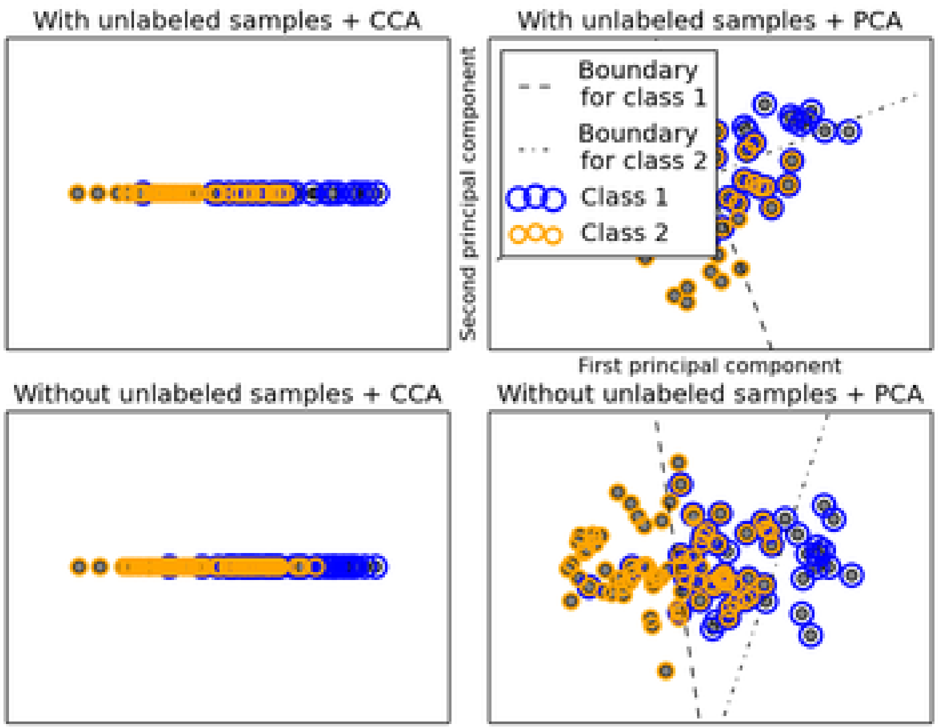


Figure 42.555: *Multilabel classification*

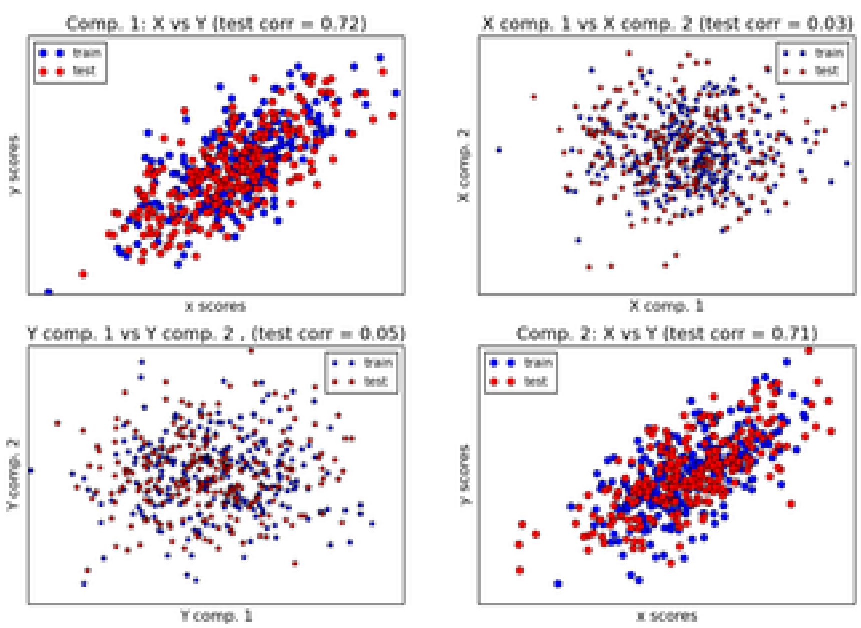


Figure 42.556: *Compare cross decomposition methods*

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

Methods

fit(X, Y)	
fit_transform(X[, y])	Learn and apply the dimension reduction on the train data.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.
transform(X[, Y])	Apply the dimension reduction learned on the train data.

static __init__(*n_components=2*, *scale=True*, *copy=True*)

static fit_transform(*X*, *y=None*, ***fit_params*)

Learn and apply the dimension reduction on the train data.

ParametersX : array-like of predictors, shape = [n_samples, p]

Training vectors, where n_samples in the number of samples and p is the number of predictors.

Y : array-like of response, shape = [n_samples, q], optional

Training vectors, where n_samples in the number of samples and q is the number of response variables.

Returnsx_scores if Y is not given, (x_scores, y_scores) otherwise. :

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static transform(*X*, *Y=None*)
Apply the dimension reduction learned on the train data.

42.29 sklearn.pipeline: Pipeline

The sklearn.pipeline module implements utilities to build a composite estimator, as a chain of transforms and estimators.

pipeline.Pipeline(steps)	Pipeline of transforms with a final estimator.
pipeline.FeatureUnion(transformer_list[, ...])	Concatenates results of multiple transformer objects.

42.29.1 sklearn.pipeline.Pipeline

class sklearn.pipeline.**Pipeline**(*steps*)
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 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.

Parameterssteps: 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.

Examples

```
>>> from sklearn import svm
>>> from sklearn.datasets import samples_generator
>>> 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 = samples_generator.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
>>> anova_svm.set_params(anova__k=10, svc__C=.1).fit(X, y)
...
Pipeline(steps=[...])
>>> prediction = anova_svm.predict(X)
>>> anova_svm.score(X, y)
0.77...
```

Methods

decision_function	Applies transforms to the data, and the decision_function method of the final estimator.
fit(X[, y])	Fit all the transforms one after the other and transform the data, then fit the transformed data using the fi
fit_transform(X[, y])	Fit all the transforms one after the other and transform the data, then use fit_transform on transformed d
get_params([deep])	
inverse_transform	Applies inverse transform to the data.
predict	Applies transforms to the data, and the predict method of the final estimator.
predict_log_proba	Applies transforms to the data, and the predict_log_proba method of the final estimator.
predict_proba	Applies transforms to the data, and the predict_proba method of the final estimator.
score	Applies transforms to the data, and the score method of the final estimator.
set_params(**params)	Set the parameters of this estimator.
transform	Applies transforms to the data, and the transform method of the final estimator.

static `__init__`(*steps*)

`decision_function`

Applies transforms to the data, and the decision_function method of the final estimator. Valid only if the final estimator implements decision_function.

ParametersX : iterable

Data to predict on. Must fulfill input requirements of first step of the pipeline.

static `fit`(*X*, *y=None*, ***fit_params*)

Fit all the transforms one after the other and transform the data, then fit the transformed data using the final estimator.

ParametersX : 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.

static `fit_transform`(*X*, *y=None*, ***fit_params*)

Fit all the transforms one after the other and transform the data, then use fit_transform on transformed data using the final estimator.

ParametersX : 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.

`inverse_transform`

Applies inverse transform to the data. Starts with the last step of the pipeline and applies `inverse_transform` in inverse order of the pipeline steps. Valid only if all steps of the pipeline implement inverse_transform.

ParametersX : iterable

Data to inverse transform. Must fulfill output requirements of the last step of the pipeline.

`predict`

Applies transforms to the data, and the predict method of the final estimator. Valid only if the final estimator implements predict.

ParametersX : iterable

Data to predict on. Must fulfill input requirements of first step of the pipeline.

predict_log_proba

Applies transforms to the data, and the predict_log_proba method of the final estimator. Valid only if the final estimator implements predict_log_proba.

ParametersX : iterable

Data to predict on. Must fulfill input requirements of first step of the pipeline.

predict_proba

Applies transforms to the data, and the predict_proba method of the final estimator. Valid only if the final estimator implements predict_proba.

ParametersX : iterable

Data to predict on. Must fulfill input requirements of first step of the pipeline.

score

Applies transforms to the data, and the score method of the final estimator. Valid only if the final estimator implements score.

ParametersX : iterable

Data to score. 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.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform

Applies transforms to the data, and the transform method of the final estimator. Valid only if the final estimator implements transform.

ParametersX : iterable

Data to predict on. Must fulfill input requirements of first step of the pipeline.

Examples using `sklearn.pipeline.Pipeline`

42.29.2 `sklearn.pipeline`.FeatureUnion

class sklearn.pipeline.**FeatureUnion**(*transformer_list*, *n_jobs=1*, *transformer_weights=None*)

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.

Parameterstransformer_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.

Figure 42.557: *Concatenating multiple feature extraction methods*

Figure 42.558: *Imputing missing values before building an estimator*

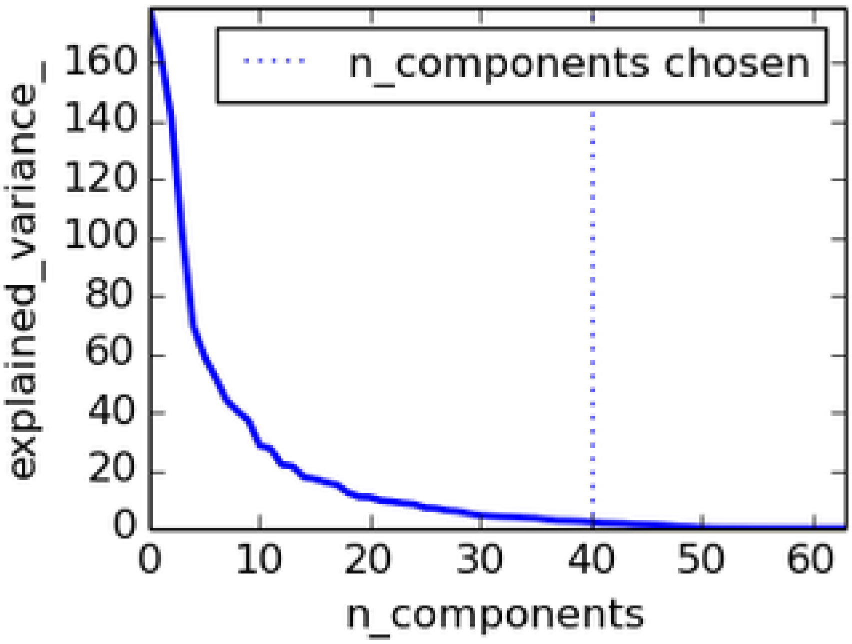


Figure 42.559: *Pipelining: chaining a PCA and a logistic regression*

Figure 42.560: *Feature Union with Heterogeneous Data Sources*

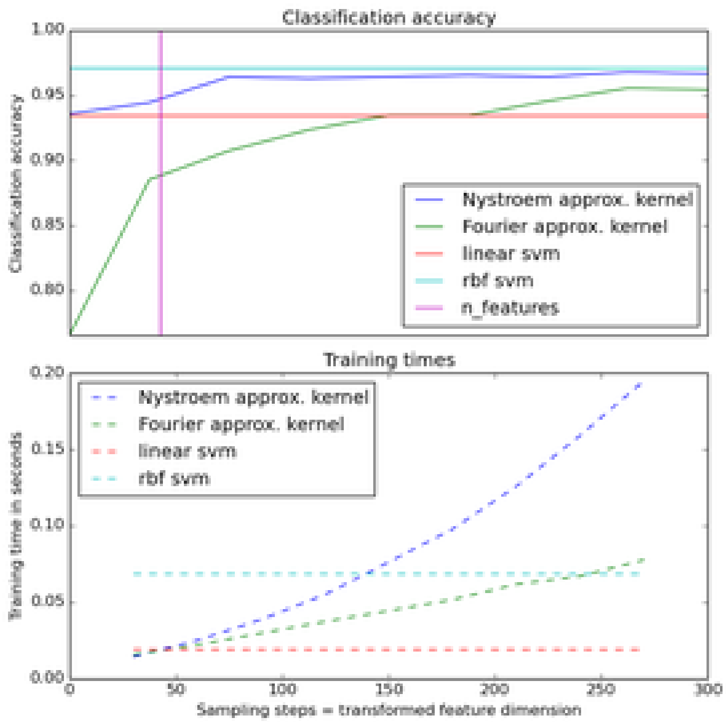


Figure 42.561: *Explicit feature map approximation for RBF kernels*

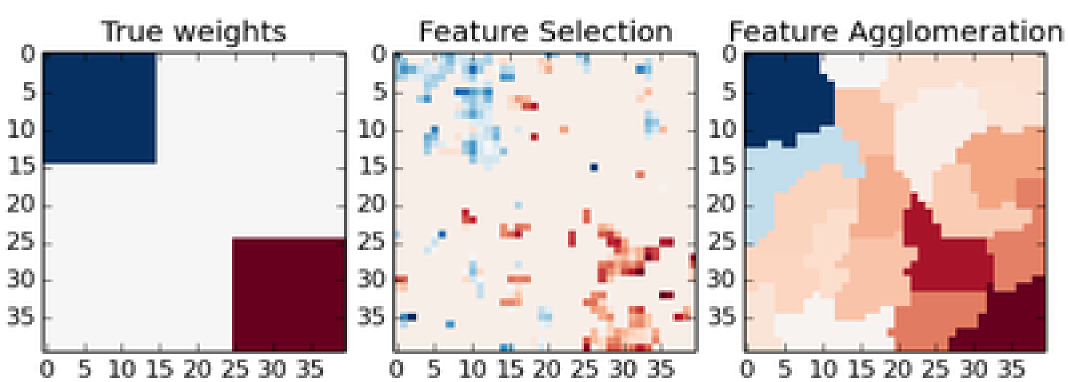


Figure 42.562: *Feature agglomeration vs. univariate selection*

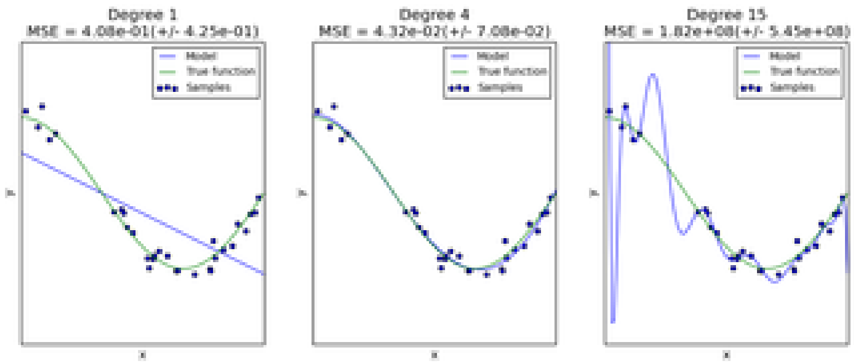


Figure 42.563: *Underfitting vs. Overfitting*

Figure 42.564: *Sample pipeline for text feature extraction and evaluation*

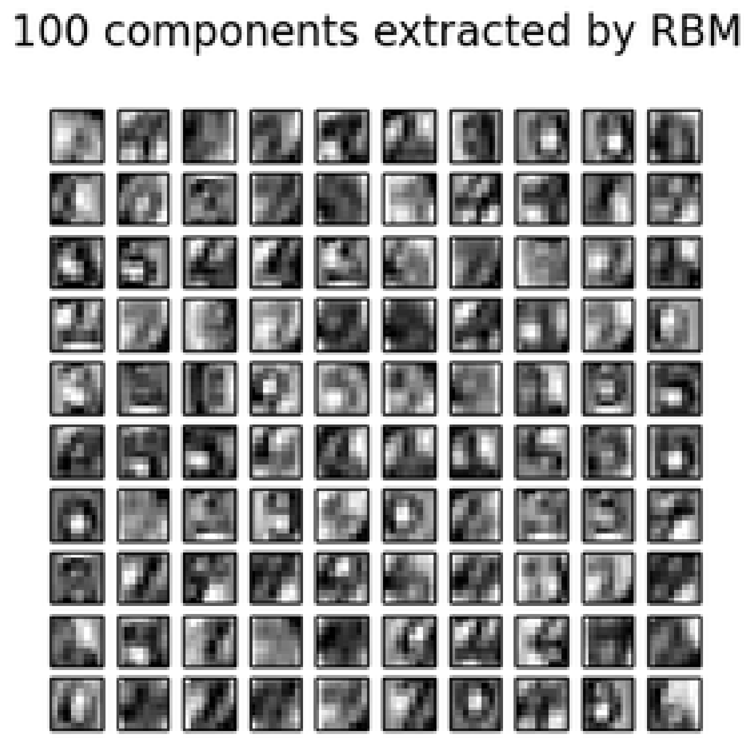


Figure 42.565: *Restricted Boltzmann Machine features for digit classification*

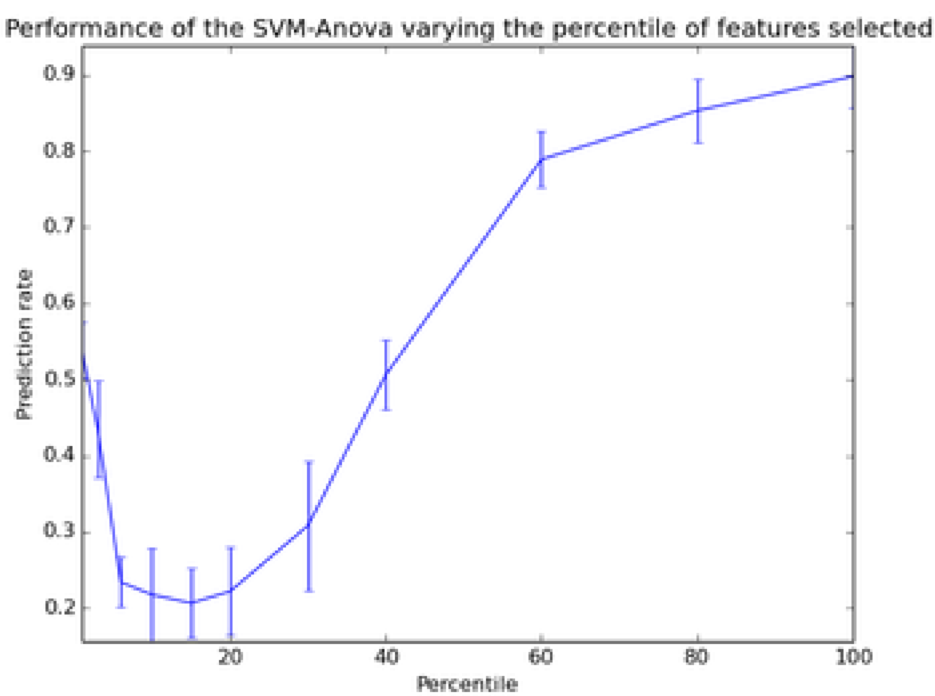


Figure 42.566: *SVM-Anova: SVM with univariate feature selection*

Figure 42.567: *Classification of text documents using sparse features*

n_jobs: int, optional :

Number of jobs to run in parallel (default 1).

transformer_weights: dict, optional :

Multiplicative weights for features per transformer. Keys are transformer names, values the weights.

Methods

`fit`(X[, y])	Fit all transformers using X.
`fit_transform`(X[, y])	Fit all transformers using X, transform the data and concatenate results.
`get_feature_names`()	Get feature names from all transformers.
`get_params`([deep])	
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Transform X separately by each transformer, concatenate results.

static **`__init__`** (*transformer_list*, *n_jobs=1*, *transformer_weights=None*)

static **`fit`** (*X*, *y=None*)

Fit all transformers using X.

ParametersX : array-like or sparse matrix, shape (n_samples, n_features)

Input data, used to fit transformers.

static **`fit_transform`** (*X*, *y=None*, ***fit_params*)

Fit all transformers using X, transform the data and concatenate results.

ParametersX : array-like or sparse matrix, shape (n_samples, n_features)

Input data to be transformed.

ReturnsX_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.

static **`get_feature_names`** ()

Get feature names from all transformers.

Returnsfeature_names : list of strings

Names of the features produced by transform.

`set_params` (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component

ReturnsX_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`

Figure 42.568: *Concatenating multiple feature extraction methods*

Figure 42.569: *Feature Union with Heterogeneous Data Sources*

`pipeline.make_pipeline`(*steps)	Construct a Pipeline from the given estimators.
`pipeline.make_union`(*transformers)	Construct a FeatureUnion from the given transformers.

42.29.3 `sklearn.pipeline`.make_pipeline

`sklearn.pipeline.`**`make_pipeline`**(**steps*)

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, they will be given names automatically based on their types.

Returnsp : Pipeline

Examples

```
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.preprocessing import StandardScaler
>>> make_pipeline(StandardScaler(), GaussianNB())
Pipeline(steps=[('standardscaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('gaussiannb', GaussianNB())])
```

Examples using `sklearn.pipeline.make_pipeline`

Figure 42.570: *Pipeline Anova SVM*

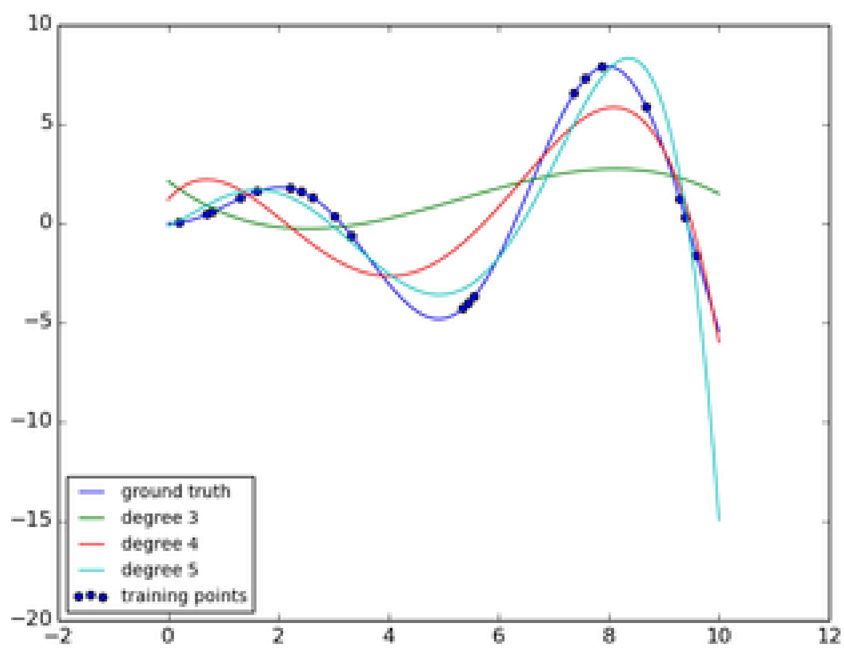


Figure 42.571: *Polynomial interpolation*

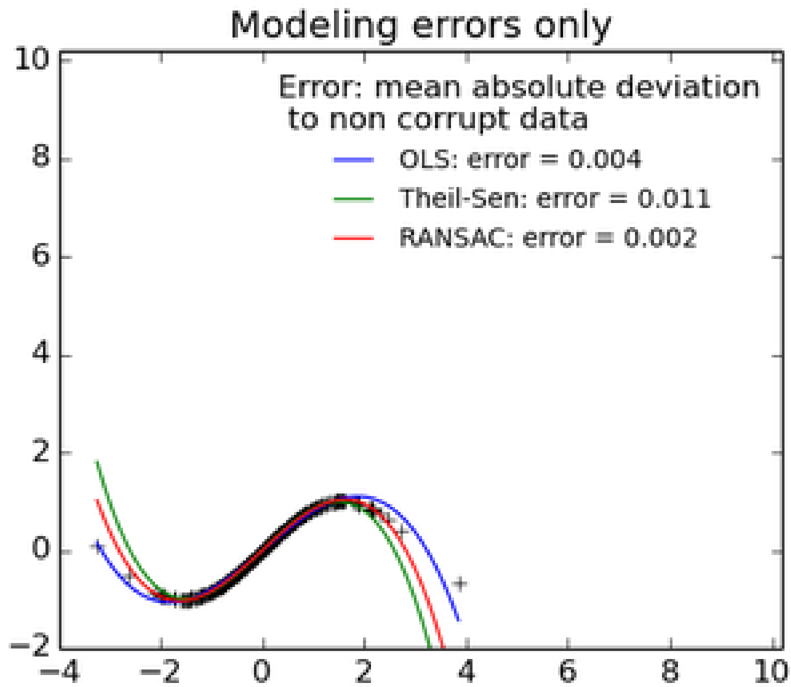


Figure 42.572: *Robust linear estimator fitting*

Figure 42.573: *Clustering text documents using k-means*

42.29.4 `sklearn.pipeline`.make_union

`sklearn.pipeline.`**`make_union`**`(`**transformers*`)`

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.

Returnsf : FeatureUnion

Examples

```
>>> from sklearn.decomposition import PCA, TruncatedSVD
>>> make_union(PCA(), TruncatedSVD())
FeatureUnion(n_jobs=1,
             transformer_list=[('pca', PCA(copy=True, n_components=None,
                                           whiten=False)),
                               ('truncatedsvd',
                                TruncatedSVD(algorithm='randomized',
                                             n_components=2, n_iter=5,
                                             random_state=None, tol=0.0))],
             transformer_weights=None)
```

42.30 `sklearn.preprocessing`: Preprocessing and Normalization

The `sklearn.preprocessing` module includes scaling, centering, normalization, binarization and imputation 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.Imputer`([missing_values, ...])	Imputation transformer for completing missing values.
`preprocessing.KernelCenterer`	Center a kernel matrix
`preprocessing.LabelBinarizer`([neg_label, ...])	Binarize labels in a one-vs-all fashion
`preprocessing.LabelEncoder`	Encode labels with value between 0 and n_classes-1.
`preprocessing.MultiLabelBinarizer`([classes, ...])	Transform between iterable of iterables and a multilabel format
`preprocessing.MinMaxScaler`([feature_range, copy])	Standardizes features by scaling each feature to a given range.
`preprocessing.Normalizer`([norm, copy])	Normalize samples individually to unit norm.
`preprocessing.OneHotEncoder`([n_values, ...])	Encode categorical integer features using a one-hot aka one-of-K sc
`preprocessing.StandardScaler`([copy, ...])	Standardize features by removing the mean and scaling to unit varia
`preprocessing.PolynomialFeatures`([degree, ...])	Generate polynomial and interaction features.

42.30.1 `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).

Parametersthreshold : 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).

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.

Methods

`fit`(X[, y])	Do nothing and return the estimator unchanged
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y, copy])	Binarize each element of X

static `__init__`(*threshold=0.0*, *copy=True*)

static `fit`(*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.

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **transform**(*X*, *y=None*, *copy=None*)

Binarize each element of X

ParametersX : array or scipy.sparse matrix with 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.

42.30.2 sklearn.preprocessing.Imputer

class sklearn.preprocessing.**Imputer**(*missing_values='NaN'*, *strategy='mean'*, *axis=0*, *verbose=0*, *copy=True*)

Imputation transformer for completing missing values.

Parametersmissing_values : integer or "NaN", optional (default="NaN")

The placeholder for the missing values. All occurrences of *missing_values* will be imputed. For missing values encoded as np.nan, use the string value "NaN".

strategy : string, optional (default="mean")

The imputation strategy.

- If "mean", then replace missing values using the mean along the axis.
- If "median", then replace missing values using the median along the axis.
- If "most_frequent", then replace missing using the most frequent value along the axis.

axis : integer, optional (default=0)

The axis along which to impute.

- If *axis=0*, then impute along columns.
- If *axis=1*, then impute along rows.

verbose : integer, optional (default=0)

Controls the verbosity of the imputer.

copy : boolean, optional (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 sparse and *missing_values=0*;
- If *axis=0* and X is encoded as a CSR matrix;

•If *axis=1* and X is encoded as a CSC matrix.

Attributesstatistics_ : array of shape (n_features,)

The imputation fill value for each feature if axis == 0.

Notes

•When `axis=0`, columns which only contained missing values at *fit* are discarded upon *transform*.

•When `axis=1`, an exception is raised if there are rows for which it is not possible to fill in the missing values (e.g., because they only contain missing values).

Methods

`fit`(X[, y])	Fit the imputer on X.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Impute all missing values in X.

static `__init__` (*missing_values='NaN'*, *strategy='mean'*, *axis=0*, *verbose=0*, *copy=True*)

static `fit` (*X*, *y=None*)

Fit the imputer on X.

ParametersX : {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.

Returnsself : object

Returns self.

`fit_transform` (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

`get_params` (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **transform** (*X*)

Impute all missing values in X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

The input data to complete.

Examples using sklearn.preprocessing.Imputer

Figure 42.574: *Imputing missing values before building an estimator*

42.30.3 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).

Methods

fit(K[, y])	Fit KernelCenterer
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.
transform(K[, y, copy])	Center kernel matrix.

__init__ ()

static fit (*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 (*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.

Returns X_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*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 (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returns self :

static transform (*K*, *y=None*, *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]

42.30.4 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 the scikit. 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.

Parametersneg_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.

Attributesclasses_ : 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', 'mutliclass-multioutput', 'multilabel-sequences', 'multilabel-indicator', and 'unknown'.

multilabel_ : boolean

True if the transformer was fitted on a multilabel rather than a multiclass set of labels. The `multilabel_` attribute is deprecated and will be removed in 0.18

sparse_input_ : boolean,

True if the input data to transform is given as a sparse matrix, False otherwise.

indicator_matrix_ : str

'sparse' when the input data to tansform is a multilable-indicator and is sparse, None otherwise. The `indicator_matrix_` attribute is deprecated as of version 0.16 and will be removed in 0.18

See also:

`label_binarize`function to perform the transform operation of LabelBinarizer with fixed classes.

Examples

```
>>> from sklearn import preprocessing
>>> lb = preprocessing.LabelBinarizer()
>>> lb.fit([1, 2, 6, 4, 2])
LabelBinarizer(neg_label=0, pos_label=1, sparse_output=False)
>>> 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(neg_label=0, pos_label=1, sparse_output=False)
>>> 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`(y)	Fit label binarizer
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(Y[, threshold])	Transform binary labels back to multi-class labels
`set_params`(**params)	Set the parameters of this estimator.
`transform`(y)	Transform multi-class labels to binary labels

static `__init__`(*neg_label=0, pos_label=1, sparse_output=False*)

static `fit`(*y*)

Fit label binarizer

Parametersy : numpy 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.

Returnsself : returns an instance of self.

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **inverse_transform** (*Y*, *threshold=None*)
Transform binary labels back to multi-class labels

ParametersY : 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.

Returnsy : 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 (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **transform** (*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.

Parametersy : numpy 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.

ReturnsY : numpy array or CSR matrix of shape [n_samples, n_classes]

Shape will be [n_samples, 1] for binary problems.

42.30.5 sklearn.preprocessing.LabelEncoder

class sklearn.preprocessing.**LabelEncoder**

Encode labels with value between 0 and n_classes-1.

Attributesclasses_ : array of shape (n_class,)

Holds the label for each class.

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(y)	Fit label encoder
fit_transform(y)	Fit label encoder and return encoded labels
get_params([deep])	Get parameters for this estimator.
inverse_transform(y)	Transform labels back to original encoding.
set_params(**params)	Set the parameters of this estimator.
transform(y)	Transform labels to normalized encoding.

__init__()

static fit (*y*)

Fit label encoder

Parametersy : array-like of shape (n_samples,)

Target values.

Returnsself : returns an instance of self.

static **`fit_transform`**(*y*)

Fit label encoder and return encoded labels

Parametersy : array-like of shape [n_samples]

Target values.

Returnsy : array-like of shape [n_samples]

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **`inverse_transform`**(*y*)

Transform labels back to original encoding.

Parametersy : numpy array of shape [n_samples]

Target values.

Returnsy : numpy array of shape [n_samples]

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static **`transform`**(*y*)

Transform labels to normalized encoding.

Parametersy : array-like of shape [n_samples]

Target values.

Returnsy : array-like of shape [n_samples]

42.30.6 `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.

Parametersclasses : array-like of shape [n_classes] (optional)

Indicates an ordering for the class labels

sparse_output : boolean (default: False),

Set to true if output binary array is desired in CSR sparse format

Attributesclasses_ : array of labels

A copy of the *classes* parameter where provided, or otherwise, the sorted set of classes found when fitting.

Examples

```
>>> mlb = MultiLabelBinarizer()
>>> mlb.fit_transform([(1, 2), (3,)])
array([[1, 1, 0],
       [0, 0, 1]])
>>> mlb.classes_
array([1, 2, 3])

>>> mlb.fit_transform([set(['sci-fi', 'thriller']), set(['comedy'])])
array([[0, 1, 1],
       [1, 0, 0]])
>>> list(mlb.classes_)
['comedy', 'sci-fi', 'thriller']
```

Methods

`fit`(y)	Fit the label sets binarizer, storing *classes_*
`fit_transform`(y)	Fit the label sets binarizer and transform the given label sets
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(yt)	Transform the given indicator matrix into label sets
`set_params`(**params)	Set the parameters of this estimator.
`transform`(y)	Transform the given label sets

static `__init__`(*classes=None*, *sparse_output=False*)

static `fit`(*y*)

Fit the label sets binarizer, storing *classes_*

Parametersy : 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.

Returnsself : returns this MultiLabelBinarizer instance

static `fit_transform`(*y*)

Fit the label sets binarizer and transform the given label sets

Parametersy : 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.

Returnsy_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`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **inverse_transform**(*yt*)
Transform the given indicator matrix into label sets

Parametersyt : array or sparse matrix of shape (n_samples, n_classes)

A matrix containing only 1s ands 0s.

Returnsy : 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(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static **transform**(*y*)
Transform the given label sets

Parametersy : 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.

Returnsy_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.

42.30.7 sklearn.preprocessing.MinMaxScaler

class sklearn.preprocessing.**MinMaxScaler**(*feature_range=(0, 1)*, *copy=True*)
Standardizes 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 standardization 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.

This standardization is often used as an alternative to zero mean, unit variance scaling.

Parametersfeature_range: tuple (min, max), default=(0, 1) :

Desired range of transformed data.

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).

Attributesmin_ : ndarray, shape (n_features,)

Per feature adjustment for minimum.

scale_ : ndarray, shape (n_features,)

Per feature relative scaling of the data.

Methods

`fit`(X[, y])	Compute the minimum and maximum to be used for later scaling.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(X)	Undo the scaling of X according to feature_range.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Scaling features of X according to feature_range.

static `__init__` (*feature_range=(0, 1)*, *copy=True*)

static `fit` (*X*, *y=None*)

Compute the minimum and maximum to be used for later scaling.

ParametersX : array-like, 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` (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

`get_params` (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static `inverse_transform` (*X*)

Undo the scaling of X according to feature_range.

ParametersX : array-like with shape [n_samples, n_features]

Input data that will be transformed.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **transform** (*X*)

Scaling features of X according to feature_range.

ParametersX : array-like with shape [n_samples, n_features]

Input data that will be transformed.

42.30.8 `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.

Parametersnorm : 'l1' or 'l2', 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:

`sklearn.preprocessing.normalize`, `without`

Notes

This estimator is stateless (besides constructor parameters), the fit method does nothing but is useful when used in a pipeline.

Methods

`fit`(X[, y])	Do nothing and return the estimator unchanged
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y, copy])	Scale each non zero row of X to unit norm

static `__init__` (*norm='l2'*, *copy=True*)

static `fit` (*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.

`fit_transform` (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

`get_params` (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

`set_params` (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

static `transform` (*X*, *y=None*, *copy=None*)

Scale each non zero row of X to unit norm

ParametersX : array or scipy.sparse matrix with 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.

Examples using `sklearn.preprocessing.Normalizer`

42.30.9 `sklearn.preprocessing.OneHotEncoder`

class `sklearn.preprocessing.OneHotEncoder` (*n_values='auto'*, *categorical_features='all'*, *dtype=<type 'float'>*, *sparse=True*, *handle_unknown='error'*)

Encode categorical integer features using a one-hot aka one-of-K scheme.

Figure 42.575: *Clustering text documents using k-means*

The input to this transformer should be a matrix of integers, denoting the values taken on by categorical (discrete) features. The output will be a sparse matrix where each column corresponds to one possible value of one feature. It is assumed that input features take on values in the range [0, n_values).

This encoding is needed for feeding categorical data to many scikit-learn estimators, notably linear models and SVMs with the standard kernels.

Parametersn_values : 'auto', int or array of ints

Number of values per feature.

- 'auto' : determine value range from training data.
- int : maximum value for all features.
- array : maximum value per feature.

categorical_features: "all" or array of indices or mask :

Specify what features are treated as categorical.

- 'all' (default): All features are treated as categorical.
- array of indices: Array of categorical feature indices.
- mask: Array of length n_features and with dtype=bool.

Non-categorical features are always stacked to the right of the matrix.

dtype : number type, default=np.float

Desired dtype of output.

sparse : boolean, default=True

Will return sparse matrix if set True else will return an array.

handle_unknown : str, 'error' or 'ignore'

Whether to raise an error or ignore if a unknown categorical feature is present during transform.

Attributesactive_features_ : array

Indices for active features, meaning values that actually occur in the training set. Only available when n_values is `'auto'`.

feature_indices_ : array of shape (n_features,) Indices to feature ranges. Feature `i` in the original data is mapped to features from `feature_indices_[i]` to `feature_indices_[i+1]` (and then potentially masked by *active_features_* afterwards)

n_values_ : array of shape (n_features,)

Maximum number of values per feature.

See also:

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.

Examples

Given a dataset with three features and two samples, we let the encoder find the maximum value per feature and transform the data to a binary one-hot encoding.

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> enc = OneHotEncoder()
>>> enc.fit([[0, 0, 3], [1, 1, 0], [0, 2, 1], [1, 0, 2]])
OneHotEncoder(categorical_features='all', dtype=<... 'float'>,
       handle_unknown='error', n_values='auto', sparse=True)
>>> enc.n_values_
array([2, 3, 4])
>>> enc.feature_indices_
array([0, 2, 5, 9])
>>> enc.transform([[0, 1, 1]]).toarray()
array([[ 1.,  0.,  0.,  1.,  0.,  0.,  1.,  0.,  0.]])
```

Methods

`fit`(X[, y])	Fit OneHotEncoder to X.
`fit_transform`(X[, y])	Fit OneHotEncoder to X, then transform X.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Transform X using one-hot encoding.

static `__init__`(*n_values='auto'*, *categorical_features='all'*, *dtype=<type 'float'>*, *sparse=True*, *handle_unknown='error'*)

static `fit`(*X*, *y=None*)

Fit OneHotEncoder to X.

Parameters X : array-like, shape=(n_samples, n_feature)

Input array of type int.

Returns self :

static `fit_transform`(*X*, *y=None*)

Fit OneHotEncoder to X, then transform X.

Equivalent to self.fit(X).transform(X), but more convenient and more efficient. See fit for the parameters, transform for the return value.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static transform(*X*)
Transform X using one-hot encoding.

ParametersX : array-like, shape=(n_samples, n_features)

Input array of type int.

ReturnsX_out : sparse matrix if sparse=True else a 2-d array, dtype=int

Transformed input.

42.30.10 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

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 the *transform* method.

Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual feature 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.

Parameterswith_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).

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.

Attributesmean_ : array of floats with shape [n_features]

The mean value for each feature in the training set.

std_ : array of floats with shape [n_features]

The standard deviation for each feature in the training set.

See also:

`sklearn.preprocessing.scale`, `scaling`, `sklearn.decomposition.RandomizedPCA`, `to`

Methods

`fit`(X[, y])	Compute the mean and std to be used for later scaling.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(X[, copy])	Scale back the data to the original representation
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y, copy])	Perform standardization by centering and scaling

static `__init__`(*copy=True*, *with_mean=True*, *with_std=True*)

static `fit`(*X*, *y=None*)

Compute the mean and std to be used for later scaling.

ParametersX : array-like or CSR matrix with shape [n_samples, n_features]

The data used to compute the mean and standard deviation used for later scaling along the features axis.

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static **inverse_transform**(*X*, *copy=None*)

Scale back the data to the original representation

ParametersX : array-like with shape [n_samples, n_features]

The data used to scale along the features axis.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **transform**(*X*, *y=None*, *copy=None*)

Perform standardization by centering and scaling

ParametersX : array-like with shape [n_samples, n_features]

The data used to scale along the features axis.

Examples using `sklearn.preprocessing.StandardScaler`

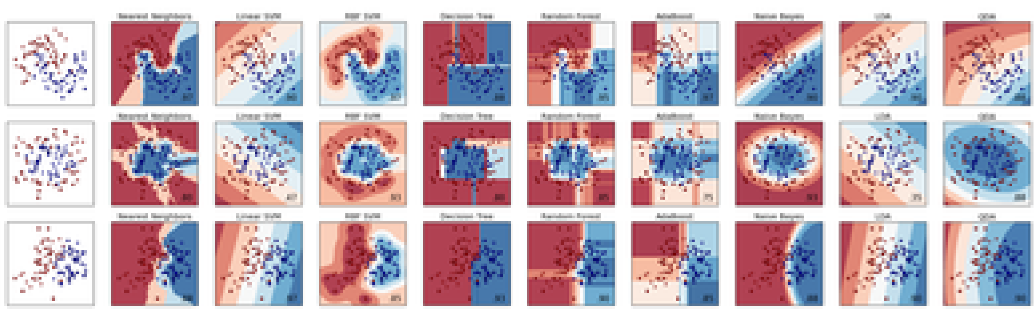


Figure 42.576: *Classifier comparison*

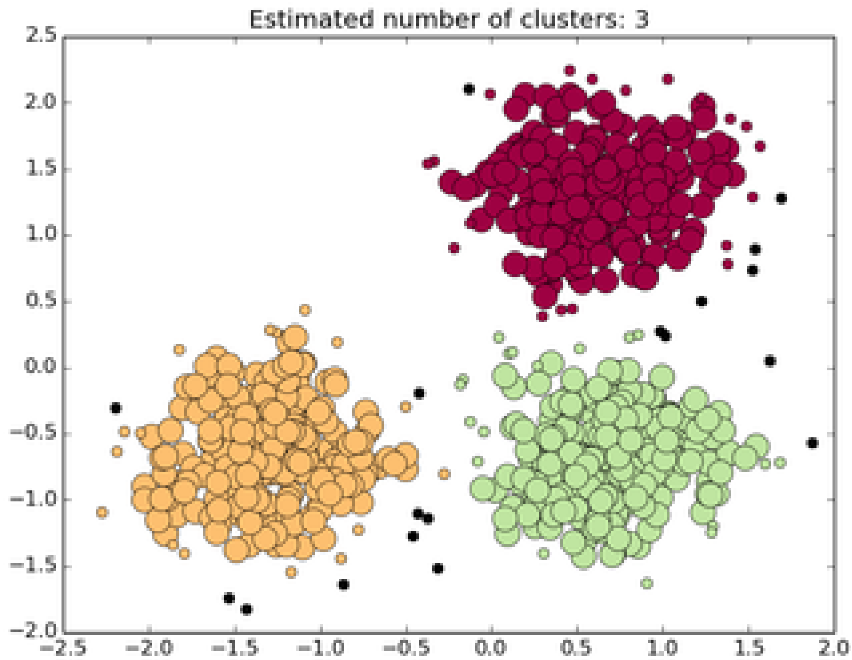


Figure 42.577: *Demo of DBSCAN clustering algorithm*

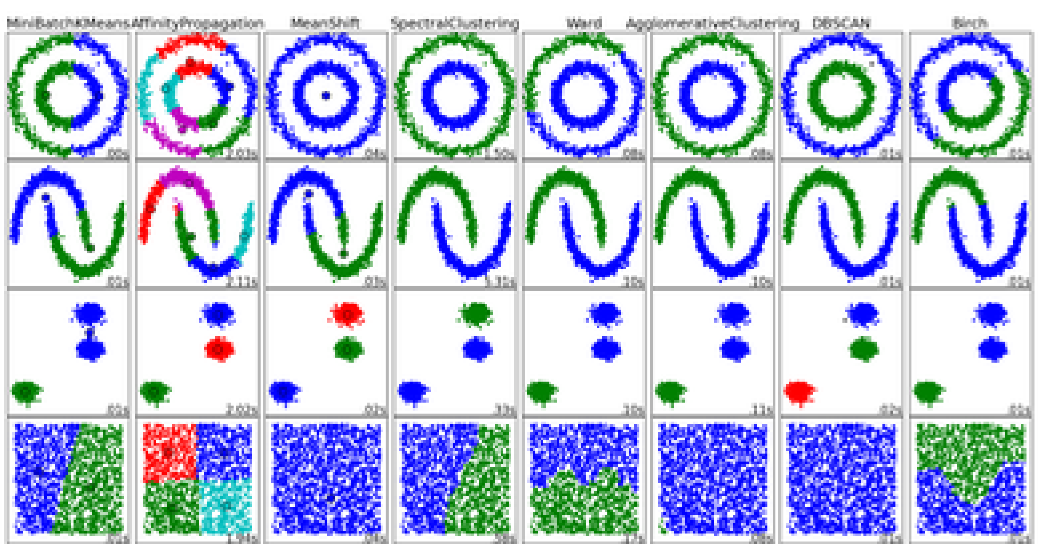


Figure 42.578: *Comparing different clustering algorithms on toy datasets*

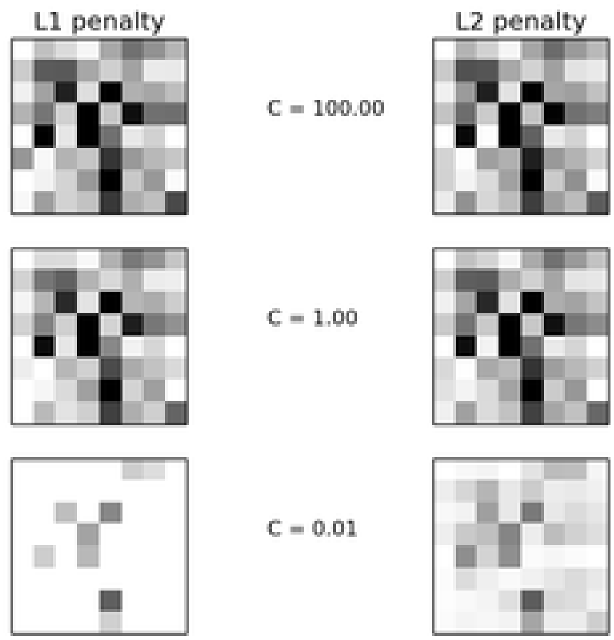


Figure 42.579: *L1 Penalty and Sparsity in Logistic Regression*

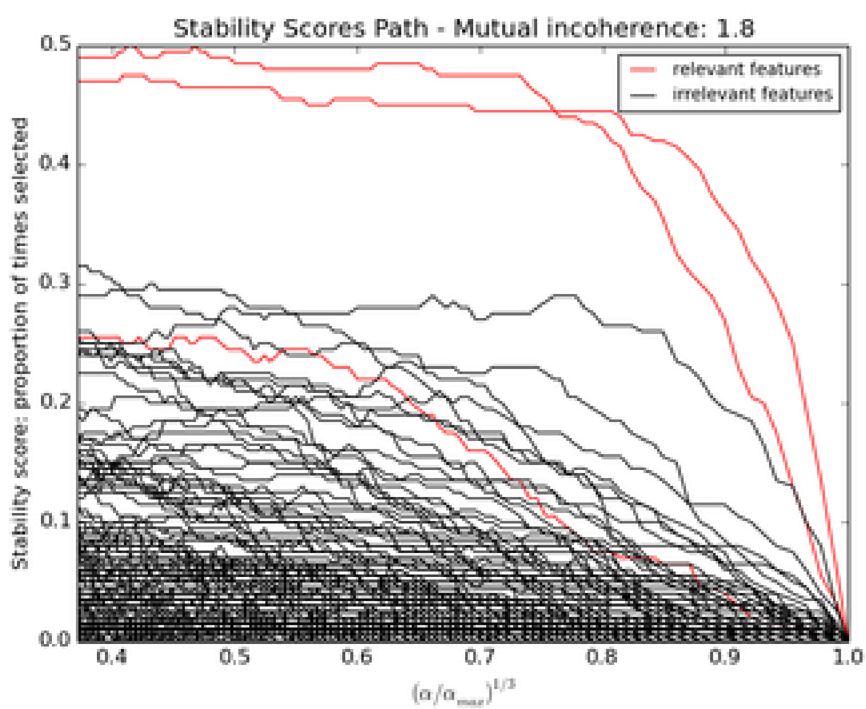


Figure 42.580: *Sparse recovery: feature selection for sparse linear models*

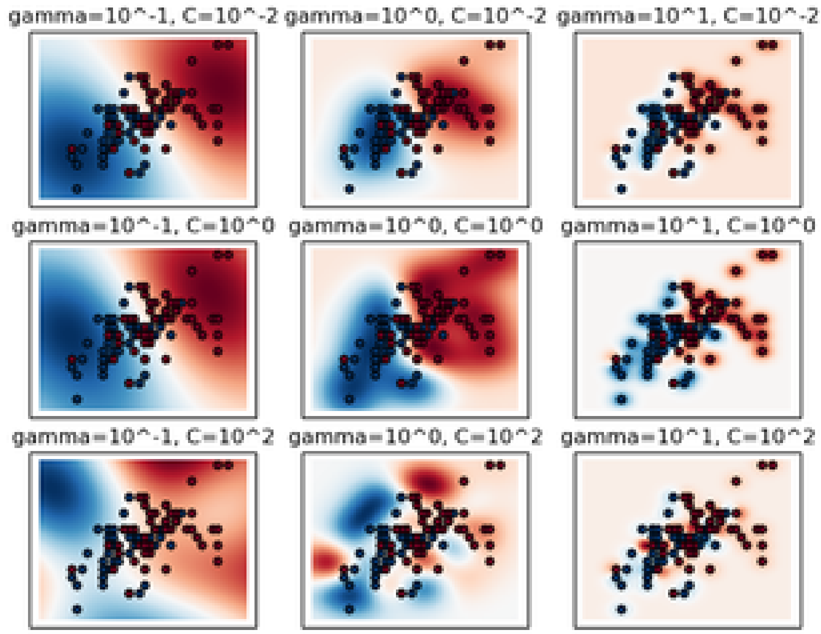


Figure 42.581: *RBF SVM parameters*

42.30.11 `sklearn.preprocessing`.PolynomialFeatures

class `sklearn.preprocessing.`**`PolynomialFeatures`**(*degree=2*, *interaction_only=False*, *include_bias=True*)

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].

Parametersdegree : 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).

Attributespowers_ : array, shape (n_input_features, n_output_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

```
>>> 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`(X[, y])	Compute number of output features.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y])	Transform data to polynomial features

static `__init__`(*degree=2*, *interaction_only=False*, *include_bias=True*)

static `fit`(*X*, *y=None*)

Compute number of output features.

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

static **transform** (*X*, *y=None*)

Transform data to polynomial features

ParametersX : array with shape [n_samples, n_features]

The data to transform, row by row.

ReturnsXP : np.ndarray 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`

`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.normalize`(X[, norm, axis, copy])	Scale input vectors individually to unit norm (vector length).
`preprocessing.scale`(X[, axis, with_mean, ...])	Standardize a dataset along any axis

42.30.12 `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.

ParametersX : array or scipy.sparse matrix with shape [n_samples, n_features]

Data.

value : float

Value to use for the dummy feature.

ReturnsX : array or scipy.sparse matrix with shape [n_samples, n_features + 1]

Same data with dummy feature added as first column.

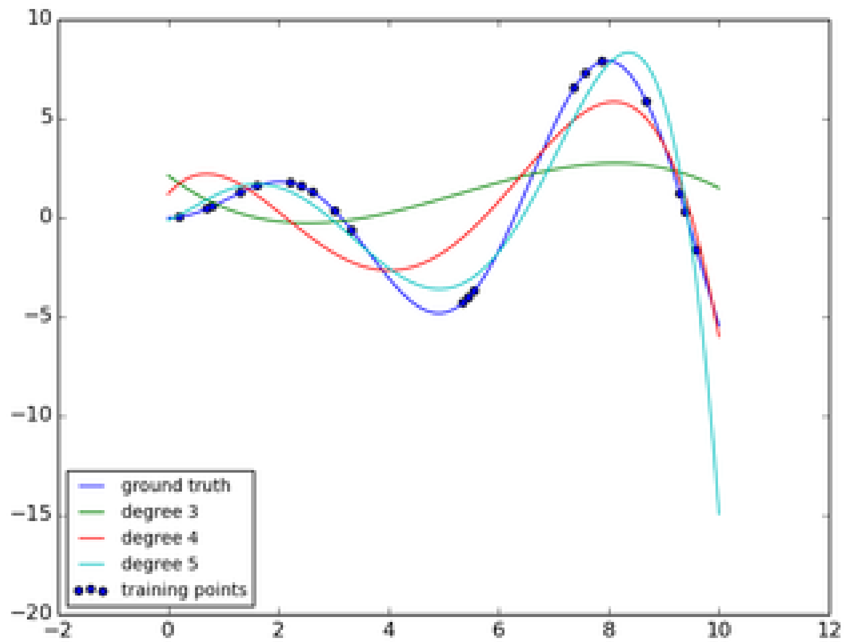


Figure 42.582: *Polynomial interpolation*

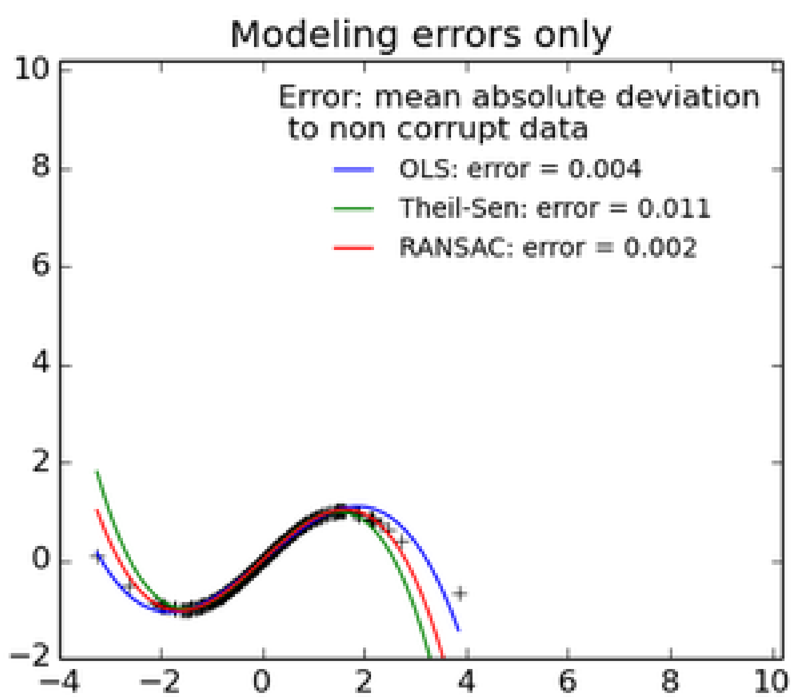


Figure 42.583: *Robust linear estimator fitting*

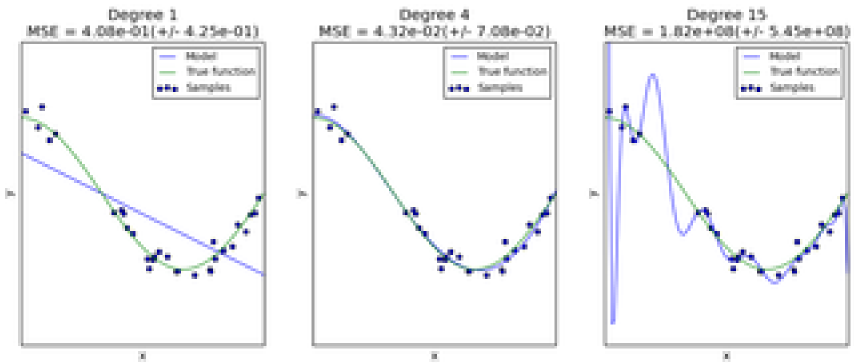


Figure 42.584: *Underfitting vs. Overfitting*

Examples

```
>>> from sklearn.preprocessing import add_dummy_feature
>>> add_dummy_feature([[0, 1], [1, 0]])
array([[ 1.,  0.,  1.],
       [ 1.,  1.,  0.]])
```

42.30.13 sklearn.preprocessing.binarize

`sklearn.preprocessing.binarize`(*X*, *threshold=0.0*, *copy=True*)

Boolean thresholding of array-like or scipy.sparse matrix

Parameters X : array or scipy.sparse matrix with 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:

`sklearn.preprocessing.Binarizer`, using, `sklearn.pipeline.Pipeline`

42.30.14 sklearn.preprocessing.label_binarize

sklearn.preprocessing.**label_binarize**(*y*, *classes*, *neg_label=0*, *pos_label=1*, *sparse_output=False*, *multilabel=None*)

Binarize labels in a one-vs-all fashion

Several regression and binary classification algorithms are available in the scikit. 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.

Parametersy : 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

ReturnsY : numpy array or CSR matrix of shape [n_samples, n_classes]

Shape will be [n_samples, 1] for binary problems.

See also:

LabelBinarizerclass 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`

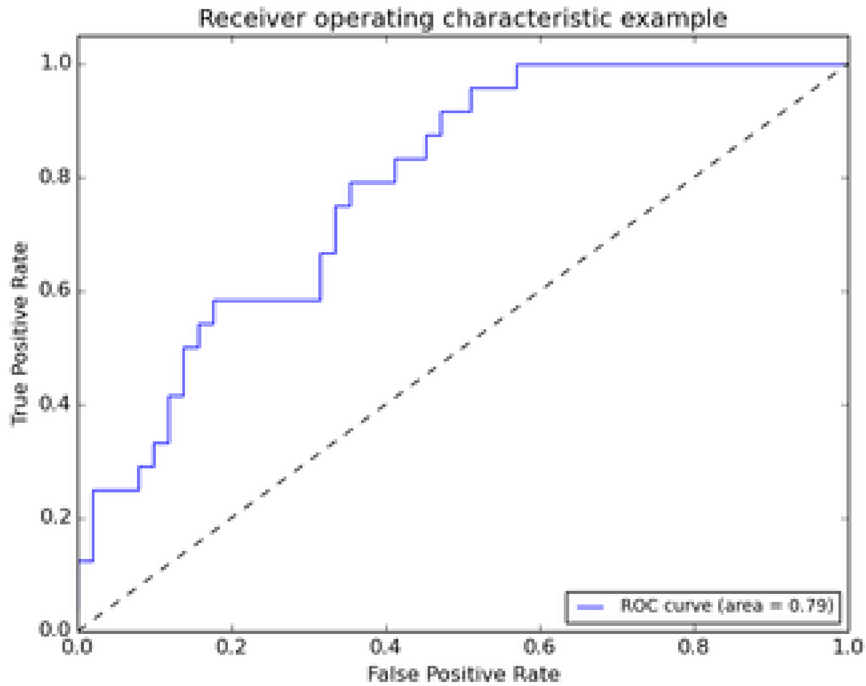


Figure 42.585: *Receiver Operating Characteristic (ROC)*

42.30.15 `sklearn.preprocessing`.normalize

`sklearn.preprocessing.`**`normalize`**(*X*, *norm='l2'*, *axis=1*, *copy=True*)

Scale input vectors individually to unit norm (vector length).

Parameters **X** : array or scipy.sparse matrix with 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' or 'l2', 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).

See also:

`sklearn.preprocessing.Normalizer`, using, `sklearn.pipeline.Pipeline`

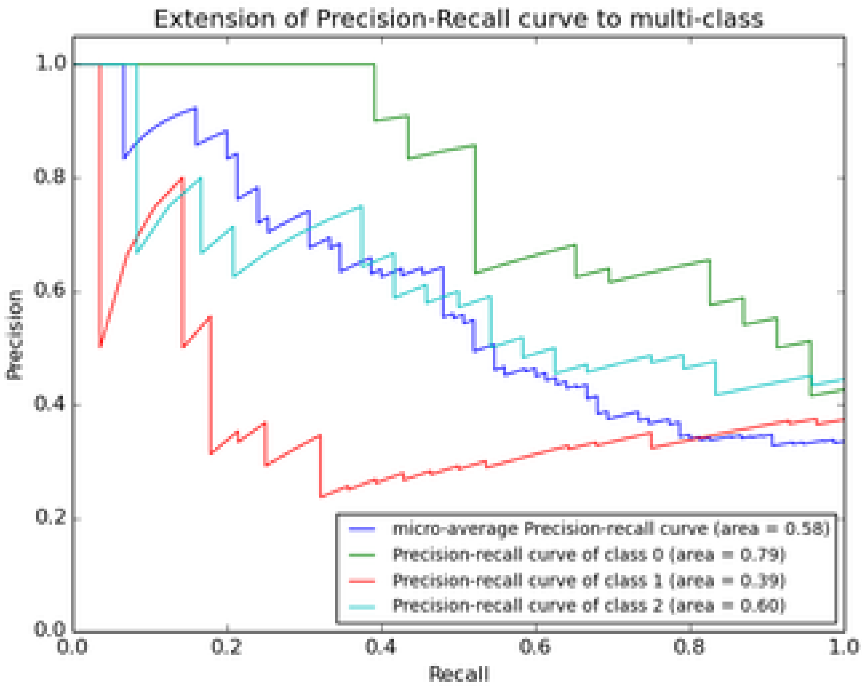


Figure 42.586: *Precision-Recall*

42.30.16 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.

ParametersX : array-like or CSR 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 CSR matrix and if axis is 1).

See also:

sklearn.preprocessing.StandardScaler, scaling, 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 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.

Examples using `sklearn.preprocessing.scale`

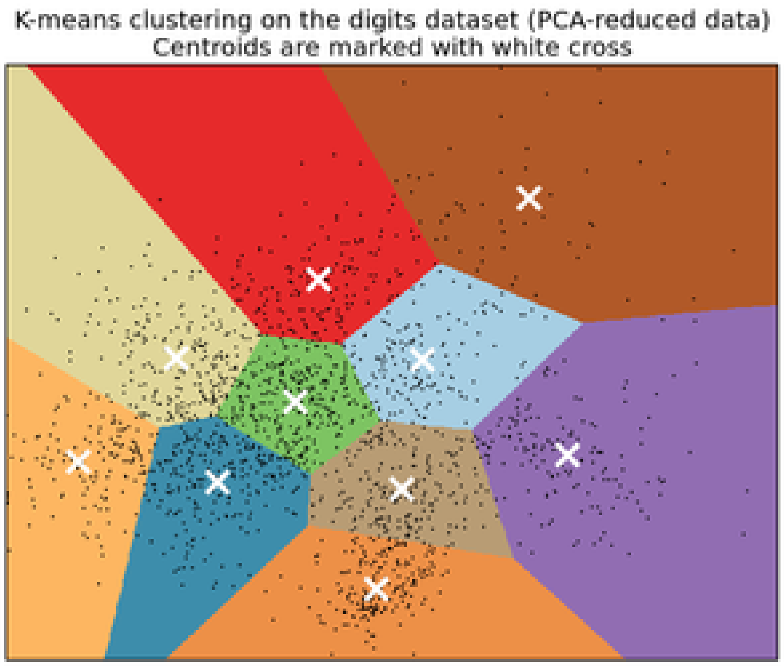


Figure 42.587: *A demo of K-Means clustering on the handwritten digits data*

42.31 `sklearn.qda`: Quadratic Discriminant Analysis

Quadratic Discriminant Analysis

User guide: See the *Linear and quadratic discriminant analysis* section for further details.

`qda.QDA`([priors, reg_param])	Quadratic Discriminant Analysis (QDA)

42.31.1 sklearn.qda.QDA

class `sklearn.qda.QDA`(*priors=None*, *reg_param=0.0*)

Quadratic Discriminant Analysis (QDA)

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.

Parameterspriors : 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)`

Attributescovariances_ : list of array-like, shape = [n_features, n_features]

Covariance matrices of each class.

means_ : array-like, shape = [n_classes, n_features]

Class means.

priors_ : array-like, 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.

See also:

`sklearn.lda.LDA`Linear discriminant analysis

Examples

```
>>> from sklearn.qda import QDA
>>> 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 = QDA()
>>> clf.fit(X, y)
QDA(priors=None, reg_param=0.0)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

`decision_function`(X)	Apply decision function to an array of samples.
`fit`(X, y[, store_covariances, tol])	Fit the QDA model according to the given training data and parameters.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Perform classification on an array of test vectors X.
`predict_log_proba`(X)	Return posterior probabilities of classification.
`predict_proba`(X)	Return posterior probabilities of classification.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*priors=None*, *reg_param=0.0*)

static `decision_function`(*X*)

Apply decision function to an array of samples.

ParametersX : array-like, shape = [n_samples, n_features]

Array of samples (test vectors).

ReturnsC : array, shape = [n_samples, n_classes] or [n_samples,]

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.

static `fit`(*X*, *y*, *store_covariances=False*, *tol=0.0001*)

Fit the QDA model according to the given training data and parameters.

ParametersX : 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, shape = [n_samples]

Target values (integers)

store_covariances : boolean

If True the covariance matrices are computed and stored in the *self.covariances_* attribute.

tol : float, optional, default 1.0e-4

Threshold used for rank estimation.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

static `predict`(*X*)

Perform classification on an array of test vectors X.

The predicted class C for each sample in X is returned.

ParametersX : array-like, shape = [n_samples, n_features]

ReturnsC : array, shape = [n_samples]

static **predict_log_proba**(*X*)
Return posterior probabilities of classification.

ParametersX : array-like, shape = [n_samples, n_features]

Array of samples/test vectors.

ReturnsC : array, shape = [n_samples, n_classes]

Posterior log-probabilities of classification per class.

static **predict_proba**(*X*)
Return posterior probabilities of classification.

ParametersX : array-like, shape = [n_samples, n_features]

Array of samples/test vectors.

ReturnsC : array, shape = [n_samples, n_classes]

Posterior probabilities of classification per class.

score(*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.qda.QDA`

42.32 `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.

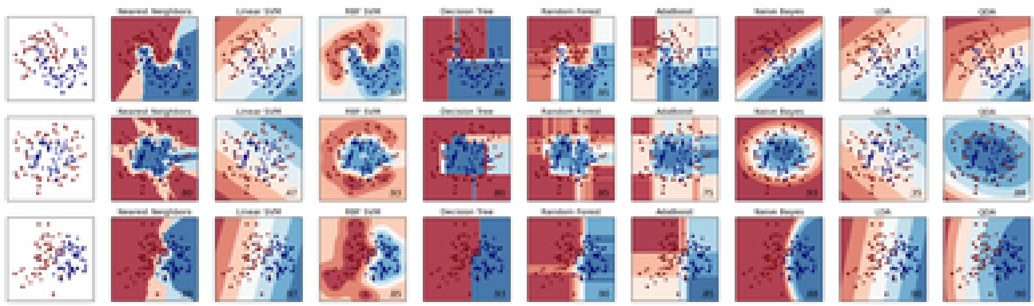


Figure 42.588: *Classifier comparison*

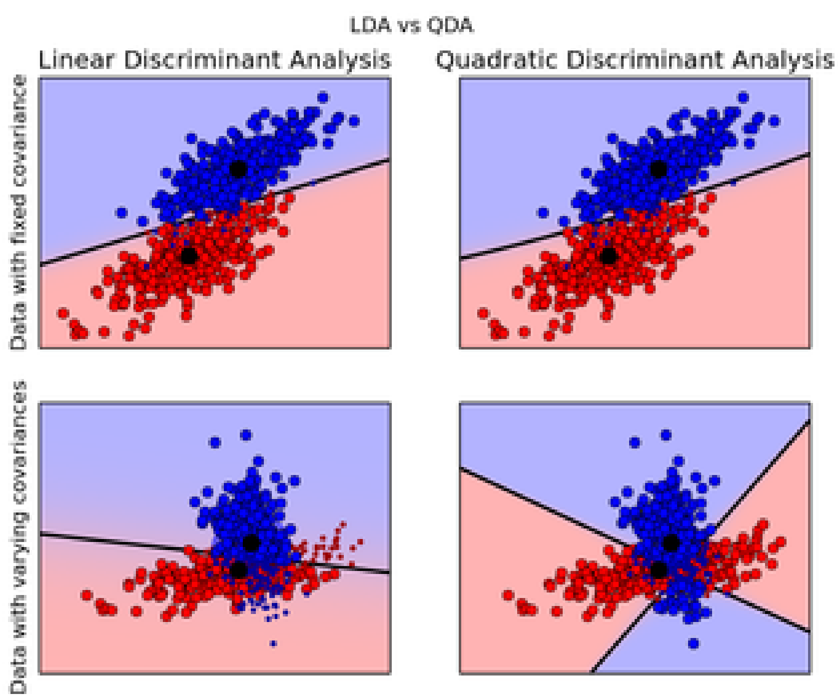


Figure 42.589: *Linear and Quadratic Discriminant Analysis with confidence ellipsoid*

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

42.32.1 `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).

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 : integer, RandomState instance or None (default=None)

Control the pseudo random number generator used to generate the matrix at fit time.

Attributes **n_component_** : 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`

Methods

fit(X[, y])	Generate a sparse random projection matrix
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Project the data by using matrix product with the random matrix

static __init__(*n_components='auto'*, *eps=0.1*, *random_state=None*)

fit(*X*, *y=None*)

Generate a sparse random projection matrix

ParametersX : 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 : is not used: placeholder to allow for usage in a Pipeline.

Returnsself :

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*, *y=None*)

Project the data by using matrix product with the random matrix

ParametersX : numpy array or scipy.sparse of shape [n_samples, n_features]

The input data to project into a smaller dimensional space.

y : is not used: placeholder to allow for usage in a Pipeline.

ReturnsX_new : numpy array or scipy sparse of shape [n_samples, n_components]

Projected array.

42.32.2 `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

Parametersn_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 : integer, RandomState instance or None (default=None)

Control the pseudo random number generator used to generate the matrix at fit time.

Attributesn_component_ : 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

[R188], [R189]

Methods

`fit`(X[, y])	Generate a sparse random projection matrix
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, y])	Project the data by using matrix product with the random matrix

static __init__(*n_components='auto'*, *density='auto'*, *eps=0.1*, *dense_output=False*, *random_state=None*)

fit(*X*, *y=None*)

Generate a sparse random projection matrix

ParametersX : 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 : is not used: placeholder to allow for usage in a Pipeline.

Returnsself :

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*, *y=None*)
Project the data by using matrix product with the random matrix

ParametersX : numpy array or scipy.sparse of shape [n_samples, n_features]

The input data to project into a smaller dimensional space.

y : is not used: placeholder to allow for usage in a Pipeline.

ReturnsX_new : numpy array or scipy sparse of shape [n_samples, n_components]

Projected array.

Examples using `sklearn.random_projection.SparseRandomProjection`

`random_projection.johnson_lindenstrauss_min_dim`(...)	Find a 'safe' number of components to randomly project

42.32.3 `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.

Parametersn_samples : int or numpy array of int greater than 0,

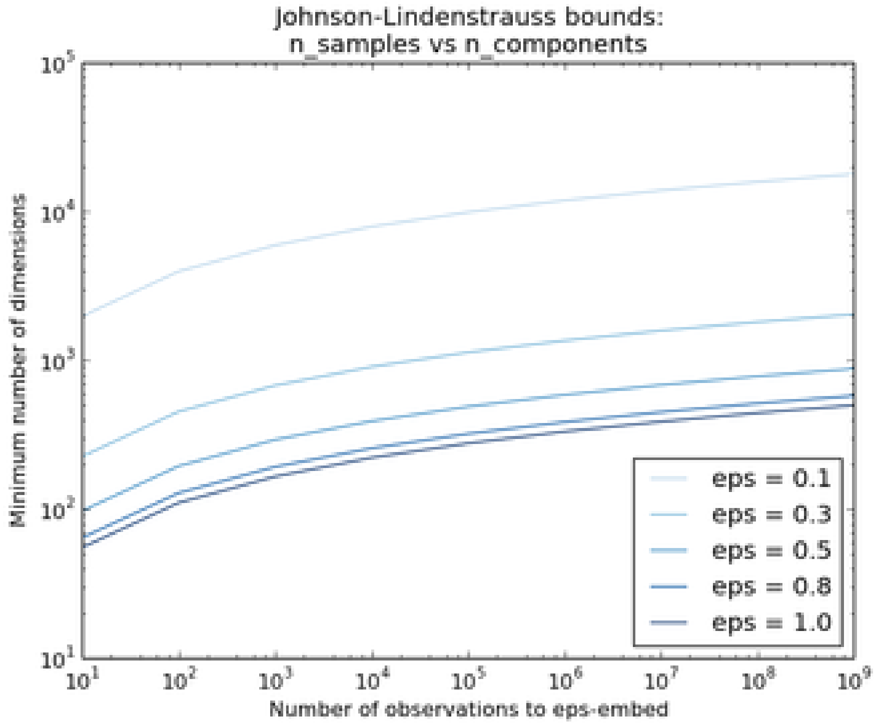


Figure 42.590: *The Johnson-Lindenstrauss bound for embedding with random projections*

Figure 42.591: *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

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.

Returnsn_components : int or numpy array of int,

The minimal number of components to guarantee with good probability an eps-embedding with n_samples.

References

[R190], [R191]

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`

42.33 `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

42.33.1 `sklearn.semi_supervised.LabelPropagation`

class `sklearn.semi_supervised.`**`LabelPropagation`**(*kernel='rbf', gamma=20, n_neighbors=7, alpha=1, max_iter=30, tol=0.001*)

Label Propagation classifier

Parameterskernel : {'knn', 'rbf'}

String identifier for kernel function to use. Only 'rbf' and 'knn' kernels are currently supported..

gamma : float

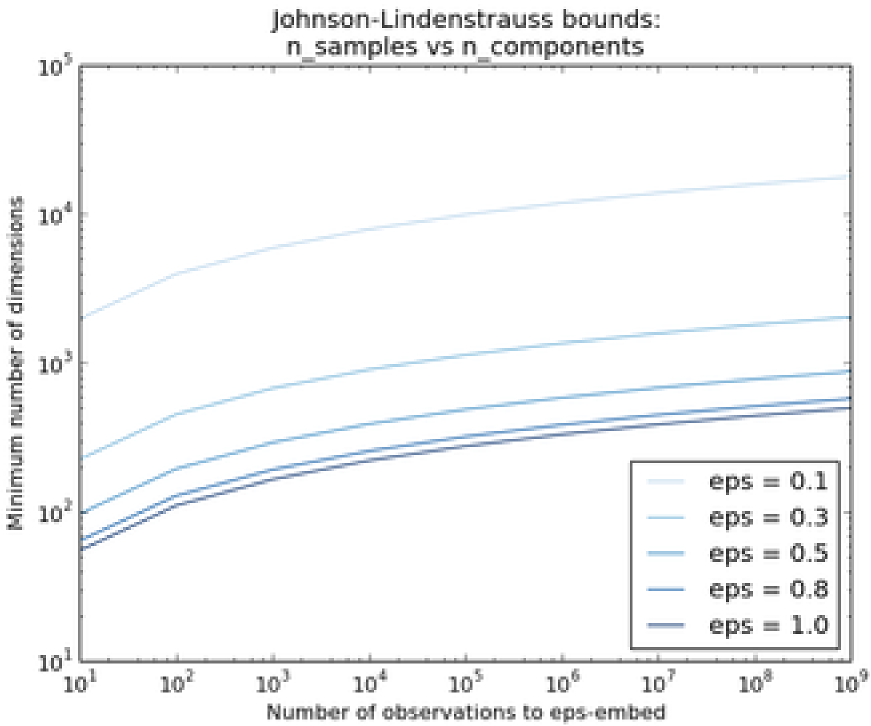


Figure 42.592: *The Johnson-Lindenstrauss bound for embedding with random projections*

Parameter for rbf kernel

n_neighbors : integer > 0

Parameter for knn kernel

alpha : float

Clamping factor

max_iter : float

Change maximum number of iterations allowed

tol : float

Convergence tolerance: threshold to consider the system at steady state

AttributesX_ : 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

```
>>> from sklearn import datasets
>>> from sklearn.semi_supervised import LabelPropagation
>>> label_prop_model = LabelPropagation()
>>> iris = datasets.load_iris()
>>> random_unlabeled_points = np.where(np.random.random_integers(0, 1,
...     size=len(iris.target)))
>>> labels = np.copy(iris.target)
>>> labels[random_unlabeled_points] = -1
>>> label_prop_model.fit(iris.data, labels)
...
LabelPropagation(...)
```

Methods

fit(X, y)	Fit a semi-supervised label propagation model based
get_params([deep])	Get parameters for this estimator.
predict(X)	Performs inductive inference across the model.
predict_proba(X)	Predict probability for each possible outcome.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

__init__(*kernel='rbf', gamma=20, n_neighbors=7, alpha=1, max_iter=30, tol=0.001*)

fit(*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.

ParametersX : array-like, 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

Returnsself : returns an instance of self.

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)
Performs inductive inference across the model.

ParametersX : array_like, shape = [n_samples, n_features]

Returnsy : array_like, shape = [n_samples]

Predictions for input data

predict_proba (*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).

ParametersX : array_like, shape = [n_samples, n_features]

Returnsprobabilities : array, shape = [n_samples, n_classes]

Normalized probability distributions across class labels

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

42.33.2 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*)

LabelSpreading model for semi-supervised learning

This model is similar to the basic Label Propgation algorithm, but uses affinity matrix based on the normalized graph Laplacian and soft clamping across the labels.

Parameters **kernel** : {'knn', 'rbf'}

String identifier for kernel function to use. Only 'rbf' and 'knn' kernels are currently supported.

gamma : float

parameter for rbf kernel

n_neighbors : integer > 0

parameter for knn kernel

alpha : float

clamping factor

max_iter : float

maximum number of iterations allowed

tol : float

Convergence tolerance: threshold to consider the system at steady state

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

```
>>> from sklearn import datasets
>>> from sklearn.semi_supervised import LabelSpreading
>>> label_prop_model = LabelSpreading()
>>> iris = datasets.load_iris()
>>> random_unlabeled_points = np.where(np.random.random_integers(0, 1,
...     size=len(iris.target)))
>>> labels = np.copy(iris.target)
>>> labels[random_unlabeled_points] = -1
>>> label_prop_model.fit(iris.data, labels)
...
LabelSpreading(...)
```

Methods

fit(X, y)	Fit a semi-supervised label propagation model based
get_params([deep])	Get parameters for this estimator.
predict(X)	Performs inductive inference across the model.
predict_proba(X)	Predict probability for each possible outcome.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

static __init__(*kernel='rbf', gamma=20, n_neighbors=7, alpha=0.2, max_iter=30, tol=0.001*)

fit(*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.

ParametersX : array-like, 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

Returnsself : returns an instance of self.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*X*)

Performs inductive inference across the model.

ParametersX : array_like, shape = [n_samples, n_features]

Returnsy : array_like, shape = [n_samples]

Predictions for input data

predict_proba (*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).

ParametersX : array_like, shape = [n_samples, n_features]

Returnsprobabilities : array, shape = [n_samples, n_classes]

Normalized probability distributions across class labels

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.semi_supervised.LabelSpreading`

42.34 `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.

42.34.1 Estimators

`svm.SVC`([C, kernel, degree, gamma, coef0, ...])	C-Support Vector Classification.
`svm.LinearSVC`([penalty, loss, dual, tol, C, ...])	Linear Support Vector Classification.
`svm.NuSVC`([nu, kernel, degree, gamma, ...])	Nu-Support Vector Classification.

Continued on next page

Table 42.221 – continued from previous page

svm.SVR([kernel, degree, gamma, coef0, tol, ...])	Epsilon-Support Vector Regression.
svm.LinearSVR([epsilon, tol, C, loss, ...])	Linear Support Vector Regression.
svm.NuSVR([nu, C, kernel, degree, gamma, ...])	Nu Support Vector Regression.
svm.OneClassSVM([kernel, degree, gamma, ...])	Unsupervised Outlier Detection.

sklearn.svm.SVC

class sklearn.svm.**SVC**(*C=1.0*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=-1*, *random_state=None*)

C-Support Vector Classification.

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 dataset with more than a couple of 10000 samples.

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*.

Parameters

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 : float, optional (default=0.0)

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. If gamma is 0.0 then 1/n_features will be used instead.

coef0 : float, optional (default=0.0)

Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

probability: boolean, optional (default=False) :

Whether to enable probability estimates. This must be enabled prior to calling *fit*, and will slow down that method.

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).

class_weight : {dict, 'auto'}, optional

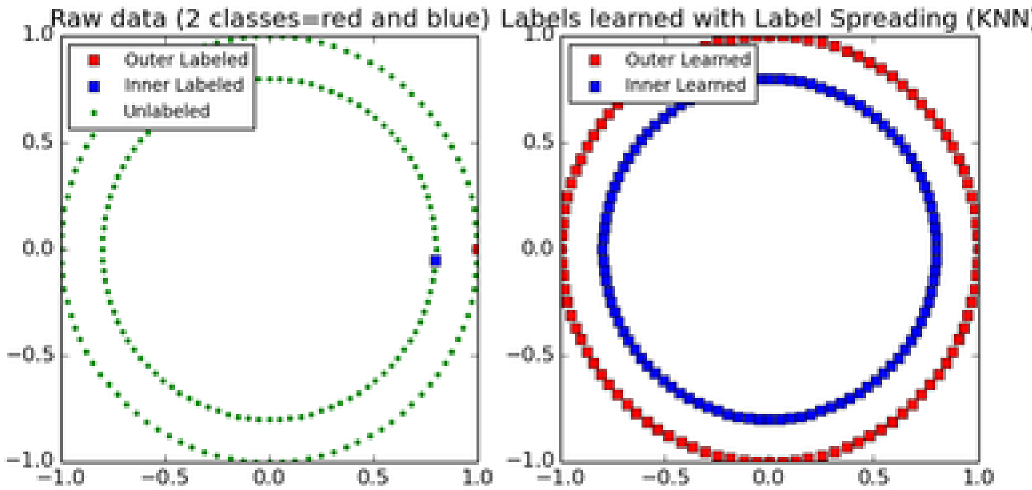


Figure 42.593: *Label Propagation learning a complex structure*

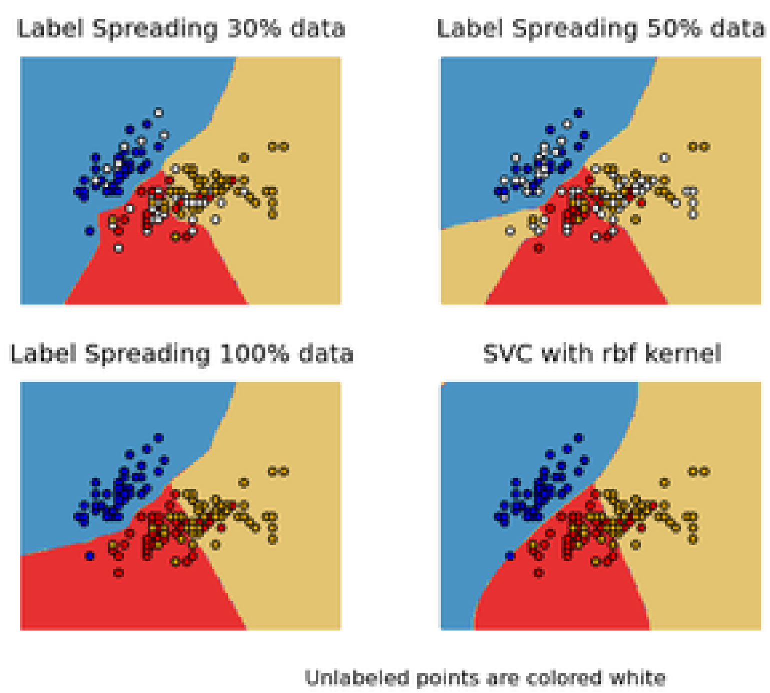


Figure 42.594: *Decision boundary of label propagation versus SVM on the Iris dataset*

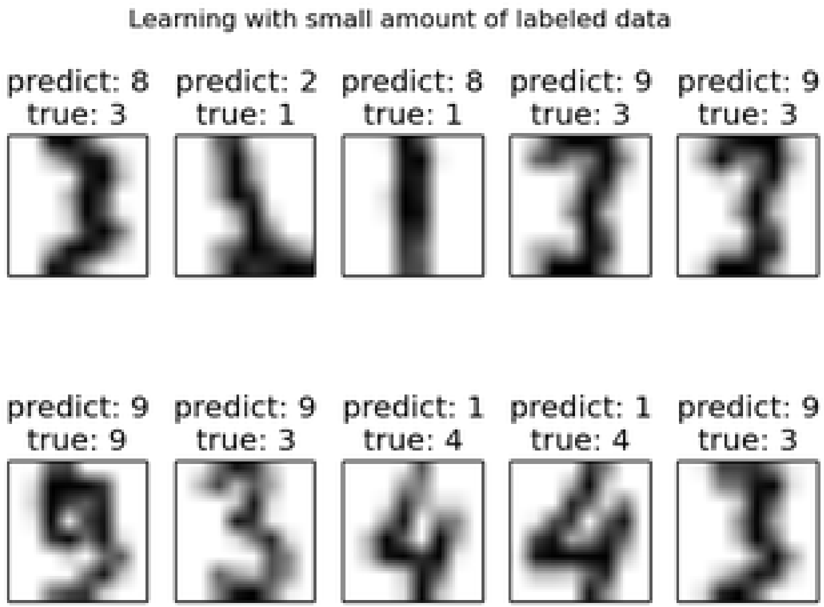


Figure 42.595: *Label Propagation digits: Demonstrating performance*

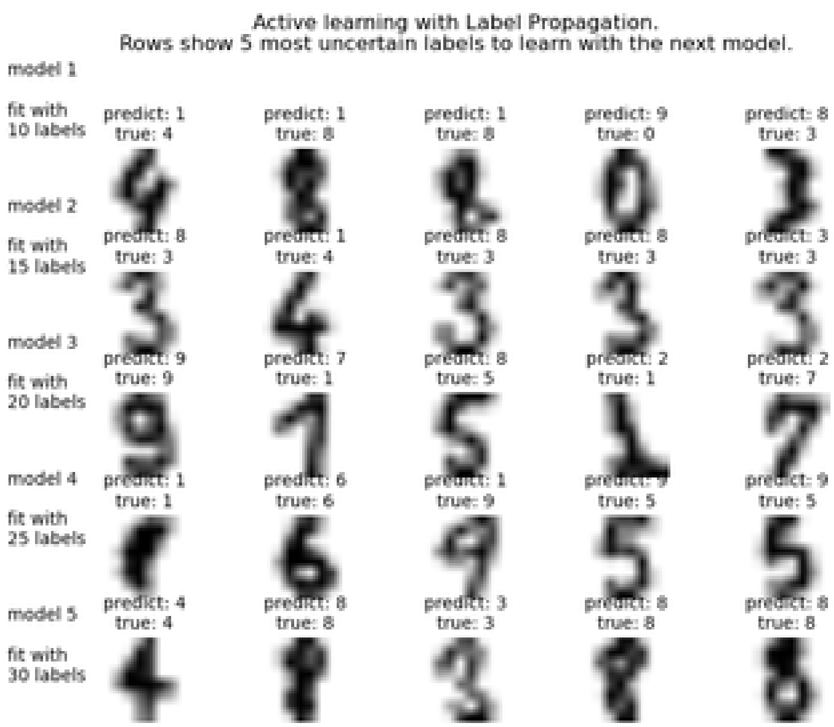


Figure 42.596: *Label Propagation digits active learning*

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 'auto' mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

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.

random_state : int seed, RandomState instance, or None (default)

The seed of the pseudo random number generator to use when shuffling the data for probability estimation.

Attributessupport_ : array-like, shape = [n_SV]

Indices of support vectors.

support_vectors_ : array-like, 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-1, n_features]

Weights assigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.

*coef_* is a readonly property derived from *dual_coef_* and *support_vectors_*.

intercept_ : array, shape = [n_class * (n_class-1) / 2]

Constants in decision function.

See also:

SVRSupport Vector Machine for Regression implemented using libsvm.

LinearSVCScalable Linear Support Vector Machine for classification implemented using liblinear. Check the See also section of LinearSVC for more comparison element.

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()
>>> clf.fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
```

```
    gamma=0.0, kernel='rbf', max_iter=-1, probability=False,
    random_state=None, shrinking=True, tol=0.001, verbose=False)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

decision_function(X)	Distance of the samples X to the separating hyperplane.
fit(X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_params([deep])	Get parameters for this estimator.
predict(X)	Perform classification on samples in X.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

static __init__(*C=1.0*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=-1*, *random_state=None*)

decision_function(*X*)

Distance of the samples X to the separating hyperplane.

ParametersX : array-like, shape = [n_samples, n_features]

For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train].

ReturnsX : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

fit(*X*, *y*, *sample_weight=None*)

Fit the SVM model according to the given training data.

ParametersX : {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.

Returnsself : object

Returns self.

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 (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)
Perform classification on samples in X.

For an one-class model, +1 or -1 is returned.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returnsy_pred : array, shape = [n_samples]

Class labels for samples in X.

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.svm.SVC`

`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.

Figure 42.597: *Concatenating multiple feature extraction methods*

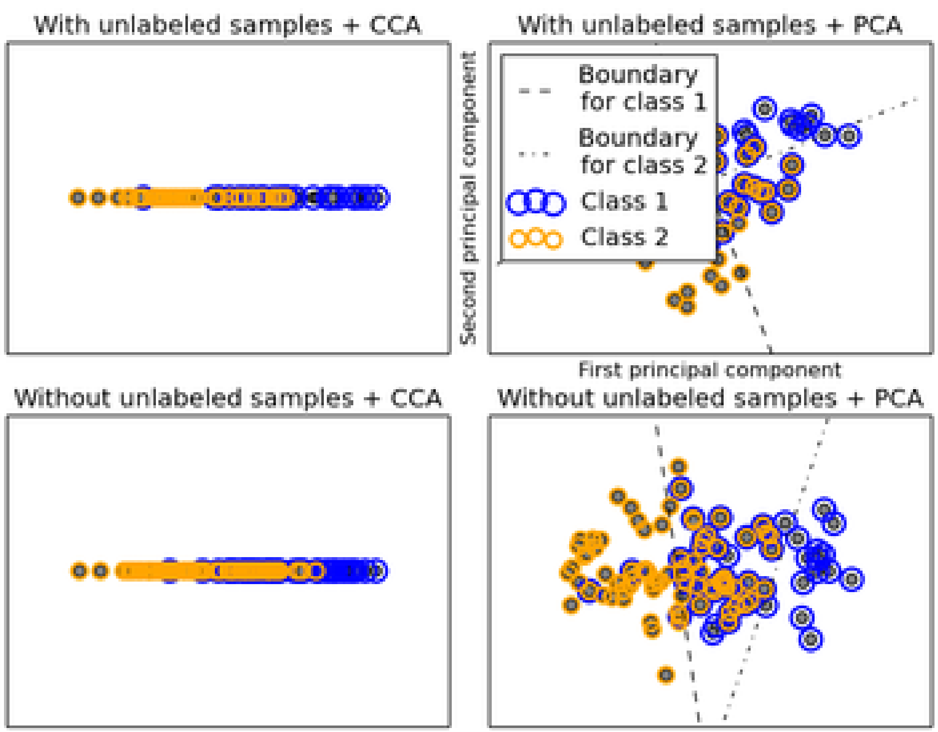


Figure 42.598: *Multilabel classification*

Figure 42.599: *Feature Union with Heterogeneous Data Sources*

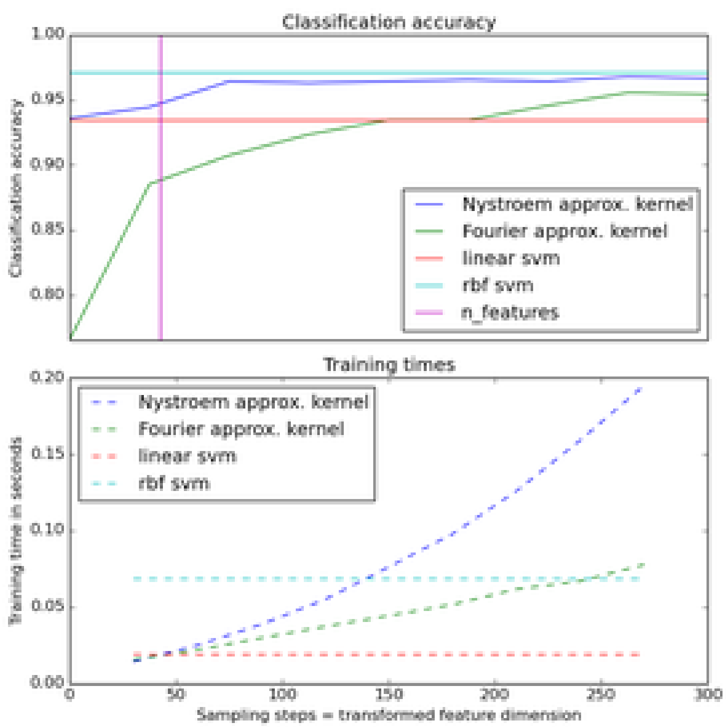


Figure 42.600: *Explicit feature map approximation for RBF kernels*

Figure 42.601: *Faces recognition example using eigenfaces and SVMs*

Figure 42.602: *Libsvm GUI*

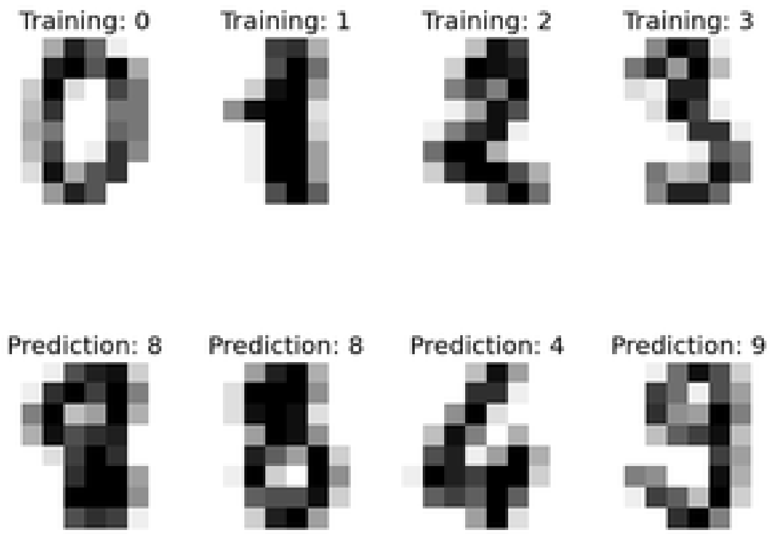


Figure 42.603: *Recognizing hand-written digits*

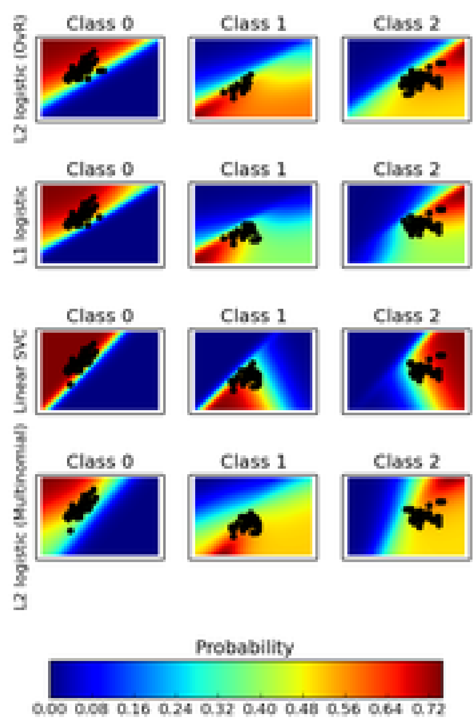


Figure 42.604: *Plot classification probability*

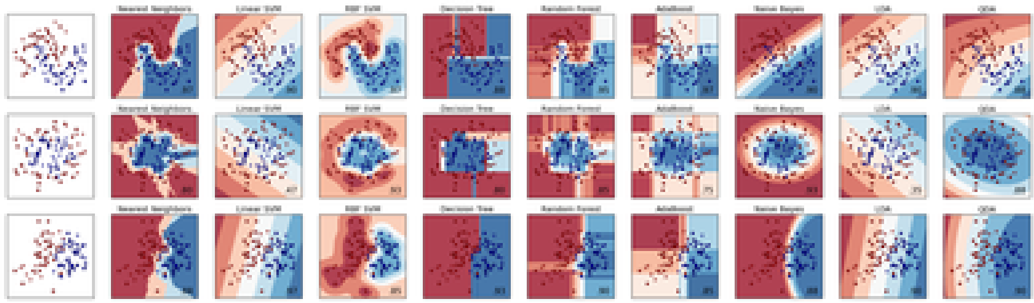


Figure 42.605: *Classifier comparison*

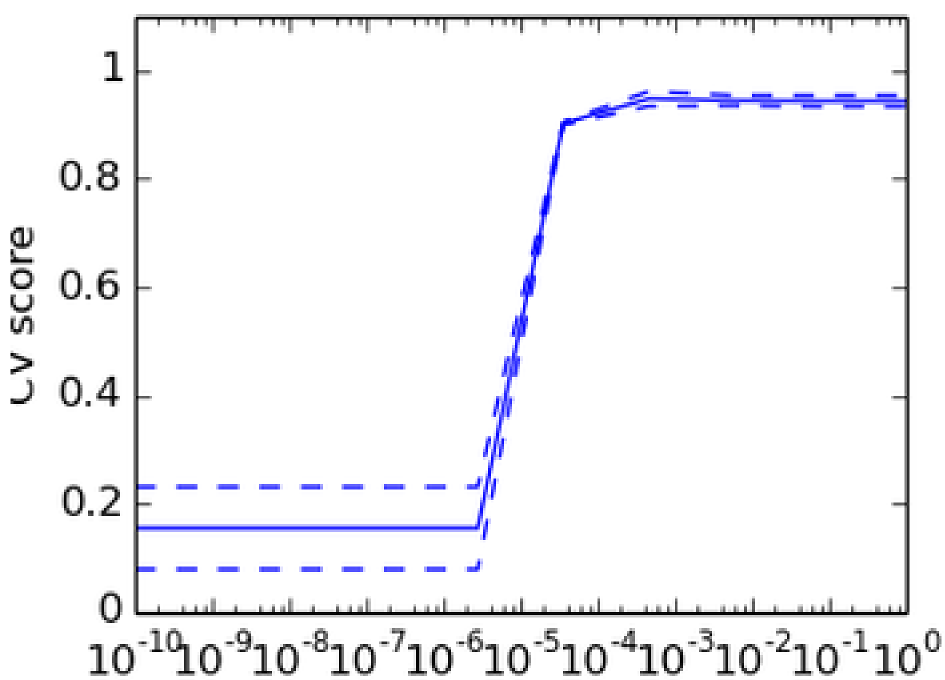


Figure 42.606: *Cross-validation on Digits Dataset Exercise*

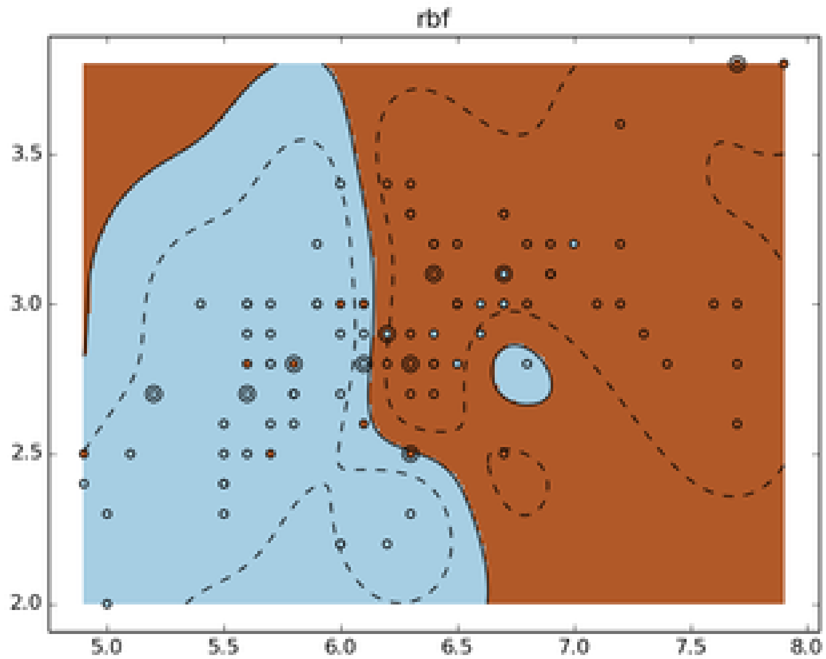


Figure 42.607: *SVM Exercise*

Figure 42.608: *Pipeline Anova SVM*

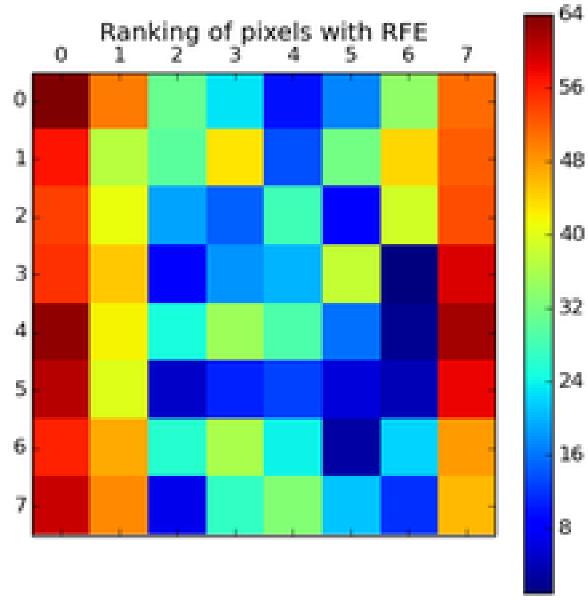


Figure 42.609: *Recursive feature elimination*

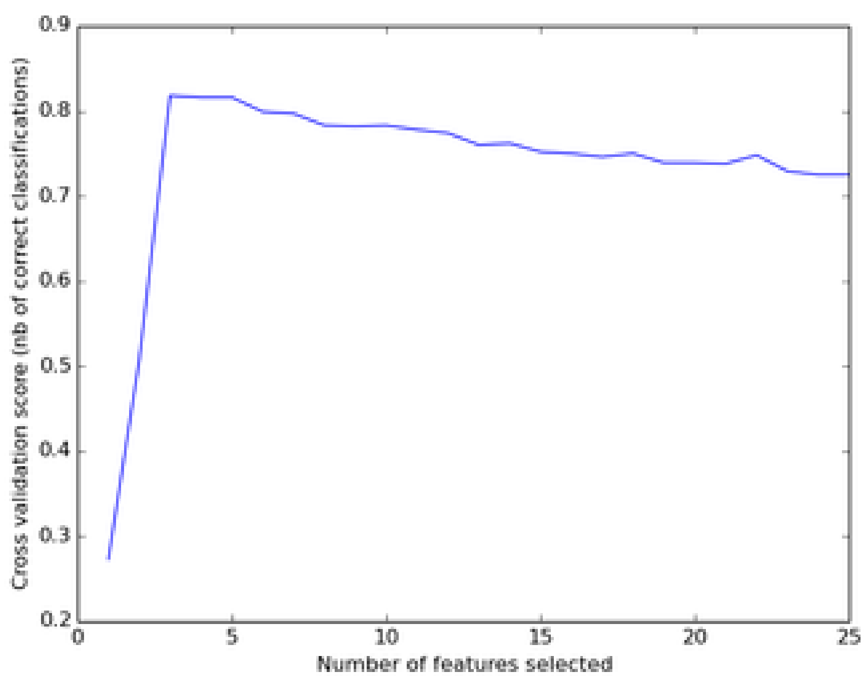


Figure 42.610: *Recursive feature elimination with cross-validation*

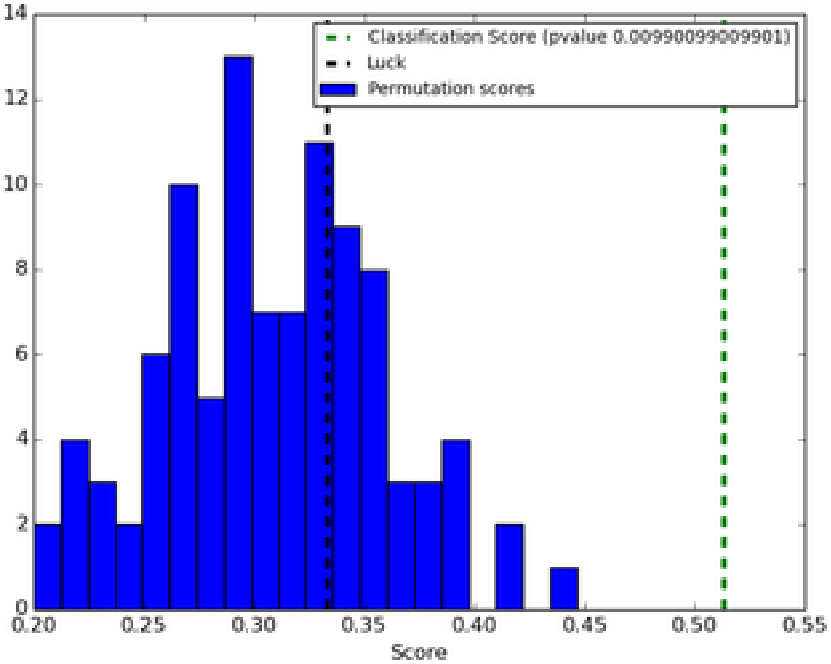


Figure 42.611: *Test with permutations the significance of a classification score*

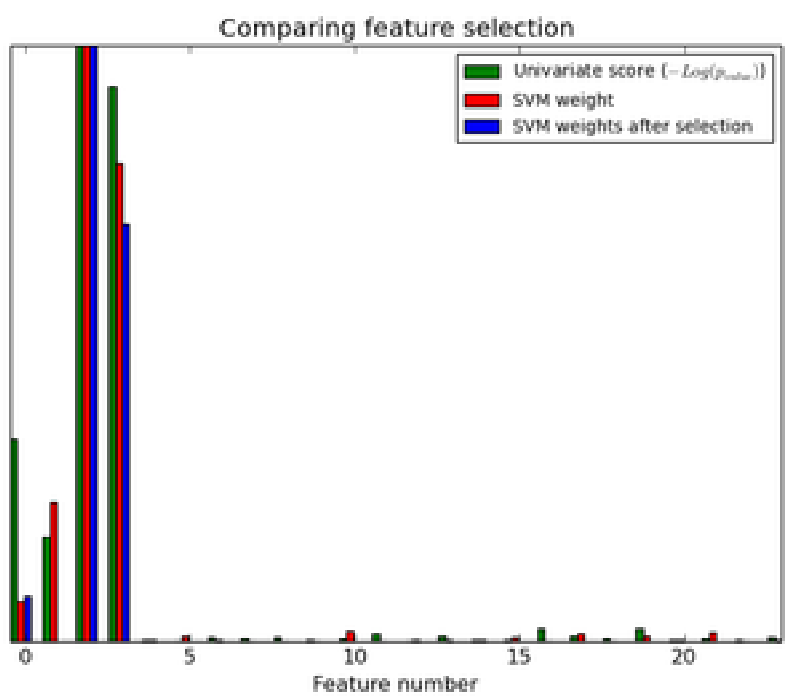


Figure 42.612: *Univariate Feature Selection*

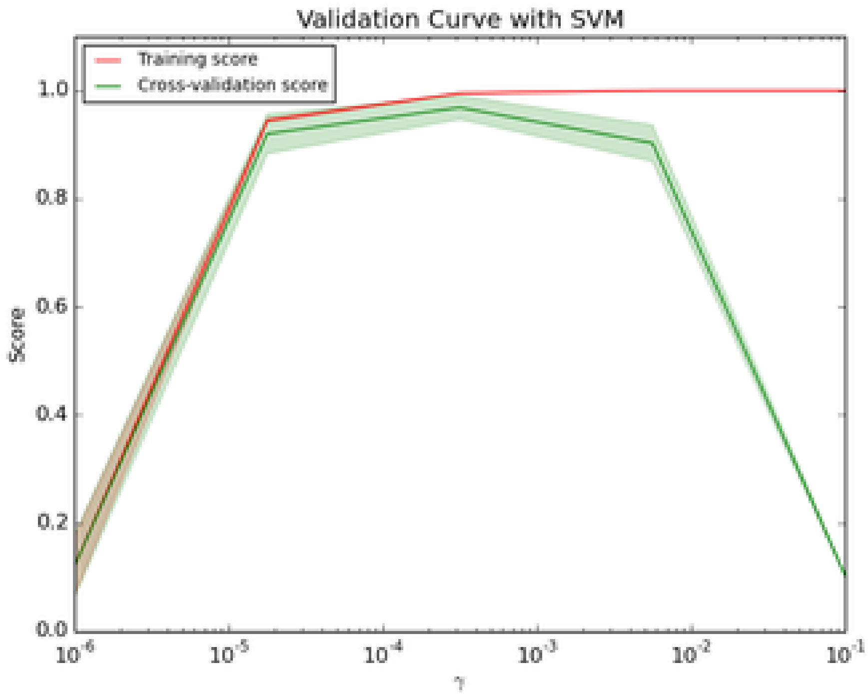


Figure 42.613: *Plotting Validation Curves*

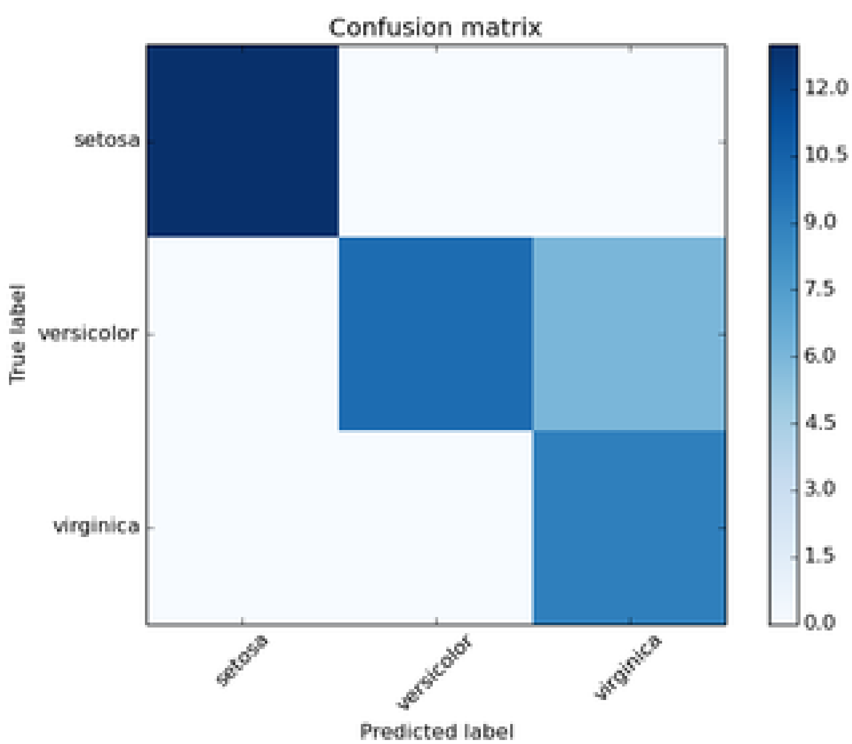


Figure 42.614: *Confusion matrix*

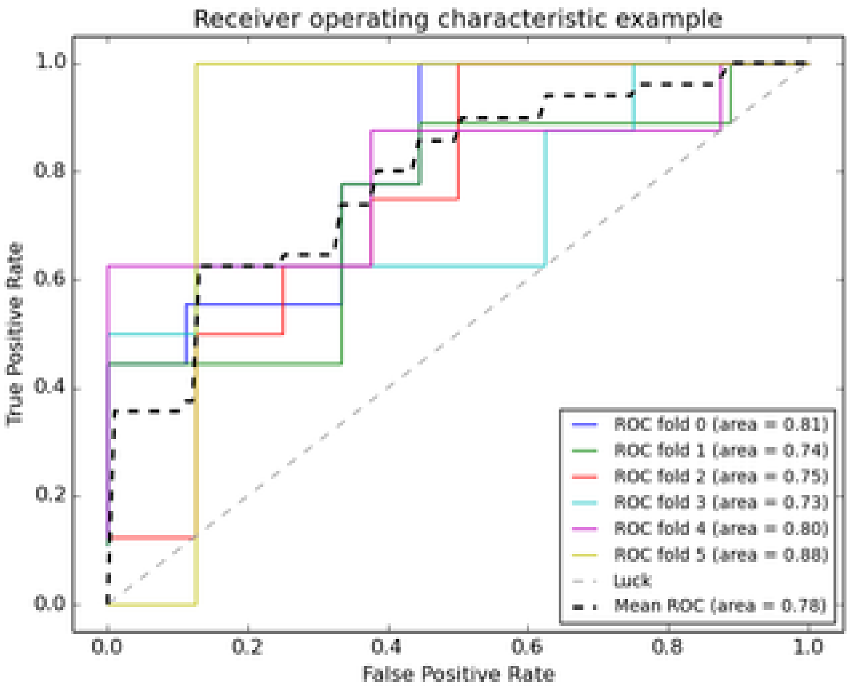


Figure 42.615: *Receiver Operating Characteristic (ROC) with cross validation*

Figure 42.616: *Parameter estimation using grid search with cross-validation*

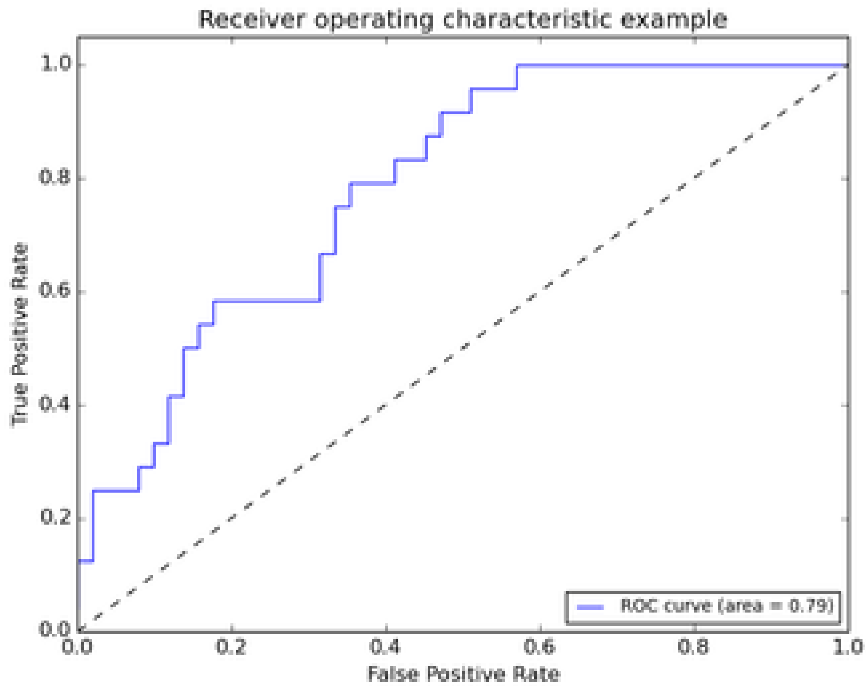


Figure 42.617: *Receiver Operating Characteristic (ROC)*

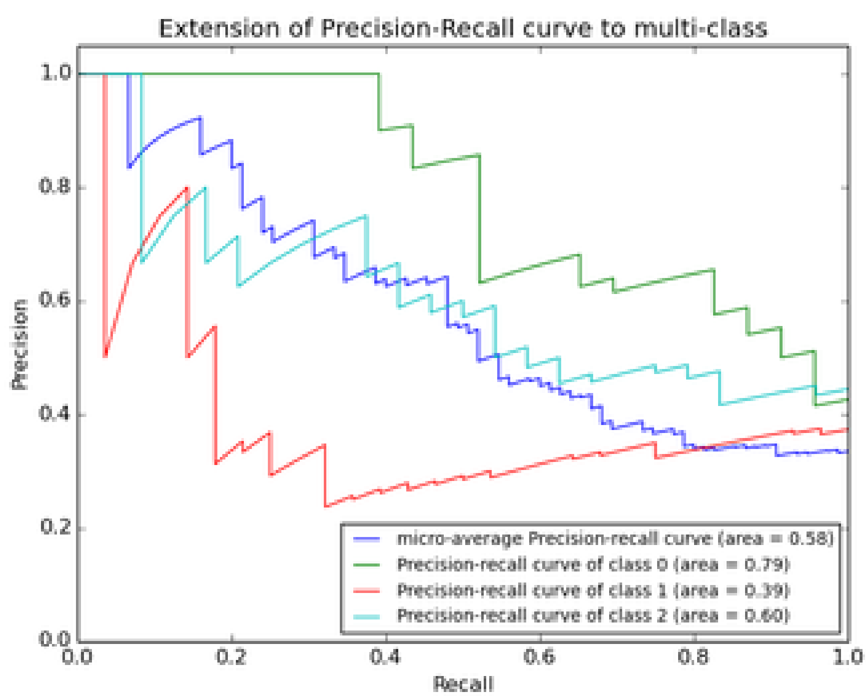


Figure 42.618: *Precision-Recall*

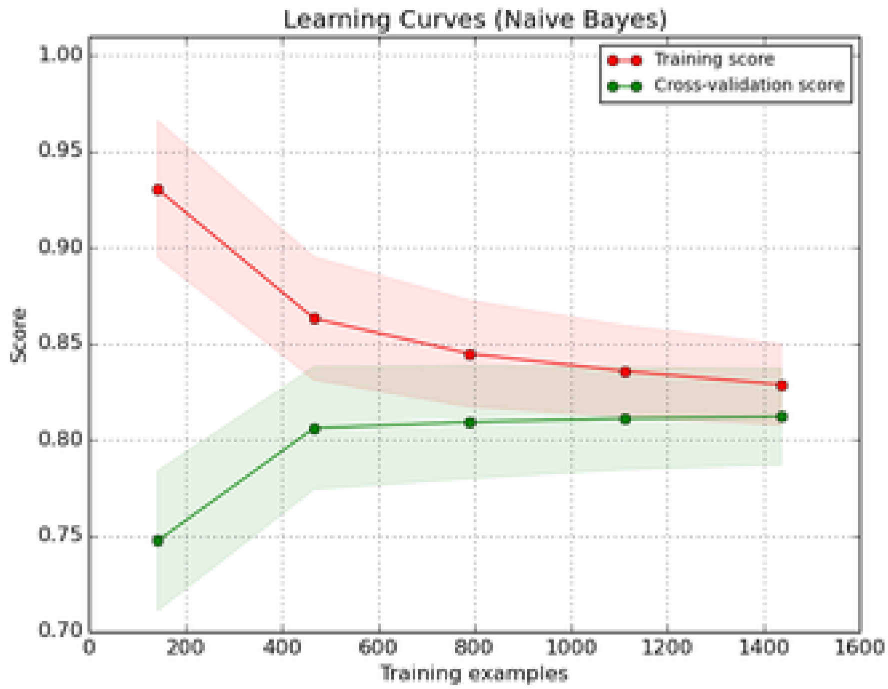


Figure 42.619: *Plotting Learning Curves*

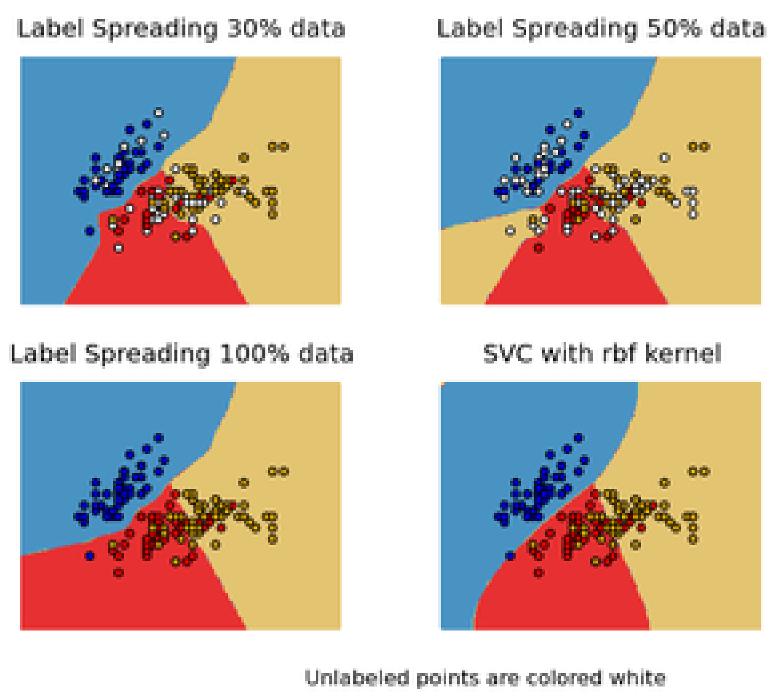


Figure 42.620: *Decision boundary of label propagation versus SVM on the Iris dataset*

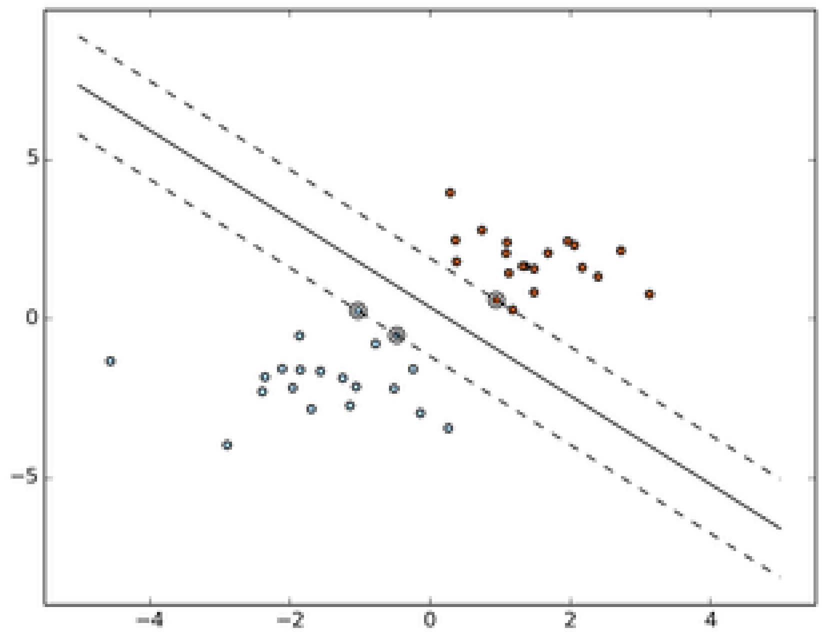


Figure 42.621: *SVM: Maximum margin separating hyperplane*

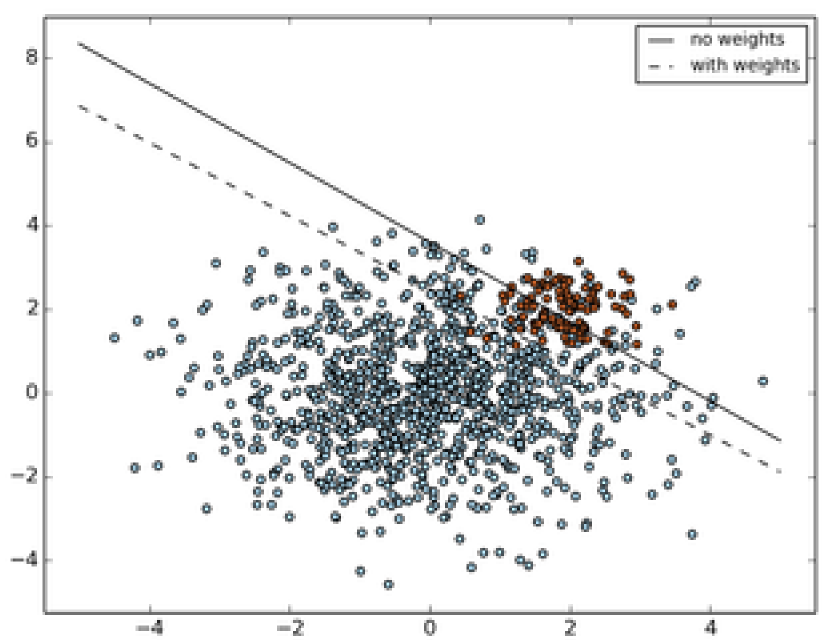


Figure 42.622: *SVM: Separating hyperplane for unbalanced classes*

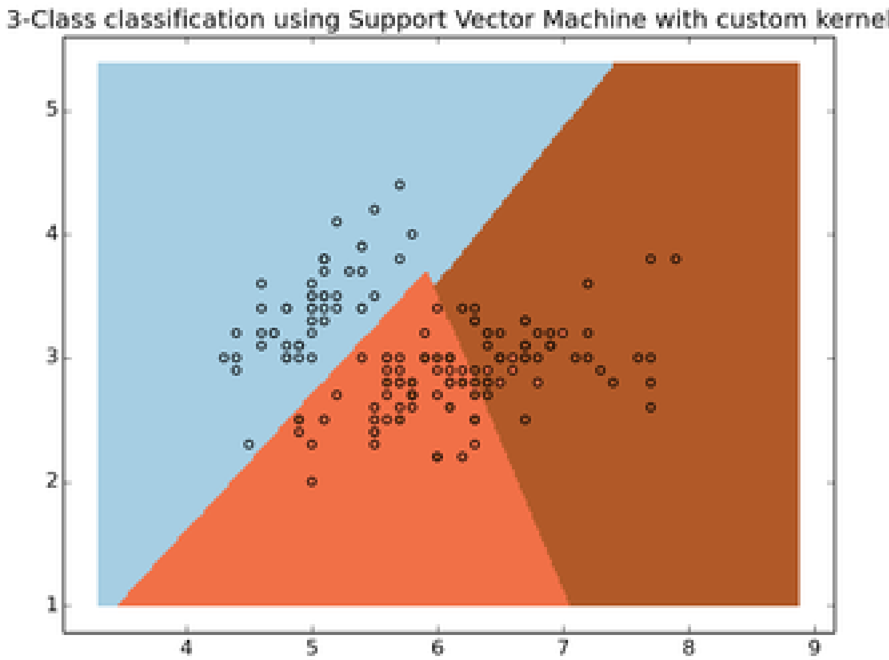


Figure 42.623: *SVM with custom kernel*

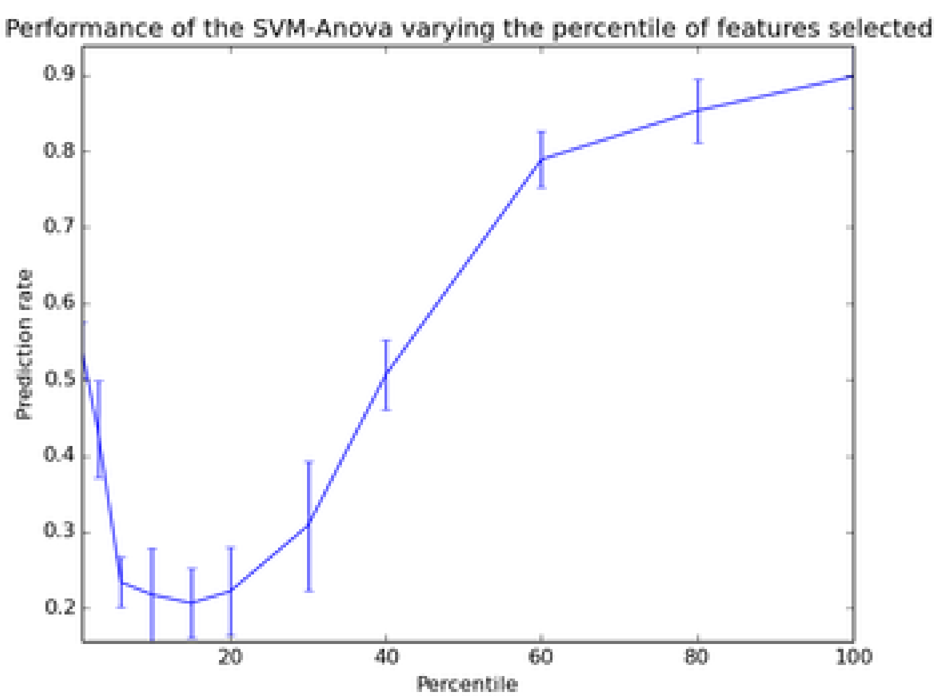


Figure 42.624: *SVM-Anova: SVM with univariate feature selection*

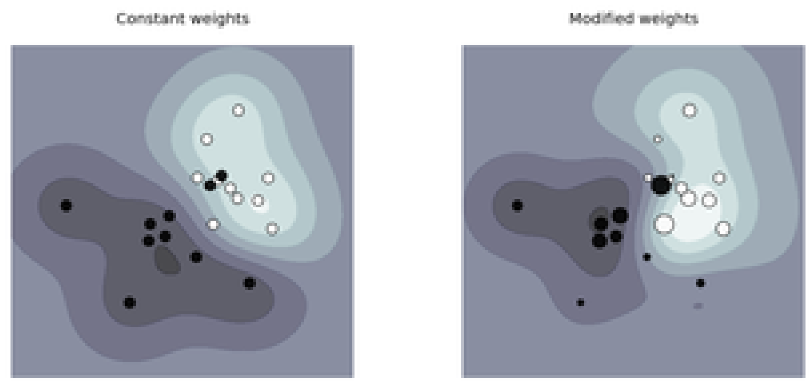


Figure 42.625: *SVM: Weighted samples*

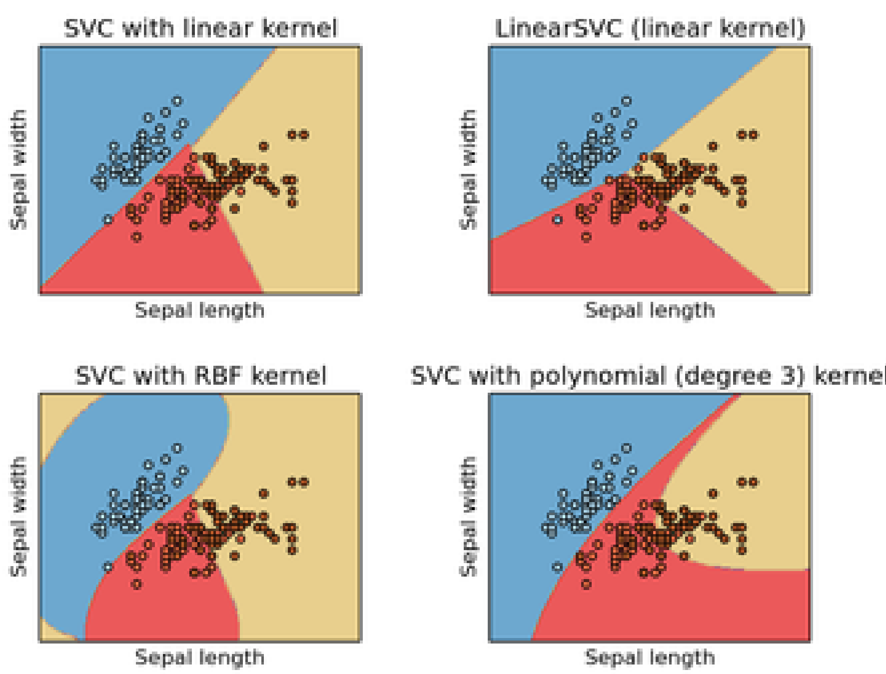


Figure 42.626: *Plot different SVM classifiers in the iris dataset*

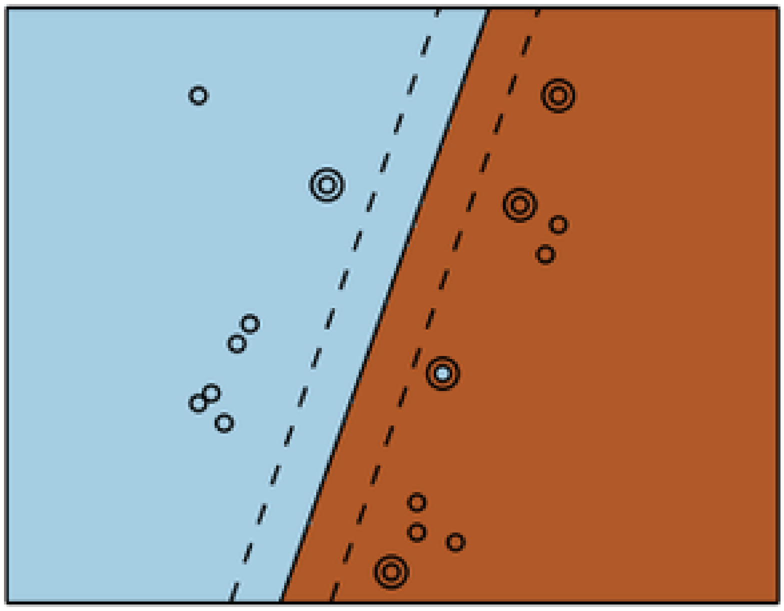


Figure 42.627: *SVM-Kernels*

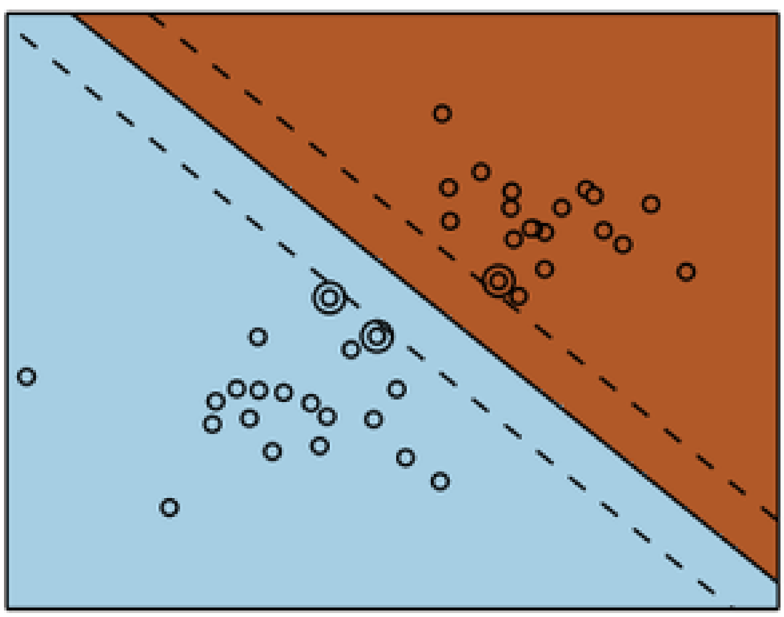


Figure 42.628: *SVM Margins Example*

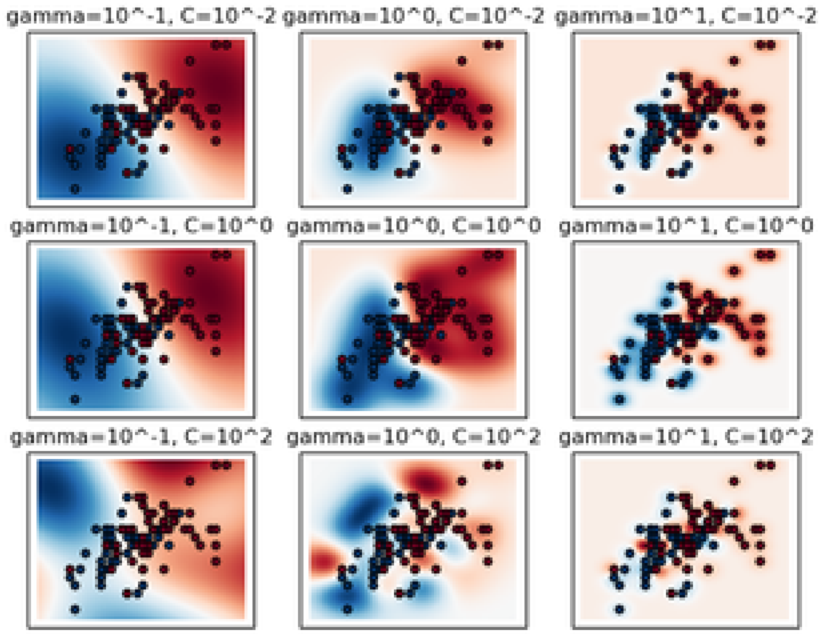


Figure 42.629: *RBF SVM parameters*

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.

ParametersC : float, optional (default=1.0)

Penalty parameter C of the error term.

loss : string, '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.

penalty : string, '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.

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.

multi_class: string, '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 an theoretical perspective as it is consistent it is seldom used in practice and 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 : boolean, optional (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).

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, 'auto'}, 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 'auto' mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

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 seed, RandomState instance, or None (default=None)

The seed of the pseudo random number generator to use when shuffling the data.

max_iter : int, (default=1000)

The maximum number of iterations to be run.

Attributescoef_ : 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 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.

See also:

SVCImplementation 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.SGDClassifierSGDClassifier 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

Methods

decision_function(X)	Predict confidence scores for samples.
densify()	Convert coefficient matrix to dense array format.
fit(X, y)	Fit the model according to the given training data.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict class labels for samples in X.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.
sparsify()	Convert coefficient matrix to sparse format.
transform(X[, threshold])	Reduce X to its most important features.

static __init__(*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*)

decision_function(*X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

Returnsarray, 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()

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.

Returnsself: estimator :

static fit (*X*, *y*)
Fit the model according to the given training data.

ParametersX : {array-like, sparse matrix}, 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

Returnsself : object

Returns self.

fit_transform (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)
Predict class labels for samples in X.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

Samples.

ReturnsC : array, shape = [n_samples]

Predicted class label per sample.

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

sparsify()

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.

Returnsself: 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.

transform(*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

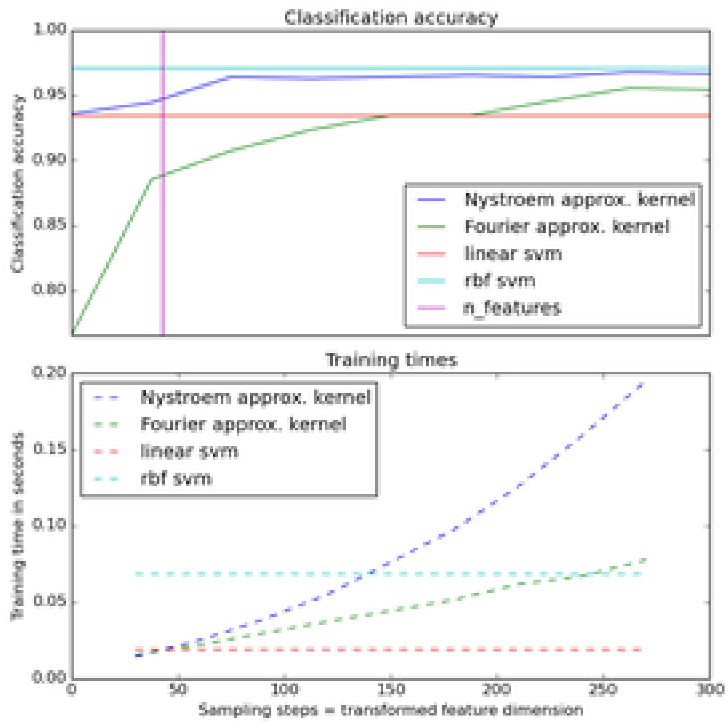


Figure 42.630: *Explicit feature map approximation for RBF kernels*

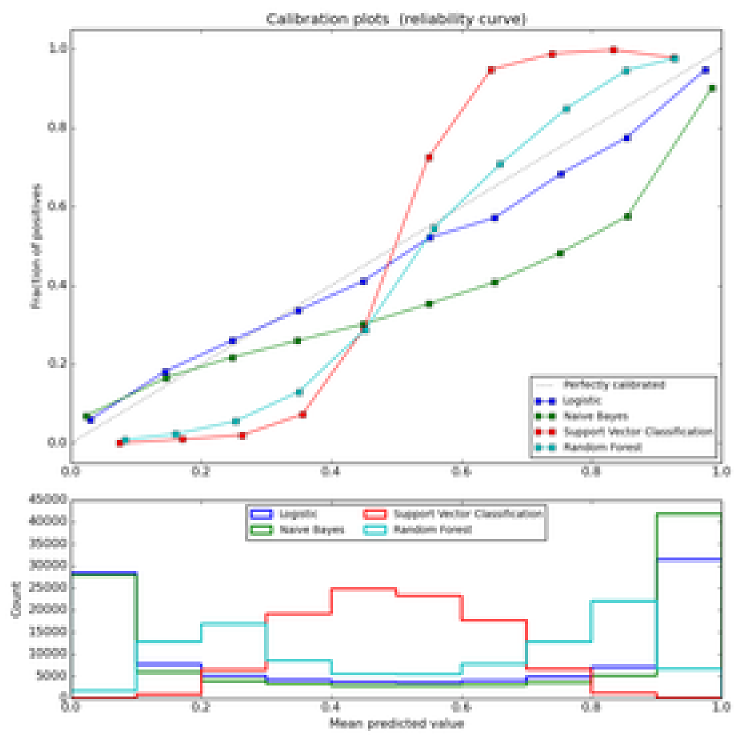


Figure 42.631: *Comparison of Calibration of Classifiers*

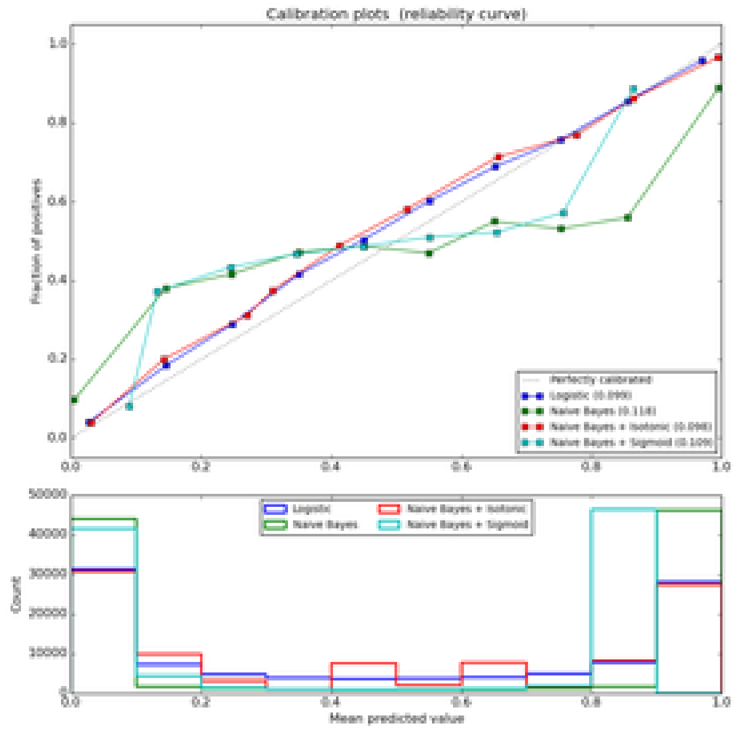


Figure 42.632: *Probability Calibration curves*

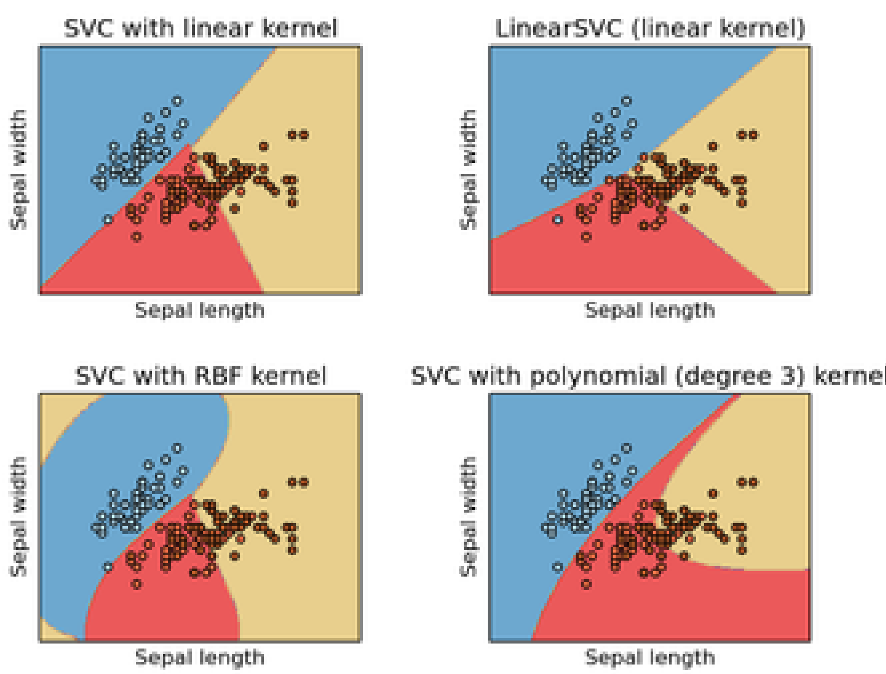


Figure 42.633: *Plot different SVM classifiers in the iris dataset*

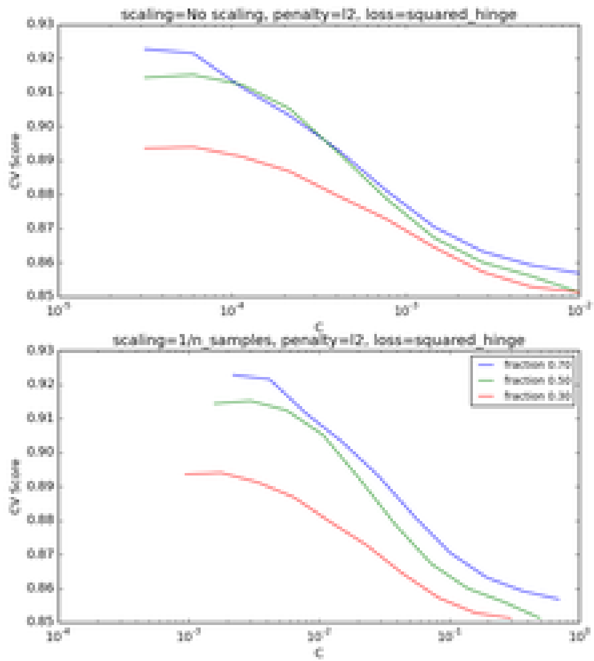


Figure 42.634: *Scaling the regularization parameter for SVCs*

Figure 42.635: *Classification of text documents using sparse features*

Examples using `sklearn.svm.LinearSVC`

`sklearn.svm.NuSVC`

class `sklearn.svm.`**`NuSVC`**(*nu=0.5*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *verbose=False*, *max_iter=-1*, *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.

Parametersnu : 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 kernel function is significant only in poly, rbf, sigmoid.

gamma : float, optional (default=0.0)

Kernel coefficient for rbf and poly, if gamma is 0.0 then 1/n_features will be taken.

coef0 : float, optional (default=0.0)

Independent term in kernel function. It is only significant in poly/sigmoid.

probability: boolean, optional (default=False) :

Whether to enable probability estimates. This must be enabled prior to calling *fit*, and will slow down that method.

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.

random_state : int seed, RandomState instance, or None (default)

The seed of the pseudo random number generator to use when shuffling the data for probability estimation.

Attributessupport_ : array-like, shape = [n_SV]

Indices of support vectors.

support_vectors_ : array-like, shape = [n_SV, n_features]

Support vectors.

n_support_ : array-like, dtype=int32, shape = [n_class]

Number of support vector 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-1, n_features]

Weights assigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.

*coef_* is readonly property derived from *dual_coef_* and *support_vectors_*.

intercept_ : array, shape = [n_class * (n_class-1) / 2]

Constants in decision function.

See also:

`SVC`Support Vector Machine for classification using libsvm.

`LinearSVC`Scalable linear Support Vector Machine for classification using liblinear.

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(cache_size=200, coef0=0.0, degree=3, gamma=0.0, kernel='rbf',
      max_iter=-1, nu=0.5, probability=False, random_state=None,
      shrinking=True, tol=0.001, verbose=False)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

`decision_function`(X)	Distance of the samples X to the separating hyperplane.
`fit`(X, y[, sample_weight])	Fit the SVM model according to the given training data.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Perform classification on samples in X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.

static `__init__`(*nu=0.5*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *verbose=False*, *max_iter=-1*, *random_state=None*)

`decision_function`(*X*)
Distance of the samples X to the separating hyperplane.

ParametersX : array-like, shape = [n_samples, n_features]

For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train].

ReturnsX : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

`fit`(*X*, *y*, *sample_weight=None*)
Fit the SVM model according to the given training data.

ParametersX : {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.

Returnsself : object

Returns self.

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`(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

`predict`(*X*)
Perform classification on samples in X.

For an one-class model, +1 or -1 is returned.

ParametersX : {array-like, sparse matrix}, shape = [n_samples, n_features]

For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returnsy_pred : array, shape = [n_samples]

Class labels for samples in X.

score (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

Examples using sklearn.svm.NuSVC

sklearn.svm.SVR

class sklearn.svm.**SVR** (*kernel='rbf'*, *degree=3*, *gamma=0.0*, *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.

ParametersC : float, optional (default=1.0)

Penalty parameter C of the error term.

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.

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.

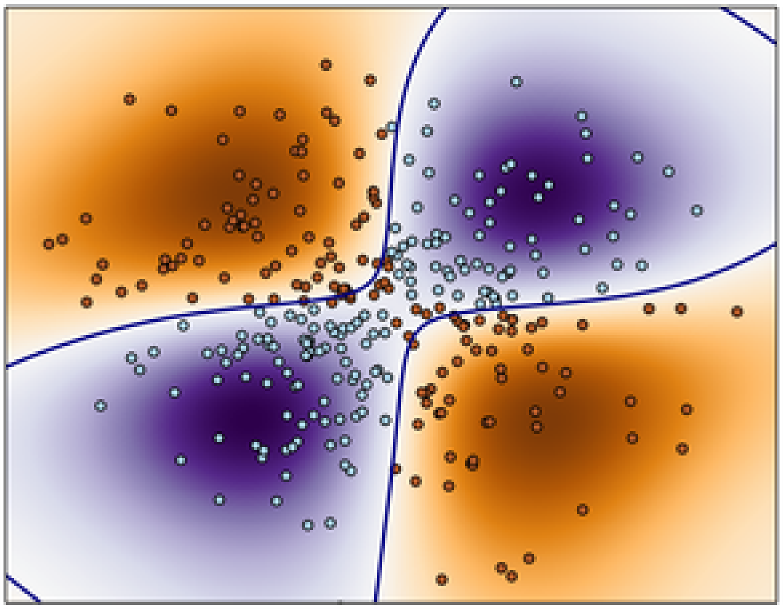


Figure 42.636: *Non-linear SVM*

degree : int, optional (default=3)

Degree of kernel function is significant only in poly, rbf, sigmoid.

gamma : float, optional (default=0.0)

Kernel coefficient for rbf and poly, if gamma is 0.0 then 1/n_features will be taken.

coef0 : float, optional (default=0.0)

independent term in kernel function. It is only significant in poly/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.

Attributessupport_ : array-like, shape = [n_SV]

Indices of support vectors.

support_vectors_ : array-like, shape = [nSV, 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 linear kernel.

*coef_* is readonly property derived from *dual_coef_* and *support_vectors_*.

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.

Examples

```
>>> from sklearn.svm import SVR
>>> 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 = SVR(C=1.0, epsilon=0.2)
>>> clf.fit(X, y)
SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.2, gamma=0.0,
    kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

Methods

decision_function(X)	Distance of the samples X to the separating hyperplane.
fit(X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_params([deep])	Get parameters for this estimator.
predict(X)	Perform regression on samples in X.
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*kernel='rbf', degree=3, gamma=0.0, coef0=0.0, tol=0.001, C=1.0, epsilon=0.1, shrinking=True, cache_size=200, verbose=False, max_iter=-1*)

decision_function(*X*)

Distance of the samples X to the separating hyperplane.

Parameters X : array-like, shape = [n_samples, n_features]

For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train].

ReturnsX : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

fit (*X*, *y*, *sample_weight=None*)

Fit the SVM model according to the given training data.

ParametersX : {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.

Returnsself : object

Returns self.

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 (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Perform regression on samples in X.

For an one-class model, +1 or -1 is returned.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

For kernel="precomputed", the expected shape of X is (n_samples_test, n_samples_train).

Returnsy_pred : array, shape (n_samples,)

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.svm.SVR`

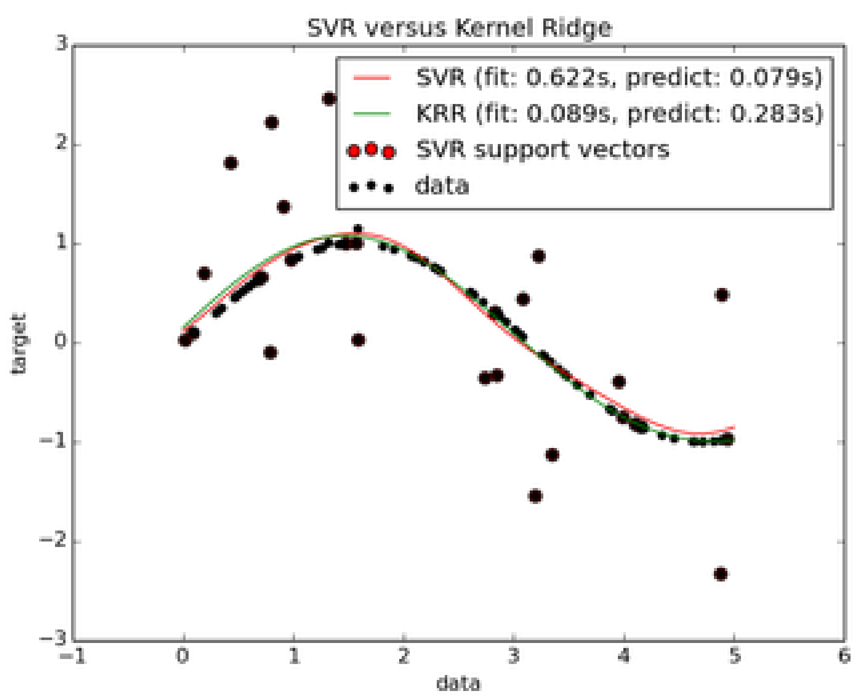


Figure 42.637: *Comparison of kernel ridge regression and SVR*

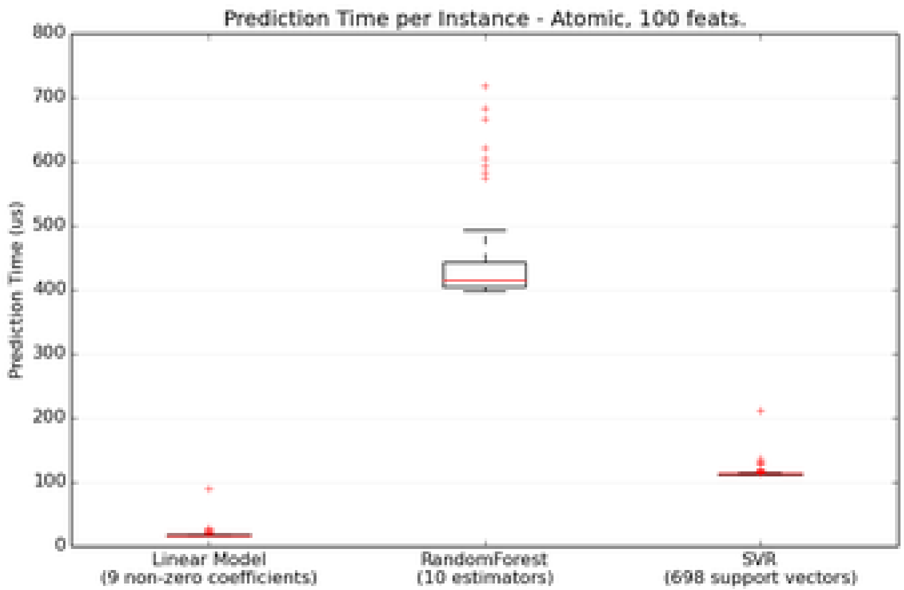


Figure 42.638: *Prediction Latency*

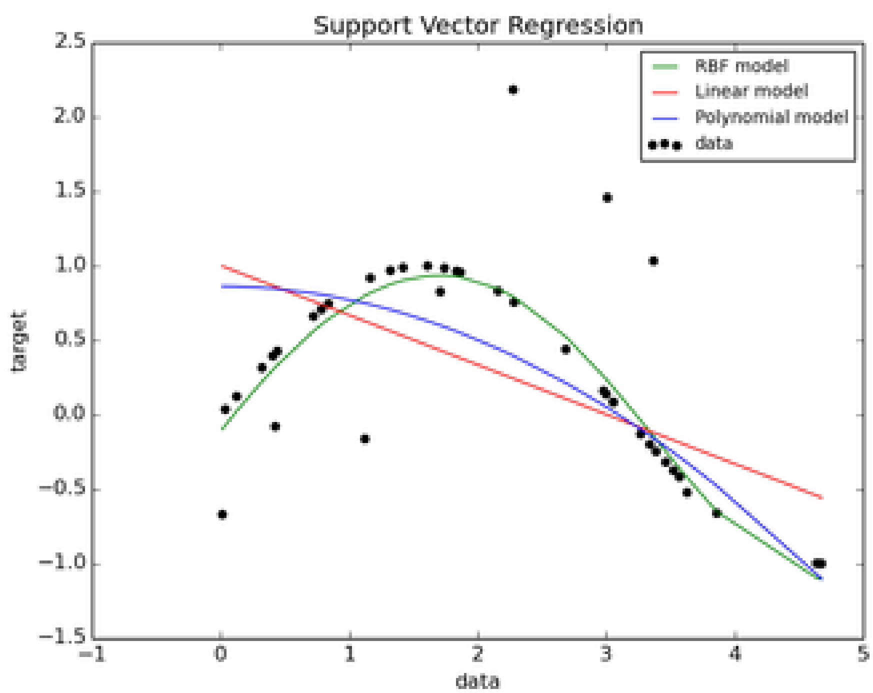


Figure 42.639: *Support Vector Regression (SVR) using linear and non-linear kernels*

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.

ParametersC : float, optional (default=1.0)

Penalty parameter C of the error term. The penalty is a squared l2 penalty. The bigger this parameter, the less regularization is used.

loss : string, 'epsilon_insensitive' or 'squared_epsilon_insensitive' (default='epsilon_insensitive')

Specifies the loss function. 'l1' is the epsilon-insensitive loss (standard SVR) while 'l2' is the squared epsilon-insensitive loss.

epsilon : float, optional (default=0.1)

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.

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.

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 (e.g. 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.

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 seed, RandomState instance, or None (default=None)

The seed of the pseudo random number generator to use when shuffling the data.

max_iter : int, (default=1000)

The maximum number of iterations to be run.

Attributescoef_ : 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 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.

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.

Methods

decision_function(X)	Decision function of the linear model.
fit(X, y)	Fit the model according to the given training data.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict using the linear model
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*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*)

decision_function(*X*)

Decision function of the linear model.

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

static fit(*X*, *y*)

Fit the model according to the given training data.

ParametersX : {array-like, sparse matrix}, 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

Returnsself : object

Returns self.

get_params (*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)
Predict using the linear model

ParametersX : {array-like, sparse matrix}, shape = (n_samples, n_features)

Samples.

ReturnsC : array, shape = (n_samples,)

Returns predicted values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

sklearn.svm.NuSVR

class sklearn.svm.**NuSVR** (*nu=0.5*, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *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.

ParametersC : float, optional (default=1.0)

Penalty parameter C of the error term.

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.

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 kernel function is significant only in poly, rbf, sigmoid.

gamma : float, optional (default=0.0)

Kernel coefficient for rbf and poly, if gamma is 0.0 then 1/n_features will be taken.

coef0 : float, optional (default=0.0)

Independent term in kernel function. It is only significant in poly/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.

Attributessupport_ : array-like, shape = [n_SV]

Indices of support vectors.

support_vectors_ : array-like, shape = [nSV, 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 linear kernel.

*coef_* is readonly property derived from *dual_coef_* and *support_vectors_*.

intercept_ : array, shape = [1]

Constants in decision function.

See also:

NuSVCSupport Vector Machine for classification implemented with libsvm with a parameter to control the number of support vectors.

SVRepsilon Support Vector Machine for regression implemented with libsvm.

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(C=1.0, cache_size=200, coef0=0.0, degree=3, gamma=0.0, kernel='rbf',
      max_iter=-1, nu=0.1, shrinking=True, tol=0.001, verbose=False)
```

Methods

decision_function(X)	Distance of the samples X to the separating hyperplane.
fit(X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_params([deep])	Get parameters for this estimator.
predict(X)	Perform regression on samples in X.
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

static __init__(*nu=0.5*, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *tol=0.001*, *cache_size=200*, *verbose=False*, *max_iter=-1*)

decision_function(*X*)

Distance of the samples X to the separating hyperplane.

ParametersX : array-like, shape = [n_samples, n_features]

For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train].

ReturnsX : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

fit(*X*, *y*, *sample_weight=None*)

Fit the SVM model according to the given training data.

ParametersX : {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.

Returnsself : object

Returns self.

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 (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*X*)

Perform regression on samples in X.

For an one-class model, +1 or -1 is returned.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

For kernel="precomputed", the expected shape of X is (n_samples_test, n_samples_train).

Returnsy_pred : array, shape (n_samples,)

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.svm.NuSVR`

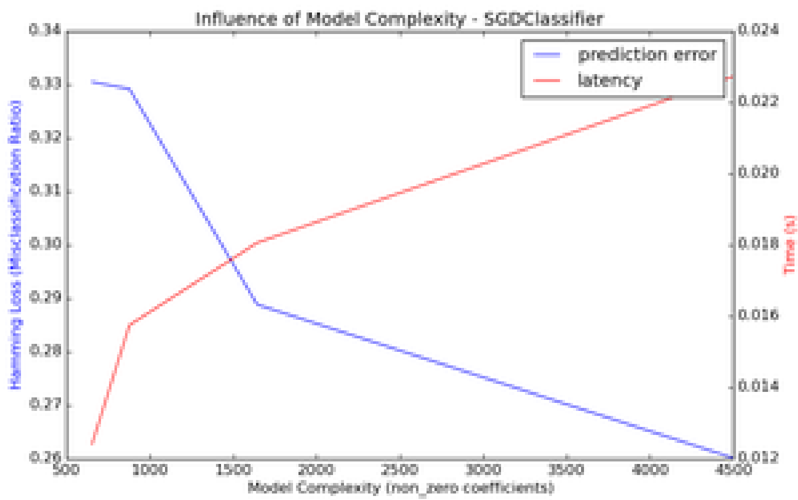


Figure 42.640: *Model Complexity Influence*

`sklearn.svm`.OneClassSVM

class `sklearn.svm.`**OneClassSVM**(*kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *tol=0.001*, *nu=0.5*, *shrinking=True*, *cache_size=200*, *verbose=False*, *max_iter=-1*, *random_state=None*)

Unsupervised Outlier Detection.

Estimate the support of a high-dimensional distribution.

The implementation is based on libsvm.

Parameterskernel : 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.

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.

degree : int, optional (default=3)

Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma : float, optional (default=0.0)

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. If gamma is 0.0 then 1/n_features will be used instead.

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.

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.

random_state : int seed, RandomState instance, or None (default)

The seed of the pseudo random number generator to use when shuffling the data for probability estimation.

Attributessupport_ : array-like, shape = [n_SV]

Indices of support vectors.

support_vectors_ : array-like, shape = [nSV, n_features]

Support vectors.

dual_coef_ : array, shape = [n_classes-1, n_SV]

Coefficients of the support vectors in the decision function.

coef_ : array, shape = [n_classes-1, n_features]

Weights assigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.

*coef_* is readonly property derived from *dual_coef_* and *support_vectors_*

intercept_ : array, shape = [n_classes-1]

Constants in decision function.

Methods

decision_function(X)	Distance of the samples X to the separating hyperplane.
fit(X[, y, sample_weight])	Detects the soft boundary of the set of samples X.
get_params([deep])	Get parameters for this estimator.
predict(X)	Perform regression on samples in X.
set_params(**params)	Set the parameters of this estimator.

static **__init__**(*kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *tol=0.001*, *nu=0.5*, *shrinking=True*, *cache_size=200*, *verbose=False*, *max_iter=-1*, *random_state=None*)

decision_function(*X*)

Distance of the samples X to the separating hyperplane.

ParametersX : array-like, shape = [n_samples, n_features]

For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train].

ReturnsX : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

static **fit**(*X*, *y=None*, *sample_weight=None*, ***params*)

Detects the soft boundary of the set of samples X.

ParametersX : {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.

Returnsself : object

Returns self.

Notes

If X is not a C-ordered contiguous array it is copied.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*X*)

Perform regression on samples in X.

For an one-class model, +1 or -1 is returned.

ParametersX : {array-like, sparse matrix}, shape (n_samples, n_features)

For kernel="precomputed", the expected shape of X is (n_samples_test, n_samples_train).

Returnsy_pred : array, shape (n_samples,)

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

Examples using `sklearn.svm.OneClassSVM`

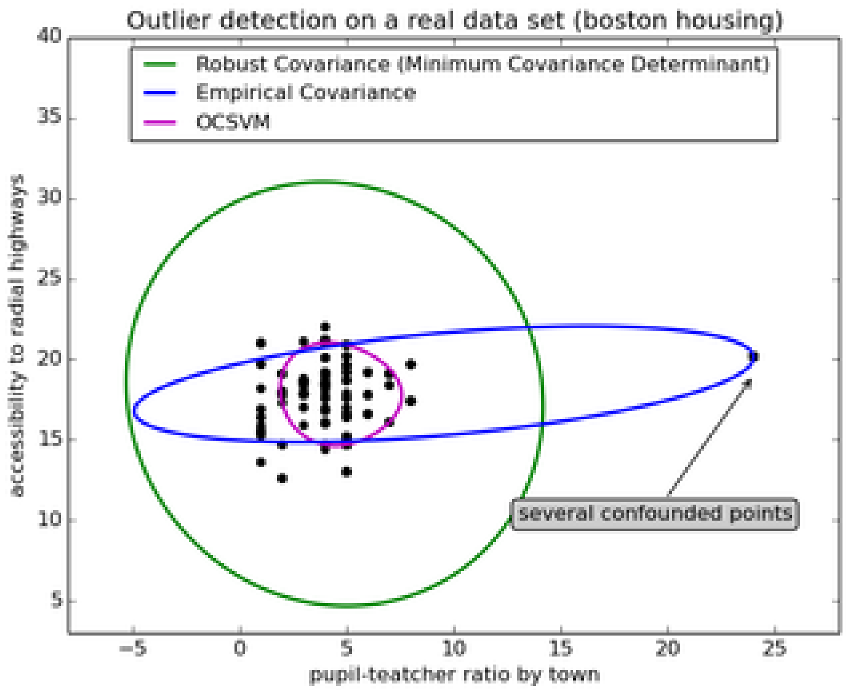


Figure 42.641: *Outlier detection on a real data set*

`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 g

`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.

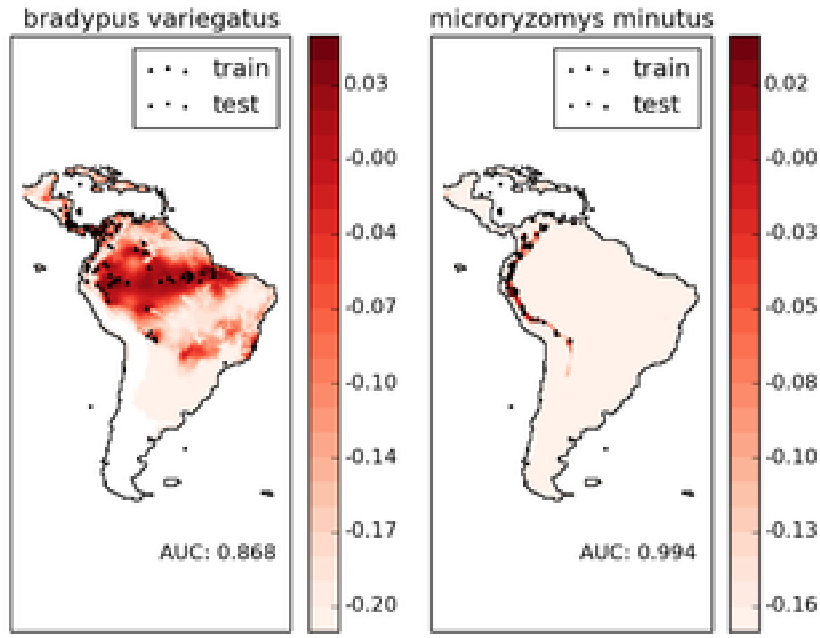


Figure 42.642: *Species distribution modeling*

Figure 42.643: *Libsvm GUI*

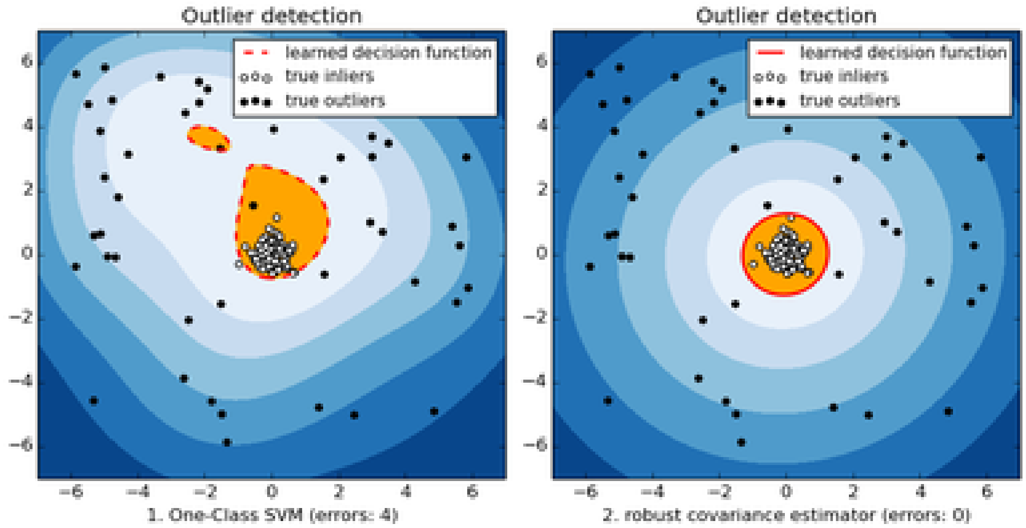


Figure 42.644: *Outlier detection with several methods.*

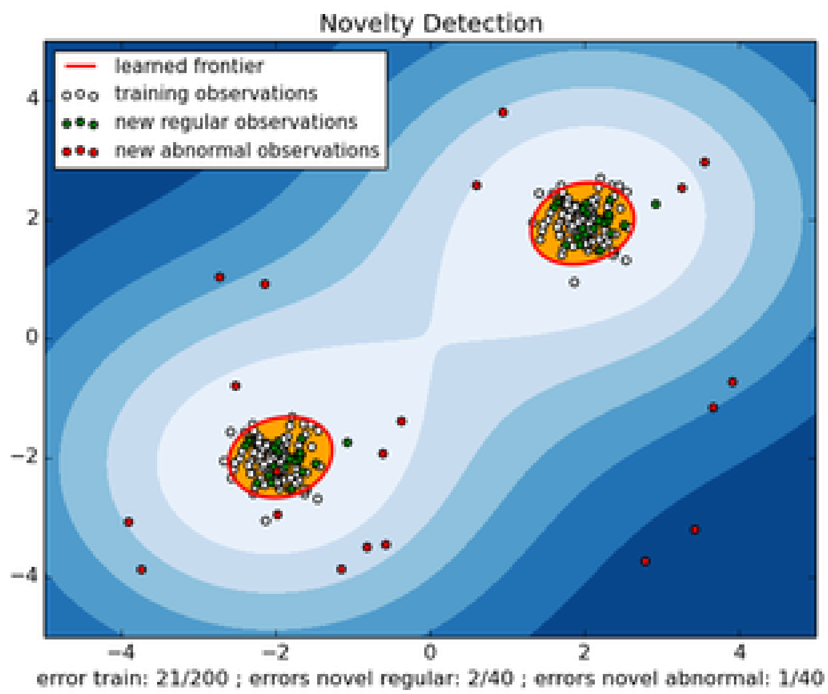


Figure 42.645: *One-class SVM with non-linear kernel (RBF)*

ParametersX : array-like or sparse matrix, 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. 'l2' is accepted as an alias for 'squared_hinge', for backward compatibility reasons, but should not be used in new code.

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.

Returnsl1_min_c: float :

minimum value for C

Examples using `sklearn.svm.l1_min_c`

42.34.2 Low-level methods

`svm.libsvm.fit`
`svm.libsvm.decision_function`
`svm.libsvm.predict`
`svm.libsvm.predict_proba`
`svm.libsvm.cross_validation`

`sklearn.svm.libsvm`.fit

`sklearn.svm.libsvm.`**`fit`**`()`

`sklearn.svm.libsvm`.decision_function

`sklearn.svm.libsvm.`**`decision_function`**`()`

`sklearn.svm.libsvm`.predict

`sklearn.svm.libsvm.`**`predict`**`()`

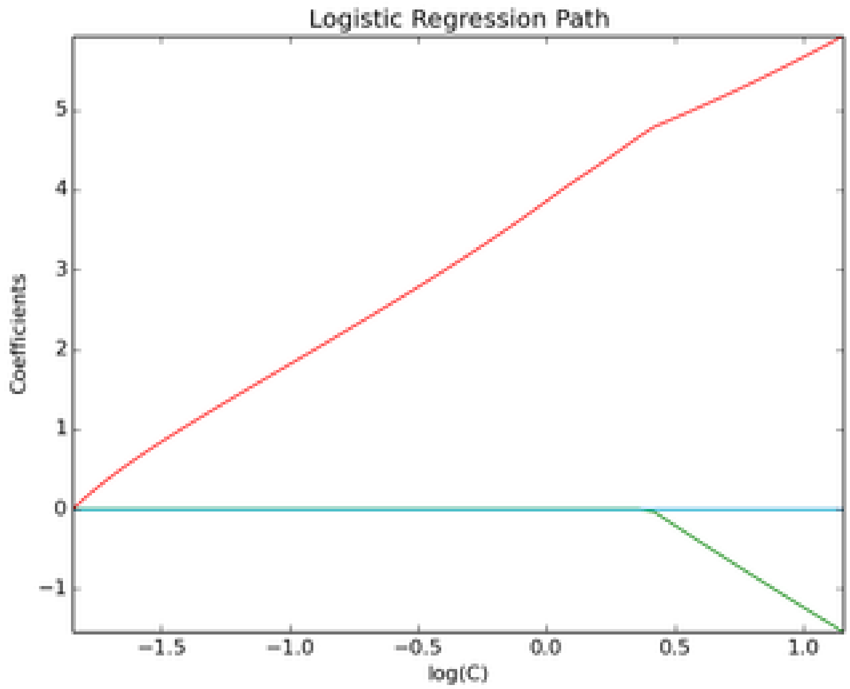


Figure 42.646: *Path with L1- Logistic Regression*

`sklearn.svm.libsvm`.predict_proba

`sklearn.svm.libsvm.`**`predict_proba`**`()`

`sklearn.svm.libsvm`.cross_validation

`sklearn.svm.libsvm.`**`cross_validation`**`()`

42.35 `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.

42.35.1 `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*, *class_weight=None*)

A decision tree classifier.

Parameterscriterion : 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 : int 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. Ignored if `max_leaf_nodes` is not None.

min_samples_split : int, optional (default=2)

The minimum number of samples required to split an internal node.

min_samples_leaf : int, optional (default=1)

The minimum number of samples required to be at a leaf node.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node.

max_leaf_nodes : int or None, optional (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. If not None then `max_depth` will be ignored.

class_weight : dict, list of dicts, "auto" 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.

The "auto" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data.

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.

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*.

Attributestree_ : Tree object

The underlying Tree object.

max_features_ : int,

The inferred value of max_features.

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 (for single output problems), or a list containing the number of classes for each output (for multi-output problems).

feature_importances_ : array of shape = [n_features]

The feature importances. The higher, the more important the feature. 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 [R195].

See also:

DecisionTreeRegressor

References

[R192], [R193], [R194], [R195]

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.cross_validation 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)
...
...
array([ 1.     ,  0.93...,  0.86...,  0.93...,  0.93...,
        0.93...,  0.93...,  1.     ,  0.93...,  1.      ])
```

Methods

fit(X, y[, sample_weight, check_input])	Build a decision tree from the training set (X, y).
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
predict(X[, check_input])	Predict class or regression value for X.
predict_log_proba(X)	Predict class log-probabilities of the input samples X.
predict_proba(X[, check_input])	Predict class probabilities of the input samples X.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.
transform(X[, threshold])	Reduce X to its most important features.

static __init__(*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*, *class_weight=None*)

fit(*X*, *y*, *sample_weight=None*, *check_input=True*)

Build a decision tree from the training set (X, y).

ParametersX : array-like or sparse matrix, 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, shape = [n_samples] or [n_samples, n_outputs]

The target values (class labels in classification, real numbers in regression). In the regression case, use `dtype=np.float64` and `order='C'` for maximum efficiency.

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.

check_input : boolean, (default=True)

Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returnsself : object

Returns self.

fit_transform(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*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.

ParametersX : 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`.

check_input : boolean, (default=True)

Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returnsy : array of shape = [n_samples] or [n_samples, n_outputs]

The predicted classes, or the predict values.

static **predict_log_proba**(*X*)

Predict class log-probabilities of the input samples X.

ParametersX : 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`.

Returnsp : array of shape = [n_samples, n_classes], or a 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_*.

static predict_proba (*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.

check_input[boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

ParametersX : 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`.

Returnsp : 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 (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Examples using `sklearn.tree.DecisionTreeClassifier`

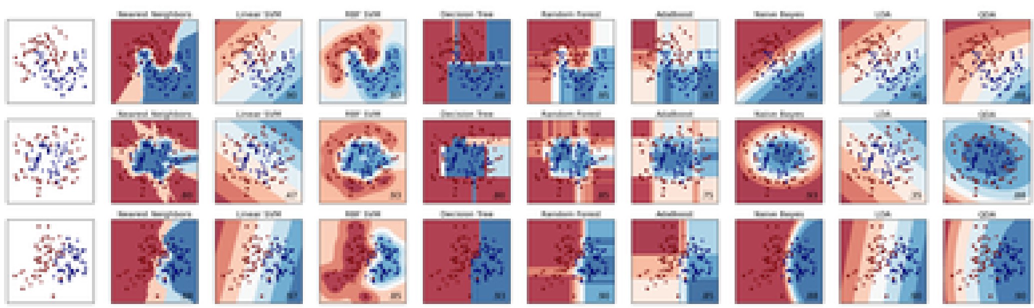


Figure 42.647: *Classifier comparison*

42.35.2 `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*)

A decision tree regressor.

Parameterscriterion : string, optional (default="mse")

The function to measure the quality of a split. The only supported criterion is "mse" for the mean squared error.

splitter : string, optional (default="best")

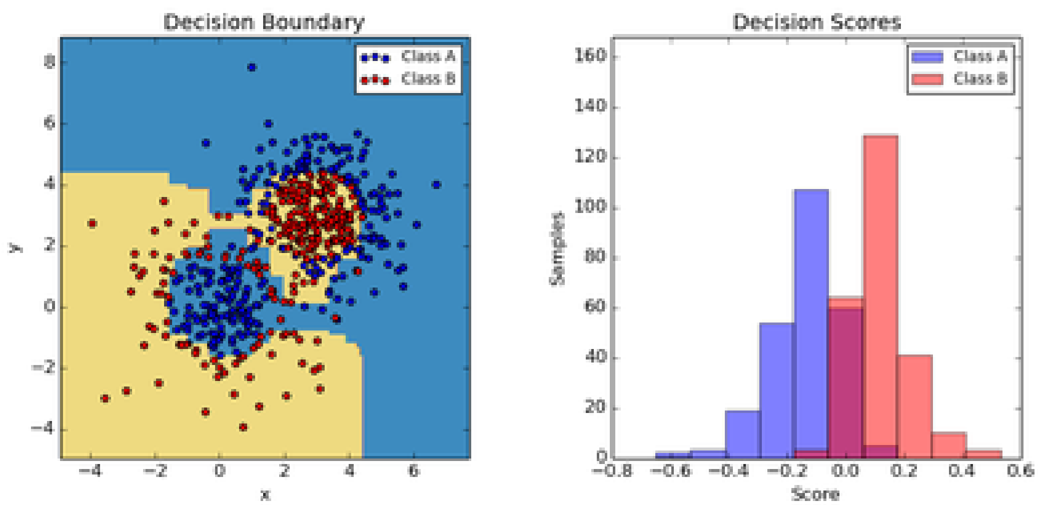


Figure 42.648: *Two-class AdaBoost*

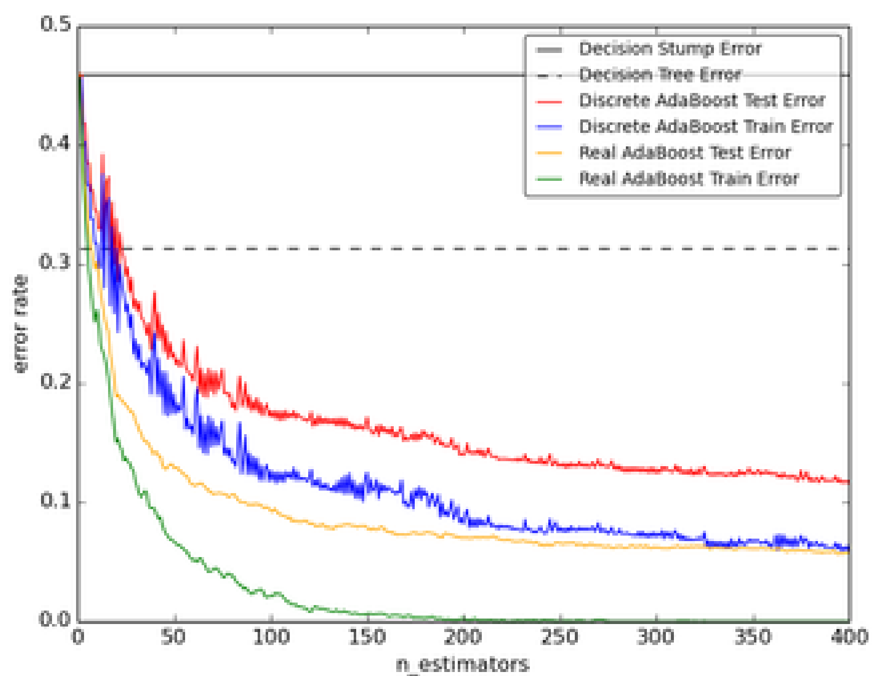


Figure 42.649: *Discrete versus Real AdaBoost*

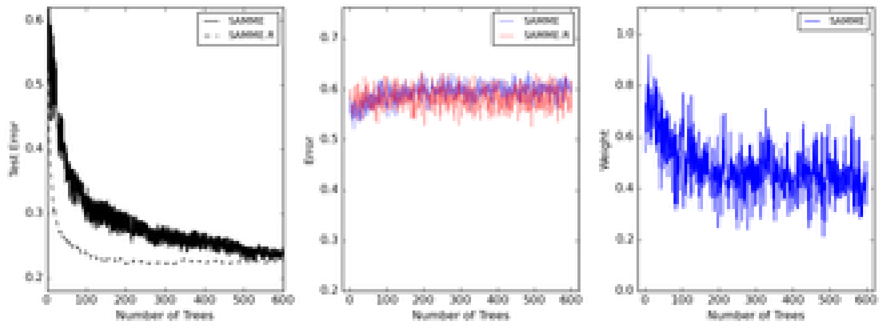


Figure 42.650: *Multi-class AdaBoosted Decision Trees*

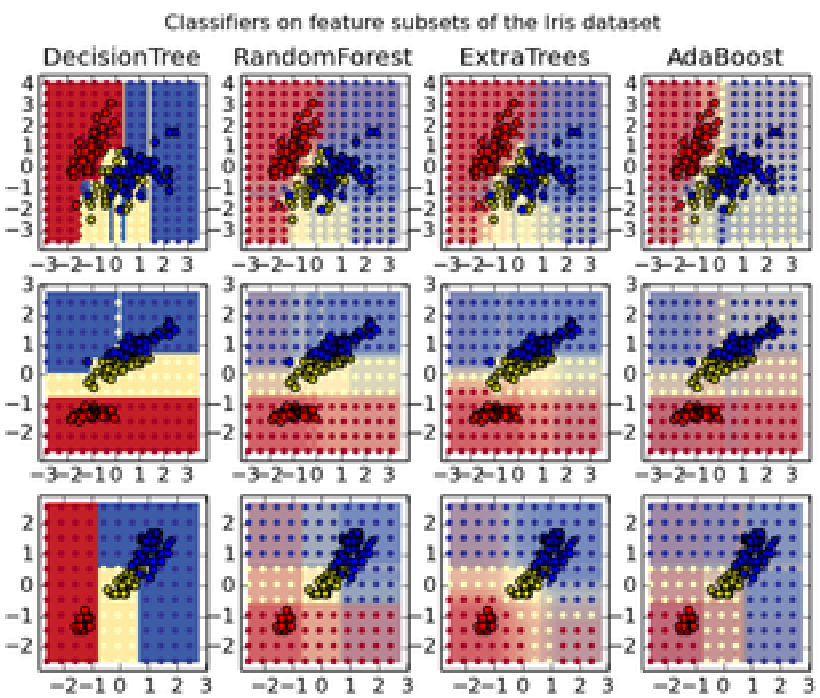


Figure 42.651: *Plot the decision surfaces of ensembles of trees on the iris dataset*

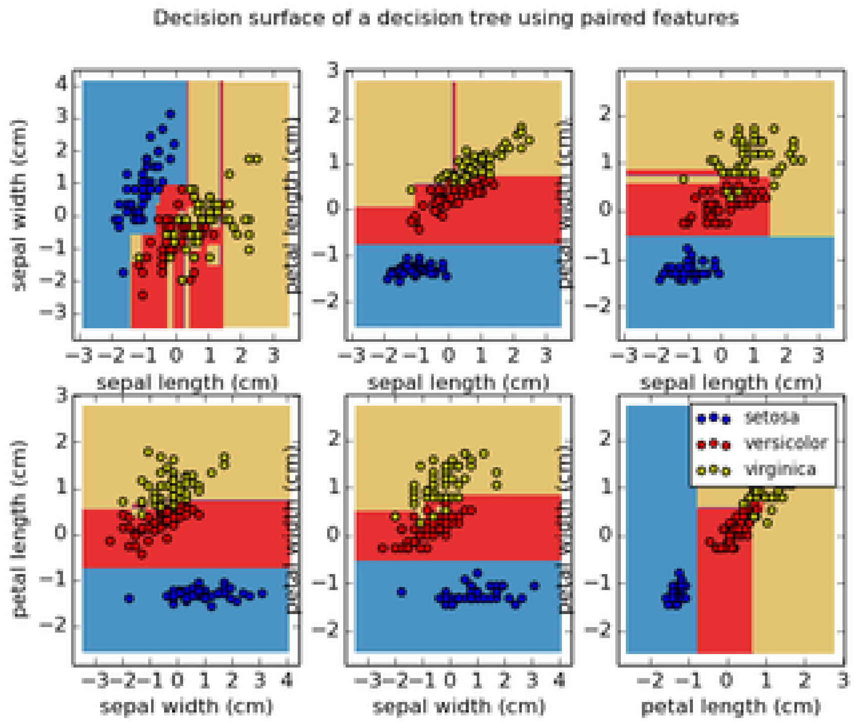


Figure 42.652: *Plot the decision surface of a decision tree on the iris dataset*

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_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 percentage 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_depth : int 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. Ignored if `max_leaf_nodes` is not None.

min_samples_split : int, optional (default=2)

The minimum number of samples required to split an internal node.

min_samples_leaf : int, optional (default=1)

The minimum number of samples required to be at a leaf node.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the input samples required to be at a leaf node.

max_leaf_nodes : int or None, optional (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. If not None then `max_depth` will be ignored.

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*.

Attributestree_ : Tree object

The underlying Tree object.

max_features_ : int,

The inferred value of max_features.

feature_importances_ : array of shape = [n_features]

The feature importances. The higher, the more important the feature. 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 [R199].

See also:

`DecisionTreeClassifier`

References

[R196], [R197], [R198], [R199]

Examples

```
>>> from sklearn.datasets import load_boston
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.tree import DecisionTreeRegressor
>>> boston = load_boston()
>>> regressor = DecisionTreeRegressor(random_state=0)
>>> cross_val_score(regressor, boston.data, boston.target, cv=10)
...
...
array([ 0.61..., 0.57..., -0.34..., 0.41..., 0.75...,
        0.07..., 0.29..., 0.33..., -1.42..., -1.77...])
```

Methods

`fit`(X, y[, sample_weight, check_input])	Build a decision tree from the training set (X, y).

Continued on next page

Table 42.233 – continued from previous page

fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
predict(X[, check_input])	Predict class or regression value for X.
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.
transform(X[, threshold])	Reduce X to its most important features.

static `__init__` (*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*)

`fit` (*X*, *y*, *sample_weight=None*, *check_input=True*)

Build a decision tree from the training set (X, y).

ParametersX : array-like or sparse matrix, 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, shape = [n_samples] or [n_samples, n_outputs]

The target values (class labels in classification, real numbers in regression). In the regression case, use `dtype=np.float64` and `order='C'` for maximum efficiency.

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.

check_input : boolean, (default=True)

Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returnsself : object

Returns self.

`fit_transform` (*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

`get_params` (*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict (*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.

ParametersX : 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`.

check_input : boolean, (default=True)

Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returnsy : array of shape = [n_samples] or [n_samples, n_outputs]

The predicted classes, or the predict values.

score (*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

Examples using `sklearn.tree.DecisionTreeRegressor`

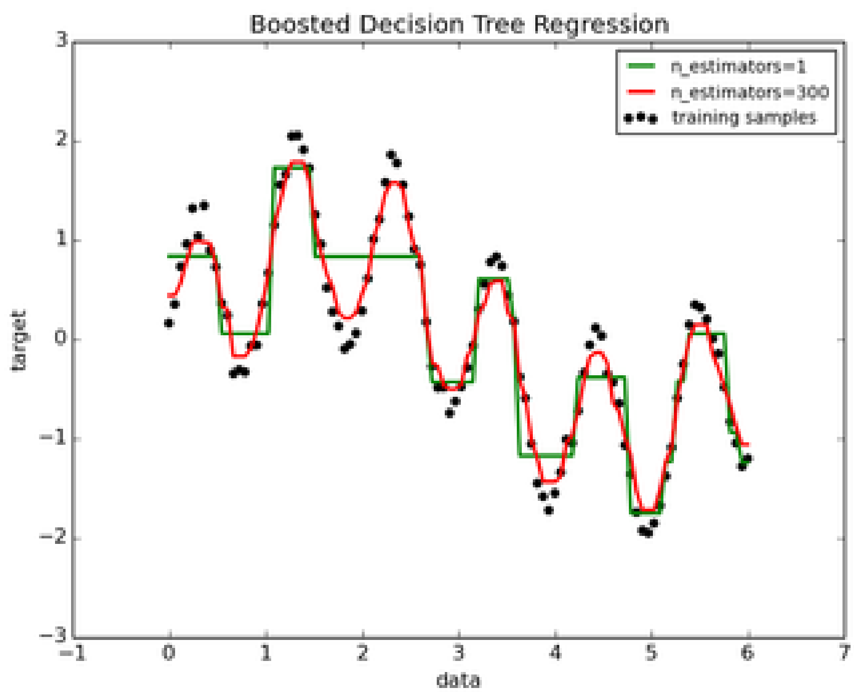


Figure 42.653: *Decision Tree Regression with AdaBoost*

42.35.3 `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*, *class_weight=None*)

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.

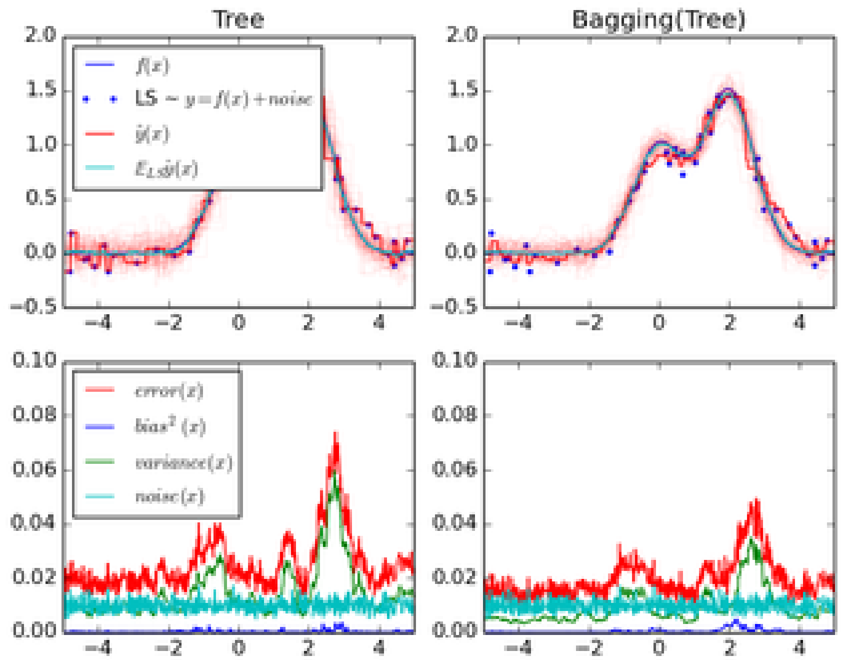


Figure 42.654: *Single estimator versus bagging: bias-variance decomposition*

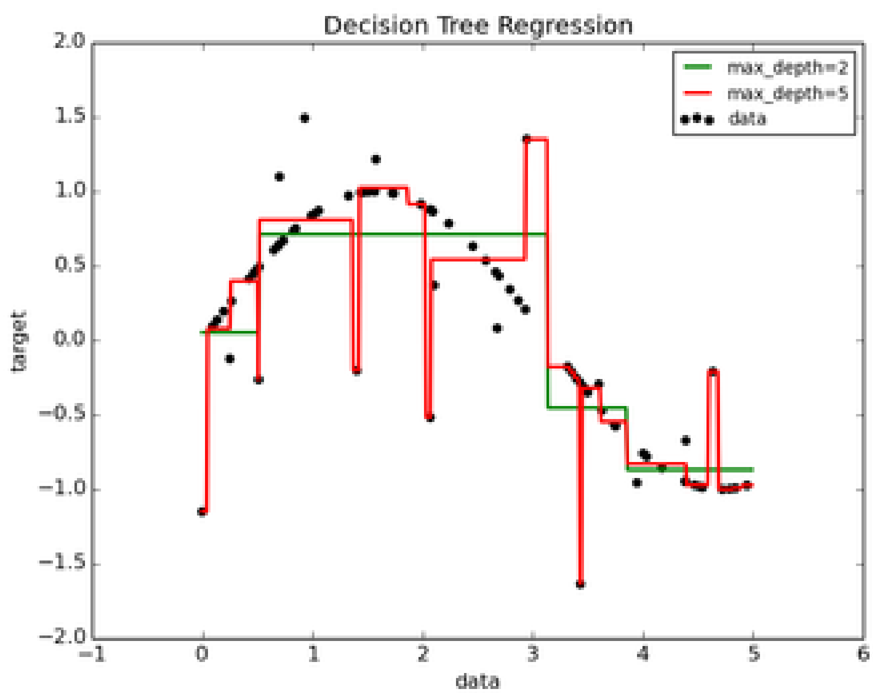


Figure 42.655: *Decision Tree Regression*

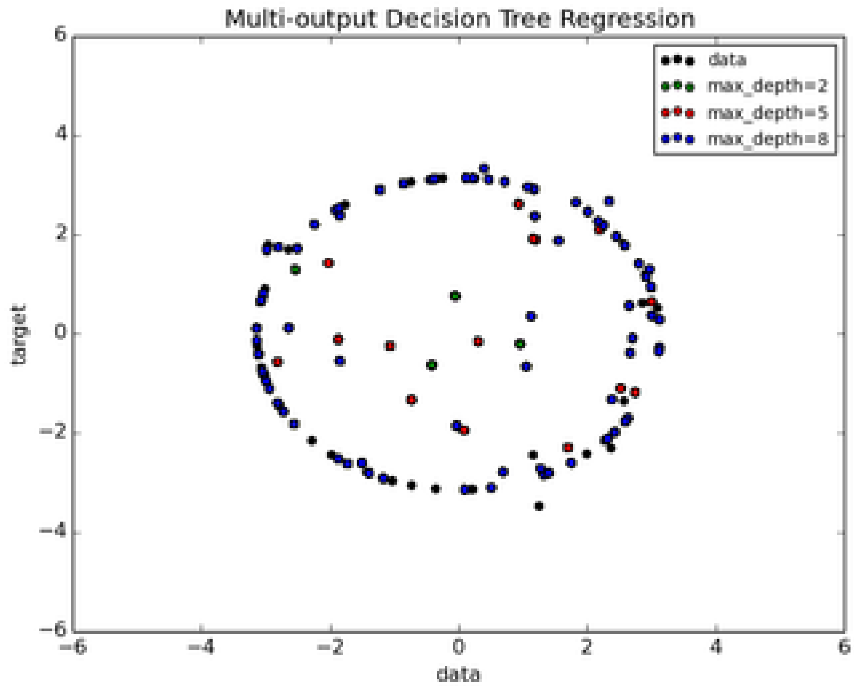


Figure 42.656: *Multi-output Decision Tree Regression*

See also:

`ExtraTreeRegressor`, `ExtraTreesClassifier`, `ExtraTreesRegressor`

References

[R200]

Methods

`fit`(X, y[, sample_weight, check_input])	Build a decision tree from the training set (X, y).
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X[, check_input])	Predict class or regression value for X.
`predict_log_proba`(X)	Predict class log-probabilities of the input samples X.
`predict_proba`(X[, check_input])	Predict class probabilities of the input samples X.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, threshold])	Reduce X to its most important features.

static `__init__`(*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*, *class_weight=None*)

`fit`(*X*, *y*, *sample_weight=None*, *check_input=True*)

Build a decision tree from the training set (X, y).

Parameters X : array-like or sparse matrix, 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, shape = [n_samples] or [n_samples, n_outputs]

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)
Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

predict(*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.

ParametersX : 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`.

check_input : boolean, (default=True)

Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returnsy : array of shape = [n_samples] or [n_samples, n_outputs]

The predicted classes, or the predict values.

predict_log_proba(*X*)
Predict class log-probabilities of the input samples X.

ParametersX : 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`.

Returnsp : array of shape = [n_samples, n_classes], or a 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(*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.

check_input[boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

ParametersX : 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`.

Returnsp : 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 (*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.

ParametersX : 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.

Returnsscore : float

Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform (*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

42.35.4 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*, *max_leaf_nodes=None*)

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.

See also:

ExtraTreeClassifier, ExtraTreesClassifier, ExtraTreesRegressor

References

[R201]

Methods

fit(X, y[, sample_weight, check_input])	Build a decision tree from the training set (X, y).
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
predict(X[, check_input])	Predict class or regression value for X.
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.
transform(X[, threshold])	Reduce X to its most important features.

static __init__(*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*, *max_leaf_nodes=None*)

fit(*X*, *y*, *sample_weight=None*, *check_input=True*)

Build a decision tree from the training set (X, y).

ParametersX : array-like or sparse matrix, 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, shape = [n_samples] or [n_samples, n_outputs]

The target values (class labels in classification, real numbers in regression). In the regression case, use `dtype=np.float64` and `order='C'` for maximum efficiency.

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.

check_input : boolean, (default=True)

Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returnsself : object

Returns self.

`fit_transform`(*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.

ParametersX : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

ReturnsX_new : numpy array of shape [n_samples, n_features_new]

Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parametersdeep: boolean, optional :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returnsparams : mapping of string to any

Parameter names mapped to their values.

`predict`(*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.

ParametersX : 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`.

check_input : boolean, (default=True)

Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returnsy : array of shape = [n_samples] or [n_samples, n_outputs]

The predicted classes, or the predict values.

`score`(*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 regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

ParametersX : 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.

Returnsscore : float

R^2 of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returnsself :

transform(*X*, *threshold=None*)

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

ParametersX : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold : string, float or None, 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 available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

ReturnsX_r : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

`tree.export_graphviz`(decision_tree[, ...])	Export a decision tree in DOT format.

42.35.5 `sklearn.tree`.export_graphviz

`sklearn.tree.`**export_graphviz**(*decision_tree*, *out_file='tree.dot'*, *feature_names=None*, *max_depth=None*)

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.

Parametersdecision_tree : decision tree classifier

The decision tree to be exported to GraphViz.

out_file : file object or string, optional (default="tree.dot")

Handle or name of the output file.

feature_names : list of strings, optional (default=None)

Names of each of the features.

max_depth : int, optional (default=None)

The maximum depth of the representation. If None, the tree is fully generated.

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,
...     out_file='tree.dot')
```

42.36 sklearn.utils: Utilities

The sklearn.utils module includes various utilities.

Developer guide: See the *Utilities for Developers* page for further details.

utils.check_random_state(seed)	Turn seed into a np.random.RandomState instance
utils.resample(*arrays, **options)	Resample arrays or sparse matrices in a consistent way
utils.shuffle(*arrays, **options)	Shuffle arrays or sparse matrices in a consistent way

42.36.1 sklearn.utils.check_random_state

sklearn.utils.**check_random_state**(*seed*)

Turn seed into a np.random.RandomState instance

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

42.36.2 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.

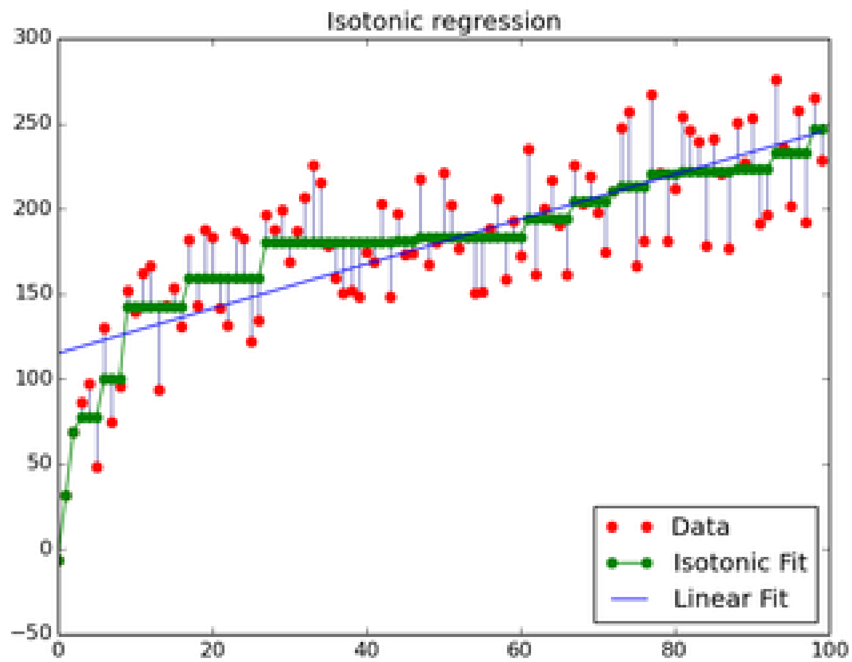


Figure 42.657: *Isotonic Regression*

Figure 42.658: *Face completion with a multi-output estimators*

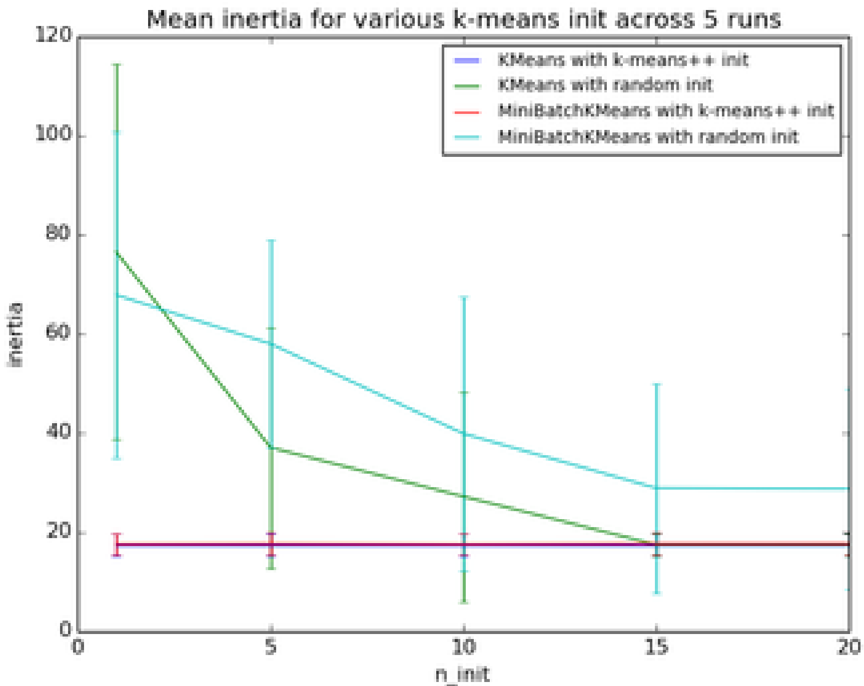


Figure 42.659: *Empirical evaluation of the impact of k-means initialization*

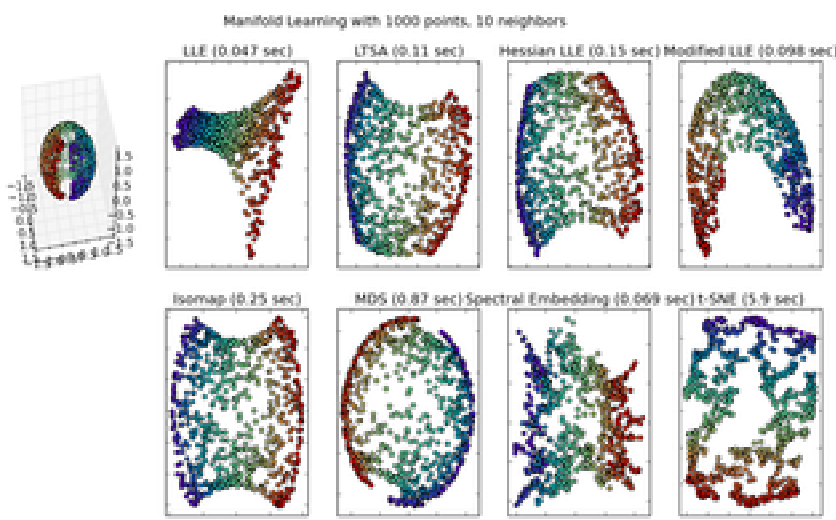


Figure 42.660: *Manifold Learning methods on a severed sphere*

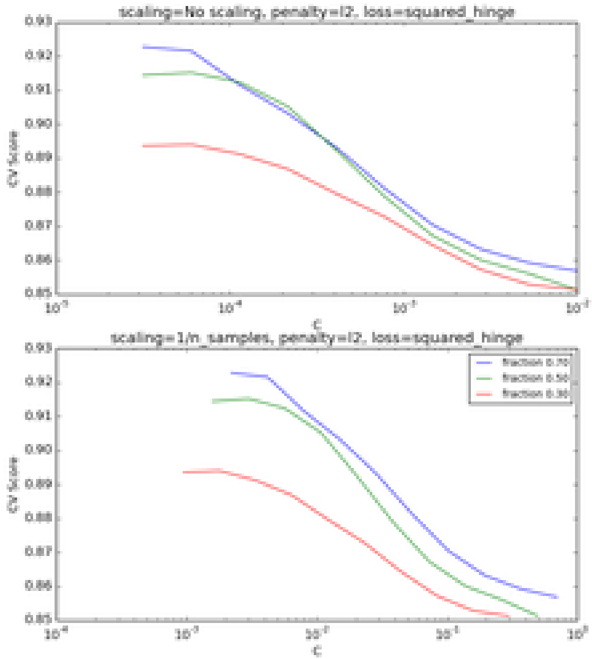


Figure 42.661: *Scaling the regularization parameter for SVCs*

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.

random_state : int or RandomState instance

Control the shuffling for reproducible behavior.

Returnsresampled_arrays : sequence of indexable data-structures

Sequence of resampled views of the collections. The original arrays are not impacted.

See also:

`sklearn.cross_validation.Bootstrap`, `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.]])

>>> y
array([0, 1, 0])

>>> resample(y, n_samples=2, random_state=0)
array([0, 1])
```

42.36.3 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.

random_state : int or RandomState instance

Control the shuffling for reproducible behavior.

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.

Returnsshuffled_arrays : sequence of indexable data-structures

Sequence of shuffled views of the collections. The original arrays are not impacted.

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`

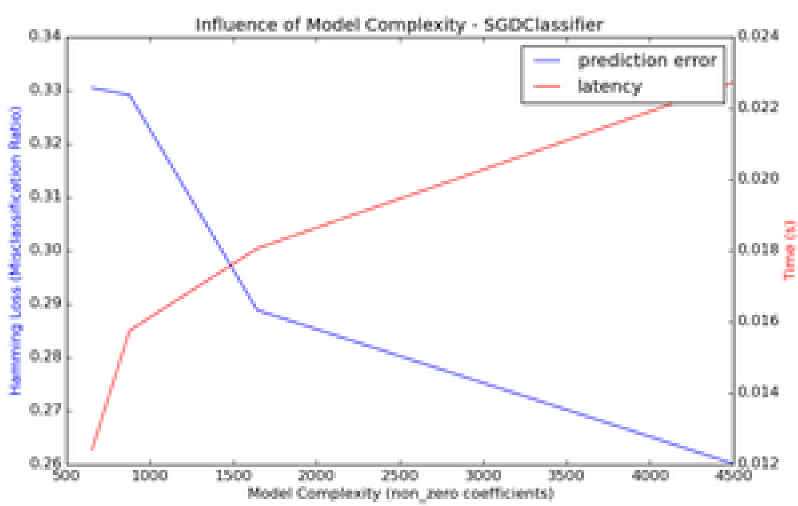


Figure 42.662: *Model Complexity Influence*

Figure 42.663: *Color Quantization using K-Means*

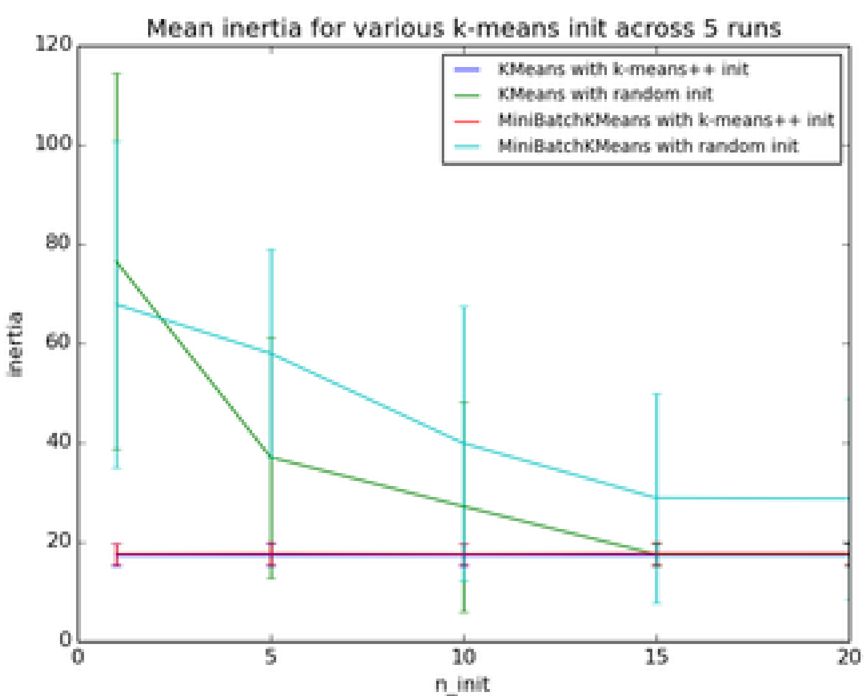


Figure 42.664: *Empirical evaluation of the impact of k-means initialization*

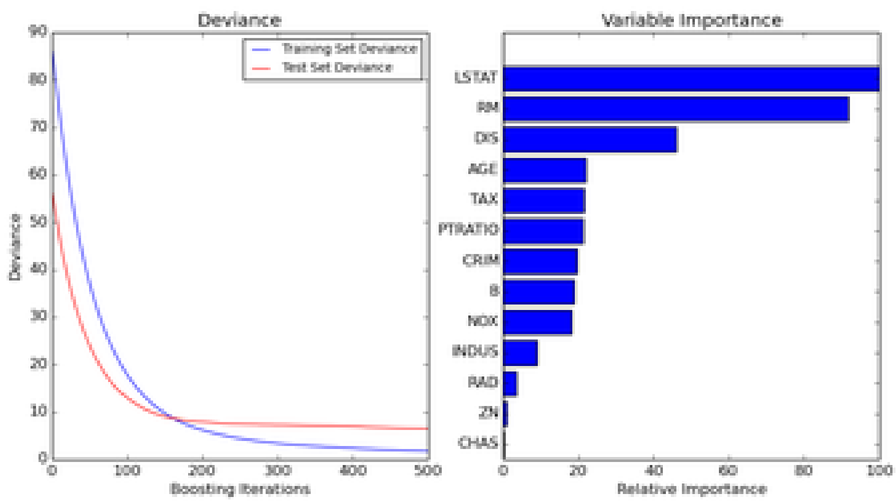


Figure 42.665: *Gradient Boosting regression*

CHAPTER FORTYTHREE

WHO IS USING SCIKIT-LEARN?

43.1 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

43.2 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 learing 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

43.3 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

43.4 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

43.5 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

43.6 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

43.7 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

43.8 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

43.9 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

43.10 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

43.11 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

43.12 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

43.13 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

43.14 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.

43.15 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

43.16 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

43.17 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

43.18 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

43.19 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.

CHAPTER

FORTYFOUR

CONTRIBUTING

This project is a community effort, and everyone is welcome to contribute.

The project is hosted on http://github.com/scikit-learn/scikit-learn

44.1 Submitting a bug report

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 links to pull requests.

44.2 Retrieving the latest code

We use Git for version control and GitHub for hosting our main repository.

You can check out the latest sources with the command:

```
git clone git://github.com/scikit-learn/scikit-learn.git
```

or if you have write privileges:

```
git clone git@github.com:scikit-learn/scikit-learn.git
```

If you run the development version, it is cumbersome to reinstall the package each time you update the sources. It is thus preferred that you add the scikit-learn directory to your `PYTHONPATH` and build the extension in place:

```
python setup.py build_ext --inplace
```

Another option is to use the `develop` option if you change your code a lot and do not want to have to reinstall every time. This basically builds the extension in place and creates a link to the development directory (see the setuptool docs):

```
python setup.py develop
```

Note: if you decide to do that you have to rerun:

```
python setup.py build_ext --inplace
```

every time the source code of a compiled extension is changed (for instance when switching branches or pulling changes from upstream).

On Unix-like systems, you can simply type `make` in the top-level folder to build in-place and launch all the tests. Have a look at the `Makefile` for additional utilities.

44.3 Contributing code

Note: To avoid duplicating work, it is highly advised that you contact the developers on the mailing list before starting work on a non-trivial feature.

https://lists.sourceforge.net/lists/listinfo/scikit-learn-general

44.3.1 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):

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 server.
3. Clone this copy to your local disk:

   ```
   $ git clone git@github.com:YourLogin/scikit-learn.git
   ```

4. Create a branch to hold your changes:

   ```
   $ git checkout -b my-feature
   ```

   and start making changes. Never work in the `master` branch!
5. Work on this copy, on your computer, using Git to do the version control. When you're done editing, do:

   ```
   $ git add modified_files
   $ git commit
   ```

   to record your changes in Git, then push them to GitHub with:

   ```
   $ git push -u origin my-feature
   ```

Finally, go to the web page of the your fork of the scikit-learn repo, and click 'Pull request' to send your changes to the maintainers for review. request. This will send an email to the committers, but might also send an email to the mailing list in order to get more visibility.

Note: In the above setup, your `origin` remote repository points to YourLogin/scikit-learn.git. If you wish to fetch/merge from the main repository instead of your forked one, you will need to add another remote to use instead of `origin`. If we choose the name `upstream` for it, the command will be:

```
$ git remote add upstream https://github.com/scikit-learn/scikit-learn.git
```

(If any of the above seems like magic to you, then look up the Git documentation on the web.)

It is recommended to check that your contribution complies with the following rules before submitting a pull request:

- Follow the coding-guidelines (see below).
- When applicable, use the Validation tools and other code in the `sklearn.utils` submodule. A list of utility routines available for developers can be found in the *Utilities for Developers* page.
- All public methods should have informative docstrings with sample usage presented as doctests when appropriate.

- All other tests pass when everything is rebuilt from scratch. On Unix-like systems, check with (from the toplevel source folder):

```
$ make
```

- When adding additional functionality, provide at least one example script in the `examples/` folder. Have a look at other examples 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.
- At least one paragraph of narrative documentation with links to references in the literature (with PDF links when possible) and the example. For more details on writing and building the documentation, see the *Documentation* section.

You can also check for common programming errors with the following tools:

- Code with a good unittest coverage (at least 90%, better 100%), check with:

```
$ pip install nose coverage
$ nosetests --with-coverage path/to/tests_for_package
```

  see also *Testing and improving test coverage*

- No pyflakes warnings, check with:

```
$ pip install pyflakes
$ pyflakes path/to/module.py
```

- No PEP8 warnings, check with:

```
$ pip install pep8
$ pep8 path/to/module.py
```

- AutoPEP8 can help you fix some of the easy redundant errors:

```
$ pip install autopep8
$ autopep8 path/to/pep8.py
```

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 wiki).

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.

44.3.2 Easy Issues

A great way to start contributing to scikit-learn is to pick an item from the list of Easy issues in the issue tracker. Resolving these issues allow you to start contributing to the project without much prior knowledge. 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.

44.3.3 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 html` from the doc/ directory. Alternatively, `make html-noplot` can be used to quickly generate the documentation without the example gallery. The resulting HTML files will be placed in _build/html/ and are viewable in a web browser. See the README file in the doc/ directory for more information.

For building the documentation, you will need sphinx, matplotlib and pillow.

When you are 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-waiving 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.
  ```

- For unwritten formatting rules, try to follow existing good works:
  - For "References" in docstrings, see the Silhouette Coefficient (`sklearn.metrics.silhouette_score`).

> **Warning: Sphinx version**
> While we do our best to have the documentation build under as many version of Sphinx as possible, the different versions tend to behave slightly differently. To get the best results, you should use version 1.0.

44.3.4 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 nose package. The tests are functions appropriately names, located in *tests* subdirectories, that check the validity of the algorithms and the different options of the code.

The full scikit-learn tests can be run using 'make' in the root folder. Alternatively, running 'nosetests' in a folder will run all the tests of the corresponding subpackages.

We expect code coverage of new features to be at least around 90%.

Note: Workflow to improve test coverage

To test code coverage, you need to install the coverage package in addition to nose.

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.

44.3.5 Developers web site

More information can be found on the developer's wiki.

44.3.6 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 two other tags to help new contributors:

Easy This issue can be tackled by anyone, no experience needed. Ask for help if the formulation is unclear.

Moderate Might need some knowledge of machine learning or the package, but is still approachable for someone new to the project.

44.4 Other ways to contribute

Code is not the only way to contribute to scikit-learn. For instance, documentation is also a very important part of the project and often doesn't get as much attention as it deserves. If you find a typo in the documentation, or have made improvements, do not hesitate to send an email to the mailing list or submit a GitHub pull request. Full documentation can be found under the doc/ directory.

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 say "I use it":

44.5 Coding guidelines

The following are some guidelines on how new code should be written. 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.

44.5.1 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.

44.5.2 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 array2d, 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.

    Returns
    -------
    x : numpy array, shape = (n_features,)
        A random point selected from X
```

```
    """
    X = array2d(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__`.

44.5.3 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

@deprecated("Function 'zero_one' has been renamed to "
            "'zero_one_loss' 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. E.g., renaming an attribute `labels_` to `classes_` can be done as:

```
@property
@deprecated("Attribute labels_ is deprecated and "
            "will be removed in 0.15. Use 'classes_' instead")
def labels_(self):
    return self.classes_
```

If a parameter has to be deprecated, use `DeprecationWarning` appropriately. In following example, k is deprecated and renamed to n_clusters:

```
import warnings

def example_function(n_clusters=8, k=None):
    if k is not None:
        warnings.warn("'k' was renamed to n_clusters and will "
                      "be removed in 0.15.",
                      DeprecationWarning)
        n_clusters = k
```

44.5.4 Python 3.x support

All scikit-learn code should work unchanged in both Python 2.[67] and 3.2 or newer. Since Python 3.x is not backwards compatible, that may require changes to code and it certainly requires testing on both 2.6 or 2.7, and 3.2 or newer.

For most numerical algorithms, Python 3.x support is easy: just remember that `print` is a function and integer division is written `//`. String handling has been overhauled, though, as have parts of the Python standard library. The six package helps with cross-compatibility and is included in scikit-learn as `sklearn.externals.six`.

44.6 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.

44.6.1 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 = obj.fit(data, targets)
```

or:

```
estimator = obj.fit(data)
```

Predictor For supervised learning, or some unsupervised problems, implements:

```
prediction = obj.predict(data)
```

Classification algorithms usually also offer a way to quantify certainty of a prediction, either using `decision_function` or `predict_proba`:

```
probability = obj.predict_proba(data)
```

Transformer For filtering or modifying the data, in a supervised or unsupervised way, implements:

```
new_data = obj.transform(data)
```

When fitting and transforming can be performed much more efficiently together than separately, implements:

```
new_data = obj.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 = obj.score(data)
```

44.6.2 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, with shape = [N, D], where N is the number of samples and D is the number of features.
y	array, with shape = [N], where N is the number of 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).

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 last-mentioned attributes are expected to be overridden when you call `fit` a second time without taking any previous value into account: **fit should be idempotent**.

Optional Arguments

In iterative algorithms, the number of iterations should be specified by an integer called `n_iter`.

44.7 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 sklearn, there are several internals of scikit-learn that you should be aware of in addition to the sklearn API outlined above.

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 assessment and selection tools such as `grid_search.GridSearchCV`.

For this to work, you need to implement the following interface. If a dependency on scikit-learn is okay for your code, you can prevent a lot of boilerplate code by deriving a class from `BaseEstimator` and optionally the mixin classes in `sklearn.base`. E.g., here's a custom classifier:

```
>>> import numpy as np
>>> from sklearn.base import BaseEstimator, ClassifierMixin
>>> class MajorityClassifier(BaseEstimator, ClassifierMixin):
...     """Predicts the majority class of its training data."""
...     def __init__(self):
...         pass
...     def fit(self, X, y):
...         self.classes_, indices = np.unique(["foo", "bar", "foo"],
...                                            return_inverse=True)
...         self.majority_ = np.argmax(np.bincount(indices))
...         return self
...     def predict(self, X):
...         return np.repeat(self.classes_[self.majority_], len(X))
```

44.7.1 get_params and set_params

All sklearn estimator 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():
        self.setattr(parameter, value)
    return self
```

44.7.2 Parameters and init

As `grid_search.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.

44.7.3 Cloning

For using `grid_search.GridSearch` or any functionality of the `cross_validation` 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`.

44.7.4 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`.

44.7.5 Working notes

For unresolved issues, TODOs, and remarks on ongoing work, developers are advised to maintain notes on the GitHub wiki.

44.7.6 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.

CHAPTER

FORTYFIVE

DEVELOPERS' TIPS FOR DEBUGGING

45.1 Memory errors: debugging 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.

CHAPTER

FORTYSIX

MAINTAINER / CORE-DEVELOPER INFORMATION

46.1 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 -ns 0.998..
     ```

   - edit the doc/conf.py to increase the version number
   - edit the doc/themes/scikit-learn/layout.html to change the 'News' entry of the front page.
2. Update the version number in sklearn/__init__.py, the __version__ variable
3. Create the tag and push it:

   ```
   $ git tag 0.999
   ```

   ```
   $ git push origin --tags
   ```

4. create tarballs:
   - Wipe clean your repo:

     ```
     $ git clean -xfd
     ```

   - Register and upload on PyPI:

     ```
     $ python setup.py sdist register upload
     ```

   - Upload manually the tarball on SourceForge: https://sourceforge.net/projects/scikit-learn/files/
5. Push the documentation to the website (see README in doc folder)
6. Build binaries for windows and push them to PyPI:

   ```
   $ python setup.py bdist_wininst upload
   ```

   And upload them also to sourceforge

CHAPTER

FORTYSEVEN

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 sample algorithmic trick: warm restarts for cross validation* gives an example of such a trick.

47.1 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: http://wiki.scipy.org/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, include the generated C source code alongside with the Cython source code. The goal is to make it possible to install the scikit on any machine with Python, Numpy, Scipy and C/C++ compiler.

47.2 Fast matrix multiplications

Matrix multiplications (matrix-matrix and matrix-vector) are usually handled using the NumPy function `np.dot`, but in versions of NumPy before 1.7.2 this function is suboptimal when the inputs are not both in the C (row-major) layout; in that case, the inputs may be implicitly copied to obtain the right layout. This obviously consumes memory and takes time.

The function `fast_dot` in `sklearn.utils.extmath` offers a fast replacement for `np.dot` that prevents copies from being made in some cases. In all other cases, it dispatches to `np.dot` and when the NumPy version is new enough, it is in fact an alias for that function, making it a drop-in replacement. Example usage of `fast_dot`:

```
>>> import numpy as np
>>> from sklearn.utils.extmath import fast_dot
>>> X = np.random.random_sample([2, 10])
>>> np.allclose(np.dot(X, X.T), fast_dot(X, X.T))
True
```

This function operates optimally on 2-dimensional arrays, both of the same dtype, which should be either single or double precision float. If these requirements aren't met or the BLAS package is not available, the call is silently dispatched to `numpy.dot`. If you want to be sure when the original `numpy.dot` has been invoked in a situation where it is suboptimal, you can activate the related warning:

```
>>> import warnings
>>> from sklearn.utils.validation import NonBLASDotWarning
>>> warnings.simplefilter('always', NonBLASDotWarning)
```

47.3 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 the scikit. 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().data
```

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 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

   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 cumulated time of the module. In order to better understand the profile of this specific function, let us install `line-prof` and wire it to IPython:

```
$ pip install line-profiler
```

- **Under IPython <= 0.10**, edit `~/.ipython/ipy_user_conf.py` and ensure the following lines are present:

```
import IPython.ipapi
ip = IPython.ipapi.get()
```

Towards the end of the file, define the `%lprun` magic:

```
import line_profiler
ip.expose_magic('lprun', line_profiler.magic_lprun)
```

- **Under IPython 0.11+**, first create a configuration profile:

  ```
  $ ipython profile create
  ```

  Then create a file named `~/.ipython/extensions/line_profiler_ext.py` with the following content:

  ```
  import line_profiler

  def load_ipython_extension(ip):
      ip.define_magic('lprun', line_profiler.magic_lprun)
  ```

  Then register it in `~/.ipython/profile_default/ipython_config.py`:

  ```
  c.TerminalIPythonApp.extensions = [
      'line_profiler_ext',
  ]
  c.InteractiveShellApp.extensions = [
      'line_profiler_ext',
  ]
  ```

  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.nmf import _nls_subproblem, NMF

In [3]: X = load_digits().data

In [4]: %lprun -f _nls_subproblem NMF(n_components=16, tol=1e-2).fit(X)
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
   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 xrange(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
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 xrange(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.

47.4 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 <= 0.10**, edit `~/.ipython/ipy_user_conf.py` and ensure the following lines are present:

  ```
  import IPython.ipapi
  ip = IPython.ipapi.get()
  ```

  Towards the end of the file, define the `%memit` and `%mprun` magics:

  ```
  import memory_profiler
  ip.expose_magic('memit', memory_profiler.magic_memit)
  ip.expose_magic('mprun', memory_profiler.magic_mprun)
  ```

- **Under IPython 0.11+**, first create a configuration profile:

  ```
  $ ipython profile create
  ```

  Then create a file named `~/.ipython/extensions/memory_profiler_ext.py` with the following content:

  ```
  import memory_profiler

  def load_ipython_extension(ip):
      ip.define_magic('memit', memory_profiler.magic_memit)
      ip.define_magic('mprun', memory_profiler.magic_mprun)
  ```

  Then register it in `~/.ipython/profile_default/ipython_config.py`:

```
c.TerminalIPythonApp.extensions = [
    'memory_profiler_ext',
]
c.InteractiveShellApp.extensions = [
    'memory_profiler_ext',
]
```

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?`.

47.5 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...

- http://www.euroscipy.org/file/3696?vid=download
- http://conference.scipy.org/proceedings/SciPy2009/paper_1/
- http://conference.scipy.org/proceedings/SciPy2009/paper_2/

47.6 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.

47.6.1 Using yep and google-perftools

Easy profiling without special compilation options use yep:

- http://pypi.python.org/pypi/yep
- http://fseoane.net/blog/2011/a-profiler-for-python-extensions/

Note: google-perftools provides a nice 'line by line' report mode that can be triggered with the `--lines` option. However this does not seem to work correctly at the time of writing. This issue can be tracked on the project issue tracker.

47.6.2 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.

47.6.3 Using valgrind / callgrind / kcachegrind

TODO

47.7 Multi-core parallelism using `joblib.Parallel`

TODO: give a simple teaser example here.

Checkout the official joblib documentation:

- http://packages.python.org/joblib/

47.8 A sample algorithmic trick: warm restarts for cross validation

TODO: demonstrate the warm restart tricks for cross validation of linear regression with Coordinate Descent.

CHAPTER

FORTYEIGHT

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.

48.1 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`: convert input to 2d array, raise error on sparse matrices. Allowed sparse matrix formats can be given optionally, as well as allowing 1d or nd 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.
- `warn_if_not_float`: Warn if input is not a floating-point value. the input `X` is assumed to have `X.dtype`.

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])
```

48.2 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.least_angle.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.norm`: computes Euclidean (L2) vector norm by directly calling the BLAS `nrm2` function. This is more stable than `scipy.linalg.norm`. See Fabian's blog post for a discussion.
- `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.logsumexp`: compute the sum of X assuming X is in the log domain. This is equivalent to calling `np.log(np.sum(np.exp(X)))`, but is robust to overflow/underflow errors. Note that there is similar functionality in `np.logaddexp.reduce`, but because of the pairwise nature of this routine, it is slower for large arrays. Scipy has a similar routine in `scipy.misc.logsumexp` (In scipy versions < 0.10, this is found in `scipy.maxentropy.logsumexp`), but the scipy version does not accept an `axis` keyword.
- `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`.

48.3 Efficient Random Sampling

- `random.sample_without_replacement`: implements efficient algorithms for sampling `n_samples` integers from a population of size `n_population` without replacement.

48.4 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.k_means_.KMeans`.
- `sparsefuncs.inplace_csr_row_normalize_l1` and `sparsefuncs.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`.

48.5 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.graph_laplacian`: (used in `sklearn.cluster.spectral.spectral_embedding`) Return the Laplacian of a given graph. There is specialized code for both dense and sparse connectivity matrices.
- `graph_shortest_path.graph_shortest_path`: (used in :class:`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`.

48.6 Backports

- `fixes.expit`: Logistic sigmoid function. Replacement for SciPy 0.10's `scipy.special.expit`.
- `sparsetools.connected_components` (backported from `scipy.sparse.connected_components` in scipy 0.12). Used in `sklearn.cluster.hierarchical`, as well as in tests for `sklearn.feature_extraction`.
- `fixes.isclose` (backported from `numpy.isclose` in numpy 1.8.1). In versions before 1.7, this function was not available in numpy. Used in `sklearn.metrics`.

48.6.1 ARPACK

- `arpack.eigs` (backported from `scipy.sparse.linalg.eigs` in scipy 0.10) Sparse non-symmetric eigenvalue decomposition using the Arnoldi method. A limited version of `eigs` is available in earlier scipy versions.
- `arpack.eigsh` (backported from `scipy.sparse.linalg.eigsh` in scipy 0.10) Sparse non-symmetric eigenvalue decomposition using the Arnoldi method. A limited version of `eigsh` is available in earlier scipy versions.
- `arpack.svds` (backported from `scipy.sparse.linalg.svds` in scipy 0.10) Sparse non-symmetric eigenvalue decomposition using the Arnoldi method. A limited version of `svds` is available in earlier scipy versions.

48.6.2 Benchmarking

- `bench.total_seconds` (back-ported from `timedelta.total_seconds` in Python 2.7). Used in `benchmarks/bench_glm.py`.

48.7 Testing Functions

- `testing.assert_in`, `testing.assert_not_in`: Assertions for container membership. Designed for forward compatibility with Nose 1.0.
- `testing.assert_raise_message`: Assertions for checking the error raise message.
- `testing.mock_mldata_urlopen`: Mocks the urlopen function to fake requests to mldata.org. Used in tests of `sklearn.datasets`.
- `testing.all_estimators` : returns a list of all estimators in sklearn to test for consistent behavior and interfaces.

48.8 Multiclass and multilabel utility function

- `multiclass.is_multilabel`: Helper function to check if the task is a multi-label classification one.
- `multiclass.is_label_indicator_matrix`: Helper function to check if a classification output is in label indicator matrix format.
- `multiclass.unique_labels`: Helper function to extract an ordered array of unique labels from different formats of target.

48.9 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.

48.10 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.

48.11 Warnings and Exceptions

- `deprecated`: Decorator to mark a function or class as deprecated.
- `ConvergenceWarning`: Custom warning to catch convergence problems. Used in `sklearn.covariance.graph_lasso`.

CHAPTER

FORTYNINE

INSTALLING SCIKIT-LEARN

There are different ways to get scikit-learn installed:

- Install the version of scikit-learn provided by your *operating system or Python distribution*. This is the quickest option for those who have operating systems that distribute scikit-learn.
- *Install an official release*. This is the best approach for users who want a stable version number and aren't concerned about running a slightly older version of scikit-learn.
- *Install the latest development version*. This is best for users who want the latest-and-greatest features and aren't afraid of running brand-new code.

Note: If you wish to contribute to the project, it's recommended you *install the latest development version*.

49.1 Installing an official release

Scikit-learn requires:

- Python (>= 2.6 or >= 3.3),
- NumPy (>= 1.6.1),
- SciPy (>= 0.9).

49.1.1 Windows

First you need to install numpy and scipy from their own official installers.

Wheel packages (.whl files) for scikit-learn from PyPI can be installed with the pip utility. Open a console and type the following to install or upgrade scikit-learn to the latest stable release:

```
pip install -U scikit-learn
```

If there are no binary packages matching your Python version you might to try to install scikit-learn and its dependencies from Christoph Gohlke Unofficial Windows installers or from a *Python distribution* instead.

49.1.2 Mac OSX

Scikit-learn and its dependencies are all available as wheel packages for OSX:

```
pip install -U numpy scipy scikit-learn
```

49.1.3 Linux

At this time scikit-learn does not provide official binary packages for Linux so you have to build from source.

Installing build dependencies

Installing from source requires you to have installed the scikit-learn runtime dependencies, Python development headers and a working C/C++ compiler. Under Debian-based operating systems, which include Ubuntu, if you have Python 2 you can install all these requirements by issuing:

```
sudo apt-get install build-essential python-dev python-setuptools \
                     python-numpy python-scipy \
                     libatlas-dev libatlas3gf-base
```

If you have Python 3:

```
sudo apt-get install build-essential python3-dev python3-setuptools \
                     python3-numpy python3-scipy \
                     libatlas-dev libatlas3gf-base
```

On recent Debian and Ubuntu (e.g. Ubuntu 13.04 or later) make sure that ATLAS is used to provide the implementation of the BLAS and LAPACK linear algebra routines:

```
sudo update-alternatives --set libblas.so.3 \
    /usr/lib/atlas-base/atlas/libblas.so.3
sudo update-alternatives --set liblapack.so.3 \
    /usr/lib/atlas-base/atlas/liblapack.so.3
```

Note: In order to build the documentation and run the example code contains in this documentation you will need matplotlib:

```
sudo apt-get install python-matplotlib
```

Note: The above installs the ATLAS implementation of BLAS (the Basic Linear Algebra Subprograms library). Ubuntu 11.10 and later, and recent (testing) versions of Debian, offer an alternative implementation called OpenBLAS.

Using OpenBLAS can give speedups in some scikit-learn modules, but can freeze joblib/multiprocessing prior to OpenBLAS version 0.2.8-4, so using it is not recommended unless you know what you're doing.

If you do want to use OpenBLAS, then replacing ATLAS only requires a couple of commands. ATLAS has to be removed, otherwise NumPy may not work:

```
sudo apt-get remove libatlas3gf-base libatlas-dev
sudo apt-get install libopenblas-dev

sudo update-alternatives  --set libblas.so.3 \
    /usr/lib/openblas-base/libopenblas.so.0
sudo update-alternatives --set liblapack.so.3 \
    /usr/lib/lapack/liblapack.so.3
```

On Red Hat and clones (e.g. CentOS), install the dependencies using:

```
sudo yum -y install gcc gcc-c++ numpy python-devel scipy
```

Building scikit-learn with pip

This is usually the fastest way to install or upgrade to the latest stable release:

```
pip install --user --install-option="--prefix=" -U scikit-learn
```

The `--user` flag ask pip to install scikit-learn in the `$HOME/.local` folder therefore not requiring root permission. This flag should make pip ignore any old version of scikit-learn previously installed on the system while benefiting from system packages for numpy and scipy. Those dependencies can be long and complex to build correctly from source.

The `--install-option="--prefix="` flag is only required if Python has a `distutils.cfg` configuration with a predefined `prefix=` entry.

From source package

Download the source package from http://pypi.python.org/pypi/scikit-learn/ , unpack the sources and cd into the source directory.

This packages uses distutils, which is the default way of installing python modules. The install command is:

```
python setup.py install
```

or alternatively (also from within the scikit-learn source folder):

```
pip install .
```

Warning: Packages installed with the `python setup.py install` command cannot be uninstalled nor upgraded by `pip` later. To properly uninstall scikit-learn in that case it is necessary to delete the `sklearn` folder from your Python `site-packages` directory.

49.2 Third party distributions of scikit-learn

Some third-party distributions are now providing 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 Python and OS distributions that provide their own version of scikit-learn.

49.2.1 Debian and derivatives (Ubuntu)

The Debian package is named `python-sklearn` (formerly `python-scikits-learn`) and can be installed using the following command:

```
sudo apt-get install python-sklearn
```

Additionally, backport builds of the most recent release of scikit-learn for existing releases of Debian and Ubuntu are available from the NeuroDebian repository .

A quick-'n'-dirty way of rolling your own `.deb` package is to use stdeb.

49.2.2 Python(x,y) for Windows

The Python(x,y) project distributes scikit-learn as an additional plugin, which can be found in the Additional plugins page.

49.2.3 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.

> **Warning:** To upgrade or uninstall scikit-learn installed with Anaconda or `conda` you **should not use the pip command**. Instead:
>
> To upgrade `scikit-learn`:
>
> ```
> conda update scikit-learn
> ```
>
> To uninstall `scikit-learn`:
>
> ```
> conda remove scikit-learn
> ```
>
> Upgrading with `pip install -U scikit-learn` or uninstalling `pip uninstall scikit-learn` is likely fail to properly remove files installed by the `conda` command.
>
> pip upgrade and uninstall operations only work on packages installed via `pip install`.

49.2.4 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 py26-scikit-learn
```

or:

```
sudo port install py27-scikit-learn
```

49.2.5 Arch Linux

Arch Linux's package is provided through the official repositories as `python-scikit-learn` for Python 3 and `python2-scikit-learn` for Python 2. It can be installed by typing the following command:

```
# pacman -S python-scikit-learn
```

or:

```
# pacman -S python2-scikit-learn
```

depending on the version of Python you use.

49.2.6 NetBSD

scikit-learn is available via pkgsrc-wip:

http://pkgsrc.se/wip/py-scikit_learn

49.2.7 Fedora

The Fedora package is called `python-scikit-learn` for the Python 2 version and `python3-scikit-learn` for the Python 3 version. Both versions can be installed using `yum`:

```
$ sudo yum install python-scikit-learn
```

or:

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

49.3 Building on windows

To build scikit-learn on Windows you need a working C/C++ compiler in addition to numpy, scipy and setuptools.

Picking the right compiler depends on the version of Python (2 or 3) and the architecture of the Python interpreter, 32-bit or 64-bit. You can check the Python version by running the following in `cmd` or `powershell` console:

```
python --version
```

and the architecture with:

```
python -c "import struct; print(struct.calcsize('P') * 8)"
```

The above commands assume that you have the Python installation folder in your PATH environment variable.

49.3.1 32-bit Python

For 32-bit Python it is possible use the standalone installers for Microsoft Visual C++ Express 2008 for Python 2 or Microsoft Visual C++ Express 2010 or Python 3.

Once installed you should be able to build scikit-learn without any particular configuration by running the following command in the scikit-learn folder:

```
python setup.py install
```

49.3.2 64-bit Python

For the 64-bit architecture, you either need the full Visual Studio or the free Windows SDKs that can be downloaded from the links below.

The Windows SDKs include the MSVC compilers both for 32 and 64-bit architectures. They come as a `GRMSDKX_EN_DVD.iso` file that can be mounted as a new drive with a `setup.exe` installer in it.

- For Python 2 you need SDK **v7.0**: MS Windows SDK for Windows 7 and .NET Framework 3.5 SP1
- For Python 3 you need SDK **v7.1**: MS Windows SDK for Windows 7 and .NET Framework 4

Both SDKs can be installed in parallel on the same host. To use the Windows SDKs, you need to setup the environment of a `cmd` console launched with the following flags (at least for SDK v7.0):

```
cmd /E:ON /V:ON /K
```

Then configure the build environment with:

```
SET DISTUTILS_USE_SDK=1
SET MSSdk=1
"C:\Program Files\Microsoft SDKs\Windows\v7.0\Setup\WindowsSdkVer.exe" -q -version:v7.0
"C:\Program Files\Microsoft SDKs\Windows\v7.0\Bin\SetEnv.cmd" /x64 /release
```

Finally you can build scikit-learn in the same `cmd` console:

```
python setup.py install
```

Replace `v7.0` by the `v7.1` in the above commands to do the same for Python 3 instead of Python 2.

Replace `/x64` by `/x86` to build for 32-bit Python instead of 64-bit Python.

49.3.3 Building binary packages and installers

The `.whl` package and `.exe` installers can be built with:

```
pip install wheel
python setup.py bdist_wheel bdist_wininst -b doc/logos/scikit-learn-logo.bmp
```

The resulting packages are generated in the `dist/` folder.

49.3.4 Using an alternative compiler

It is possible to use MinGW (a port of GCC to Windows OS) as an alternative to MSVC for 32-bit Python. Not that extensions built with mingw32 can be redistributed as reusable packages as they depend on GCC runtime libraries typically not installed on end-users environment.

To force the use of a particular compiler, pass the `--compiler` flag to the build step:

```
python setup.py build --compiler=my_compiler install
```

where `my_compiler` should be one of `mingw32` or `msvc`.

49.4 Bleeding Edge

See section *Retrieving the latest code* on how to get the development version. Then follow the previous instructions to build from source depending on your platform.

49.5 Testing

49.5.1 Testing scikit-learn once installed

Testing requires having the nose library. After installation, the package can be tested by executing *from outside* the source directory:

```
$ nosetests -v sklearn
```

Under Windows, it is recommended to use the following command (adjust the path to the `python.exe` program) as using the `nosetests.exe` program can badly interact with tests that use `multiprocessing`:

```
C:\Python34\python.exe -c "import nose; nose.main()" -v sklearn
```

This should give you a lot of output (and some warnings) but eventually should finish with a message similar to:

```
Ran 3246 tests in 260.618s
OK (SKIP=20)
```

Otherwise, please consider posting an issue into the bug tracker or to the *Mailing List* including the traceback of the individual failures and errors.

49.5.2 Testing scikit-learn from within the source folder

Scikit-learn can also be tested without having the package installed. For this you must compile the sources inplace from the source directory:

```
python setup.py build_ext --inplace
```

Test can now be run using nosetests:

```
nosetests -v sklearn/
```

This is automated by the commands:

```
make in
```

and:

```
make test
```

You can also install a symlink named `site-packages/scikit-learn.egg-link` to the development folder of scikit-learn with:

```
pip install --editable .
```

CHAPTER

FIFTY

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.

50.1 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), is it said to have several attributes or **features**.

We can separate learning problems in a few large 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 classification problem would be the handwritten digit recognition example, 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 applying them to new data. This is why a common practice in machine learning to evaluate an algorithm is to split the data at hand into two sets, one that we call the **training set** on which we learn data properties and one that we call the **testing set** on which we test these properties.

50.2 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.

50.3 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` that implements support vector classification. The constructor of an estimator takes as arguments the parameters of the model, but for the time being, 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. It is possible to automatically find good values for the parameters by using tools such as *grid search* and *cross validation*.

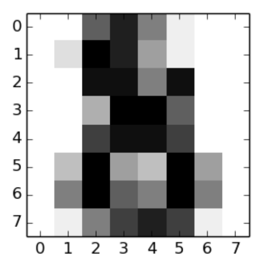
We call our estimator instance `clf`, as it is a classifier. It now must be fitted to the model, that is, it must *learn* from the model. This is done by passing our training set to the `fit` method. As a training set, let us use all the images of our dataset apart from the last one. We select this training set with the `[:-1]` Python syntax, which produces a new array that contains all but the last entry of `digits.data`:

```
>>> clf.fit(digits.data[:-1], digits.target[:-1])
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
  gamma=0.001, kernel='rbf', max_iter=-1, probability=False,
  random_state=None, shrinking=True, tol=0.001, verbose=False)
```

Now you can predict new values, in particular, we can ask to the classifier what is the digit of our last image in the `digits` dataset, which we have not used to train the classifier:

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

The corresponding image is the following:



As you can see, it is a challenging task: 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*.

50.4 Model persistence

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

```
>>> from sklearn import svm
>>> from sklearn import datasets
>>> clf = svm.SVC()
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> clf.fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3, gamma=0.0,
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)

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

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

```
>>> from sklearn.externals import joblib
>>> joblib.dump(clf, 'filename.pkl')
```

Later you can load back the pickled model (possibly in another Python process) with:

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

Note: joblib.dump returns a list of filenames. Each individual numpy array contained in the `clf` object is serialized as a separate file on the filesystem. All files are required in the same folder when reloading the model with joblib.load.

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.

CHAPTER

FIFTYONE

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 it's documentation.

Classification

SVM, *nearest neighbors*, *random forest*, ...

Examples

Regression

SVR, *ridge regression*, *Lasso*, ...

Examples

Clustering

k-Means, *spectral clustering*, *mean-shift*, ...

Examples

Dimensionality reduction

PCA, *feature selection*, *non-negative matrix factorization*.

Examples

Model selection

grid search, *cross validation*, *metrics*.

Examples

Preprocessing

preprocessing, *feature extraction*.

Examples

[M2012] "Machine Learning: A Probabilistic Perspective" Murphy, K. P. - chapter 14.4.3, pp. 492-493, The MIT Press, 2012

[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] 12. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001.

[B1998] 12. Breiman, "Arcing Classifiers", Annals of Statistics 1998.

[GEW2006] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.

[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] 8. Drucker. "Improving Regressors using Boosting Techniques", 1997.

[HTF] T. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning Ed. 2", Springer, 2009.

[F2001] J. Friedman, "Greedy Function Approximation: A Gradient Boosting Machine", The Annals of Statistics, Vol. 29, No. 5, 2001.

[F1999] 10. Friedman, "Stochastic Gradient Boosting", 1999

[HTF2009] 20. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning Ed. 2", Springer, 2009.

[R2007] 7. Ridgeway, "Generalized Boosted Models: A guide to the gbm package", 2007

[RH2007] V-Measure: A conditional entropy-based external cluster evaluation measure Andrew Rosenberg and Julia Hirschberg, 2007

[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

[RD1999] Rousseeuw, P.J., Van Driessen, K. "A fast algorithm for the minimum covariance determinant estimator" Technometrics 41(3), 212 (1999)

[R134] 12. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001.

[R135] 12. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001.

[R132] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.

[R133] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.

[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

[HTF2009] T. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning Ed. 2", Springer, 2009.

[Rouseeuw1984] *P. J. Rousseeuw. Least median of squares regression. J. Am Stat Ass, 79:871, 1984.*

[Rouseeuw1999] *A Fast Algorithm for the Minimum Covariance Determinant Estimator, 1999, American Statistical Association and the American Society for Quality, TECHNOMETRICS*

[Butler1993] *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*

[R109] I. Guyon, "Design of experiments for the NIPS 2003 variable selection benchmark", 2003.

[R110] J. Friedman, "Multivariate adaptive regression splines", The Annals of Statistics 19 (1), pages 1-67, 1991.

[R111] L. Breiman, "Bagging predictors", Machine Learning 24, pages 123-140, 1996.

[R112] J. Friedman, "Multivariate adaptive regression splines", The Annals of Statistics 19 (1), pages 1-67, 1991.

[R113] L. Breiman, "Bagging predictors", Machine Learning 24, pages 123-140, 1996.

[R114] J. Friedman, "Multivariate adaptive regression splines", The Annals of Statistics 19 (1), pages 1-67, 1991.

[R115] L. Breiman, "Bagging predictors", Machine Learning 24, pages 123-140, 1996.

[R116] 10. Zhu, H. Zou, S. Rosset, T. Hastie, "Multi-class AdaBoost", 2009.

[R117] T. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning Ed. 2", Springer, 2009.

[R118] 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.

[R119] S. Marsland, "Machine Learning: An Algorithmic Perspective", Chapter 10, 2009. http://www-ist.massey.ac.nz/smarsland/Code/10/lle.py

[R107] 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.

[R108] 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.

[Halko2009] *Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions Halko, et al., 2009 (arXiv:909)*

[MRT] *A randomized algorithm for the decomposition of matrices Per-Gunnar Martinsson, Vladimir Rokhlin and Mark Tygert*

[R120] Y. Freund, R. Schapire, “A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting”, 1995.

[R121] 10. Zhu, H. Zou, S. Rosset, T. Hastie, “Multi-class AdaBoost”, 2009.

[R122] Y. Freund, R. Schapire, “A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting”, 1995.

[R123] 8. Drucker, “Improving Regressors using Boosting Techniques”, 1997.

[R124] L. Breiman, “Pasting small votes for classification in large databases and on-line”, Machine Learning, 36(1), 85-103, 1999.

[R125] L. Breiman, “Bagging predictors”, Machine Learning, 24(2), 123-140, 1996.

[R126] T. Ho, “The random subspace method for constructing decision forests”, Pattern Analysis and Machine Intelligence, 20(8), 832-844, 1998.

[R127] G. Louppe and P. Geurts, “Ensembles on Random Patches”, Machine Learning and Knowledge Discovery in Databases, 346-361, 2012.

[R128] L. Breiman, “Pasting small votes for classification in large databases and on-line”, Machine Learning, 36(1), 85-103, 1999.

[R129] L. Breiman, “Bagging predictors”, Machine Learning, 24(2), 123-140, 1996.

[R130] T. Ho, “The random subspace method for constructing decision forests”, Pattern Analysis and Machine Intelligence, 20(8), 832-844, 1998.

[R131] G. Louppe and P. Geurts, “Ensembles on Random Patches”, Machine Learning and Knowledge Discovery in Databases, 346-361, 2012.

[R132] P. Geurts, D. Ernst., and L. Wehenkel, “Extremely randomized trees”, Machine Learning, 63(1), 3-42, 2006.

[R133] P. Geurts, D. Ernst., and L. Wehenkel, “Extremely randomized trees”, Machine Learning, 63(1), 3-42, 2006.

[R134] 12. Breiman, “Random Forests”, Machine Learning, 45(1), 5-32, 2001.

[R136] P. Geurts, D. Ernst., and L. Wehenkel, “Extremely randomized trees”, Machine Learning, 63(1), 3-42, 2006.

[R137] Moosmann, F. and Triggs, B. and Jurie, F. “Fast discriminative visual codebooks using randomized clustering forests” NIPS 2007

[R135] 12. Breiman, “Random Forests”, Machine Learning, 45(1), 5-32, 2001.

[Yates2011] *R. Baeza-Yates and B. Ribeiro-Neto (2011). Modern Information Retrieval. Addison Wesley, pp. 68-74.*

[MRS2008] *C.D. Manning, P. Raghavan and H. Schuetze (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 118-120.*

[R138] 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.

[R139] 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.

[NLNS2002] *H.B. Nielsen, S.N. Lophaven, H. B. Nielsen and J. Sondergaard. DACE - A MATLAB Kriging Toolbox.* (2002) http://www2.imm.dtu.dk/~hbn/dace/dace.pdf

[WBSWM1992] *W.J. Welch, R.J. Buck, J. Sacks, H.P. Wynn, T.J. Mitchell, and M.D. Morris (1992). Screening, predicting, and computer experiments. Technometrics, 34(1) 15–25.* http://www.jstor.org/pss/1269548

[R140] http://en.wikipedia.org/wiki/RANSAC

[R141] http://www.cs.columbia.edu/~belhumeur/courses/compPhoto/ransac.pdf

[R142] http://www.bmva.org/bmvc/2009/Papers/Paper355/Paper355.pdf

[R143] "Least Angle Regression", Effron et al. http://www-stat.stanford.edu/~tibs/ftp/lars.pdf

[R144] Wikipedia entry on the Least-angle regression

[R145] Wikipedia entry on the Lasso

[R147] *Roweis, S. & Saul, L. Nonlinear dimensionality reduction by locally linear embedding. Science 290:2323 (2000).*

[R148] *Donoho, D. & Grimes, C. Hessian eigenmaps: Locally linear embedding techniques for high-dimensional data. Proc Natl Acad Sci U S A. 100:5591 (2003).*

[R149] *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

[R150] *Zhang, Z. & Zha, H. Principal manifolds and nonlinear dimensionality reduction via tangent space alignment. Journal of Shanghai Univ. 8:406 (2004)*

[R146] Tenenbaum, J.B.; De Silva, V.; & Langford, J.C. A global geometric framework for nonlinear dimensionality reduction. Science 290 (5500)

[R151] *Roweis, S. & Saul, L. Nonlinear dimensionality reduction by locally linear embedding. Science 290:2323 (2000).*

[R152] *Donoho, D. & Grimes, C. Hessian eigenmaps: Locally linear embedding techniques for high-dimensional data. Proc Natl Acad Sci U S A. 100:5591 (2003).*

[R153] *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

[R154] *Zhang, Z. & Zha, H. Principal manifolds and nonlinear dimensionality reduction via tangent space alignment. Journal of Shanghai Univ. 8:406 (2004)*

[R157] Wikipedia entry for the Average precision

[R159] Wikipedia entry for the Confusion matrix

[R161] Wikipedia entry for the F1-score

[R162] R. Baeza-Yates and B. Ribeiro-Neto (2011). Modern Information Retrieval. Addison Wesley, pp. 327-328.

[R163] Wikipedia entry for the F1-score

[R164] Grigorios Tsoumakas, Ioannis Katakis. Multi-Label Classification: An Overview. International Journal of Data Warehousing & Mining, 3(3), 1-13, July-September 2007.

[R165] Wikipedia entry on the Hamming distance

[R166] Wikipedia entry on the Hinge loss

[R167] Koby Crammer, Yoram Singer. On the Algorithmic Implementation of Multiclass Kernel-based Vector Machines. Journal of Machine Learning Research 2, (2001), 265-292

[R168] L1 AND L2 Regularization for Multiclass Hinge Loss Models by Robert C. Moore, John DeNero.

[R170] Wikipedia entry for the Jaccard index

[R171] Baldi, Brunak, Chauvin, Andersen and Nielsen, (2000). Assessing the accuracy of prediction algorithms for classification: an overview

[R172] Wikipedia entry for the Matthews Correlation Coefficient

[R173] Wikipedia entry for the Precision and recall

[R174] Wikipedia entry for the F1-score

[R175] *Discriminative Methods for Multi-labeled Classification Advances in Knowledge Discovery and Data Mining (2004), pp. 22-30 by Shantanu Godbole, Sunita Sarawagi <http://www.godbole.net/shantanu/pubs/multilabelsvm-pakdd04.pdf>*

[R177] Wikipedia entry for the Receiver operating characteristic

[R178] Wikipedia entry for the Receiver operating characteristic

[R176] Wikipedia entry on the Coefficient of determination

[R160] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

[R155] Vinh, Epps, and Bailey, (2010). Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance, JMLR

[R156] Wikipedia entry for the Adjusted Mutual Information

[Hubert1985] *L. Hubert and P. Arabie, Comparing Partitions, Journal of Classification 1985* http://www.springerlink.com/content/x64124718341j1j0/

[wk] http://en.wikipedia.org/wiki/Rand_index#Adjusted_Rand_index

[R158] Andrew Rosenberg and Julia Hirschberg, 2007. V-Measure: A conditional entropy-based external cluster evaluation measure

[R169] Andrew Rosenberg and Julia Hirschberg, 2007. V-Measure: A conditional entropy-based external cluster evaluation measure

[R181] Peter J. Rousseeuw (1987). "Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis". Computational and Applied Mathematics 20: 53-65.

[R182] Wikipedia entry on the Silhouette Coefficient

[R179] Peter J. Rousseeuw (1987). "Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis". Computational and Applied Mathematics 20: 53-65.

[R180] Wikipedia entry on the Silhouette Coefficient

[R183] Andrew Rosenberg and Julia Hirschberg, 2007. V-Measure: A conditional entropy-based external cluster evaluation measure

[R184] "Solving multiclass learning problems via error-correcting output codes", Dietterich T., Bakiri G., Journal of Artificial Intelligence Research 2, 1995.

[R185] "The error coding method and PICTs", James G., Hastie T., Journal of Computational and Graphical statistics 7, 1998.

[R186] "The Elements of Statistical Learning", Hastie T., Tibshirani R., Friedman J., page 606 (second-edition) 2008.

[R187] M. Bawa, T. Condie and P. Ganesan, "LSH Forest: Self-Tuning Indexes for Similarity Search", WWW '05 Proceedings of the 14th international conference on World Wide Web, 651-660, 2005.

[R103] Obtaining calibrated probability estimates from decision trees and naive Bayesian classifiers, B. Zadrozny & C. Elkan, ICML 2001

[R104] Transforming Classifier Scores into Accurate Multiclass Probability Estimates, B. Zadrozny & C. Elkan, (KDD 2002)

[R105] Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods, J. Platt, (1999)

[R106] Predicting Good Probabilities with Supervised Learning, A. Niculescu-Mizil & R. Caruana, ICML 2005

[R188] Ping Li, T. Hastie and K. W. Church, 2006, "Very Sparse Random Projections". http://www.stanford.edu/~hastie/Papers/Ping/KDD06_rp.pdf

[R189] D. Achlioptas, 2001, "Database-friendly random projections", http://www.cs.ucsc.edu/~optas/papers/jl.pdf

[R190] http://en.wikipedia.org/wiki/Johnson%E2%80%93Lindenstrauss_lemma

[R191] 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

[R192] http://en.wikipedia.org/wiki/Decision_tree_learning

[R193] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification and Regression Trees", Wadsworth, Belmont, CA, 1984.

[R194] T. Hastie, R. Tibshirani and J. Friedman. "Elements of Statistical Learning", Springer, 2009.

[R195] L. Breiman, and A. Cutler, "Random Forests", http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

[R196] http://en.wikipedia.org/wiki/Decision_tree_learning

[R197] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification and Regression Trees", Wadsworth, Belmont, CA, 1984.

[R198] T. Hastie, R. Tibshirani and J. Friedman. "Elements of Statistical Learning", Springer, 2009.

[R199] L. Breiman, and A. Cutler, "Random Forests", http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

[R200] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.

[R201] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.

S

PYTHON MODULE INDEX

S

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

V

W

Z