



# **scikit-learn user guide**

*Release 0.13-git*

**scikit-learn developers**

January 26, 2013



# **CONTENTS**



Scikit-learn integrates **machine learning** algorithms in the tightly-knit scientific **Python** world, building upon [numpy](#), [scipy](#), and [matplotlib](#). As a machine-learning module, it provides versatile tools for data mining and analysis in any field of science and engineering. It strives to be **simple and efficient**, accessible to everybody, and reusable in various contexts.

**License:** Open source, commercially usable: [BSD license \(3 clause\)](#)

Documentation for scikit-learn **version** 0.13-git. For other versions and printable format, see *Documentation resources*.



# USER GUIDE

## 1.1 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*.

---

### 1.1.1 Installing an official release

#### Getting the dependencies

Installing from source requires you to have installed Python (>= 2.6), NumPy (>= 1.3), SciPy (>= 0.7), setuptools, Python development headers and a working C++ compiler. Under Debian-based operating systems, which include Ubuntu, you can install all these requirements by issuing:

```
sudo apt-get install build-essential python-dev python-numpy python-setuptools python-scipy libatlas
```

---

**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:** On older versions of Ubuntu, you might need to apt-get install python-numpy-dev to get the header files for NumPy.

---

On Ubuntu 10.04 LTS, the package *libatlas-dev* is called *libatlas-headers*.

---

---

**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.

---

While this implementation has some issues (please don't file bug reports about this), it may offer a significant speedup to some modules of scikit-learn, especially on multicore hardware. Replacing ATLAS with OpenBLAS only requires two commands:

```
# NumPy may not run when both ATLAS and OpenBLAS are installed,  
# so remove the former.  
sudo apt-get remove libatlas3gf-base libatlas-dev  
sudo apt-get install libopenblas-dev
```

## Easy install

This is usually the fastest way to install the latest stable release. If you have pip or easy\_install, you can install or update with the command:

```
pip install -U scikit-learn
```

or:

```
easy_install -U scikit-learn
```

for easy\_install. Note that you might need root privileges to run these commands.

## From source package

Download the 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
```

## Windows installer

You can download a Windows installer from downloads in the project's web page. Note that must also have installed the packages numpy and setuptools.

This package is also expected to work with python(x,y) as of 2.6.5.5.

### Installing on Windows 64-bit

To install a 64-bit version of scikit-learn, you can download the binaries from <http://www.lfd.uci.edu/~gohlke/pythonlibs/#scikit-learn>. Note that this will require a compatible version of numpy, scipy and matplotlib. The easiest option is to also download them from the same URL.

## Building on windows

To build scikit-learn on windows you will need a C/C++ compiler in addition to numpy, scipy and setuptools. At least [MinGW](#) (a port of GCC to Windows OS) and Microsoft Visual C++ 2008 should work out of the box. To force the use of a particular compiler, write a file named `setup.cfg` in the source directory with the content:

```
[build_ext]
compiler=my_compiler

[build]
compiler=my_compiler
```

where `my_compiler` should be one of `mingw32` or `msvc`.

When the appropriate compiler has been set, and assuming Python is in your PATH (see [Python FAQ for windows](#) for more details), installation is done by executing the command:

```
python setup.py install
```

To build a precompiled package like the ones distributed at [the downloads section](#), the command to execute is:

```
python setup.py bdist_wininst -b doc/logos/scikit-learn-logo.bmp
```

This will create an installable binary under directory `dist/`.

## 1.1.2 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:

### Debian and derivatives (Ubuntu)

The Debian package is named `python-sklearn` (formerly `python-scikits-learn`) and can be installed using the following commands with root privileges:

```
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 [NeuroDebian repository](#).

### Python(x, y)

The `Python(x, y)` distributes scikit-learn as an additional plugin, which can be found in the [Additional plugins page](#).

### Enthought Python distribution

The [Enthought Python Distribution](#) already ships a recent version.

### Macports

The macport's package is named `py26-sklearn` or `py27-sklearn` depending on the version of Python. It can be installed by typing the following command:

```
sudo port install py26-sklearn
```

or:

```
sudo port install py27-sklearn
```

depending on the version of Python you want to use.

## Archlinux

Archlinux's package is provided at [Arch User Repository \(AUR\)](#) with name *python2-scikit-learn* for latest stable version and *python2-scikit-learn-git* for building from git version. If *yaourt* is available, it can be installed by typing the following command:

```
sudo yaourt -S python2-scikit-learn
```

or:

```
sudo yaourt -S python2-scikit-learn-git
```

depending on the version of scikit-learn you want to use.

## NetBSD

scikit-learn is available via `pkgsrc-wip`:

[http://pkgsrc.se/wip/py-scikit\\_learn](http://pkgsrc.se/wip/py-scikit_learn)

### 1.1.3 Bleeding Edge

See section *Retrieving the latest code* on how to get the development version.

### 1.1.4 Testing

Testing requires having the `nose` library. After installation, the package can be tested by executing *from outside* the source directory:

```
nosetests sklearn --exe
```

This should give you a lot of output (and some warnings) but eventually should finish with a message similar to:

```
Ran 601 tests in 27.920s
OK (SKIP=2)
```

Otherwise, please consider posting an issue into the [bug tracker](#) or to the [Mailing List](#).

---

#### Note: Alternative testing method

If for some reason the recommended method is failing for you, please try the alternate method:

```
python -c "import sklearn; sklearn.test()"
```

This method might display doctest failures because of nosetests issues.

---

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 sklearn/
```

This is automated by the commands:

```
make in
```

and:

```
make test
```

## 1.2 Tutorials: From the bottom up with scikit-learn

### Quick start

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

### 1.2.1 An introduction to machine learning with scikit-learn

#### Section contents

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

#### 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.

### 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.:

```
>>> 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, explanatory 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,  $n\_samples$ ,  $n\_features$ , 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 the *scikit-learn*.

## 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

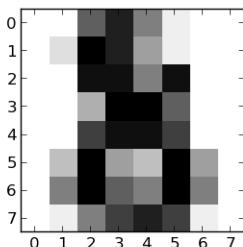
In this example we set the value of `gamma` manually. It is possible to automatically find good values for the parameters by using tools such as *grid search* and *cross validation*.

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:

```
>>> 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, 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:  
The images are of poor resolution. Do you agree with the classifier?

As you can see, it is a challenging task: the

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

## 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, 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')
```

### Statistical-learning Tutorial

This tutorial covers some of the models and tools available to do data-processing with Scikit Learn and how to learn from your data.

## 1.2.2 A tutorial on statistical-learning for scientific data processing

## Statistical learning

Machine learning is a technique with a growing importance, as the size of the datasets experimental sciences are facing is rapidly growing. Problems it tackles range from building a prediction function linking different observations, to classifying observations, or learning the structure in an unlabeled dataset.

This tutorial will explore *statistical learning*, that is the use of machine learning techniques with the goal of *statistical inference*: drawing conclusions on the data at hand.

`sklearn` is a Python module integrating classic machine learning algorithms in the tightly-knit world of scientific Python packages (`numpy`, `scipy`, `matplotlib`).

**Warning:** In scikit-learn release 0.9, the import path has changed from `scikits.learn` to `sklearn`. To import with cross-version compatibility, use:

```
try:
    from sklearn import something
except ImportError:
    from scikits.learn import something
```

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

### Datasets

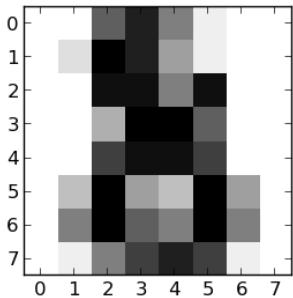
The *scikit-learn* deals with learning information from one or more datasets that are represented as 2D arrays. They can be understood as a list of multi-dimensional observations. We say that the first axis of these arrays is the **samples** axis, while the second is the **features** axis.

### 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.

**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))
```

**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_
```

**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 the *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

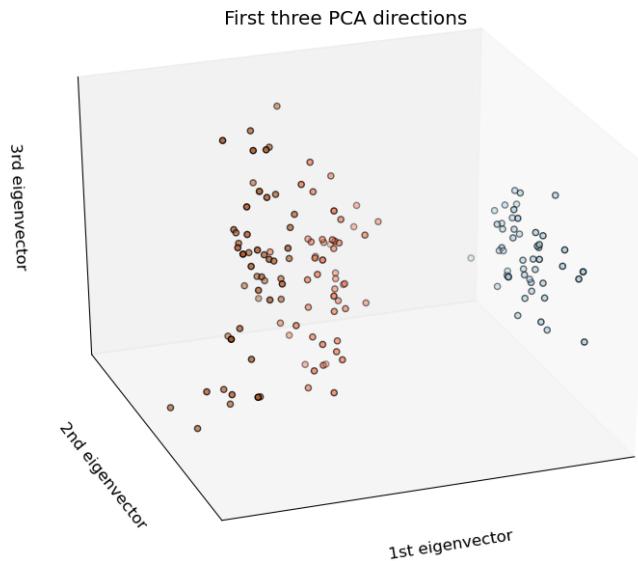
If the prediction task is to classify the observations in a set of finite labels, in other words to “name” the objects observed, the task is said to be a **classification** task. On the other hand, if the goal is to predict a continuous target variable, it is said to be a **regression** task.

In the *scikit-learn* for classification tasks,  $y$  is a vector of integers.

Note: See the *Introduction to machine learning with Scikit-learn Tutorial* for a quick run-through on the basic machine learning vocabulary used within Scikit-learn.

## Nearest neighbor and the curse of dimensionality

### Classifying irises:



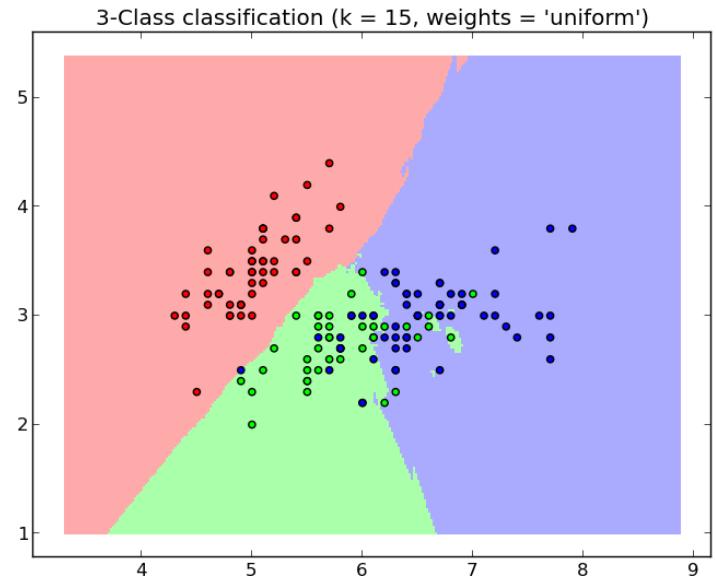
The iris dataset is a classification task consisting in identifying 3 different types of irises (Setosa, Versicolour, and Virginica) from their petal and sepal length and width:

```
>>> import numpy as np
>>> from sklearn import datasets
>>> iris = 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  $x_{\text{test}}$ , 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, n_neighbors=5, p=2,
                     warn_on_equidistant=True, 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  $n \sim 1/d$  points.

In the context of the above *KNN* example, if the data is described by just one feature with values ranging from 0 to 1 and with  $n$  training observations, then new data will be no further away than  $1/n$ . Therefore, the nearest neighbor decision rule will be efficient as soon as  $1/n$  is small compared to the scale of between-class feature variations.

If the number of features is  $p$ , you now require  $n \sim 1/d^p$  points. Let's say that we require 10 points in one dimension: Now  $10^p$  points are required in  $p$  dimensions to pave the  $[0, 1]$  space. As  $p$  becomes large, the number of training points required for a good estimator grows exponentially.

For example, if each point is just a single number (8 bytes), then an effective *KNN* estimator in a paltry  $p \sim 20$  dimensions would require more training data than the current estimated size of the entire internet! ( $\pm 1000$  Exabytes or so).

This is called the [curse of dimensionality](#) and is a core problem that machine learning addresses.

### Linear model: from regression to sparsity

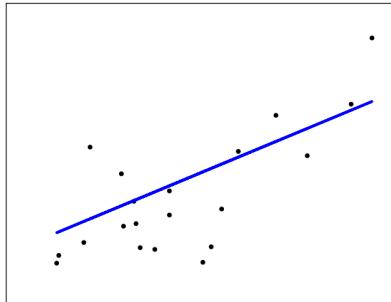
#### Diabetes dataset

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

```
>>> diabetes = 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
- $\beta$ : Coefficients
- $\epsilon$ : Observation noise

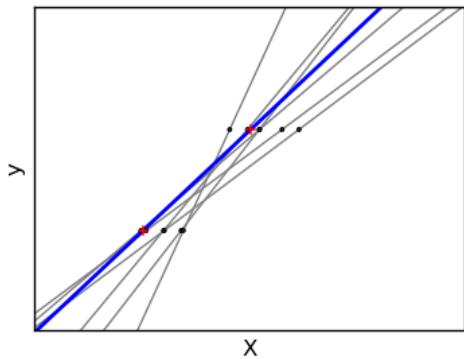
```
>>> from sklearn import linear_model
>>> regr = linear_model.LinearRegression()
>>> regr.fit(diabetes_X_train, diabetes_y_train)
LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
```

```
>>> 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:

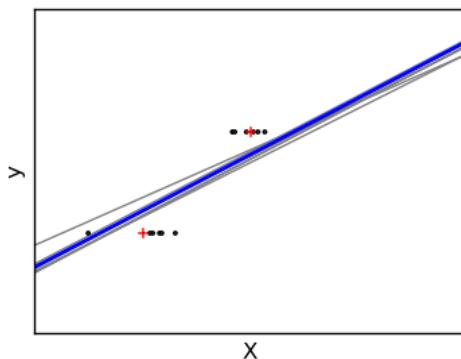


```
>>> X = np.c_[ .5, 1].T
>>> y = [.5, 1]
>>> test = np.c_[ 0, 2].T
>>> regr = linear_model.LinearRegression()

>>> 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:

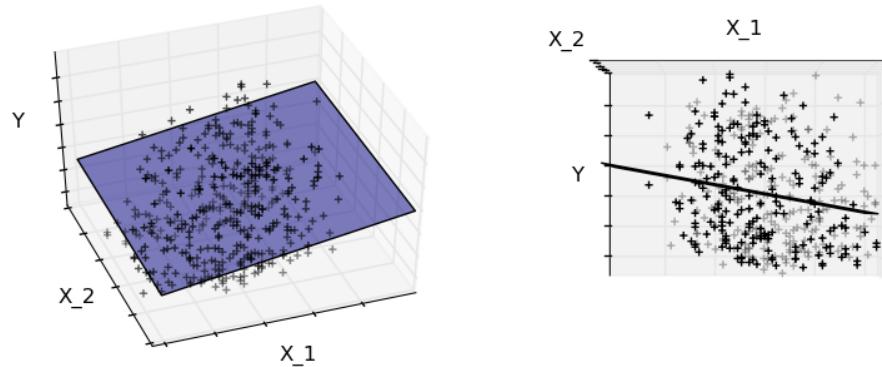
```
>>> alphas = np.logspace(-4, -1, 6)
>>> print [regr.set_params(alpha=alpha
...                 ).fit(diabetes_X_train, diabetes_y_train,
...                 ).score(diabetes_X_test, diabetes_y_test) for alpha in alphas]
[0.5851110683883..., 0.5852073015444..., 0.5854677540698..., 0.5855512036503...,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

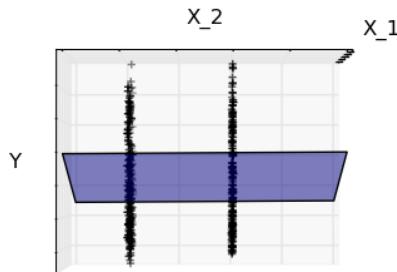
---

**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.

---

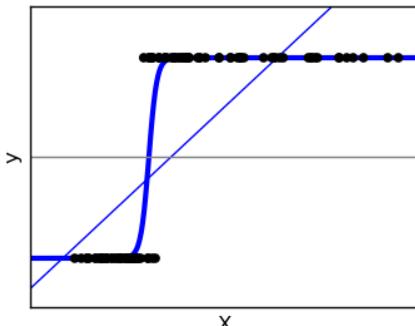
We can see that, although feature 2 has a strong coefficient on the full model, it conveys little information on  $y$  when considered with feature 1.

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='auto',
      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

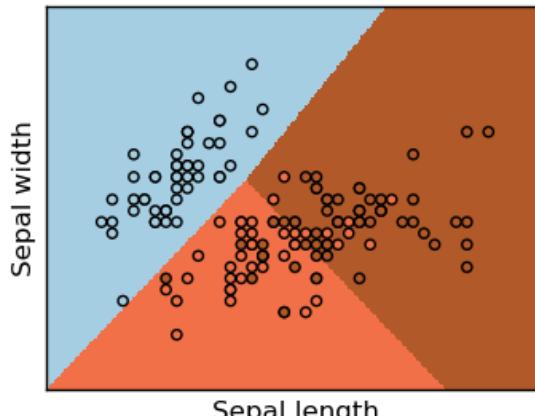
Different algorithms can be used to solve the same mathematical problem. For instance the *Lasso* object in the *scikit-learn* solves the lasso regression problem using a [coordinate decent](#) method, that is efficient on large datasets. However, the *scikit-learn* also provides the *LassoLars* object using the *LARS* which is very efficient for problems in which the weight vector estimated is very sparse, (i.e. problems with very few observations).

**Classification**

For classification, as in the labeling `iris` task, linear regression is not the right approach as it will give too much weight to data far from the decision frontier. A linear approach is to fit a sigmoid function or **logistic** function:

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

```
>>> logistic = linear_model.LogisticRegression(C=1e5)
>>> logistic.fit(iris_X_train, iris_y_train)
LogisticRegression(C=1000000.0, class_weight=None, dual=False,
    fit_intercept=True, intercept_scaling=1, penalty='l2',
    random_state=None, tol=0.0001)
```



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.

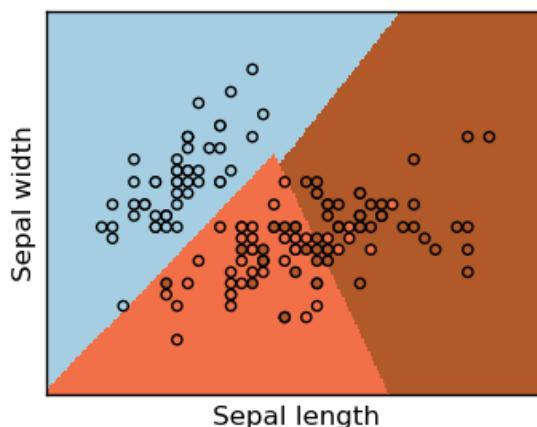
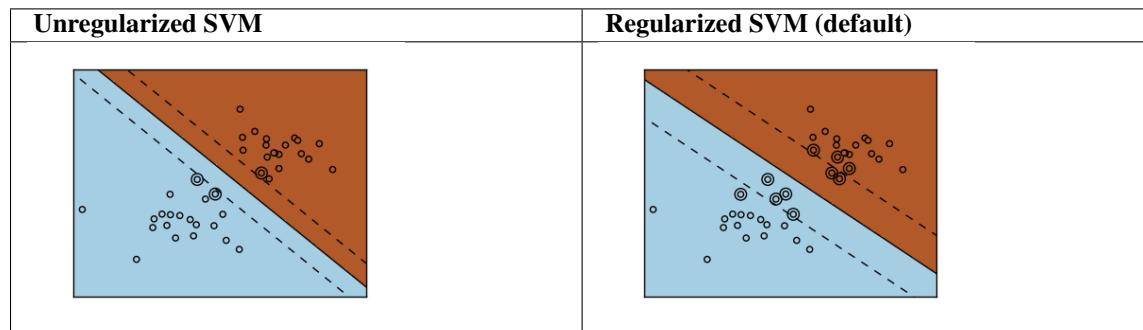
```
from sklearn import datasets, neighbors, linear_model
```

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

Solution: [..../auto\\_examples/exercises/plot\\_digits\\_classification\\_exercise.py](#)

**Support vector machines (SVMs)**

**Linear SVMs** *Support Vector Machines* belong to the discriminant model family: they try to find a combination of samples to build a plane maximizing the margin between the two classes. Regularization is set by the  $C$  parameter: a small value for  $C$  means the margin is calculated using many or all of the observations around the separating line (more regularization); a large value for  $C$  means the margin is calculated on observations close to the separating line (less regularization).



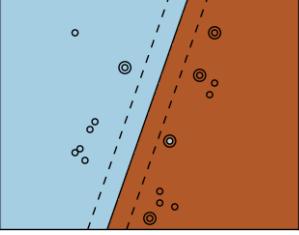
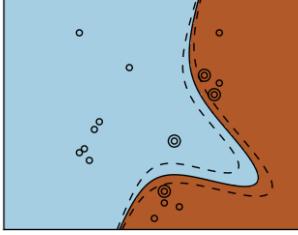
SVMs can be used in regression –SVR (Support Vector Regression)–, or in classification –SVC (Support Vector Classification).

```
>>> 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, shrinking=True, tol=0.001,
    verbose=False)
```

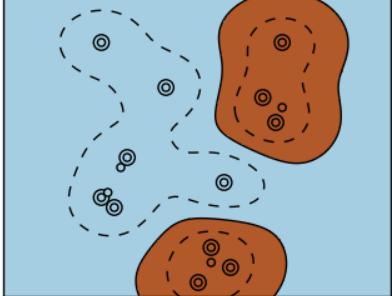
**Warning: Normalizing data**

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:

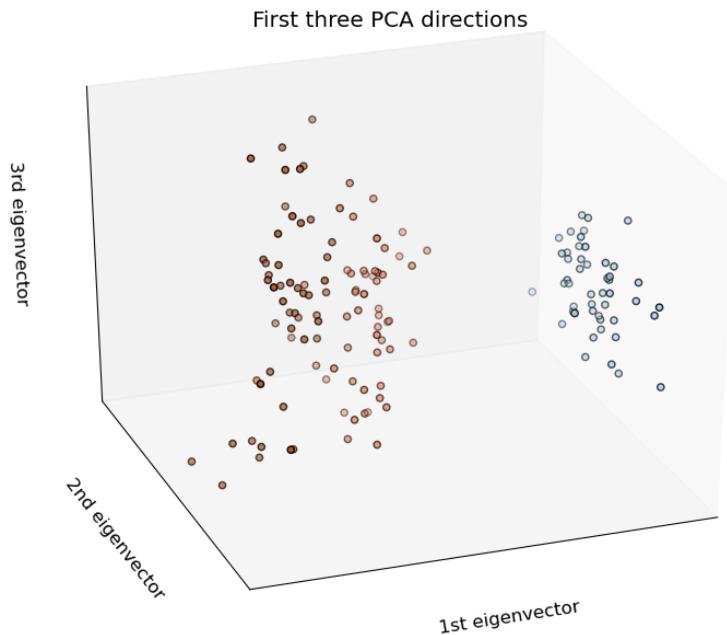
Linear kernel	Polynomial kernel
	
<pre>&gt;&gt;&gt; svc = svm.SVC(kernel='linear')</pre>	<pre>&gt;&gt;&gt; svc = svm.SVC(kernel='poly', ...                  degree=3) &gt;&gt;&gt; # degree: polynomial degree</pre>

**RBF kernel (Radial Basis Function)**


<pre>&gt;&gt;&gt; svc = svm.SVC(kernel='rbf') &gt;&gt;&gt; # gamma: inverse of size of &gt;&gt;&gt; # radial kernel</pre>

**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](#)

## Model selection: choosing estimators and their parameters

### Score, and cross-validated scores

As we have seen, every estimator exposes a *score* method that can judge the quality of the fit (or the prediction) on new data. **Bigger is better.**

```
>>> from sklearn import datasets, svm
>>> 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:])
0.9799999999999998
```

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

### Cross-validation generators

The code above to split data in train and test sets is tedious to write. The *sklearn* 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, indices=True)
>>> 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]
```

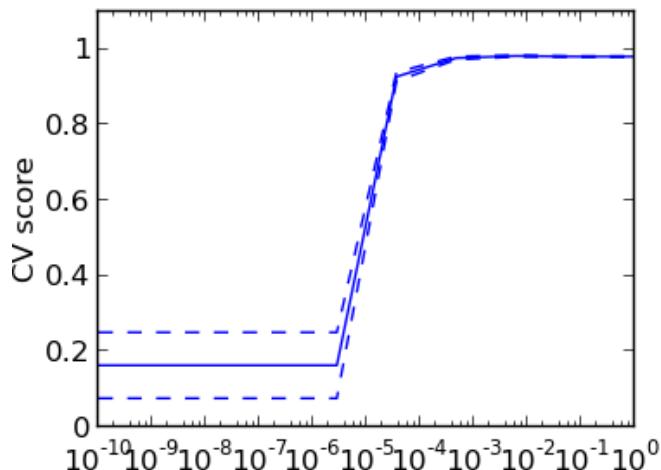
To compute the *score* method of an estimator, the *sklearn* exposes a helper function:

```
>>> 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 ( <b>n, k</b> )	StratifiedKFold ( <b>y, k</b> )	LeaveOneOut ( <b>n</b> )	LeaveOneLabelOut ( <b>labels</b> )
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 a linear kernel as a function of parameter  $C$  (use a logarithmic grid of points, from 1 to 10).

```
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()
```

**Solution:** *Cross-validation on Digits Dataset Exercise*

**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
>>> gammas = np.logspace(-6, -1, 10)
>>> clf = GridSearchCV(estimator=svc, param_grid=dict(gamma=gammas),
...                     n_jobs=-1)
>>> clf.fit(X_digits[:1000], y_digits[:1000])
GridSearchCV(cv=None, ...
>>> clf.best_score_
0.988991985997974
>>> clf.best_estimator_.gamma
9.99999999999995e-07

>>> # Prediction performance on test set is not as good as on train set
>>> clf.score(X_digits[1000:], y_digits[1000:])
0.94228356336260977
```

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([ 0.97996661,  0.98163606,  0.98330551])
```

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, normalize=False, precompute='auto',
        tol=0.0001, verbose=False)
>>> # The estimator chose automatically its lambda:
>>> lasso.alpha_
0.01318...
```

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?

```
import numpy as np
import pylab as pl

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*

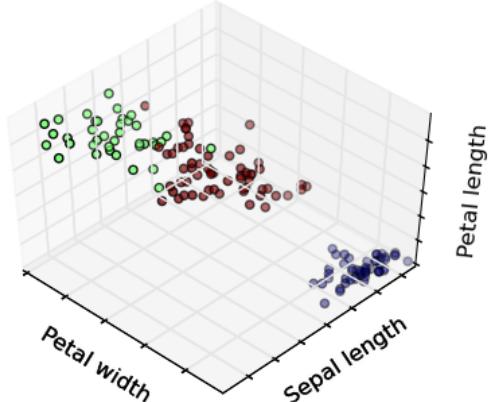
## Unsupervised learning: seeking representations of the data

### Clustering: grouping observations together

#### The problem solved in clustering

Given the iris dataset, if we knew that there were 3 types of iris, but did not have access to a taxonomist to label them: we could try a **clustering task**: split the observations into well-separated group called *clusters*.

**K-means clustering** Note that there exist a lot of different clustering criteria and associated algorithms. The simplest

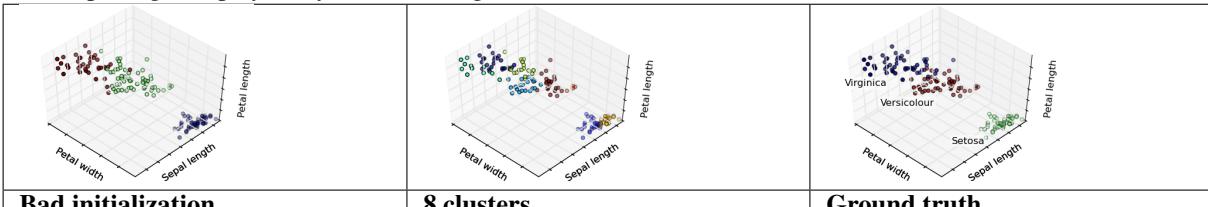


clustering algorithm is *K-means*.

```
>>> from sklearn import cluster, datasets
>>> 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 in the `sklearn` package we play many tricks to mitigate this issue.

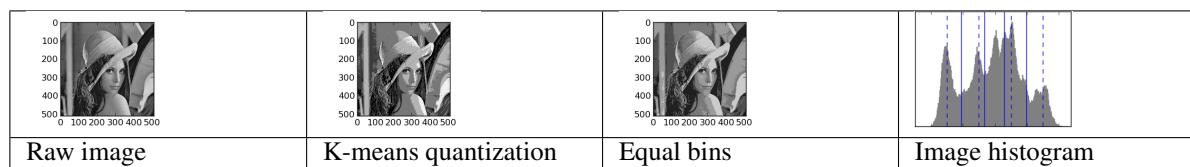


**Don't over-interpret clustering results**

### 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
```



**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, or
- **Divisive** - *top-down* approaches.

For estimating a large number of clusters, top-down approaches are both statistically ill-posed and slow due to it starting with all observations as one cluster, which it splits recursively. Agglomerative hierarchical-clustering is a bottom-up approach that successively merges observations together and is particularly useful when the clusters of interest are made of only a few observations. *Ward* clustering minimizes a criterion similar to k-means in a bottom-up approach. When the number of clusters is large, it is much more computationally efficient than k-means.

**Connectivity-constrained clustering** With Ward 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



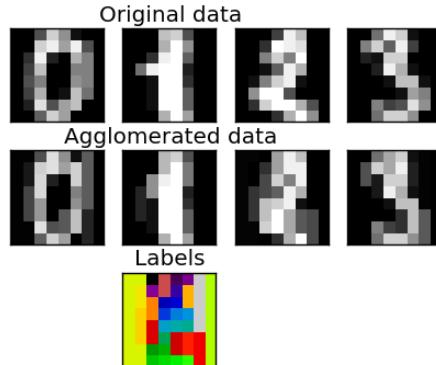
components) when clustering an image:

```
#####
# 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 = Ward(n_clusters=n_clusters, 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.WardAgglomeration(connectivity=connectivity,
...                                         n_clusters=32)
>>> agglo.fit(X)
WardAgglomeration(compute_full_tree='auto',...
>>> X_reduced = agglo.transform(X)

>>> X_approx = agglo.inverse_transform(X_reduced)
>>> images_approx = np.reshape(X_approx, images.shape)
```

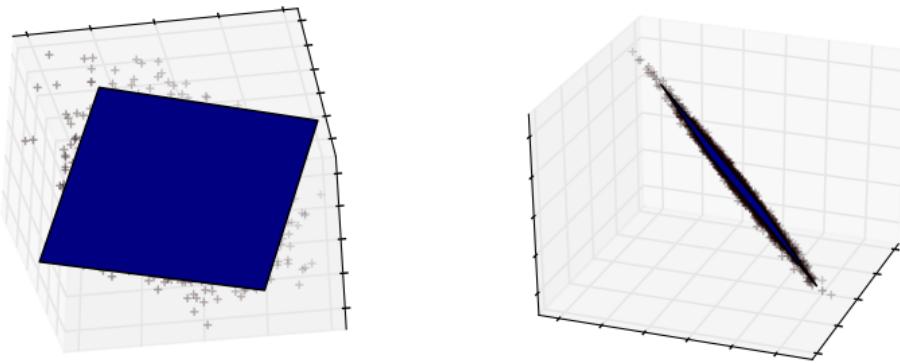
***transform* and *inverse\_transform* methods**

Some estimators expose a *transform* method, for instance to reduce the dimensionality of the dataset.

**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*

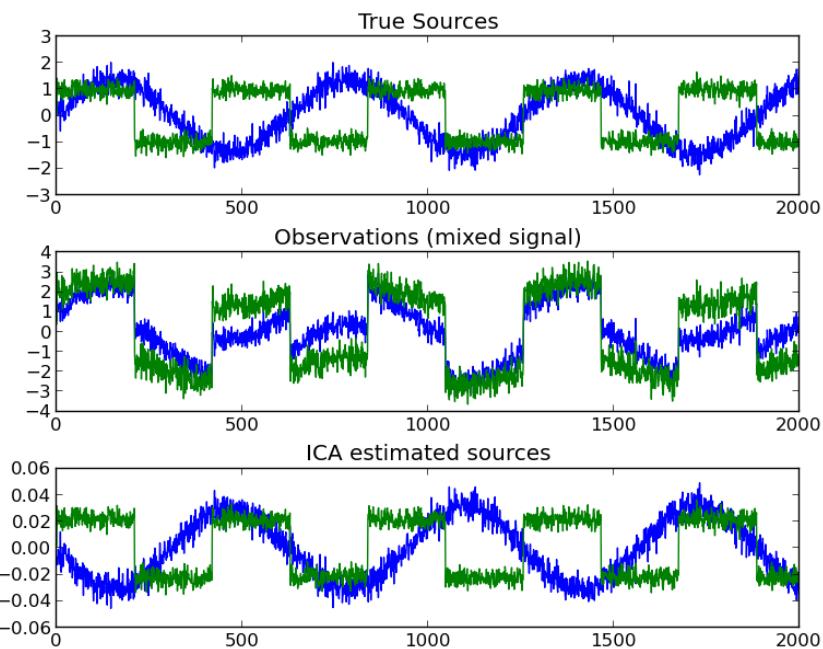
When used to *transform* data, PCA can reduce the dimensionality of the data by projecting on a principal subspace.

```
>>> # 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:



Gaussian independent signals:

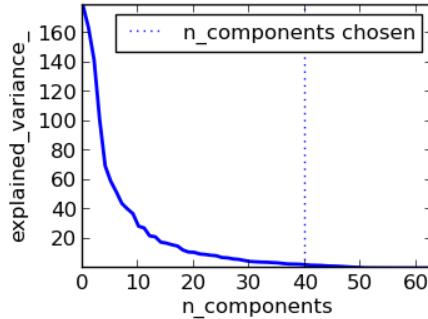
```
>>> # Generate sample data
>>> time = np.linspace(0, 10, 2000)
>>> s1 = np.sin(2 * time) # Signal 1 : sinusoidal signal
>>> s2 = np.sign(np.sin(3 * time)) # Signal 2 : square signal
>>> S = np.c_[s1, s2]
>>> S += 0.2 * np.random.normal(size=S.shape) # Add noise
>>> S /= S.std(axis=0) # Standardize 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(X).transform(X) # Get the estimated sources
>>> A_ = ica.get_mixing_matrix() # Get estimated mixing matrix
>>> np.allclose(X, np.dot(S_, A_.T))
True
```

## Putting it all together

### Pipelining

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



create combined estimators:

```
import pylab as pl

from sklearn import linear_model, decomposition, datasets

logistic = linear_model.LogisticRegression()

pca = decomposition.PCA()
from sklearn.pipeline import Pipeline
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)

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

#####
# Prediction

from sklearn.grid_search import GridSearchCV

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)

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

## Face recognition with eigenfaces

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

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

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

*The dataset used in this example is a preprocessed excerpt of the “Labeled Faces in the Wild”, aka LFW\_:*

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

.. \_LFW: <http://vis-www.cs.umass.edu/lfw/>

*Expected results for the top 5 most represented people in the dataset::*

	precision	recall	f1-score	support
<i>Gerhard_Schroeder</i>	0.91	0.75	0.82	28
<i>Donald_Rumsfeld</i>	0.84	0.82	0.83	33
<i>Tony_Blair</i>	0.65	0.82	0.73	34
<i>Colin_Powell</i>	0.78	0.88	0.83	58
<i>George_W_Bush</i>	0.93	0.86	0.90	129
<i>avg / total</i>	0.86	0.84	0.85	282

```
"""
print __doc__

from time import time
import logging
import pylab as pl

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

# 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 the people names on the testing 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"""
    pl.figure(figsize=(1.8 * n_col, 2.4 * n_row))
    pl.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
    for i in range(n_row * n_col):
        pl.subplot(n_row, n_col, i + 1)
        pl.imshow(images[i].reshape((h, w)), cmap=pl.cm.gray)
        pl.title(titles[i], size=12)
        pl.xticks(())
        pl.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)

pl.show()
```

<p><b>Prediction</b></p>	<p><b>Eigenfaces</b></p>
--------------------------	--------------------------

Expected results for the top 5 most represented people in the dataset:

	precision	recall	f1-score	support
Gerhard_Schroeder	0.91	0.75	0.82	28
Donald_Rumsfeld	0.84	0.82	0.83	33
Tony_Blair	0.65	0.82	0.73	34
Colin_Powell	0.78	0.88	0.83	58
George_W_Bush	0.93	0.86	0.90	129
avg / total	0.86	0.84	0.85	282

## Open problem: Stock Market Structure

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

*Visualizing the stock market structure*

## Finding help

### The project mailing list

If you encounter a bug with scikit-learn or something that needs clarification in the docstring or the online documentation, please feel free to ask on the [Mailing List](#)

### Q&A communities with Machine Learning practitioners

**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>

### 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

### Videos

Videos with tutorials can also be found in the *Videos* section.

### 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.

## 1.3 Supervised learning

### 1.3.1 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,

$$\hat{y}(w, x) = w_0 + w_1x_1 + \dots + w_px_p$$

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

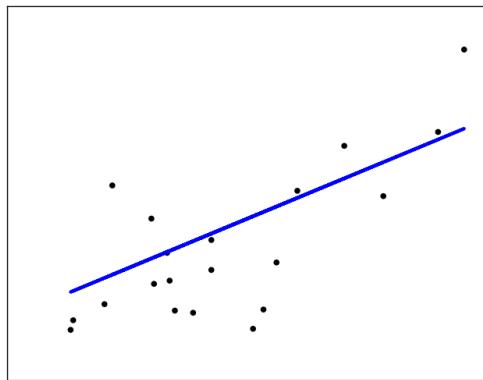
To perform classification with generalized linear models, see *Logistic regression*.

### Ordinary Least Squares

`LinearRegression` fits a linear model with coefficients  $w = (w_1, \dots, w_p)$  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:

$$\min_w \|Xw - y\|_2^2$$

`LinearRegression` will take in its *fit* method arrays `X`, `y` and will store the coefficients `w` of the linear model in its `coef_` member:



```
>>> from sklearn import linear_model
>>> clf = linear_model.LinearRegression()
>>> clf.fit ([[0, 0], [1, 1], [2, 2]], [0, 1, 2])
LinearRegression(copy_X=True, fit_intercept=True, 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  $n \geq p$ .

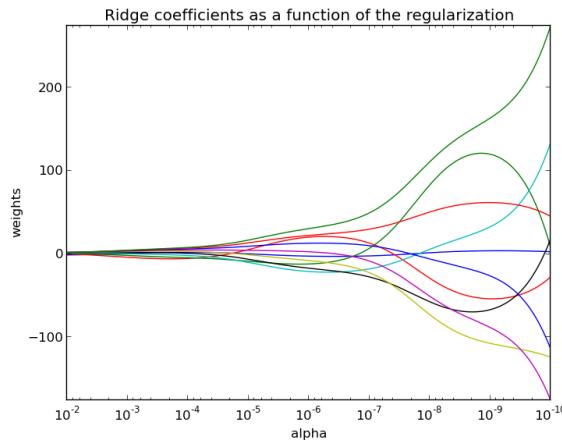
## Ridge Regression

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,

$$\min_w \|Xw - y\|_2^2 + \alpha \|w\|_2^2$$

Here,  $\alpha \geq 0$  is a complexity parameter that controls the amount of shrinkage: the larger the value of  $\alpha$ , 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, loss_func=None,
        normalize=False, score_func=None)
>>> clf.alpha_
0.1
```

## References

- “Notes on Regularized Least Squares”, Rifkin & Lippert ([technical report, course slides](#)).

## Lasso

The Lasso is a linear model that estimates sparse coefficients. It is useful in some contexts due to its tendency to prefer solutions with fewer 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  $\ell_1$  prior as regularizer. The objective function to minimize is:

$$\min_w \frac{1}{2n_{samples}} \|Xw - y\|_2^2 + \alpha \|w\|_1$$

The lasso estimate thus solves the minimization of the least-squares penalty with  $\alpha \|w\|_1$  added, where  $\alpha$  is a constant and  $\|w\|_1$  is the  $\ell_1$ -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='auto', 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*.

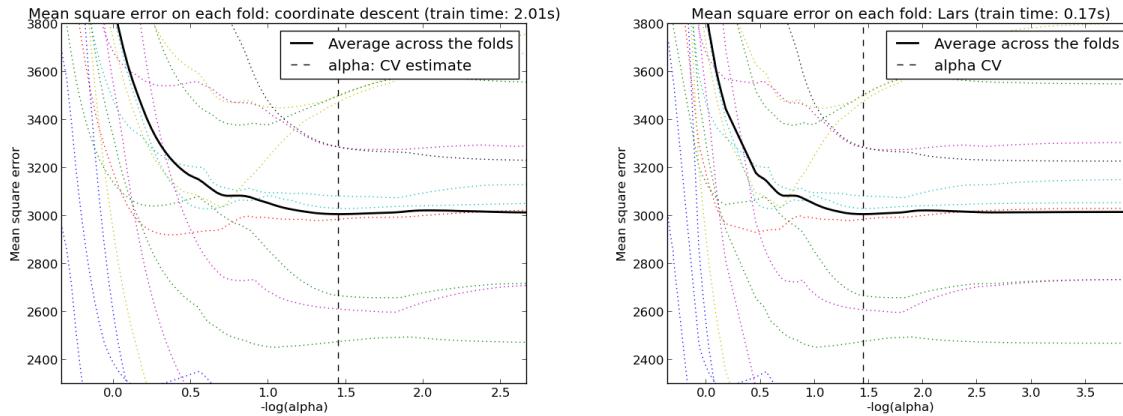
---

## Setting regularization parameter

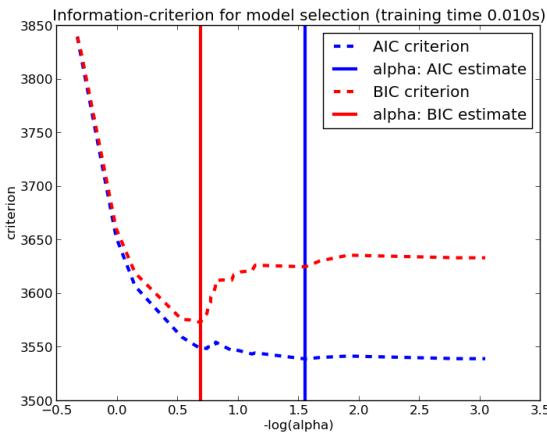
The `alpha` parameter control the degree of sparsity of the coefficients estimated.

**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*

## Elastic Net

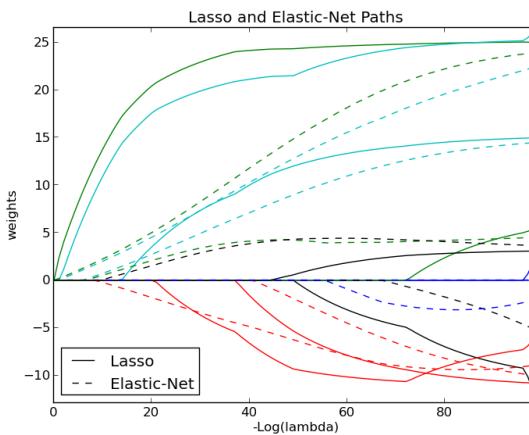
ElasticNet is a linear model trained with L1 and L2 prior as regularizer. This combination allows for learning a sparse model where few of the weights are non-zero like Lasso, while still maintaining the regularization properties of Ridge. We control this tradeoff using the `l1_ratio` parameter.

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

$$\min_w \frac{1}{2n_{samples}} \|Xw - y\|_2^2 + \alpha\rho\|w\|_1 + \frac{\alpha(1-\rho)}{2}\|w\|_2^2$$



The class `ElasticNetCV` can be used to set the parameters `alpha` and `rho` by cross-validation.

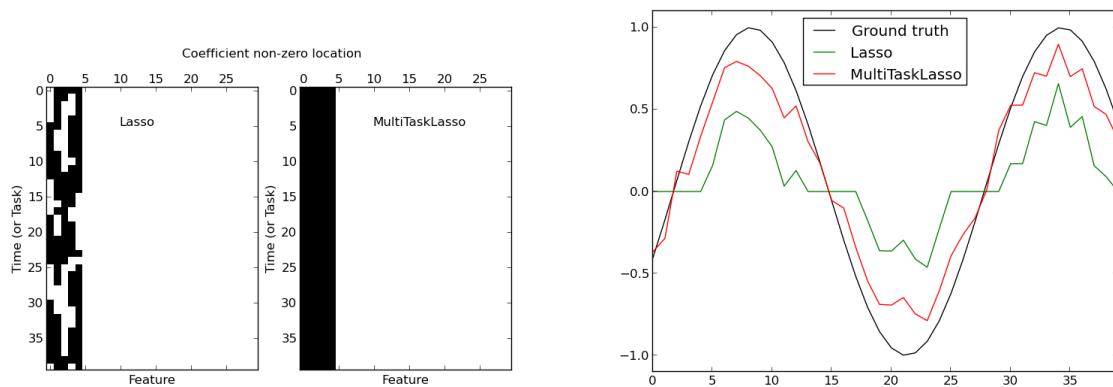
### Examples:

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

## 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  $\ell_1 \ell_2$  prior as regularizer. The objective function to minimize is:

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

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.

## 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  $p \gg n$  (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.

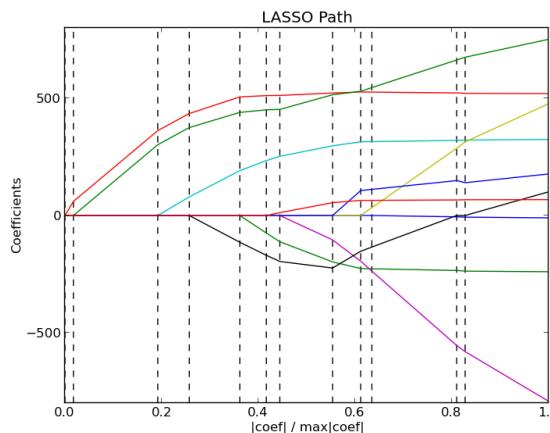
The disadvantages of the LARS method include:

- 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`.

## 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.

## Orthogonal Matching Pursuit (OMP)

`OrthogonalMatchingPursuit` and `orthogonal_mp` implements the OMP algorithm for approximating the fit of a linear model with constraints imposed on the number of non-zero coefficients (ie. the  $L_0$  pseudo-norm).

Being a forward feature selection method like *Least Angle Regression*, orthogonal matching pursuit can approximate the optimum solution vector with a fixed number of non-zero elements:

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

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

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

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,

## 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  $\ell_2$  regularization used in Ridge Regression is equivalent to finding a maximum a-posteriori solution under a Gaussian prior over the parameters  $w$  with precision  $\lambda^{-1}$ . 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

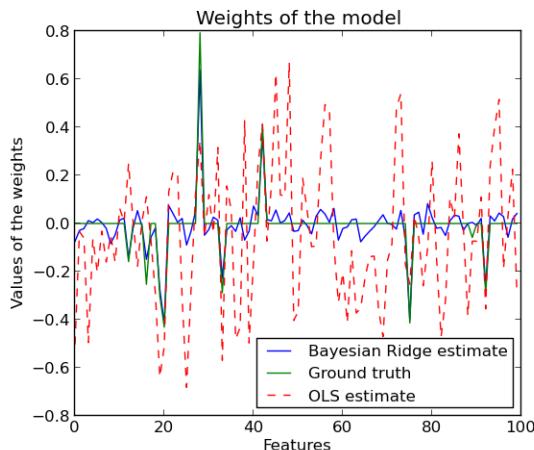
`BayesianRidge` estimates a probabilistic model of the regression problem as described above. The prior for the parameter  $w$  is given by a spherical Gaussian:

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

The priors over  $\alpha$  and  $\lambda$  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  $w$ ,  $\alpha$  and  $\lambda$  are estimated jointly during the fit of the model. The remaining hyperparameters are the parameters of the gamma priors over  $\alpha$  and  $\lambda$ . 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  $w$ <sup>1</sup><sup>2</sup>. 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  $\lambda_i$ :

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

with  $diag(A) = \lambda = \{\lambda_1, \dots, \lambda_p\}$ .

In contrast to Bayesian Ridge Regression, each coordinate of  $w_i$  has its own standard deviation  $\lambda_i$ . The prior over all  $\lambda_i$  is chosen to be the same gamma distribution given by hyperparameters  $\lambda_1$  and  $\lambda_2$ .

### Examples:

- *Automatic Relevance Determination Regression (ARD)*

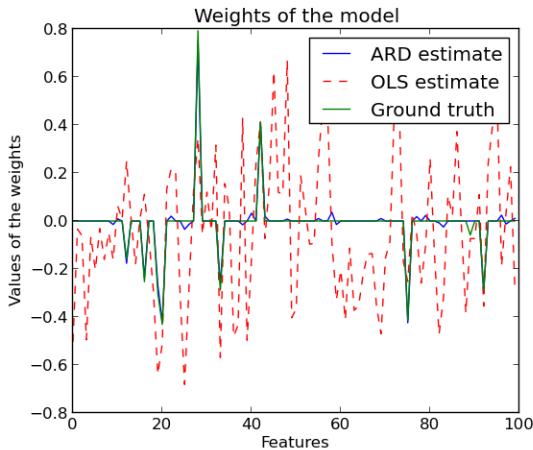
### References

## Logistic regression

Logistic regression, despite its name, is a linear model for classification rather than regression. As such, it minimizes a “hit or miss” cost function rather than the sum of square residuals (as in ordinary regression). Logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier.

<sup>1</sup> Christopher M. Bishop: Pattern Recognition and Machine Learning, Chapter 7.2.1

<sup>2</sup> David Wipf and Srikantan Nagarajan: A new view of automatic relevance determination.



The `LogisticRegression` class can be used to do L1 or L2 penalized logistic regression. L1 penalization yields sparse predicting weights. 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.

#### Examples:

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

---

#### 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*.

---

### 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 classes `SGDClassifier` and `SGDRegressor` provide functionality to fit linear models for classification and regression using different (convex) loss functions and different penalties.

---

#### References

- *Stochastic Gradient Descent*
- 

### 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.

## 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. Keshet, S. Shalev-Shwartz, Y. Singer - JMLR 7 (2006)

## 1.3.2 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:

- 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 five-fold cross-validation, and thus performance can suffer.

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`.

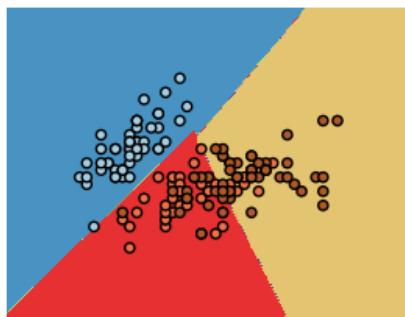
In previous versions of scikit-learn, sparse input support existed only in the `sklearn.svm.sparse` module which duplicated the `sklearn.svm` interface. This module still exists for backward compatibility, but is deprecated and will be removed in scikit-learn 0.12.

### 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

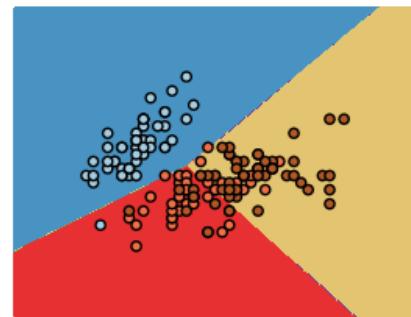
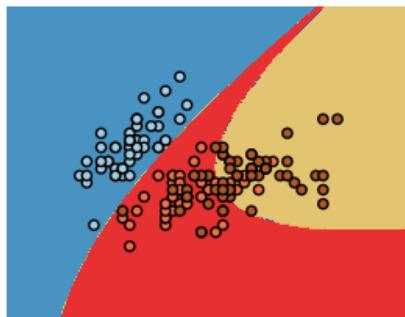
SVC with linear kernel



SVC with RBF kernel



SVC with polynomial (degree 3) kernel    LinearSVC (linear kernel)



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 integer values, size [`n_samples`], holding the class labels for the training 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, 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, 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='l2', 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, on-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 class 0 having three support vectors  $v_0^0, v_0^1, v_0^2$  and class 1 and 2 having two support vectors  $v_1^0, v_1^1$  and  $v_2^0, v_2^1$  respectively. For each support vector  $v_i^j$ , there are two dual coefficients. Let's call the coefficient of support vector  $v_i^j$  in the classifier between classes  $i$  and  $k$   $\alpha_{i,k}^j$ . Then `dual_coef_` looks like this:

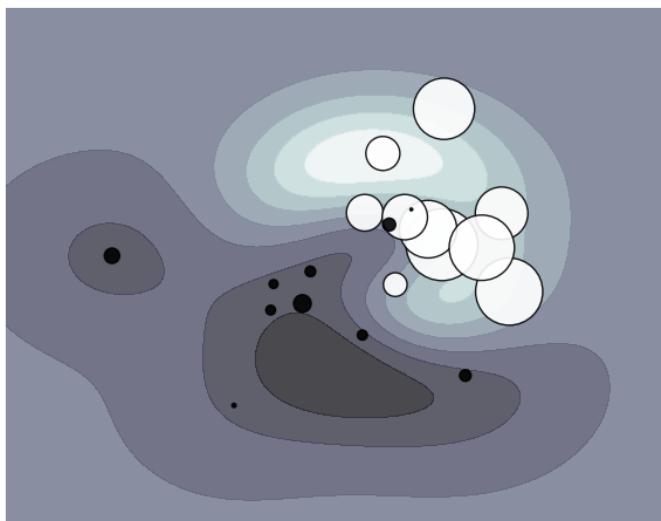
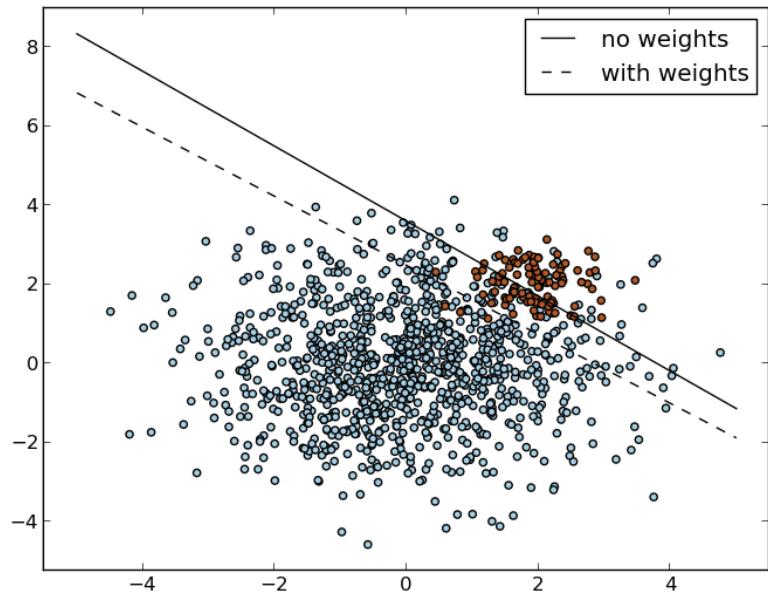
$\alpha_{0,1}^0$	$\alpha_{0,2}^0$	Coefficients for SVs of class 0
$\alpha_{0,1}^1$	$\alpha_{0,2}^1$	
$\alpha_{0,1}^2$	$\alpha_{0,2}^2$	
$\alpha_{1,0}^0$	$\alpha_{1,2}^0$	Coefficients for SVs of class 1
$\alpha_{1,0}^1$	$\alpha_{1,2}^1$	
$\alpha_{2,0}^0$	$\alpha_{2,1}^0$	Coefficients for SVs of class 2
$\alpha_{2,0}^1$	$\alpha_{2,1}^1$	

## 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`.



**Examples:**

- Plot different SVM classifiers in the iris dataset,
- SVM: Maximum margin separating hyperplane,
- SVM: Separating hyperplane for unbalanced classes
- SVM-Anova: SVM with univariate feature selection,
- Non-linear SVM
- SVM: Weighted samples,

**Regression**

The method of Support Vector Classification can be extended to solve regression problems. This method is called Support Vector Regression.

The model produced by support vector classification (as described above) depends only on a subset of the training data, because the cost function for building the model does not care about training points that lie beyond the margin. Analogously, the model produced by Support Vector Regression depends only on a subset of the training data, because the cost function for building the model ignores any training data close to the model prediction.

There are two flavors of Support Vector Regression: SVR and NuSVR.

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, probability=False,
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

**Density estimation, novelty detection**

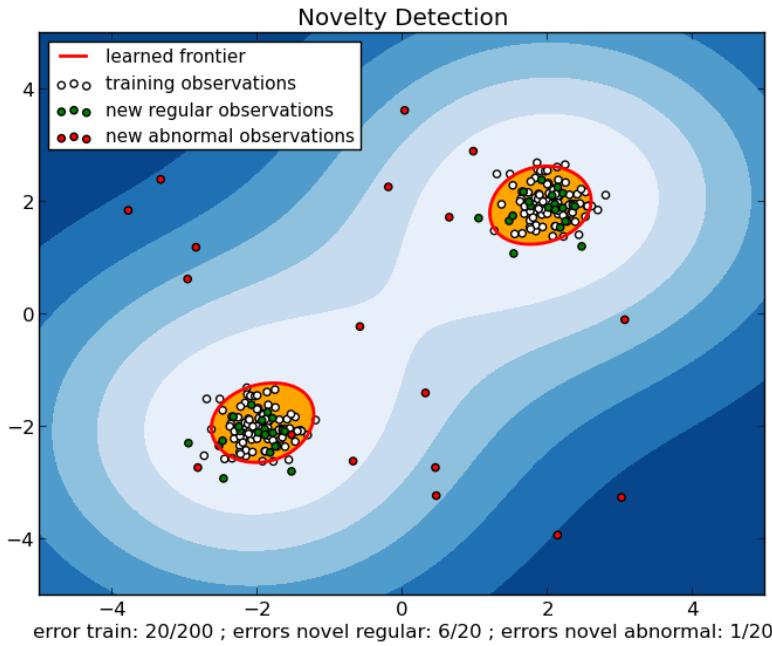
One-class SVM is used for novelty detection, that is, given a set of samples, it will detect the soft boundary of that set so as to classify new points as belonging to that set or not. The class that implements this is called OneClassSVM.

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



## 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  $O(n_{\text{features}} \times n_{\text{samples}}^2)$  and  $O(n_{\text{features}} \times n_{\text{samples}}^3)$  depending on how efficiently the `libsvm` cache is used in practice (dataset dependent). If the data is very sparse  $n_{\text{features}}$  should be replaced by the average number of non-zero features in a sample vector.

Also note that for the linear case, the algorithm used in `LinearSVC` by the `liblinear` implementation is much more efficient than its `libsvm`-based `SVC` counterpart and can scale almost linearly to millions of samples and/or features.

## Tips on Practical Use

- **Avoiding data copy:** For `SVC`, `SVR`, `NuSVC` and `NuSVR`, if the data passed to certain methods is not C-ordered contiguous, and double precision, it will be copied before calling the underlying C implementation. You can check whether a 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`.

## Kernel functions

The *kernel function* can be any of the following:

- linear:  $\langle x_i, x'_j \rangle$ .
- polynomial:  $(\gamma \langle x, x' \rangle + r)^d$ .  $d$  is specified by keyword `degree`,  $r$  by `coef0`.
- rbf ( $\exp(-\gamma|x - x'|^2)$ ,  $\gamma > 0$ ).  $\gamma$  is specified by keyword `gamma`.
- sigmoid ( $\tanh(\langle x_i, x_j \rangle + r)$ ), 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,
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.

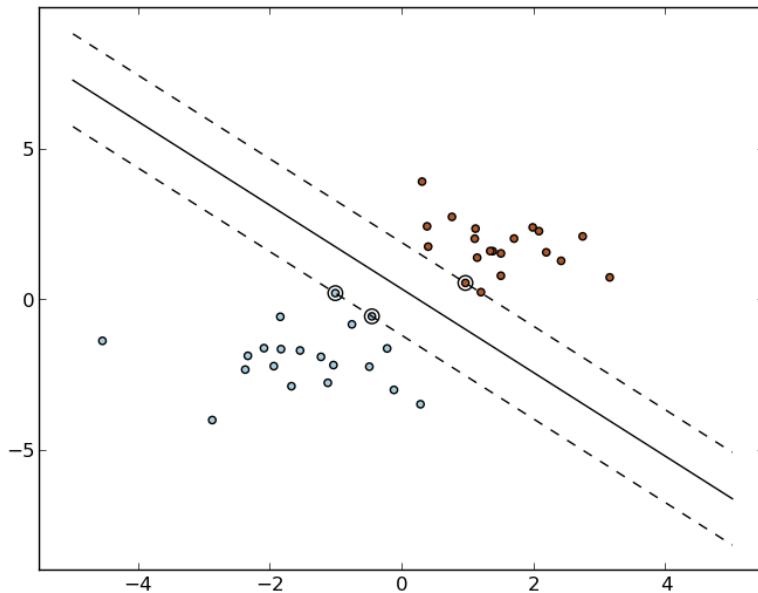
Proper choice of `C` and `gamma` is critical to the SVM's performance. One is advised to use `GridSearchCV` with `C` and `gamma` spaced exponentially far apart to choose good values.

**Examples:**

- RBF SVM parameters

## Mathematical formulation

A support vector machine constructs a hyper-plane or set of hyper-planes in a high or infinite dimensional space, which can be used for classification, regression or other tasks. Intuitively, a good separation is achieved by the hyper-plane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier.



## SVC

Given training vectors  $x_i \in R^p$ ,  $i=1, \dots, n$ , in two classes, and a vector  $y \in R^n$  such that  $y_i \in \{1, -1\}$ , SVC solves the following primal problem:

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1,n} \zeta_i$$

$$\text{subject to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i, \\ \zeta_i \geq 0, i = 1, \dots, n$$

Its dual is

$$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha \\ \text{subject to } y^T \alpha = 0 \\ 0 \leq \alpha_i \leq C, i = 1, \dots, l$$

where  $e$  is the vector of all ones,  $C > 0$  is the upper bound,  $Q$  is an  $n$  by  $n$  positive semidefinite matrix,  $Q_{ij} \equiv K(x_i, x_j)$  and  $\phi(x_i)^T \phi(x)$  is the kernel. Here training vectors are mapped into a higher (maybe infinite) dimensional space by the function  $\phi$ .

The decision function is:

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

**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  $-\rho$ :

#### References:

- “Automatic Capacity Tuning of Very Large VC-dimension Classifiers” I Guyon, B Boser, V Vapnik - Advances in neural information processing 1993,
- “Support-vector networks” C. Cortes, V. Vapnik, Machine Learning, 20, 273-297 (1995)

## NuSVC

We introduce a new parameter  $\nu$  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  $\nu$ -SVC formulation is a reparametrization of the  $C$ -SVC and therefore mathematically equivalent.

#### 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

### 1.3.3 Stochastic Gradient Descent

**Stochastic Gradient Descent (SGD)** is a simple yet very efficient approach to discriminative learning of linear classifiers under convex loss functions such as (linear) `Support Vector Machines` and `Logistic Regression`. Even though SGD has been around in the machine learning community for a long time, it has received a considerable amount of attention just recently in the context of large-scale learning.

SGD has been successfully applied to large-scale and sparse machine learning problems often encountered in text classification and natural language processing. Given that the data is sparse, the classifiers in this module easily scale to problems with more than  $10^5$  training examples and more than  $10^5$  features.

The advantages of Stochastic Gradient Descent are:

- Efficiency.
- Ease of implementation (lots of opportunities for code tuning).

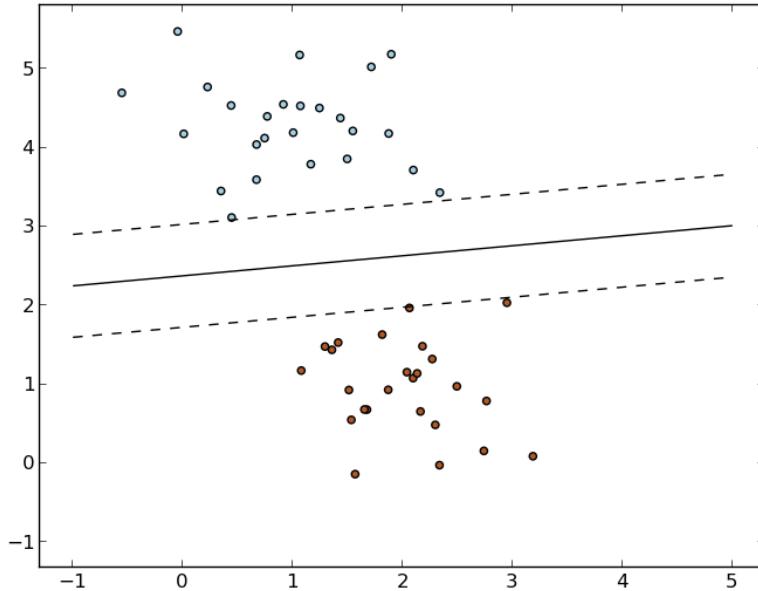
The disadvantages of Stochastic Gradient Descent include:

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

## Classification

**Warning:** Make sure you permute (shuffle) your training data before fitting the model or use `shuffle=True` to shuffle after each 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, 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, rho=None, shuffle=False, 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([[ 9.90090187,  9.90090187]])
```

Member `intercept_` holds the intercept (aka offset or bias):

```
>>> clf.intercept_
array([-9.990...])
```

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([ 29.61357756])
```

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.

In the case of binary classification and `loss="log"` or `loss="modified_huber"` you get a probability estimate  $P(y = C|x)$  using `SGDClassifier.predict_proba`, where  $C$  is the largest class label:

```
>>> clf = SGDClassifier(loss="log").fit(X, y)
>>> clf.predict_proba([[1., 1.]])
array([[ 0.00000051,  0.99999949]])
```

The concrete penalty can be set via the `penalty` parameter. SGD supports the following penalties:

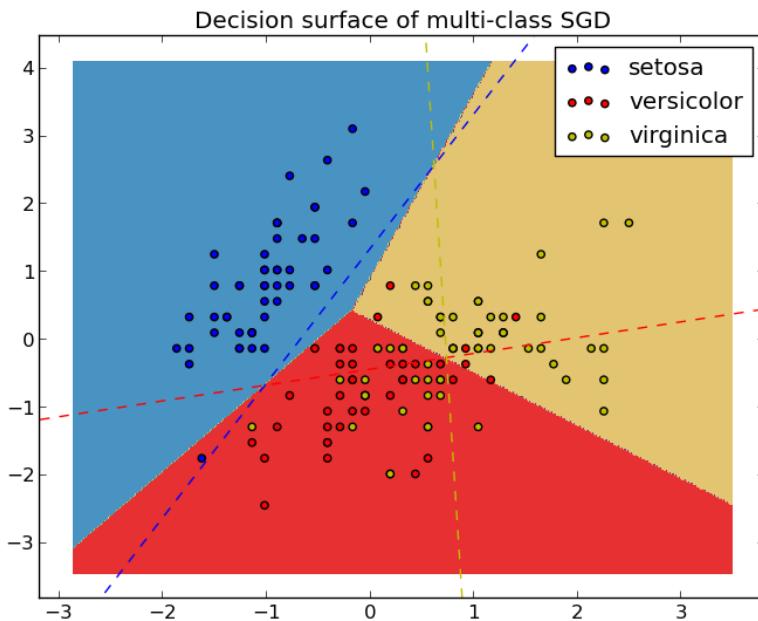
- `penalty="l2"`: L2 norm penalty on `coef_`,
- `penalty="l1"`: L1 norm penalty on `coef_`,
- `penalty="elasticnet"`: Convex combination of L2 and L1;  $\rho * L2 + (1 - \rho) * L1$ .

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 `rho` has to be specified by the user.

`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-dimensionaly 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: Separating hyperplane with weighted classes*
- *SGD: Weighted samples*

## 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.

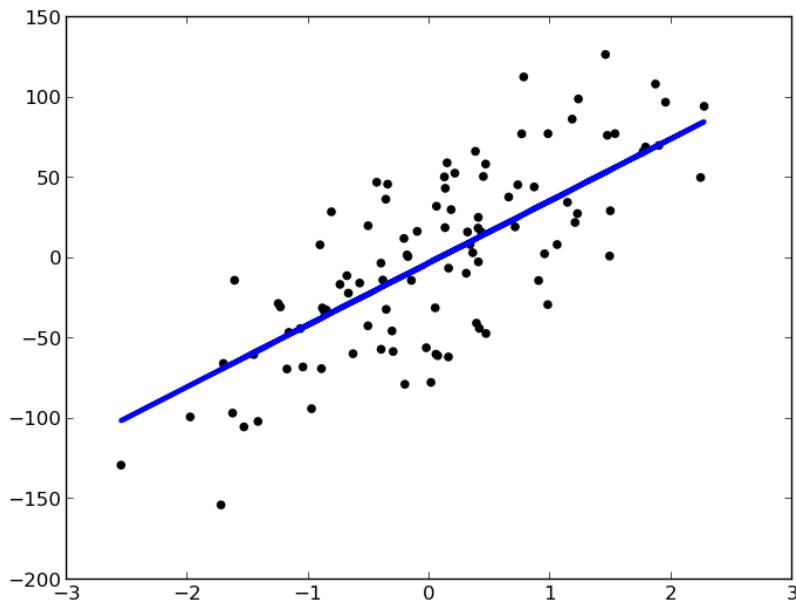
The concrete loss function can be set via the `loss` parameter. `SGDRegressor` supports the following loss functions:

- `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.

#### Examples:

- *Ordinary Least Squares with SGD,*



## Stochastic Gradient Descent for sparse data

---

**Note:** The sparse implementation produces slightly different results than the dense implementation due to a shrunk learning rate for the intercept.

---

There is built-in support for sparse data given in any matrix in a format supported by `scipy.sparse`. For maximum efficiency, however, use the CSR matrix format as defined in [scipy.sparse.csr\\_matrix](#).

### Examples:

- *Classification of text documents using sparse features*

## Complexity

The major advantage of SGD is its efficiency, which is basically linear in the number of training examples. If  $X$  is a matrix of size  $(n, p)$  training has a cost of  $O(knp)$ , 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.

## 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  $\alpha$  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  $n_{\text{iter}} = np.ceil(10^{**6} / n)$ , 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.

## References:

- “Efficient BackProp” Y. LeCun, L. Bottou, G. Orr, K. Müller - In Neural Networks: Tricks of the Trade 1998.

## Mathematical formulation

Given a set of training examples  $(x_1, y_1), \dots, (x_n, y_n)$  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

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

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

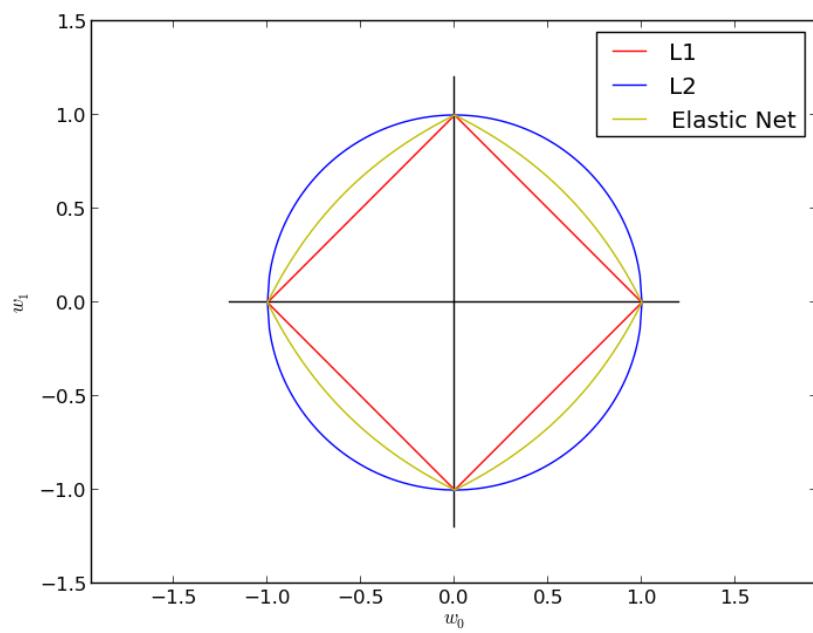
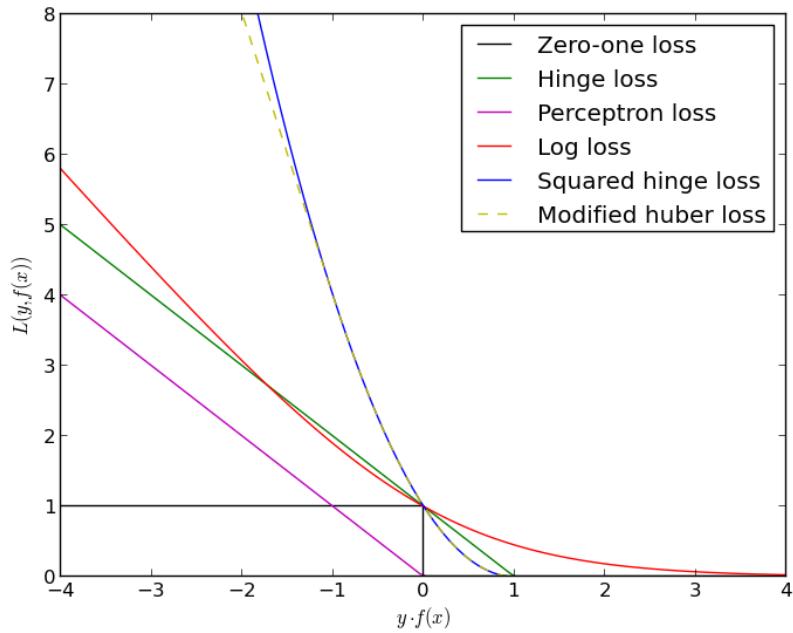
- Hinge: (soft-margin) Support Vector Machines.
- Log: Logistic Regression.
- Least-Squares: Ridge Regression.
- Epsilon-Insensitive: (soft-margin) Support Vector Regression.

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:  $R(w) := \rho \frac{1}{2} \sum_{i=1}^n w_i^2 + (1 - \rho) \sum_{i=1}^n |w_i|$ , a convex combination of L2 and L1.

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

$$w \leftarrow w - \eta \left( \alpha \frac{\partial R(w)}{\partial w} + \frac{\partial L(w^T x_i + b, y_i)}{\partial w} \right)$$

where  $\eta$  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  $\eta$  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.

### 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.

### 1.3.4 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. 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. Arbitrary Minkowski metrics are supported for searches.

#### Unsupervised Nearest Neighbors

`NearestNeighbors` implements unsupervised nearest neighbors learning. It acts as a uniform interface to three different nearest neighbors algorithms: `BallTree`, `scipy.spatial.cKDTree`, 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.

#### Nearest Neighbors Classification

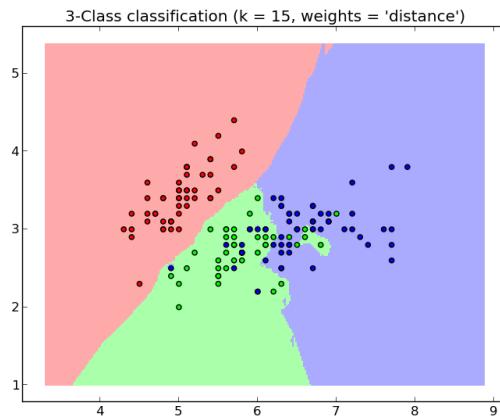
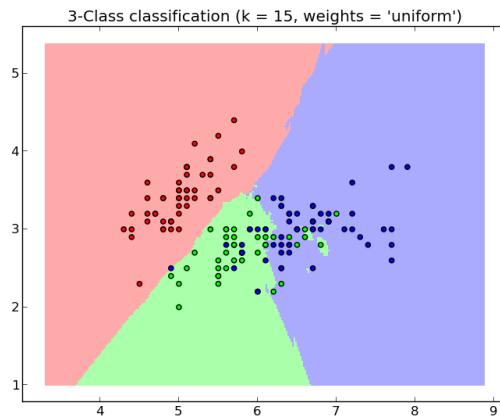
Neighbors-based classification is a type of *instance-based learning* or *non-generalizing learning*: it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the nearest neighbors of each point: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

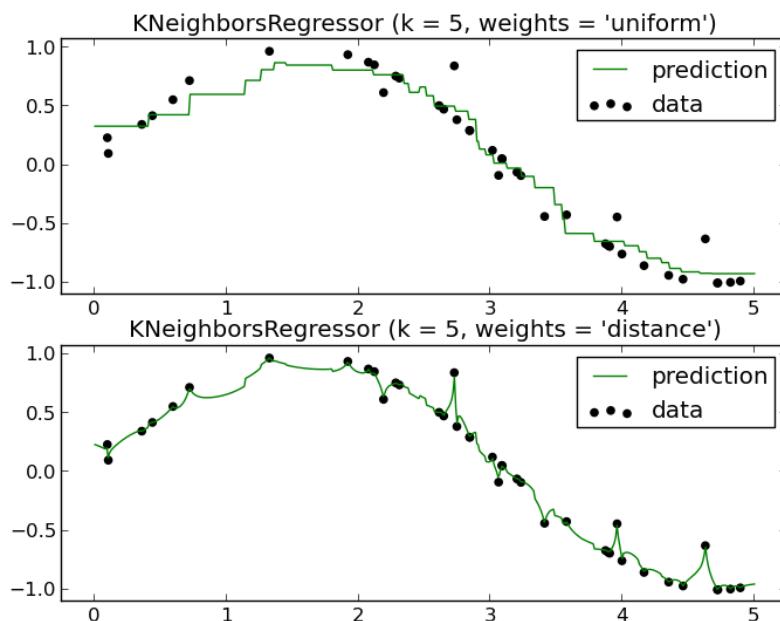
- *Nearest Neighbors Classification*: an example of classification using nearest neighbors.

## 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.



### Examples:

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

## 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  $N$ .

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 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 `scipy.spatial.cKDTree`.

### 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 allows for efficient neighbors searches 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, its performance does not degrade at high dimensions. 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 `cKDTree` 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 nonlinearly 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 '`ball_tree`' if  $k < N/2$ , and '`brute`' otherwise. 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  $1 / \text{leaf\_size}$  times the size of the training set.

`leaf_size` is not referenced for brute force queries.

## Nearest Centroid Classifier

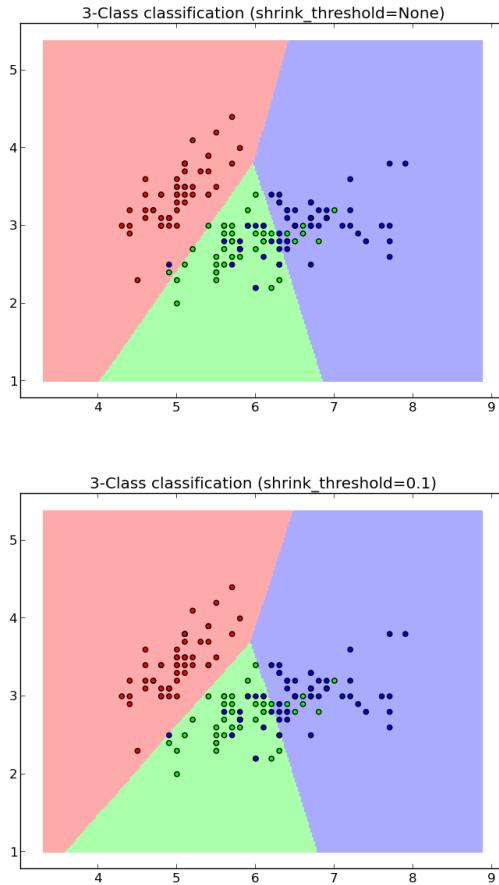
The `NearestCentroid` classifier is a simple algorithm that represents each class by the centroid of its members. In effect, this makes it similar to the label updating phase of the `sklearn.KMeans` algorithm. It also has no parameters to choose, making it a good baseline classifier. It does, however, 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.

### 1.3.5 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 exceedence 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.

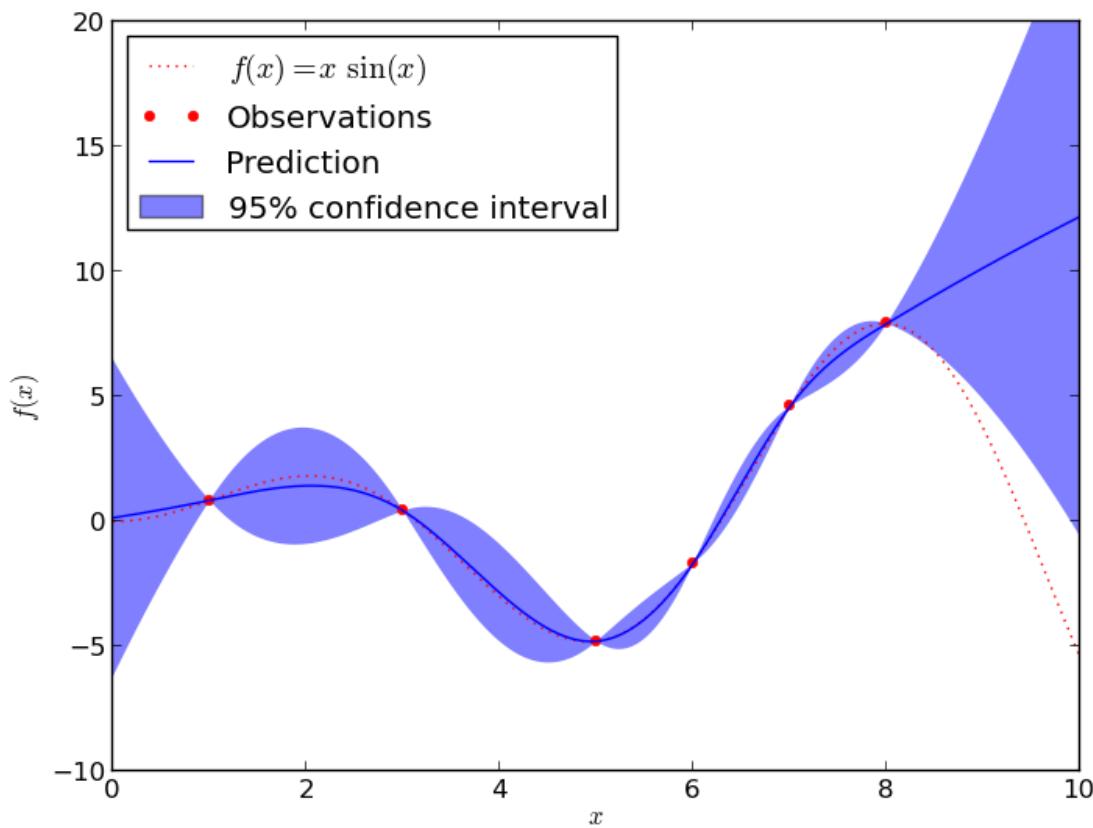
- It loses efficiency in high dimensional spaces – namely when the number of features exceeds a few dozens. It might indeed give poor performance and it loses computational efficiency.
- Classification is only a post-processing, meaning that one first need to solve a regression problem by providing the 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.

## 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)
>>> 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

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

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*

## Mathematical formulation

### The initial assumption

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

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

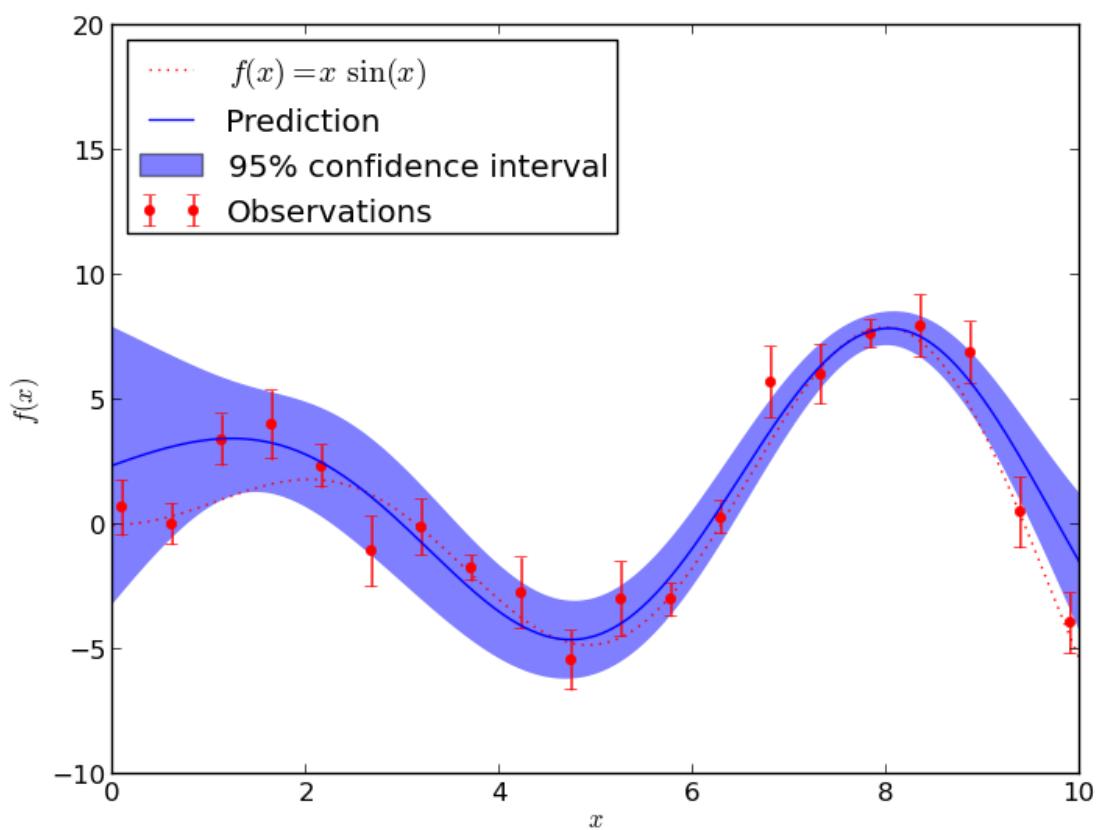
GPML starts with the assumption that this function is a conditional sample path of a Gaussian process  $G$  which is additionally assumed to read as follows:

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

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:

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

$\sigma^2$  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:

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

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:

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

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  $\theta$ :

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

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

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

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

$$F_{ij} = f_i(X_j), \quad i = 1, \dots, p, \quad j = 1, \dots, m$$

- the generalized least square regression weights:

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

- and the vectors:

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

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 Dietrich)
- [The design and analysis of computer experiments](#) TJ Santner, BJ Williams, W Notz Springer, 2003

### 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.

## 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*.

### Implementation details

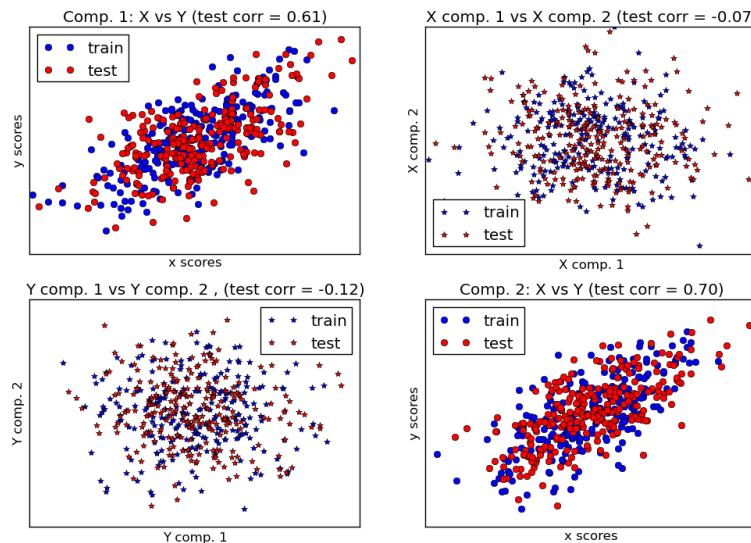
The present implementation is based on a translation of the DACE Matlab toolbox.

#### 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.

## 1.3.6 Partial Least Squares

Partial least squares (PLS) models are useful to find linear relations between two multivariate datasets: in PLS the  $X$  and  $Y$  arguments of the *fit* method are 2D arrays.



PLS finds the fundamental relations between two matrices (X and Y): it is a latent variable approach to modeling the covariance structures in these two spaces. A PLS model 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:

- PLS Partial Least Squares*

### 1.3.7 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:

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

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

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

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

$$\begin{aligned} P(y | x_1, \dots, x_n) &\propto P(y) \prod_{i=1}^n P(x_i | y) \\ &\Downarrow \\ \hat{y} &= \arg \max_y P(y) \prod_{i=1}^n P(x_i | y), \end{aligned}$$

and we can use Maximum A Posteriori (MAP) estimation to estimate  $P(y)$  and  $P(x_i | y)$ ; 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  $P(x_i | y)$ .

In spite of their apparently over-simplified assumptions, Naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering. They require a small amount of training data to estimate the necessary parameters. (For theoretical reasons why Naive Bayes works well, and on which types of data it does, see the references below.)

Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods. The decoupling of the class conditional feature distributions means that each distribution can be independently estimated as a one dimensional distribution. This in turn helps to alleviate problems stemming from the curse of dimensionality.

On the flip side, although Naive Bayes is known as a decent classifier, it is known to be a bad estimator, so the probability outputs from `predict_proba` are not to be taken too seriously.

### References:

- H. Zhang (2004). The optimality of Naive Bayes. Proc. FLAIRS.

## Gaussian Naive Bayes

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

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

The parameters  $\sigma_y$  and  $\mu_y$  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 : %d" % (iris.target != y_pred).sum()
Number of mislabeled points : 6
```

## 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  $\theta_y = (\theta_{y1}, \dots, \theta_{yn})$  for each class  $y$ , where  $n$  is the number of features (in text classification, the size of the vocabulary) and  $\theta_{yi}$  is the probability  $P(x_i | y)$  of feature  $i$  appearing in a sample belonging to class  $y$ .

The parameters  $\theta_y$  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  $\alpha \geq 0$  accounts for features not present in the learning samples and prevents zero probabilities in further computations. Setting  $\alpha = 1$  is called Laplace smoothing, while  $\alpha < 1$  is called Lidstone smoothing.

## 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

$$P(x_i | y) = P(i | y)x_i \times (1 - P(i | y))(1 - x_i)$$

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.

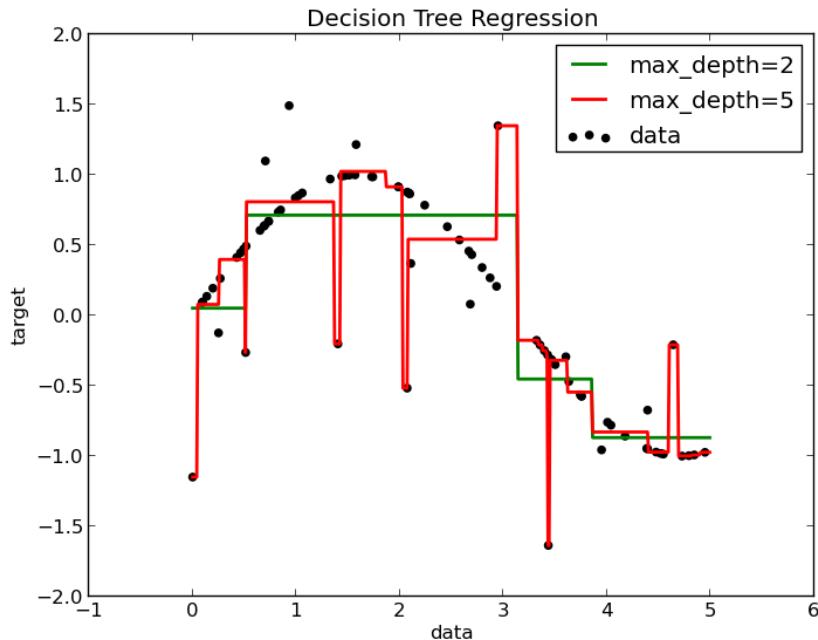
#### References:

- 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. Androultsopoulos and G. Palioras (2006). Spam filtering with Naive Bayes – Which Naive Bayes? 3rd Conf. on Email and Anti-Spam (CEAS).

### 1.3.8 Decision Trees

**Decision Trees (DTs)** are a non-parametric supervised learning method used for *classification* and *regression*. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

For instance, in the example below, decision trees learn from data to approximate a sine curve with a set of if-then-else decision rules. The deeper the tree, the more complex the decision rules and the fitter the model.



Some advantages of decision trees are:

- Simple to understand and to interpret. Trees can be visualised.
- Requires little data preparation. Other techniques often require data normalisation, dummy variables need to be created and blank values to be removed. Note however that this module does not support missing values.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data. Other techniques are usually specialised in analysing datasets that have only one type of variable. See *algorithms* for more information.
- Able to handle multi-output problems.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
- Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

The disadvantages of decision trees include:

- Decision-tree learners can create over-complex trees that do not generalise the data well. This is called overfitting. Mechanisms such as pruning (not currently supported), setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.
- The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.
- There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

## Classification

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

As other classifiers, `DecisionTreeClassifier` take as input two arrays: an array `X` 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 new values:

```
>>> clf.predict([[2., 2.]])
array([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:

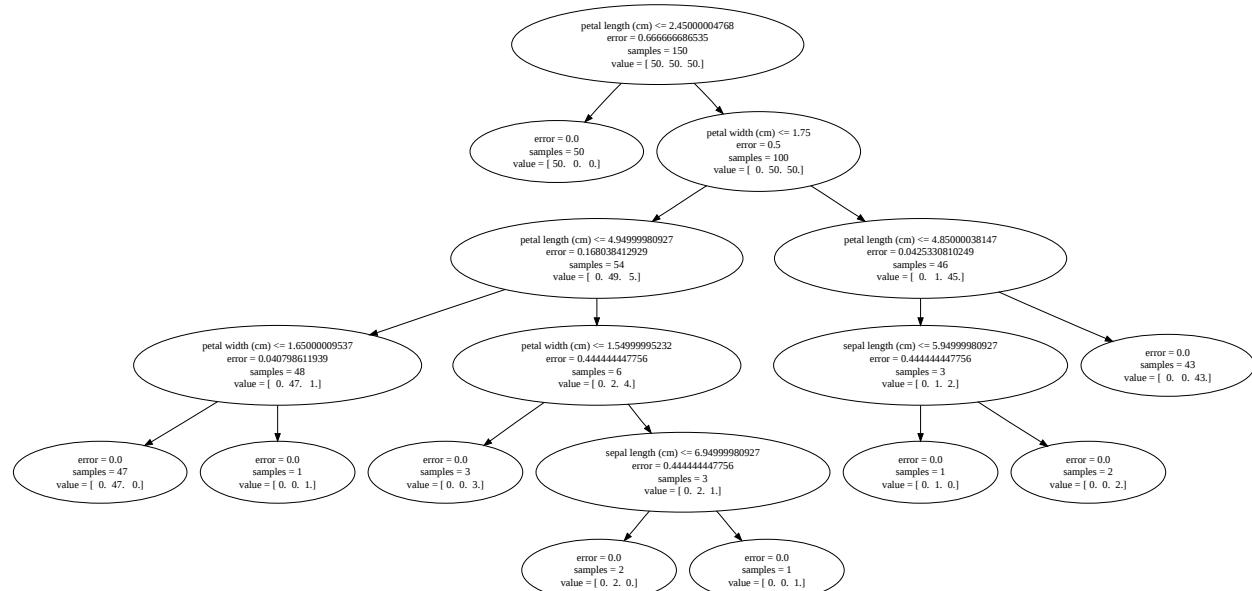
```
>>> import StringIO
>>> with open("iris.dot", 'w') as f:
...     f = tree.export_graphviz(clf, out_file=f)
```

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')
```

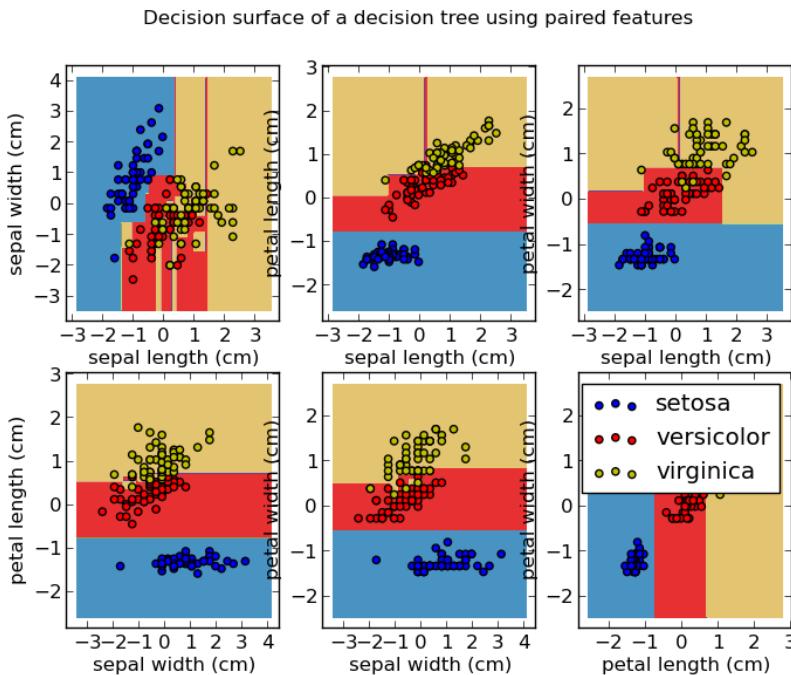
Alternatively, if we have Python module `pydot` installed, we can generate a PDF file (or any other supported file type) directly in Python:

```
>>> import StringIO, pydot
>>> dot_data = StringIO.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 new values:

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

**Examples:**

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

## Regression

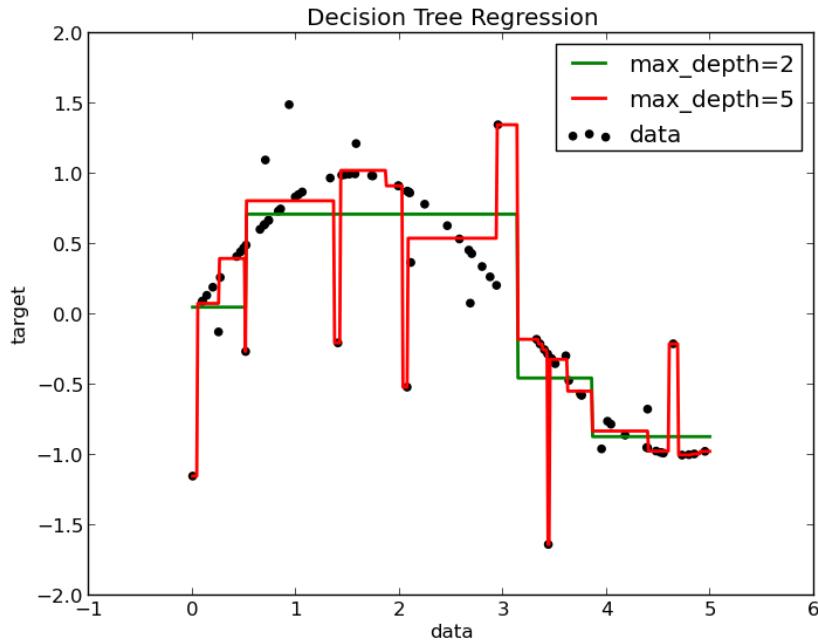
Decision trees can also be applied to regression problems, using the `DecisionTreeRegressor` class.

As in the classification setting, the `fit` method will take as argument arrays `X` and `y`, only that in this case `y` is expected to have floating point values instead of integer values:

```
>>> from sklearn import tree
>>> X = [[0, 0], [2, 2]]
>>> y = [0.5, 2.5]
>>> clf = tree.DecisionTreeRegressor()
>>> clf = clf.fit(X, y)
>>> clf.predict([[1, 1]])
array([ 0.5])
```

**Examples:**

- Decision Tree Regression



## Multi-output problems

A multi-output problem is a supervised learning problem with several outputs to predict, that is when  $Y$  is a 2d array of size  $[n\_samples, n\_outputs]$ .

When there is no correlation between the outputs, a very simple way to solve this kind of problem is to build  $n$  independent models, i.e. one for each output, and then to use those models to independently predict each one of the  $n$  outputs. However, because it is likely that the output values related to the same input are themselves correlated, an often better way is to build a single model capable of predicting simultaneously all  $n$  outputs. First, it requires lower training time since only a single estimator is built. Second, the generalization accuracy of the resulting estimator may often be increased.

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

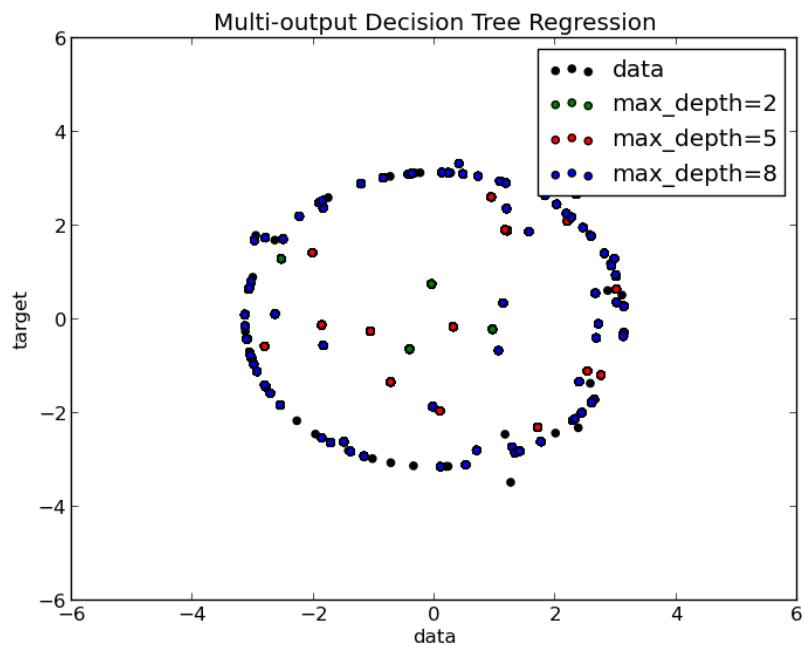
- Store  $n$  output values in leaves, instead of 1;
- Use splitting criteria that compute the average reduction across all  $n$  outputs.

This module offers support for multi-output problems by implementing this strategy in both `DecisionTreeClassifier` and `DecisionTreeRegressor`. If a decision tree is fit on an output array  $Y$  of size  $[n\_samples, n\_outputs]$  then the resulting estimator will:

- Output  $n\_output$  values upon `predict`;
- Output a list of  $n\_output$  arrays of class probabilities upon `predict_proba`.

The use of multi-output trees for regression is demonstrated in *Multi-output Decision Tree Regression*. In this example, the input  $X$  is a single real value and the outputs  $Y$  are the sine and cosine of  $X$ .

The use of multi-output trees for classification is demonstrated in *Face completion with multi-output forests*. 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.



Face completion with multi-output forests



**Examples:**

- Multi-output Decision Tree Regression
- Face completion with multi-output forests

**References:**

- M. Dumont et al, [Fast multi-class image annotation with random subwindows and multiple output randomized trees](#), International Conference on Computer Vision Theory and Applications 2009

**Complexity**

In general, the run time cost to construct a balanced binary tree is  $O(n_{samples}n_{features}\log(n_{samples}))$  and query time  $O(\log(n_{samples}))$ . Although the tree construction algorithm attempts to generate balanced trees, they will not 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  $O(n_{features}n_{samples}\log(n_{samples}))$ .

This implementation also offers a parameter `min_density` to control an optimization heuristic. A sample mask is used to mask data points that are inactive at a given node, which avoids the copying of data (important for large datasets or training trees within an ensemble). Density is defined as the ratio of ‘active’ data samples to total samples at a given node. The minimum density parameter specifies the level below which fancy indexing (and therefore data copied) and the sample mask reset. If `min_density` is 1, then fancy indexing is always used for data partitioning during the tree building phase. In this case, the size of memory (as a proportion of the input data  $a$ ) required at a node of depth  $n$  can be approximated using a geometric series:  $size = a \frac{1-r^n}{1-r}$  where  $r$  is the ratio of samples used at each node. A best case analysis shows that the lowest memory requirement (for an infinitely deep tree) is  $2 \times a$ , where each partition divides the data in half. A worst case analysis shows that the memory requirement can increase to  $n \times a$ . In practise it usually requires 3 to 4 times  $a$ . Setting `min_density` to 0 will always use the sample mask to select the subset of samples at each node. This results in little to no additional memory being allocated, making it appropriate for massive datasets or within ensemble learners. The default value for `min_density` is 0.1 which empirically leads to fast training for many problems. Typically high values of `min_density` will lead to excessive reallocation, slowing down the algorithm significantly.

**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 creating a tree biased toward the classes that are dominant.
- All decision trees use Fortran ordered `np.float32` arrays internally. If training data is not in this format, a copy of the dataset will be made.

## 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.

## Mathematical formulation

Given training vectors  $x_i \in R^n$ ,  $i=1, \dots, l$  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

$$\begin{aligned} Q_{left}(\theta) &= (x, y) | x_j \leq t_m \\ Q_{right}(\theta) &= Q \setminus Q_{left}(\theta) \end{aligned}$$

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/Decision_tree_learning)
- [http://en.wikipedia.org/wiki/Predictive\\_analytics](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.

### 1.3.9 Ensemble methods

The goal of **ensemble methods** is to combine the predictions of several models built with a given learning algorithm in order to improve generalizability / robustness over a single model.

Two families of ensemble methods are usually distinguished:

- In **averaging methods**, the driving principle is to build several models independently and then to average their predictions. On average, the combined model is usually better than any of the single model because its variance is reduced.

**Examples:** Bagging methods, *Forests of randomized trees*...

- By contrast, in **boosting methods**, models are built sequentially and one tries to reduce the bias of the combined model. The motivation is to combine several weak models to produce a powerful ensemble.

**Examples:** AdaBoost, Gradient Tree Boosting, ...

#### 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: 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.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:

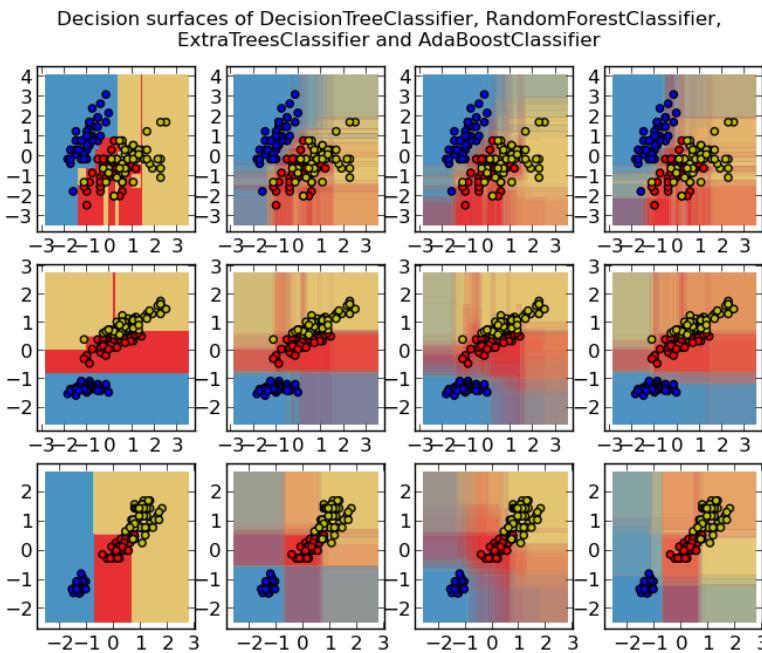
```
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.datasets import make_blobs
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.tree import DecisionTreeClassifier

>>> 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). The best results are also usually reached 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. The best parameter values should always be cross-validated. In addition, note that bootstrap samples are used by default in random forests (`bootstrap=True`) while the default strategy is to use the original dataset for building extra-trees (`bootstrap=False`).

When training on large datasets, where runtime and memory requirements are important, it might also be beneficial to adjust the `min_density` parameter, that controls a heuristic for speeding up computations in each tree. See *Complexity of trees* for details.

## 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 multi-output forests*

### References

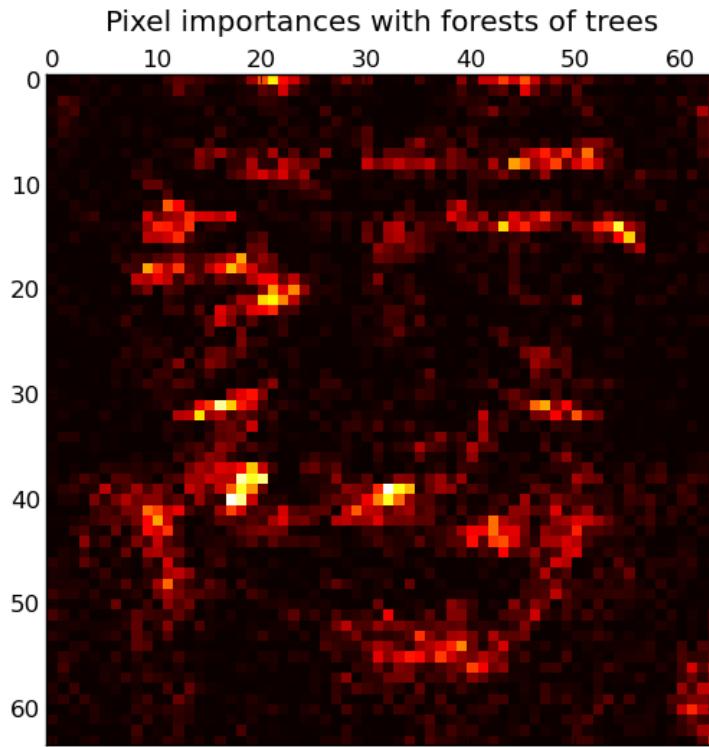
## Feature importance evaluation

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 can be computed by explicitly passing `compute_importances=True` to the constructor of the decision trees, random forest and extremely randomized trees models. The result is 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.

## AdaBoost

The module `sklearn.ensemble` implements the popular boosting algorithm known as AdaBoost. This algorithm was first introduced by Freud and Schapire [FS1995] back in 1995.

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  $w_1, w_2, \dots, w_N$  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 boosting 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 [HTF2009].

AdaBoost can be used both for classification and regression problems:

- For multi-class classification, `AdaBoostClassifier` implements AdaBoost-SAMME [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()
0.92...
```

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.

**Examples:**

- *example\_ensemble\_plot\_adaboost\_classification.py*
- *Boosted Decision Tree Regression*
- *example\_ensemble\_plot\_adaboost\_error.py*

**References**

## 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 input 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 via the deviance loss function (`loss='deviance'`). 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 maximum depth of each tree is controlled via `max_depth`. The `learning_rate` is a hyper-parameter in the range (0.0, 1.0] that controls overfitting via *shrinkage*.

---

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

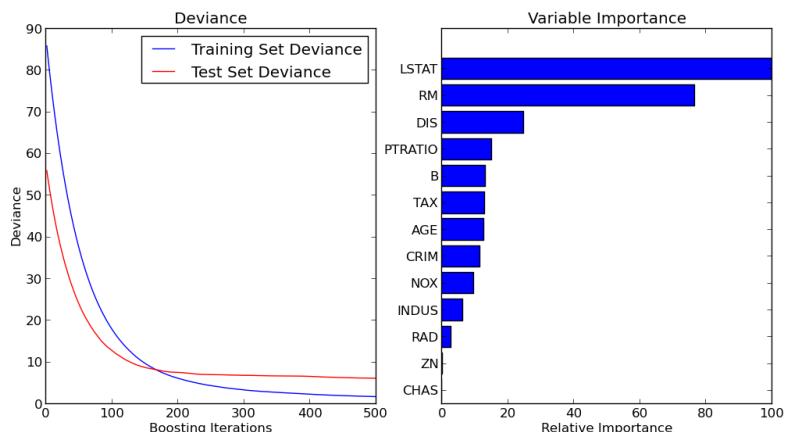
## Regression

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:]
>>> clf = GradientBoostingRegressor(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0, loss='ls').fit(X_train, y_train)
>>> mean_squared_error(y_test, clf.predict(X_test))
6.90...
```

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*

### 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  $\gamma_m$  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 w.r.t. outliers (see [F2001] for more details).
  - Quantile ('quantile'): A loss function for quantile regression. Use  $0 < \text{alpha} < 1$  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.

## Regularization

**Shrinkage** [F2001] proposed a simple regularization strategy that scales the contribution of each weak learner by a factor  $\nu$ :

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

The parameter  $\nu$  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.

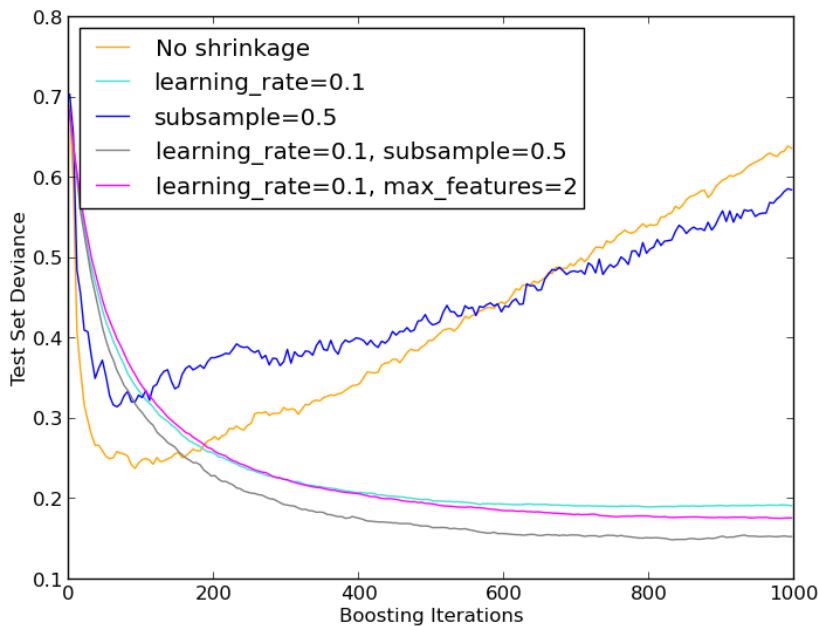
The figure below illustrates the effect of shrinkage and subsampling on the goodness-of-fit of the model. We can clearly see that shrinkage outperforms no-shrinkage. Subsampling with shrinkage can further increase the accuracy of the model. Subsampling without shrinkage, on the other hand, does poorly.

For `subsample < 1`, the deviance on the out-of-bag samples in the  $i$ -the iteration is stored in the attribute `oob_score_[i]`. Out-of-bag estimates can be used for model selection (e.g. to determine the optimal number of iterations).

Another strategy to reduce the variance is by subsampling the features analogous to the random splits in Random Forests. The size of the subsample can be controled via the `max_features` parameter.

### Examples:

- *Gradient Boosting regularization*



## 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, ...]
```

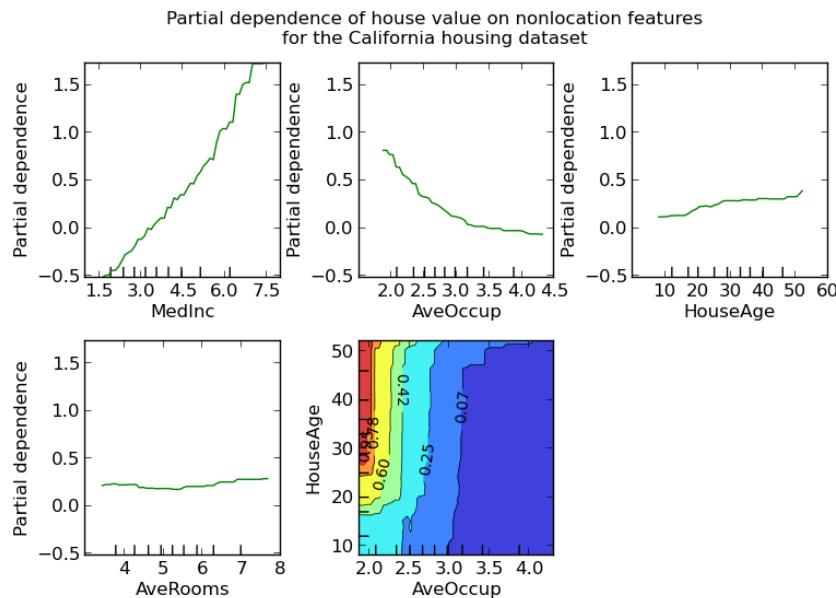
**Examples:**

- 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 <sup>3</sup> as a function of the ‘target’ features <sup>4</sup>.

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
```

<sup>3</sup> For classification with `loss='deviance'` the target response is  $\text{logit}(p)$ .

<sup>4</sup> More precisely its the expectation of the target response after accounting for the initial model; partial dependence plots do not include the init model.

```
>>> X, y = make_hastie_10_2(random_state=0)
>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X, y)
>>> features = [0, 1, (0, 1)]
>>> 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

## 1.3.10 Multiclass and multilabel algorithms

This module implements multiclass and multilabel learning algorithms:

- one-vs-the-rest / one-vs-all
- one-vs-one
- error correcting output codes

Multiclass classification means classification with more than two classes. Multilabel classification is a different task, where a classifier is used to predict a set of target labels for each instance; i.e., the set of target classes is not assumed to be disjoint as in ordinary (binary or multiclass) classification. This is also called any-of classification.

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.

---

**Note:** You don't need to use these estimators unless you want to experiment with different multiclass strategies: all classifiers in scikit-learn support multiclass classification out-of-the-box. Below is a summary of the classifiers supported in scikit-learn grouped by the strategy used.

- Inherently multiclass: *Naive Bayes*, `sklearn.lda.LDA`, *Decision Trees*, *Random Forests*
  - One-Vs-One: `sklearn.svm.SVC`.
  - One-Vs-All: `sklearn.svm.LinearSVC`, `sklearn.linear_model.LogisticRegression`,  
`sklearn.linear_model.SGDClassifier`, `sklearn.linear_model.RidgeClassifier`.
- 

**Note:** At the moment there are no evaluation metrics implemented for multilabel learnings.

---

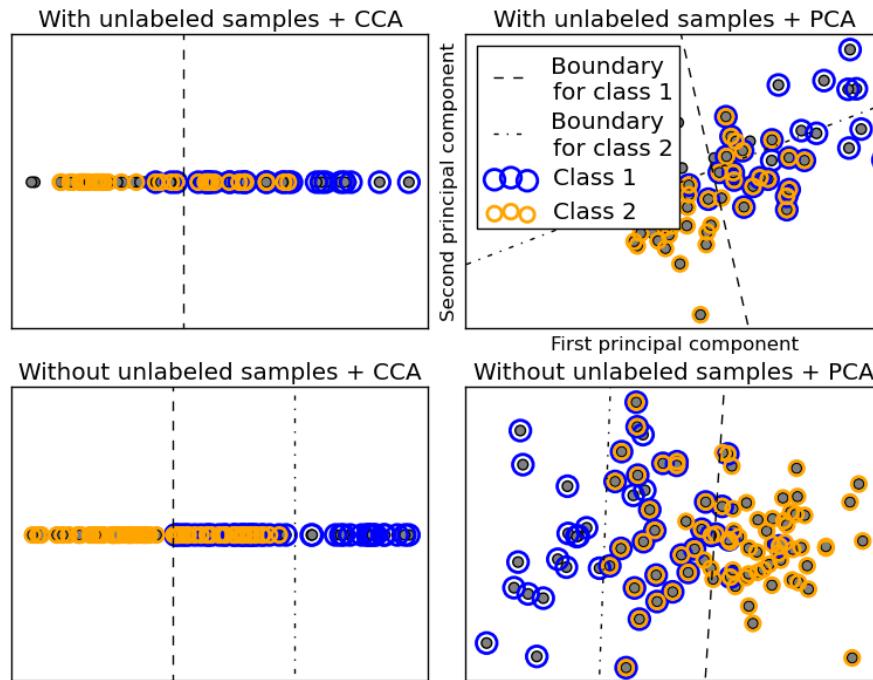
## One-Vs-The-Rest

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. Below is an example:

```
>>> 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, 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, 2, 2, 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])
```

## Multilabel learning with OvR

`OneVsRestClassifier` also supports multilabel classification. To use this feature, feed the classifier a list of tuples containing target labels, like in the example below.

**Examples:**

- *Multilabel classification*

**One-Vs-One**

`OneVsOneClassifier` constructs one classifier per pair of classes. At prediction time, the class which received the most votes is selected. Since it requires to fit  $n_{\text{classes}} * (n_{\text{classes}} - 1) / 2$  classifiers, this method is usually slower than one-vs-the-rest, due to its  $O(n_{\text{classes}}^2)$  complexity. However, this method may be advantageous for algorithms such as kernel algorithms which don't scale well with  $n_{\text{samples}}$ . 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  $n_{\text{classes}}$  times. Below is an example:

```
>>> 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, 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, 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,
```

## 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 <sup>5</sup> 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,  $\log_2(n_{\text{classes}}) / n_{\text{classes}}$  is sufficient to represent each class unambiguously. However, in practice, it may not lead to good accuracy since  $\log_2(n_{\text{classes}})$  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.

Example:

```
>>> 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, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1,
       1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 2, 2, 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,
```

### References:

#### 1.3.11 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.

<sup>5</sup> “The error coding method and PICTs”, James G., Hastie T., Journal of Computational and Graphical statistics 7, 1998.

## Univariate feature selection

Univariate feature selection works by selecting the best features based on univariate statistical tests. It can 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 percentile of features
- using common univariate statistical tests for each feature: false positive rate `SelectFpr`, false discovery rate `SelectFdr`, or family wise error `SelectFwe`.

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*

## 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.

## 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.

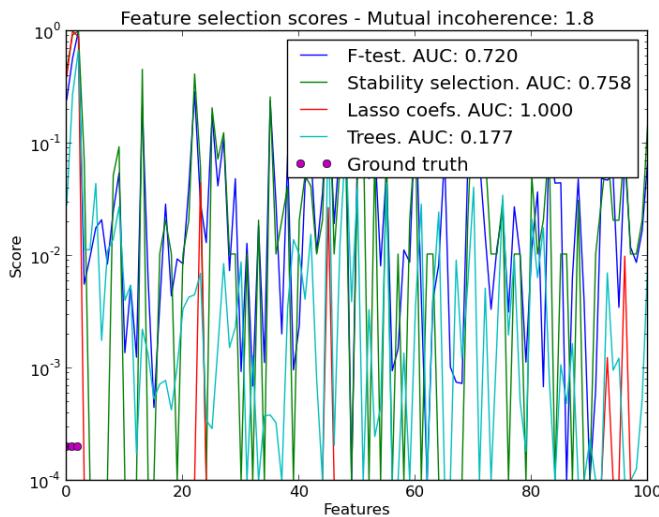
There is no general rule to select an alpha parameter for recovery of non-zero coefficients. It can be set by cross-validation (`LassoCV` or `LassoLarsCV`), though this may lead to under-penalized models: including a small number of non-relevant variables is not detrimental to prediction score. BIC (`LassoLarsIC`) 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. Bühlmann, “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/>

### 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(compute_importances=True, random_state=0)
>>> X_new = clf.fit(X, y).transform(X)
>>> X_new.shape
(150, 2)
```

**Examples:**

- *Feature importances with forests of trees*: example on synthetic data showing the recovery of the actually meaningful features.
- *Pixel importances with a parallel forest of trees*: example on face recognition data.

### 1.3.12 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 addition unlabeled data to capture better 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 the integer value `-1`.

### Label Propagation

Label propagation denote 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.

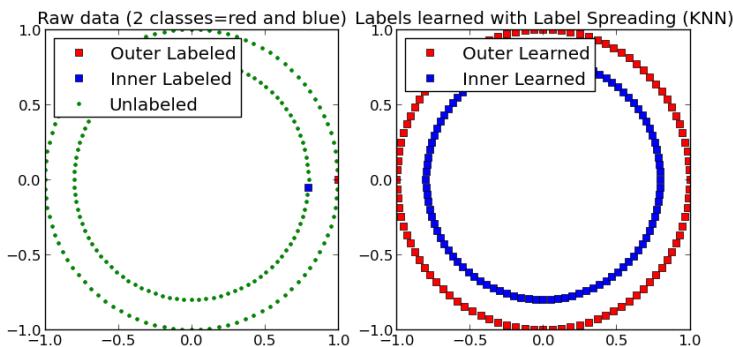


Figure 1.1: **An illustration of label-propagation:** the structure of unlabeled observations is consistent with the class structure, and thus the class label can be propagated to the unlabeled observations of the training set.

`LabelPropagation` and `LabelSpreading` differ in modifications to the similarity matrix that graph and the clamping effect on the label distributions. Clamping allows the algorithm to change the weight of the true ground labeled data to some degree. The `LabelPropagation` algorithm performs hard clamping of input labels, which means  $\alpha = 1$ . 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 its confidence of the distribution within 20 percent.

`LabelPropagation` uses the raw similarity matrix constructed from the data with no modifications. In contrast, `LabelSpreading` minimizes a loss function that has regularization properties, as such it is often more robust to noise. The algorithm iterates on a modified version of the original graph and normalizes the edge weights by computing the normalized graph Laplacian matrix. This procedure is also used in *Spectral clustering*.

Label propagation models have two built-in kernel methods. Choice of kernel effects both scalability and performance of the algorithms. The following are available:

- `rbf` ( $\exp(-\gamma|x - y|^2)$ ,  $\gamma > 0$ ).  $\gamma$  is specified by keyword `gamma`.
- `knn` ( $1[x' \in kNN(x)]$ ).  $k$  is specified by keyword `n_neighbors`.

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*
- *Decision boundary of label propagation versus SVM on the Iris dataset*
- *Label Propagation digits active learning*

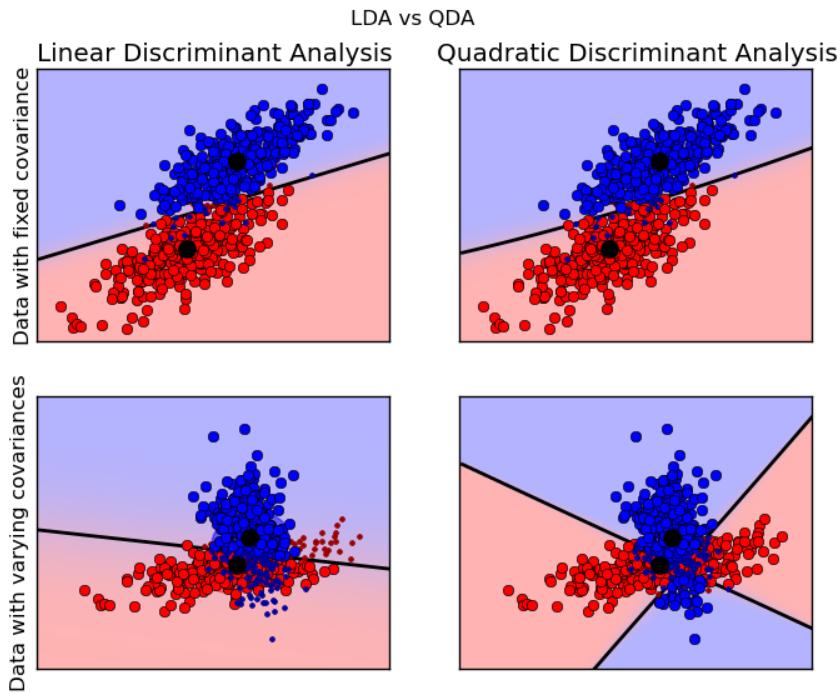
### References

- [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](http://research.microsoft.com/en-us/people/nicolasl/efficient_ssl.pdf)

### 1.3.13 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 multi-class, 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.

#### References:

### 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`.

### 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 modeled 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 log-probability ratios  $\log[P(y = k|X)/P(y = l|X)]$ .

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

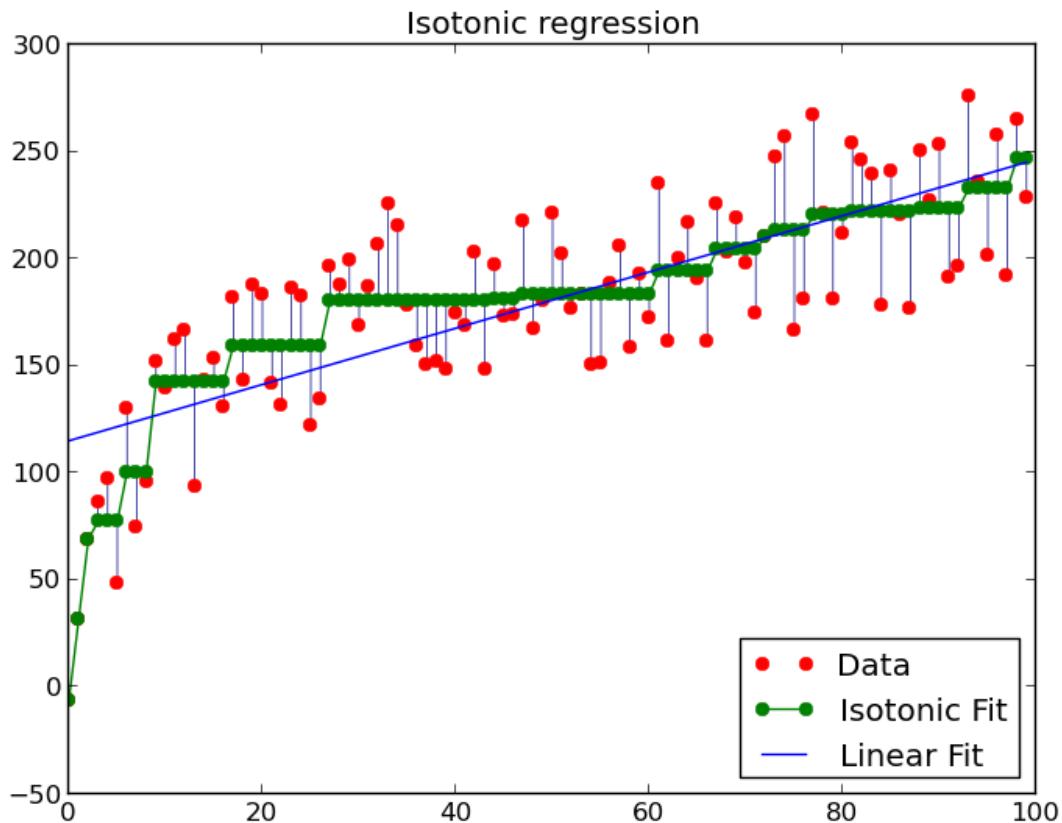
### 1.3.14 Isotonic regression

The class `IsotonicRegression` fits a non-decreasing function to data. It solves the following problem:

$$\text{minimize } \sum_i w_i (y_i - \hat{y}_i)^2$$

$$\text{subject to } \hat{y}_{\min} = \hat{y}_1 \leq \hat{y}_2 \dots \leq \hat{y}_n = \hat{y}_{\max}$$

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.



## 1.4 Unsupervised learning

### 1.4.1 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.

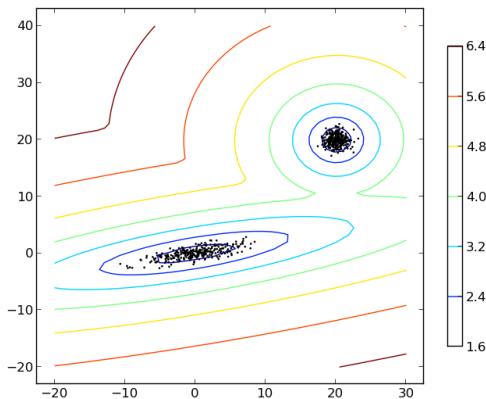


Figure 1.2: Two-component Gaussian mixture model: data points, and equi-probability surfaces of the model.

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.

The `scikit-learn` implements different classes to estimate Gaussian mixture models, that correspond to different estimation strategies, detailed below.

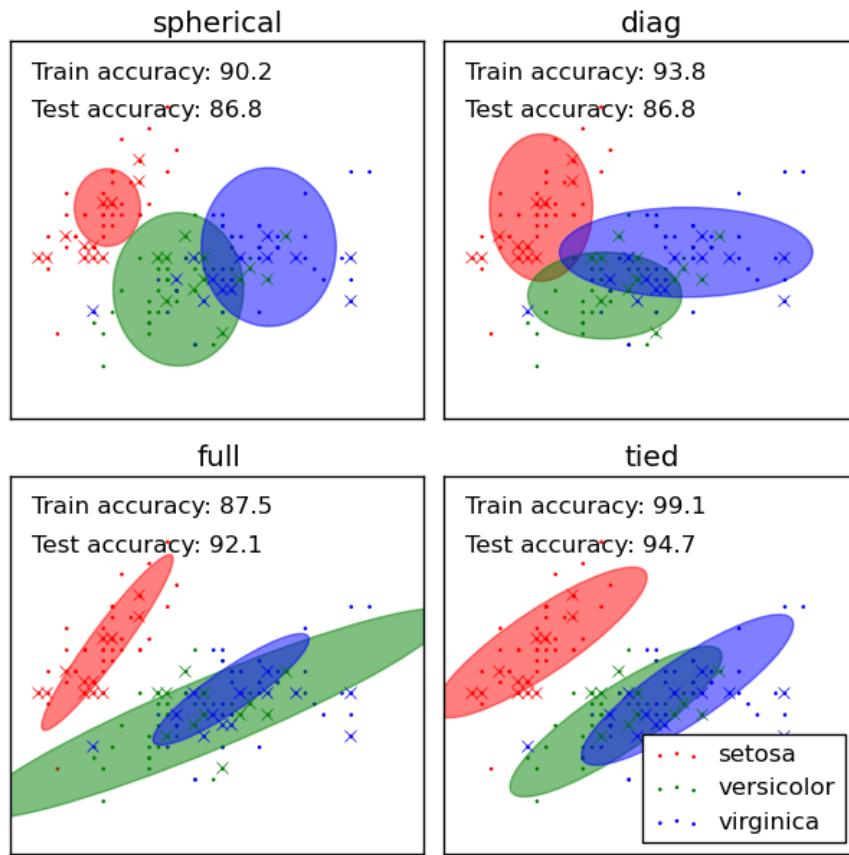
#### GMM classifier

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.

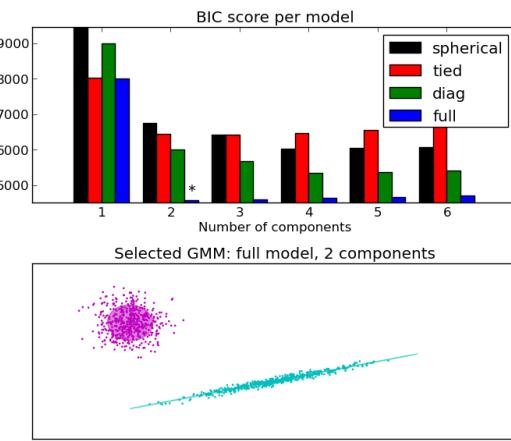
#### Cons

**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-fundamented 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.

## 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 DP-GMM, 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.

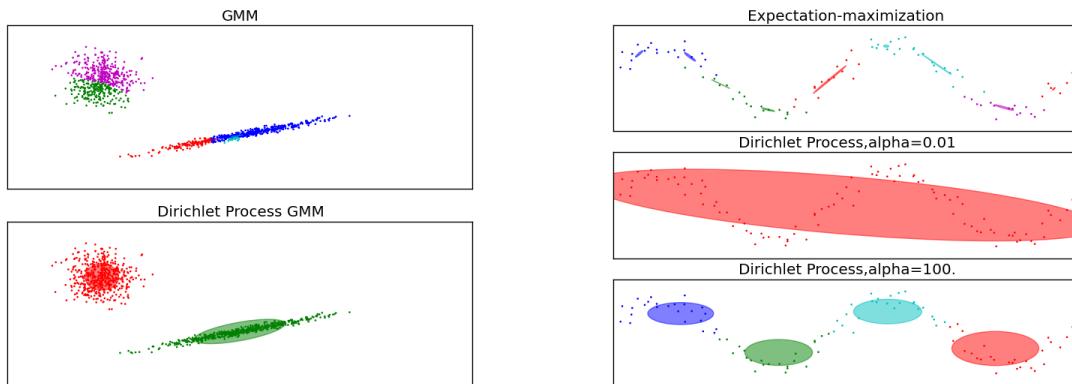
#### Estimation algorithm: variational inference

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.

#### 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: Diriclet 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  $O(n_{\text{states}} * n_{\text{points}} * \text{dimension})$  and for *tied* or *full* it is  $O(n_{\text{states}} * n_{\text{points}} * \text{dimension}^2 + n_{\text{states}} * \text{dimension}^3)$  (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  $\phi_k$ , the mean parameters  $\mu_k$ , the covariance parameters  $\Sigma_k$ , 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 parameterization,  $B_k$  can be a positive scalar, a positive vector or a Symmetric Positive Definite matrix.

**The spherical model** The model then is

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

The variational distribution we'll use is

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

**The bound** The variational bound is

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

### The bound for $\phi_k$

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

### The bound for $\mu_k$

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

### The bound for $\sigma_k$

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

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

### The bound for $z$

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

### The bound for $X$

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

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

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

### The updates Updating $\gamma$

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

### Updating $\mu$

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 w.r.t.  $\nu_\mu$  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  $\Psi'$  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  $\sigma_k$  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

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

The variational distribution we'll use is

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

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

The bound for  $\sigma_{k,d}$  is the same bound for  $\sigma_k$  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

$$\begin{aligned} E_q[\log P(X_i)] &= \sum_k \nu_{z_k} \left( -\frac{D}{2} \log 2\pi + \frac{1}{2} \sum_d (\Psi(a_{k,d}) - \log(b_{k,d})) \right. \\ &\quad \left. - \frac{1}{2} ((X_i - \nu_{\mu_k})^T \frac{\mathbf{a}_k}{b_k} (X_i - \nu_{\mu_k}) + \sum_d \sigma_{k,d}) - \log 2\pi e \right) \end{aligned}$$

**The updates** The updates only change for  $\mu$  (to weight them with the new  $\sigma$ ),  $z$  (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 $\mu$

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

### 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

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

The variational distribution we'll use is

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

**The lower bound** There are two changes in the lower-bound: for  $\Sigma$  and for  $X$ .

**The bound for  $\Sigma$**

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

**The bound for  $X$**

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

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

**The update for  $\mu$**

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

**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.

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

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

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

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

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

The variational distribution we'll use is

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

**The lower bound** All that changes in this lower bound in comparison to the previous one is that there are K priors on different  $\Sigma$  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

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

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

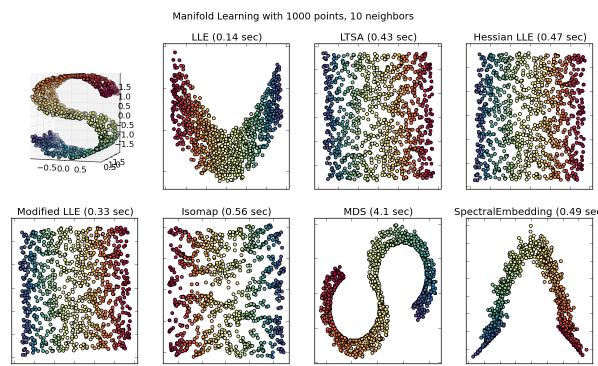
and

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

## 1.4.2 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]



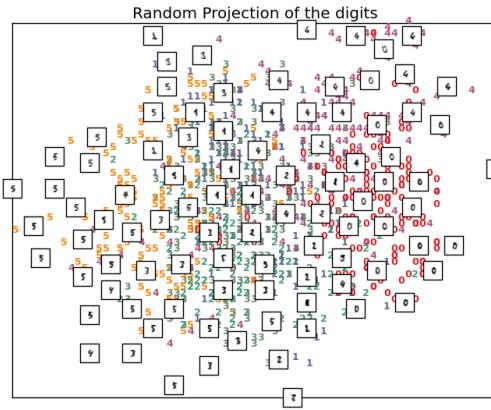
Manifold learning is an approach to nonlinear dimensionality reduction. Algorithms for this task are based on the idea that the dimensionality of many data sets is only artificially high.

### Introduction

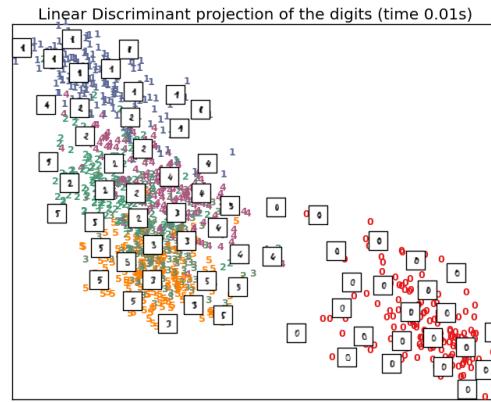
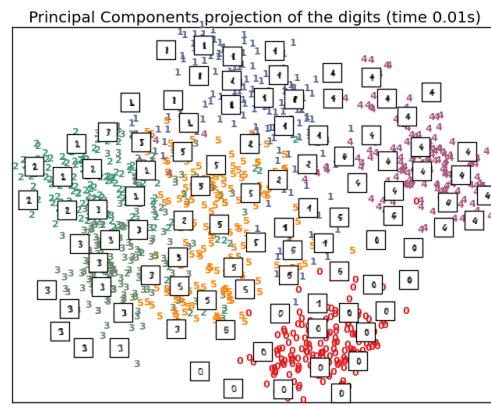
High-dimensional datasets can be very difficult to visualize. While data in two or three dimensions can be plotted to show the inherent structure of the data, equivalent high-dimensional plots are much less intuitive. To aid visualization of the structure of a dataset, the dimension must be reduced in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

The simplest way to accomplish this dimensionality reduction is by taking a random projection of the data. Though this allows some degree of visualization of the data structure, the randomness of the choice leaves much to be desired. In a random projection, it is likely that the more interesting structure within the data will be lost.

A selection from the 64-dimensional digits dataset
0 1 2 3 4 5 0 1 1 3 4 5 0 1 2 3 4 5 0 5
5 5 0 4 1 3 5 1 0 0 2 2 0 1 2 3 3 3 3 3
6 4 1 5 0 5 4 2 0 0 1 3 2 1 4 3 1 3 1 4
3 4 4 0 5 7 4 5 4 2 2 2 5 5 4 4 0 0 1
2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 5 5 5
0 4 1 3 5 1 0 0 2 2 2 0 1 2 3 3 3 3 4 4
4 5 0 5 2 2 0 0 1 3 2 1 3 1 3 4 4 3 4 4
0 5 7 4 5 4 4 1 2 2 2 5 5 4 4 0 0 1 2 3 4
5 0 4 2 3 4 5 0 4 2 3 4 5 0 5 5 5 5 0 4 1
3 5 4 0 0 2 2 2 0 4 2 3 3 3 3 4 4 1 5 0
5 2 2 0 0 4 3 2 4 3 4 3 1 4 3 1 4 0 5
3 6 5 4 4 2 2 2 5 5 4 6 0 3 0 1 2 3 4 5
0 1 2 3 4 5 0 4 2 3 4 5 0 5 5 5 5 0 4 1 3
5 1 0 0 1 2 2 0 1 2 3 3 3 3 5 4 4 1 5 0 5
1 2 0 0 1 3 2 4 4 3 1 3 1 4 3 1 4 0 5 3
4 5 4 4 2 2 2 5 6 4 4 0 0 1 2 3 4 5 0 1
2 3 4 5 0 1 2 3 4 5 0 5 5 5 0 4 1 3 5 4
0 0 1 2 2 0 1 2 3 3 3 3 4 4 4 5 0 5 2 2
0 0 1 3 1 1 4 3 1 3 1 4 3 1 4 0 5 3 1 5
4 9 1 2 1 5 5 4 4 0 0 1 2 3 4 5 0 1 2 3



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 nonlinear 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.

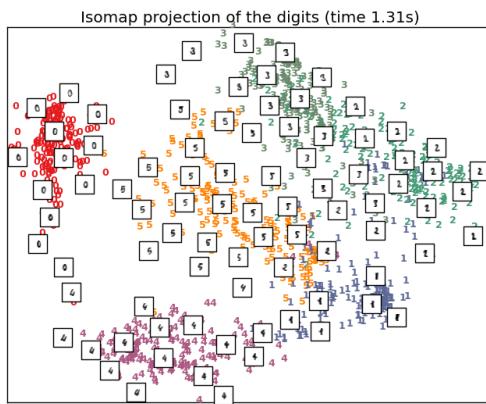
**Examples:**

- 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

**Isomap**

One of the earliest approaches to manifold learning is the Isomap algorithm, short for Isometric Mapping. Isomap can be viewed as an extension of Multi-dimensional Scaling (MDS) or Kernel PCA. Isomap seeks a lower-dimensional embedding which maintains geodesic distances between all points. Isomap can be performed with the object `Isomap`.

**Complexity**

The Isomap algorithm comprises three stages:

1. **Nearest neighbor search.** Isomap uses `sklearn.neighbors.BallTree` for efficient neighbor search. The cost is approximately  $O[D \log(k)N \log(N)]$ , for  $k$  nearest neighbors of  $N$  points in  $D$  dimensions.
2. **Shortest-path graph search.** The most efficient known algorithms for this are *Dijkstra's Algorithm*, which is approximately  $O[N^2(k + \log(N))]$ , or the *Floyd-Warshall algorithm*, which is  $O[N^3]$ . The algorithm can be selected by the user with the `path_method` keyword of `Isomap`. If unspecified, the code attempts to choose the best algorithm for the input data.
3. **Partial eigenvalue decomposition.** The embedding is encoded in the eigenvectors corresponding to the  $d$  largest eigenvalues of the  $N \times N$  isomap kernel. For a dense solver, the cost is approximately  $O[dN^2]$ . This cost can often be improved using the ARPACK solver. The eigensolver can be specified by the user with the `path_method` keyword of `Isomap`. If unspecified, the code attempts to choose the best algorithm for the input data.

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

- $N$  : number of training data points

- $D$  : input dimension
- $k$  : number of nearest neighbors
- $d$  : output dimension

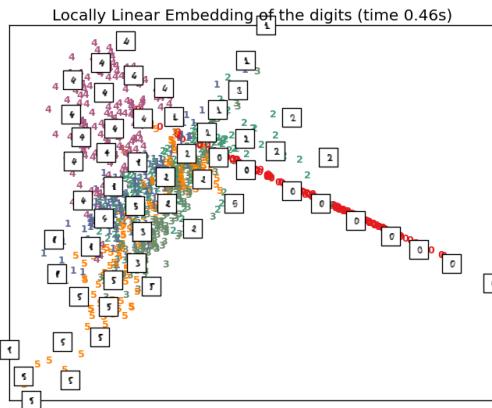
## References:

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

## Locally Linear Embedding

Locally linear embedding (LLE) seeks a lower-dimensional projection of the data which preserves distances within local neighborhoods. It can be thought of as a series of local Principal Component Analyses which are globally compared to find the best nonlinear 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.**  $O[D N k^3]$ . The construction of the LLE weight matrix involves the solution of a  $k \times k$  linear equation for each of the  $N$  local neighborhoods
3. **Partial Eigenvalue Decomposition.** See discussion under Isomap above.

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

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

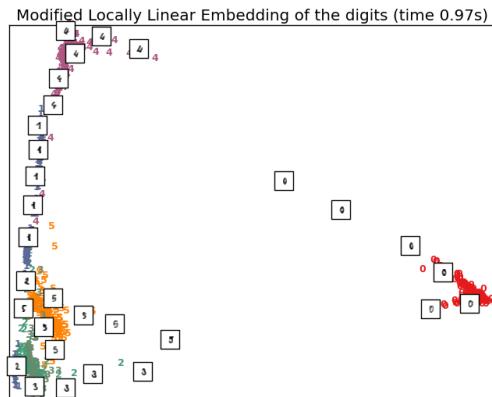
**References:**

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

**Modified Locally Linear Embedding**

One well-known issue with LLE is the regularization problem. When the number of neighbors is greater than the number of input dimensions, the matrix defining each local neighborhood is rank-deficient. To address this, standard LLE applies an arbitrary regularization parameter  $r$ , which is chosen relative to the trace of the local weight matrix. Though it can be shown formally that as  $r \rightarrow 0$ , 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

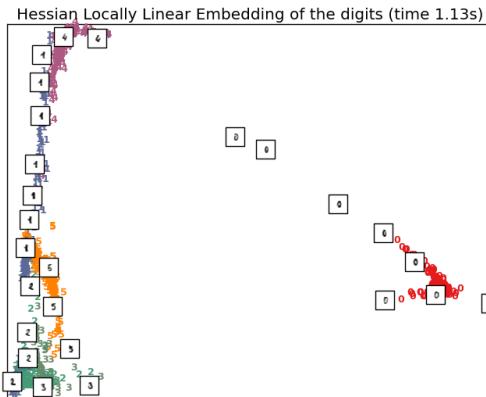
- $k$  : number of nearest neighbors
- $d$  : output dimension

### References:

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

## Hessian Eigenmapping

Hessian Eigenmapping (also known as Hessian-based LLE: HLLE) is another method of solving the regularization problem of LLE. It revolves around a hessian-based quadratic form at each neighborhood which is used to recover the locally linear structure. Though other implementations note its poor scaling with data size, sklearn implements some algorithmic improvements which make its cost comparable to that of other LLE variants for small output dimension. HLLE can be performed with function `locally_linear_embedding` or its object-oriented counterpart `LocallyLinearEmbedding`, with the keyword `method = 'hessian'`. It requires `n_neighbors > n_components * (n_components + 3) / 2`.



## Complexity

The HLLE algorithm comprises three stages:

1. **Nearest Neighbors Search.** Same as standard LLE
2. **Weight Matrix Construction.** Approximately  $O[DNk^3] + O[Nd^6]$ . The first term reflects a similar cost to that of standard LLE. The second term comes from a QR decomposition of the local hessian estimator.
3. **Partial Eigenvalue Decomposition.** Same as standard LLE

The overall complexity of standard HLLE is  $O[D \log(k)N \log(N)] + O[DNk^3] + O[Nd^6] + O[dN^2]$ .

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

**References:**

- “Hessian Eigenmaps: Locally linear embedding techniques for high-dimensional data” Donoho, D. & Grimes, C. Proc Natl Acad Sci USA. 100:5591 (2003)

## Spectral Embedding

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

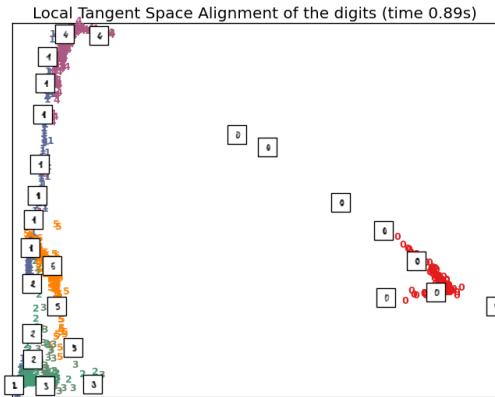
- $N$  : number of training data points
- $D$  : input dimension
- $k$  : number of nearest neighbors
- $d$  : output dimension

**References:**

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

## Local Tangent Space Alignment

Though not technically a variant of LLE, Local tangent space alignment (LTSA) is algorithmically similar enough to LLE that it can be put in this category. Rather than focusing on preserving neighborhood distances as in LLE, LTSA seeks to characterize the local geometry at each neighborhood via its tangent space, and performs a global optimization to align these local tangent spaces to learn the embedding. LTSA can be performed with function `locally_linear_embedding` or its object-oriented counterpart `LocallyLinearEmbedding`, with the keyword `method = 'ltsa'`.



## Complexity

The LTSA algorithm comprises three stages:

1. **Nearest Neighbors Search.** Same as standard LLE
2. **Weight Matrix Construction.** Approximately  $O[DNk^3] + O[k^2d]$ . The first term reflects a similar cost to that of standard LLE.
3. **Partial Eigenvalue Decomposition.** Same as standard LLE

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

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

### References:

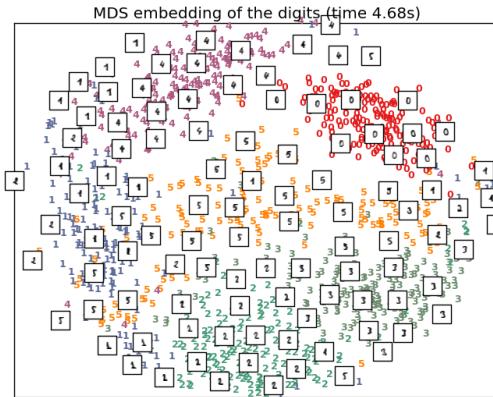
- “Principal manifolds and nonlinear dimensionality reduction via tangent space alignment” Zhang, Z. & Zha, H. Journal of Shanghai Univ. 8:406 (2004)

## Multi-dimensional Scaling (MDS)

Multidimensional scaling (MDS) seeks a low-dimensional representation of the data in which the distances respect well the distances in the original high-dimensional space.

In general, 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 vision, 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  $S_{ij}$ .

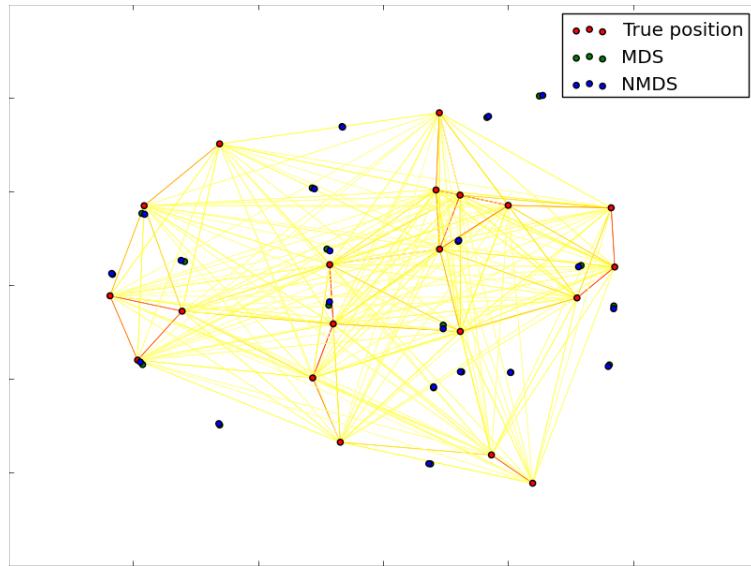
A trivial solution to this problem is to set all the points on the origin. In order to avoid that, the disparities  $\hat{d}_{ij}$  are normalized.

## 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)

## 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.

### 1.4.3 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.

## Overview of clustering methods

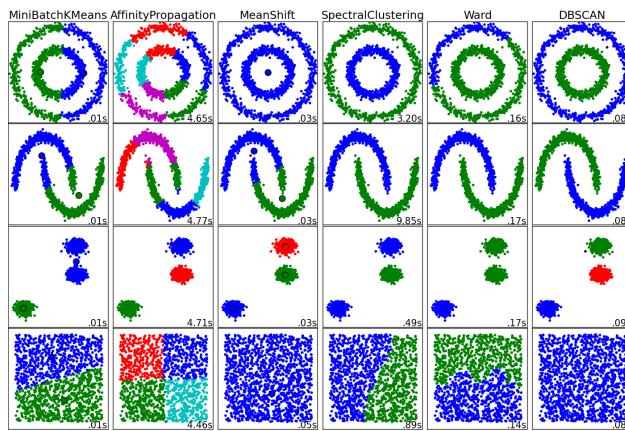


Figure 1.3: A comparison of the clustering algorithms in scikit-learn

Method name	Parameters	Scalability	Use case	Geometry (metric used)
K-Means	number of clusters	Very large $n\_samples$ , medium $n\_clusters$ with <i>MiniBatch</i> 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)
Hierarchical clustering	number of clusters	Large $n\_samples$ and $n\_clusters$	Many clusters, possibly connectivity constraints	Distances between points
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

Non-flat geometry clustering is useful when the clusters have a specific shape, i.e. a non-flat manifold, and the standard euclidean distance is not the right metric. This case arises in the two top rows of the figure above.

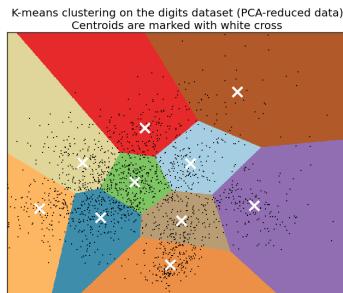
Gaussian mixture models, useful for clustering, are described in *another chapter of the documentation* dedicated to mixture models. KMeans can be seen as a special case of Gaussian mixture model with equal covariance per component.

## K-means

The KMeans algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the ‘inertia’ of the groups. This algorithm requires the number of cluster 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. It is also equivalent to the expectation-maximization algorithm when setting the covariance matrix to be diagonal, equal and small. The K-means algorithm aims to choose centroids  $C$  that minimise the within cluster sum of squares objective function with a dataset  $X$  with  $n$  samples:

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

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 other two major 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 is the inertia 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.

The algorithm can be identified 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, as in this implementation, the algorithm stops when the relative decrease in the objective function between iterations is less than the given tolerance value.

Given enough time, K-means will always converge, however this may be to a local minimum. This is highly dependent on the the initialisation of the centroids. As a result, the computation is often done several times, with different initialisation of the centroids. One method to help address this issue is the k-means++ initialisation algorithm, which has been implemented in scikit-learn (use the `init='kmeans++'` parameter). This initialises the centroids to be (generally) distant from each other, leading to provably better results than random initialisation.

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 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.

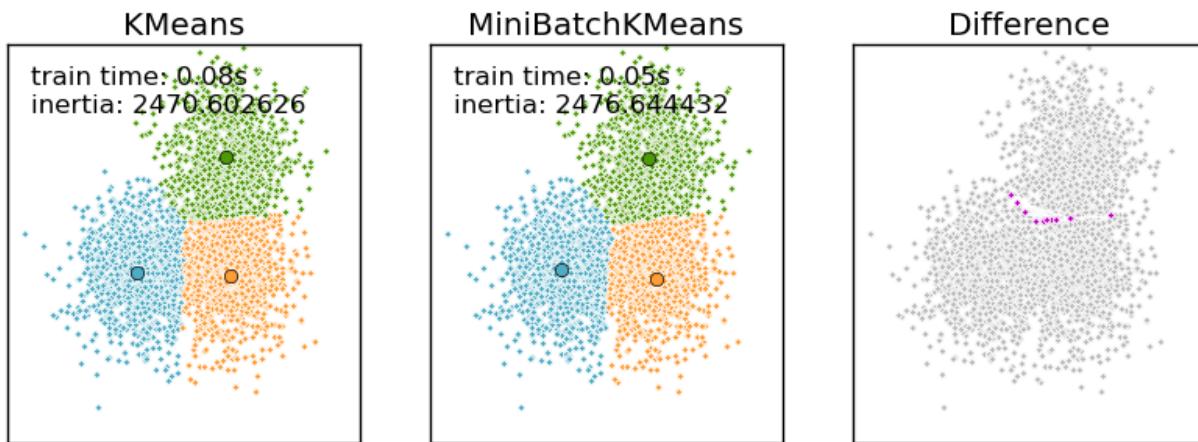
#### Examples:

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

#### Mini Batch K-Means

The MiniBatchKMeans is a variant of the KMeans algorithm using mini-batches, random subset of the dataset, to compute the centroids.

Although the MiniBatchKMeans converge faster than the KMeans version, the quality of the results, measured by the inertia, the sum of the distance of each points to the nearest centroid, is not as good as the KMeans algorithm.



#### Examples:

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

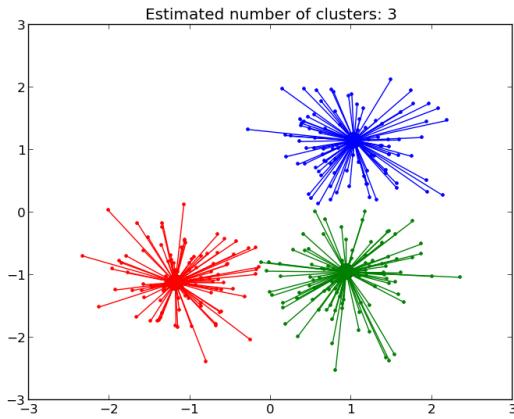
#### References:

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

#### Affinity Propagation

AffinityPropagation creates clusters by sending messages between pairs of samples until convergence. A

dataset is then described using a small number of exemplars, which are identified as those most representative of other samples. The messages sent between pairs represent the suitability for one sample to be the exemplar of the other, which is updated in response to the values from other pairs. This updating happens iteratively until convergence, at which point the final exemplars are chosen, and hence the final clustering is given.



Affinity Propagation can be interesting as it chooses the number of clusters based on the data provided. For this purpose, the two important parameters are the *preference*, which controls how many exemplars are used, and the *damping* factor.

The main drawback of Affinity Propagation is its complexity. The algorithm has a time complexity of the order  $O(N^2T)$ , where  $N$  is the number of samples and  $T$  is the number of iterations until convergence. Further, the memory complexity is of the order  $O(N^2)$  if a dense similarity matrix is used, but reducible if a sparse similarity matrix is used. This makes Affinity Propagation most appropriate for small to medium sized datasets.

### Examples:

- *Demo of affinity propagation clustering algorithm*: Affinity Propagation on a synthetic 2D datasets with 3 classes.
- *Visualizing the stock market structure* Affinity Propagation on Financial time series to find groups of companies

**Algorithm description:** The messages sent between points belong to one of two categories. The first is the responsibility  $r(i, k)$ , which is the accumulated evidence that sample  $k$  should be the exemplar for sample  $i$ . The second is the availability  $a(i, k)$  which is the accumulated evidence that sample  $i$  should choose sample  $k$  to be its exemplar, and considers the values for all other samples that  $k$  should be an exemplar. In this way, exemplars are chosen by samples if they are (1) similar enough to many samples and (2) chosen by many samples to be representative of themselves.

More formally, the responsibility of a sample  $k$  to be the exemplar of sample  $i$  is given by:

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

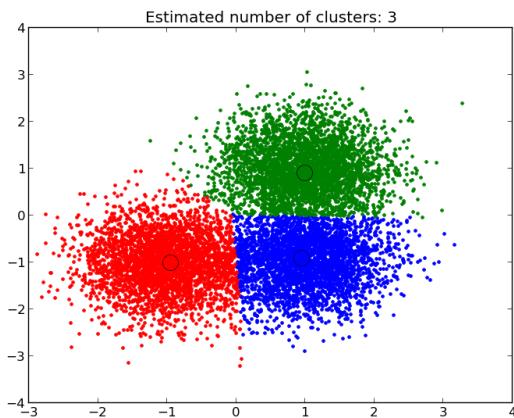
Where  $s(i, k)$  is the similarity between samples  $i$  and  $k$ . The availability of sample  $k$  to be the exemplar of sample  $i$  is given by:

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

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

## Mean Shift

MeanShift clusters data by estimating *blobs* in a smooth density of points matrix. This algorithm automatically sets its numbers of cluster. It will have difficulties scaling to thousands of samples. The utility function `estimate_bandwidth` can be used to guess the optimal bandwidth for MeanShift from the data.



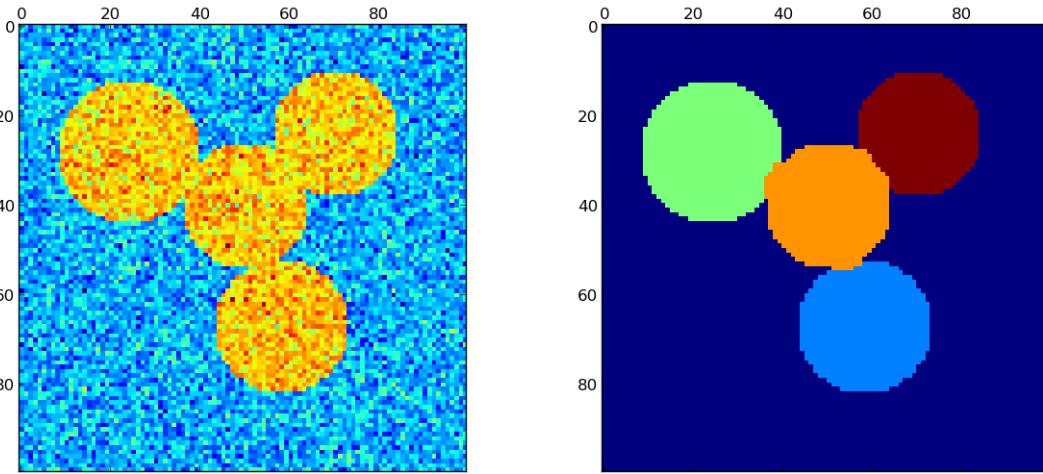
### Examples:

- A demo of the mean-shift clustering algorithm: Mean Shift clustering on a synthetic 2D datasets with 3 classes.

## 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 in of 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.

<code>assign_labels="kmeans"</code>	<code>assign_labels="discretize"</code>
<p>Spectral clustering: kmeans, 176.54s</p>	<p>Spectral clustering: discretize, 201.55s</p>

**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

**Hierarchical clustering**

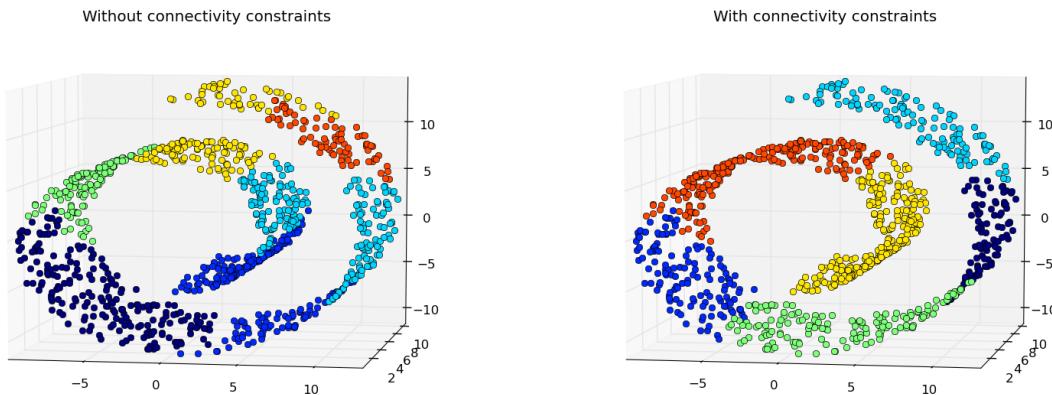
Hierarchical clustering is a general family of clustering algorithms that build nested clusters by merging them successively. This hierarchy of clusters represented as a tree (or dendrogram). The root of the tree is the unique cluster that gathers all the samples, the leaves being the clusters with only one sample. See the [Wikipedia page](#) for more details.

The `Ward` object performs a hierarchical clustering based on the Ward algorithm, that is a variance-minimizing approach. At each step, it minimizes the sum of squared differences within all clusters (inertia criterion).

This algorithm can scale to large number of samples when it is used jointly with an connectivity matrix, but can be computationally expensive when no connectivity constraints are added between samples: it considers at each step all the possible merges.

**Adding connectivity constraints**

An interesting aspect of the `Ward` object is that connectivity constraints can be added to this algorithm (only adjacent clusters can be merged together), through an 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.



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 if you wish to cluster web pages, but 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 the `swiss roll` 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.

## DBSCAN

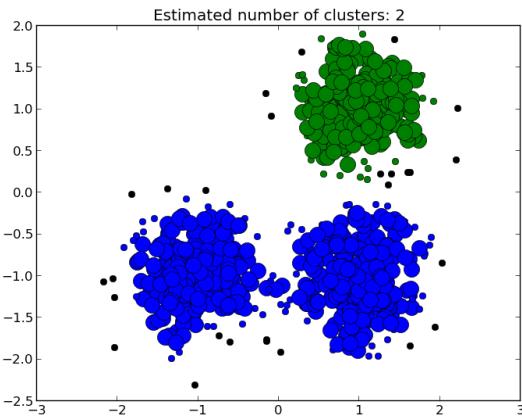
The DBSCAN algorithm views clusters as areas of high density separated by areas of low density. Due to this rather generic view, clusters found by DBSCAN can be any shape, as opposed to k-means which assumes that clusters are convex shaped. The central component to the DBSCAN is the concept of *core samples*, which are samples that are in areas of high density. A cluster is therefore a set of core samples, each highly similar to each other and a set of non-core samples that are similar to a core sample (but are not themselves core samples). There are two parameters to the algorithm, *min\_points* and *eps*, which define formally what we mean when we say *dense*. A higher *min\_points* 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 exists *min\_samples* other samples with a similarity higher than *eps* to it, 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 not have a similarity higher than *eps* to a core sample, it is considered an outlier by the algorithm.

The algorithm is non-deterministic, however the core samples themselves will always belong to the same clusters (although the labels themselves may be different). The non-determinism comes from deciding on which cluster a non-core sample belongs to. A non-core sample can have a similarity higher than  $\text{eps}$  to two core samples in different classes. Following from the triangular inequality, those two core samples would be less similar than  $\text{eps}$  from each other – else they would be in the same class. The non-core sample is simply assigned to whichever cluster is generated first, where the order is determined randomly within the code. Other than the ordering of, the dataset, the algorithm is deterministic, making the results relatively stable between iterations on the same data.

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: Clustering synthetic data with DBSCAN*

#### References:

- “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise” Ester, M., H. P. Kriegel, J. Sander, and X. Xu, In Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press, pp. 226–231. 1996

## Clustering performance evaluation

Evaluating the performance of a clustering algorithm is not as trivial as counting the number of errors or the precision and recall of a supervised classification algorithm. In particular any evaluation metric should not take the absolute values of the cluster labels into account but rather if this clustering define separations of the data similar to some ground truth set of classes or satisfying some assumption such that members belong to the same class are more similar than members of different classes according to some similarity metric.

### Inertia

**Presentation and usage** TODO: factorize inertia computation out of kmeans and then write me!

## Advantages

- No need for the ground truth knowledge of the “real” classes.

## Drawbacks

- Inertia makes the assumption that clusters are convex and isotropic which is not always the case especially if the clusters are manifolds with weird shapes: for instance inertia is a useless metric to evaluate clustering algorithm that tries to identify nested circles on a 2D plane.
- Inertia is not a normalized metric: we just know that lower values are better and bounded by zero. One potential solution would be to adjust inertia for random clustering (assuming the number of ground truth classes is known).

## 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 and rename 2 by 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 and rename 2 by 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 data),  $U$  with  $R$  classes and  $V$  with  $C$  classes. The entropy of either is the amount of uncertainty for an array, and can be calculated as:

$$H(U) = \sum_{i=1}^{|R|} P(i) \log(P(i))$$

Where  $P(i)$  is the number of instances in  $U$  that are in class  $R_i$ . Likewise, for  $V$ :

$$H(V) = \sum_{j=1}^{|C|} P'(j) \log(P'(j))$$

Where  $P'(j)$  is the number of instances in  $V$  that are in class  $C_j$ .

The mutual information between  $U$  and  $V$  is calculated by:

$$\text{MI}(U, V) = \sum_{i=1}^{|R|} \sum_{j=1}^{|C|} P(i, j) \log \left( \frac{P(i, j)}{P(i)P'(j)} \right)$$

Where  $P(i, j)$  is the number of instances with label  $R_i$  and also with label  $C_j$ .

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,  $a_i$  is the number of instances with label  $U_i$  and  $b_j$  is the number of instances with label  $V_j$ .

$$E[\text{MI}(U, V)] = \sum_{i=1}^R \sum_{j=1}^C \sum_{n_{ij}=(a_i+b_j-N)^+}^{\min(a_i, b_j)} \frac{n_{ij}}{N} \log \left( \frac{N \cdot n_{ij}}{a_i b_j} \right) \frac{a_i! b_j! (N - a_i)! (N - b_j)!}{N! n_{ij}! (a_i - n_{ij})! (b_j - n_{ij})! (N - a_i - b_j + n_{ij})!}$$

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 clusters with “folded” shapes.

## Drawbacks

- The previously introduced metrics are **not normalized w.r.t. 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)**.

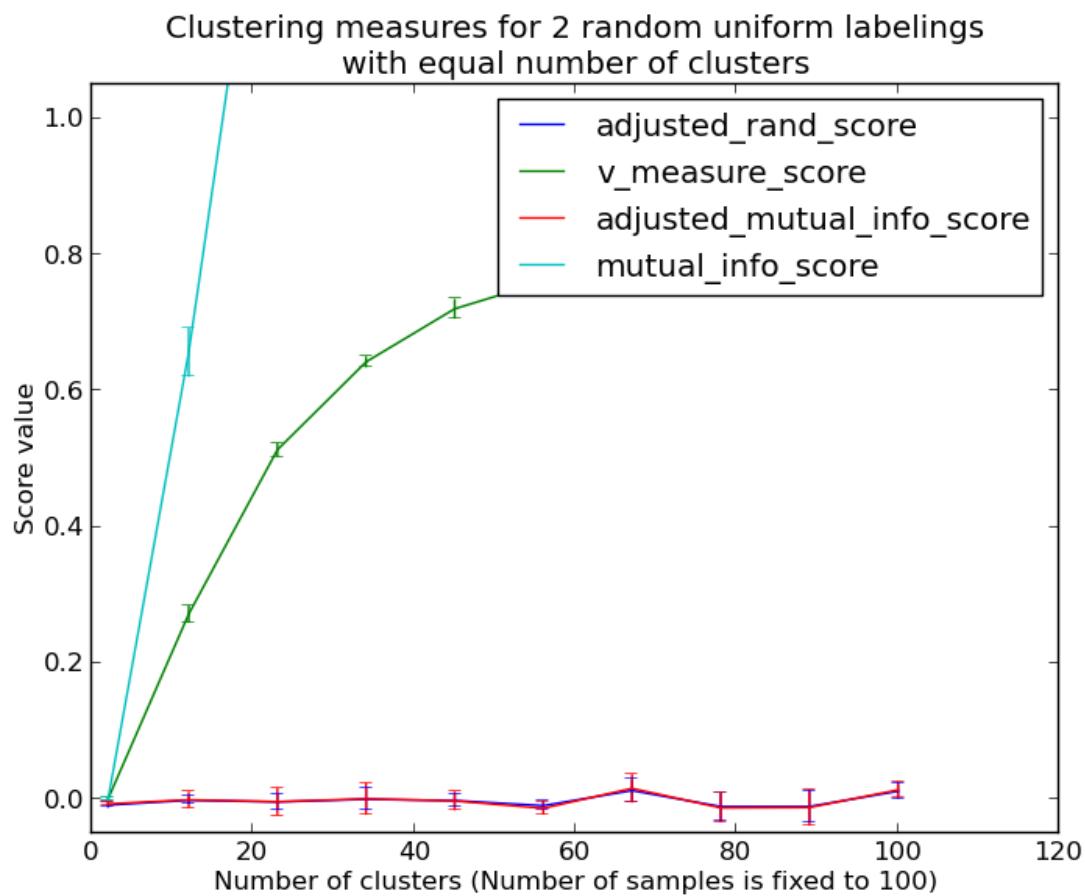
- These metrics **require the knowledge of the ground truth classes** while almost never available in practice or requires manual assignment by human annotators (as in the supervised learning setting).

## 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.

### 1.4.4 Decomposing signals in components (matrix factorization problems)

#### Principal component analysis (PCA)

##### Exact PCA and probabilistic interpretation

PCA is used to decompose a multivariate dataset in a set of successive orthogonal components that explain a maximum amount of the variance. In scikit-learn, PCA is implemented as a *transformer* object that learns n components in its *fit* method, and can be used on new data to project it on these components.

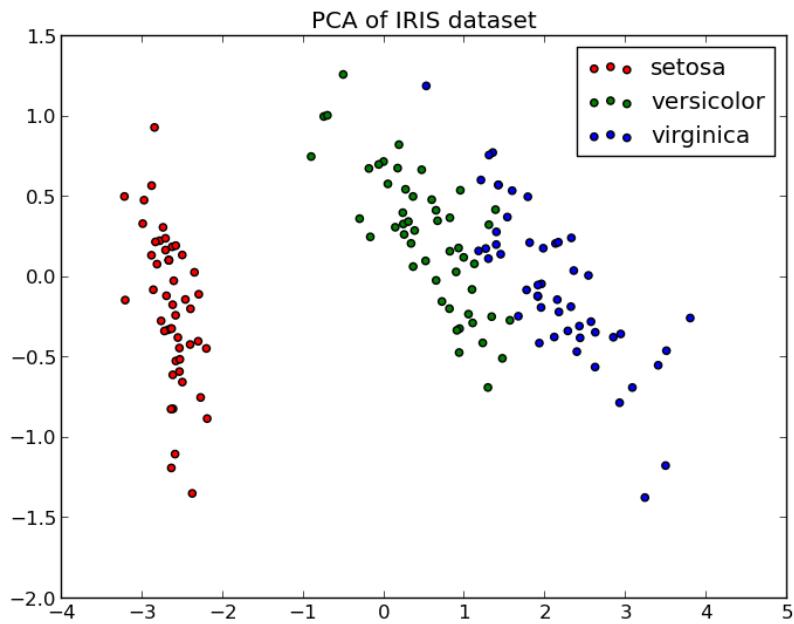
The optional parameter *whiten=True* parameter make it possible to project the data onto the singular space while scaling each component to unit variance. This is often useful if the models down-stream make strong assumptions on the isotropy of the signal: this is for example the case for Support Vector Machines with the RBF kernel and the K-Means clustering algorithm. However in that case the inverse transform is no longer exact since some information is lost while forward transforming.

In addition, the *ProbabilisticPCA* object 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.

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

#### Examples:

- Comparison of LDA and PCA 2D projection of Iris dataset



## Approximate PCA

Often we are interested in projecting the data onto a lower dimensional space that preserves most of the variance by dropping the singular vector of components associated with lower singular values.

For instance for face recognition, if we work with  $64 \times 64$  gray level pixel pictures the dimensionality of the data is 4096 and it is slow to train a RBF Support Vector Machine on such wide data. Furthermore we know that intrinsic dimensionality of the data is much lower than 4096 since all faces pictures look alike. The samples lie on a manifold of much lower dimension (say around 200 for instance). The PCA algorithm can be used to linearly transform the data while both reducing the dimensionality and preserve most of the explained variance at the same time.

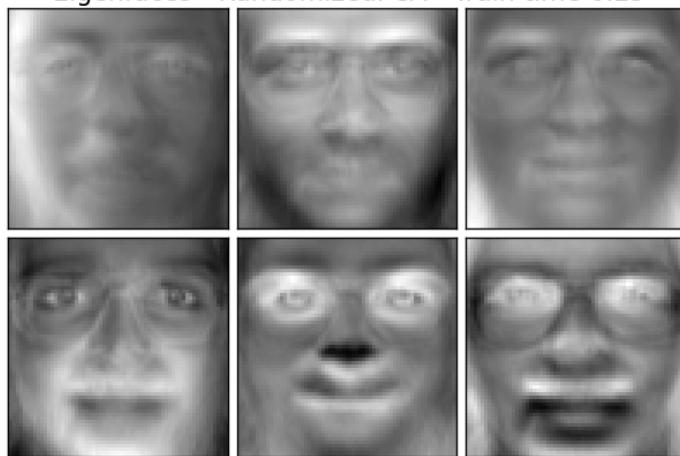
The class `RandomizedPCA` is very useful in that case: since we are going to drop most of the singular vectors it is much more efficient to limit the computation to an approximated estimate of the singular vectors we will keep to actually perform the transform.

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 is less than 1s:

First centered Olivetti faces



Eigenfaces - RandomizedPCA - Train time 0.2s



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

Furthermore RandomizedPCA is able to work with `scipy.sparse` matrices as input which make it suitable for reducing the dimensionality of features extracted from text documents for instance.

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

#### Examples:

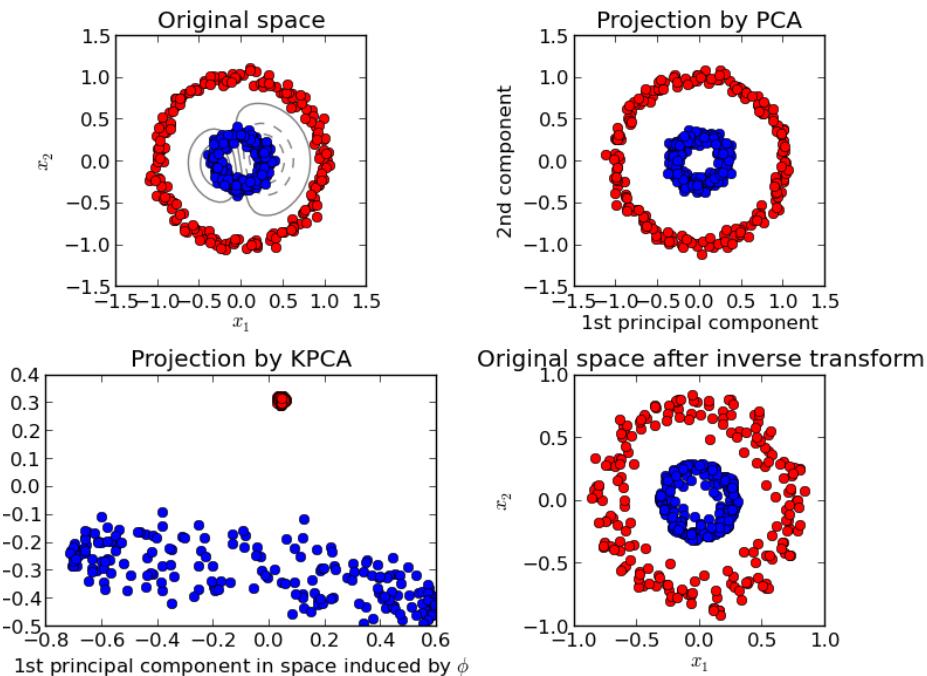
- *Faces recognition example using eigenfaces and SVMs*
- *Faces dataset decompositions*

**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. 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)**

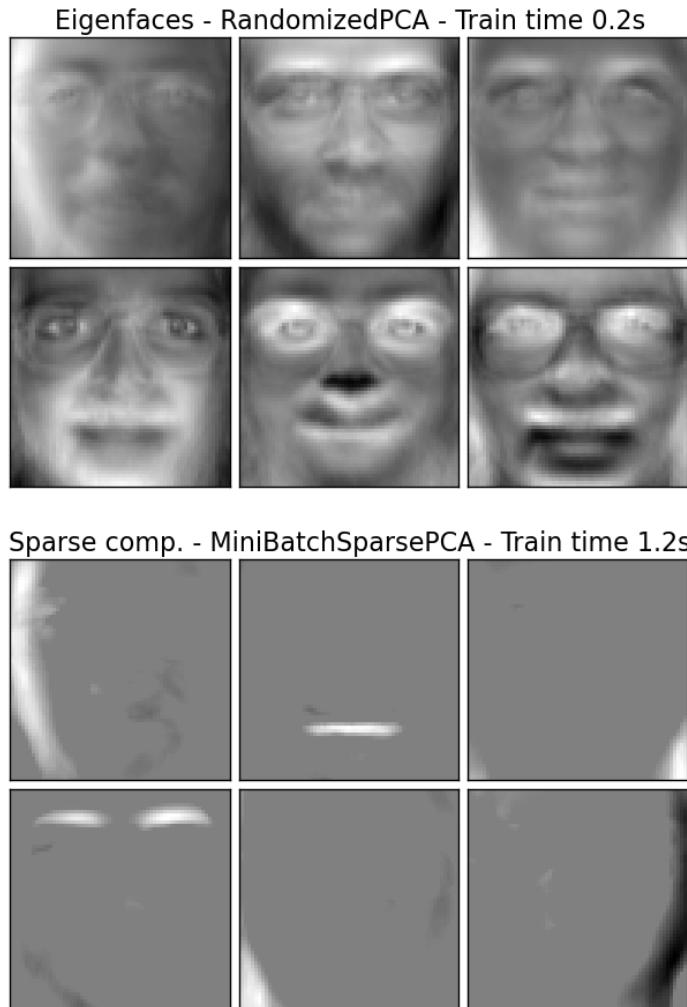
SparsePCA is a variant of PCA, with the goal of extracting the set of sparse components that best reconstruct the data.

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  $\ell_1$  penalty on the components:

$$(U^*, V^*) = \arg \min_{U, V} \frac{1}{2} \|X - UV\|_2^2 + \alpha \|V\|_1$$

subject to  $\|U_k\|_2 = 1$  for all  $0 \leq k < n_{components}$

The sparsity inducing  $\ell_1$  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:**

## 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.

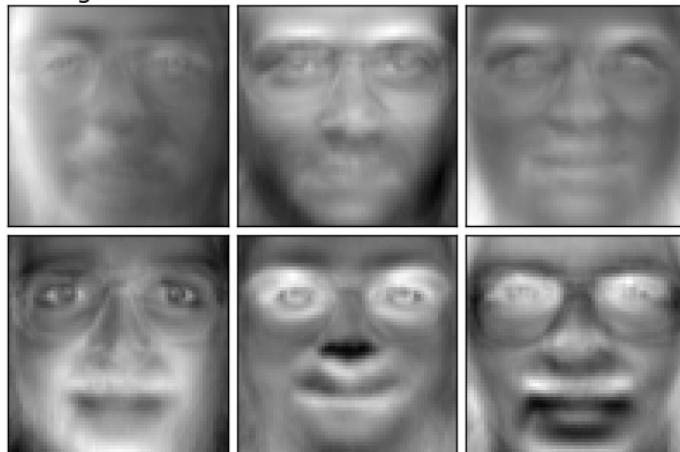
Representing data as sparse combinations of atoms from an overcomplete dictionary is suggested to be the way the mammal primary visual cortex works. Consequently, dictionary learning applied on image patches has been shown to give good results in image processing tasks such as image completion, inpainting 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.

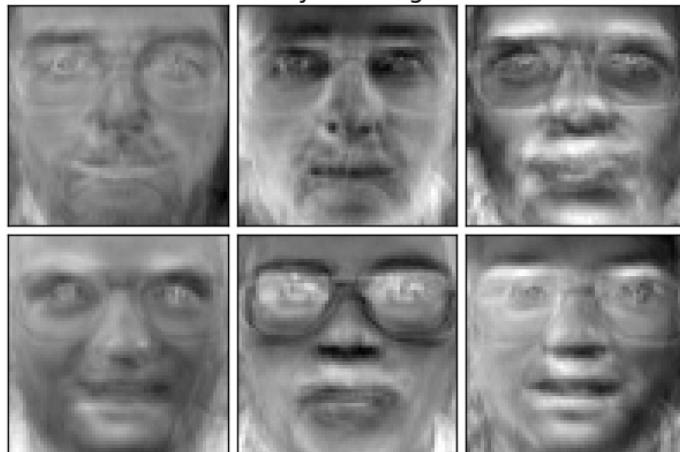
$$(U^*, V^*) = \arg \min_{U, V} \frac{1}{2} \|X - UV\|_2^2 + \alpha \|U\|_1$$

subject to  $\|V_k\|_2 = 1$  for all  $0 \leq k < n_{atoms}$

Eigenfaces - RandomizedPCA - Train time 0.2s

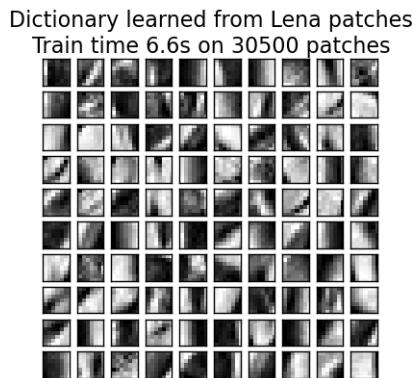


MiniBatchDictionaryLearning - Train time 0.9s



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:**

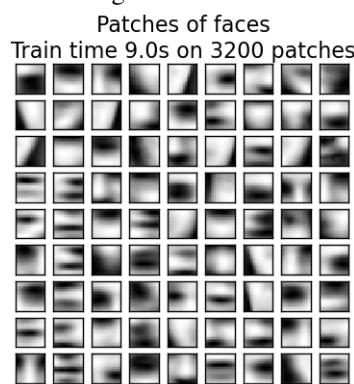
- “Online dictionary learning for sparse coding” J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009

## 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 *partial\_fit* method.

Example: *Online learning of a dictionary of parts of faces*

## Factor Analysis

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

$$x_i = Wh_i + \mu + \epsilon$$

The vector  $h_i$  is called *latent* because it is unobserved.  $\epsilon$  is considered a noise term distributed according to a Gaussian with mean 0 and covariance  $\Psi$  (i.e.  $\epsilon \sim \mathcal{N}(0, \Psi)$ ),  $\mu$  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  $x$ :

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

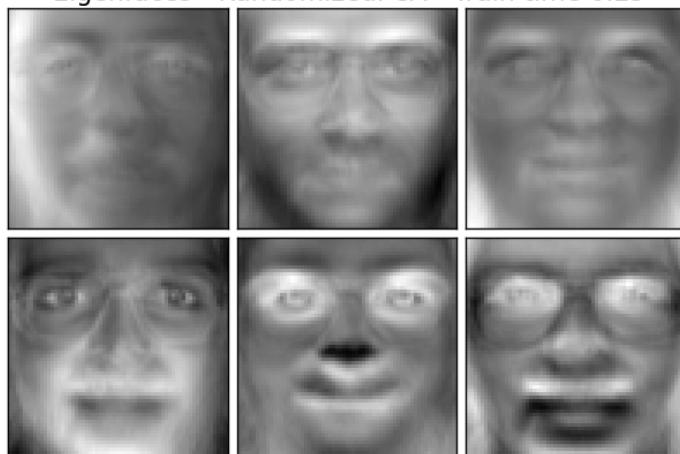
Now, without any further assumptions the idea of having a latent variable  $h$  would be superfluous –  $x$  can be completely modelled with a mean and a covariance. We need to impose some more specific structure on one of these two parameters. A simple additional assumption regards the structure of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

- $\Psi = \sigma^2 \mathbf{I}$ : This assumption leads to ProbabilisticPCA.
- $\Psi = \text{diag}(\psi_1, \psi_2, \dots, \psi_n)$ : This model is called Factor Analysis, a classical statistical model. The matrix  $W$  is sometimes called *factor loading matrix*.

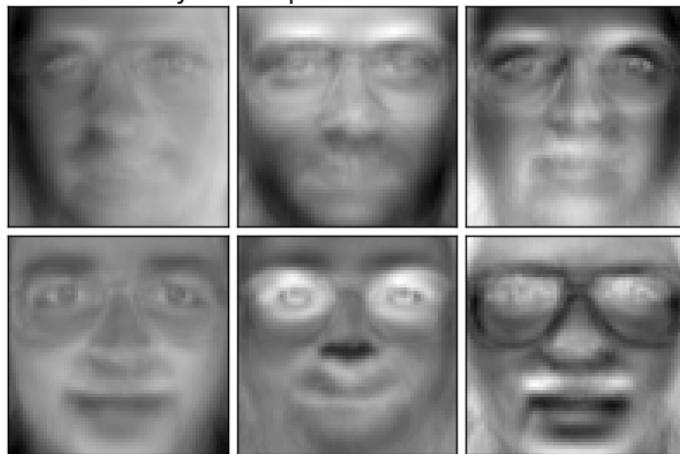
Both model essentially estimate a Gaussian with a low-rank covariance matrix. Because both models are probabilistic they can be integrated in more complex models, e.g. Mixture of Factor Analysers. One gets very different models (e.g. FastICA) if non-Gaussian priors on the latent variables are assumed.

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):

Eigenfaces - RandomizedPCA - Train time 0.2s



Factor Analysis components - FA - Train time 3.3s



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

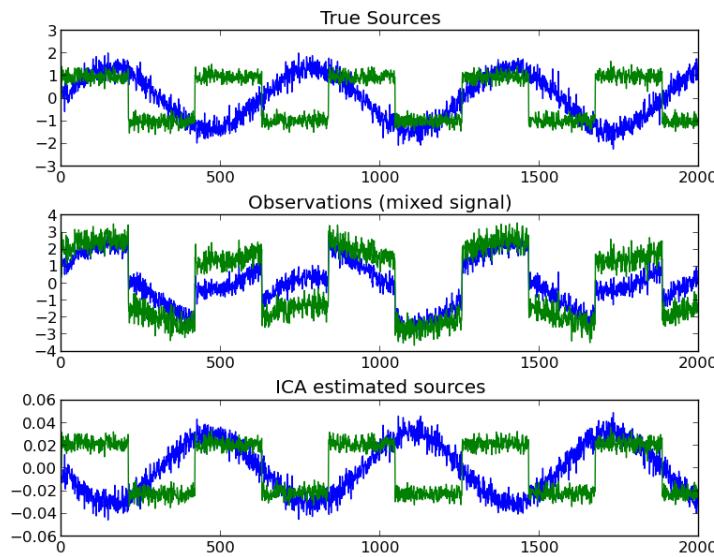
Pixelwise variance



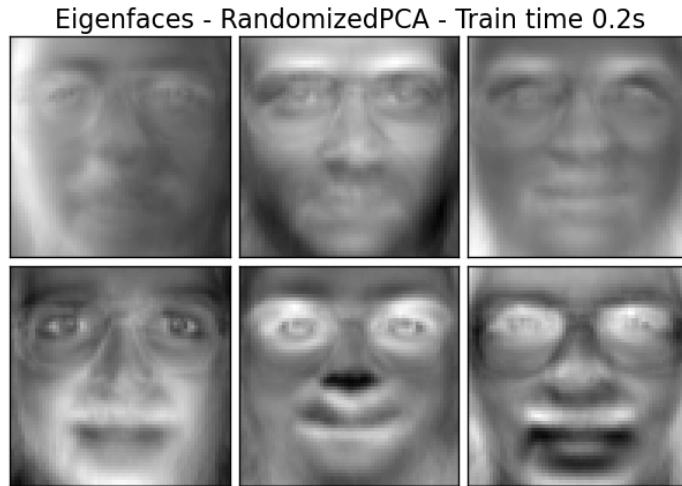
## Independent component analysis (ICA)

Independent component analysis separates a multivariate signal into additive subcomponents that are maximally independent. It is implemented in scikit-learn using the `FastICA` algorithm.

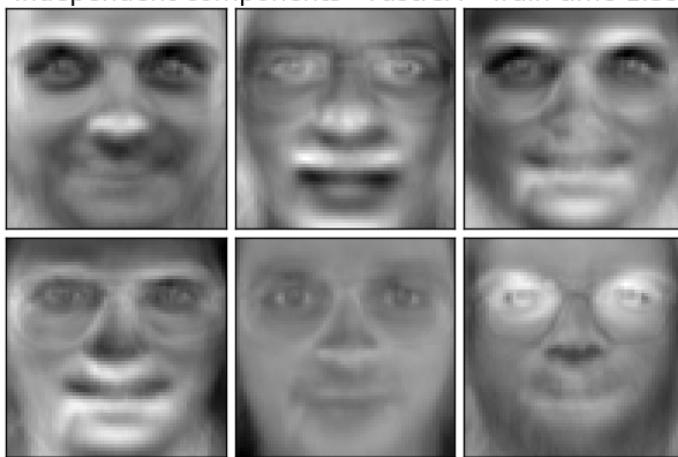
It is classically used to separate mixed signals (a problem known as *blind source separation*), as in the example below:



ICA can also be used as yet another non linear decomposition that finds components with some sparsity:



Independent components - FastICA - Train time 1.3s

**Examples:**

- *Blind source separation using FastICA*
- *FastICA on 2D point clouds*
- *Faces dataset decompositions*

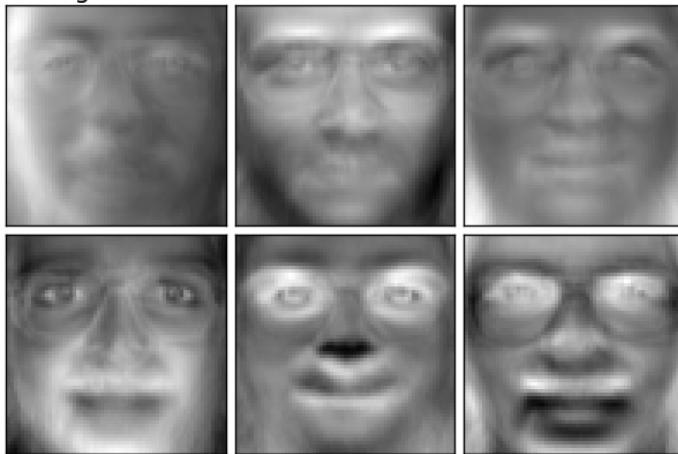
## Non-negative matrix factorization (NMF or NNMF)

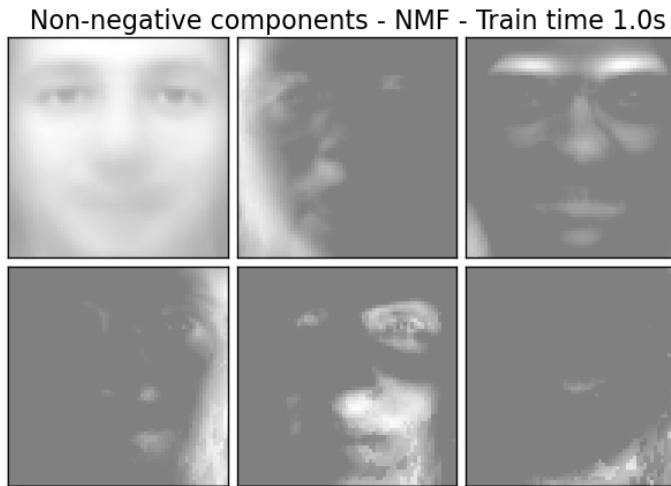
NMF is an alternative approach to decomposition that assumes that the data and the components are non-negative. NMF can be plugged in instead of PCA or its variants, in the cases where the data matrix does not contain negative values.

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.

Eigenfaces - RandomizedPCA - Train time 0.2s





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. Gallopolous, 2008

### 1.4.5 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.).

## 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.

### Examples:

- See *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood* for an example on how to fit an `EmpiricalCovariance` object to data.

## Shrunk Covariance

### Basic shrinkage

Despite being an unbiased estimator of the covariance matrix, the Maximum Likelihood Estimator is not a good estimator of the eigenvalues of the covariance matrix, so the precision matrix obtained from its inversion is not accurate. Sometimes, it even occurs that the empirical covariance matrix cannot be inverted for numerical reasons. To avoid such an inversion problem, a transformation of the empirical covariance matrix has been introduced: the *shrinkage*.

In 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 :  $\Sigma_{\text{shrunk}} = (1 - \alpha)\hat{\Sigma} + \alpha \frac{\text{Tr}\hat{\Sigma}}{p} \text{Id}$ .

Choosing the amount of shrinkage,  $\alpha$  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  $\alpha$  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.

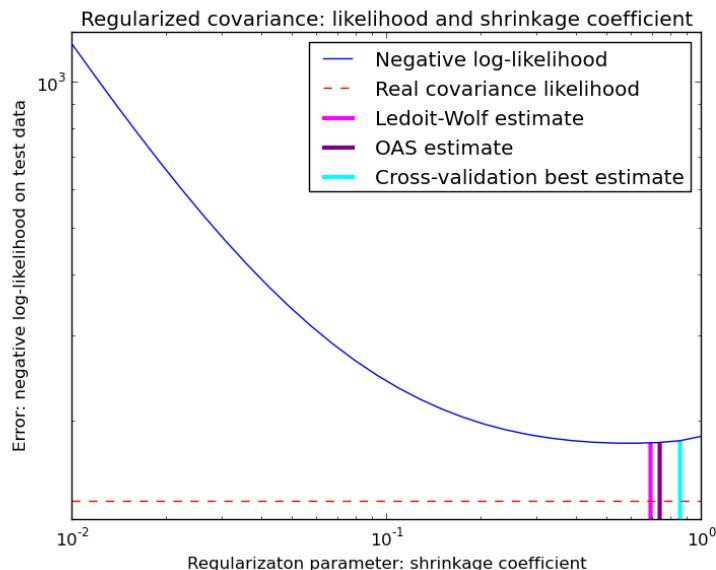
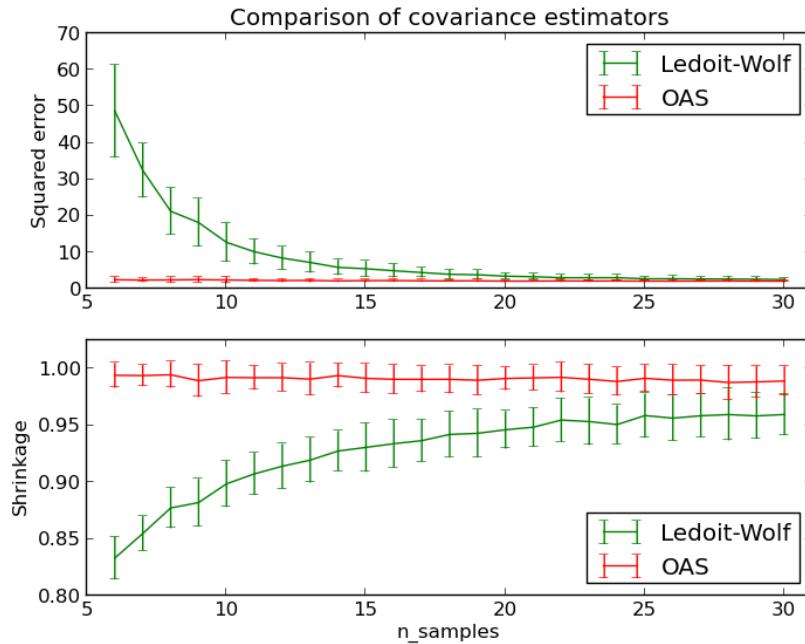


Figure 1.4: Bias-variance trade-off when setting the shrinkage: comparing the choices of Ledoit-Wolf and OAS estimators

[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.



## 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 magnitude 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.

---

### 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

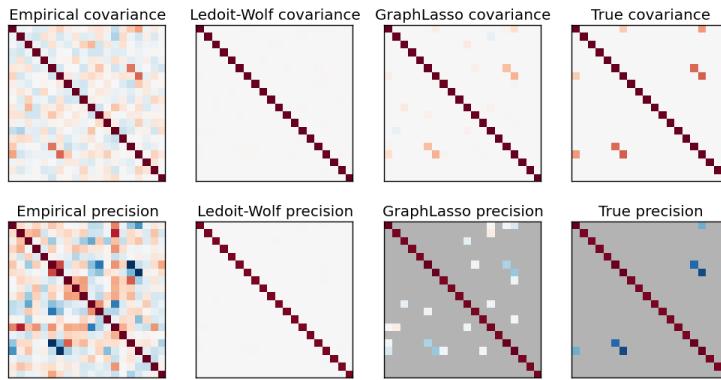


Figure 1.5: A comparison of maximum likelihood, shrinkage and sparse estimates of the covariance and precision matrix in the very small samples settings.

weights than the irrelevant ones.

The mathematical formulation is the following:

$$\hat{K} = \operatorname{argmin}_K (\operatorname{tr} SK - \log \det K + \alpha \|K\|_1)$$

Where  $K$  is the precision matrix to be estimated, and  $S$  is the sample covariance matrix.  $\|K\|_1$  is the sum of the absolute values of off-diagonal coefficients of  $K$ . The algorithm employed to solve this problem is the GLasso algorithm, from the Friedman 2008 Biostatistics paper. It is the same algorithm as in the R *glasso* package.

#### Examples:

- *Sparse inverse covariance estimation*: example on synthetic data showing some recovery of a structure, and comparing to other covariance estimators.
- *Visualizing the stock market structure*: example on real stock market data, finding which symbols are most linked.

#### References:

- Friedman et al, “Sparse inverse covariance estimation with the graphical lasso”, Biostatistics 9, pp 432, 2008

### Robust Covariance Estimation

Real data 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.

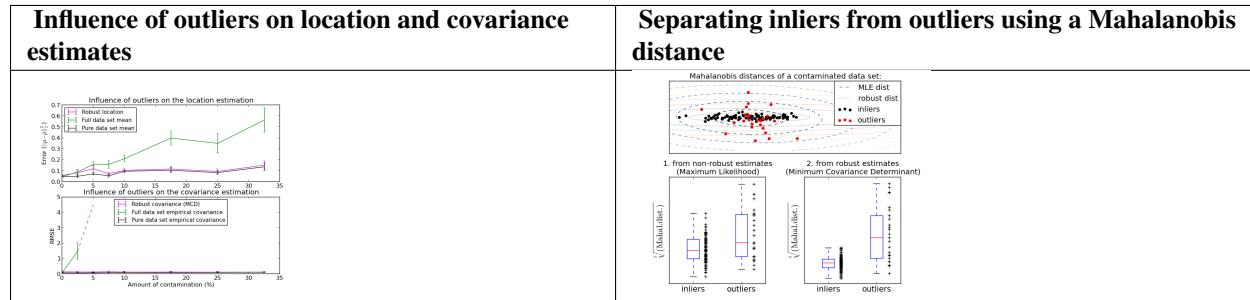
### [3] P. J. Rousseeuw. Least median of squares regression.

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).



## 1.4.6 Novelty and Outlier Detection

Many applications require being able to decide whether a new observation belongs to the same distribution as exiting 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 0, while outliers are labeled 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 in [1] 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 exist no exact formula or algorithm to set its bandwidth parameter. This is the default in the scikit-learn implementation. The  $\nu$  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.

### Examples:

- See *One-class SVM with non-linear kernel (RBF)* for visualizing the frontier learned around some data by a `svm.OneClassSVM` object.

## Outlier Detection

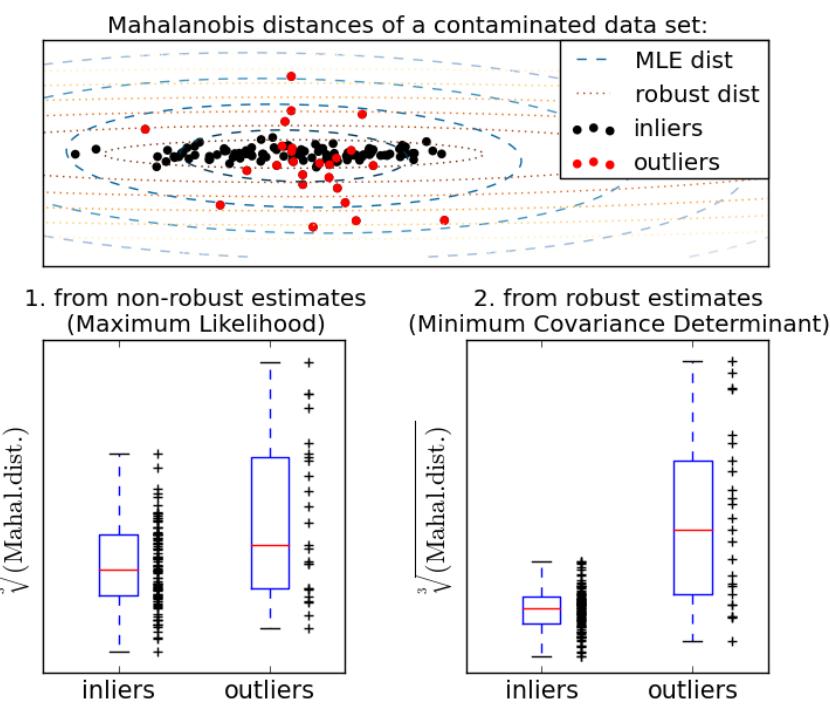
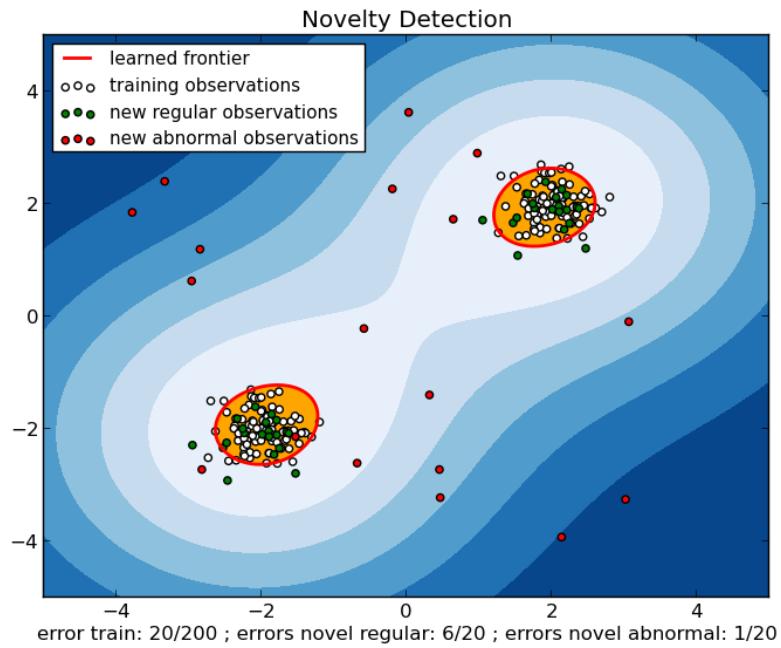
Outlier detection is similar to novelty detection in the sense that the goal is to separate a core of regular observations from some polluting ones, called “outliers”. Yet, in the case of outlier detection, we don’t have a clean data set representing the population of regular observations that can be used to train any tool.

### Fitting an elliptic envelop

One common way of performing outlier detection is to assume that the regular data come from a known distribution (e.g. data are Gaussian distributed). From this assumption, we generally try to define the “shape” of the data, and can define outlying observations as observations which stand far enough from the fit shape.

The scikit-learn provides an object `covariance.EllipticEnvelope` that fits a robust covariance estimate to the data, and thus fits an ellipse to the central data points, ignoring points outside the central mode.

For instance, assuming that the inlier data are Gaussian distributed, it will estimate the inlier location and covariance in a robust way (i.e. without being influenced by outliers). The Mahalanobis distances obtained from this estimate is used to derive a measure of outlyingness. This strategy is illustrated below.



**Examples:**

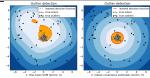
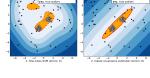
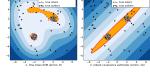
- See [Robust covariance estimation and Mahalanobis distances relevance](#) for an illustration of the difference between using a standard (`covariance.EmpiricalCovariance`) or a robust estimate (`covariance.MinCovDet`) of location and covariance to assess the degree of outlyingness of an observation.

**References:****One-class SVM versus elliptic envelop**

Strictly-speaking, the One-class SVM is not an outlier-detection method, but a novelty-detection method: it's 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 1.1: Comparing One-class SVM approach, and elliptic envelopp

<p>For a inlier mode well-centered and elliptic, the <code>svm.OneClassSVM</code> 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 <code>covariance.EllipticEnvelope</code> learns an ellipse, which fits well the inlier distribution.</p>	
<p>As the inlier distribution becomes bimodal, the <code>covariance.EllipticEnvelope</code> does not fit well the inliers. However, we can see that the <code>svm.OneClassSVM</code> tends to overfit: because it has not model of inliers, it interprets a region where, by chance some outliers are clustered, as inliers.</p>	
<p>If the inlier distribution is strongly non Gaussian, the <code>svm.OneClassSVM</code> is able to recover a reasonable approximation, whereas the <code>covariance.EllipticEnvelope</code> completely fails.</p>	

**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`.

## 1.4.7 Hidden Markov Models

`sklearn.hmm` implements the Hidden Markov Models (HMMs). The HMM is a generative probabilistic model, in which a sequence of observable  $\mathbf{X}$  variable is generated by a sequence of internal hidden state  $\mathbf{Z}$ . The hidden states can not be observed directly. The transitions between hidden states are assumed to have the form of a (first-order) Markov chain. They can be specified by the start probability vector  $\Pi$  and a transition probability matrix  $\mathbf{A}$ . The emission probability of an observable can be any distribution with parameters  $\Theta_i$  conditioned on the current hidden state (e.g. multinomial, Gaussian). The HMM is completely determined by  $\Pi$ ,  $\mathbf{A}$  and  $\Theta_i$ .

There are three fundamental problems for HMMs:

- Given the model parameters and observed data, estimate the optimal sequence of hidden states.
- Given the model parameters and observed data, calculate the likelihood of the data.
- Given just the observed data, estimate the model parameters.

The first and the second problem can be solved by the dynamic programming algorithms known as the Viterbi algorithm and the Forward-Backward algorithm, respectively. The last one can be solved by an iterative Expectation-Maximization (EM) algorithm, known as the Baum-Welch algorithm.

See the ref listed below for further detailed information.

### References:

[Rabiner89] A tutorial on hidden Markov models and selected applications in speech recognition Lawrence, R. Rabiner, 1989

## Using HMM

Classes in this module include `MultinomialHMM`, `GaussianHMM`, and `GMMHMM`. They implement HMM with emission probabilities determined by multimomial distributions, Gaussian distributions and mixtures of Gaussian distributions.

### Building HMM and generating samples

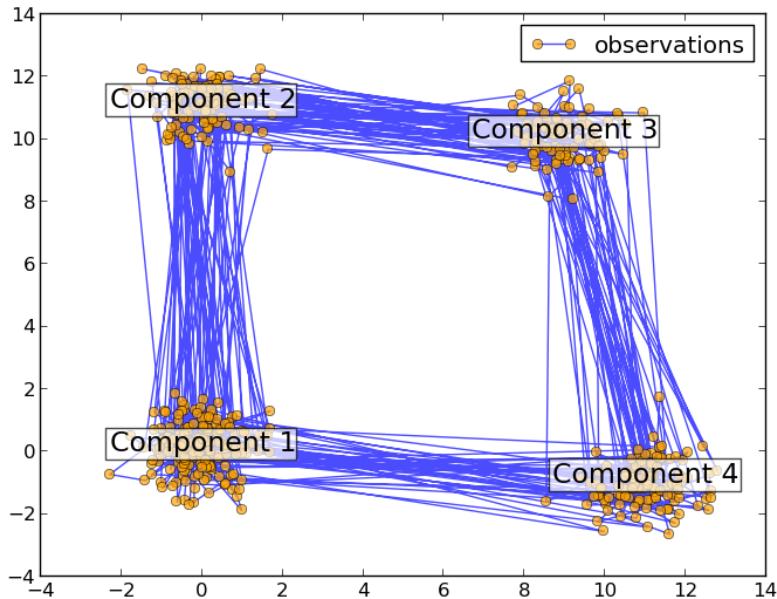
You can build an HMM instance by passing the parameters described above to the constructor. Then, you can generate samples from the HMM by calling `sample`:

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

>>> startprob = np.array([0.6, 0.3, 0.1])
>>> transmat = np.array([[0.7, 0.2, 0.1], [0.3, 0.5, 0.2], [0.3, 0.3, 0.4]])
>>> means = np.array([[0.0, 0.0], [3.0, -3.0], [5.0, 10.0]])
>>> covars = np.tile(np.identity(2), (3, 1, 1))
>>> model = hmm.GaussianHMM(3, "full", startprob, transmat)
>>> model.means_ = means
>>> model.covars_ = covars
>>> X, Z = model.sample(100)
```

### Examples:

- Demonstration of sampling from HMM



### Training HMM parameters and inferring the hidden states

You can train an HMM by calling the `fit` method. The input is “the list” of the sequence of observed value. Note, since the EM algorithm is a gradient-based optimization method, it will generally get stuck in local optima. You should try to run `fit` with various initializations and select the highest scored model. The score of the model can be calculated by the `score` method. The inferred optimal hidden states can be obtained by calling `predict` method. The `predict` method can be specified with decoder algorithm. Currently the Viterbi algorithm (`viterbi`), and maximum a posteriori estimation (`map`) are supported. This time, the input is a single sequence of observed values.:

```
>>> model2 = hmm.GaussianHMM(3, "full")
>>> model2.fit([X])
GaussianHMM(algorithm='viterbi',...
>>> Z2 = model.predict(X)
```

#### Examples:

- *Gaussian HMM of stock data*

### Implementing HMMs with custom emission probabilities

If you want to implement other emission probability (e.g. Poisson), you have to implement a new HMM class by inheriting the `_BaseHMM` and overriding the methods `__init__`, `_compute_log_likelihood`, `_set` and `_get` for additional parameters, `_initialize_sufficient_statistics`, `_accumulate_sufficient_statistics` and `_do_mstep`.

## 1.5 Model selection and evaluation

### 1.5.1 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.

To avoid **over-fitting**, we have to define two different sets : a **training set** `X_train`, `y_train` which is used for learning the parameters of a predictive model, and a **testing set** `X_test`, `y_test` which is used for evaluating the fitted predictive model.

In scikit-learn such a random split can be quickly computed with the `train_test_split` helper function. Let load the iris data set to fit a linear Support Vector Machine model 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...
```

However, by defining these two 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, test) sets.

A solution is to **split the whole data several consecutive times in different train set and test set**, and to return the averaged value of the prediction scores obtained with the different sets. Such a procedure is called **cross-validation**. 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.

#### Computing cross-validated metrics

The simplest way to use perform cross-validation in 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 and 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([ 1.    ...,  0.96...,  0.9 ... ,  0.96...,  1.        ])
```

The mean score and the standard deviation of the score estimate are hence given by:

```
>>> print "Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() / 2)
Accuracy: 0.97 (+/- 0.02)
```

By default, the score computed at each CV iteration is the `score` method of the estimator. It is possible to change this by passing a custom scoring function, e.g. from the metrics module:

```
>>> from sklearn import metrics
>>> cross_validation.cross_val_score(clf, iris.data, iris.target, cv=5,
...     score_func=metrics.f1_score)
...
array([ 1.    ...,  0.96...,  0.89...,  0.96...,  1.        ])
```

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 (depending on the absence or presence of the target array).

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.        ])
```

The available cross validation iterators are introduced in the following.

## Examples

- *Receiver operating characteristic (ROC) with cross validation,*
- *Recursive feature elimination with cross-validation,*
- *Parameter estimation using grid search with a nested cross-validation,*
- *Sample pipeline for text feature extraction and evaluation,*

## Cross validation iterators

The following sections list utilities to generate boolean masks or indices that can be used to generate dataset splits according to different cross validation strategies.

### Boolean mask vs integer indices

Most cross validators support generating both boolean masks or integer indices to select the samples from a given fold.

When the data matrix is sparse, only the integer indices will work as expected. Integer indexing is hence the default behavior (since version 0.10).

You can explicitly pass `indices=False` to the constructor of the CV object (when supported) to use the boolean mask method instead.

## K-fold

`KFold` divides all the samples in  $\mathcal{K}$  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:

```
>>> import numpy as np
>>> from sklearn.cross_validation import KFold
>>> X = np.array([[0., 0.], [1., 1.], [-1., -1.], [2., 2.]])
>>> Y = np.array([0, 1, 0, 1])

>>> kf = KFold(len(Y), n_folds=2, indices=False)
>>> print kf
sklearn.cross_validation.KFold(n=4, n_folds=2)

>>> for train, test in kf:
...     print train, test
[False False  True  True] [ True  True False False]
[ True  True False False] [False False  True  True]
```

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:

```
>>> X_train, X_test, y_train, y_test = X[train], X[test], Y[train], Y[test]
```

If `X` or `Y` are `scipy.sparse` matrices, train and test need to be integer indices. It can be obtained by setting the parameter `indices` to `True` when creating the cross-validation procedure:

```
>>> X = np.array([[0., 0.], [1., 1.], [-1., -1.], [2., 2.]])
>>> Y = np.array([0, 1, 0, 1])

>>> kf = KFold(len(Y), n_folds=2, indices=True)
>>> for train, test in kf:
...     print train, test
[2 3] [0 1]
[0 1] [2 3]
```

## Stratified K-Fold

`StratifiedKFold` is a variation of *K-fold*, which returns stratified folds, *i.e* which creates folds by preserving the same percentage for each target class as in the complete set.

Example of stratified 2-fold:

```
>>> from sklearn.cross_validation import StratifiedKFold
>>> X = [[0., 0.],
...       [1., 1.],
...       [-1., -1.],
...       [2., 2.],
...       [3., 3.],
...       [4., 4.],
...       [0., 1.]]
>>> Y = [0, 0, 0, 1, 1, 1, 0]

>>> skf = StratifiedKFold(Y, 2)
>>> print skf
sklearn.cross_validation.StratifiedKFold(labels=[0 0 0 1 1 1 0], n_folds=2)

>>> for train, test in skf:
...     print train, test
[1 4 6] [0 2 3 5]
[0 2 3 5] [1 4 6]
```

### 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 learning sets and  $n$  different tests set. This cross-validation procedure does not waste much data as only one sample is removed from the learning set:

```
>>> from sklearn.cross_validation import LeaveOneOut
>>> X = np.array([[0., 0.], [1., 1.], [-1., -1.], [2., 2.]])
>>> Y = np.array([0, 1, 0, 1])

>>> loo = LeaveOneOut(len(Y))
>>> print loo
sklearn.cross_validation.LeaveOneOut(n=4)

>>> for train, test in loo:
...     print train, test
[1 2 3] [0]
[0 2 3] [1]
[0 1 3] [2]
[0 1 2] [3]
```

### Leave-P-Out - LPO

LeavePOut is very similar to *Leave-One-Out*, as it creates all the possible training/test sets by removing  $P$  samples from the complete set.

Example of Leave-2-Out:

```
>>> from sklearn.cross_validation import LeavePOut
>>> X = [[0., 0.], [1., 1.], [-1., -1.], [2., 2.]]
>>> Y = [0, 1, 0, 1]

>>> lpo = LeavePOut(len(Y), 2)
>>> print lpo
sklearn.cross_validation.LeavePOut(n=4, p=2)
```

```
>>> for train, test in lpo:  
...     print 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 label. This label information can be used to encode arbitrary domain specific stratifications of the samples as integers.

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  
>>> X = [[0., 0.], [1., 1.], [-1., -1.], [2., 2.]]  
>>> Y = [0, 1, 0, 1]  
>>> labels = [1, 1, 2, 2]  
  
>>> lolo = LeaveOneLabelOut(labels)  
>>> print lolo  
sklearn.cross_validation.LeaveOneLabelOut(labels=[1, 1, 2, 2])  
  
>>> for train, test in lolo:  
...     print 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.

### 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  
>>> X = [[0., 0.], [1., 1.], [-1., -1.], [2., 2.], [3., 3.], [4., 4.]]  
>>> Y = [0, 1, 0, 1, 0, 1]  
>>> labels = [1, 1, 2, 2, 3, 3]  
  
>>> lplo = LeavePLabelOut(labels, 2)  
>>> print lplo  
sklearn.cross_validation.LeavePLabelOut(labels=[1, 1, 2, 2, 3, 3], p=2)  
  
>>> for train, test in lplo:  
...     print 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 splitted 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)
>>> len(ss)
3
>>> print ss
ShuffleSplit(5, n_iter=3, test_size=0.25, indices=True, ...)

>>> for train_index, test_index in ss:
...     print 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.

### 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.

## Bootstrapping cross-validation

### Bootstrap

`Bootstrapping` is a general statistics technique that iterates the computation of an estimator on a resampled dataset.

The `Bootstrap` iterator will generate a user defined number of independent train / test dataset splits. Samples are then drawn (with replacement) on each side of the split. It furthermore possible to control the size of the train and test subset to make their union smaller than the total dataset if it is very large.

---

**Note:** Contrary to other cross-validation strategies, bootstrapping will allow some samples to occur several times in each splits.

---

```
>>> bs = cross_validation.Bootstrap(9, random_state=0)
>>> len(bs)
3
```

```
>>> print bs
Bootstrap(9, n_iter=3, train_size=5, test_size=4, random_state=0)

>>> for train_index, test_index in bs:
...     print train_index, test_index
...
[1 8 7 7 8] [0 3 0 5]
[5 4 2 4 2] [6 7 1 0]
[4 7 0 1 1] [5 3 6 5]
```

## Cross validation and model selection

Cross validation iterators can also be used to directly perform model selection using Grid Search for the optimal hyperparameters of the model. This is the topic of the next section: *Grid Search: setting estimator parameters*.

### 1.5.2 Grid Search: setting estimator parameters

Grid Search is used to optimize the parameters of a model (e.g. C, kernel and gamma for Support Vector Classifier, alpha for Lasso, etc.) using an internal *Cross-Validation: evaluating estimator performance* scheme).

#### GridSearchCV

The main class for implementing hyperparameters grid search in scikit-learn is `grid_search.GridSearchCV`. This class is passed a base model instance (for example `sklearn.svm.SVC()`) along with a grid of potential hyper-parameter values specified with the `param_grid` attribute. 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 RBG kernel, and the cross-product of C values ranging in [1, 10, 100, 1000] and gamma values in [0.001, 0.0001].

The `grid_search.GridSearchCV` instance implements the usual estimator API: when “fitting” it on a dataset all the possible combinations of hyperparameter values are evaluated and the best combinations are retained.

#### Model selection: development and evaluation

Model selection with `GridSearchCV` can be seen as a way to use the labeled data to “train” the hyper-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.

## Examples

- See *Parameter estimation using grid search with a nested 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.

---

**Note:** Computations can be run in parallel if your OS supports it, by using the keyword `n_jobs=-1`, see function signature for more details.

---

## Alternatives to brute force grid 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:

<code>linear_model.RidgeCV([alphas, ...])</code>	Ridge regression with built-in cross-validation.
<code>linear_model.RidgeClassifierCV([alphas, ...])</code>	Ridge classifier with built-in cross-validation.
<code>linear_model.LarsCV([fit_intercept, ...])</code>	Cross-validated Least Angle Regression model
<code>linear_model.LassoLarsCV([fit_intercept, ...])</code>	Cross-validated Lasso, using the LARS algorithm
<code>linear_model.LassoCV([eps, n_alphas, ...])</code>	Lasso linear model with iterative fitting along a regularization path
<code>linear_model.ElasticNetCV([l1_ratio, eps, ...])</code>	Elastic Net model with iterative fitting along a regularization path

### `sklearn.linear_model.RidgeCV`

```
class sklearn.linear_model.RidgeCV(alphas=array([ 0.1,  1., 10.]), fit_intercept=True, normalize=False, score_func=None, loss_func=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.

#### 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  $(2 \times C)^{-1}$  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

If True, the regressors X are normalized

#### `score_func`: callable, optional :

function that takes 2 arguments and compares them in order to evaluate the performance of prediction (big is good) if None is passed, the score of the estimator is maximized

**loss\_func: callable, optional :**

function that takes 2 arguments and compares them in order to evaluate the performance of prediction (small is good) if None is passed, the score of the estimator is maximized

**cv : cross-validation generator, optional**

If None, Generalized Cross-Validation (efficient Leave-One-Out) will be used.

**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, otherwise use eigen  
'svd' : force computation via singular value decomposition of X  
'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 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).

**See Also:****Ridge**Ridge regression**RidgeClassifier**Ridge classifier**RidgeClassifierCV**Ridge classifier with built-in cross validation**Attributes**

<code>cv_values_</code>	array, shape = [n_samples, n_alphas] or shape = [n_samples, n_targets, n_alphas], optional	Cross-validation values for each alpha (if <code>store_cv_values=True</code> and <code>cv=None</code> ). After <code>fit()</code> has been called, this attribute will contain the mean squared errors (by default) or the values of the <code>{loss,score}_func</code> function (if provided in the constructor).
<code>coef_</code>	array, shape = [n_features] or [n_targets, n_features]	Weight vector(s).
<code>alpha_</code>	float	Estimated regularization parameter.

**Methods**

<code>decision_function(X)</code>	Decision function of the linear model
<code>fit(X, y[, sample_weight])</code>	Fit Ridge regression model
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

---

**\_\_init\_\_(alphas=array([ 0.1, 1., 10.]), fit\_intercept=True, normalize=False, score\_func=None, loss\_func=None, cv=None, gcv\_mode=None, store\_cv\_values=False)**

**decision\_function(X)**

Decision function of the linear model

**Parameters X** : numpy array of shape [n\_samples, n\_features]

**Returns C** : array, shape = [n\_samples]

Returns predicted values.

**fit(X, y, sample\_weight=1.0)**

Fit Ridge regression model

**Parameters X** : 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

**Returns self** : Returns self.

**get\_params(deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**predict(X)**

Predict using the linear model

**Parameters X** : numpy array of shape [n\_samples, n\_features]

**Returns C** : array, shape = [n\_samples]

Returns predicted values.

**score(X, y)**

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as  $(1 - \frac{u}{v})$ , where u is the regression sum of squares  $((y - y_{\text{pred}})^2 \cdot \text{sum}())$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}.mean()})^2 \cdot \text{sum}())$ . Best possible score is 1.0, lower values are worse.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params(\*\*params)**

Set the parameters of the 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.RidgeClassifierCV**

```
class sklearn.linear_model.RidgeClassifierCV(alphas=array([ 0.1,    1.,   10.]),  
                                             fit_intercept=True, normalize=False,  
                                             score_func=None, loss_func=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

If True, the regressors X are normalized

**score\_func:** callable, optional :

function that takes 2 arguments and compares them in order to evaluate the performance of prediction (big is good) if None is passed, the score of the estimator is maximized

**loss\_func:** callable, optional :

function that takes 2 arguments and compares them in order to evaluate the performance of prediction (small is good) if None is passed, the score of the estimator is maximized

**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.

**See Also:**

**Ridge**Ridge regression

**RidgeClassifier**Ridge classifier

**RidgeCV**Ridge regression with built-in cross validation

**Notes**

For multi-class classification, n\_class classifiers are trained in a one-versus-all approach. Concretely, this is implemented by taking advantage of the multi-variate response support in Ridge.

## Attributes

<code>cv_values_</code>	array, shape = [n_samples, n_alphas] or shape = [n_samples, n_responses, n_alphas], optional	Cross-validation values for each alpha (if <code>store_cv_values=True</code> and
<code>cv=None</code> ). After <code>fit()</code> has been called, this attribute will contain the mean squared errors (by default) or the values of the <code>{loss,score}_func</code> function (if provided in the constructor).		
<code>coef_</code>	array, shape = [n_features] or [n_targets, n_features]	Weight vector(s).
<code>alpha_</code>	float	Estimated regularization parameter

## Methods

<code>decision_function(X)</code>	Predict confidence scores for samples.
<code>fit(X, y[, sample_weight, class_weight])</code>	Fit the ridge classifier.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class labels for samples in X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(alphas=array([ 0.1, 1., 10.]), fit_intercept=True, normalize=False, score_func=None, loss_func=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.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Samples.

**Returns** array, shape = [n\_samples] if n\_classes == 2 else [n\_samples,n\_classes] :

Confidence scores per (sample, class) combination. In the binary case, confidence score for the “positive” class.

`fit(X, y, sample_weight=1.0, class_weight=None)`

Fit the ridge classifier.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features.

`y` : array-like, shape = [n\_samples]

Target values.

`sample_weight` : float or numpy array of shape [n\_samples]

Sample weight

**Returns self** : object

Returns self.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**predict (X)**

Predict class labels for samples in X.

**Parameters X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Samples.

**Returns C** : array, shape = [n\_samples]

Predicted class label per sample.

**score (X, y)**

Returns the mean accuracy on the given test data and labels.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns z** : float

**set\_params (\*\*params)**

Set the parameters of the 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.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

**Parameters 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

If True, the regressors X are normalized

**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** : crossvalidation 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.

## See Also:

`lars_path`, `LassoLars`, `LassoLarsCV`

## Attributes

<code>coef_</code>	array, shape = [n_features]	parameter vector (w in the formulation formula)
<code>intercept_</code>	float	independent term in decision function
<code>coef_path_</code> : array, shape = [n_features, n_alpha]		the varying values of the coefficients along the path
<code>alpha_</code> : float		the estimated regularization parameter alpha
<code>alphas_</code> : array, shape = [n_alpha]		the different values of alpha along the path
<code>cv_alphas_</code> : array, shape = [n_cv_alphas]		all the values of alpha along the path for the different folds
<code>cv_mse_path_</code> : 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)

## 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 the estimator

---

`predict(X)` Predict using the linear model

---

`score(X, y)` Returns the coefficient of determination R^2 of the prediction.

---

`set_params(**params)` Set the parameters of the 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** : numpy array of shape [n\_samples, n\_features]

**Returns C** : array, shape = [n\_samples]

Returns predicted values.

**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 the estimator

**Parameters deep: boolean, optional :**

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

**predict(X)**

Predict using the linear model

**Parameters X** : numpy array of shape [n\_samples, n\_features]

**Returns C** : array, shape = [n\_samples]

Returns predicted values.

**score(X, y)**

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  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params(\*\*params)**

Set the parameters of the 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.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:

$$(1 / (2 * n\_samples)) * ||y - Xw||^2_2 + \alpha * ||w||_1$$

**Parameters****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

If True, the regressors X are normalized

**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** : crossvalidation 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.

**See Also:**

`lars_path`, `LassoLars`, `LarsCV`, `LassoCV`

**Notes**

The object solves the same problem as the `LassoCV` object. However, unlike the `LassoCV`, it finds 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.

## Attributes

<code>coef_</code>	array, shape = [n_features]	parameter vector (w in the formulation formula)
<code>intercept_</code>	float	independent term in decision function.
<code>coef_path_</code> : array, shape = [n_features, n_alpha]		the varying values of the coefficients along the path
<code>alpha_</code> : float		the estimated regularization parameter alpha
<code>alphas_</code> : array, shape = [n_alpha]		the different values of alpha along the path
<code>cv_alphas_</code> : array, shape = [n_cv_alphas]		all the values of alpha along the path for the different folds
<code>cv_mse_path_</code> : array, shape = [n_folds, n_cv_alphas]		the mean square error on left-out for each fold along the path (alpha values given by <code>cv_alphas</code> )

## Methods

<code>decision_function(X)</code>	Decision function of the linear model
<code>fit(X, y)</code>	Fit the model using X, y as training data.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the 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` : numpy array of shape [n\_samples, n\_features]

**Returns** `C` : array, shape = [n\_samples]

Returns predicted values.

**`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 the estimator

**Parameters deep: boolean, optional :**

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

**predict (X)**

Predict using the linear model

**Parameters X** : numpy array of shape [n\_samples, n\_features]

**Returns C** : array, shape = [n\_samples]

Returns predicted values.

**score (X, y)**

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  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params (\*\*params)**

Set the parameters of the 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.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)
```

Lasso linear model with iterative fitting along a regularization path

The best model is selected by cross-validation.

The optimization objective for Lasso is:

$$(1 / (2 * n\_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1$$

**Parameters eps** : float, optional

Length of the path. eps=1e-3 means that alpha\_min / alpha\_max = 1e-3.

**n\_alphas** : int, optional

Number of alphas along the regularization path

**alphas** : numpy array, optional

List of alphas where to compute the models. If None alphas are set automatically

**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 crossvalidation generator, optional**

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

**verbose : bool or integer**

amount of verbosity

**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.

**Attributes**

<code>alpha_</code> : float		The amount of penalization choosen by cross validation
<code>coef_</code>	array, shape = (n_features,)	parameter vector ( $w$ in the cost function formula)
<code>intercept_</code>	float	independent term in decision function.
<code>mse_path_</code> : array, shape = (n_alphas, n_folds)		mean square error for the test set on each fold, varying alpha
<code>alphas_</code> : numpy array		The grid of alphas used for fitting

**Methods**

<code>decision_function(X)</code>	Decision function of the linear model
<code>fit(X, y)</code>	Fit linear model with coordinate descent
<code>get_params([deep])</code>	Get parameters for the estimator
<code>path(X, y[, eps, n_alphas, alphas, ...])</code>	Compute Lasso path with coordinate descent
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

---

**\_\_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*)**

**decision\_function(*X*)**

Decision function of the linear model

**Parameters X** : numpy array of shape [n\_samples, n\_features]

**Returns C** : 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.

**Parameters X** : array-like, shape (n\_samples, n\_features)

Training data. Pass directly as fortran contiguous data to avoid unnecessary memory duplication

**y** : ndarray, shape (n\_samples,) or (n\_samples, n\_targets)

Target values

**get\_params(deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**static path(*X, y, eps=0.001, n\_alphas=100, alphas=None, precompute='auto', Xy=None, fit\_intercept=True, normalize=False, copy\_X=True, verbose=False, \*\*params*)**

Compute Lasso path with coordinate descent

The optimization objective for Lasso is:

$$(1 / (2 * n\_samples)) * ||y - Xw||^2_2 + \alpha * ||w||_1$$

**Parameters X** : ndarray, shape = (n\_samples, n\_features)

Training data. Pass directly as fortran contiguous data to avoid unnecessary memory duplication

**y** : ndarray, shape = (n\_samples,)

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.

**fit\_intercept** : bool

Fit or not an intercept

**normalize** : boolean, optional

If True, the regressors X are normalized

**copy\_X** : boolean, optional, default True

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

**verbose** : bool or integer

Amount of verbosity

**params** : kwargs

keyword arguments passed to the Lasso objects

**Returns models** : a list of models along the regularization path

#### 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.

**predict** ( $X$ )

Predict using the linear model

**Parameters**  $X$  : numpy array of shape [n\_samples, n\_features]

**Returns**  $C$  : array, shape = [n\_samples]

Returns predicted values.

**score** ( $X, y$ )

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  $((y - y_{pred})^2).sum()$  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]

Training set.

$y$  : array-like, shape = [n\_samples]

**Returns**  $z$  : float

---

**set\_params** (\*\**params*)  
Set the parameters of the 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.ElasticNetCV
class sklearn.linear_model.ElasticNetCV(l1_ratio=0.5, eps=0.001, n_alphas=100, al-
phas=None, fit_intercept=True, normalize=False,
precompute='auto', max_iter=1000, tol=0.0001,
cv=None, copy_X=True, verbose=0, n_jobs=1,
rho=None)
```

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

**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 crossvalidation generator, optional

If an integer is passed, it is the number of fold (default 3). Specific crossvalidation objects can be passed, see `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.

**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).
```

**Attributes**

<code>alpha_</code>	float	The amount of penalization choosen by cross validation
<code>l1_ratio_</code>	float	The compromise between l1 and l2 penalization choosen by cross validation
<code>coef_</code>	array, shape = (n_features,)	Parameter vector (w in the cost function formula),
<code>intercept_</code>	float	Independent term in the decision function.
<code>mse_path</code>	array, shape = (n_l1_ratio, n_alpha, n_folds)	Mean square error for the test set on each fold, varying l1_ratio and alpha.

**Methods**

<code>decision_function(X)</code>	Decision function of the linear model
<code>fit(X, y)</code>	Fit linear model with coordinate descent
<code>get_params([deep])</code>	Get parameters for the estimator
<code>path(X, y[, l1_ratio, eps, n_alphas, ...])</code>	Compute Elastic-Net path with coordinate descent
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

---

```
__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, rho=None)
```

**decision\_function(X)**

Decision function of the linear model

**Parameters X** : numpy array of shape [n\_samples, n\_features]

**Returns C** : 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.

**Parameters X** : array-like, shape (n\_samples, n\_features)

Training data. Pass directly as fortran contiguous data to avoid unnecessary memory duplication

**y** : ndarray, shape (n\_samples,) or (n\_samples, n\_targets)

Target values

**get\_params(deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

```
static path(X, y, l1_ratio=0.5, eps=0.001, n_alphas=100, alphas=None, precompute='auto', Xy=None, fit_intercept=True, normalize=False, copy_X=True, verbose=False, rho=None, **params)
```

Compute Elastic-Net path with coordinate descent

The Elastic Net optimization function is:

$$\begin{aligned} & \frac{1}{2} * (2 * n\_samples) * ||y - Xw||^2_2 + \\ & + \text{alpha} * l1\_ratio * ||w||_1 \\ & + 0.5 * \text{alpha} * (1 - l1\_ratio) * ||w||^2_2 \end{aligned}$$

**Parameters X** : ndarray, shape = (n\_samples, n\_features)

Training data. Pass directly as fortran contiguous data to avoid unnecessary memory duplication

**y** : ndarray, shape = (n\_samples,)

Target values

**l1\_ratio** : float, optional

float between 0 and 1 passed to ElasticNet (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.

**fit\_intercept** : bool

Fit or not an intercept

**normalize** : boolean, optional

If True, the regressors X are normalized

**copy\_X** : boolean, optional, default True

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

**verbose** : bool or integer

Amount of verbosity

**params** : kwargs

keyword arguments passed to the Lasso objects

**Returns** **models** : a list of models along the regularization path

#### See Also:

ElasticNet, ElasticNetCV

#### Notes

See examples/plot\_lasso\_coordinate\_descent\_path.py for an example.

**predict** (*X*)

Predict using the linear model

**Parameters** **X** : numpy array of shape [n\_samples, n\_features]

**Returns** **C** : array, shape = [n\_samples]

Returns predicted values.

**rho**

DEPRECATED: rho was renamed to `l1_ratio` and will be removed in 0.15

**score** (*X*, *y*)

Returns the coefficient of determination R<sup>2</sup> of the prediction.

The coefficient R<sup>2</sup> is defined as  $(1 - u/v)$ , where  $u$  is the regression sum of squares  $((y - y_{pred})^2).sum()$  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]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params** (\*\*params)

Set the parameters of the 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** :

## 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 Akaike Information Criterion (AIC) or the Bayesian Information Criterion (BIC) for automated model selection:

---

`linear_model.LassoLarsIC([criterion, ...])` Lasso model fit with Lars using BIC or AIC for model selection

---

### `sklearn.linear_model.LassoLarsIC`

```
class sklearn.linear_model.LassoLarsIC(criterion='aic', fit_intercept=True, verbose=False,
                                         normalize=True, precompute='auto', max_iter=500,
                                         eps=2.2204460492503131e-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.

**Parameters** `criterion`: ‘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

If True, the regressors X are normalized

**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.

**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/Akaike_information_criterion) [http://en.wikipedia.org/wiki/Bayesian\\_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...]
```

**Attributes**

<code>coef_</code>	array, shape = [n_features]	parameter vector ( $w$ in the formulation formula)
<code>intercept_</code>	float	independent term in decision function.
<code>alpha_</code>	float	the alpha parameter chosen by the information criterion

**Methods**

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

---

```
__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

**Parameters X** : numpy array of shape [n\_samples, n\_features]

**Returns C** : array, shape = [n\_samples]

Returns predicted values.

**fit(X, y, copy\_X=True)**

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 the estimator

**Parameters deep: boolean, optional :**

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

**predict(X)**

Predict using the linear model

**Parameters X** : numpy array of shape [n\_samples, n\_features]

**Returns C** : array, shape = [n\_samples]

Returns predicted values.

**score(X, y)**

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  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params(\*\*params)**

Set the parameters of the 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** :

## 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:

<code>ensemble.RandomForestClassifier(...)</code>	A random forest classifier.
<code>ensemble.RandomForestRegressor(...)</code>	A random forest regressor.
<code>ensemble.ExtraTreesClassifier(...)</code>	An extra-trees classifier.
<code>ensemble.ExtraTreesRegressor([n_estimators, ...])</code>	An extra-trees regressor.
<code>ensemble.GradientBoostingClassifier([loss, ...])</code>	Gradient Boosting for classification.
<code>ensemble.GradientBoostingRegressor([loss, ...])</code>	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_density=0.1,
                                             max_features='auto', bootstrap=True, compute_importances=False, oob_score=False,
                                             n_jobs=1, random_state=None, verbose=0)
```

A random forest classifier.

A random forest is a meta estimator that fits a number of classifical decision trees 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, string or None, optional (default="auto")

**The number of features to consider when looking for the best split:**

- If “auto”, then  $\text{max\_features} = \sqrt{n\_features}$  on classification tasks and  $\text{max\_features} = n\_features$  on regression problems.
- If “sqrt”, then  $\text{max\_features} = \sqrt{n\_features}$ .
- If “log2”, then  $\text{max\_features} = \log_2(n\_features)$ .
- If None, then  $\text{max\_features} = n\_features$ .

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. 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 than `min_samples_leaf` samples. Note: this parameter is tree-specific.

**min\_density** : float, optional (default=0.1)

This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the `sample_mask` (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If `min_density` equals to one, the partitions are always represented as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks). Note: this parameter is tree-specific.

**bootstrap** : boolean, optional (default=True)

Whether bootstrap samples are used when building trees.

**compute\_importances** : boolean, optional (default=True)

Whether feature importances are computed and stored into the `feature_importances_` attribute when calling `fit`.

**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. 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.

## See Also:

`DecisionTreeClassifier`, `ExtraTreesClassifier`

## References

[R82]

## Attributes

<i>estimators_</i> : list of DecisionTreeClassifier		The collection of fitted sub-estimators.
<i>classes_</i> : 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).
<i>n_classes_</i> : int or list		The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).
<i>feature_importances_</i>	array of shape = [n_features]	The feature importances (the higher, the more important the feature).
<i>oob_score_</i>	float	Score of the training dataset obtained using an out-of-bag estimate.
<i>oob_decision_function_</i>	array of shape = [n_samples, n_classes]	Decision function computed with out-of-bag estimate on the training set.

## Methods

<code>apply(X)</code>	Apply trees in the forest to X, return leaf indices.
<code>fit(X, y[, sample_weight])</code>	Build a forest of trees from the training set (X, y).
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class for X.
<code>predict_log_proba(X)</code>	Predict class log-probabilities for X.
<code>predict_proba(X)</code>	Predict class probabilities for X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

`__init__(n_estimators=10, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_density=0.1, max_features='auto', bootstrap=True, compute_importances=False, oob_score=False, n_jobs=1, random_state=None, verbose=0)`

### `apply(X)`

Apply trees in the forest to X, return leaf indices.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Input data.

**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 of shape = [n\_samples, n\_features]

The training input samples.

`y` : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]

The target values (integers that correspond to classes 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.

**Returns X\_new** : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params**(deep=True)

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**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.

**Parameters X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns y** : 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 mean predicted class log-probabilities of the trees in the forest.

**Parameters X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns p** : 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. Classes are ordered by arithmetical order.

**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.

**Parameters** **X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**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. Classes are ordered by arithmetical order.

**score**(*X*, *y*)

Returns the mean accuracy on the given test data and labels.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for *X*.

**Returns** **z** : float

**set\_params**(\*\*params)

Set the parameters of the 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.

**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.

**sklearn.ensemble.RandomForestRegressor**

```
class sklearn.ensemble.RandomForestRegressor(n_estimators=10, criterion='mse',
                                             max_depth=None, min_samples_split=2,
                                             min_samples_leaf=1, min_density=0.1,
                                             max_features='auto', bootstrap=True, compute_importances=False, oob_score=False,
                                             n_jobs=1, random_state=None, verbose=0)
```

A random forest regressor.

A random forest is a meta estimator that fits a number of classifical decision trees 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="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, string or None, optional (default="auto")

**The number of features to consider when looking for the best split:**

- If "auto", then  $\text{max\_features} = \sqrt{n\_features}$  on classification tasks and  $\text{max\_features} = n\_features$  on regression problems.
- If "sqrt", then  $\text{max\_features} = \sqrt{n\_features}$ .
- If "log2", then  $\text{max\_features} = \log_2(n\_features)$ .
- If None, then  $\text{max\_features} = n\_features$ .

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. 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_density` : float, optional (default=0.1)

This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the `sample_mask` (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If `min_density` equals to one, the partitions are always represented as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks). Note: this parameter is tree-specific.

`bootstrap` : boolean, optional (default=True)

Whether bootstrap samples are used when building trees.

`compute_importances` : boolean, optional (default=True)

Whether feature importances are computed and stored into the `feature_importances_` attribute when calling `fit`.

`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. 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.

#### See Also:

`DecisionTreeRegressor`, `ExtraTreesRegressor`

#### References

[R83]

#### Attributes

<i>estimators_</i> : list of DecisionTreeRegressor		The collection of fitted sub-estimators.
<i>feature_importances_</i>	array of shape = [n_features]	The feature importances (the higher, the more important the feature).
<i>oob_score_</i>	float	Score of the training dataset obtained using an out-of-bag estimate.
<i>oob_prediction_</i>	array of shape = [n_samples]	Prediction computed with out-of-bag estimate on the training set.

#### Methods

<code>apply(X)</code>	Apply trees in the forest to X, return leaf indices.
<code>fit(X, y[, sample_weight])</code>	Build a forest of trees from the training set (X, y).
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict regression target for X.
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

**`__init__(n_estimators=10, criterion='mse', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_density=0.1, max_features='auto', bootstrap=True, compute_importances=False, oob_score=False, n_jobs=1, random_state=None, verbose=0)`**

**`apply(X)`**

Apply trees in the forest to X, return leaf indices.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Input data.

**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 of shape = [n\_samples, n\_features]

The training input samples.

 $y$  : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]

The target values (integers that correspond to classes 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**( $X, y=None, \text{**fit\_params}$ )

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**  $X_{\text{new}}$  : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params**( $deep=True$ )

Get parameters for the estimator

**Parameters**  $deep$ : boolean, optional :

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

**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.

**Parameters**  $X$  : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns**  $y$ : array of shape = [n\_samples] or [n\_samples, n\_outputs] :

The predicted values.

**score**(X, y)

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  $((y - y_{\text{pred}})^2 \cdot \text{sum}()$  and  $v$  is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2 \cdot \text{sum}()$ . Best possible score is 1.0, lower values are worse.

**Parameters X**: array-like, shape = [n\_samples, n\_features]

Training set.

**y**: array-like, shape = [n\_samples]**Returns z**: float**set\_params**(\*\*params)

Set the parameters of the 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.

**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.

**sklearn.ensemble.ExtraTreesClassifier**

```
class sklearn.ensemble.ExtraTreesClassifier(n_estimators=10, criterion='gini',
                                            max_depth=None, min_samples_split=2,
                                            min_samples_leaf=1, min_density=0.1,
                                            max_features='auto', bootstrap=False, compute_importances=False, oob_score=False,
                                            n_jobs=1, random_state=None, verbose=0)
```

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.

**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, string or None, optional (default="auto")

The number of features to consider when looking for the best split.

- If “auto”, then  $\text{max\_features}=\sqrt{n\_features}$  on classification tasks and  $\text{max\_features}=n\_features$  on regression problems.
- If “sqrt”, then  $\text{max\_features}=\sqrt{n\_features}$ .
- If “log2”, then  $\text{max\_features}=\log_2(n\_features)$ .
- If None, then  $\text{max\_features}=n\_features$ .

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. 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 than `min_samples_leaf` samples. Note: this parameter is tree-specific.

**min\_density** : float, optional (default=0.1)

This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the `sample_mask` (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If `min_density` equals to one, the partitions are always represented as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks). Note: this parameter is tree-specific.

**bootstrap** : boolean, optional (default=False)

Whether bootstrap samples are used when building trees.

**compute\_importances** : boolean, optional (default=True)

Whether feature importances are computed and stored into the `feature_importances_` attribute when calling `fit`.

**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. 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.

**See Also:**

`sklearn.tree.ExtraTreeClassifier`Base classifier for this ensemble.

`RandomForestClassifier`Ensemble Classifier based on trees with optimal splits.

**References**

[R80]

**Attributes**

<code>estimators_</code> : list of DecisionTreeClassifier		The collection of fitted sub-estimators.
<code>classes_</code> : 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).
<code>n_classes_</code> : int or list		The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).
<code>feature_importances_</code>	array of shape = [n_features]	The feature importances (the higher, the more important the feature).
<code>oob_score_</code>	float	Score of the training dataset obtained using an out-of-bag estimate.
<code>oob_decision_function_</code>	array of shape = [n_samples, n_classes]	Decision function computed with out-of-bag estimate on the training set.

**Methods**

<code>apply(X)</code>	Apply trees in the forest to X, return leaf indices.
<code>fit(X, y[, sample_weight])</code>	Build a forest of trees from the training set (X, y).
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class for X.
<code>predict_log_proba(X)</code>	Predict class log-probabilities for X.
<code>predict_proba(X)</code>	Predict class probabilities for X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

`__init__(n_estimators=10, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_density=0.1, max_features='auto', bootstrap=False, compute_importances=False, oob_score=False, n_jobs=1, random_state=None, verbose=0)`

`apply(X)`

Apply trees in the forest to X, return leaf indices.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Input data.

**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 of shape = [n\_samples, n\_features]

The training input samples.

**y** : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]

The target values (integers that correspond to classes 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.

**Returns X\_new** : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params** (*deep=True*)

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**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.

**Parameters X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns y** : 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 mean predicted class log-probabilities of the trees in the forest.

**Parameters** **X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns** **p** : 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. Classes are ordered by arithmetical order.

**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.

**Parameters** **X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**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. Classes are ordered by arithmetical order.

**score**(*X*, *y*)

Returns the mean accuracy on the given test data and labels.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for *X*.

**Returns** **z** : float

**set\_params**(\**params*)

Set the parameters of the 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.

**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.

### `sklearn.ensemble.ExtraTreesRegressor`

```
class sklearn.ensemble.ExtraTreesRegressor(n_estimators=10, criterion='mse',
                                           max_depth=None, min_samples_split=2,
                                           min_samples_leaf=1, min_density=0.1,
                                           max_features='auto', bootstrap=False, compute_importances=False, oob_score=False,
                                           n_jobs=1, random_state=None, verbose=0)
```

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.

**Parameters** `n_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, string or None, optional (default="auto")

**The number of features to consider when looking for the best split:**

- If "auto", then `max_features=sqrt(n_features)` on classification tasks and `max_features=n_features` on regression problems.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_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. 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 than `min_samples_leaf` samples. Note: this parameter is tree-specific.

`min_density` : float, optional (default=0.1)

This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the `sample_mask` (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If `min_density` equals to one, the partitions are always represented

as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks). 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.

**compute\_importances** : boolean, optional (default=True)

Whether feature importances are computed and stored into the `feature_importances_` attribute when calling `fit`.

**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. 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.

#### See Also:

`sklearn.tree.ExtraTreeRegressor`Base estimator for this ensemble.

`RandomForestRegressor`Ensemble regressor using trees with optimal splits.

#### References

[R81]

#### Attributes

<code>estimators_</code> : list of <code>DecisionTreeRegressor</code>		The collection of fitted sub-estimators.
<code>feature_importances_</code>	array of shape = [n_features]	The feature importances (the higher, the more important the feature).
<code>oob_score_</code>	float	Score of the training dataset obtained using an out-of-bag estimate.
<code>oob_prediction_</code>	array of shape = [n_samples]	Prediction computed with out-of-bag estimate on the training set.

#### Methods

`apply(X)`

Apply trees in the forest to X, return leaf indices.

Continued on next page

**Table 1.15 – continued from previous page**

<code>fit(X, y[, sample_weight])</code>	Build a forest of trees from the training set (X, y).
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict regression target for X.
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

**`__init__(n_estimators=10, criterion='mse', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_density=0.1, max_features='auto', bootstrap=False, compute_importances=False, oob_score=False, n_jobs=1, random_state=None, verbose=0)`**

### **apply(X)**

Apply trees in the forest to X, return leaf indices.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Input data.

**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 of shape = [n\_samples, n\_features]

The training input samples.

**y** : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]

The target values (integers that correspond to classes 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.

**Returns** **X\_new** : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**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.

**Parameters X : array-like of shape = [n\_samples, n\_features]**

The input samples.

**Returns y: array of shape = [n\_samples] or [n\_samples, n\_outputs] :**

The predicted values.

**score (X, y)**

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  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters X : array-like, shape = [n\_samples, n\_features]**

Training set.

**y : array-like, shape = [n\_samples]****Returns z : float****set\_params (\*\*params)**

Set the parameters of the 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.

**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.

**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, max_depth=3,
                                                init=None, random_state=None,
                                                max_features=None, verbose=0,
                                                learn_rate=None)
```

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’}, optional (default=‘deviance’)

loss function to be optimized. ‘deviance’ refers to deviance (= logistic regression) for classification with probabilistic outputs.

**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 overfitting 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.

**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.

**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, None, optional (default=None)

The number of features to consider when looking for the best split. Features are chosen randomly at each split point. If None, then `max_features=n_features`. Choosing `max_features < n_features` leads to a reduction of variance and an increase in bias.

**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 ‘.’ for every tree built. If greater than 1 then it prints the score for every tree.

#### See Also:

`sklearn.tree.DecisionTreeClassifier`, `RandomForestClassifier`

#### 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.

#### Examples

```
>>> samples = [[0, 0, 2], [1, 0, 0]]
>>> labels = [0, 1]
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> gb = GradientBoostingClassifier().fit(samples, labels)
>>> print(gb.predict([[0.5, 0, 0]]))
[0]
```

#### Attributes

<code>feature_importances_</code>	array, shape = [n_features]	The feature importances (the higher, the more important the feature).
<code>oob_score_</code>	array, shape = [n_estimators]	Score of the training dataset obtained using an out-of-bag estimate. The i-th score <code>oob_score_[i]</code> is the deviance (= loss) of the model at iteration <code>i</code> on the out-of-bag sample.
<code>train_score_</code>	array, shape = [n_estimators]	The i-th score <code>train_score_[i]</code> is the deviance (= loss) of the model at iteration <code>i</code> on the in-bag sample. If <code>subsample == 1</code> this is the deviance on the training data.
<code>loss_</code>	LossFunction	The concrete <code>LossFunction</code> object.
<code>init</code>	BaseEstimator	The estimator that provides the initial predictions. Set via the <code>init</code> argument or <code>loss.init_estimator</code> .
<code>estimators_</code> : list of Decision-TreeRegressor		The collection of fitted sub-estimators.

#### Methods

<code>decision_function(X)</code>	Compute the decision function of X.
<code>fit(X, y)</code>	Fit the gradient boosting model.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class for X.

Continued on next page

**Table 1.16 – continued from previous page**

<code>predict_proba(X)</code>	Predict class probabilities for X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>staged_decision_function(X)</code>	Compute decision function of X for each iteration.
<code>staged_predict(X)</code>	Predict class probabilities at each stage for X.
<code>staged_predict_proba(X)</code>	Predict class probabilities at each stage for X.

`__init__(loss='deviance', learning_rate=0.1, n_estimators=100, subsample=1.0, min_samples_split=2, min_samples_leaf=1, max_depth=3, init=None, random_state=None, max_features=None, verbose=0, learn_rate=None)`

#### `decision_function(X)`

Compute the decision function of X.

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns** `score` : array, shape = [n\_samples, k]

The decision function of the input samples. Classes are ordered by arithmetical order. Regression and binary classification are special cases with k == 1, otherwise k==n\_classes.

#### `fit(X, y)`

Fit the gradient boosting model.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features. Use fortran-style to avoid memory copies.

`y` : array-like, shape = [n\_samples]

Target values (integers in classification, real numbers in regression) For classification, labels must correspond to classes 0, 1, ..., n\_classes\_-1

**Returns** `self` : object

Returns self.

#### `get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep: boolean, optional` :

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

#### `predict(X)`

Predict class for X.

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns** `y` : array of shape = [n\_samples]

The predicted classes.

#### `predict_proba(X)`

Predict class probabilities for X.

**Parameters** **X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns** **p** : array of shape = [n\_samples]

The class probabilities of the input samples. Classes are ordered by arithmetical order.

**score** (*X*, *y*)

Returns the mean accuracy on the given test data and labels.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for *X*.

**Returns** **z** : float

**set\_params** (\*\**params*)

Set the parameters of the 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** :

**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. Classes are ordered by arithmetical order. Regression and binary classification are special cases with k == 1, otherwise k==n\_classes.

**staged\_predict** (*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** : array of shape = [n\_samples]

The predicted value of the input samples.

**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** : array of shape = [n\_samples]

The predicted value of the input samples.

### `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,      max_depth=3,
                                                init=None,               random_state=None,
                                                max_features=None,       alpha=0.9,    verbose=0, learn_rate=None)
```

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.

**Parameters** `loss` : {‘ls’, ‘lad’, ‘huber’, ‘quantile’}, optional (default=’ls’)

loss function to be optimized. ‘ls’ refers to least squares regression. ‘lad’ (least absolute deviation) is a highly robust loss function solely based on order information of the input variables. ‘huber’ is a combination of the two. ‘quantile’ allows quantile regression (use `alpha` to specify the quantile).

**learning\_rate** : float, optional (default=0.1)

learning rate shrinks the contribution of each tree by `learning_rate`. There is a trade-off between `learning_rate` and `n_estimators`.

**n\_estimators** : int (default=100)

The number of boosting stages to perform. Gradient boosting is fairly robust to overfitting 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.

**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.

**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, None, optional (default=None)

The number of features to consider when looking for the best split. Features are chosen randomly at each split point. If None, then `max_features=n_features`. Choosing `max_features < n_features` leads to a reduction of variance and an increase in bias.

**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 ‘.’ for every tree built. If greater than 1 then it prints the score for every tree.

#### 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.

#### Examples

```
>>> samples = [[0, 0, 2], [1, 0, 0]]
>>> labels = [0, 1]
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> gb = GradientBoostingRegressor().fit(samples, labels)
>>> print(gb.predict([[0, 0, 0]]))
...
[ 1.32806...
```

#### Attributes

<code>feature_importances_</code>	array, shape = [n_features]	The feature importances (the higher, the more important the feature).
<code>oob_score_</code>	array, shape = [n_estimators]	Score of the training dataset obtained using an out-of-bag estimate. The i-th score <code>oob_score_[i]</code> is the deviance (= loss) of the model at iteration i on the out-of-bag sample.
<code>train_score_</code>	array, shape = [n_estimators]	The i-th score <code>train_score_[i]</code> is the deviance (= loss) of the model at iteration i on the in-bag sample. If <code>subsample == 1</code> this is the deviance on the training data.
<code>loss_</code>	LossFunc- tion	The concrete <code>LossFunction</code> object.
<code>init</code>	BaseEsti- mator	The estimator that provides the initial predictions. Set via the <code>init</code> argument or <code>loss.init_estimator</code> .
<code>estimators_</code> : list of Decision- TreeRegressor		The collection of fitted sub-estimators.

## Methods

<code>decision_function(X)</code>	Compute the decision function of X.
<code>fit(X, y)</code>	Fit the gradient boosting model.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict regression target for X.
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>staged_decision_function(X)</code>	Compute decision function of X for each iteration.
<code>staged_predict(X)</code>	Predict regression target at each stage for X.

`__init__(loss='ls', learning_rate=0.1, n_estimators=100, subsample=1.0, min_samples_split=2, min_samples_leaf=1, max_depth=3, init=None, random_state=None, max_features=None, alpha=0.9, verbose=0, learn_rate=None)`

**decision\_function(X)**

Compute the decision function of X.

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns score** : array, shape = [n\_samples, k]

The decision function of the input samples. Classes are ordered by arithmetical order. Regression and binary classification are special cases with k == 1, otherwise k==n\_classes.

**fit(X, y)**

Fit the gradient boosting model.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features. Use fortran-style to avoid memory copies.

`y` : array-like, shape = [n\_samples]

Target values (integers in classification, real numbers in regression) For classification, labels must correspond to classes 0, 1, ..., n\_classes\_-1

**Returns self** : object

Returns self.

**get\_params(deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**predict(X)**

Predict regression target for X.

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns y: array of shape = [n\_samples] :**

The predicted values.

**score**(X, y)

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  $((y - y_{\text{pred}})^2 \cdot \text{sum}()$  and  $v$  is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2 \cdot \text{sum}()$ . Best possible score is 1.0, lower values are worse.

**Parameters** X : array-like, shape = [n\_samples, n\_features]

Training set.

y : array-like, shape = [n\_samples]

**Returns** z : float

**set\_params**(\*\*params)

Set the parameters of the 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 :

**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. Classes are ordered by arithmetical order. Regression and binary classification are special cases with k == 1, otherwise k==n\_classes.

**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.

**Parameters** X : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns** y : array of shape = [n\_samples]

The predicted value of the input samples.

### 1.5.3 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.

For estimators to be usable within a pipeline, all except the last one need to have a `transform` function. Otherwise, the dataset can not be passed through this estimator.

## 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, shrinking=True, tol=0.001, verbose=False))])
```

The estimators of the 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, 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*

### 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.

## 1.5.4 FeatureUnion: Combining feature extractors

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=0, kernel='linear', max_iter=None, n_components=None, tol=0))],
    transformer_weights=None)
```

#### Examples:

- Concatenating multiple feature extraction methods

## 1.5.5 Model evaluation

The `sklearn.metrics` implements score functions, performance metrics, pairwise metrics and distance computations. Those functions are useful to assess the performance of an estimator under a specific criterion. Note that in many cases, the `score` method of the underlying estimator is sufficient and appropriate.

In this module, functions named as

- `*_score` return a scalar value to maximize: the higher the better;
- `*_error` or `*_loss` return a scalar value to minimize: the lower the better.

### Classification metrics

The `sklearn.metrics` implements several losses, scores and utility functions to measure classification performance.

Some of these are restricted to the binary classification case:

<code>auc_score(y_true, y_score)</code>	Compute Area Under the Curve (AUC) from prediction scores
<code>average_precision_score(y_true, y_score)</code>	Compute average precision (AP) from prediction scores
<code>hinge_loss(y_true, pred_decision[, ...])</code>	Average hinge loss (non-regularized)
<code>matthews_corrcoef(y_true, y_pred)</code>	Compute the Matthews correlation coefficient (MCC) for binary classes
<code>precision_recall_curve(y_true, probas_pred)</code>	Compute precision-recall pairs for different probability thresholds
<code>roc_curve(y_true, y_score[, pos_label])</code>	Compute Receiver operating characteristic (ROC)

Others also work in the multiclass case:

<code>accuracy_score(y_true, y_pred)</code>	Accuracy classification score
<code>classification_report(y_true, y_pred[, ...])</code>	Build a text report showing the main classification metrics
<code>confusion_matrix(y_true, y_pred[, labels])</code>	Compute confusion matrix to evaluate the accuracy of a classification
<code>f1_score(y_true, y_pred[, labels, ...])</code>	Compute the F1 score, also known as balanced F-score or F-measure
<code>fbeta_score(y_true, y_pred, beta[, labels, ...])</code>	Compute the F-beta score
<code>precision_recall_fscore_support(y_true, y_pred)</code>	Compute precision, recall, F-measure and support for each class
<code>precision_score(y_true, y_pred[, labels, ...])</code>	Compute the precision
<code>recall_score(y_true, y_pred[, labels, ...])</code>	Compute the recall
<code>zero_one_loss(y_true, y_pred[, normalize])</code>	Zero-One classification loss

Some metrics might require probability estimates of the positive class, confidence values or binary decisions value.

In the following sub-sections, we will describe each of those functions.

## Accuracy score

The `accuracy_score` function computes the `accuracy`, the fraction of correct predictions.

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  $n_{\text{samples}}$  is defined as

$$\text{accuracy}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1(\hat{y} = y)$$

where  $1(x)$  is the indicator function.

```
>>> 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
```

### Example:

- See *Test with permutations the significance of a classification score* for an example of accuracy score usage using permutations of the dataset.

## Area under the curve (AUC)

The `auc_score` function computes the ‘area under the curve’ (AUC) which is the area under the receiver operating characteristic (ROC) curve.

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.

```
>>> import numpy as np
>>> from sklearn.metrics import auc_score
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> auc_score(y_true, y_scores)
0.75
```

For more information see the [Wikipedia article on AUC](#) and the *Receiver operating characteristic (ROC)* section.

### Average precision score

The `average_precision_score` function computes the average precision (AP) from prediction scores. This score corresponds to the area under the precision-recall curve.

```
>>> 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...
```

For more information see the [Wikipedia article on average precision](#) and the *Precision, recall and F-measures* section.

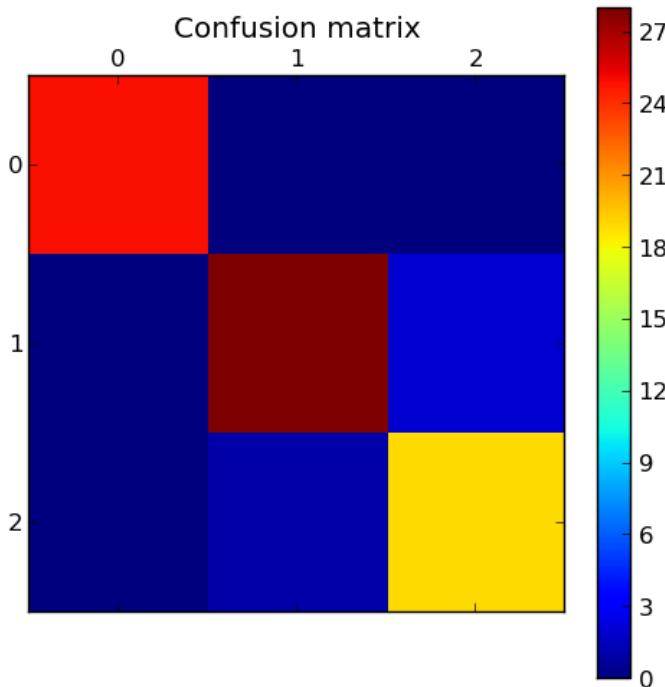
### Confusion matrix

The `confusion_matrix` function computes the `confusion matrix` to evaluate the accuracy on a classification problem.

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  $j$ . Here an example of such confusion matrix:

```
>>> 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 a visual representation of such confusion matrix (this figure comes from the *Confusion matrix* example):

**Example:**

- See *Confusion matrix* for an example of confusion matrix usage to evaluate the quality of the output of a classifier.
- See *Recognizing hand-written digits* for an example of confusion matrix usage in the classification of hand-written digits.
- See *Classification of text documents using sparse features* for an example of confusion matrix usage in the classification of text documents.

**Classification report**

The `classification_report` function builds a text report showing the main classification metrics. Here 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 in the classification of the hand-written digits.
- See *Classification of text documents using sparse features* for an example of classification report usage in the classification of text documents.
- See *Parameter estimation using grid search with a nested cross-validation* for an example of classification report usage in parameter estimation using grid search with a nested cross-validation.

**Precision, recall and F-measures**

The **precision** is intuitively the ability of the classifier not to label as positive a sample that is negative.

The **recall** is intuitively the ability of the classifier to find all the positive samples.

The **F-measure** ( $F_\beta$  and  $F_1$  measures) can be interpreted as a weighted harmonic mean of the precision and recall. A  $F_\beta$  measure reaches its best value at 1 and worst score at 0. With  $\beta = 1$ , the  $F_\beta$  measure leads to the  $F_1$  measure, where the recall and the precision are equally important.

Several functions allow you to analyze the precision, recall and F-measures score:

<code>f1_score(y_true, y_pred[, labels, ...])</code>	Compute the F1 score, also known as balanced F-score or F-measure
<code>fbeta_score(y_true, y_pred, beta[, labels, ...])</code>	Compute the F-beta score
<code>precision_recall_curve(y_true, probas_pred)</code>	Compute precision-recall pairs for different probability thresholds
<code>precision_recall_fscore_support(y_true, y_pred)</code>	Compute precision, recall, F-measure and support for each class
<code>precision_score(y_true, y_pred[, labels, ...])</code>	Compute the precision
<code>recall_score(y_true, y_pred[, labels, ...])</code>	Compute the recall

Note that the `precision_recall_curve` function is restricted to the binary case.

The average precision score might also interest you. See the *Average precision score* section.

**Examples:**

- See *Classification of text documents using sparse features* for an example of `f1_score` usage with classification of text documents.
- See *Parameter estimation using grid search with a nested cross-validation* for an example of `precision_score` and `recall_score` usage in parameter estimation using grid search with a nested cross-validation.
- See *Precision-Recall* for an example of precision-Recall metric to evaluate the quality of the output of a classifier with `precision_recall_curve`.
- See *Sparse recovery: feature selection for sparse linear models* for an example of `precision_recall_curve` usage in feature selection 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:

	Actual class (observation)	
Predicted class (expectation)	tp (true positive) Correct result	fp (false positive) Unexpected result
fn (false negative) Missing result	tn (true negative) Correct absence of result	

In this context, we can define the notions of precision, recall and F-measure:

$$\text{precision} = \frac{tp}{tp + fp},$$

$$\text{recall} = \frac{tp}{tp + fn},$$

$$F_\beta = (1 + \beta^2) \frac{\text{precision} \times \text{recall}}{\beta^2 \text{precision} + \text{recall}}.$$

Here 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
>>> 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 ])
```

**Multiclass and multilabels classification** In multiclass and multilabels classification task, the notions of precision, recall and F-measures can be applied to each label independently.

Moreover, these notions can be further extended. The functions `f1_score`, `fbeta_score`, `precision_recall_fscore_support`, `precision_score` and `recall_score` support an argument called `average` which defines the type of averaging:

- "macro": average over classes (does not take imbalance into account).
- "micro": average over instances (takes imbalance into account).

- "weighted": average weighted by support (takes imbalance into account). It can result in F1 score that is not between precision and recall.
- None: no averaging is performed.

**Warning:** Currently those functions support only the multiclass case. However the following definitions are general and remain valid in the multilabel case.

Let's define some notations:

- $n_{\text{labels}}$  and  $n_{\text{samples}}$  denotes respectively the number of labels and the number of samples.
- $\text{precision}_j$ ,  $\text{recall}_j$  and  $F_{\beta_j}$  are respectively the precision, the recall and  $F_{\beta}$  measure for the  $j$ -th label;
- $tp_j$ ,  $fp_j$  and  $fn_j$  respectively the number of true positives, false positives and false negatives for the  $j$ -th label;
- $y_i$  is the set of true label and  $\hat{y}_i$  is the set of predicted for the  $i$ -th sample;

The macro precision, recall and  $F_{\beta}$  are averaged over all labels

$$\text{macro\_precision} = \frac{1}{n_{\text{labels}}} \sum_{j=0}^{n_{\text{labels}}-1} \text{precision}_j,$$

$$\text{macro\_recall} = \frac{1}{n_{\text{labels}}} \sum_{j=0}^{n_{\text{labels}}-1} \text{recall}_j,$$

$$\text{macro\_F\_beta} = \frac{1}{n_{\text{labels}}} \sum_{j=0}^{n_{\text{labels}}-1} F_{\beta_j}.$$

The micro precision, recall and  $F_{\beta}$  are averaged over all instances

$$\text{micro\_precision} = \frac{\sum_{j=0}^{n_{\text{labels}}-1} tp_j}{\sum_{j=0}^{n_{\text{labels}}-1} tp_j + \sum_{j=0}^{n_{\text{labels}}-1} fp_j},$$

$$\text{micro\_recall} = \frac{\sum_{j=0}^{n_{\text{labels}}-1} tp_j}{\sum_{j=0}^{n_{\text{labels}}-1} tp_j + \sum_{j=0}^{n_{\text{labels}}-1} fn_j},$$

$$\text{micro\_F\_beta} = (1 + \beta^2) \frac{\text{micro\_precision} \times \text{micro\_recall}}{\beta^2 \text{micro\_precision} + \text{micro\_recall}}.$$

The weighted precision, recall and  $F_{\beta}$  are averaged weighted by their support

$$\text{weighted\_precision}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \frac{|y_i \cap \hat{y}_i|}{|y_i|},$$

$$\text{weighted\_recall}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \frac{|y_i \cap \hat{y}_i|}{|\hat{y}_i|},$$

$$\text{weighted\_F\_beta}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} (1 + \beta^2) \frac{|y_i \cap \hat{y}_i|}{\beta^2 |\hat{y}_i| + |y_i|}.$$

Here an example where average is set to average to macro:

```
>>> 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='macro')
0.33...
>>> metrics.fbeta_score(y_true, y_pred, average='macro', beta=0.5)
0.23...
>>> metrics.f1_score(y_true, y_pred, average='macro')
0.26...
>>> metrics.precision_recall_fscore_support(y_true, y_pred, average='macro')
(0.22..., 0.33..., 0.26..., None)
```

Here an example where average is set to to micro:

```
>>> 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='micro')
0.33...
>>> metrics.recall_score(y_true, y_pred, average='micro')
0.33...
>>> metrics.f1_score(y_true, y_pred, average='micro')
0.33...
>>> metrics.fbeta_score(y_true, y_pred, average='micro', beta=0.5)
0.33...
>>> metrics.precision_recall_fscore_support(y_true, y_pred, average='micro')
(0.33..., 0.33..., 0.33..., None)
```

Here an example where average is set to to weighted:

```
>>> 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='weighted')
0.22...
>>> metrics.recall_score(y_true, y_pred, average='weighted')
0.33...
>>> metrics.fbeta_score(y_true, y_pred, average='weighted', beta=0.5)
0.23...
>>> metrics.f1_score(y_true, y_pred, average='weighted')
0.26...
>>> metrics.precision_recall_fscore_support(y_true, y_pred, average='weighted')
(0.22..., 0.33..., 0.26..., None)
```

Here an example where average is set to None:

```
>>> 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=None)
array([ 0.66...,  0.        ,  0.        ])
>>> metrics.recall_score(y_true, y_pred, average=None)
array([ 1.,  0.,  0.])
>>> metrics.f1_score(y_true, y_pred, average=None)
array([ 0.8,  0.,  0.])
>>> metrics.fbeta_score(y_true, y_pred, average=None, beta=0.5)
array([ 0.71...,  0.        ,  0.        ])
```

```
>>> metrics.precision_recall_fscore_support(y_true, y_pred, beta=0.5)
(array([ 0.66...,  0.        ,  0.        ]), array([ 1.,  0.,  0.]), array([ 0.71...,  0.
```

## Hinge loss

The `hinge_loss` function computes the average hinge loss function. The hinge loss is used in maximal margin classification as support vector machines.

If the labels are encoded with +1 and -1,  $y$ : is the true value and  $w$  is the predicted decisions as output by `decision_function`, then the hinge loss is defined as:

$$L_{\text{Hinge}}(y, w) = \max \{1 - wy, 0\} = |1 - wy|_+$$

Here a small example demonstrating the use of the `hinge_loss` function with a `svm` classifier:

```
>>> 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='l2', 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...
```

## Matthews correlation coefficient

The `matthews_corrcoef` function computes the Matthew's correlation coefficient (MCC) for binary classes (quoting the [Wikipedia article on the Matthew's correlation coefficient](#)):

“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 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...
```

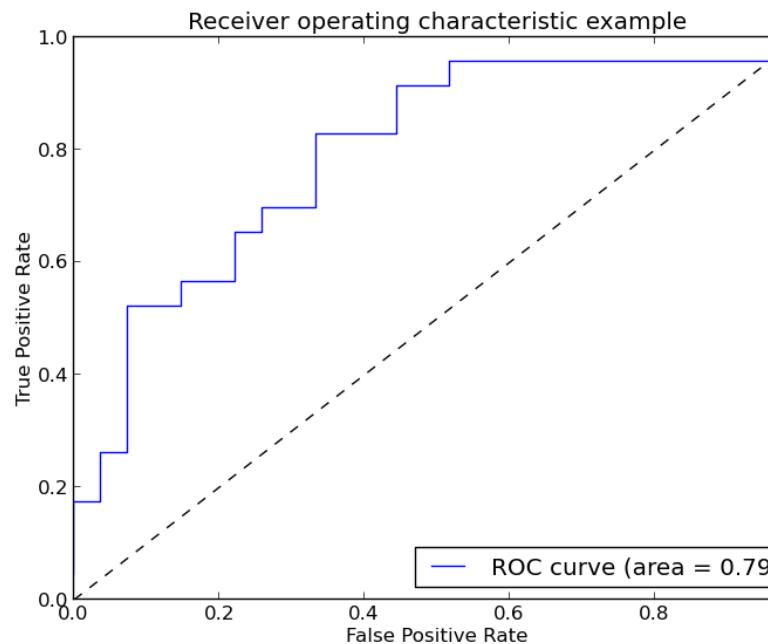
## Receiver operating characteristic (ROC)

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.”

Here a small example of how to use the `roc_curve` function:

```
>>> 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. ])
```



The following figure shows an example of such ROC curve.

### Examples:

- See *Receiver operating characteristic (ROC)* for an example of receiver operating characteristic (ROC) metric to evaluate the quality of the output of a classifier.
- See *Receiver operating characteristic (ROC) with cross validation* for an example of receiver operating characteristic (ROC) metric to evaluate the quality of the output of a classifier using cross-validation.
- See *Species distribution modeling* for an example of receiver operating characteristic (ROC) metric 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  $n_{\text{samples}}$ . By default, the function normalizes over the sample. To get the sum of the  $L_{0-1}$ , 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:

$$L_{0-1}(y_i, \hat{y}_i) = 1(\hat{y} \neq y)$$

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
```

### Example:

- See *Recursive feature elimination with cross-validation* for an example of the zero one loss usage to perform recursive feature elimination with cross-validation.

## Regression metrics

The `sklearn.metrics` implements several losses, scores and utility functions to measure regression performance. Some of those have been enhanced to handle the multioutput case: `mean_absolute_error`, `mean_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 and  $y$  is the corresponding (correct) target output, then the explained variance is estimated as follow:

$$\text{explained\_variance}(y, \hat{y}) = 1 - \frac{\text{Var}\{y - \hat{y}\}}{\text{Var}\{y\}}$$

The best possible score is 1.0, lower values are worse.

Here 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 the mean absolute error, which is a risk function 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  $n_{\text{samples}}$  is defined as

$$\text{MAE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} |y_i - \hat{y}_i|.$$

Here 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 the `mean square error`, which is a risk function corresponding to the expected value of the squared error loss or quadratic 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  $n_{\text{samples}}$  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 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.

### R<sup>2</sup> score, the coefficient of determination

The `r2_score` function computes R<sup>2</sup>, 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  $R^2$  estimated over  $n_{\text{samples}}$  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 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  $R^2$  score usage to evaluate Lasso and Elastic Net on sparse signals.

## Clustering metrics

The `sklearn.metrics` implements several losses, scores and utility function for more information see the *Clustering performance evaluation* section.

## Dummy estimators

When doing supervised learning, a simple sanity check consists in comparing one's estimator against simple rules of thumb. `DummyClassifier` implements three such simple strategies for classification:

- `stratified` generates randomly 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.

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(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%. For a better estimate of the accuracy, it is recommended to use a cross validation strategy, 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: setting estimator parameters* section.

More generally, when the accuracy of a classifier is too close to random classification, it probably means that something went wrong: features are not helpful, a hyper parameter is not correctly tuned, the classifier is suffering from class imbalance, etc...

`DummyRegressor` implements a simple rule of thumb for regression: always predict the mean of the training targets.

## 1.6 Dataset transformations

### 1.6.1 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.

#### 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 than others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

The function `scale` provides a quick and easy way to perform this operation on a single array-like dataset:

```
>>> from sklearn import preprocessing
>>> import numpy as np
>>> X = 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.

## 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.

## 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 most common class of (**Restricted**) Boltzmann Machines (not yet implemented in the scikit).

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.

## 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(dtype=<type 'float'>, n_values='auto')
>>> enc.transform([[0, 1, 3]]).toarray()
array([[ 1.,  0.,  0.,  1.,  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.

## Label preprocessing

### Label binarization

`LabelBinarizer` is a utility class to help create a label indicator matrix from a list of multi-class labels:

```
>>> lb = preprocessing.LabelBinarizer()
>>> lb.fit([1, 2, 6, 4, 2])
LabelBinarizer(neg_label=0, pos_label=1)
>>> lb.classes_
array([1, 2, 4, 6])
>>> lb.transform([1, 6])
array([[1, 0, 0, 0],
       [0, 0, 0, 1]])
```

`LabelBinarizer` also supports multiple labels per instance:

```
>>> lb.fit_transform([(1, 2), (3,)])
array([[1, 1, 0],
       [0, 0, 1]])
>>> lb.classes_
array([1, 2, 3])
```

### Label encoding

`LabelEncoder` is a utility class to help normalize labels such that they contain only values between 0 and `n_classes`-1. This is sometimes useful for writing efficient Cython routines. `LabelEncoder` can be used as follows:

```
>>> from sklearn import preprocessing
>>> le = preprocessing.LabelEncoder()
>>> le.fit([1, 2, 2, 6])
LabelEncoder()
>>> le.classes_
array([1, 2, 6])
```

```
>>> le.transform([1, 1, 2, 6])
array([0, 0, 1, 2])
>>> le.inverse_transform([0, 0, 1, 2])
array([1, 1, 2, 6])
```

It can also be used to transform non-numerical labels (as long as they are hashable and comparable) to numerical labels:

```
>>> le = preprocessing.LabelEncoder()
>>> le.fit(["paris", "paris", "tokyo", "amsterdam"])
LabelEncoder()
>>> list(le.classes_)
['amsterdam', 'paris', 'tokyo']
>>> le.transform(["tokyo", "tokyo", "paris"])
array([2, 2, 1])
>>> list(le.inverse_transform([2, 2, 1]))
['tokyo', 'tokyo', 'paris']
```

## 1.6.2 Feature extraction

The `sklearn.feature_extraction` module can be used to extract features in a format supported by machine learning algorithms from datasets consisting of formats such as text and image.

---

**Note:** Feature extraction is very different from *Feature selection*: the former consists in transforming arbitrary data, such as text or images, into numerical features usable for machine learning. The latter is a machine learning technique applied on these features.

---

### Loading features from dicts

The class `DictVectorizer` can be used to convert feature arrays represented as lists of standard Python `dict` objects to the NumPy/SciPy representation used by scikit-learn estimators.

While not particularly fast to process, Python's `dict` has the advantages of being convenient to use, being sparse (absent features need not be stored) and storing feature names in addition to values.

`DictVectorizer` implements what is called one-of-K or “one-hot” coding for categorical (aka nominal, discrete) features. Categorical features are “attribute-value” pairs where the value is restricted to a list of discrete of possibilities without ordering (e.g. topic identifiers, types of objects, tags, names...).

In the following, “city” is a categorical attribute while “temperature” is a traditional numerical feature:

```
>>> measurements = [
...     {'city': 'Dubai', 'temperature': 33.},
...     {'city': 'London', 'temperature': 12.},
...     {'city': 'San Francisco', 'temperature': 18.},
... ]

>>> from sklearn.feature_extraction import DictVectorizer
>>> vec = DictVectorizer()

>>> vec.fit_transform(measurements).toarray()
array([[ 1.,  0.,  0.,  33.],
       [ 0.,  1.,  0.,  12.],
       [ 0.,  0.,  1.,  18.]])
```

```
>>> vec.get_feature_names()
['city=Dubai', 'city=London', 'city=San 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 '<type 'numpy.float64'>'>
    with 6 stored elements in Compressed Sparse Row format>
>>> pos_vectorized.toarray()
array([[ 1.,  1.,  1.,  1.,  1.,  1.]])
>>> vec.get_feature_names()
['pos+1=PP', 'pos-1=NN', 'pos-2=DT', 'word+1=on', 'word-1=cat', 'word-2=the']
```

As you can imagine, if one extracts such a context around each individual word of a corpus of documents the resulting matrix will be very wide (many one-hot-features) with most of them being valued to zero most of the time. So as to make the resulting data structure able to fit in memory the `DictVectorizer` class uses a `scipy.sparse` matrix by default instead of a `numpy.ndarray`.

## Feature hashing

The class `FeatureHasher` is a high-speed, low-memory vectorizer that uses a technique known as [feature hashing](#), or the “hashing trick”. Instead of building a hash table of the features encountered in training, as the vectorizers do, instances of `FeatureHasher` apply a hash function to the features to determine their column index in sample matrices directly. The result is increased speed and reduced memory usage, at the expense of inspectability; the hasher does not remember what the input features looked like and has no `inverse_transform` method.

Since the hash function might cause collisions between (unrelated) features, a signed hash function is used and the sign of the hash value determines the sign of the value stored in the output matrix for a feature; this way, collisions are likely to cancel out rather than accumulate error, and the expected mean of any output feature’s value is zero

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 `MultinomialNB` or `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. If a feature occurs multiple times in a sample, the values will be summed. 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. The output from `FeatureHasher` is always a `scipy.sparse` matrix in the CSR format.

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  $\xi$  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.

### References:

- Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola and Josh Attenberg (2009). [Feature hashing for large scale multitask learning](#). Proc. ICML.
- [MurmurHash3](#).

## Text feature extraction

### The Bag of Words representation

Text Analysis is a major application field for machine learning algorithms. However the raw data, a sequence of symbols cannot be fed directly to the algorithms themselves as most of them expect numerical feature vectors with a

fixed size rather than the raw text documents with variable length.

In order to address this, scikit-learn provides utilities for the most common ways to extract numerical features from text content, namely:

- **tokenizing** strings and giving an integer id for each possible token, for instance by using whitespaces 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 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, charset='utf-8',
                charset_error='strict', dtype=<type 'long'>, input='content',
                lowercase=True, max_df=1.0, max_features=None, max_n=None, min_df=1,
                min_n=None, ngram_range=(1, 1), preprocessor=None, stop_words=None,
                strip_accents=None, token_pattern=u'(?u)\\b\\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 '<type 'numpy.int64'>'>
    with 19 stored elements in COOrdinate 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.")
[u'this', u'is', u'text', u'document', u'to', u'analyze']
```

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()
[u'and', u'document', u'first', u'is', u'one', u'second', u'the', u'third', u'this']

>>> 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 (the word themselves):

```
>>> 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!')
[u'bi', u'grams', u'are', u'cool', u'bi grams', u'grams are', u'are cool']
```

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, 0, 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, 0, 1, 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(u'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 '<type 'numpy.float64'>'>
      with 9 stored elements in Compressed Sparse Row 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 a very often used for text features, there is also another class called `TfidfVectorizer` that combines all the option 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 '<type 'numpy.float64'>'>
    with 19 stored elements in Compressed Sparse Row 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 text are likely to have noisy tf-idf values while the binary occurrence info is more stable.

As usual the only way how to best 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*

## 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 assignement 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, 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 resilant 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()
[u' w', u'ds', u'or', u'pr', u'rd', u's ', u'wo', u'wp']
>>> counts.toarray().astype(int)
array([[1, 1, 1, 0, 1, 1, 1, 0],
       [1, 1, 0, 1, 1, 1, 0, 1]])
```

In 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 '<type 'numpy.int64'>'>
  with 4 stored elements in COOrdinate format
>>> ngram_vectorizer.get_feature_names()
[u' fox ', u' jump', u'jumpy', u'umpy ']

>>> ngram_vectorizer = CountVectorizer(analyzer='char', ngram_range=(5, 5), min_df=1)
>>> ngram_vectorizer.fit_transform(['jumpy fox'])
...
<1x5 sparse matrix of type '<type 'numpy.int64'>'>
  with 5 stored elements in COOrdinate format
>>> ngram_vectorizer.get_feature_names()
[u'jumpy', u'mpy f', u'py fo', u'umpy ', u'y fox']
```

The word boundaries-aware variant `char_wb` is especially interesting for languages that use whitespaces 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 w.r.t. 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 implemented 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 '<type 'numpy.float64'>'>
    with 16 stored elements in Compressed Sparse Row 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 algorithm 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 '<type 'numpy.float64'>'>
    with 19 stored elements in Compressed Sparse Row 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.

## 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!")
[u'some...', u'punctuation!']
```

In particular we name:

- `preprocessor` a callable that takes a string as input and return another string (removing HTML tags or converting to lower case for instance)
- `tokenizer` a callable that takes a string as input and output a sequence of feature occurrences (a.k.a. the tokens).
- `analyzer` a callable that wraps calls to the preprocessor and tokenizer and further perform some filtering or n-grams extractions on the tokens.

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 method instead.

Customizing the vectorizer can be very useful to handle Asian languages that do not use an explicit word separator such as whitespace.

## Image feature extraction

### Patch extraction

The `extract_patches_2d` function extracts patches from an image stored as a two-dimensional array, or three-dimensional with color information along the third axis. For rebuilding an image from all its patches, use `reconstruct_from_patches_2d`. For example let use generate a 4x4 pixel picture with 3 color channels (e.g. in RGB format):

```
>>> import numpy as np
>>> from sklearn.feature_extraction import image

>>> one_image = np.arange(4 * 4 * 3).reshape((4, 4, 3))
>>> one_image[:, :, 0] # R channel of a fake RGB picture
array([[ 0,  3,  6,  9],
       [12, 15, 18, 21],
       [24, 27, 30, 33],
       [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
- 

### 1.6.3 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 nonlinear 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.

## Nystroem Method for Kernel Approximation

The Nystroem method, as implemented in `Nystroem` is a general method for low-rank approximations of kernels. It achieves this by essentially subsampling the data on which the kernel is evaluated. By default `Nystroem` uses the `rbf` kernel, but it can use any kernel function or a precomputed kernel matrix. The number of samples used - which is also the dimensionality of the features computed - is given by the parameter `n_components`.

## Radial Basis Function Kernel

The `RBFSampler` constructs an approximate mapping for the radial basis function kernel. 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, 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, rho=None, shuffle=False, 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 than `Nystroem`. `RBFSampler` is cheaper to compute, though, making use of larger feature spaces more efficient.

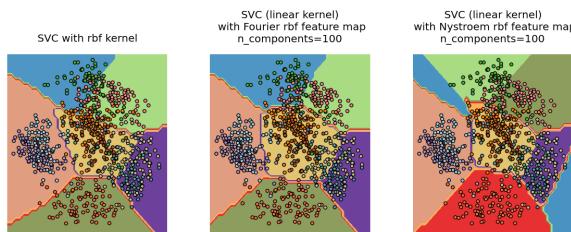


Figure 1.6: Comparing an exact RBF kernel (left) with the approximation (right)

**Examples:**

- *Explicit feature map approximation for RBF kernels*

## Additive Chi Squared Kernel

The additive chi squared kernel is a kernel on histograms, often used in computer vision.

The additive chi squared kernel as used here is given by

$$k(x, y) = \sum_i \frac{2x_i y_i}{x_i + y_i}$$

This is not exactly the same as `sklearn.metrics.additive_chi2_kernel`. The authors of [VZ2010] prefer the version above as it is always positive definite. Since the kernel is additive, it is possible to treat all components  $x_i$  separately for embedding. This makes it possible to sample the Fourier transform in regular intervals, instead of approximating using Monte Carlo sampling.

The class `AdditiveChi2Sampler` implements this component wise deterministic sampling. Each component is sampled  $n$  times, yielding  $2n+1$  dimensions per input dimension (the multiple of two stems from the real and complex part of the Fourier transform). In the literature,  $n$  is usually chosen to be 1 or 2, transforming the dataset to size  $n\_samples \times 5 * n\_features$  (in the case of  $n=2$ ).

The approximate feature map provided by `AdditiveChi2Sampler` can be combined with the approximate feature map provided by `RBFSampler` to yield an approximate feature map for the exponentiated chi squared kernel. See the [VZ2010] for details and [VVZ2010] for combination with the `RBFSampler`.

## Skewed Chi Squared Kernel

The skewed chi squared kernel is given by:

$$k(x, y) = \prod_i \frac{2\sqrt{x_i + c}\sqrt{y_i + c}}{x_i + y_i + 2c}$$

It has properties that are similar to the exponentiated chi squared kernel often used in computer vision, but allows for a simple Monte Carlo approximation of the feature map.

The usage of the `SkewedChi2Sampler` is the same as the usage described above for the `RBFSampler`. The only difference is in the free parameter, that is called  $c$ . For a motivation for this mapping and the mathematical details see [LS2010].

## Mathematical Details

Kernel methods like support vector machines or kernelized PCA rely on a property of reproducing kernel Hilbert spaces. For any positive definite kernel function  $k$  (a so called Mercer kernel), it is guaranteed that there exists a mapping  $\phi$  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  $\phi$  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  $\phi$ , thereby working explicitly with the representations  $\phi(x_i)$ , which obviates the need to apply the kernel or store training examples.

#### References:

### 1.6.4 Random Projection

The `sklearn.random_projection` module implements a simple and computationally efficient way to reduce the dimensionality of the data by trading a controlled amount of accuracy (as additional variance) for faster processing times and smaller model sizes. This module implements two types of unstructured random matrix: *Gaussian random matrix* and *sparse random matrix*.

The dimensions and distribution of random projections matrices are controlled so as to preserve the pairwise distances between any two samples of the dataset. Thus random projection is a suitable approximation technique for distance based method.

#### References:

- Sanjoy Dasgupta. 2000. [Experiments with random projection](#). In Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence (UAI'00), Craig Boutilier and Moisés Goldszmidt (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 143-151.
- Ella Bingham and Heikki Mannila. 2001. [Random projection in dimensionality reduction: applications to image and text data](#). In Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '01). ACM, New York, NY, USA, 245-250.

### The Johnson-Lindenstrauss lemma

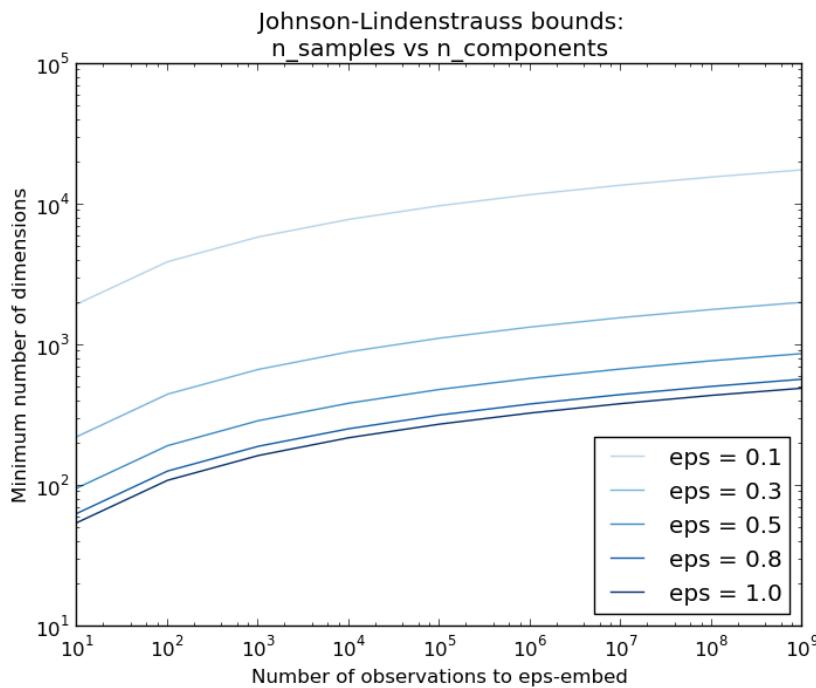
The main theoretical result behind the efficiency of random projection is the [Johnson-Lindenstrauss lemma](#) (quoting Wikipedia):

In mathematics, the Johnson-Lindenstrauss lemma is a result concerning low-distortion embeddings of points from high-dimensional into low-dimensional Euclidean space. The lemma states that a small set of points in a high-dimensional space can be embedded into a space of much lower dimension in such a way that distances between the points are nearly preserved. The map used for the embedding is at least Lipschitz, and can even be taken to be an orthogonal projection.

Knowing only the number of samples, the `sklearn.random_projection.johnson_lindenstrauss_min_dim` estimates conservatively the minimal size of the random subspace to guarantee a bounded distortion introduced by the random projection:

```
>>> from sklearn.random_projection import johnson_lindenstrauss_min_dim
>>> johnson_lindenstrauss_min_dim(n_samples=1e6, eps=0.5)
663
>>> johnson_lindenstrauss_min_dim(n_samples=1e6, eps=[0.5, 0.1, 0.01])
array([ 663, 11841, 1112658])
```

```
>>> johnson_lindenstrauss_min_dim(n_samples=[1e4, 1e5, 1e6], eps=0.1)
array([ 7894,  9868, 11841])
```

**Example:**

- See *The Johnson-Lindenstrauss bound for embedding with random projections* for a theoretical explication on the Johnson-Lindenstrauss lemma and an empirical validation using sparse random matrices.

**References:**

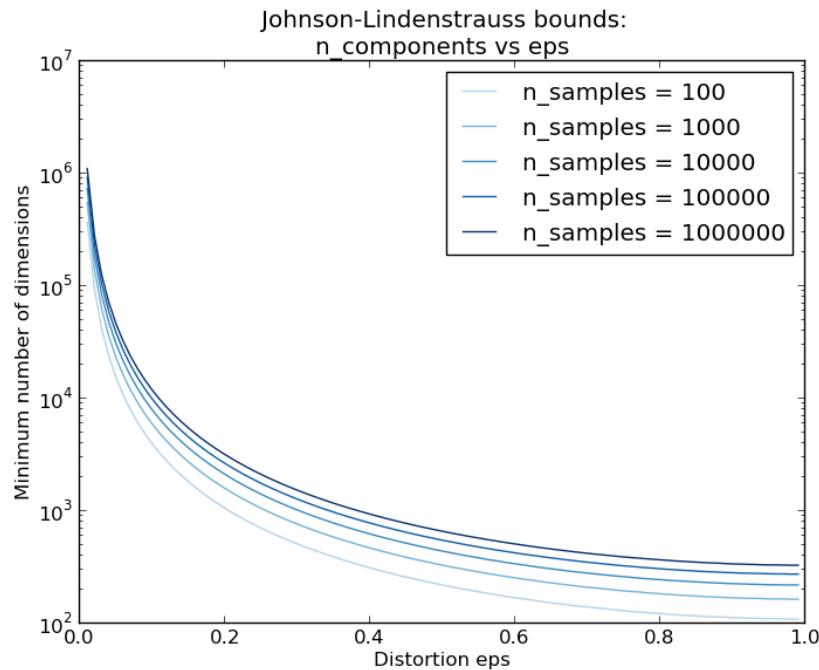
- Sanjoy Dasgupta and Anupam Gupta, 1999. [An elementary proof of the Johnson-Lindenstrauss Lemma](#).

**Gaussian random projection**

The `sklearn.random_projection.GaussianRandomProjection` reduces the dimensionality by projecting the original input space on a randomly generated matrix where components are drawn from the following distribution  $N(0, \frac{1}{n_{components}})$ .

Here a small excerpt which illustrates how to use the Gaussian random projection transformer:

```
>>> import numpy as np
>>> from sklearn import random_projection
>>> X = np.random.rand(100, 10000)
>>> transformer = random_projection.GaussianRandomProjection()
>>> X_new = transformer.fit_transform(X)
>>> X_new.shape
(100, 3947)
```



## Sparse random projection

The `sklearn.random_projection.SparseRandomProjection` reduces the dimensionality by projecting the original input space using a sparse random matrix.

Sparse random matrices are an alternative to dense Gaussian random projection matrix that guarantees similar embedding quality while being much more memory efficient and allowing faster computation of the projected data.

If we note  $s = 1 / \text{density}$ , the elements of the random matrix are drawn from

$$\begin{cases} -\sqrt{\frac{s}{n_{\text{components}}}} & 1/2s \\ 0 & \text{with probability } 1 - 1/s \\ +\sqrt{\frac{s}{n_{\text{components}}}} & 1/2s \end{cases}$$

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.

## 1.6.5 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 a function  $d(a, b)$  such that  $d(a, b) < d(a, c)$  if objects  $a$  and  $b$  are considered “more similar” to 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.  $d(a, b) \geq 0$ , 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.  $d(a, c) \leq d(a, b) + d(b, c)$ , 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” to 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  $1 / \text{num\_features}$
2.  $S = 1. / (D / np.max(D))$

### 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>

## 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

$$k(x, y) = \exp(-\gamma * \sum_i (x[i] - y[i]) * 2 / (x[i] + y[i]))$$

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>

## 1.7 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’.

## 1.7.1 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` x `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 bunch (which is a dictionary that is accessible with the ‘`dict.key`’ syntax). All datasets have at least two keys, `data`, containing an array of shape `n_samples` x `n_features` (except for `20newsgroups`) and `target`, a numpy array of length `n_features`, containing the targets.

The datasets also contain a description in `DESCR` and some contain `feature_names` and `target_names`. See the dataset descriptions below for details.

## 1.7.2 Toy datasets

scikit-learn comes with a few small standard datasets that do not require to download any file from some external website.

<code>load_boston()</code>	Load and return the boston house-prices dataset (regression).
<code>load_iris()</code>	Load and return the iris dataset (classification).
<code>load_diabetes()</code>	Load and return the diabetes dataset (regression).
<code>load_digits([n_class])</code>	Load and return the digits dataset (classification).
<code>load_linnerud()</code>	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.

## 1.7.3 Sample images

The scikit also embed a couple of sample JPEG images published under Creative Commons license by their authors. Those images can be useful to test algorithms and pipeline on 2D data.

<code>load_sample_images()</code>	Load sample images for image manipulation.
<code>load_sample_image(image_name)</code>	Load the numpy array of a single sample image



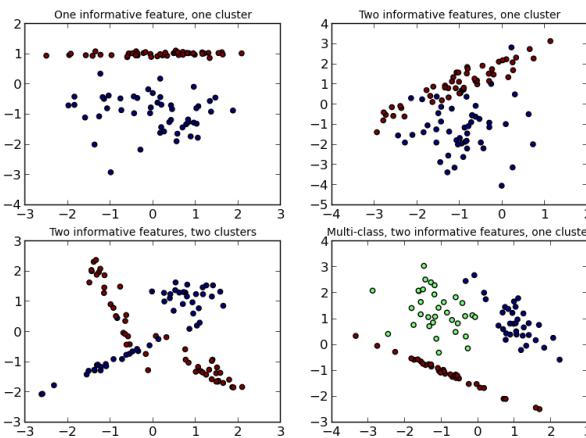
**Warning:** The default coding of images is based on the `uint8` dtype to spare memory. Often, algorithms work best if the input is converted to a floating point representation first. Also, `pylab.imshow` don't forget to scale to the range 0 - 1 as done in the following example.

**Examples:**

- *Color Quantization using K-Means*

## 1.7.4 Sample generators

In addition, scikit-learn includes various random sample generators that can be used to build artificial datasets of



controlled size and complexity.

<code>make_classification([n_samples, n_features, ...])</code>	Generate a random n-class classification problem.
<code>make_multilabel_classification([n_samples, ...])</code>	Generate a random multilabel classification problem.
<code>make_regression([n_samples, n_features, ...])</code>	Generate a random regression problem.
<code>make_blobs([n_samples, n_features, centers, ...])</code>	Generate isotropic Gaussian blobs for clustering.
<code>make_friedman1([n_samples, n_features, ...])</code>	Generate the “Friedman #1” regression problem
<code>make_friedman2([n_samples, noise, random_state])</code>	Generate the “Friedman #2” regression problem
<code>make_friedman3([n_samples, noise, random_state])</code>	Generate the “Friedman #3” regression problem
<code>make_hastie_10_2([n_samples, random_state])</code>	Generates data for binary classification used in
<code>make_low_rank_matrix([n_samples, ...])</code>	Generate a mostly low rank matrix with bell-shaped singular values
<code>make_sparse_coded_signal(n_samples, ...[, ...])</code>	Generate a signal as a sparse combination of dictionary elements.
<code>make_sparse_uncorrelated([n_samples, ...])</code>	Generate a random regression problem with sparse uncorrelated design matrix.
<code>make_spd_matrix(n_dim[, random_state])</code>	Generate a random symmetric, positive-definite matrix.
<code>make_swiss_roll([n_samples, noise, random_state])</code>	Generate a swiss roll dataset.
<code>make_s_curve([n_samples, noise, random_state])</code>	Generate an S curve dataset.
<code>make_sparse_spd_matrix([dim, alpha, ...])</code>	Generate a sparse symmetric definite positive matrix.

## 1.7.5 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>

## 1.7.6 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](#).

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.

## 1.7.7 The 20 newsgroups text dataset

The 20 newsgroups dataset comprises around 18000 newsgroups posts on 20 topics splitted 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 text files that can be fed to text feature extractors such as `sklearn.feature_extraction.text.Vectorizer` 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.

## 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_file` 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 `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])
```

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 sklearn.feature_extraction.text import Vectorizer
>>> documents = [open(f).read() for f in newsgroups_train.filenames]
>>> vectorizer = Vectorizer()
>>> vectors = vectorizer.fit_transform(documents)
>>> vectors.shape
(1073, 21108)
```

The extracted TF-IDF vectors are very sparse with an average of 118 non zero components by sample in a more than 20000 dimensional space (less than 1% non zero features):

```
>>> vectors.nnz / vectors.shape[0]
118
```

`sklearn.datasets.fetch_20newsgroups_vectorized` is a function which returns ready-to-use tfidf features instead of file names.

## Examples

- *Sample pipeline for text feature extraction and evaluation*
- *Classification of text documents using sparse features*

## 1.7.8 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 `fetch_mldata(dataname)`.

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. `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 `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, `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)
... 
```

## 1.7.9 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.

### Usage

scikit-learn provides two loaders that will automatically download, cache, parse the metadata files, decode the jpeg and convert the interesting slices into memmapped numpy arrays. This dataset size is more than 200 MB. The first load typically takes more than a couple of minutes to fully decode the relevant part of the JPEG files into numpy arrays. If the dataset has been loaded once, the following times the loading times less than 200ms by using a memmapped version memoized on the disk in the `~/scikit_learn_data/lfw_home/` folder using `joblib`.

The first loader is used for the Face Identification task: a multi-class classification task (hence supervised learning):

```
>>> from sklearn.datasets import fetch_lfw_people
>>> lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)

>>> for name in lfw_people.target_names:
...     print name
...
Ariel Sharon
Colin Powell
Donald Rumsfeld
George W Bush
Gerhard Schroeder
```

Hugo Chavez  
Tony Blair

The default slice is a rectangular shape around the face, removing most of the background:

```
>>> lfw_people.data.dtype
dtype('float32')

>>> lfw_people.data.shape
(1288, 1850)

>>> lfw_people.images.shape
(1288, 50, 37)
```

Each of the 1140 faces is assigned to a single person id in the `target` array:

```
>>> lfw_people.target.shape
(1288,)

>>> list(lfw_people.target[:10])
[5, 6, 3, 1, 0, 1, 3, 4, 3, 0]
```

The second loader is typically used for the face verification task: each sample is a pair of two picture belonging or not to the same person:

```
>>> from sklearn.datasets import fetch_lfw_pairs
>>> lfw_pairs_train = fetch_lfw_pairs(subset='train')

>>> list(lfw_pairs_train.target_names)
['Different persons', 'Same person']

>>> lfw_pairs_train.pairs.shape
(2200, 2, 62, 47)

>>> lfw_pairs_train.data.shape
(2200, 5828)

>>> lfw_pairs_train.target.shape
(2200,)
```

Both for the `fetch_lfw_people` and `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 `fetch_lfw_pairs` datasets is subdived in 3 subsets: the development `train` set, the development `test` set and an evaluation `10_folds` set meant to compute performance metrics using a 10-folds cross validation scheme.

## References:

- Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments.  
Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. University of Massachusetts, Amherst, Technical Report 07-49, October, 2007.

## Examples

*Faces recognition example using eigenfaces and SVMs*

## 1.8 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.

### List of modules

- `sklearn.cluster`: Clustering
  - Classes
  - Functions
- `sklearn.covariance`: Covariance Estimators
- `sklearn.cross_validation`: Cross Validation
- `sklearn.datasets`: Datasets
  - Loaders
  - Samples generator
- `sklearn.decomposition`: Matrix Decomposition
- `sklearn.dummy`: Dummy estimators
- `sklearn.ensemble`: Ensemble Methods
  - partial dependence
- `sklearn.feature_extraction`: Feature Extraction
  - From images
  - From text
- `sklearn.feature_selection`: Feature Selection
- `sklearn.gaussian_process`: Gaussian Processes
- `sklearn.grid_search`: Grid Search
- `sklearn.hmm`: Hidden Markov Models
- `sklearn.isotonic`: Isotonic regression
- `sklearn.kernel_approximation`: Kernel Approximation
- `sklearn.semi_supervised`: Semi-Supervised Learning
- `sklearn.lda`: Linear Discriminant Analysis
- `sklearn.linear_model`: Generalized Linear Models
- `sklearn.manifold`: Manifold Learning
- `sklearn.metrics`: Metrics
  - Classification metrics
  - Regression metrics
  - Clustering metrics
  - Pairwise metrics
- `sklearn.mixture`: Gaussian Mixture Models
- `sklearn.multiclass`: Multiclass and multilabel classification
  - Multiclass and multilabel classification strategies
- `sklearn.naive_bayes`: Naive Bayes
- `sklearn.neighbors`: Nearest Neighbors
- `sklearn.pls`: Partial Least Squares
- `sklearn.pipeline`: Pipeline
- `sklearn.preprocessing`: Preprocessing and Normalization
- `sklearn.qda`: Quadratic Discriminant Analysis
- `sklearn.random_projection`: Random projection
- `sklearn.svm`: Support Vector Machines
  - Estimators
  - Low-level methods
- `sklearn.tree`: Decision Trees
- `sklearn.utils`: Utilities

## 1.8.1 `sklearn.cluster`: Clustering

The `sklearn.cluster` module gathers popular unsupervised clustering algorithms.

**User guide:** See the *Clustering* section for further details.

### Classes

<code>cluster.AffinityPropagation([damping, ...])</code>	Perform Affinity Propagation Clustering of data
<code>cluster.DBSCAN([eps, min_samples, metric, ...])</code>	Perform DBSCAN clustering from vector array or distance matrix.
<code>cluster.KMeans([n_clusters, init, n_init, ...])</code>	K-Means clustering
<code>cluster.MiniBatchKMeans([n_clusters, init, ...])</code>	Mini-Batch K-Means clustering
<code>cluster.MeanShift([bandwidth, seeds, ...])</code>	MeanShift clustering
<code>cluster.SpectralClustering([n_clusters, ...])</code>	Apply clustering to a projection to the normalized laplacian.
<code>cluster.Ward([n_clusters, memory, ...])</code>	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,  convit=None,  copy=True,  preference=None,  p=None,    affinity='euclidean', verbose=False)
```

Perform Affinity Propagation Clustering of data

**Parameters** `damping: 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

`copy: boolean, optional, default: True :`

Make a copy of input data.

`preference: array [n_samples,] or float, optional, default: None :`

Preferences for each point - points with larger values of preferences are more likely to be chosen as exemplars. The number of exemplars, ie of clusters, is influenced by the input preferences value. If the preferences are not passed as arguments, they will be set to the median of the input similarities.

`affinity: 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.

## Notes

See examples/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

## Attributes

<i>cluster_centers_indices_</i>	array, [n_clusters]	Indices of cluster centers
<i>labels_</i>	array, [n_samples]	Labels of each point
<i>affinity_matrix_</i>	array-like, [n_samples, n_samples]	Stores the affinity matrix used in <code>fit</code> .

## Methods

<code>fit(X)</code>	Create affinity matrix from negative euclidean distances, then apply affinity propagation clustering.
<code>fit_predict(X[, y])</code>	Performs clustering on X and returns cluster labels.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>__init__(damping=0.5, max_iter=200, convergence_iter=15, convit=None, copy=True, preference=None, p=None, affinity='euclidean', verbose=False)</code>	
<code>fit(X)</code>	Create affinity matrix from negative euclidean distances, then apply affinity propagation clustering.
<b>Parameters X:</b> array [n_samples, n_features] or [n_samples, n_samples] :	
	Data matrix or, if affinity is precomputed, matrix of similarities / affinities.
<code>fit_predict(X, y=None)</code>	
	Performs clustering on X and returns cluster labels.
<b>Parameters X :</b> ndarray, shape (n_samples, n_features)	
	Input data.
<b>Returns y :</b> ndarray, shape (n_samples,)	
	cluster labels
<code>get_params(deep=True)</code>	
	Get parameters for the estimator
<b>Parameters deep:</b> boolean, optional :	
	If True, will return the parameters for this estimator and contained subobjects that are estimators.
<code>set_params(**params)</code>	
	Set the parameters of the 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.cluster.DBSCAN**

**class** `sklearn.cluster.DBSCAN(eps=0.5, min_samples=5, metric='euclidean', 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.

**Parameters** `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 in a neighborhood for a point to be considered as a core point.

**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.

**random\_state** : `numpy.RandomState`, optional

The generator used to initialize the centers. Defaults to `numpy.random`.

## Notes

See examples/plot\_dbscan.py for an example.

## 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

## Attributes

<code>core_sample_indices_</code>	<code>array, shape = [n_core_samples]</code>	Indices of core samples.
<code>components_</code>	<code>array, shape = [n_core_samples, n_features]</code>	Copy of each core sample found by training.
<code>labels_</code>	<code>array, shape = [n_samples]</code>	Cluster labels for each point in the dataset given to fit(). Noisy samples are given the label -1.

## Methods

<code>fit(X, **params)</code>	Perform DBSCAN clustering from vector array or distance matrix.
<code>fit_predict(X[, y])</code>	Performs clustering on X and returns cluster labels.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(eps=0.5, min_samples=5, metric='euclidean', random_state=None)`

`fit(X, **params)`

Perform DBSCAN clustering from vector array or distance matrix.

**Parameters X:** array [n\_samples, n\_samples] or [n\_samples, n\_features] :

Array of distances between samples, or a feature array. The array is treated as a feature array unless the metric is given as ‘precomputed’.

**params:** dict :

Overwrite keywords from `__init__`.

`fit_predict(X, y=None)`

Performs clustering on X and returns cluster labels.

**Parameters X :** ndarray, shape (n\_samples, n\_features)

Input data.

**Returns y :** ndarray, shape (n\_samples,)

cluster labels

`get_params(deep=True)`

Get parameters for the estimator

**Parameters deep:** boolean, optional :

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

`set_params(**params)`

Set the parameters of the 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.cluster.KMeans

```
class sklearn.cluster.KMeans(n_clusters=8, init='k-means++', n_init=10, max_iter=300,
                             tol=0.0001, precompute_distances=True, verbose=0, random_state=None,
                             copy_x=True, n_jobs=1, k=None)
```

K-Means clustering

**Parameters n\_clusters :** int, optional, default: 8

The number of clusters to form as well as the number of centroids to generate.

**max\_iter :** int

Maximum number of iterations of the k-means algorithm for a single 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 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.

If an ndarray is passed, it should be of shape (n\_clusters, n\_features) and gives the initial centers.

**precompute\_distances : boolean**

Precompute distances (faster but takes more memory).

**tol: float, optional default: 1e-4 :**

Relative tolerance w.r.t. inertia to declare convergence

**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.

**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.

**See Also:**

**MiniBatchKMeans** Alternative online implementation that does incremental updates of the centers positions using mini-batches. For large scale learning (say n\_samples > 10k) MiniBatchKMeans is probably much faster than the default batch implementation.

**Notes**

The k-means problem is solved using 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.

## Attributes

<code>cluster_centers_</code> : array, [n_clusters, n_features]	Coordinates of cluster centers
<code>labels_</code> :	Labels of each point
<code>inertia_</code> : float	The value of the inertia criterion associated with the chosen partition.

## Methods

<code>fit(X[, y])</code>	Compute k-means clustering.
<code>fit_predict(X)</code>	Compute cluster centers and predict cluster index for each sample.
<code>fit_transform(X[, y])</code>	Compute clustering and transform X to cluster-distance space.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict the closest cluster each sample in X belongs to.
<code>score(X)</code>	Opposite of the value of X on the K-means objective.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, y])</code>	Transform X to a cluster-distance space

<code>__init__(n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001, precompute_distances=True, verbose=0, random_state=None, copy_x=True, n_jobs=1, k=None)</code>	
<b>fit (X, y=None)</b>	Compute k-means clustering.
<b>Parameters</b>	<b>X</b> : array-like or sparse matrix, shape=(n_samples, n_features)
<b>fit_predict (X)</b>	Compute cluster centers and predict cluster index for each sample.
	Convenience method; equivalent to calling fit(X) followed by predict(X).
<b>fit_transform (X, y=None)</b>	Compute clustering and transform X to cluster-distance space.
	Equivalent to fit(X).transform(X), but more efficiently implemented.
<b>get_params (deep=True)</b>	Get parameters for the estimator
	<b>Parameters deep: boolean, optional :</b>
	If True, will return the parameters for this estimator and contained subobjects that are estimators.
<b>predict (X)</b>	Predict the closest cluster each sample in X belongs to.
	In the vector quantization literature, <code>cluster_centers_</code> is called the code book and each value returned by <code>predict</code> is the index of the closest code in the code book.
	<b>Parameters</b> <b>X:</b> {array-like, sparse matrix}, <b>shape = [n_samples, n_features] :</b>
	New data to predict.
	<b>Returns</b> <b>Y</b> : array, shape [n_samples, ]
	Index of the closest center each sample belongs to.

**score (X)**

Opposite of the value of X on the K-means objective.

**Parameters** **X**: {array-like, sparse matrix}, shape = [n\_samples, n\_features] :

New data.

**Returns score:** float :

Opposite of the value of X on the K-means objective.

**set\_params (\*\*params)**

Set the parameters of the 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, 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.

**Parameters** **X**: {array-like, sparse matrix}, shape = [n\_samples, n\_features] :

New data to transform.

**Returns X\_new** : array, shape [n\_samples, k]

X transformed in the new space.

**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,      k=None,      reassignment_ratio=0.01)
```

Mini-Batch K-Means clustering

**Parameters** **n\_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, optional

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, optional

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 k.

**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.

If an ndarray is passed, it should be of shape (n\_clusters, n\_features) and gives the initial centers.

**compute\_labels: boolean :**

Compute label assignments 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, optional :**

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.

## Notes

See <http://www.eecs.tufts.edu/~dsculley/papers/fastkmeans.pdf>

## Attributes

<i>cluster_centers_</i> : array, [n_clusters, n_features]	Coordinates of cluster centers
<i>labels_</i> :	Labels of each point (if compute_labels is set to True).
<i>inertia_</i> : 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.

## Methods

<code>fit(X[, y])</code>	Compute the centroids on X by chunking it into mini-batches.
<code>fit_predict(X)</code>	Compute cluster centers and predict cluster index for each sample.
<code>fit_transform(X[, y])</code>	Compute clustering and transform X to cluster-distance space.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>partial_fit(X[, y])</code>	Update k means estimate on a single mini-batch X.
<code>predict(X)</code>	Predict the closest cluster each sample in X belongs to.
<code>score(X)</code>	Opposite of the value of X on the K-means objective.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, y])</code>	Transform X to a cluster-distance space

`__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, k=None, reassignment_ratio=0.01)`

### `fit(X, y=None)`

Compute the centroids on X by chunking it into mini-batches.

**Parameters X: array-like, shape = [n\_samples, n\_features] :**

Coordinates of the data points to cluster

### `fit_predict(X)`

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 the estimator

**Parameters deep: boolean, optional :**

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

### `partial_fit(X, y=None)`

Update k means estimate on a single mini-batch X.

**Parameters X: array-like, shape = [n\_samples, n\_features] :**

Coordinates of the data points to cluster.

### `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.

**Parameters X: {array-like, sparse matrix}, shape = [n\_samples, n\_features] :**

New data to predict.

**Returns Y : array, shape [n\_samples, ]**

Index of the closest center each sample belongs to.

**score (X)**

Opposite of the value of X on the K-means objective.

**Parameters** **X**: {array-like, sparse matrix}, shape = [n\_samples, n\_features] :

New data.

**Returns score:** float :

Opposite of the value of X on the K-means objective.

**set\_params (\*\*params)**

Set the parameters of the 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, 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.

**Parameters** **X**: {array-like, sparse matrix}, shape = [n\_samples, n\_features] :

New data to transform.

**Returns X\_new** : array, shape [n\_samples, k]

X transformed in the new space.

**sklearn.cluster.MeanShift**

```
class sklearn.cluster.MeanShift(bandwidth=None,    seeds=None,    bin_seeding=False,    cluster_all=True)
```

MeanShift clustering

**Parameters** **bandwidth** : float, optional

Bandwith used in the RBF kernel If not set, the bandwidth is estimated. See clustering.estimate\_bandwidth.

**seeds** : array [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.

**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.

**Notes**

Scalability:

Because this implementation uses a flat kernel and a Ball Tree to look up members of each kernel, the complexity will be 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.

## Attributes

<code>cluster_centers_</code>	array, [n_clusters, n_features]	Coordinates of cluster centers.
<code>labels_</code> :		Labels of each point.

## Methods

<code>fit(X)</code>	Compute MeanShift
<code>fit_predict(X[, y])</code>	Performs clustering on X and returns cluster labels.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(bandwidth=None, seeds=None, bin_seeding=False, cluster_all=True)`

`fit(X)`

Compute MeanShift

**Parameters** `X` : array-like, shape=[n\_samples, n\_features]

Input points.

`fit_predict(X, y=None)`

Performs clustering on X and returns cluster labels.

**Parameters** `X` : ndarray, shape (n\_samples, n\_features)

Input data.

**Returns** `y` : ndarray, shape (n\_samples,)

cluster labels

`get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep: boolean, optional :`

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

**set\_params** (\*\*params)

Set the parameters of the 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.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, k=None, eigen_tol=0.0, assign_labels='kmeans', mode=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 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.

**Parameters** `n_clusters` : integer, optional

The dimension of the projection subspace.

**affinity:** string, ‘nearest\_neighbors’, ‘rbf’ or ‘precomputed’ :

**gamma:** float :

Scaling factor of Gaussian (rbf) affinity kernel. Ignored for `affinity='nearest_neighbors'`.

**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’ 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.

## 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>

## Attributes

<i>affinity_matrix_</i>	array-like, shape (n_samples, n_samples)	Affinity matrix used for clustering. Available only if after calling <code>fit</code> .
<i>labels_</i> :		Labels of each point

## Methods

<code>fit(X)</code>	Creates an affinity matrix for X using the selected affinity,
<code>fit_predict(X[, y])</code>	Performs clustering on X and returns cluster labels.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(n_clusters=8, eigen_solver=None, random_state=None, n_init=10, gamma=1.0, affinity='rbf', n_neighbors=10, k=None, eigen_tol=0.0, assign_labels='kmeans', mode=None)`

`fit(X)`

Creates an affinity matrix for X using the selected affinity, then applies spectral clustering to this affinity matrix.

**Parameters** **X** : 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.

**Parameters** **X** : ndarray, shape (n\_samples, n\_features)  
Input data.

**Returns** **y** : ndarray, shape (n\_samples,)  
cluster labels

**get\_params** (*deep=True*)  
Get parameters for the estimator

**Parameters** **deep: boolean, optional** :  
If True, will return the parameters for this estimator and contained subobjects that are estimators.

**k**  
DEPRECATED: 'k' was renamed to n\_clusters and will be removed in 0.15.

**mode**  
DEPRECATED: 'mode' was renamed to eigen\_solver and will be removed in 0.15.

**set\_params** (\*\*params)  
Set the parameters of the 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.cluster.Ward

**class** `sklearn.cluster.Ward(n_clusters=2, memory=Memory(cachedir=None), connectivity=None, copy=True, n_components=None, compute_full_tree='auto')`

Ward hierarchical clustering: constructs a tree and cuts it.

**Parameters** **n\_clusters** : int or ndarray

The number of clusters to find.

**connectivity** : sparse matrix.

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

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.

**copy** : bool

Copy the connectivity matrix or work inplace.

**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 cluster and using caching, it may be advantageous to compute the full tree.

## Attributes

<code>children_</code>	array-like, shape = [n_nodes, 2]	List of the children of each nodes. Leaves of the tree do not appear.
<code>labels_</code>	array [n_samples]	cluster labels for each point
<code>n_leaves_</code>	int	Number of leaves in the hierarchical tree.
<code>n_components_</code>	sparse matrix.	The estimated number of connected components in the graph.

## Methods

<code>fit(X)</code>	Fit the hierarchical clustering on the data
<code>fit_predict(X[, y])</code>	Performs clustering on X and returns cluster labels.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.

**`__init__(n_clusters=2, memory=Memory(cachedir=None), connectivity=None, copy=True, n_components=None, compute_full_tree='auto')`**

**fit (X)**

Fit the hierarchical clustering on the data

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

The samples a.k.a. observations.

**Returns self :**

**fit\_predict (X, y=None)**

Performs clustering on X and returns cluster labels.

**Parameters** `X` : ndarray, shape (n\_samples, n\_features)

Input data.

**Returns y** : ndarray, shape (n\_samples,)

cluster labels

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters** `deep`: boolean, optional :

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

**set\_params** (\*\**params*)

Set the parameters of the 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 :**

## Functions

<code>cluster.estimate_bandwidth(X[, quantile, ...])</code>	Estimate the bandwidth to use with MeanShift algorithm
<code>cluster.k_means(X, n_clusters[, init, ...])</code>	K-means clustering algorithm.
<code>cluster.ward_tree(X[, connectivity, ...])</code>	Ward clustering based on a Feature matrix.
<code>cluster.affinity_propagation(S[, ...])</code>	Perform Affinity Propagation Clustering of data
<code>cluster.dbscan(X[, eps, min_samples, ...])</code>	Perform DBSCAN clustering from vector array or distance matrix.
<code>cluster.mean_shift(X[, bandwidth, seeds, ...])</code>	Perform MeanShift Clustering of data using a flat kernel
<code>cluster.spectral_clustering(affinity[, ...])</code>	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 MeanShift algorithm

**Parameters** `X` : array [n\_samples, n\_features]

Input points.

`quantile` : float, default 0.3

should be between [0, 1] 0.5 means that the median is all pairwise distances is used.

`n_samples` : int

The number of samples to use. If None, all samples are used.

`random_state` : int or RandomState

Pseudo number generator state used for random sampling.

**Returns bandwidth** : float

The bandwidth parameter.

### sklearn.cluster.k\_means

`sklearn.cluster.k_means(X, n_clusters, init='k-means++', precompute_distances=True, n_init=10, max_iter=300, verbose=False, tol=0.0001, random_state=None, copy_x=True, n_jobs=1, k=None)`

K-means clustering algorithm.

**Parameters** `X` : 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

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.

**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 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.

**Returns** **centroid** : float ndarray with shape (k, n\_features)

Centroids found at the last iteration of k-means.

**label** : integer ndarray with shape (n\_samples,)

label[i] is the code or index of the centroid the i'th observation is closest to.

**inertia** : float

The final value of the inertia criterion (sum of squared distances to the closest centroid for all observations in the training set).

**sklearn.cluster.ward\_tree**

```
sklearn.cluster.ward_tree(X,      connectivity=None,      n_components=None,      copy=True,
                           n_clusters=None)
```

Ward clustering based on a Feature matrix.

The inertia matrix uses a Heapq-based representation.

This is the structured version, that takes into account a some topological structure between samples.

**Parameters X** : array of shape (n\_samples, n\_features)

feature matrix representing n\_samples samples to be clustered

**connectivity** : sparse matrix.

connectivity matrix. Defines for each sample the neigbhoring 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.

**copy** : bool (optional)

Make a copy of connectivity or work inplace. If connectivity is not of LIL type there will be a copy in any case.

**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.

**Returns children** : 2D array, shape (n\_nodes, 2)

list of the children of each nodes. Leaves of the tree have empty list of children.

**n\_components** : sparse matrix.

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.

**sklearn.cluster.affinity\_propagation**

```
sklearn.cluster.affinity_propagation(S, preference=None, p=None, convergence_iter=15,
                                      convit=None, max_iter=200, damping=0.5,
                                      copy=True, verbose=False)
```

Perform Affinity Propagation Clustering of data

**Parameters S:** array [n\_samples, n\_samples] :

Matrix of similarities between points

**preference: array [n\_samples,] or float, optional, default: None :**

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: 200 :**

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

**Returns cluster\_centers\_indices: array [n\_clusters] :**

index of clusters centers

**labels : array [n\_samples]**

cluster labels for each point

## Notes

See examples/plot\_affinity\_propagation.py for an example.

## References

Brendan J. Frey and Delbert Dueck, “Clustering by Passing Messages Between Data Points”, Science Feb. 2007

## sklearn.cluster.dbscan

`sklearn.cluster.dbscan(X, eps=0.5, min_samples=5, metric='euclidean', random_state=None)`

Perform DBSCAN clustering from vector array or distance matrix.

**Parameters X: array [n\_samples, n\_samples] or [n\_samples, n\_features] :**

Array of distances between samples, or a feature array. The array is treated as a feature array unless the metric is given as ‘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 in a neighborhood for a point to be considered as a core point.

**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.

**random\_state: numpy.RandomState, optional :**

The generator used to initialize the centers. Defaults to numpy.random.

**Returns** **core\_samples: array [n\_core\_samples] :**

Indices of core samples.

**labels : array [n\_samples]**

Cluster labels for each point. Noisy samples are given the label -1.

## Notes

See examples/plot\_dbSCAN.py for an example.

## 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, cluster_all=True, max_iterations=300)
```

Perform MeanShift Clustering of data using a flat kernel

Seed using a binning technique for scalability.

**Parameters** **X : array-like shape=[n\_samples, n\_features]**

Input data.

**bandwidth : float, optional**

Kernel bandwidth. If bandwidth is not defined, it is set using a heuristic given by the median of all pairwise distances.

**seeds : array [n\_seeds, n\_features]**

Point used as initial kernel locations.

**bin\_seeding : boolean**

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.

**Returns** `cluster_centers` : array [n\_clusters, n\_features]

Coordinates of cluster centers.

**labels** : array [n\_samples]

Cluster labels for each point.

## Notes

See examples/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,
                                    k=None,      eigen_tol=0.0,      assign_labels='kmeans',
                                    mode=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.

**Parameters** `affinity`: array-like or sparse matrix, shape: (n\_samples, n\_samples) :

The affinity matrix describing the relationship of the samples to embed. **Must be symmetric**.

**Possible examples:**

- adjacency matrix of a graph,
- heat kernel of the pairwise distance matrix of the samples,
- symmetric k-nearest neighbours connectivity matrix of the samples.

**n\_clusters**: integer, optional :

Number of clusters to extract.

**n\_components**: integer, optional, default is k :

Number of eigen vectors to use for the spectral embedding

**eigen\_solver**: {None, ‘arpack’ 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.

**Returns labels: array of integers, shape: n\_samples :**

The labels of the clusters.

**Notes**

The graph should contain only one connect component, elsewhere the results make little sense.

This algorithm solves the normalized cut for k=2: it is a normalized spectral clustering.

**References**

- Normalized cuts and image segmentation, 2000 Jianbo Shi, Jitendra Malik  
<http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.160.2324>
- A Tutorial on Spectral Clustering, 2007 Ulrike von Luxburg  
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.165.9323>
- Multiclass spectral clustering, 2003 Stella X. Yu, Jianbo Shi  
<http://www1.icsi.berkeley.edu/~stellayu/publication/doc/2003kwayICCV.pdf>

## 1.8.2 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.

<code>covariance.EmpiricalCovariance(...)</code>	Maximum likelihood covariance estimator
<code>covariance.EllipticEnvelope(...)</code>	An object for detecting outliers in a Gaussian distributed dataset.
<code>covariance.GraphLasso([alpha, mode, tol, ...])</code>	Sparse inverse covariance estimation with an l1-penalized estimator.
<code>covariance.GraphLassoCV([alphas, ...])</code>	Sparse inverse covariance w/ cross-validated choice of the l1 penalty
<code>covariance.LedoitWolf([store_precision, ...])</code>	LedoitWolf Estimator

Continued on next page

**Table 1.33 – continued from previous page**

<code>covariance.MinCovDet([store_precision, ...])</code>	Minimum Covariance Determinant (MCD): robust estimator of covariance.
<code>covariance.OAS([store_precision, ...])</code>	Oracle Approximating Shrinkage Estimator
<code>covariance.ShrunkCovariance(...)</code>	Covariance estimator with shrinkage

**sklearn.covariance.EmpiricalCovariance**

**class** `sklearn.covariance.EmpiricalCovariance(store_precision=True, sume_centered=False)` *as-*

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**

<code>covariance_</code>	2D ndarray, shape (n_features, n_features)	Estimated covariance matrix
<code>precision_</code>	2D ndarray, shape (n_features, n_features)	Estimated pseudo-inverse matrix. (stored only if store_precision is True)

**Methods**

<code>error_norm(comp_cov[, norm, scaling, squared])</code>	Computes the Mean Squared Error between two covariance estimators.
<code>fit(X[, y])</code>	Fits the Maximum Likelihood Estimator covariance model
<code>get_params([deep])</code>	Get parameters for the estimator
<code>get_precision()</code>	Getter for the precision matrix.
<code>mahalanobis(observations)</code>	Computes the mahalanobis distances of given observations.
<code>score(X_test[, y])</code>	Computes the log-likelihood of a gaussian data set with <code>self.covariance_</code> as an estimator.
<code>set_params(**params)</code>	Set the parameters of the 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)

**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):  $\sqrt{\text{tr}(A^T A)}$  - ‘spectral’:  $\sqrt{\max(\text{eigenvalues}(A^T A))}$  where A is the error (`comp_cov - self.covariance_`).

**scaling**: bool :

If True (default), the squared error norm is divided by n\_features. If False, the squared error norm is not rescaled.

**squared: bool :**

Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

**Returns The Mean Squared Error (in the sense of the Frobenius norm) between :**

**‘self’ and ‘comp\_cov’ covariance estimators. :**

**fit (X, y=None)**

Fits the Maximum Likelihood Estimator covariance model according to the given training data and parameters.

**Parameters X: array-like, shape = [n\_samples, n\_features] :**

Training data, where n\_samples is the number of samples and n\_features is the number of features.

**y: not used, present for API consistence purpose. :**

**Returns self : object**

Returns self.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**get\_precision ()**

Getter for the precision matrix.

**Returns precision\_: array-like, :**

The precision matrix associated to the current covariance object.

**mahalanobis (observations)**

Computes the mahalanobis distances of given observations.

The provided observations are assumed to be centered. One may want to center them using a location estimate first.

**Parameters observations: array-like, shape = [n\_observations, n\_features] :**

The observations, the Mahalanobis distances of which we compute. Observations are assumed to be drawn from the same distribution than tha data used in fit (including centering).

**Returns mahalanobis\_distance: array, shape = [n\_observations,] :**

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.

**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 the 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.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.

**Parameters store\_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.

**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

### Attributes

<i>contamination</i> : float, 0. < contamination < 0.5	The amount of contamination of the data set, i.e. the proportion of outliers in the data set.
<i>location_</i> : array-like, shape (n_features,)	Estimated robust location
<i>covariance_</i> : array-like, shape (n_features, n_features)	Estimated robust covariance matrix
<i>precision_</i> : array-like, shape (n_features, n_features)	Estimated pseudo inverse matrix. (stored only if store_precision is True)
<i>support_</i> : array-like, shape (n_samples,)	A mask of the observations that have been used to compute the robust estimates of location and shape.

### Methods

<code>correct_covariance(data)</code>	Apply a correction to raw Minimum Covariance Determinant estimates.
<code>decision_function(X[, raw_values])</code>	Compute the decision function of the given observations.
<code>error_norm(comp_cov[, norm, scaling, squared])</code>	Computes the Mean Squared Error between two covariance estimators.
<code>fit(X[, y])</code>	
<code>get_params([deep])</code>	Get parameters for the estimator
<code>get_precision()</code>	Getter for the precision matrix.
<code>mahalanobis(observations)</code>	Computes the mahalanobis distances of given observations.
<code>predict(X)</code>	Outlyingness of observations in X according to the fitted model.
<code>reweight_covariance(data)</code>	Reweighting raw Minimum Covariance Determinant estimates.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__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 [Rousseeuw1984].

**Parameters data: array-like, shape (n\_samples, n\_features) :**

The data matrix, with p features and n samples. The data set must be the one which was used to compute the raw estimates.

**Returns covariance\_corrected: array-like, shape (n\_features, n\_features) :**

Corrected robust covariance estimate.

`decision_function(X, raw_values=False)`

Compute the decision function of the given observations.

**Parameters X: 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.

**Returns decision: 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)

**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):  $\sqrt{\text{tr}(A^t \cdot A)}$  - ‘spectral’:  $\sqrt{\max(\text{eigenvalues}(A^t \cdot A))}$  where  $A$  is the error (`comp_cov - self.covariance_`).

**scaling: bool :**

If True (default), the squared error norm is divided by `n_features`. If False, the squared error norm is not rescaled.

**squared: bool :**

Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

**Returns The Mean Squared Error (in the sense of the Frobenius norm) between :**

**‘self’ and ‘comp\_cov’ covariance estimators. :**

**fit (X, y=None)**

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**get\_precision ()**

Getter for the precision matrix.

**Returns precision\_: array-like, :**

The precision matrix associated to the current covariance object.

**mahalanobis (observations)**

Computes the mahalanobis distances of given observations.

The provided observations are assumed to be centered. One may want to center them using a location estimate first.

**Parameters observations: array-like, shape = [n\_observations, n\_features] :**

The observations, the Mahalanobis distances of which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit (including centering).

**Returns mahalanobis\_distance: array, shape = [n\_observations, ] :**

Mahalanobis distances of the observations.

**predict (X)**

Outlyingness of observations in X according to the fitted model.

**Parameters X: array-like, shape = (n\_samples, n\_features) :**

**Returns is\_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)**

Reweighting raw Minimum Covariance Determinant estimates.

Reweighting observations using Rousseeuw's method (equivalent to deleting outlying observations from the data set before computing location and covariance estimates). [Rousseeuw1984]

**Parameters data: array-like, shape (n\_samples, n\_features) :**

The data matrix, with p features and n samples. The data set must be the one which was used to compute the raw estimates.

**Returns location\_reweighted: array-like, shape (n\_features,) :**

Reweighted robust location estimate.

**covariance\_reweighted: array-like, shape (n\_features, n\_features) :**

Reweighted robust covariance estimate.

**support\_reweighted: array-like, type boolean, shape (n\_samples,) :**

A mask of the observations that have been used to compute the reweighted robust location and covariance estimates.

**score (X, y)**

Returns the mean accuracy on the given test data and labels.

**Parameters X : array-like, shape = [n\_samples, n\_features]**

Training set.

**y : array-like, shape = [n\_samples]**

Labels for X.

**Returns z : float**

**set\_params (\*\*params)**

Set the parameters of the 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.covariance.GraphLasso

```
class sklearn.covariance.GraphLasso(alpha=0.01, mode='cd', tol=0.0001, max_iter=100, verbose=False)
```

Sparse inverse covariance estimation with an l1-penalized estimator.

**Parameters alpha:** **positive float, optional :**

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 plotted at each iteration

**See Also:**

`graph_lasso`, `GraphLassoCV`

### Attributes

<code>covariance_</code>	array-like, shape (n_features, n_features)	Estimated covariance matrix
<code>precision_</code>	array-like, shape (n_features, n_features)	Estimated pseudo inverse matrix.

### Methods

<code>error_norm(comp_cov[, norm, scaling, squared])</code>	Computes the Mean Squared Error between two covariance estimators.
<code>fit(X[, y])</code>	
<code>get_params([deep])</code>	Get parameters for the estimator
<code>get_precision()</code>	Getter for the precision matrix.
<code>mahalanobis(observations)</code>	Computes the mahalanobis distances of given observations.
<code>score(X_test[, y])</code>	Computes the log-likelihood of a gaussian data set with <code>self.covariance_</code> as an estimate.
<code>set_params(**params)</code>	Set the parameters of the estimator.

**`__init__(alpha=0.01, mode='cd', tol=0.0001, max_iter=100, verbose=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)

**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):  $\sqrt{\text{tr}(A^t A)}$  - ‘spectral’:  $\sqrt{\max(\text{eigenvalues}(A^t A))}$  where  $A$  is the error (`comp_cov` – `self.covariance_`).

**scaling: bool :**

If True (default), the squared error norm is divided by `n_features`. If False, the squared error norm is not rescaled.

**squared: bool :**

Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

**Returns The Mean Squared Error (in the sense of the Frobenius norm) between :**

**‘self’ and ‘comp\_cov’ covariance estimators. :**

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**get\_precision ()**

Getter for the precision matrix.

**Returns precision\_: array-like, :**

The precision matrix associated to the current covariance object.

**mahalanobis (observations)**

Computes the mahalanobis distances of given observations.

The provided observations are assumed to be centered. One may want to center them using a location estimate first.

**Parameters observations: array-like, shape = [n\_observations, n\_features] :**

The observations, the Mahalanobis distances of which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit (including centering).

**Returns mahalanobis\_distance: array, shape = [n\_observations,] :**

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.

**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 the 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.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)
```

Sparse inverse covariance w/ cross-validated choice of the l1 penalty

**Parameters alphas: integer, or list positive float, optional :**

If an integer is given, it fixes the number of points on the grids of alpha to be used. If a list is given, it gives the grid to be used. See the notes in the class docstring for more details.

**n\_refinements: strictly positive integer :**

The number of time the grid is refined. Not used if explicit values of alphas are passed.

**cv : crossvalidation generator, optional**

see `sklearn.cross_validation` module. If `None` is passed, default 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 :**

The maximum number of iterations

**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.

**n\_jobs: int, optional :**

number of jobs to run in parallel (default 1)

**verbose: boolean, optional :**

If `verbose` is `True`, the objective function and dual gap are print at each iteration

**See Also:**

`graph_lasso`, `GraphLasso`

**Notes**

The search for the optimal alpha is done on an iteratively refined grid: first the cross-validated scores on a grid are computed, then a new refined grid is center around the maximum...

One of the challenges that we have to face 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.

## Attributes

<code>covariance_</code>	array-like, shape (n_features, n_features)	Estimated covariance matrix
<code>precision_</code>	array-like, shape (n_features, n_features)	Estimated precision matrix (inverse covariance).
<code>alpha_</code> : float		Penalization parameter selected
<code>cv_alphas_</code> : list of float		All the penalization parameters explored
<code>cv_scores</code> : 2D array (n_alphas, n_folds)		The log-likelihood score on left-out data across the folds.

## Methods

<code>error_norm(comp_cov[, norm, scaling, squared])</code>	Computes the Mean Squared Error between two covariance estimators.
<code>fit(X[, y])</code>	
<code>get_params([deep])</code>	Get parameters for the estimator
<code>get_precision()</code>	Getter for the precision matrix.
<code>mahalanobis(observations)</code>	Computes the mahalanobis distances of given observations.
<code>score(X_test[, y])</code>	Computes the log-likelihood of a gaussian data set with <code>self.covariance_</code> as an estimator.
<code>set_params(**params)</code>	Set the parameters of the estimator.

**`__init__(alphas=4, n_refinements=4, cv=None, tol=0.0001, max_iter=100, mode='cd', n_jobs=1, verbose=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)

**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):  $\sqrt{\text{tr}(A^t \cdot A)}$  - ‘spectral’:  $\sqrt{\max(\text{eigenvalues}(A^t \cdot A))}$  where A is the error (`comp_cov - self.covariance_`).

**scaling: bool :**

If True (default), the squared error norm is divided by n\_features. If False, the squared error norm is not rescaled.

**squared: bool :**

Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

**Returns The Mean Squared Error (in the sense of the Frobenius norm) between :**

**‘self’ and ‘comp\_cov’ covariance estimators. :**

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**get\_precision ()**

Getter for the precision matrix.

**Returns precision\_: array-like, :**

The precision matrix associated to the current covariance object.

**mahalanobis (observations)**

Computes the mahalanobis distances of given observations.

The provided observations are assumed to be centered. One may want to center them using a location estimate first.

**Parameters observations: array-like, shape = [n\_observations, n\_features] :**

The observations, the Mahalanobis distances of which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit (including centering).

**Returns mahalanobis\_distance: array, shape = [n\_observations,] :**

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.

**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 the 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.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.

#### Parameters

**store\_precision** : bool

Specify 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.

**block\_size**: int :

Size of the blocks into which the covariance matrix will be split during its Ledoit-Wolf estimation. If `n_features > block_size`, an error will be raised since the shrunk covariance matrix will be considered as too large regarding the available memory.

#### 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.

#### Attributes

<code>covariance_</code>	array-like, shape (n_features, n_features)	Estimated covariance matrix
<code>precision_</code>	array-like, shape (n_features, n_features)	Estimated pseudo inverse matrix. (stored only if store_precision is True)
<code>shrinkage_</code> : float, 0 <= shrinkage <= 1		coefficient in the convex combination used for the computation of the shrunk estimate.

#### Methods

<code>error_norm(comp_cov[, norm, scaling, squared])</code>	Computes the Mean Squared Error between two covariance estimators.
<code>fit(X[, y])</code>	Fits the Ledoit-Wolf shrunk covariance model
<code>get_params([deep])</code>	Get parameters for the estimator
<code>get_precision()</code>	Getter for the precision matrix.
<code>mahalanobis(observations)</code>	Computes the mahalanobis distances of given observations.
<code>score(X_test[, y])</code>	Computes the log-likelihood of a gaussian data set with <code>self.covariance_</code> as an estimator.
<code>set_params(**params)</code>	Set the parameters of the estimator.

```
__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)

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’ (de-
    fault):  $\sqrt{\text{tr}(A^t A)}$  - ‘spectral’:  $\sqrt{\max(\text{eigenvalues}(A^t A))}$  where A is the error
    (comp_cov - self.covariance_).

scaling: bool :
    If True (default), the squared error norm is divided by n_features. If False, the squared
    error norm is not rescaled.

squared: bool :
    Whether to compute the squared error norm or the error norm. If True (default), the
    squared error norm is returned. If False, the error norm is returned.

Returns The Mean Squared Error (in the sense of the Frobenius norm) between :
    ‘self’ and ‘comp_cov’ covariance estimators. :

fit(X, y=None)
    Fits the Ledoit-Wolf shrunk covariance model according to the given training data and parameters.

Parameters X: array-like, shape = [n_samples, n_features] :
    Training data, where n_samples is the number of samples and n_features is the number
    of features.

y: not used, present for API consistence purpose. :

Returns self: object :
    Returns self.

get_params(deep=True)
    Get parameters for the estimator

Parameters deep: boolean, optional :
    If True, will return the parameters for this estimator and contained subobjects that are
    estimators.

get_precision()
    Getter for the precision matrix.

Returns precision_: array-like :
    The precision matrix associated to the current covariance object.

mahalanobis(observations)
    Computes the mahalanobis distances of given observations.

    The provided observations are assumed to be centered. One may want to center them using a location
    estimate first.

Parameters observations: 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 tha data used in fit (including centering).

**Returns mahalanobis\_distance: array, shape = [n\_observations] :**

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.

**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 tha 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 the 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.covariance.MinCovDet

```
class sklearn.covariance.MinCovDet(store_precision=True,      assume_centered=False,      sup-
                                         port_fraction=None, random_state=None)
```

Minimum Covariance Determinant (MCD): robust estimator of covariance.

The Minimum Covariance Determinant covariance estimator is to be applied on Gaussian-distributed data, but could still be relevant on data drawn from a unimodal, symetric distribution. It is not meant to be used with multimodal 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 multimodal datasets.

**Parameters store\_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

**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.

## References

[Rousseeuw1984], [Rousseeuw1999], [Butler1993]

## Attributes

<i>raw_location_</i> : array-like, shape (n_features,)	The raw robust estimated location before correction and reweighting
<i>raw_covariance_</i> : array-like, shape (n_features, n_features)	The raw robust estimated covariance before correction and reweighting
<i>raw_support_</i> : 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 reweighting.
<i>location_</i> : array-like, shape (n_features,)	Estimated robust location
<i>covariance_</i> : array-like, shape (n_features, n_features)	Estimated robust covariance matrix
<i>precision_</i> : array-like, shape (n_features, n_features)	Estimated pseudo inverse matrix. (stored only if store_precision is True)
<i>support_</i> : array-like, shape (n_samples,)	A mask of the observations that have been used to compute the robust estimates of location and shape.
<i>dist_</i> : array-like, shape (n_samples,)	Mahalanobis distances of the training set (on which <i>fit</i> is called) observations.

## Methods

<code>correct_covariance(data)</code>	Apply a correction to raw Minimum Covariance Determinant estimates.
<code>error_norm(comp_cov[, norm, scaling, squared])</code>	Computes the Mean Squared Error between two covariance estimators.
<code>fit(X[, y])</code>	Fits a Minimum Covariance Determinant with the FastMCD algorithm.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>get_precision()</code>	Getter for the precision matrix.
<code>mahalanobis(observations)</code>	Computes the mahalanobis distances of given observations.
<code>reweight_covariance(data)</code>	Reweight raw Minimum Covariance Determinant estimates.
<code>score(X_test[, y])</code>	Computes the log-likelihood of a gaussian data set with <i>self.covariance_</i> as an estimate.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(store_precision=True, assume_centered=False, support_fraction=None, 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 [Rousseeuw1984].

**Parameters data: array-like, shape (n\_samples, n\_features) :**

The data matrix, with p features and n samples. The data set must be the one which was

used to compute the raw estimates.

**Returns covariance\_corrected: array-like, shape (n\_features, n\_features) :**

Corrected robust covariance estimate.

**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):  $\sqrt{\text{tr}(A^t \cdot A)}$  - ‘spectral’:  $\sqrt{\max(\text{eigenvalues}(A^t \cdot A))}$  where A is the error ( $\text{comp\_cov} - \text{self.covariance}_\text{}$ ).

**scaling: bool :**

If True (default), the squared error norm is divided by n\_features. If False, the squared error norm is not rescaled.

**squared: bool :**

Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

**Returns The Mean Squared Error (in the sense of the Frobenius norm) between :**

**‘self’ and ‘comp\_cov’ covariance estimators. :**

**fit (X, y=None)**

Fits a Minimum Covariance Determinant with the FastMCD algorithm.

**Parameters X: array-like, shape = [n\_samples, n\_features] :**

Training data, where n\_samples is the number of samples and n\_features is the number of features.

**y: not used, present for API consistence purpose. :**

**Returns self: object :**

Returns self.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**get\_precision ()**

Getter for the precision matrix.

**Returns precision\_: array-like, :**

The precision matrix associated to the current covariance object.

**mahalanobis (observations)**

Computes the mahalanobis distances of given observations.

The provided observations are assumed to be centered. One may want to center them using a location estimate first.

**Parameters** **observations:** array-like, shape = [n\_observations, n\_features] :

The observations, the Mahalanobis distances of which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit (including centering).

**Returns** **mahalanobis\_distance:** array, shape = [n\_observations, ] :

Mahalanobis distances of the observations.

**reweight\_covariance (data)**

Reweighting raw Minimum Covariance Determinant estimates.

Reweighting observations using Rousseeuw's method (equivalent to deleting outlying observations from the data set before computing location and covariance estimates). [Rousseeuw1984]

**Parameters** **data:** array-like, shape (n\_samples, n\_features) :

The data matrix, with p features and n samples. The data set must be the one which was used to compute the raw estimates.

**Returns** **location\_reweighted:** array-like, shape (n\_features, ) :

Reweighted robust location estimate.

**covariance\_reweighted:** array-like, shape (n\_features, n\_features) :

Reweighted robust covariance estimate.

**support\_reweighted:** array-like, type boolean, shape (n\_samples,) :

A mask of the observations that have been used to compute the reweighted 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 the 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.covariance.OAS

```
class sklearn.covariance.OAS (store_precision=True, assume_centered=False)
    Oracle Approximating Shrinkage Estimator
```

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>).

**Parameters** `store_precision` : bool

Specify 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.

### 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.

### Attributes

<code>covariance_</code>	array-like, shape (n_features, n_features)	Estimated covariance matrix
<code>precision_</code>	array-like, shape (n_features, n_features)	Estimated pseudo inverse matrix. (stored only if store_precision is True)
<code>shrinkage_</code> : float, 0 <= shrinkage <= 1		coefficient in the convex combination used for the computation of the shrunk estimate.

### Methods

<code>error_norm(comp_cov[, norm, scaling, squared])</code>	Computes the Mean Squared Error between two covariance estimators.
<code>fit(X[, y])</code>	Fits the Oracle Approximating Shrinkage covariance model
<code>get_params([deep])</code>	Get parameters for the estimator
<code>get_precision()</code>	Getter for the precision matrix.
<code>mahalanobis(observations)</code>	Computes the mahalanobis distances of given observations.
<code>score(X_test[, y])</code>	Computes the log-likelihood of a gaussian data set with <code>self.covariance_</code> as an
	Continued on next page

**Table 1.40 – continued from previous page**

<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>__init__(store_precision=True, assume_centered=False)</code>	
<code>error_norm(comp_cov, norm='frobenius', scaling=True, squared=True)</code>	Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm)
<b>Parameters</b> <code>comp_cov: array-like, shape = [n_features, n_features]</code> :	
The covariance to compare with.	
<b>norm: str</b> :	
The type of norm used to compute the error. Available error types: - ‘frobenius’ (default): $\sqrt{\text{tr}(A^t \cdot A)}$ - ‘spectral’: $\sqrt{\max(\text{eigenvalues}(A^t \cdot A))}$ where $A$ is the error ( <code>comp_cov - self.covariance_</code> ).	
<b>scaling: bool</b> :	
If True (default), the squared error norm is divided by <code>n_features</code> . If False, the squared error norm is not rescaled.	
<b>squared: bool</b> :	
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.	
<b>Returns</b> The Mean Squared Error (in the sense of the Frobenius norm) between :	
‘self’ and ‘comp_cov’ covariance estimators. :	
<code>fit(X, y=None)</code>	Fits the Oracle Approximating Shrinkage covariance model according to the given training data and parameters.
<b>Parameters</b> <code>X: array-like, shape = [n_samples, n_features]</code> :	
Training data, where <code>n_samples</code> is the number of samples and <code>n_features</code> is the number of features.	
<b>y: not used, present for API consistence purpose.</b> :	
<b>Returns</b> <code>self: object</code> :	
Returns self.	
<code>get_params(deep=True)</code>	Get parameters for the estimator
<b>Parameters</b> <code>deep: boolean, optional</code> :	
If True, will return the parameters for this estimator and contained subobjects that are estimators.	
<code>get_precision()</code>	Getter for the precision matrix.
<b>Returns</b> <code>precision_: array-like</code> , :	
The precision matrix associated to the current covariance object.	

**mahalanobis** (*observations*)

Computes the mahalanobis distances of given observations.

The provided observations are assumed to be centered. One may want to center them using a location estimate first.

**Parameters** **observations**: array-like, shape = [n\_observations, n\_features] :

The observations, the Mahalanobis distances of which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit (including centering).

**Returns** **mahalanobis\_distance**: array, shape = [n\_observations,] :

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.

**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 the 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.covariance.ShrunkCovariance**

```
class sklearn.covariance.ShrunkCovariance(store_precision=True,      assume_centered=False,
                                         shrinkage=0.1)
```

Covariance estimator with shrinkage

**Parameters** **store\_precision** : bool

Specify if the estimated precision is stored

**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

$$(1 - \text{shrinkage}) * \text{cov}$$

• $\text{shrinkage} * \mu * \text{np.identity}(n\_features)$   
 where  $\mu = \text{trace}(\text{cov}) / n\_features$

## Attributes

<i>covariance_</i>	array-like, shape (n_features, n_features)	Estimated covariance matrix
<i>precision_</i>	array-like, shape (n_features, n_features)	Estimated pseudo inverse matrix. (stored only if store_precision is True)
<i>shrinkage</i> : float, 0 <= shrinkage <= 1		coefficient in the convex combination used for the computation of the shrunk estimate.

## Methods

<code>error_norm(comp_cov[, norm, scaling, squared])</code>	Computes the Mean Squared Error between two covariance estimators.
<code>fit(X[, y])</code>	Fits the shrunk covariance model according to the given training data and parameters.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>get_precision()</code>	Getter for the precision matrix.
<code>mahalanobis(observations)</code>	Computes the mahalanobis distances of given observations.
<code>score(X_test[, y])</code>	Computes the log-likelihood of a gaussian data set with <i>self.covariance_</i> as an estimator.
<code>set_params(**params)</code>	Set the parameters of the estimator.

**`__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)

**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):  $\sqrt{\text{tr}(A^t A)}$  - ‘spectral’:  $\sqrt{\max(\text{eigenvalues}(A^t A))}$  where A is the error (`comp_cov - self.covariance_`).

**scaling: bool :**

If True (default), the squared error norm is divided by n\_features. If False, the squared error norm is not rescaled.

**squared: bool :**

Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

**Returns The Mean Squared Error (in the sense of the Frobenius norm) between :**

**‘self’ and ‘comp\_cov’ covariance estimators. :**

**`fit(X, y=None)`**

Fits the shrunk covariance model according to the given training data and parameters.

**Parameters X: array-like, shape = [n\_samples, n\_features] :**

Training data, where n\_samples is the number of samples and n\_features is the number of features.

**y: not used, present for API consistence purpose. :**

**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 self: object :**

Returns self.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**get\_precision ()**

Getter for the precision matrix.

**Returns precision\_: array-like, :**

The precision matrix associated to the current covariance object.

**mahalanobis (observations)**

Computes the mahalanobis distances of given observations.

The provided observations are assumed to be centered. One may want to center them using a location estimate first.

**Parameters observations: 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 tha data used in fit (including centering).

**Returns mahalanobis\_distance: array, shape = [n\_observations,] :**

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.

**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 tha 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 the 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 :**

<code>covariance.empirical_covariance(X[, ...])</code>	Computes the Maximum likelihood covariance estimator
<code>covariance.ledoit_wolf(X[, assume_centered, ...])</code>	Estimates the shrunk Ledoit-Wolf covariance matrix.
<code>covariance.shrunk_covariance(emp_cov[, ...])</code>	Calculates a covariance matrix shrunk on the diagonal
<code>covariance.oas(X[, assume_centered])</code>	Estimate covariance with the Oracle Approximating Shrinkage algorithm
<code>covariance.graph_lasso(emp_cov, alpha[, ...])</code>	l1-penalized covariance estimator

## sklearn.covariance.empirical\_covariance

`sklearn.covariance.empirical_covariance(X, assume_centered=False)`

Computes the Maximum likelihood covariance estimator

**Parameters X: 2D 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.

**Returns covariance: 2D ndarray, shape (n\_features, n\_features) :**

Empirical covariance (Maximum Likelihood Estimator)

## sklearn.covariance.ledoit\_wolf

`sklearn.covariance.ledoit_wolf(X, assume_centered=False, block_size=1000)`

Estimates the shrunk Ledoit-Wolf covariance matrix.

**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. Usefull 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 :**

Size of the blocks into which the covariance matrix will be split. If `n_features > block_size`, an error will be raised since the shrunk covariance matrix will be considered as too large regarding the available memory.

**Returns shrunk\_cov: array-like, shape (n\_features, n\_features) :**

Shrunk covariance.

**shrinkage: float :**

Coefficient in the convex combination used for the computation of the shrunk estimate.

## Notes

The regularized (shrunk) covariance is:

**(1 - shrinkage)\*cov**

•shrinkage \* mu \* np.identity(n\_features)

where mu = trace(cov) / n\_features

## sklearn.covariance.shrunk\_covariance

`sklearn.covariance.shrunk_covariance(emp_cov, shrinkage=0.1)`

Calculates a covariance matrix shrunk on the diagonal

**Parameters emp\_cov: array-like, shape (n\_features, n\_features) :**

Covariance matrix to be shrunk

**shrinkage: float, 0 <= shrinkage <= 1 :**

coefficient in the convex combination used for the computation of the shrunk estimate.

**Returns shrunk\_cov: array-like :**

shrunk covariance

## Notes

The regularized (shrunk) covariance is given by

**(1 - shrinkage)\*cov**

•shrinkage\*mu\*np.identity(n\_features)

where mu = trace(cov) / n\_features

## 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>).

## 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=2.2204460492503131e-16)
```

11-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.

**Returns covariance** : 2D ndarray, shape (n\_features, n\_features)

The estimated covariance matrix

**precision** : 2D ndarray, shape (n\_features, n\_features)

The estimated (sparse) precision matrix

**costs** : list of (objective, dual\_gap) pairs

The list of values of the objective function and the dual gap at each iteration. Returned only if return\_costs is True

#### 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.

### 1.8.3 `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.

<code>cross_validation.Bootstrap(n[, n_iter, ...])</code>	Random sampling with replacement cross-validation iterator
<code>cross_validation.KFold(n[, n_folds, ...])</code>	K-Folds cross validation iterator.
<code>cross_validation.LeaveOneLabelOut(labels[, ...])</code>	Leave-One-Label_Out cross-validation iterator
<code>cross_validation.LeaveOneOut(n[, indices])</code>	Leave-One-Out cross validation iterator.
<code>cross_validation.LeavePLabelOut(labels, p[, ...])</code>	Leave-P-Label_Out cross-validation iterator
<code>cross_validation.LeavePOut(n, p[, indices])</code>	Leave-P-Out cross validation iterator
<code>cross_validation.StratifiedKFold(y[, ...])</code>	Stratified K-Folds cross validation iterator
<code>cross_validation.ShuffleSplit(n[, n_iter, ...])</code>	Random permutation cross-validation iterator.
<code>cross_validation.StratifiedShuffleSplit(y[, ...])</code>	Stratified ShuffleSplit cross validation iterator

#### `sklearn.cross_validation.Bootstrap`

```
class sklearn.cross_validation.Bootstrap(n, n_iter=3, train_size=0.5, test_size=None, random_state=None, n_bootstraps=None)
```

Random sampling with replacement cross-validation iterator

Provides train/test indices to split data in train test sets while resampling the input n\_iter times: each time a new random split of the data is performed and then samples are drawn (with replacement) on each side of the split to build the training and test sets.

Note: contrary to other cross-validation strategies, bootstrapping will allow some samples to occur several times in each splits. However a sample that occurs in the train split will never occur in the test split and vice-versa.

If you want each sample to occur at most once you should probably use `ShuffleSplit` cross validation instead.

##### Parameters

**n** : int

Total number of elements in the dataset.

**n\_iter** : int (default is 3)

Number of bootstrapping iterations

**train\_size** : int or float (default is 0.5)

If int, number of samples to include in the training split (should be smaller than the total number of samples passed in the dataset).

If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split.

**test\_size** : int or float or None (default is None)

If int, number of samples to include in the training set (should be smaller than the total number of samples passed in the dataset).

If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split.

If None, n\_test is set as the complement of n\_train.

**random\_state** : int or RandomState

Pseudo number generator state used for random sampling.

**See Also:**

**ShuffleSplit** cross validation using random permutations.

**Examples**

```
>>> from sklearn import cross_validation
>>> bs = cross_validation.Bootstrap(9, random_state=0)
>>> len(bs)
3
>>> print(bs)
Bootstrap(9, n_iter=3, train_size=5, test_size=4, random_state=0)
>>> for train_index, test_index in bs:
...     print("TRAIN:", train_index, "TEST:", test_index)
...
TRAIN: [1 8 7 7 8] TEST: [0 3 0 5]
TRAIN: [5 4 2 4 2] TEST: [6 7 1 0]
TRAIN: [4 7 0 1 1] TEST: [5 3 6 5]

__init__(n, n_iter=3, train_size=0.5, test_size=None, random_state=None, n_bootstraps=None)
```

**sklearn.cross\_validation.KFold**

```
class sklearn.cross_validation.KFold(n, n_folds=3, indices=True, shuffle=False, random_state=None, k=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.

**Parameters** **n** : int

Total number of elements.

**n\_folds** : int, default=3

Number of folds.

**indices** : boolean, optional (default True)

Return train/test split as arrays of indices, rather than a boolean mask array. Integer indices are required when dealing with sparse matrices, since those cannot be indexed by boolean masks.

**shuffle** : boolean, optional

Whether to shuffle the data before splitting into batches.

**random\_state** : int or RandomState

Pseudo number generator state used for random sampling.

#### See Also:

**StratifiedKFold** take label information into account to avoid building folds, classification

#### Notes

All the folds have size  $\text{trunc}(n\_samples / n\_folds)$ , 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([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)
>>> 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]

__init__(n, n_folds=3, indices=True, shuffle=False, random_state=None, k=None)
```

### sklearn.cross\_validation.LeaveOneLabelOut

```
class sklearn.cross_validation.LeaveOneLabelOut(labels, indices=True)
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.

**Parameters labels** : array-like of int with shape (n\_samples,)

Arbitrary domain-specific stratification of the data to be used to draw the splits.

**indices** : boolean, optional (default True)

Return train/test split as arrays of indices, rather than a boolean mask array. Integer indices are required when dealing with sparse matrices, since those cannot be indexed by boolean masks.

## 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]

__init__(labels, indices=True)
```

## sklearn.cross\_validation.LeaveOneOut

```
class sklearn.cross_validation.LeaveOneOut(n, indices=True)
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.

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.

### Parameters n : int

Total number of elements in dataset.

### indices : boolean, optional (default True)

Return train/test split as arrays of indices, rather than a boolean mask array. Integer indices are required when dealing with sparse matrices, since those cannot be indexed by boolean masks.

### 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]

__init__(n, indices=True)
```

## sklearn.cross\_validation.LeavePLabelOut

```
class sklearn.cross_validation.LeavePLabelOut(labels, p, indices=True)
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.

**Parameters** **labels** : 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.

**indices** : boolean, optional (default True)

Return train/test split as arrays of indices, rather than a boolean mask array. Integer indices are required when dealing with sparse matrices, since those cannot be indexed by boolean masks.

## 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]

__init__(labels, p, indices=True)
```

## sklearn.cross\_validation.LeavePOut

```
class sklearn.cross_validation.LeavePOut(n, p, indices=True)
Leave-P-out cross validation iterator
```

Provides train/test indices to split data in train test sets. The test set is built using p samples while the remaining samples form the training set.

Due to the high number of iterations which grows with the number of samples this cross validation method can be very costly. For large datasets one should favor KFold, StratifiedKFold or ShuffleSplit.

### Parameters n : int

Total number of elements in dataset.

### p : int

Size of the test sets.

### indices : boolean, optional (default True)

Return train/test split as arrays of indices, rather than a boolean mask array. Integer indices are required when dealing with sparse matrices, since those cannot be indexed by boolean masks.

## 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]

__init__(n, p, indices=True)

```

## sklearn.cross\_validation.StratifiedKFold

**class** `sklearn.cross_validation.StratifiedKFold(y, n_folds=3, indices=True, k=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, which returns stratified folds. The folds are made by preserving the percentage of samples for each class.

**Parameters** `y` : array-like, [n\_samples]

Samples to split in K folds.

`n_folds` : int, default=3

Number of folds.

`indices` : boolean, optional (default True)

Return train/test split as arrays of indices, rather than a boolean mask array. Integer indices are required when dealing with sparse matrices, since those cannot be indexed by boolean masks.

### 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)
>>> 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]

__init__(y, n_folds=3, indices=True, k=None)

```

## sklearn.cross\_validation.ShuffleSplit

```
class sklearn.cross_validation.ShuffleSplit(n, n_iter=10, test_size=None,
                                             indices=True, 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.

### Parameters

**n** : 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

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.

**indices** : boolean, optional (default True)

Return train/test split as arrays of indices, rather than a boolean mask array. Integer indices are required when dealing with sparse matrices, since those cannot be indexed by boolean masks.

**random\_state** : int or RandomState

Pseudo-random number generator state used for random sampling.

### See Also:

**Bootstrap**cross-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, indices=True, ...)
>>> 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]

__init__(n, n_iter=10, test_size=0.1, train_size=None, indices=True, random_state=None,
n_iterations=None)

```

## sklearn.cross\_validation.StratifiedShuffleSplit

```

class sklearn.cross_validation.StratifiedShuffleSplit(y, n_iter=10, test_size=0.1,
                                                      train_size=None, indices=True,
                                                      random_state=None,
                                                      n_iterations=None)

```

Stratified ShuffleSplit cross validation iterator

Provides train/test indices to split data in train test sets.

This cross-validation object is a merge of StratifiedKFold and ShuffleSplit, which returns stratified randomized folds. The folds are made by preserving the percentage of samples for each class.

Note: like the ShuffleSplit strategy, stratified random splits do not guarantee that all folds will be different, although this is still very likely for sizeable datasets.

**Parameters** `y` : 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.

`indices` : boolean, optional (default True)

Return train/test split as arrays of indices, rather than a boolean mask array. Integer indices are required when dealing with sparse matrices, since those cannot be indexed by boolean masks.

## 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]

__init__(y, n_iter=10, test_size=0.1, train_size=None, indices=True, random_state=None,
n_iterations=None)
```

<code>cross_validation.train_test_split(*arrays, ...)</code>	Split arrays or matrices into random train and test subsets
<code>cross_validation.cross_val_score(estimator, X)</code>	Evaluate a score by cross-validation
<code>cross_validation.permutation_test_score(...)</code>	Evaluate the significance of a cross-validated score with permutations
<code>cross_validation.check_cv(cv[, X, y, classifier])</code>	Input checker utility for building a CV in a user friendly way.

## 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 calls to `check_arrays` 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.

`dtype` : a numpy dtype instance, None by default

Enforce a specific dtype.

## 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
array([2, 0, 3])
>>> a_test
array([[2, 3],
       [8, 9]])
>>> b_test
array([1, 4])
```

## `sklearn.cross_validation.cross_val_score`

```
sklearn.cross_validation.cross_val_score(estimator, X, y=None, score_func=None,
                                         cv=None, n_jobs=1, verbose=0,
                                         fit_params=None)
```

Evaluate a score by cross-validation

**Parameters** `estimator` : estimator object implementing ‘fit’

The object to use to fit the data.

`X` : array-like of shape at least 2D

The data to fit.

`y` : array-like, optional

The target variable to try to predict in the case of supervised learning.

`score_func` : callable, optional

Score function to use for evaluation. Has priority over the score function in the estimator. In a non-supervised setting, where `y` is None, it takes the test data (`X_test`) as its only argument. In a supervised setting it takes the test target (`y_true`) and the test prediction (`y_pred`) as arguments.

`cv` : cross-validation generator, optional

A cross-validation generator. If None, a 3-fold cross validation is used or 3-fold stratified cross-validation when `y` is supplied and estimator is a classifier.

**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.

## sklearn.cross\_validation.permutation\_test\_score

```
sklearn.cross_validation.permutation_test_score(estimator, X, y, score_func, cv=None,  
n_permutations=100, n_jobs=1,  
labels=None, random_state=0, verbose=0)
```

Evaluate the significance of a cross-validated score with permutations

**Parameters estimator** : estimator object implementing ‘fit’

The object to use to fit the data.

**X** : array-like of shape at least 2D

The data to fit.

**y** : array-like

The target variable to try to predict in the case of supervised learning.

**score\_func** : callable

Callable taking as arguments the test targets (`y_test`) and the predicted targets (`y_pred`) and returns a float. The score functions are expected to return a bigger value for a better result otherwise the returned value does not correspond to a p-value (see Returns below for further details).

**cv** : integer or crossvalidation generator, optional

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

**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.

**Returns score** : 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 *score\_func* returns bigger numbers for better scores (e.g., `accuracy_score`). If *score\_func* 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

## 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.

**Parameters** `cv` : 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.

## 1.8.4 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.

## Loaders

<code>datasets.fetch_20newsgroups([data_home, ...])</code>	Load the filenames of the 20 newsgroups dataset.
<code>datasets.fetch_20newsgroups_vectorized([...])</code>	Load the 20 newsgroups dataset and transform it into tf-idf vectors.
<code>datasets.load_boston()</code>	Load and return the boston house-prices dataset (regression).
<code>datasets.load_diabetes()</code>	Load and return the diabetes dataset (regression).
<code>datasets.load_digits([n_class])</code>	Load and return the digits dataset (classification).
<code>datasets.load_files(container_path[, ...])</code>	Load text files with categories as subfolder names.
<code>datasets.load_iris()</code>	Load and return the iris dataset (classification).

Continued on next page

**Table 1.45 – continued from previous page**

<code>datasets.load_lfw_pairs([download_if_missing])</code>	Alias for <code>fetch_lfw_pairs(download_if_missing=False)</code>
<code>datasets.fetch_lfw_pairs([subset, ...])</code>	Loader for the Labeled Faces in the Wild (LFW) pairs dataset
<code>datasets.load_lfw_people([download_if_missing])</code>	Alias for <code>fetch_lfw_people(download_if_missing=False)</code>
<code>datasets.fetch_lfw_people([data_home, ...])</code>	Loader for the Labeled Faces in the Wild (LFW) people dataset
<code>datasets.load_linnerud()</code>	Load and return the linnerud dataset (multivariate regression).
<code>datasets.fetch_mldata(dataname[, ...])</code>	Fetch an mldata.org data set
<code>datasets.fetch_olivetti_faces([data_home, ...])</code>	Loader for the Olivetti faces data-set from AT&T.
<code>datasets.fetch_california_housing([...])</code>	Loader for the California housing dataset from StatLib.
<code>datasets.load_sample_image(image_name)</code>	Load the numpy array of a single sample image
<code>datasets.load_sample_images()</code>	Load sample images for image manipulation.
<code>datasets.load_svmlight_file(f[, n_features, ...])</code>	Load datasets in the svmlight / libsvm format into sparse CSR matrix
<code>datasets.dump_svmlight_file(X, y, f[, ...])</code>	Dump the dataset in svmlight / libsvm file format.

## sklearn.datasets.fetch\_20newsgroups

```
sklearn.datasets.fetch_20newsgroups(data_home=None, subset='train', categories=None, shuffle=True, random_state=42, download_if_missing=True)
```

Load the filenames of the 20 newsgroups dataset.

**Parameters** `subset: ‘train’ or ‘test’, ‘all’, optional` :

Select the dataset to load: ‘train’ for the training set, ‘test’ for the test set, ‘all’ for both, with shuffled ordering.

`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.

## sklearn.datasets.fetch\_20newsgroups\_vectorized

```
sklearn.datasets.fetch_20newsgroups_vectorized(subset='train', 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`.

**Parameters subset: ‘train’ or ‘test’, ‘all’, optional :**

Select the dataset to load: ‘train’ for the training set, ‘test’ for the test set, ‘all’ for both, with shuffled ordering.

**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.

**Returns bunch : Bunch object**

bunch.data: sparse matrix, shape [n\_samples, n\_features] bunch.target: array, shape [n\_samples] bunch.target\_names: list, length [n\_classes]

**`sklearn.datasets.load_boston`****`sklearn.datasets.load_boston()`**

Load and return the boston house-prices dataset (regression).

Samples total	506
Dimensionality	13
Features	real, positive
Targets	real 5. - 50.

**Returns data : Bunch**

Dictionary-like object, the interesting attributes are: ‘data’, the data to learn, ‘target’, the regression targets, ‘target\_names’, the meaning of the labels, and ‘DESCR’, the full description of the dataset.

**Examples**

```
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()
>>> print boston.data.shape
(506, 13)
```

**`sklearn.datasets.load_diabetes`****`sklearn.datasets.load_diabetes()`**

Load and return the diabetes dataset (regression).

Samples total	442
Dimensionality	10
Features	real, -.2 < x < .2
Targets	integer 25 - 346

**Returns data : Bunch**

Dictionary-like object, the interesting attributes are: ‘data’, the data to learn and ‘target’, the regression target for each sample.

## 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

**Parameters n\_class** : integer, between 0 and 10, optional (default=10)

The number of classes to return.

**Returns data** : Bunch

Dictionary-like object, the interesting attributes are: ‘data’, the data to learn, ‘images’, the images corresponding to each sample, ‘target’, the classification labels for each sample, ‘target\_names’, the meaning of the labels, and ‘DESCR’, the full description of the dataset.

## 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()
```

## sklearn.datasets.load\_files

```
sklearn.datasets.load_files(container_path,           description=None,           categories=None,
                           load_content=True,          shuffle=True,          charset=None,
                           encoding_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 has 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 utf-8 text files in a scikit-learn classification or clustering algorithm you will first need to use the `sklearn.features.text` module to build a feature extraction transformer that suits your problem.

Similar feature extractors should be build for other kind of unstructured data input such as images, audio, video,

...

**Parameters**

**container\_path** : string or unicode

Path to the main folder holding one subfolder per category

**description: string or unicode, optional (default=None)** :

A paragraph describing the characteristic of the dataset: its source, reference, etc.

**categories** : A collection of strings or None, optional (default=None)

If None (default), load all the categories. If not None, list of category names to load (other categories ignored).

**load\_content** : boolean, optional (default=True)

Whether to load or not the content of the different files. If true a ‘data’ attribute containing the text information is present in the data structure returned. If not, a filenames attribute gives the path to the files.

**charset** : 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, charset to use to decode text files if load\_content is True.

**charset\_error: {‘strict’, ‘ignore’, ‘replace’}** :

Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given *charset*. By default, it is ‘strict’, meaning that a UnicodeDecodeError will be raised. Other values are ‘ignore’ and ‘replace’.

**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*.

**Returns**

**data** : Bunch

Dictionary-like object, the interesting attributes are: either data, the raw text data to learn, or ‘filenames’, the files holding it, ‘target’, the classification labels (integer index), ‘target\_names’, the meaning of the labels, and ‘DESCR’, the full description of the dataset.

## sklearn.datasets.load\_iris

### sklearn.datasets.load\_iris()

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

**Returns data :** Bunch

Dictionary-like object, the interesting attributes are: ‘data’, the data to learn, ‘target’, the classification labels, ‘target\_names’, the meaning of the labels, ‘feature\_names’, the meaning of the features, and ‘DESCR’, the full description of the dataset.

## 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']
```

## 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.

## 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.

**Parameters subset: optional, default: ‘train’ :**

Select the dataset to load: ‘train’ for the development training set, ‘test’ for the development test set, and ‘10\_folds’ for the official evaluation set that is meant to be used with a 10-folds cross validation.

**data\_home: optional, default: None :**

Specify another download and cache folder for the datasets. By default all scikit learn data is stored in ‘~/scikit\_learn\_data’ subfolders.

**funneled: boolean, optional, default: True :**

Download and use the funneled variant of the dataset.

**resize: float, optional, default 0.5 :**

Ratio used to resize the each face picture.

**color: boolean, optional, default False :**

Keep the 3 RGB channels instead of averaging them to a single gray level channel. If color is True the shape of the data has one more dimension than 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.

## 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=None,   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).

**Parameters data\_home: optional, default: None :**

Specify another download and cache folder for the datasets. By default all scikit learn data is stored in ‘~/scikit\_learn\_data’ subfolders.

**funneled: boolean, optional, default: True :**

Download and use the funneled variant of the dataset.

**resize: float, optional, default 0.5 :**

Ratio used to resize the each face picture.

**min\_faces\_per\_person: int, optional, default None :**

The extracted dataset will only retain pictures of people that have at least *min\_faces\_per\_person* different pictures.

**color: boolean, optional, default False :**

Keep the 3 RGB channels instead of averaging them to a single gray level channel. If color is True the shape of the data has one more dimension than 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.

**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

**Returns data : 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.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.

**Parameters data name:** :

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.

**Returns data :** 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 >>> iris = fetch_mldata('iris') >>> iris.target[0] 1 >>> print(iris.data[0]) [-0.555556 0.25 -0.864407 -0.916667]
```

```
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) >>> print(leuk.data.shape[0]) 72
```

```
Load an alternative ‘iris’ dataset, which has different names for the columns: >>> iris2 = fetch_mldata('datasets-UCI iris', target_name=1, ... data_name=0) >>> iris3 = fetch_mldata('datasets-UCI iris', ... target_name='class', data_name='double0')
```

## 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.

## 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/>

## sklearn.datasets.fetch\_california\_housing

```
sklearn.datasets.fetch_california_housing(data_home=None,           down-
                                           load_if_missing=True)
```

Loader for the California housing dataset from StatLib.

**Parameters data\_home** : optional, default: None

Specify another download and cache folder for the datasets. By default all scikit learn data is stored in ‘~/scikit\_learn\_data’ subfolders.

**download\_if\_missing: optional, True by default :**

If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

## Notes

This dataset consists of 20,640 samples and 9 features.

## sklearn.datasets.load\_sample\_image

```
sklearn.datasets.load_sample_image(image_name)
```

Load the numpy array of a single sample image

**Parameters image\_name: {‘china.jpg’, ‘flower.jpg’} :**

The name of the sample image loaded

**Returns img: 3D array :**

The image as a numpy array: height x width x color

## Examples

```
>>> from sklearn.datasets import load_sample_image
>>> china = load_sample_image('china.jpg')
>>> china.dtype
dtype('uint8')
>>> china.shape
(427, 640, 3)
>>> flower = load_sample_image('flower.jpg')
>>> flower.dtype
dtype('uint8')
>>> flower.shape
(427, 640, 3)
```

## sklearn.datasets.load\_sample\_images

```
sklearn.datasets.load_sample_images()
```

Load sample images for image manipulation. Loads both, `china` and `flower`.

**Returns data** : Bunch

Dictionary-like object with the following attributes : ‘images’, the two sample images, ‘filenames’, the file names for the images, and ‘DESCR’ the full description of the dataset.

## Examples

To load the data and visualize the images:

```
>>> from sklearn.datasets import load_sample_images
>>> dataset = load_sample_images()
>>> 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.

This implementation is naive: it does allocate too much memory and is slow since written in python. On large datasets it is recommended to use an optimized loader such as:

<https://github.com/mblondel/svmlight-loader>

In case the file contains a pairwise preference constraint (known as “qid” in the svmlight format) these are ignored unless the query\_id parameter is set to True. These pairwise preference constraints can be used to constraint the combination of samples when using pairwise loss functions (as is the case in some learning to rank problems) so that only pairs with the same query\_id value are considered.

**Parameters f: {str, file-like, int} :**

(Path to) a file to load. If a path ends in ”.gz” or ”.bz2”, it will be uncompressed on the fly. If an integer is passed, it is assumed to be a file descriptor. A file-like or file descriptor will not be closed by this function. A file-like object must be opened in binary mode.

**n\_features: int or None :**

The number of features to use. If None, it will be inferred. This argument is useful to load several files that are subsets of a bigger sliced dataset: each subset might not have example of every feature, hence the inferred shape might vary from one slice to another.

**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.

**Returns X: scipy.sparse matrix of shape (n\_samples, n\_features) :**

**y: ndarray of shape (n\_samples,), or, in the multilabel a list of :**

tuples of length n\_samples.

**query\_id: array of shape (n\_samples,) :**

query\_id for each sample. Only returned when query\_id is set to True.

**See Also:**

**load\_svmlight\_file** similar function for loading multiple files in this

format, enforcing

**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.

**Parameters X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features.

**y** : array-like, shape = [n\_samples]

Target values.

**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).

## Samples generator

<code>datasets.make_blobs([n_samples, n_features, ...])</code>	Generate isotropic Gaussian blobs for clustering.
<code>datasets.make_classification([n_samples, ...])</code>	Generate a random n-class classification problem.
<code>datasets.make_circles([n_samples, shuffle, ...])</code>	Make a large circle containing a smaller circle in 2d.
<code>datasets.make_friedman1([n_samples, ...])</code>	Generate the “Friedman #1” regression problem
<code>datasets.make_friedman2([n_samples, noise, ...])</code>	Generate the “Friedman #2” regression problem
<code>datasets.make_friedman3([n_samples, noise, ...])</code>	Generate the “Friedman #3” regression problem
<code>datasets.make_hastie_10_2([n_samples, ...])</code>	Generates data for binary classification used in
<code>datasets.make_low_rank_matrix([n_samples, ...])</code>	Generate a mostly low rank matrix with bell-shaped singular value
<code>datasets.make_moons([n_samples, shuffle, ...])</code>	Make two interleaving half circles
<code>datasets.make_multilabel_classification([...])</code>	Generate a random multilabel classification problem.
<code>datasets.make_regression([n_samples, ...])</code>	Generate a random regression problem.
<code>datasets.make_s_curve([n_samples, noise, ...])</code>	Generate an S curve dataset.
<code>datasets.make_sparse_coded_signal(n_samples, ...)</code>	Generate a signal as a sparse combination of dictionary elements.
<code>datasets.make_sparse_spd_matrix([dim, ...])</code>	Generate a sparse symmetric definite positive matrix.
<code>datasets.make_sparse_uncorrelated([...])</code>	Generate a random regression problem with sparse uncorrelated data.
<code>datasets.make_spd_matrix(n_dim[, random_state])</code>	Generate a random symmetric, positive-definite matrix.
<code>datasets.make_swiss_roll([n_samples, noise, ...])</code>	Generate a swiss roll dataset.

### `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.

**Parameters n\_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*.

**Returns X** : array of shape [n\_samples, n\_features]

The generated samples.

**y** : array of shape [n\_samples]

The integer labels for cluster membership of each sample.

## 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])
```

## 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.

**Parameters n\_samples** : int, optional (default=100)

The number of samples.

**n\_features** : int, optional (default=20)

The total number of features. These comprise *n\_informative* informative features, *n\_redundant* redundant features, *n\_repeated* duplicated features and *n\_features-n\_informative-n\_redundant-n\_repeated* useless features drawn at random.

**n\_informative** : int, optional (default=2)

The number of informative features. Each class is composed of a number of gaussian clusters each located around the vertices of a hypercube in a subspace of dimension  $n_{informative}$ . For each cluster, informative features are drawn independently from  $N(0, 1)$  and then randomly linearly combined 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=2)

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  $\text{len}(weights) == n_{classes} - 1$ , then the last class weight is automatically inferred.

**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 or None, optional (default=0.0)

Shift all features by the specified value. If None, then features are shifted by a random value drawn in [-class\_sep, class\_sep].

**scale** : float or None, optional (default=1.0)

Multiply all 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`.

**Returns** **X** : array of shape [n\_samples, n\_features]

The generated samples.

**y** : array of shape [n\_samples]

The integer labels for class membership of each sample.

## Notes

The algorithm is adapted from Guyon [1] and was designed to generate the “Madelon” dataset.

## References

[R67]

### 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.

**Parameters** **n\_samples** : int, optional (default=100)

The total number of points generated.

**shuffle: bool, optional (default=True)** :

Whether to shuffle the samples.

**noise** : double or None (default=None)

Standard deviation of Gaussian noise added to the data.

**factor** : double < 1 (default=.8)

Scale factor between inner and outer circle.

**Returns** **X** : array of shape [n\_samples, 2]

The generated samples.

**y** : array of shape [n\_samples]

The integer labels (0 or 1) for class membership of each sample.

### sklearn.datasets.make\_friedman1

```
sklearn.datasets.make_friedman1(n_samples=100,      n_features=10,      noise=0.0,      ran-  
                                dom_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  $\geq 5$ .

**Parameters** `n_samples` : int, optional (default=100)

The number of samples.

`n_features` : int, optional (default=10)

The number of features. Should be at least 5.

`noise` : float, optional (default=0.0)

The standard deviation of the gaussian noise applied to the output.

`random_state` : int, RandomState instance or None, 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 input samples.

`y` : array of shape [n\_samples]

The output values.

## References

[R68], [R69]

### `sklearn.datasets.make_friedman2`

`sklearn.datasets.make_friedman2(n_samples=100, noise=0.0, random_state=None)`

Generate the “Friedman #2” regression problem

This dataset is described in Friedman [1] and Breiman [2].

Inputs  $X$  are 4 independent features uniformly distributed on the intervals:

```
0 <= X[:, 0] <= 100,
40 * pi <= X[:, 1] <= 560 * pi,
0 <= X[:, 2] <= 1,
1 <= X[:, 3] <= 11.
```

The output  $y$  is created according to the formula:

```
y(X) = (X[:, 0] ** 2 + (X[:, 1] * X[:, 2] - 1 / (X[:, 1] * X[:, 3])) ** 2) ** 0.5 + noise * N(0, 1)
```

**Parameters** `n_samples` : int, optional (default=100)

The number of samples.

`noise` : float, optional (default=0.0)

The standard deviation of the gaussian noise applied to the output.

`random_state` : int, RandomState instance or None, 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, 4]

The input samples.

**y** : array of shape [n\_samples]

The output values.

## References

[R70], [R71]

## 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).
```

**Parameters n\_samples** : int, optional (default=100)

The number of samples.

**noise** : float, optional (default=0.0)

The standard deviation of the gaussian noise applied to the output.

**random\_state** : int, RandomState instance or None, 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, 4]

The input samples.

**y** : array of shape [n\_samples]

The output values.

## References

[R72], [R73]

**sklearn.datasets.make\_hastie\_10\_2**

```
sklearn.datasets.make_hastie_10_2(n_samples=12000, random_state=None)
```

Generates data for binary classification used in Hastie et al. 2009, Example 10.2.

The ten features are standard independent Gaussian and the target  $y$  is defined by:

```
y[i] = 1 if np.sum(X[i]) > 9.34 else -1
```

**Parameters n\_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*.

**Returns X** : array of shape [n\_samples, 10]

The input samples.

**y** : array of shape [n\_samples]

The output values.

**\*\*References:\*\* :**

.. [1] T. Hastie, R. Tibshirani and J. Friedman, “Elements of Statistical Learning Ed. 2”, Springer, 2009. :

**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

**Parameters n\_samples** : int, optional (default=100)

The number of samples.

**n\_features** : int, optional (default=100)

The number of features.

**effective\_rank** : int, optional (default=10)

The approximate number of singular vectors required to explain most of the data by linear combinations.

**tail\_strength** : float between 0.0 and 1.0, optional (default=0.5)

The relative importance of the fat noisy tail of the singular values profile.

**random\_state** : int, RandomState instance or None, 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 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.

**Parameters n\_samples** : int, optional (default=100)

The total number of points generated.

**shuffle** : bool, optional (default=True)

Whether to shuffle the samples.

**noise** : double or None (default=None)

Standard deviation of Gaussian noise added to the data.

**Returns X** : array of shape [n\_samples, 2]

The generated samples.

**y** : array of shape [n\_samples]

The integer labels (0 or 1) for class membership of each sample.

### 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, random_state=None)`

Generate a random multilabel classification problem.

**For each sample, the generative process is:**

- pick the number of labels:  $n \sim \text{Poisson}(n\_labels)$
- $n$  times, choose a class  $c$ :  $c \sim \text{Multinomial}(\theta)$
- pick the document length:  $k \sim \text{Poisson}(\text{length})$

- k times, choose a word:  $w \sim \text{Multinomial}(\theta_c)$

In the above process, rejection sampling is used to make sure that n is never zero or more than  $n_{\text{classes}}$ , 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. Number of labels follows a Poisson distribution that never takes the value 0.

**length** : int, optional (default=50)

Sum of the features (number of words if documents).

**allow\_unlabeled** : bool, optional (default=True)

If True, some instances might not belong to any class.

**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** : list of tuples

The label sets.

## 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 the `make_low_rank_matrix` for more details.

The output is generated by applying a (potentially biased) random linear regression model with  $n_{\text{informative}}$  nonzero regressors to the previously generated input and some gaussian centered noise with some adjustable scale.

**Parameters** **n\_samples** : int, optional (default=100)

The number of samples.

**n\_features** : int, optional (default=100)

The number of features.

**n\_informative** : int, optional (default=10)

The number of informative features, i.e., the number of features used to build the linear model used to generate the output.

**n\_targets** : int, optional (default=1)

The number of regression targets, i.e., the dimension of the y output vector associated with a sample. By default, the output is a scalar.

**bias** : float, optional (default=0.0)

The bias term in the underlying linear model.

**effective\_rank** : int or None, optional (default=None)

**if not None:** The approximate number of singular vectors required to explain most of the input data by linear combinations. Using this kind of singular spectrum in the input allows the generator to reproduce the correlations often observed in practice.

**if None:** The input set is well conditioned, centered and gaussian with unit variance.

**tail\_strength** : float between 0.0 and 1.0, optional (default=0.5)

The relative importance of the fat noisy tail of the singular values profile if *effective\_rank* is not None.

**noise** : float, optional (default=0.0)

The standard deviation of the gaussian noise applied to the output.

**shuffle** : boolean, optional (default=True)

Shuffle the samples and the features.

**coef** : boolean, optional (default=False)

If True, the coefficients of the underlying linear model are returned.

**random\_state** : int, RandomState instance or None, 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 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.

## sklearn.datasets.make\_s\_curve

`sklearn.datasets.make_s_curve(n_samples=100, noise=0.0, random_state=None)`

Generate an S curve dataset.

**Parameters n\_samples** : int, optional (default=100)

The number of sample points on the S curve.

**noise** : float, optional (default=0.0)

The standard deviation of the gaussian noise.

**random\_state** : int, RandomState instance or None, 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, 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.

### `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.

**Parameters** **n\_samples** : int

number of samples to generate

**n\_components** : int :

number of components in the dictionary

**n\_features** : int

number of features of the dataset to generate

**n\_nonzero\_coefs** : int

number of active (non-zero) coefficients in each sample

**random\_state**: int or RandomState instance, optional (default=None) :

seed used by the pseudo random number generator

**Returns data**: array of shape [n\_features, n\_samples] :

The encoded signal (Y).

**dictionary**: array of shape [n\_features, n\_components] :

The dictionary with normalized components (D).

**code**: array of shape [n\_components, n\_samples] :

The sparse code such that each column of this matrix has exactly n\_nonzero\_coefs non-zero items (X).

**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.

**Parameters dim: integer, optional (default=1) :**

The size of the random (matrix to generate).

**alpha: float between 0 and 1, optional (default=0.95) :**

The probability that a coefficient is 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*.

**Returns prec: array of shape = [dim, dim] :**

**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.

**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.

**Parameters n\_samples : int, optional (default=100)**

The number of samples.

**n\_features : int, optional (default=10)**

The number of features.

**random\_state : int, RandomState instance or None, 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 input samples.

**y : array of shape [n\_samples]**

The output values.

## References

[R74]

### sklearn.datasets.make\_spd\_matrix

```
sklearn.datasets.make_spd_matrix(n_dim, random_state=None)
```

Generate a random symmetric, positive-definite matrix.

**Parameters** `n_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`.

**Returns** `X` : array of shape [n\_dim, n\_dim]

The random symmetric, positive-definite 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.

**Parameters** `n_samples` : int, optional (default=100)

The number of sample points on the S curve.

`noise` : float, optional (default=0.0)

The standard deviation of the gaussian noise.

`random_state` : int, RandomState instance or None, 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, 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

[R75]

## 1.8.5 `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.

<code>decomposition.PCA([n_components, copy, whiten])</code>	Principal component analysis (PCA)
<code>decomposition.ProbabilisticPCA(...)</code>	Additional layer on top of PCA that adds a probabilistic evaluation
<code>decomposition.ProjectiveGradientNMF(...)</code>	Non-Negative matrix factorization by Projected Gradient (NMF)
<code>decomposition.RandomizedPCA([n_components, ...])</code>	Principal component analysis (PCA) using randomized SVD
<code>decomposition.KernelPCA([n_components, ...])</code>	Kernel Principal component analysis (KPCA)
<code>decomposition.FactorAnalysis([n_components, ...])</code>	Factor Analysis (FA)
<code>decomposition.FastICA([n_components, ...])</code>	FastICA; a fast algorithm for Independent Component Analysis
<code>decomposition.NMF([n_components, init, ...])</code>	Non-Negative matrix factorization by Projected Gradient (NMF)
<code>decomposition.SparsePCA([n_components, ...])</code>	Sparse Principal Components Analysis (SparsePCA)
<code>decomposition.MiniBatchSparsePCA(...)</code>	Mini-batch Sparse Principal Components Analysis
<code>decomposition.SparseCoder(dictionary[, ...])</code>	Sparse coding
<code>decomposition.DictionaryLearning(...)</code>	Dictionary learning
<code>decomposition.MiniBatchDictionaryLearning(...)</code>	Mini-batch dictionary learning

### `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  $O(n \times n \times 3)$  assuming  $n \sim n_{samples} \sim n_{features}$ .

**Parameters** `n_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

`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.

**See Also:**

ProbabilisticPCA, RandomizedPCA, KernelPCA, SparsePCA

**Notes**

For n\_components='mle', this class uses the method of *Thomas P. Minka: Automatic Choice of Dimensionality for PCA. NIPS 2000: 598-604*

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...]
```

**Attributes**

<i>components_</i>	array, [n_components, n_features]	Components with maximum variance.
<i>explained_variance_</i>	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

**Methods**

<code>fit(X[, y])</code>	Fit the model with X.
<code>fit_transform(X[, y])</code>	Fit the model with X and apply the dimensionality reduction on X.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>inverse_transform(X)</code>	Transform data back to its original space, i.e.,
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Apply the dimensionality reduction on X.

`__init__(n_components=None, copy=True, whiten=False)`

`fit(X, y=None, **params)`

Fit the model with X.

**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 instance itself.

**fit\_transform**(X, y=None, \*\*params)

Fit the model with X and apply the dimensionality reduction on X.

**Parameters X** : array-like, shape (n\_samples, n\_features)

Training data, where n\_samples in the number of samples and n\_features is the number of features.

**Returns X\_new** : array-like, shape (n\_samples, n\_components)

**get\_params**(deep=True)

Get parameters for the estimator

**Parameters deep: boolean, optional** :

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

**inverse\_transform**(X)

Transform data back to its original space, i.e., return an input X\_original whose transform would be X

**Parameters X** : 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.

**Returns X\_original array-like, shape (n\_samples, n\_features)** :

## Notes

If whitening is enabled, inverse\_transform does not compute the exact inverse operation as transform.

**set\_params**(\*\*params)

Set the parameters of the 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)

Apply the dimensionality reduction on X.

**Parameters X** : array-like, shape (n\_samples, n\_features)

New data, where n\_samples in the number of samples and n\_features is the number of features.

**Returns X\_new** : array-like, shape (n\_samples, n\_components)

## sklearn.decomposition.ProbabilisticPCA

**class** sklearn.decomposition.ProbabilisticPCA(n\_components=None, copy=True, whiten=False)

Additional layer on top of PCA that adds a probabilistic evaluationPrincipal 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  $O(n \times 3)$  assuming  $n \sim n_{samples} \sim n_{features}$ .

**Parameters**

**n\_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

**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.

### See Also:

`ProbabilisticPCA`, `RandomizedPCA`, `KernelPCA`, `SparsePCA`

### Notes

For `n_components='mle'`, this class uses the method of *Thomas P. Minka: Automatic Choice of Dimensionality for PCA. NIPS 2000: 598-604*

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...]
```

## Attributes

<i>components_</i>	array, [n_components, n_features]	Components with maximum variance.
<i>explained_variance_[n_components]</i>	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

## Methods

<code>fit(X[, y, homoscedastic])</code>	Additionally to PCA.fit, learns a covariance model
<code>fit_transform(X[, y])</code>	Fit the model with X and apply the dimensionality reduction on X.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>inverse_transform(X)</code>	Transform data back to its original space, i.e.,
<code>score(X[, y])</code>	Return a score associated to new data
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Apply the dimensionality reduction on X.

**`__init__(n_components=None, copy=True, whiten=False)`**

**`fit(X, y=None, homoscedastic=True)`**

Additionally to PCA.fit, learns a covariance model

**Parameters** `X` : array of shape(n\_samples, n\_features)

The data to fit

**homoscedastic** : bool, optional,

If True, average variance across remaining dimensions

**`fit_transform(X, y=None, **params)`**

Fit the model with X and apply the dimensionality reduction on X.

**Parameters** `X` : array-like, shape (n\_samples, n\_features)

Training data, where n\_samples in the number of samples and n\_features is the number of features.

**Returns** `X_new` : array-like, shape (n\_samples, n\_components)

**`get_params(deep=True)`**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**`inverse_transform(X)`**

Transform data back to its original space, i.e., return an input X\_original whose transform would be X

**Parameters** `X` : 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.

**Returns** `X_original` array-like, shape (n\_samples, n\_features) :

## Notes

If whitening is enabled, inverse\_transform does not compute the exact inverse operation as transform.

**score** ( $X, y=None$ )

Return a score associated to new data

**Parameters X: array of shape(n\_samples, n\_features) :**

The data to test

**Returns ll:** array of shape (n\_samples), :

log-likelihood of each row of X under the current model

**set\_params** (\*\*params)

Set the parameters of the 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** ( $X$ )

Apply the dimensionality reduction on X.

**Parameters X :** array-like, shape (n\_samples, n\_features)

New data, where n\_samples in the number of samples and n\_features is the number of features.

**Returns X\_new :** array-like, shape (n\_samples, n\_components)

## sklearn.decomposition.ProjecteGradientNMF

```
class sklearn.decomposition.ProjecteGradientNMF(n_components=None,      init=None,
                                                sparseness=None,   beta=1,   eta=0.1,
                                                tol=0.0001,          max_iter=200,
                                                nls_max_iter=2000,      ran-
                                                dom_state=None)
```

Non-Negative matrix factorization by Projected Gradient (NMF)

**Parameters n\_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.

## Notes

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://scgroup.hpclab.ceid.upatras.gr/faculty/stratis/Papers/HPCLAB020107.pdf>

## 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...
```

## Attributes

<i>components_</i>	array, [n_components, n_features]	Non-negative components of the data
<i>reconstruction_err_</i>	number	Frobenius norm of the matrix difference between the training data and the reconstructed data from the fit produced by the model. $\  X - WH \ _2^2$ Not computed for sparse input matrices because it is too expensive in terms of memory.

## Methods

<code>fit(X[, y])</code>	Learn a NMF model for the data X.
<code>fit_transform(X[, y])</code>	Learn a NMF model for the data X and returns the transformed data.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	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.

**Parameters X:** {array-like, sparse matrix}, **shape = [n\_samples, n\_features] :**

Data matrix to be decomposed

**Returns self :**

`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.

**Parameters X:** {array-like, sparse matrix}, **shape = [n\_samples, n\_features] :**

Data matrix to be decomposed

**Returns data:** array, [n\_samples, n\_components] :

Transformed data

`get_params(deep=True)`

Get parameters for the estimator

**Parameters deep:** boolean, optional :

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

**set\_params**(\**params*)

Set the parameters of the 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*)

Transform the data *X* according to the fitted NMF model

**Parameters X: {array-like, sparse matrix}, shape = [n\_samples, n\_features] :**

Data matrix to be transformed by the model

**Returns data: array, [n\_samples, n\_components] :**

Transformed data

## sklearn.decomposition.RandomizedPCA

```
class sklearn.decomposition.RandomizedPCA(n_components=None, copy=True, it-
                                             erated_power=3, whiten=False, ran-
                                             dom_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.

This implementation uses a randomized SVD implementation and can handle both `scipy.sparse` and `numpy` dense arrays as input.

**Parameters n\_components : int**

Maximum number of components to keep: default is 50.

**copy : bool**

If False, data passed to fit are overwritten

**iterated\_power : int, optional**

Number of iteration 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.

**See Also:**

`PCA`, `ProbabilisticPCA`

## 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...]
```

## Attributes

<i>components_</i>	array, [n_components, n_features]	Components with maximum variance.
<i>explained_variance_</i>	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

## Methods

<code>fit(X[, y])</code>	Fit the model to the data X.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>inverse_transform(X)</code>	Transform data back to its original space.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Apply dimensionality reduction on X.

`__init__(n_components=None, copy=True, iterated_power=3, whiten=False, random_state=None)`

**fit (X, y=None)**

Fit the model to the data X.

**Parameters X: array-like or scipy.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.

**Returns self : 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.

**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 the estimator

**Parameters** **deep**: boolean, optional :

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

**inverse\_transform**(*X*)

Transform data back to its original space.

Returns an array *X\_original* whose transform would be *X*.

**Parameters** **X** : array-like or scipy.sparse matrix, shape (n\_samples, n\_components)

New data, where n\_samples in the number of samples and n\_components is the number of components.

**Returns** **X\_original** array-like, shape (n\_samples, n\_features) :

## Notes

If whitening is enabled, inverse\_transform does not compute the exact inverse operation of transform.

**set\_params** (\*\**params*)

Set the parameters of the 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*)

Apply dimensionality reduction on *X*.

**Parameters** **X** : array-like or scipy.sparse matrix, shape (n\_samples, n\_features)

New data, where n\_samples in the number of samples and n\_features is the number of features.

**Returns** **X\_new** : array-like, shape (n\_samples, n\_components)

## sklearn.decomposition.KernelPCA

```
class sklearn.decomposition.KernelPCA(n_components=None, kernel='linear',
                                         gamma=0, degree=3, coef0=1, alpha=1.0,
                                         fit_inverse_transform=False, eigen_solver='auto',
                                         tol=0, max_iter=None)
```

Kernel Principal component analysis (KPCA)

Non-linear dimensionality reduction through the use of kernels.

**Parameters n\_components: int or None :**

Number of components. If None, all non-zero components are kept.

**kernel: “linear” | “poly” | “rbf” | “sigmoid” | “cosine” | “precomputed” :**

Kernel. Default: “linear”

**degree : int, optional**

Degree for poly, rbf and sigmoid kernels. Default: 3.

**gamma : float, optional**

Kernel coefficient for rbf and poly kernels. Default: 1/n\_features.

**coef0 : float, optional**

Independent term in poly and sigmoid 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)

## 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.

## Attributes

<i>lambdas_, alphas_:</i>	Eigenvalues and eigenvectors of the centered kernel matrix
<i>dual_coef_:</i>	Inverse transform matrix
<i>X_transformed_fit_:</i>	Projection of the fitted data on the kernel principal components

## Methods

<code>fit(X[, y])</code>	Fit the model from data in X.
<code>fit_transform(X[, y])</code>	Fit the model from data in X and transform X.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>inverse_transform(X)</code>	Transform X back to original space.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Transform X.

**`__init__(n_components=None, kernel='linear', gamma=0, degree=3, coef0=1, alpha=1.0, fit_inverse_transform=False, eigen_solver='auto', tol=0, max_iter=None)`**

**`fit(X, y=None)`**

Fit the model from data in X.

**Parameters X: array-like, shape (n\_samples, n\_features) :**

Training vector, where n\_samples in the number of samples and n\_features is the number of features.

**Returns self : object**

Returns the instance itself.

**`fit_transform(X, y=None, **params)`**

Fit the model from data in X and transform X.

**Parameters X: array-like, shape (n\_samples, n\_features) :**

Training vector, where n\_samples in the number of samples and n\_features is the number of features.

**Returns X\_new: array-like, shape (n\_samples, n\_components) :**

**`get_params(deep=True)`**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**`inverse_transform(X)`**

Transform X back to original space.

**Parameters X: array-like, shape (n\_samples, n\_components) :**

**Returns X\_new: array-like, shape (n\_samples, n\_features) :**

## References

“Learning to Find Pre-Images”, G BakIr et al, 2004.

**`set_params(**params)`**

Set the parameters of the 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)**  
Transform X.

**Parameters X:** array-like, shape (n\_samples, n\_features) :

**Returns X\_new:** array-like, shape (n\_samples, n\_components) :

## sklearn.decomposition.FactorAnalysis

```
class sklearn.decomposition.FactorAnalysis(n_components=None, tol=0.01,
                                           copy=True, max_iter=1000, verbose=0,
                                           noise_variance_init=None)
```

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.

`verbose` : int | bool

Print verbose output.

`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)

### See Also:

**PCA**Principal component analysis, a simliar non-probabilistic model model that can be computed in closed form.

**ProbabilisticPCA**probabilistic PCA.

**FastICA**Independent component analysis, a latent variable model with non-Gaussian latent variables.

## References

### Attributes

<code>components_</code>	array, [n_components, n_features]	Components with maximum variance.
<code>loglike_</code>	list, [n_iterations]	The log likelihood at each iteration.
<code>noise_variance_</code>	array, shape=(n_features,)	The estimated noise variance for each feature.

### Methods

<code>fit(X[, y])</code>	Fit the FactorAnalysis model to X using EM
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_covariance()</code>	Compute data covariance with the FactorAnalysis model.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>score(X)</code>	Compute score of X under FactorAnalysis model.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Apply dimensionality reduction to X using the model.

**`__init__(n_components=None, tol=0.01, copy=True, max_iter=1000, verbose=0, noise_variance_init=None)`**

**`fit(X, y=None)`**

Fit the FactorAnalysis model to X using EM

**Parameters** `X` : array-like, shape (n\_samples, n\_features)

Training data.

**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.

**Returns** `X_new` : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**`get_covariance()`**

Compute data covariance with the FactorAnalysis model.

`cov = components_.T * components_ + diag(noise_variance)`

**Returns** `cov` : array, shape=(n\_features, n\_features)

Estimated covariance of data.

**`get_params(deep=True)`**

Get parameters for the estimator

**Parameters** `deep: boolean, optional` :

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

**score (X)**

Compute score of X under FactorAnalysis model.

**Parameters X: array of shape(n\_samples, n\_features) :**

The data to test

**Returns ll: array of shape (n\_samples), :**

log-likelihood of each row of X under the current model

**set\_params (\*\*params)**

Set the parameters of the 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)**

Apply dimensionality reduction to X using the model.

Compute the expected mean of the latent variables. See Barber, 21.2.33 (or Bishop, 12.66).

**Parameters X : array-like, shape (n\_samples, n\_features)**

Training data.

**Returns X\_new : array-like, shape (n\_samples, n\_components)**

The latent variables of X.

## sklearn.decomposition.FastICA

```
class sklearn.decomposition.FastICA(n_components=None, algorithm='parallel', whiten=True,
                                    fun='logcosh', fun_prime=''', fun_args=None,
                                    max_iter=200, tol=0.0001, w_init=None, random_state=None)
```

FastICA; a fast algorithm for Independent Component Analysis

**Parameters n\_components : int, optional**

Number of components to use. If none is passed, all are used.

**algorithm : { 'parallel', 'deflation' }**

Apply parallel or deflational algorithm for FastICA

**whiten : boolean, optional**

If whiten is false, the data is already considered to be whitened, and no whitening is performed.

**fun : string or function, optional. Default: 'logcosh'**

The functional form of the G function used in the approximation to neg-entropy. Could be either 'logcosh', 'exp', or 'cube'. You can also provide your own function. It should return a tuple containing the value of the function, and of its derivative, in the point. Example:

```
def my_g(x):return x ** 3, 3 * x ** 2
```

Supplying the derivative through the *fun\_prime* attribute is still supported, but deprecated.

**fun\_prime** : empty string ('') or function, optional, deprecated.

See *fun*.

**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 or 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.

## Notes

Implementation based on *A. Hyvärinen and E. Oja, Independent Component Analysis: Algorithms and Applications, Neural Networks, 13(4-5), 2000, pp. 411-430*

## Attributes

<i>components_</i>	2D array, [n_components, n_features]	The unmixing matrix
<i>sources_</i> : 2D array, [n_samples, n_components]		The estimated latent sources of the data.

## Methods

<b>fit(X[, y])</b>	
<b>fit_transform(X[, y])</b>	Fit to data, then transform it
<b>get_mixing_matrix()</b>	Compute the mixing matrix
<b>get_params([deep])</b>	Get parameters for the estimator
<b>set_params(**params)</b>	Set the parameters of the estimator.
<b>transform(X[, y])</b>	Apply un-mixing matrix “W” to X to recover the sources

**\_\_init\_\_(n\_components=None, algorithm='parallel', whiten=True, fun='logcosh', fun\_prime='', fun\_args=None, max\_iter=200, tol=0.0001, w\_init=None, random\_state=None)**

**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\_mixing\_matrix()**  
 Compute the mixing matrix

**get\_params(deep=True)**  
 Get parameters for the estimator

**Parameters deep: boolean, optional :**  
 If True, will return the parameters for this estimator and contained subobjects that are estimators.

**set\_params(\*\*params)**  
 Set the parameters of the 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, y=None)**  
 Apply un-mixing matrix “W” to X to recover the sources  
 $S = X * W.T$

**unmixing\_matrix\_**  
 DEPRECATED: Renamed to `components_`. This will be removed in 0.14.

## 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)

**Parameters n\_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)
'srandom': 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 mantain, 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.
```

## Notes

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://scgroup.hpclab.ceid.upatras.gr/faculty/stratis/Papers/HPCLAB020107.pdf>

## 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...
```

## Attributes

<i>compo-</i> <i>nents_</i>	array, [n_components, n_features]	Non-negative components of the data
<i>recon-</i> <i>struc-</i> <i>tion_err_</i>	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 Not computed for sparse input matrices because it is too expensive in terms of memory.

## Methods

<b>fit(X[, y])</b>	Learn a NMF model for the data X.
<b>fit_transform(X[, y])</b>	Learn a NMF model for the data X and returns the transformed data.
<b>get_params([deep])</b>	Get parameters for the estimator
<b>set_params(**params)</b>	Set the parameters of the estimator.
<b>transform(X)</b>	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.

**Parameters X: {array-like, sparse matrix}, shape = [n\_samples, n\_features] :**

Data matrix to be decomposed

**Returns self :**

**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.

**Parameters X: {array-like, sparse matrix}, shape = [n\_samples, n\_features] :**

Data matrix to be decomposed

**Returns data: array, [n\_samples, n\_components] :**

Transformed data

**get\_params**(*deep=True*)

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**set\_params**(\**params*)

Set the parameters of the 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*)Transform the data *X* according to the fitted NMF model**Parameters X: {array-like, sparse matrix}, shape = [n\_samples, n\_features] :**

Data matrix to be transformed by the model

**Returns data: array, [n\_samples, n\_components] :**

Transformed data

## 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.

**See Also:**

`PCA`, `MiniBatchSparsePCA`, `DictionaryLearning`

**Attributes**

<code>components_</code>	array, [n_components, n_features]	Sparse components extracted from the data.
<code>error_</code>	array	Vector of errors at each iteration.

**Methods**

<code>fit(X[, y])</code>	Fit the model from data in X.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, ridge_alpha])</code>	Least Squares projection of the data onto the sparse components.

`__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)`

**fit** (*X*, *y*=None)  
     Fit the model from data in X.

**Parameters X: array-like, shape (n\_samples, n\_features) :**

Training vector, where n\_samples in the number of samples and n\_features is the number of features.

**Returns self** : object

    Returns the instance itself.

**fit\_transform** (*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 the estimator

**Parameters deep: boolean, optional :**

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

**set\_params** (\*\*params)

Set the parameters of the 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, 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.

**Parameters X: array of shape (n\_samples, n\_features) :**

Test data to be transformed, must have the same number of features as the data used to train the model.

**ridge\_alpha: float, default: 0.01 :**

Amount of ridge shrinkage to apply in order to improve conditioning.

**Returns X\_new array, shape (n\_samples, n\_components) :**

Transformed data.

## sklearn.decomposition.MiniBatchSparsePCA

```
class sklearn.decomposition.MiniBatchSparsePCA(n_components=None, alpha=1,
                                              ridge_alpha=0.01, n_iter=100, call-
                                              back=None, batch_size=3, verbose=False,
                                              shuffle=True, n_jobs=1, method='lars',
                                              random_state=None, chunk_size=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.

**Parameters n\_components : int,**

number of sparse atoms to extract

**alpha** : int,

Sparsity controlling parameter. Higher values lead to sparser components.

**ridge\_alpha** : float,

Amount of ridge shrinkage to apply in order to improve conditioning when calling the transform method.

**n\_iter** : int,

number of iterations to perform for each mini batch

**callback** : callable,

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.

## See Also:

[PCA](#), [SparsePCA](#), [DictionaryLearning](#)

## Attributes

<i>components_</i>	array, [n_components, n_features]	Sparse components extracted from the data.
<i>error_</i>	array	Vector of errors at each iteration.

## Methods

<code>fit(X[, y])</code>	Fit the model from data in X.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, ridge_alpha])</code>	Least Squares projection of the data onto the sparse components.

```
__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,
chunk_size=None)
```

**fit(X, y=None)**

Fit the model from data in X.

**Parameters X: array-like, shape (n\_samples, n\_features) :**

Training vector, where n\_samples in the number of samples and n\_features is the number of features.

**Returns self : object**

Returns the instance itself.

```
fit_transform(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 the estimator

**Parameters deep: boolean, optional :**

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

```
set_params(**params)
```

Set the parameters of the 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, 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.

**Parameters X: array of shape (n\_samples, n\_features) :**

Test data to be transformed, must have the same number of features as the data used to train the model.

**ridge\_alpha: float, default: 0.01 :**

Amount of ridge shrinkage to apply in order to improve conditioning.

**Returns X\_new array, shape (n\_samples, n\_components) :**

Transformed data.

## 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

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
```

**Parameters** **dictionary** : array, [n\_components, n\_features]

The dictionary atoms used for sparse coding. Lines are assumed to be normalized to unit norm.

**transform\_algorithm** : {‘lasso\_lars’, ‘lasso\_cd’, ‘lars’, ‘omp’, ‘threshold’}

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

**See Also:**

DictionaryLearning,	MiniBatchDictionaryLearning,	SparsePCA,
MiniBatchSparsePCA, sparse_encode		

## Attributes

<code>components_</code>	array, [n_components, n_features]	The unchanged dictionary atoms
--------------------------	-----------------------------------	--------------------------------

## Methods

<code>fit(X[, y])</code>	Do nothing and return the estimator unchanged
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, y])</code>	Encode the data as a sparse combination of the dictionary atoms.

`__init__(dictionary, transform_algorithm='omp', transform_n_nonzero_coefs=None, transform_alpha=None, split_sign=False, n_jobs=1)`

`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.

**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 the estimator

**Parameters** `deep: boolean, optional` :

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

`set_params(**params)`

Set the parameters of the 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, y=None)`

Encode the data as a sparse combination of the dictionary atoms.

Coding method is determined by the object parameter `transform_algorithm`.

**Parameters** `X` : array of shape (n\_samples, n\_features)

Test data to be transformed, must have the same number of features as the data used to train the model.

**Returns X\_new** : array, shape (n\_samples, n\_components)

Transformed data

## sklearn.decomposition.DictionaryLearning

```
class sklearn.decomposition.DictionaryLearning(n_components=None, alpha=1,
                                              max_iter=1000, tol=1e-08,
                                              fit_algorithm='lars', trans-
                                              form_algorithm='omp', trans-
                                              form_n_nonzero_coefs=None, trans-
                                              form_alpha=None, n_jobs=1,
                                              code_init=None, dict_init=None, ver-
                                              bose=False, split_sign=False, ran-
                                              dom_state=None, n_atoms=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:

$$(U^*, V^*) = \underset{(U, V)}{\operatorname{argmin}} \frac{1}{2} \|Y - U V\|_F^2 + \alpha \|U\|_1 \\ \text{with } \|V_k\|_2 = 1 \text{ for all } 0 \leq k < n_{\text{components}}$$

**Parameters** `n_components` : int,

number of dictionary elements to extract

`alpha` : int,

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.

## 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>)

## Attributes

<i>components_</i>	array, [n_components, n_features]	dictionary atoms extracted from the data
<i>error_</i>	array	vector of errors at each iteration

## Methods

---

`fit(X[, y])` Fit the model from data in X.

---

`fit_transform(X[, y])` Fit to data, then transform it

Continued on next page

**Table 1.59 – continued from previous page**

<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, y])</code>	Encode the data as a sparse combination of the dictionary atoms.

**`__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, n_atoms=None)`**

**`fit(X, y=None)`**

Fit the model from data in X.

**Parameters X: array-like, shape (n\_samples, n\_features) :**

Training vector, where n\_samples in the number of samples and n\_features is the number of features.

**Returns self: object :**

Returns the 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.

**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 the estimator

**Parameters deep: boolean, optional :**

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

**`set_params(**params)`**

Set the parameters of the 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, y=None)`**

Encode the data as a sparse combination of the dictionary atoms.

Coding method is determined by the object parameter *transform\_algorithm*.

**Parameters X : array of shape (n\_samples, n\_features)**

Test data to be transformed, must have the same number of features as the data used to train the model.

**Returns** `X_new` : array, shape (n\_samples, n\_components)

Transformed data

## sklearn.decomposition.MiniBatchDictionaryLearning

```
class sklearn.decomposition.MiniBatchDictionaryLearning(n_components=None,      al-
                                                       pha=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,
                                                       n_atoms=None,
                                                       chunk_size=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:

$$(U^*, V^*) = \underset{(U, V)}{\operatorname{argmin}} 0.5 \| Y - U V \|_2^2 + \alpha * \| U \|_1$$

with  $\| V_k \|_2 = 1$  for all  $0 \leq k < n_{\text{components}}$

**Parameters** `n_components` : int,

number of dictionary elements to extract

`alpha` : int,

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,  $0.1 * n_{\text{features}}$  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.

## 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>)

## Attributes

<i>components_</i>	array, [n_components, n_features]	components extracted from the data
--------------------	-----------------------------------	------------------------------------

## Methods

<code>fit(X[, y])</code>	Fit the model from data in X.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator

Continued on next page

**Table 1.60 – continued from previous page**

<code>partial_fit(X[, y, iter_offset])</code>	Updates the model using the data in X as a mini-batch.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, y])</code>	Encode the data as a sparse combination of the dictionary atoms.

**`__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, n_atoms=None, chunk_size=None)`**

**`fit(X, y=None)`**

Fit the model from data in X.

**Parameters X: array-like, shape (n\_samples, n\_features) :**

Training vector, where n\_samples in the number of samples and n\_features is the number of features.

**Returns self : object**

Returns the instance itself.

**`fit_transform(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 the estimator

**Parameters deep: boolean, optional :**

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

**`partial_fit(X, y=None, iter_offset=0)`**

Updates the model using the data in X as a mini-batch.

**Parameters X: array-like, shape (n\_samples, n\_features) :**

Training vector, where n\_samples in the number of samples and n\_features is the number of features.

**Returns self : object**

Returns the instance itself.

**`set_params(**params)`**

Set the parameters of the 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, y=None)

Encode the data as a sparse combination of the dictionary atoms.

Coding method is determined by the object parameter *transform\_algorithm*.

**Parameters X** : array of shape (n\_samples, n\_features)

Test data to be transformed, must have the same number of features as the data used to train the model.

**Returns X\_new** : array, shape (n\_samples, n\_components)

Transformed data

<code>decomposition.fastica(X[, n_components, ...])</code>	Perform Fast Independent Component Analysis.
<code>decomposition.dict_learning(X, n_components, ...)</code>	Solves a dictionary learning matrix factorization problem.
<code>decomposition.dict_learning_online(X, ...)</code>	Solves a dictionary learning matrix factorization problem online.
<code>decomposition.sparse_encode(X, dictionary[, ...])</code>	Sparse coding

## sklearn.decomposition.fastica

```
sklearn.decomposition.fastica(X, n_components=None, algorithm='parallel', whiten=True,
                           fun='logcosh', fun_prime='', fun_args={}, max_iter=200,
                           tol=0.0001, w_init=None, random_state=None)
```

Perform Fast Independent Component Analysis.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training vector, where n\_samples is the number of samples and n\_features is the number of features.

**n\_components** : int, optional

Number of components to extract. If None no dimension reduction is performed.

**algorithm** : {‘parallel’, ‘deflation’}, optional

Apply a parallel or deflational FASTICA algorithm.

**whiten: boolean, optional :**

If True perform an initial whitening of the data. If False, the data is assumed to have already been preprocessed: it should be centered, normed and white. Otherwise you will get incorrect results. In this case the parameter n\_components will be ignored.

**fun** : string or function, optional. Default: ‘logcosh’

The functional form of the G function used in the approximation to neg-entropy. Could be either ‘logcosh’, ‘exp’, or ‘cube’. You can also provide your own function. It should return a tuple containing the value of the function, and of its derivative, in the point. Example:

```
def my_g(x):return x ** 3, 3 * x ** 2
```

Supplying the derivative through the *fun\_prime* attribute is still supported, but deprecated.

**fun\_prime** : empty string (‘’) or function, optional, deprecated.

See *fun*.

**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 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

**source\_only: boolean, optional :**

If True, only the sources matrix is returned.

**random\_state: int or RandomState :**

Pseudo number generator state used for random sampling.

**Returns K: (n\_components, p) array or None. :**

If whiten is ‘True’, K is the pre-whitening matrix that projects data onto the first n.comp principal components. If whiten is ‘False’, K is ‘None’.

**W: (n\_components, n\_components) array :**

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: (n\_components, n) array :**

estimated source matrix

## 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 ‘ $S = WKX$ .‘

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*

## 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, n_atoms=None)
```

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.

**Returns code: array of shape (n\_samples, n\_components) :**

The sparse code factor in the matrix factorization.

**dictionary: array of shape (n\_components, n\_features), :**

The dictionary factor in the matrix factorization.

**errors: array :**

Vector of errors at each iteration.

**See Also:**

`dict_learning_online`, `DictionaryLearning`, `MiniBatchDictionaryLearning`,  
`SparsePCA`, `MiniBatchSparsePCA`

**`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, call-
                                             back=None, batch_size=3, verbose=False,
                                             shuffle=True, n_jobs=1, method='lars',
                                             iter_offset=0, random_state=None,
                                             n_atoms=None, chunk_size=None)
```

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:

$$(U^*, V^*) = \underset{(U, V)}{\operatorname{argmin}} \frac{1}{2} \|X - U V\|_F^2 + \alpha \|U\|_1 \\ \text{with } \|V_k\|_F = 1 \text{ for all } 0 \leq k < n_{\text{components}}$$

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.

**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.

**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.

**Returns** **code** : array of shape (n\_samples, n\_components),

the sparse code (only returned if *return\_code=True*)

**dictionary** : array of shape (n\_components, n\_features),

the solutions to the dictionary learning problem

#### See Also:

`dict_learning`, `DictionaryLearning`, `MiniBatchDictionaryLearning`, `SparsePCA`, `MiniBatchSparsePCA`

## sklearn.decomposition.sparse\_encode

```
sklearn.decomposition.sparse_encode(X, dictionary, gram=None, cov=None, algorithm='lasso_lars', n_nonzero_coefs=None, alphas=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
```

**Parameters** **X**: array of shape (n\_samples, n\_features) :

Data matrix

**dictionary**: array of shape (n\_components, n\_features) :

The dictionary matrix against which to solve the sparse coding of the data. Some of the algorithms assume normalized rows for meaningful output.

**gram**: array, shape=(n\_components, n\_components) :

Precomputed Gram matrix, `dictionary` \* `dictionary`’

**cov**: array, shape=(n\_components, n\_samples) :

Precomputed covariance, `dictionary`’ \* X

**algorithm**: {‘lasso\_lars’, ‘lasso\_cd’, ‘lars’, ‘omp’, ‘threshold’} :

lars: uses the least angle regression method (linear\_model.lars\_path)  
 lasso\_lars: uses Lars to compute the Lasso solution  
 lasso\_cd: uses the coordinate descent method to

compute the Lasso solution (`linear_model.Lasso`). `lasso_lars` will be faster if the estimated components are sparse. `omp`: uses orthogonal matching pursuit to estimate the sparse solution threshold: squashes to zero all coefficients less than `alpha` from the projection dictionary  $\mathbf{X}'$

**n\_nonzero\_coefs: int, 0.1 \* n\_features by default :**

Number of nonzero coefficients to target in each column of the solution. This is only used by `algorithm='lars'` and `algorithm='omp'` and is overridden by `alpha` in the `omp` case.

**alpha: float, 1. by default :**

If `algorithm='lasso_lars'` or `algorithm='lasso_cd'`, `alpha` is the penalty applied to the L1 norm. If `algorithm='threshold'`, `alpha` is the absolute value of the threshold below which coefficients will be squashed to zero. If `algorithm='omp'`, `alpha` is the tolerance parameter: the value of the reconstruction error targeted. In this case, it overrides `n_nonzero_coefs`.

**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.

**Returns code: array of shape (n\_samples, n\_components) :**

The sparse codes

**See Also:**

`sklearn.linear_model.lars_path`, `sklearn.linear_model.orthogonal_mp`,  
`sklearn.linear_model.Lasso`, `SparseCoder`

## 1.8.6 `sklearn.dummy`: Dummy estimators

**User guide:** See the *Model evaluation* section for further details.

<code>dummy.DummyClassifier([strategy, random_state])</code>	DummyClassifier is a classifier that makes predictions using simple rules.
<code>dummy.DummyRegressor</code>	DummyRegressor is a regressor that always predicts the mean of the train

### `sklearn.dummy.DummyClassifier`

**class** `sklearn.dummy.DummyClassifier(strategy='stratified', random_state=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.

**Parameters strategy: 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.

**random\_state: int seed, RandomState instance, or None (default) :**

The seed of the pseudo random number generator to use.

### Attributes

<i>classes_</i>	array or list of array of shape = [n_classes]	Class labels for each output.
<i>n_classes_</i>	array or list of array of shape = [n_classes]	Number of label for each output.
<i>class_prior_</i>	array or list of array of shape = [n_classes]	Probability of each class for each output.
<i>n_outputs_</i>	int,	Number of outputs.
<i>outputs_2d_</i>	bool,	True if the output at fit is 2d, else false.

### Methods

<code>fit(X, y)</code>	Fit the random classifier.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Perform classification on test vectors X.
<code>predict_log_proba(X)</code>	Return log probability estimates for the test vectors X.
<code>predict_proba(X)</code>	Return probability estimates for the test vectors X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

**`__init__(strategy='stratified', random_state=None)`**

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

Fit the random classifier.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features.

`y` : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]

Target values.

**Returns self** : object

Returns self.

**`get_params(deep=True)`**

Get parameters for the estimator

**Parameters** `deep: boolean, optional` :

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

**`predict(X)`**

Perform classification on test vectors X.

**Parameters X** : {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.

**Returns y** : array, shape = [n\_samples] or [n\_samples, n\_outputs]

Predicted target values for X.

**predict\_log\_proba (X)**

Return log probability estimates for the test vectors X.

**Parameters X** : {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.

**Returns P** : array-like or list of array-like of shape = [n\_samples, n\_classes]

Returns the log probability of the sample for each class in the model, where classes are ordered arithmetically for each output.

**predict\_proba (X)**

Return probability estimates for the test vectors X.

**Parameters X** : {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.

**Returns P** : array-like or list of array-like 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)**

Returns the mean accuracy on the given test data and labels.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns z** : float

**set\_params (\*\*params)**

Set the parameters of the 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.dummy.DummyRegressor

**class sklearn.dummy.DummyRegressor**

DummyRegressor is a regressor that always predicts the mean of the training targets.

This regressor is useful as a simple baseline to compare with other (real) regressors. Do not use it for real problems.

## Attributes

<code>y_mean_</code>	float or array of shape [n_outputs]	Mean of the training targets.
<code>n_outputs_</code>	int,	Number of outputs.
<code>outputs_2d_</code>	bool,	True if the output at fit is 2d, else false.

## Methods

<code>fit(X, y)</code>	Fit the random regressor.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Perform classification on test vectors X.
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

### `__init__()`

`x.__init__(...)` initializes x; see `help(type(x))` for signature

### `fit(X, y)`

Fit the random regressor.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features.

`y` : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]

Target values.

**Returns self** : object

Returns self.

### `get_params(deep=True)`

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

### `predict(X)`

Perform classification on test vectors X.

**Parameters** `X` : {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.

**Returns y** : array, shape = [n\_samples] or [n\_samples, n\_outputs]

Predicted target values for X.

### `score(X, y)`

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  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns** **z** : float

**set\_params** (\*\*params)

Set the parameters of the 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** :

---

## 1.8.7 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.

<code>ensemble.RandomForestClassifier(...)</code>	A random forest classifier.
<code>ensemble.RandomTreesEmbedding(...)</code>	An ensemble of totally random trees.
<code>ensemble.RandomForestRegressor(...)</code>	A random forest regressor.
<code>ensemble.ExtraTreesClassifier(...)</code>	An extra-trees classifier.
<code>ensemble.ExtraTreesRegressor([n_estimators, ...])</code>	An extra-trees regressor.
<code>ensemble.AdaBoostClassifier(...[, criterion])</code>	An AdaBoost classifier.
<code>ensemble.AdaBoostRegressor(...[, criterion, ...])</code>	An AdaBoost regressor.
<code>ensemble.GradientBoostingClassifier([loss, ...])</code>	Gradient Boosting for classification.
<code>ensemble.GradientBoostingRegressor([loss, ...])</code>	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_density=0.1,
                                             max_features='auto', bootstrap=True, compute_importances=False, oob_score=False,
                                             n_jobs=1, random_state=None, verbose=0)
```

A random forest classifier.

A random forest is a meta estimator that fits a number of classifical decision trees 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, string or None, optional (default="auto")

**The number of features to consider when looking for the best split:**

- If “auto”, then  $\max\_features=\sqrt{n\_features}$  on classification tasks and  $\max\_features=n\_features$  on regression problems.
- If “sqrt”, then  $\max\_features=\sqrt{n\_features}$ .
- If “log2”, then  $\max\_features=\log_2(n\_features)$ .
- If None, then  $\max\_features=n\_features$ .

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. 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 than `min_samples_leaf` samples. Note: this parameter is tree-specific.

**min\_density** : float, optional (default=0.1)

This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the `sample_mask` (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If `min_density` equals to one, the partitions are always represented as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks). Note: this parameter is tree-specific.

**bootstrap** : boolean, optional (default=True)

Whether bootstrap samples are used when building trees.

**compute\_importances** : boolean, optional (default=True)

Whether feature importances are computed and stored into the `feature_importances_` attribute when calling `fit`.

**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. 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.

**See Also:**

`DecisionTreeClassifier`, `ExtraTreesClassifier`

**References**

[R82]

**Attributes**

<code>estimators_</code> : list of <code>DecisionTreeClassifier</code>		The collection of fitted sub-estimators.
<code>classes_</code> : 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).
<code>n_classes_</code> : int or list		The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).
<code>feature_importances_</code>	array of shape = [n_features]	The feature importances (the higher, the more important the feature).
<code>oob_score_</code>	float	Score of the training dataset obtained using an out-of-bag estimate.
<code>oob_decision_function_</code>	array of shape = [n_samples, n_classes]	Decision function computed with out-of-bag estimate on the training set.

**Methods**

<code>apply(X)</code>	Apply trees in the forest to X, return leaf indices.
<code>fit(X, y[, sample_weight])</code>	Build a forest of trees from the training set (X, y).
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class for X.
<code>predict_log_proba(X)</code>	Predict class log-probabilities for X.
<code>predict_proba(X)</code>	Predict class probabilities for X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

`__init__(n_estimators=10, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_density=0.1, max_features='auto', bootstrap=True, compute_importances=False, oob_score=False, n_jobs=1, random_state=None, verbose=0)`

`apply(X)`

Apply trees in the forest to X, return leaf indices.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Input data.

**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 of shape = [n\_samples, n\_features]

The training input samples.

$y$  : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]

The target values (integers that correspond to classes 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**( $X, y=None, \text{**fit\_params}$ )

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**  $X_{\text{new}}$  : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params**(deep=True)

Get parameters for the estimator

**Parameters** **deep:** boolean, optional :

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

**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.

**Parameters**  $X$  : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns**  $y$  : 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 mean predicted class log-probabilities of the trees in the forest.

**Parameters** **X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns** **p** : 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. Classes are ordered by arithmetical order.

**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.

**Parameters** **X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**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. Classes are ordered by arithmetical order.

**score** (*X*, *y*)

Returns the mean accuracy on the given test data and labels.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for *X*.

**Returns** **z** : float

**set\_params** (\*\**params*)

Set the parameters of the 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.

**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.

**sklearn.ensemble.RandomTreesEmbedding**

```
class sklearn.ensemble.RandomTreesEmbedding(n_estimators=10,           max_depth=5,
                                             min_samples_split=2,    min_samples_leaf=1,
                                             min_density=0.1,       n_jobs=1,        ran-
                                             dom_state=None, verbose=0)
```

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 trees in the forest.

The dimensionality of the resulting representation is approximately `n_estimators * 2 ** max_depth`.

**Parameters**

**n\_estimators** : int

Number of trees in the forest.

**max\_depth** : int

Maximum depth of each tree.

**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 than `min_samples_leaf` samples. Note: this parameter is tree-specific.

**min\_density** : float, optional (default=0.1)

This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the `sample_mask` (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If `min_density` equals to one, the partitions are always represented as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks).

**n\_jobs** : integer, optional (default=1)

The number of jobs to run in parallel. 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.

**References**

[R84], [R85]

## Attributes

<i>estimators_</i> : list of DecisionTreeClassifier	The collection of fitted sub-estimators.
---	--

## Methods

<code>apply(X)</code>	Apply trees in the forest to X, return leaf indices.
<code>fit(X, y)</code>	Fit estimator.
<code>fit_transform(X[, y])</code>	Fit estimator and transform dataset.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Transform dataset.

`__init__(n_estimators=10, max_depth=5, min_samples_split=2, min_samples_leaf=1, min_density=0.1, n_jobs=1, random_state=None, verbose=0)`

**apply (X)**

Apply trees in the forest to X, return leaf indices.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Input data.

**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=None)**

Fit estimator.

**Parameters** **X** : array-like, shape=(n\_samples, n\_features)

Input data used to build forests.

**fit\_transform (X, y=None)**

Fit estimator and transform dataset.

**Parameters** **X** : array-like, shape=(n\_samples, n\_features)

Input data used to build forests.

**Returns** **X\_transformed: sparse matrix, shape=(n\_samples, n\_out) :**

Transformed dataset.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters** **deep: boolean, optional :**

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

**set\_params (\*\*params)**

Set the parameters of the 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)**

Transform dataset.

**Parameters X** : array-like, shape=(n\_samples, n\_features)

Input data to be transformed.

**Returns X\_transformed: sparse matrix, shape=(n\_samples, n\_out) :**

Transformed dataset.

## sklearn.ensemble.RandomForestRegressor

```
class sklearn.ensemble.RandomForestRegressor(n_estimators=10, criterion='mse',
                                             max_depth=None, min_samples_split=2,
                                             min_samples_leaf=1, min_density=0.1,
                                             max_features='auto', bootstrap=True, compute_importances=False, oob_score=False,
                                             n_jobs=1, random_state=None, verbose=0)
```

A random forest regressor.

A random forest is a meta estimator that fits a number of classifical decision trees 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="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, string or None, optional (default="auto")

**The number of features to consider when looking for the best split:**

- If "auto", then  $\text{max\_features}=\sqrt{n\_features}$  on classification tasks and  $\text{max\_features}=n\_features$  on regression problems.
- If "sqrt", then  $\text{max\_features}=\sqrt{n\_features}$ .
- If "log2", then  $\text{max\_features}=\log_2(n\_features)$ .
- If None, then  $\text{max\_features}=n\_features$ .

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. 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 than `min_samples_leaf` samples.  
Note: this parameter is tree-specific.

**min\_density** : float, optional (default=0.1)

This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the `sample_mask` (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If `min_density` equals to one, the partitions are always represented as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks). Note: this parameter is tree-specific.

**bootstrap** : boolean, optional (default=True)

Whether bootstrap samples are used when building trees.

**compute\_importances** : boolean, optional (default=True)

Whether feature importances are computed and stored into the `feature_importances_` attribute when calling `fit`.

**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. 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.

## See Also:

`DecisionTreeRegressor`, `ExtraTreesRegressor`

## References

[R83]

## Attributes

<code>estimators_</code> : list of <code>DecisionTreeRegressor</code>		The collection of fitted sub-estimators.
<code>feature_importances_</code>	array of shape = [n_features]	The feature importances (the higher, the more important the feature).
<code>oob_score_</code>	float	Score of the training dataset obtained using an out-of-bag estimate.
<code>oob_prediction_</code>	array of shape = [n_samples]	Prediction computed with out-of-bag estimate on the training set.

## Methods

<code>apply(X)</code>	Apply trees in the forest to X, return leaf indices.
<code>fit(X, y[, sample_weight])</code>	Build a forest of trees from the training set (X, y).
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict regression target for X.
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

`__init__(n_estimators=10, criterion='mse', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_density=0.1, max_features='auto', bootstrap=True, compute_importances=False, oob_score=False, n_jobs=1, random_state=None, verbose=0)`

### `apply(X)`

Apply trees in the forest to X, return leaf indices.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Input data.

**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 of shape = [n\_samples, n\_features]

The training input samples.

`y` : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]

The target values (integers that correspond to classes 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.

**Returns X\_new** : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**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.

**Parameters X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns y: array of shape = [n\_samples] or [n\_samples, n\_outputs] :**

The predicted values.

**score (X, y)**

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  $((y - y_{\text{pred}})^2 \cdot \text{sum}()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2 \cdot \text{sum}()$ . Best possible score is 1.0, lower values are worse.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params (\*\*params)**

Set the parameters of the 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.

**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.

## sklearn.ensemble.ExtraTreesClassifier

```
class sklearn.ensemble.ExtraTreesClassifier(n_estimators=10, criterion='gini',
                                            max_depth=None, min_samples_split=2,
                                            min_samples_leaf=1, min_density=0.1,
                                            max_features='auto', bootstrap=False, compute_importances=False, oob_score=False,
                                            n_jobs=1, random_state=None, verbose=0)
```

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.

**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, string or None, optional (default="auto")

**The number of features to consider when looking for the best split.**

- If “auto”, then  $\text{max\_features}=\sqrt{n\text{\_features}}$  on classification tasks and  $\text{max\_features}=n\text{\_features}$  on regression problems.
- If “sqrt”, then  $\text{max\_features}=\sqrt{n\text{\_features}}$ .
- If “log2”, then  $\text{max\_features}=\log_2(n\text{\_features})$ .
- If None, then  $\text{max\_features}=n\text{\_features}$ .

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. 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 than **min\_samples\_leaf** samples. Note: this parameter is tree-specific.

**min\_density** : float, optional (default=0.1)

This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the *sample\_mask* (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If **min\_density** equals to one, the partitions are always represented

as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks). Note: this parameter is tree-specific.

**bootstrap** : boolean, optional (default=False)

Whether bootstrap samples are used when building trees.

**compute\_importances** : boolean, optional (default=True)

Whether feature importances are computed and stored into the `feature_importances_` attribute when calling `fit`.

**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. 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.

## See Also:

`sklearn.tree.ExtraTreeClassifier`Base classifier for this ensemble.

`RandomForestClassifier`Ensemble Classifier based on trees with optimal splits.

## References

[R80]

## Attributes

<code>estimators_</code> : list of DecisionTreeClassifier		The collection of fitted sub-estimators.
<code>classes_</code> : 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).
<code>n_classes_</code> : int or list		The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).
<code>feature_importances_</code>	array of shape = [n_features]	The feature importances (the higher, the more important the feature).
<code>oob_score_</code>	float	Score of the training dataset obtained using an out-of-bag estimate.
<code>oob_decision_function_</code>	array of shape = [n_samples, n_classes]	Decision function computed with out-of-bag estimate on the training set.

## Methods

<code>apply(X)</code>	Apply trees in the forest to X, return leaf indices.
<code>fit(X, y[, sample_weight])</code>	Build a forest of trees from the training set (X, y).
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class for X.
<code>predict_log_proba(X)</code>	Predict class log-probabilities for X.
<code>predict_proba(X)</code>	Predict class probabilities for X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

`__init__(n_estimators=10, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_density=0.1, max_features='auto', bootstrap=False, compute_importances=False, oob_score=False, n_jobs=1, random_state=None, verbose=0)`

### `apply(X)`

Apply trees in the forest to X, return leaf indices.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Input data.

**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 of shape = [n\_samples, n\_features]

The training input samples.

`y` : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]

The target values (integers that correspond to classes 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.

**Returns X\_new** : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**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.

**Parameters X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns y** : 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 mean predicted class log-probabilities of the trees in the forest.

**Parameters X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns p** : 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. Classes are ordered by arithmetical order.

**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.

**Parameters X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**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. Classes are ordered by arithmetical order.

**score (X, y)**

Returns the mean accuracy on the given test data and labels.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns z :** float

**set\_params (\*\*params)**

Set the parameters of the 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.

**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.

## sklearn.ensemble.ExtraTreesRegressor

```
class sklearn.ensemble.ExtraTreesRegressor(n_estimators=10, criterion='mse',
                                           max_depth=None, min_samples_split=2,
                                           min_samples_leaf=1, min_density=0.1,
                                           max_features='auto', bootstrap=False, compute_importances=False, oob_score=False,
                                           n_jobs=1, random_state=None, verbose=0)
```

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.

**Parameters n\_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, string or None, optional (default="auto")

**The number of features to consider when looking for the best split:**

- If “auto”, then  $\text{max\_features}=\sqrt{n\text{\_features}}$  on classification tasks and  $\text{max\_features}=n\text{\_features}$  on regression problems.
- If “sqrt”, then  $\text{max\_features}=\sqrt{n\text{\_features}}$ .
- If “log2”, then  $\text{max\_features}=\log_2(n\text{\_features})$ .
- If None, then  $\text{max\_features}=n\text{\_features}$ .

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. 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 than `min_samples_leaf` samples. Note: this parameter is tree-specific.

**min\_density** : float, optional (default=0.1)

This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the `sample_mask` (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If `min_density` equals to one, the partitions are always represented as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks). 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.

**compute\_importances** : boolean, optional (default=True)

Whether feature importances are computed and stored into the `feature_importances_` attribute when calling `fit`.

**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. 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.

**See Also:**

**`sklearn.tree.ExtraTreeRegressor`** Base estimator for this ensemble.

**`RandomForestRegressor`** Ensemble regressor using trees with optimal splits.

## References

[R81]

## Attributes

<i>estimators_</i> : list of DecisionTreeRegressor		The collection of fitted sub-estimators.
<i>feature_importances_</i>	array of shape = [n_features]	The feature importances (the higher, the more important the feature).
<i>oob_score_</i>	float	Score of the training dataset obtained using an out-of-bag estimate.
<i>oob_prediction_</i>	array of shape = [n_samples]	Prediction computed with out-of-bag estimate on the training set.

## Methods

<code>apply(X)</code>	Apply trees in the forest to X, return leaf indices.
<code>fit(X, y[, sample_weight])</code>	Build a forest of trees from the training set (X, y).
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict regression target for X.
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

**`__init__(n_estimators=10, criterion='mse', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_density=0.1, max_features='auto', bootstrap=False, compute_importances=False, oob_score=False, n_jobs=1, random_state=None, verbose=0)`**

**`apply(X)`**

Apply trees in the forest to X, return leaf indices.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Input data.

**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 of shape = [n\_samples, n\_features]

The training input samples.

`y` : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]

The target values (integers that correspond to classes 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.

**Returns** X\_new : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params**(deep=True)

Get parameters for the estimator

**Parameters** deep: boolean, optional :

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

**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.

**Parameters** X : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns** y: array of shape = [n\_samples] or [n\_samples, n\_outputs] :

The predicted values.

**score**(X, y)

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  $((y - y_{\text{pred}})^2 \cdot \text{sum}()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2 \cdot \text{sum}()$ . Best possible score is 1.0, lower values are worse.

**Parameters** X : array-like, shape = [n\_samples, n\_features]

Training set.

y : array-like, shape = [n\_samples]

**Returns** z : float

**set\_params**(\*\*params)

Set the parameters of the 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.

**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.

## sklearn.ensemble.AdaBoostClassifier

```
AdaBoostClassifier(base_estimator=DecisionTreeClassifier(compute_importances=False, criterion='gini',
max_depth=1, max_features=None, min_density=0.1,
min_samples_leaf=1, min_samples_split=2, random_state=None), n_estimators=50, learning_rate=1.0)
```

An AdaBoost classifier.

An AdaBoost 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].

**Parameters base\_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=0.1)

Learning rate shrinks the contribution of each classifier by `learning_rate`. There is a trade-off between `learning_rate` and `n_estimators`.

**algorithm** : string, 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.

**compute\_importances** : boolean, optional (default=False)

Whether feature importances are computed and stored in the `feature_importances_` attribute when calling `fit`.

#### **See Also:**

`AdaBoostRegressor`, `GradientBoostingClassifier`, `DecisionTreeClassifier`

## References

[R76], [R77]

## Attributes

<code>estimators_</code>	list of classifiers	The collection of fitted sub-estimators.
<code>classes_</code>	array of shape = [n_classes]	The classes labels.
<code>n_classes_</code>	int	The number of classes.
<code>estimator_weights_</code>	list of floats	Weights for each estimator in the boosted ensemble.
<code>errors_</code>	list of floats	Classification error for each estimator in the boosted ensemble.
<code>feature_importances_</code>	array of shape = [n_features]	The feature importances if supported by the <code>base_estimator</code> . Only computed if <code>compute_importances=True</code> .

## Methods

```
decision_function  
fit  
get_params  
predict  
predict_log_proba  
predict_proba  
score  
set_params  
staged_decision_function  
staged_predict  
staged_predict_proba  
staged_score
```

```
AdaBoostClassifier.decision_function(X, n_estimators=-1)
```

Compute the decision function of  $\mathbf{x}$ .

**Parameters X** : array-like of shape = [n samples, n features]

### The input samples.

**n\_estimators** : int, optional (default=-1)

Use only the first `n_estimators` classifiers for the prediction. This is useful for grid searching the `n_estimators` parameter since it is not necessary to fit separately for all choices of `n_estimators`, but only the highest `n_estimators`. Any negative value will result in all estimators being used.

**Returns score** : array, shape = [n\_samples, k]

The decision function of the input samples. Classes are ordered by arithmetical order. 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.

`AdaBoostClassifier.fit(X, y, sample_weight=None)`

Build a boosted classifier from the training set (X, y).

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The training input samples.

`y` : array-like of shape = [n\_samples]

The target values (integers that correspond to classes).

`sample_weight` : array-like of shape = [n\_samples], optional

Sample weights. If None, the sample weights are initialized to 1 / n\_samples.

**Returns** `self` : object

Returns self.

`AdaBoostClassifier.get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep`: boolean, optional :

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

`AdaBoostClassifier.predict(X, n_estimators=-1)`

Predict classes for X.

The predicted class of an input sample is computed as the weighted mean prediction of the classifiers in the ensemble.

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The input samples.

`n_estimators` : int, optional (default=-1)

Use only the first `n_estimators` classifiers for the prediction. This is useful for grid searching the `n_estimators` parameter since it is not necessary to fit separately for all choices of `n_estimators`, but only the highest `n_estimators`. Any negative value will result in all estimators being used.

**Returns** `y` : array of shape = [n\_samples]

The predicted classes.

`AdaBoostClassifier.predict_log_proba(X, n_estimators=-1)`

Predict class log-probabilities for X.

The predicted class log-probabilities of an input sample is computed as the weighted mean predicted class log-probabilities of the classifiers in the ensemble.

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The input samples.

`n_estimators` : int, optional (default=-1)

Use only the first `n_estimators` classifiers for the prediction. This is useful for grid searching the `n_estimators` parameter since it is not necessary to fit separately for all choices of `n_estimators`, but only the highest `n_estimators`. Any negative value will result in all estimators being used.

**Returns p** : array of shape = [n\_samples]

The class log-probabilities of the input samples. Classes are ordered by arithmetical order.

`AdaBoostClassifier.predict_proba(X, n_estimators=-1)`

Predict class probabilities for X.

The predicted class probabilities of an input sample is computed as the weighted mean predicted class probabilities of the classifiers in the ensemble.

**Parameters X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**n\_estimators** : int, optional (default=-1)

Use only the first `n_estimators` classifiers for the prediction. This is useful for grid searching the `n_estimators` parameter since it is not necessary to fit separately for all choices of `n_estimators`, but only the highest `n_estimators`. Any negative value will result in all estimators being used.

**Returns p** : array of shape = [n\_samples]

The class probabilities of the input samples. Classes are ordered by arithmetical order.

`AdaBoostClassifier.score(X, y)`

Returns the mean accuracy on the given test data and labels.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns z** : float

`AdaBoostClassifier.set_params(**params)`

Set the parameters of the 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** :

`AdaBoostClassifier.staged_decision_function(X, n_estimators=-1)`

Compute decision function of X for each boosting iteration.

This method allows monitoring (i.e. determine error on testing set) after each boosting iteration.

**Parameters X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**n\_estimators** : int, optional (default=-1)

Use only the first `n_estimators` classifiers for the prediction. This is useful for grid searching the `n_estimators` parameter since it is not necessary to fit separately for all choices of `n_estimators`, but only the highest `n_estimators`. Any negative value will result in all estimators being used.

**Returns score** : generator of array, shape = [n\_samples, k]

The decision function of the input samples. Classes are ordered by arithmetical order. 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.

`AdaBoostClassifier.staged_predict(X, n_estimators=-1)`

Return staged predictions for X.

The predicted class of an input sample is computed as the weighted mean prediction of the classifiers in the ensemble.

This generator method yields the ensemble prediction after each iteration of boosting and therefore allows monitoring, such as to determine the prediction on a test set after each boost.

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The input samples.

`n_estimators` : int, optional (default=-1)

Use only the first `n_estimators` classifiers for the prediction. This is useful for grid searching the `n_estimators` parameter since it is not necessary to fit separately for all choices of `n_estimators`, but only the highest `n_estimators`. Any negative value will result in all estimators being used.

**Returns** `y` : generator of array, shape = [n\_samples]

The predicted classes.

`AdaBoostClassifier.staged_predict_proba(X, n_estimators=-1)`

Predict class probabilities for X.

The predicted class probabilities of an input sample is computed as the weighted mean predicted class probabilities of the classifiers in the ensemble.

This generator method yields the ensemble predicted class probabilities after each iteration of boosting and therefore allows monitoring, such as to determine the predicted class probabilities on a test set after each boost.

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The input samples.

`n_estimators` : int, optional (default=-1)

Use only the first `n_estimators` classifiers for the prediction. This is useful for grid searching the `n_estimators` parameter since it is not necessary to fit separately for all choices of `n_estimators`, but only the highest `n_estimators`. Any negative value will result in all estimators being used.

**Returns** `p` : generator of array, shape = [n\_samples]

The class probabilities of the input samples. Classes are ordered by arithmetical order.

`AdaBoostClassifier.staged_score(X, y, n_estimators=-1)`

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, shape = [n\_samples, n\_features]

Training set.

`y` : array-like, shape = [n\_samples]

Labels for X.

**Returns** `z` : float

## sklearn.ensemble.AdaBoostRegressor

```
AdaBoostRegressor(base_estimator=DecisionTreeRegressor(compute_importances=False, criterion='mse',
max_features=None, min_density=0.1, min_samples_leaf=1,
min_samples_split=2, random_state=None), n_estimators=50, learning_rate=0.1, compute_importances=True)
```

An AdaBoost regressor.

An AdaBoost 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=0.1)

Learning rate shrinks the contribution of each regressor by `learning_rate`. There is a trade-off between `learning_rate` and `n_estimators`.

`compute_importances` : boolean, optional (default=False)

Whether feature importances are computed and stored in the `feature_importances_` attribute when calling `fit`.

**See Also:**

`AdaBoostClassifier`, `GradientBoostingRegressor`, `DecisionTreeRegressor`

## References

[R78], [R79]

## Attributes

<code>estimators_</code>	list of classifiers	The collection of fitted sub-estimators.
<code>estimator_weights_</code>	list of floats	Weights for each estimator in the boosted ensemble.
<code>errors_</code>	list of floats	Regression error for each estimator in the boosted ensemble.
<code>feature_importances_</code>	array of shape = [ <code>n_features</code> ]	The feature importances if supported by the <code>base_estimator</code> . Only computed if <code>compute_importances=True</code> .

## Methods

---



---



---



---



---



---



---



---

`AdaBoostRegressor.fit(X, y, sample_weight=None)`

Build a boosted regressor from the training set (X, y).

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The training input samples.

`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.

**Returns self** : object

Returns self.

`AdaBoostRegressor.get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep`: boolean, optional :

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

`AdaBoostRegressor.predict(X, n_estimators=-1)`

Predict regression value for X.

The predicted regression value of an input sample is computed as the weighted mean prediction of the classifiers in the ensemble.

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The input samples.

`n_estimators` : int, optional (default=-1)

Use only the first `n_estimators` classifiers for the prediction. This is useful for grid searching the `n_estimators` parameter since it is not necessary to fit separately for all choices of `n_estimators`, but only the highest `n_estimators`. Any negative value will result in all estimators being used.

**Returns** `y` : array of shape = [n\_samples]

The predicted regression values.

`AdaBoostRegressor.score(X, y)`

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 - 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]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns** **z** : float

`AdaBoostRegressor.set_params (**params)`

Set the parameters of the 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** :

`AdaBoostRegressor.staged_predict (X, n_estimators=-1)`

Return staged predictions for X.

The predicted regression value of an input sample is computed as the weighted mean prediction of the classifiers in the ensemble.

This generator method yields the ensemble prediction after each iteration of boosting and therefore allows monitoring, such as to determine the prediction on a test set after each boost.

**Parameters** **X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**n\_estimators** : int, optional (default=-1)

Use only the first n\_estimators classifiers for the prediction. This is useful for grid searching the n\_estimators parameter since it is not necessary to fit separately for all choices of n\_estimators, but only the highest n\_estimators. Any negative value will result in all estimators being used.

**Returns** **y** : generator of array, shape = [n\_samples]

The predicted regression values.

`AdaBoostRegressor.staged_score (X, y, n_estimators=-1)`

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, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns** **z** : float

**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, max_depth=3,
                                                 init=None, random_state=None,
                                                 max_features=None, verbose=0,
                                                 learn_rate=None)
```

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’}, optional (default=‘deviance’)

loss function to be optimized. ‘deviance’ refers to deviance (= logistic regression) for classification with probabilistic outputs.

**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 overfitting 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.

**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.

**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, None, optional (default=None)

The number of features to consider when looking for the best split. Features are chosen randomly at each split point. If None, then `max_features=n_features`. Choosing `max_features < n_features` leads to a reduction of variance and an increase in bias.

**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 ‘.’ for every tree built. If greater than 1 then it prints the score for every tree.

#### See Also:

`sklearn.tree.DecisionTreeClassifier`, `RandomForestClassifier`

#### 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.

#### Examples

```
>>> samples = [[0, 0, 2], [1, 0, 0]]
>>> labels = [0, 1]
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> gb = GradientBoostingClassifier().fit(samples, labels)
>>> print(gb.predict([[0.5, 0, 0]]))
[0]
```

#### Attributes

<code>feature_importances_</code>	array, shape = [n_features]	The feature importances (the higher, the more important the feature).
<code>oob_score_</code>	array, shape = [n_estimators]	Score of the training dataset obtained using an out-of-bag estimate. The i-th score <code>oob_score_[i]</code> is the deviance (= loss) of the model at iteration <code>i</code> on the out-of-bag sample.
<code>train_score_</code>	array, shape = [n_estimators]	The i-th score <code>train_score_[i]</code> is the deviance (= loss) of the model at iteration <code>i</code> on the in-bag sample. If <code>subsample == 1</code> this is the deviance on the training data.
<code>loss_</code>	LossFunction	The concrete <code>LossFunction</code> object.
<code>init</code>	BaseEstimator	The estimator that provides the initial predictions. Set via the <code>init</code> argument or <code>loss.init_estimator</code> .
<code>estimators_</code> : list of Decision-TreeRegressor		The collection of fitted sub-estimators.

#### Methods

<code>decision_function(X)</code>	Compute the decision function of X.
<code>fit(X, y)</code>	Fit the gradient boosting model.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class for X.

Continued on next page

**Table 1.74 – continued from previous page**

<code>predict_proba(X)</code>	Predict class probabilities for X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>staged_decision_function(X)</code>	Compute decision function of X for each iteration.
<code>staged_predict(X)</code>	Predict class probabilities at each stage for X.
<code>staged_predict_proba(X)</code>	Predict class probabilities at each stage for X.

`__init__(loss='deviance', learning_rate=0.1, n_estimators=100, subsample=1.0, min_samples_split=2, min_samples_leaf=1, max_depth=3, init=None, random_state=None, max_features=None, verbose=0, learn_rate=None)`

#### `decision_function(X)`

Compute the decision function of X.

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns** `score` : array, shape = [n\_samples, k]

The decision function of the input samples. Classes are ordered by arithmetical order. Regression and binary classification are special cases with k == 1, otherwise k==n\_classes.

#### `fit(X, y)`

Fit the gradient boosting model.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features. Use fortran-style to avoid memory copies.

`y` : array-like, shape = [n\_samples]

Target values (integers in classification, real numbers in regression) For classification, labels must correspond to classes 0, 1, ..., n\_classes\_-1

**Returns** `self` : object

Returns self.

#### `get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep: boolean, optional :`

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

#### `predict(X)`

Predict class for X.

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns** `y` : array of shape = [n\_samples]

The predicted classes.

#### `predict_proba(X)`

Predict class probabilities for X.

**Parameters** **X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns** **p** : array of shape = [n\_samples]

The class probabilities of the input samples. Classes are ordered by arithmetical order.

**score** (*X*, *y*)

Returns the mean accuracy on the given test data and labels.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for **X**.

**Returns** **z** : float

**set\_params** (\*\**params*)

Set the parameters of the 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** :

**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. Classes are ordered by arithmetical order. Regression and binary classification are special cases with k == 1, otherwise k==n\_classes.

**staged\_predict** (*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** : array of shape = [n\_samples]

The predicted value of the input samples.

**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** : array of shape = [n\_samples]

The predicted value of the input samples.

## `sklearn.ensemble.GradientBoostingRegressor`

```
class sklearn.ensemble.GradientBoostingRegressor(loss='ls',           learning_rate=0.1,
                                                n_estimators=100,         subsam-
                                                ple=1.0,                 min_samples_split=2,
                                                min_samples_leaf=1,       max_depth=3,
                                                init=None,               random_state=None,
                                                max_features=None,       alpha=0.9,   ver-
                                                bose=0, learn_rate=None)
```

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.

**Parameters** `loss` : {‘ls’, ‘lad’, ‘huber’, ‘quantile’}, optional (default=’ls’)

loss function to be optimized. ‘ls’ refers to least squares regression. ‘lad’ (least absolute deviation) is a highly robust loss function solely based on order information of the input variables. ‘huber’ is a combination of the two. ‘quantile’ allows quantile regression (use `alpha` to specify the quantile).

**learning\_rate** : float, optional (default=0.1)

learning rate shrinks the contribution of each tree by `learning_rate`. There is a trade-off between `learning_rate` and `n_estimators`.

**n\_estimators** : int (default=100)

The number of boosting stages to perform. Gradient boosting is fairly robust to overfitting 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.

**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.

**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, None, optional (default=None)

The number of features to consider when looking for the best split. Features are chosen randomly at each split point. If None, then `max_features=n_features`. Choosing `max_features < n_features` leads to a reduction of variance and an increase in bias.

**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 ‘.’ for every tree built. If greater than 1 then it prints the score for every tree.

#### 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.

#### Examples

```
>>> samples = [[0, 0, 2], [1, 0, 0]]
>>> labels = [0, 1]
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> gb = GradientBoostingRegressor().fit(samples, labels)
>>> print(gb.predict([[0, 0, 0]]))
...
[ 1.32806...]
```

#### Attributes

<code>feature_importances_</code>	array, shape = [n_features]	The feature importances (the higher, the more important the feature).
<code>oob_score_</code>	array, shape = [n_estimators]	Score of the training dataset obtained using an out-of-bag estimate. The i-th score <code>oob_score_[i]</code> is the deviance (= loss) of the model at iteration <code>i</code> on the out-of-bag sample.
<code>train_score_</code>	array, shape = [n_estimators]	The i-th score <code>train_score_[i]</code> is the deviance (= loss) of the model at iteration <code>i</code> on the in-bag sample. If <code>subsample == 1</code> this is the deviance on the training data.
<code>loss_</code>	LossFunc- tion	The concrete <code>LossFunction</code> object.
<code>init</code>	BaseEsti- mator	The estimator that provides the initial predictions. Set via the <code>init</code> argument or <code>loss.init_estimator</code> .
<code>estimators_</code> : list of Decision- TreeRegressor		The collection of fitted sub-estimators.

## Methods

<code>decision_function(X)</code>	Compute the decision function of X.
<code>fit(X, y)</code>	Fit the gradient boosting model.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict regression target for X.
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>staged_decision_function(X)</code>	Compute decision function of X for each iteration.
<code>staged_predict(X)</code>	Predict regression target at each stage for X.

`__init__(loss='ls', learning_rate=0.1, n_estimators=100, subsample=1.0, min_samples_split=2, min_samples_leaf=1, max_depth=3, init=None, random_state=None, max_features=None, alpha=0.9, verbose=0, learn_rate=None)`

### `decision_function(X)`

Compute the decision function of X.

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns** `score` : array, shape = [n\_samples, k]

The decision function of the input samples. Classes are ordered by arithmetical order. Regression and binary classification are special cases with k == 1, otherwise k==n\_classes.

### `fit(X, y)`

Fit the gradient boosting model.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features. Use fortran-style to avoid memory copies.

`y` : array-like, shape = [n\_samples]

Target values (integers in classification, real numbers in regression) For classification, labels must correspond to classes 0, 1, ..., n\_classes\_-1

**Returns** `self` : object

Returns self.

### `get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep`: boolean, optional :

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

### `predict(X)`

Predict regression target for X.

**Parameters** `X` : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns** `y`: array of shape = [n\_samples] :

The predicted values.

**score**(*X*, *y*)

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as  $(1 - \frac{u}{v})$ , where  $u$  is the regression sum of squares  $((y - y_{\text{pred}})^2).sum()$  and  $v$  is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters** *X* : array-like, shape = [n\_samples, n\_features]

Training set.

*y* : array-like, shape = [n\_samples]**Returns** *z* : float**set\_params**(\*\*params)

Set the parameters of the 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**:**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. Classes are ordered by arithmetical order. Regression and binary classification are special cases with k == 1, otherwise k==n\_classes.

**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.

**Parameters** *X* : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns y** : array of shape = [n\_samples]

The predicted value of the input samples.

---

## 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.`

---

**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 gbdt.

**Parameters** **gbdt** : BaseGradientBoosting

A fitted gradient boosting model.

**target\_variables** : array-like, dtype=int

The target features for which the partial dependency should be computed (size should be smaller than 3 for visual renderings).

**grid** : array-like, shape=(n\_points, len(target\_variables))

The grid of target\_variables values for which the partial dependency should be evaluated (either grid or X must be specified).

**X** : array-like, shape=(n\_samples, n\_features)

The data on which gbdt 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 to create the extreme values for the grid. Only if X is not None.

**grid\_resolution** : int, default=100

The number of equally spaced points on the grid.

**Returns** **pdp** : array, shape=(n\_classes, n\_points)

The partial dependence function evaluated on the grid. For regression and binary classification n\_classes==1.

**axes** : seq of ndarray or None

The axes with which the grid has been created or None if the grid has been given.

## Examples

```
>>> samples = [[0, 0, 2], [1, 0, 0]]
>>> labels = [0, 1]
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> gb = GradientBoostingClassifier().fit(samples, labels)
>>> kwargs = dict(X=samples, percentiles=(0, 1), grid_resolution=2)
>>> partial_dependence(gb, [0], **kwargs)
(array([-10.72892297,  10.72892297])), [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.

**Parameters** **gbrt** : 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.

**Returns** `fig` : figure

The Matplotlib Figure object.

**axs** : seq of Axis objects

A seq of Axis objects, one for each subplot.

## Examples

```
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> X, y = make_friedman1()
>>> clf = GradientBoostingRegressor(n_estimators=10).fit(X, y)
>>> fig, axs = plot_partial_dependence(clf, X, [0, (0, 1)])
...
...
```

## 1.8.8 `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.

<code>feature_extraction.DictVectorizer([dtype, ...])</code>	Transforms lists of feature-value mappings to vectors.
<code>feature_extraction.FeatureHasher([...])</code>	Implements feature hashing, aka the hashing trick.

### `sklearn.feature_extraction.DictVectorizer`

```
class sklearn.feature_extraction.DictVectorizer(dtype=<type 'numpy.float64'>, separator='=', sparse=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.

**Parameters** `dtype` : callable, optional

The type of feature values. Passed to Numpy array/scipy.sparse matrix constructors as the `dtype` argument.

**separator: string, optional :**

Separator string used when constructing new features for one-hot coding.

**sparse: boolean, optional. :**

Whether transform should produce `scipy.sparse` matrices. True by default.

## 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

<code>fit(X[, y])</code>	Learn a list of feature name -> indices mappings.
<code>fit_transform(X[, y])</code>	Learn a list of feature name -> indices mappings and transform X.
<code>get_feature_names()</code>	Returns a list of feature names, ordered by their indices.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>inverse_transform(X[, dict_type])</code>	Transform array or sparse matrix X back to feature mappings.
<code>restrict(support[, indices])</code>	Restrict the features to those in support.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, y])</code>	Transform feature->value dicts to array or sparse matrix.

`__init__(dtype=<type 'numpy.float64'>, separator='=', sparse=True)`

`fit(X, y=None)`

Learn a list of feature name -> indices mappings.

**Parameters** `X` : Mapping or iterable over Mappings

Dict(s) or Mapping(s) from feature names (arbitrary Python objects) to feature values (strings or convertible to `dtype`).

`y` : (ignored)

**Returns self :**

`fit_transform(X, y=None)`

Learn a list of feature name -> indices mappings and transform X.

Like `fit(X)` followed by `transform(X)`.

**Parameters** `X` : Mapping or iterable over Mappings

Dict(s) or Mapping(s) from feature names (arbitrary Python objects) to feature values (strings or convertible to dtype).

**y** : (ignored)

**Returns Xa** : {array, sparse matrix}

Feature vectors; always 2-d.

**get\_feature\_names()**

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 the estimator

**Parameters deep: boolean, optional :**

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

**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.

**Parameters X** : {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.

**Returns D** : list of dict\_type objects, length = n\_samples

Feature mappings for the samples in X.

**restrict(support, indices=False)**

Restrict the features to those in support.

**Parameters support** : array-like

Boolean mask or list of indices (as returned by the get\_support member of feature selectors).

**indices** : boolean, optional

Whether support is a list of indices.

**set\_params(\*\*params)**

Set the parameters of the 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, y=None)

Transform feature->value dicts to array or sparse matrix.

Named features not encountered during fit or fit\_transform will be silently ignored.

**Parameters** **X** : Mapping or iterable over Mappings, length = n\_samples

Dict(s) or Mapping(s) from feature names (arbitrary Python objects) to feature values (strings or convertible to dtype).

**y** : (ignored)

**Returns** **Xa** : {array, sparse matrix}

Feature vectors; always 2-d.

**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.

**Parameters** **n\_features** : integer, optional

The number of features (columns) in the output matrices. Small numbers of features are likely to cause hash collisions, but large numbers will cause larger coefficient dimensions in linear learners.

**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

Whether output matrices should contain non-negative values only; effectively calls abs on the matrix prior to returning it. When True, output values will be multinomially distributed. When False, output values will be normally distributed (Gaussian) with mean 0, assuming a good hash function.

**Methods**

<code>fit([X, y])</code>	No-op.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(raw_X[, y])</code>	Transform a sequence of instances to a scipy.sparse matrix.

**`__init__(n_features=1048576, input_type='dict', dtype=<type 'numpy.float64'>, non_negative=False)`**

**`fit(X=None, y=None)`**  
No-op.

This method doesn't do anything. It exists purely for compatibility with the scikit-learn transformer API.

**Returns self :** 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.

**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 the estimator

**Parameters deep: boolean, optional :**

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

**`set_params(**params)`**

Set the parameters of the 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(raw_X, y=None)`**

Transform a sequence of instances to a scipy.sparse matrix.

**Parameters raw\_X :** iterable over iterable over raw features, length = n\_samples

Samples. Each sample must be iterable an (e.g., a list or tuple) containing/generating feature names (and optionally values, see the input\_type constructor argument) which will be hashed. raw\_X need not support the len function, so it can be the result of a generator; n\_samples is determined on the fly.

**y :** (ignored)

**Returns X :** scipy.sparse matrix, shape = (n\_samples, self.n\_features)

Feature matrix, for use with estimators or further transformers.

## From images

The `sklearn.feature_extraction.image` submodule gathers utilities to extract features from images.

<code>feature_extraction.image.img_to_graph(img[, ...])</code>	Graph of the pixel-to-pixel gradient connections
<code>feature_extraction.image.grid_to_graph(n_x, n_y)</code>	Graph of the pixel-to-pixel connections
<code>feature_extraction.image.extract_patches_2d(...)</code>	Reshape a 2D image into a collection of patches
<code>feature_extraction.image.reconstruct_from_patches_2d(...)</code>	Reconstruct the image from all of its patches.
<code>feature_extraction.image.PatchExtractor([...])</code>	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.

**Parameters img: ndarray, 2D or 3D :**

2D or 3D image

**mask : ndarray of booleans, optional**

An optional mask of the image, to consider only part of the pixels.

**return\_as: np.ndarray or a sparse matrix class, optional :**

The class to use to build the returned adjacency matrix.

**dtype: None or dtype, optional :**

The data of the returned sparse matrix. By default it is the dtype of img

### `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.

**Parameters n\_x: int :**

Dimension in x axis

**n\_y: int :**

Dimension in y axis

**n\_z: int, optional, default 1 :**

Dimension in z axis

**mask : ndarray of booleans, optional**

An optional mask of the image, to consider only part of the pixels.

**return\_as: np.ndarray or a sparse matrix class, optional :**

The class to use to build the returned adjacency matrix.

**dtype: dtype, optional, default int :**

The data of the returned sparse matrix. By default it is int

## 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.

**Parameters image: array, shape = (image\_height, image\_width) or :**

(image\_height, image\_width, n\_channels) The original image data. For color images, the last dimension specifies the channel: a RGB image would have *n\_channels*=3.

**patch\_size: tuple of ints (patch\_height, patch\_width) :**

the dimensions of one patch

**max\_patches: integer or float, optional default is None :**

The maximum number of patches to extract. If max\_patches is a float between 0 and 1, it is taken to be a proportion of the total number of patches.

**random\_state: int or RandomState :**

Pseudo number generator state used for random sampling to use if max\_patches is not None.

**Returns patches: array, shape = (n\_patches, patch\_height, patch\_width) or :**

(n\_patches, patch\_height, patch\_width, n\_channels) The collection of patches extracted from the image, where *n\_patches* is either max\_patches or the total number of patches that can be extracted.

## Examples

```
>>> from sklearn.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]])
```

## 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

## 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

<code>fit(X[, y])</code>	Do nothing and return the estimator unchanged
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Transforms the image samples in X into a matrix of patch data.

**\_\_init\_\_(patch\_size=None, max\_patches=None, random\_state=None)**

**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 the estimator

**Parameters deep: boolean, optional :**

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

**set\_params(\*\*params)**

Set the parameters of the 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)**

Transforms the image samples in X into a matrix of patch data.

**Parameters X : array, shape = (n\_samples, image\_height, image\_width) or**

(n\_samples, image\_height, image\_width, n\_channels) Array of images from which to extract patches. For color images, the last dimension specifies the channel: a RGB image would have *n\_channels*=3.

**Returns patches: array, shape = (n\_patches, patch\_height, patch\_width) or :**

(n\_patches, patch\_height, patch\_width, n\_channels) The collection of patches extracted from the images, where *n\_patches* is either *n\_samples* \* *max\_patches* or the total number of patches that can be extracted.

## From text

The `sklearn.feature_extraction.text` submodule gathers utilities to build feature vectors from text documents.

<code>feature_extraction.text.CountVectorizer(...)</code>	Convert a collection of text documents to a matrix of token counts
<code>feature_extraction.text.HashingVectorizer(...)</code>	Convert a collection of text documents to a matrix of token occurrences
<code>feature_extraction.text.TfidfTransformer(...)</code>	Transform a count matrix to a normalized tf or tf-idf representation
<code>feature_extraction.text.TfidfVectorizer(...)</code>	Convert a collection of raw documents to a matrix of TF-IDF features

**sklearn.feature\_extraction.text.CountVectorizer**

```
class sklearn.feature_extraction.text.CountVectorizer(input='content', charset='utf-8', charset_error='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), min_n=None, max_n=None, analyzer='word', max_df=1.0, min_df=2, max_features=None, vocabulary=None, binary=False, dtype=<type 'long'>)
```

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 analysing the data. The default analyzer does simple stop word filtering for English.

**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 ‘read’ method (file-like object) it 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.

`charset` : string, ‘utf-8’ by default.

If bytes or files are given to analyze, this charset is used to decode.

`charset_error` : {‘strict’, ‘ignore’, ‘replace’}

Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given `charset`. 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 a string, it is passed to \_check\_stop\_list and the appropriate stop list is returned 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 *tokenize == ‘word’*. The default regexp select tokens of 2 or more letters characters (punctuation is completely ignored and always treated as a token separator).

**max\_df** : float in range [0.0, 1.0] or int, optional, 1.0 by default

When building the vocabulary ignore terms that have a term 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, optional, 2 by default

When building the vocabulary ignore terms that have a term 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** : optional, None by default

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, False by default.

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().

**See Also:**

[HashingVectorizer](#), [TfidfVectorizer](#)

**Attributes**

<code>vocabulary_</code>	dict	A mapping of terms to feature indices.
<code>stop_wordset</code>		Terms that were ignored because they occurred in either too many ( <i>max_df</i> ) or in too few ( <i>min_df</i> ) documents. This is only available if no vocabulary was given.

**Methods**

<code>build_analyzer()</code>	Return a callable that handles preprocessing and tokenization
<code>build_preprocessor()</code>	Return a function to preprocess the text before tokenization
<code>build_tokenizer()</code>	Return a function that split a string in sequence of tokens
<code>decode(doc)</code>	Decode the input into a string of unicode symbols
<code>fit(raw_documents[, y])</code>	Learn a vocabulary dictionary of all tokens in the raw documents.
<code>fit_transform(raw_documents[, y])</code>	Learn the vocabulary dictionary and return the count vectors.
<code>get_feature_names()</code>	Array mapping from feature integer indices to feature name
<code>get_params([deep])</code>	Get parameters for the estimator
<code>get_stop_words()</code>	Build or fetch the effective stop words list
<code>inverse_transform(X)</code>	Return terms per document with nonzero entries in X.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(raw_documents)</code>	Extract token counts out of raw text documents using the vocabulary fitted with fit or the c

```
__init__(input='content',      charset='utf-8',      charset_error='strict',      strip_accents=None,
        lowercase=True,     preprocessor=None,    tokenizer=None,      stop_words=None,      to-
        ken_pattern=u'(?u)b\\w\\w+b',   ngram_range=(1, 1),   min_n=None,      max_n=None,
        analyzer='word',      max_df=1.0,      min_df=2,      max_features=None, vocabulary=None, bi-
        nary=False,      dtype=<type 'long'>)
```

`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 split a string in sequence of tokens

`decode(doc)`

Decode the input into a string of unicode symbols

The decoding strategy depends on the vectorizer parameters.

`fit(raw_documents, y=None)`

Learn a vocabulary dictionary of all tokens in the raw documents.

**Parameters** `raw_documents` : iterable

An iterable which yields either str, unicode or file objects.

**Returns self** :

**fit\_transform**(*raw\_documents*, *y=None*)  
Learn the vocabulary dictionary and return the count vectors.  
This is more efficient than calling fit followed by transform.

**Parameters** *raw\_documents* : iterable  
An iterable which yields either str, unicode or file objects.

**Returns** *vectors* : array, [n\_samples, n\_features]

**get\_feature\_names**()  
Array mapping from feature integer indices to feature name

**get\_params**(*deep=True*)  
Get parameters for the estimator

**Parameters** *deep*: boolean, optional :  
If True, will return the parameters for this estimator and contained subobjects that are estimators.

**get\_stop\_words**()  
Build or fetch the effective stop words list

**inverse\_transform**(*X*)  
Return terms per document with nonzero entries in *X*.

**Parameters** *X* : {array, sparse matrix}, shape = [n\_samples, n\_features]  
**Returns** *X\_inv* : list of arrays, len = n\_samples  
List of arrays of terms.

**set\_params**(\**params*)  
Set the parameters of the 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**(*raw\_documents*)  
Extract token counts out of raw text documents using the vocabulary fitted with fit or the one provided in the constructor.

**Parameters** *raw\_documents* : iterable  
An iterable which yields either str, unicode or file objects.

**Returns** *vectors* : sparse matrix, [n\_samples, n\_features]

**sklearn.feature\_extraction.text.HashingVectorizer**

```
class sklearn.feature_extraction.text.HashingVectorizer(input='content', charset='utf-8', charset_error='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='word', n_features=1048576, binary=False, norm='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 advantage:

- 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.

**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 ‘read’ method (file-like object) it 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.

**charset: string, ‘utf-8’ by default. :**

If bytes or files are given to analyze, this charset is used to decode.

**charset\_error: {‘strict’, ‘ignore’, ‘replace’} :**

Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given *charset*. 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  $\text{min\_n} \leq n \leq \text{max\_n}$  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 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.

**lowercase: boolean, default True :**

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 letters characters (punctuation is completely ignored and always treated as a token separator).

**n\_features : interger, optional, (2 \*\* 20) by default**

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, False by default. :**

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, optional**

Whether output matrices should contain non-negative values only; effectively calls abs on the matrix prior to returning it. When True, output values will be multinomially distributed. When False, output values will be normally distributed (Gaussian) with mean 0, assuming a good hash function.

**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 split a string in 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 instances to a scipy.sparse matrix.
get_params([deep])	Get parameters for the 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 the estimator.
transform(X[, y])	Transform a sequence of instances to a scipy.sparse matrix.

```
__init__(input='content', charset='utf-8', charset_error='strict', strip_accents=None,
lowercase=True, preprocessor=None, tokenizer=None, stop_words=None,
token_pattern=u'(?u)\\b\\w+\\b', ngram_range=(1, 1), analyzer='word',
n_features=1048576, binary=False, norm='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 split a string in sequence of tokens

decode(doc)
    Decode the input into a string of unicode symbols
    The decoding strategy depends on the vectorizer parameters.

fit(X, y=None)
    Does nothing: this transformer is stateless.

fit_transform(X, y=None)
    Transform a sequence of instances to a scipy.sparse matrix.

Parameters X : iterable over raw text documents, length = n_samples
```

Samples. Each sample must be a text document (either bytes or unicode strings, file name or file object depending on the constructor argument) which will be tokenized and hashed.

**y** : (ignored)

**Returns X** : scipy.sparse matrix, shape = (n\_samples, self.n\_features)

Feature matrix, for use with estimators or further transformers.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**get\_stop\_words ()**

Build or fetch the effective stop words list

**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 the 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, y=None)**

Transform a sequence of instances to a scipy.sparse matrix.

**Parameters X** : iterable over raw text documents, length = n\_samples

Samples. Each sample must be a text document (either bytes or unicode strings, file name or file object depending on the constructor argument) which will be tokenized and hashed.

**y** : (ignored)

**Returns X** : scipy.sparse matrix, shape = (n\_samples, self.n\_features)

Feature matrix, for use with estimators or further transformers.

## sklearn.feature\_extraction.text.TfidfTransformer

```
class sklearn.feature_extraction.text.TfidfTransformer(norm='l2',      use_idf=True,
                                                       smooth_idf=True,      sublinear_tf=False)
```

Transform a count matrix to a normalized tf or tf-idf representation

Tf means term-frequency while tf-idf means term-frequency times inverse document-frequency. This is a common term weighting scheme in information retrieval, that has also found good use in document classification.

The goal of using tf-idf instead of the raw frequencies of occurrence of a token in a given document is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than features that occur in a small fraction of the training corpus.

In the SMART notation used in IR, this class implements several tf-idf variants. Tf is always “n” (natural), idf is “t” iff use\_idf is given, “n” otherwise, and normalization is “c” iff norm='l2', “n” iff norm=None.

**Parameters** `norm` : ‘l1’, ‘l2’ or None, optional

Norm used to normalize term vectors. None for no normalization.

`use_idf` : boolean, optional

Enable inverse-document-frequency reweighting.

`smooth_idf` : boolean, optional

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, optional

Apply sublinear tf scaling, i.e. replace tf with  $1 + \log(\text{tf})$ .

## References

[Yates2011], [MSR2008]

## Methods

<code>fit(X[, y])</code>	Learn the idf vector (global term weights)
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, copy])</code>	Transform a count matrix to a tf or tf-idf representation

`__init__(norm='l2', use_idf=True, smooth_idf=True, sublinear_tf=False)`

`fit(X, y=None)`

Learn the idf vector (global term weights)

**Parameters** `X` : sparse matrix, [n\_samples, n\_features]

a matrix of term/token counts

`fit_transform(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 the estimator

**Parameters deep: boolean, optional :**

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

**set\_params** (\*\*params)

Set the parameters of the 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, copy=True)

Transform a count matrix to a tf or tf-idf representation

**Parameters X** : sparse matrix, [n\_samples, n\_features]

a matrix of term/token counts

**Returns vectors** : sparse matrix, [n\_samples, n\_features]

**sklearn.feature\_extraction.text.TfidfVectorizer**

```
class sklearn.feature_extraction.text.TfidfVectorizer(input='content',      charset='utf-
8',          charset_error='strict',
strip_accents=None,      lower-
ercase=True,          preproces-
sor=None,          tokenizer=None, ana-
lyzer='word', stop_words=None,
token_pattern=u'(?u)\b\w\w+\b',
min_n=None,      max_n=None,
ngram_range=(1,           1),
max_df=1.0,      min_df=2,
max_features=None,      vocab-
ulary=None,      binary=False,
dtype=<type 'long'>, norm='l2',
use_idf=True,      smooth_idf=True,
sublinear_tf=False)
```

Convert a collection of raw documents to a matrix of TF-IDF features.

Equivalent to CountVectorizer followed by TfidfTransformer.

**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 ‘read’ method (file-like object) it 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.

**charset** : string, ‘utf-8’ by default.

If bytes or files are given to analyze, this charset is used to decode.

**charset\_error** : {‘strict’, ‘ignore’, ‘replace’}

Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given *charset*. 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 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 tokenize == ‘word’. The default regexp select tokens of 2 or more letters characters (punctuation is completely ignored and always treated as a token separator).

**max\_df** : float in range [0.0, 1.0] or int, optional, 1.0 by default

When building the vocabulary ignore terms that have a term 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, optional, 2 by default

When building the vocabulary ignore terms that have a term 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** : optional, None by default

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, False by default.

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().

**norm** : ‘l1’, ‘l2’ or None, optional

Norm used to normalize term vectors. None for no normalization.

**use\_idf** : boolean, optional

Enable inverse-document-frequency reweighting.

**smooth\_idf** : boolean, optional

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, optional

Apply sublinear tf scaling, i.e. replace tf with  $1 + \log(tf)$ .

## 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.

## Methods

<code>build_analyzer()</code>	Return a callable that handles preprocessing and tokenization
<code>build_preprocessor()</code>	Return a function to preprocess the text before tokenization
<code>build_tokenizer()</code>	Return a function that split a string in sequence of tokens
<code>decode(doc)</code>	Decode the input into a string of unicode symbols
<code>fit(raw_documents[, y])</code>	Learn a conversion law from documents to array data
<code>fit_transform(raw_documents[, y])</code>	Learn the representation and return the vectors.
<code>get_feature_names()</code>	Array mapping from feature integer indices to feature name
<code>get_params([deep])</code>	Get parameters for the estimator

Continued on next page

**Table 1.87 – continued from previous page**

<code>get_stop_words()</code>	Build or fetch the effective stop words list
<code>inverse_transform(X)</code>	Return terms per document with nonzero entries in X.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(raw_documents[, copy])</code>	Transform raw text documents to tf-idf vectors

`__init__(input='content', charset='utf-8', charset_error='strict', strip_accents=None, lower_case=True, preprocessor=None, tokenizer=None, analyzer='word', stop_words=None, token_pattern=u'(?u)\\b\\w+\\b', min_n=None, max_n=None, ngram_range=(1, 1), max_df=1.0, min_df=2, max_features=None, vocabulary=None, binary=False, dtype=<type 'long'>, norm='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 split a string in sequence of tokens

**`decode(doc)`**

Decode the input into a string of unicode symbols

The decoding strategy depends on the vectorizer parameters.

**`fit(raw_documents, y=None)`**

Learn a conversion law from documents to array data

**`fit_transform(raw_documents, y=None)`**

Learn the representation and return the vectors.

**Parameters `raw_documents` : iterable**

an iterable which yields either str, unicode or file objects

**Returns `vectors` : array, [n\_samples, n\_features]****`get_feature_names()`**

Array mapping from feature integer indices to feature name

**`get_params(deep=True)`**

Get parameters for the estimator

**Parameters `deep: boolean, optional` :**

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

**`get_stop_words()`**

Build or fetch the effective stop words list

**`inverse_transform(X)`**

Return terms per document with nonzero entries in X.

**Parameters `X` : {array, sparse matrix}, shape = [n\_samples, n\_features]****Returns `X_inv` : list of arrays, len = n\_samples**

List of arrays of terms.

**`set_params(**params)`**

Set the parameters of the 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**(*raw\_documents*, *copy=True*)  
Transform raw text documents to tf-idf vectors

**Parameters raw\_documents** : iterable

an iterable which yields either str, unicode or file objects

**Returns vectors** : sparse matrix, [n\_samples, n\_features]

## 1.8.9 sklearn.feature\_selection: Feature Selection

The `sklearn.feature_selection` module implements feature selection algorithms. It currently includes univariate filter selection methods and the recursive feature elimination algorithm.

**User guide:** See the *Feature selection* section for further details.

<code>feature_selection.SelectPercentile([...])</code>	Select features according to a percentile of the highest scores.
<code>feature_selection.SelectKBest([score_func, k])</code>	Select features according to the k highest scores.
<code>feature_selection.SelectFpr([score_func, alpha])</code>	Filter: Select the pvalues below alpha based on a FPR test.
<code>feature_selection.SelectFdr([score_func, alpha])</code>	Filter: Select the p-values for an estimated false discovery rate
<code>feature_selection.SelectFwe([score_func, alpha])</code>	Filter: Select the p-values corresponding to Family-wise error rate
<code>feature_selection.RFE(estimator[, ...])</code>	Feature ranking with recursive feature elimination.
<code>feature_selection.RFECV(estimator[, step, ...])</code>	Feature ranking with recursive feature elimination and cross-validated

### sklearn.feature\_selection.SelectPercentile

**class** `sklearn.feature_selection.SelectPercentile(score_func=<function f_classif at 0x3b96b18>, percentile=10)`

Select features according to a percentile of the highest scores.

**Parameters score\_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.

#### Notes

Ties between features with equal p-values will be broken in an unspecified way.

#### Attributes

<code>scores_</code>	array-like, shape=(n_features,)	Scores of features.
<code>pvalues_</code>	array-like, shape=(n_features,)	p-values of feature scores.

## Methods

<code>fit(X, y)</code>	Evaluate the function
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>get_support([indices])</code>	Return a mask, or list, of the features/indices selected.
<code>inverse_transform(X)</code>	Transform a new matrix using the selected features
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Transform a new matrix using the selected features

`__init__(score_func=<function f_classif at 0x3b96b18>, percentile=10)`

**fit(X, y)**  
Evaluate the function

**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 the estimator

**Parameters deep: boolean, optional :**  
If True, will return the parameters for this estimator and contained subobjects that are estimators.

**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 the 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)**  
Transform a new matrix using the selected features

## sklearn.feature\_selection.SelectKBest

`class sklearn.feature_selection.SelectKBest(score_func=<function f_classif at 0x3b96b18>, 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, optional, default=10

Number of top features to select.

**Notes**

Ties between features with equal scores will be broken in an unspecified way.

**Attributes**

<code>scores_</code>	array-like, shape=(n_features,)	Scores of features.
<code>pvalues_</code>	array-like, shape=(n_features,)	p-values of feature scores.

**Methods**

<code>fit(X, y)</code>	Evaluate the function
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>get_support([indices])</code>	Return a mask, or list, of the features/indices selected.
<code>inverse_transform(X)</code>	Transform a new matrix using the selected features
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Transform a new matrix using the selected features

`__init__(score_func=<function f_classif at 0x3b96b18>, k=10)`

`fit(X, y)`

Evaluate the function

`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 the estimator

**Parameters** `deep: boolean, optional` :

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

**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 the 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*)  
 Transform a new matrix using the selected features

## sklearn.feature\_selection.SelectFpr

**class** sklearn.feature\_selection.**SelectFpr** (*score\_func=<function f\_classif at 0x3b96b18>, 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.

**Parameters** **score\_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.

## Attributes

<i>scores_</i>	array-like, shape=(n_features,)	Scores of features.
<i>pvalues_</i>	array-like, shape=(n_features,)	p-values of feature scores.

## Methods

<b>fit</b> (X, y)	Evaluate the function
<b>fit_transform</b> (X[, y])	Fit to data, then transform it
<b>get_params</b> ([deep])	Get parameters for the estimator
<b>get_support</b> ([indices])	Return a mask, or list, of the features/indices selected.
<b>inverse_transform</b> (X)	Transform a new matrix using the selected features
<b>set_params</b> (**params)	Set the parameters of the estimator.
<b>transform</b> (X)	Transform a new matrix using the selected features

**\_\_init\_\_** (*score\_func=<function f\_classif at 0x3b96b18>, alpha=0.05*)

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

Evaluate the function

**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 the estimator

**Parameters** **deep**: boolean, optional :

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

**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 the 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*)

Transform a new matrix using the selected features

## sklearn.feature\_selection.SelectFdr

**class** sklearn.feature\_selection.**SelectFdr** (*score\_func*=<function f\_classif at 0x3b96b18>, *alpha*=0.05)

Filter: Select the p-values for an estimated false discovery rate

This uses the Benjamini-Hochberg procedure. *alpha* is the target 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

<i>scores_</i>	array-like, shape=(n_features,)	Scores of features.
<i>pvalues_</i>	array-like, shape=(n_features,)	p-values of feature scores.

## Methods

<code>fit(X, y)</code>	Evaluate the function
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>get_support([indices])</code>	Return a mask, or list, of the features/indices selected.
<code>inverse_transform(X)</code>	Transform a new matrix using the selected features
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Transform a new matrix using the selected features

`__init__(score_func=<function f_classif at 0x3b96b18>, alpha=0.05)`

`fit(X, y)`

Evaluate the function

`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 the estimator

**Parameters** `deep: boolean, optional` :

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

`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 the 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)`

Transform a new matrix using the selected features

**sklearn.feature\_selection.SelectFwe**

**class** sklearn.feature\_selection.**SelectFwe** (*score\_func*=<function f\_classif at 0x3b96b18>, *alpha*=0.05)

Filter: Select the p-values corresponding to Family-wise error 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**

<i>scores_</i>	array-like, shape=(n_features,)	Scores of features.
<i>pvalues_</i>	array-like, shape=(n_features,)	p-values of feature scores.

**Methods**

<b>fit(X, y)</b>	Evaluate the function
<b>fit_transform(X[, y])</b>	Fit to data, then transform it
<b>get_params([deep])</b>	Get parameters for the estimator
<b>get_support([indices])</b>	Return a mask, or list, of the features/indices selected.
<b>inverse_transform(X)</b>	Transform a new matrix using the selected features
<b>set_params(**params)</b>	Set the parameters of the estimator.
<b>transform(X)</b>	Transform a new matrix using the selected features

**\_\_init\_\_** (*score\_func*=<function f\_classif at 0x3b96b18>, *alpha*=0.05)

**fit** (X, y)

Evaluate the function

**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 the estimator

**Parameters** **deep**: boolean, optional :

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

```
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 the 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)
    Transform a new matrix using the selected features
```

## sklearn.feature\_selection.RFE

```
class sklearn.feature_selection.RFE(estimator, n_features_to_select=None, step=1, estimator_params={}, 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 of 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.

## References

[R86]

## 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])
```

## Attributes

<code>n_features_</code>	<code>int</code>	The number of selected features.
<code>support_</code>	array of shape [n_features]	The mask of selected features.
<code>ranking_</code>	array of shape [n_features]	The feature ranking, such that <code>ranking_[i]</code> corresponds to the ranking position of the i-th feature. Selected (i.e., estimated best) features are assigned rank 1.
<code>estimator_</code>	object	The external estimator fit on the reduced dataset.

## Methods

<code>decision_function(X)</code>	
<code>fit(X, y)</code>	Fit the RFE model and then the underlying estimator on the selected
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Reduce X to the selected features and then predict using the
<code>predict_proba(X)</code>	
<code>score(X, y)</code>	Reduce X to the selected features and then return the score of the
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Reduce X to the selected features during the elimination.

`__init__(estimator, n_features_to_select=None, step=1, estimator_params={}, verbose=0)`

`fit(X, y)`

Fit the RFE model and then the underlying estimator on the selected features.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

The training input samples.

`y` : array-like, shape = [n\_samples]

The target values.

`get_params(deep=True)`

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**predict (X)**

Reduce X to the selected features and then predict using the underlying estimator.

**Parameters X : array of shape [n\_samples, n\_features]**

The input samples.

**Returns y : array of shape [n\_samples]**

The predicted target values.

**score (X, y)**

Reduce X to the selected features and then return the score of the underlying estimator.

**Parameters X : array of shape [n\_samples, n\_features]**

The input samples.

**y : array of shape [n\_samples]**

The target values.

**set\_params (\*\*params)**

Set the parameters of the 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 during the elimination.

**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 features selected during the elimination.

## sklearn.feature\_selection.RFECV

```
class sklearn.feature_selection.RFECV(estimator, step=1, cv=None, loss_func=None, estimator_params={}, verbose=0)
```

**Feature ranking with recursive feature elimination and cross-validated** selection of the best number of features.

**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.

**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.

**loss\_function** : function, optional (default=None)

The loss function to minimize by cross-validation. If None, then the score function of the estimator is maximized.

**estimator\_params** : dict

Parameters for the external estimator. Useful for doing grid searches.

**verbose** : int, default=0

Controls verbosity of output.

## References

[R87]

## 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])
```

## Attributes

<code>n_features_</code>	int	The number of selected features with cross-validation.
<code>support_</code>	array of shape [n_features]	The mask of selected features.
<code>ranking_</code>	array of shape [n_features]	The feature ranking, such that <code>ranking_[i]</code> corresponds to the ranking position of the i-th feature. Selected (i.e., estimated best) features are assigned rank 1.
<code>cv_scores_</code>	array of shape [n_subsets_of_features]	The cross-validation scores such that <code>cv_scores_[i]</code> corresponds to the CV score of the i-th subset of features.
<code>estimator_</code>	object	The external estimator fit on the reduced dataset.

## Methods

<code>decision_function(X)</code>	
<code>fit(X, y)</code>	Fit the RFE model and automatically tune the number of selected
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Reduce X to the selected features and then predict using the
<code>predict_proba(X)</code>	
<code>score(X, y)</code>	Reduce X to the selected features and then return the score of the
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Reduce X to the selected features during the elimination.

`__init__(estimator, step=1, cv=None, loss_func=None, estimator_params={}, verbose=0)`

**fit (X, y)**

Fit the RFE model and automatically tune the number of selected features.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training vector, where `n_samples` is the number of samples and `n_features` is the total number of features.

`y` : array-like, shape = [n\_samples]

Target values (integers for classification, real numbers for regression).

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters** `deep: boolean, optional :`

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

**predict (X)**

Reduce X to the selected features and then predict using the underlying estimator.

**Parameters** `X` : array of shape [n\_samples, n\_features]

The input samples.

**Returns** `y` : array of shape [n\_samples]

The predicted target values.

**score**(X, y)

Reduce X to the selected features and then return the score of the underlying estimator.

**Parameters** **X** : array of shape [n\_samples, n\_features]

The input samples.

**y** : array of shape [n\_samples]

The target values.

**set\_params**(\*\*params)

Set the parameters of the 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 during the elimination.

**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 features selected during the elimination.

---

**feature\_selection.chi2**(X, y)

Compute  $\chi^2$  (chi-squared) statistic for each class/feature combination.

**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

---

**sklearn.feature\_selection.chi2****sklearn.feature\_selection.chi2**(X, y)

Compute  $\chi^2$  (chi-squared) statistic for each class/feature combination.

This score can be used to select the n\_features features with the highest values for the  $\chi^2$  (chi-square) statistic from X, which must contain booleans or frequencies (e.g., term counts in document classification), relative to the classes.

Recall that the  $\chi^2$  statistic 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.

**Parameters** **X** : {array-like, sparse matrix}, shape = (n\_samples, n\_features\_in)

Sample vectors.

**y** : array-like, shape = (n\_samples,)

Target vector (class labels).

**Returns** **chi2** : array, shape = (n\_features,)

chi2 statistics of each feature.

**pval** : array, shape = (n\_features,)

p-values of each feature.

## Notes

Complexity of this algorithm is  $O(n_{\text{classes}} * n_{\text{features}})$ .

### sklearn.feature\_selection.f\_classif

`sklearn.feature_selection.f_classif(X, y)`

Compute the Anova F-value for the provided sample

**Parameters** `X` : {array-like, sparse matrix} shape = [n\_samples, n\_features]

The set of regressors that will be tested sequentially.

`y` : array of shape(n\_samples)

The data matrix.

**Returns** `F` : array, shape = [n\_features, ]

The set of F values.

`pval` : array, shape = [n\_features, ]

The set of p-values.

### 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

**Parameters** `X` : {array-like, sparse matrix} shape = (n\_samples, n\_features)

The set of regressors that will be tested sequentially.

`y` : array of shape(n\_samples).

The data matrix

`center` : True, bool,

If true, X and y will be centered.

**Returns** `F` : array, shape=(n\_features, )

F values of features.

`pval` : array, shape=(n\_features, )

p-values of F-scores.

## 1.8.10 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.

---

## `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=2.2204460492503131e-15,
                                                random_state=None)
```

The Gaussian Process model class.

### **Parameters**

**regr** : 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 `x` and `x'`. Default assumes a squared-exponential autocorrelation model. Built-in correlation models are:

```
'absolute_exponential', 'squared_exponential',
'generalized_exponential', 'cubic', 'linear'
```

### **beta0** : double array\_like, optional

The regression weight vector to perform Ordinary Kriging (OK). Default assumes Universal Kriging (UK) so that the vector `beta` of regression weights is estimated using the maximum likelihood principle.

### **storage\_mode** : string, optional

A string specifying whether the Cholesky decomposition of the correlation matrix should be stored in the class (`storage_mode = 'full'`) or not (`storage_mode = 'light'`). Default assumes `storage_mode = 'full'`, so that the Cholesky decomposition of the correlation matrix is stored. This might be a useful parameter when one is not interested in the MSE and only plan to estimate the BLUP, for which the correlation matrix is not required.

### **verbose** : boolean, optional

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 due 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.

## 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...  
...
```

## Attributes

<i>theta_</i> : array	Specified theta OR the best set of autocorrelation parameters (the sought maximizer of the reduced likelihood function).
<i>reduced_likelihood_function_value_</i> : array	The optimal reduced likelihood function value.

## Methods

<code>arg_max_reduced_likelihood_function(*args, ...)</code>	DEPRECATED: to be removed in 0.14, access <code>self.theta_</code> etc.
<code>fit(X, y)</code>	The Gaussian Process model fitting method.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X[, eval_MSE, batch_size])</code>	This function evaluates the Gaussian Process model at <code>x</code> .
<code>reduced_likelihood_function([theta])</code>	This function determines the BLUP parameters and evaluates the reduced likelihood function.
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

```
__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)
```

```
arg_max_reduced_likelihood_function(*args, **kwargs)
```

DEPRECATED: to be removed in 0.14, access `self.theta_` etc. directly after fit.

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

The Gaussian Process model fitting method.

**Parameters** `X` : 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, ) with the observations of the scalar output to be predicted.

**Returns gp : self**

A fitted Gaussian Process model object awaiting data to perform predictions.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**predict (X, eval\_MSE=False, batch\_size=None)**

This function evaluates the Gaussian Process model at x.

**Parameters X : 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.

**Returns y : array\_like**

An array with shape (n\_eval, ) with the Best Linear Unbiased Prediction at x.

**MSE : array\_like, optional (if eval\_MSE == True)**

An array with shape (n\_eval, ) with the Mean Squared Error at x.

**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.

**Parameters theta : 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\_).

**Returns reduced\_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.

**beta**Generalized least-squares regression weights for Universal Kriging or given beta0 for Ordinary Kriging.

**gamma**Gaussian Process weights.

CCholesky decomposition of the correlation matrix [R].

**Ft**Solution of the linear equation system : [R] x Ft = F

**GQR** decomposition of the matrix Ft.

**reduced\_likelihood\_function\_value**

DEPRECATED: `reduced_likelihood_function_value` is deprecated and will be removed in 0.14, please use `reduced_likelihood_function_value_` instead.

**score**(X, y)

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  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters** X : array-like, shape = [n\_samples, n\_features]

Training set.

y : array-like, shape = [n\_samples]

**Returns** z : float

**set\_params**(\*\*params)

Set the parameters of the 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**:

**theta**

DEPRECATED: `theta` is deprecated and will be removed in 0.14, please use `theta_` instead.

<code>gaussian_process.correlation_models.absolute_exponential(...)</code>	Absolute exponential autocorrelation model
<code>gaussian_process.correlation_models.squared_exponential(...)</code>	Squared exponential correlation model
<code>gaussian_process.correlation_models.generalized_exponential(...)</code>	Generalized exponential correlation model
<code>gaussian_process.correlation_models.pure_nugget(...)</code>	Spatial independence correlation model
<code>gaussian_process.correlation_models.cubic(...)</code>	Cubic correlation model:
<code>gaussian_process.correlation_models.linear(...)</code>	Linear correlation model:
<code>gaussian_process.regression_models.constant(x)</code>	Zero order polynomial (constant, p = 1)
<code>gaussian_process.regression_models.linear(x)</code>	First order polynomial (linear, p = n+1)
<code>gaussian_process.regression_models.quadratic(x)</code>	Second order polynomial (quadratic, p = n+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):

$$\text{theta}, \text{dx} \rightarrow r(\text{theta}, \text{dx}) = \exp\left(\sum_{i=1}^n -\text{theta}_i * |\text{dx}_i|\right)$$

**Parameters** theta : array\_like

An array with shape 1 (isotropic) or n (anisotropic) giving the autocorrelation parameter(s).

**dx** : 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.

**Returns r** : array\_like

An array with shape (n\_eval, ) containing the values of the autocorrelation model.

### **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):

$$\text{theta, dx} \rightarrow r(\text{theta}, \text{dx}) = \exp\left(\sum_{i=1}^n -\theta_i * (\text{dx}_i)^2\right)$$

**Parameters theta** : array\_like

An array with shape 1 (isotropic) or n (anisotropic) giving the autocorrelation parameter(s).

**dx** : 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.

**Returns r** : array\_like

An array with shape (n\_eval, ) containing the values of the autocorrelation model.

### **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.):

$$\text{theta, dx} \rightarrow r(\text{theta}, \text{dx}) = \exp\left(\sum_{i=1}^n -\theta_i * |\text{dx}_i|^p\right)$$

**Parameters theta** : array\_like

An array with shape 1+1 (isotropic) or n+1 (anisotropic) giving the autocorrelation parameter(s) (theta, p).

**dx** : 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.

**Returns r** : array\_like

An array with shape (n\_eval, ) with the values of the autocorrelation model.

**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!):

```
theta, dx --> r(theta, dx) = 1 if sum_i |dx_i| == 0
                           i = 1
                           0 otherwise
```

**Parameters theta** : array\_like

None.

**dx** : 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.

**Returns r** : array\_like

An array with shape (n\_eval, ) with the values of the autocorrelation model.

**sklearn.gaussian\_process.correlation\_models.cubic**

```
sklearn.gaussian_process.correlation_models.cubic(theta, d)
```

Cubic correlation model:

```
theta, dx --> r(theta, dx) =
prod_j max(0, 1 - 3(theta_j*d_ij)^2 + 2(theta_j*d_ij)^3), i = 1, ..., m
j = 1
```

**Parameters theta** : array\_like

An array with shape 1 (isotropic) or n (anisotropic) giving the autocorrelation parameter(s).

**dx** : 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.

**Returns r** : array\_like

An array with shape (n\_eval, ) with the values of the autocorrelation model.

**sklearn.gaussian\_process.correlation\_models.linear**

```
sklearn.gaussian_process.correlation_models.linear(theta, d)
```

Linear correlation model:

```
theta, dx --> r(theta, dx) =
prod_j max(0, 1 - theta_j*d_ij), i = 1, ..., m
j = 1
```

**Parameters theta** : array\_like

An array with shape 1 (isotropic) or n (anisotropic) giving the autocorrelation parameter(s).

**dx** : 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.

**Returns r** : array\_like

An array with shape (n\_eval, ) with the values of the autocorrelation model.

### **sklearn.gaussian\_process.regression\_models.constant**

`sklearn.gaussian_process.regression_models.constant(x)`

Zero order polynomial (constant, p = 1) regression model.

$x \rightarrow f(x) = 1$

**Parameters x** : array\_like

An array with shape (n\_eval, n\_features) giving the locations x at which the regression model should be evaluated.

**Returns f** : array\_like

An array with shape (n\_eval, p) with the values of the regression model.

### **sklearn.gaussian\_process.regression\_models.linear**

`sklearn.gaussian_process.regression_models.linear(x)`

First order polynomial (linear, p = n+1) regression model.

$x \rightarrow f(x) = [1, x_1, \dots, x_n]^T$

**Parameters x** : array\_like

An array with shape (n\_eval, n\_features) giving the locations x at which the regression model should be evaluated.

**Returns f** : array\_like

An array with shape (n\_eval, p) with the values of the regression model.

### **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.

$x \rightarrow f(x) = [1, \{x_i, i=1,\dots,n\}, \{x_i * x_j, (i,j)=1,\dots,n\}]^T$

**Parameters x** : array\_like

An array with shape (n\_eval, n\_features) giving the locations x at which the regression model should be evaluated.

**Returns f** : array\_like

An array with shape (n\_eval, p) with the values of the regression model.

## 1.8.11 `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: setting estimator parameters* section for further details.

<code>grid_search.GridSearchCV(estimator, param_grid)</code>	Grid search on the parameters of a classifier
<code>grid_search.IterGrid(param_grid)</code>	Generators on the combination of the various parameter lists given

### `sklearn.grid_search.GridSearchCV`

```
class sklearn.grid_search.GridSearchCV(estimator, param_grid, 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')
```

Grid search on the parameters of a classifier

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.

**Parameters estimator: 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.

**loss\_func: callable, optional :**

function that takes 2 arguments and compares them in order to evaluate the performance of prediciton (small is good) if None is passed, the score of the estimator is maximized

**score\_func: callable, optional :**

A function that takes 2 arguments and compares them in order to evaluate the performance of prediction (high is good). If None is passed, the score of the estimator is maximized.

**fit\_params : dict, optional**

parameters to pass to the `fit` method

**n\_jobs: int, optional :**

number of jobs to run in parallel (default 1)

**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 immediatly 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, optional :**

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 crossvalidation generator, optional**

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

**refit: boolean :**

refit the best estimator with the entire dataset. If “False”, it is impossible to make predictions using this GridSearch instance after fitting.

**verbose: integer :**

Controls the verbosity: the higher, the more messages.

**See Also:**

`IterGrid`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.

**Notes**

The parameters selected are those that maximize the score of the left out data, unless an explicit score\_func is passed in which case it is used instead. If a loss function loss\_func is passed, it overrides the score functions and is minimized.

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,
    estimator=SVC(C=1.0, cache_size=..., coef0=..., degree=...,
                  gamma=..., kernel='rbf', max_iter=-1, probability=False,
                  shrinking=True, tol=...),
    fit_params={}, iid=True, loss_func=None, n_jobs=1,
    param_grid=...,
    ...)
```

## Attributes

<code>grid_scores_</code>	<code>dict of any to float</code>	Contains scores for all parameter combinations in param_grid.
<code>best_estimator_</code>	<code>estimator</code>	Estimator that was chosen by grid search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data.
<code>best_score_</code>	<code>float</code>	score of best_estimator on the left out data.
<code>best_params_</code>	<code>dict</code>	Parameter setting that gave the best results on the hold out data.

## Methods

<code>fit(X[, y])</code>	Run fit with all sets of parameters
<code>get_params([deep])</code>	Get parameters for the estimator
<code>score(X[, y])</code>	
<code>set_params(**params)</code>	Set the parameters of the estimator.

**`__init__(estimator, param_grid, 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')`**

**`fit(X, y=None, **params)`**  
Run fit with all sets of parameters

Returns the best classifier

**Parameters X: array, [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], optional :**

Target vector relative to X for classification; None for unsupervised learning.

**`get_params(deep=True)`**  
Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**`set_params(**params)`**  
Set the parameters of the 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.grid\_search.IterGrid

**class sklearn.grid\_search.IterGrid(param\_grid)**  
Generators on the combination of the various parameter lists given

**Parameters param\_grid: dict of string to sequence :**

The parameter grid to explore, as a dictionary mapping estimator parameters to sequences of allowed values.

**Returns params: dict of string to any :**

**Yields** dictionaries mapping each estimator parameter to one of its allowed values.

**See Also:**

`GridSearchCV` uses `IterGrid` to perform a full parallelized grid search.

## Examples

```
>>> from sklearn.grid_search import IterGrid
>>> param_grid = {'a':[1, 2], 'b':[True, False]}
>>> list(IterGrid(param_grid))
[{'a': 1, 'b': True}, {'a': 1, 'b': False},
 {'a': 2, 'b': True}, {'a': 2, 'b': False}]
__init__(param_grid)
```

## 1.8.12 `sklearn.hmm`: Hidden Markov Models

The `sklearn.hmm` module implements hidden Markov models.

**Warning:** `sklearn.hmm` is orphaned, undocumented and has known numerical stability issues. If nobody volunteers to write documentation and make it more stable, this module will be removed in version 0.11.

**User guide:** See the *Hidden Markov Models* section for further details.

<code>hmm.GaussianHMM([n_components, ...])</code>	Hidden Markov Model with Gaussian emissions
<code>hmm.MultinomialHMM([n_components, ...])</code>	Hidden Markov Model with multinomial (discrete) emissions
<code>hmm.GMMHMM([n_components, n_mix, startprob, ...])</code>	Hidden Markov Model with Gausin mixture emissions

### `sklearn.hmm.GaussianHMM`

```
class sklearn.hmm.GaussianHMM(n_components=1, covariance_type='diag', start-
                                prob=None, transmat=None, startprob_prior=None, trans-
                                mat_prior=None, algorithm='viterbi', means_prior=None,
                                means_weight=0, covars_prior=0.01, covars_weight=1,
                                random_state=None, n_iter=10, thresh=0.01,
                                params='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ',
                                init_params='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

Hidden Markov Model with Gaussian emissions

Representation of a hidden Markov model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a HMM.

**Parameters** `n_components` : int

Number of states.

`“covariance_type“` : string

String describing the type of covariance parameters to use. Must be one of ‘spherical’, ‘tied’, ‘diag’, ‘full’. Defaults to ‘diag’.

**See Also:****GMM**Gaussian mixture model**Examples**

```
>>> from sklearn.hmm import GaussianHMM
>>> GaussianHMM(n_components=2)
...
GaussianHMM(algorithm='viterbi', ...)
```

**Attributes**

_covariance_type	string	String describing the type of covariance parameters used by the model. Must be one of ‘spherical’, ‘tied’, ‘diag’, ‘full’.
n_features	int	Dimensionality of the Gaussian emissions.
n_components	int	Number of states in the model.
transmat	array, shape (n_components, n_components)	Matrix of transition probabilities between states.
startprob	array, shape (‘n_components’,)	Initial state occupation distribution.
means	array, shape (n_components, n_features)	Mean parameters for each state.
covars	array	Covariance parameters for each state. The shape depends on _covariance_type: (‘n_components’,) (‘n_features’, ‘n_features’) (‘n_components’, ‘n_features’) (‘n_components’, ‘n_features’, ‘n_features’)
random_state: RandomState or an int seed (0 by default)		A random number generator instance
n_iter	int, optional	Number of iterations to perform.
thresh	float, optional	Convergence threshold.
params	string, optional	Controls which parameters are updated in the training process. Can contain any combination of ‘s’ for startprob, ‘t’ for transmat, ‘m’ for means, and ‘c’ for covars, etc. Defaults to all parameters.
init_params	string, optional	Controls which parameters are initialized prior to training. Can contain any combination of ‘s’ for startprob, ‘t’ for transmat, ‘m’ for means, and ‘c’ for covars, etc. Defaults to all parameters.

## Methods

<code>decode(obs[, algorithm])</code>	Find most likely state sequence corresponding to <i>obs</i> .
<code>eval(obs)</code>	Compute the log probability under the model and compute posteriors
<code>fit(obs, **kwargs)</code>	Estimate model parameters.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(obs[, algorithm])</code>	Find most likely state sequence corresponding to <i>obs</i> .
<code>predict_proba(obs)</code>	Compute the posterior probability for each state in the model
<code>sample([n, random_state])</code>	Generate random samples from the model.
<code>score(obs)</code>	Compute the log probability under the model.
<code>set_params(**params)</code>	Set the parameters of the estimator.

**\_\_init\_\_(n\_components=1, covariance\_type='diag', startprob=None, transmat=None, mat=None, startprob\_prior=None, transmat\_prior=None, algorithm='viterbi', means\_prior=None, means\_weight=0, covars\_prior=0.01, covars\_weight=1, random\_state=None, n\_iter=10, thresh=0.01, params='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ', init\_params='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ')**

**algorithm**  
decoder algorithm

**covariance\_type**  
Covariance type of the model.

Must be one of ‘spherical’, ‘tied’, ‘diag’, ‘full’.

**covars\_**  
Return covars as a full matrix.

**decode(obs, algorithm='viterbi')**

Find most likely state sequence corresponding to *obs*. Uses the selected algorithm for decoding.

**Parameters obs** : array\_like, shape (n, n\_features)

List of n\_features-dimensional data points. Each row corresponds to a single data point.

**algorithm** : string, one of the *decoder\_algorithms*

decoder algorithm to be used

**Returns logprob** : float

Log probability of the maximum likelihood path through the HMM

**state\_sequence** : array\_like, shape (n,)

Index of the most likely states for each observation

**See Also:**

**eval**Compute the log probability under the model and posteriors

**score**Compute the log probability under the model

**eval(obs)**

Compute the log probability under the model and compute posteriors

Implements rank and beam pruning in the forward-backward algorithm to speed up inference in large models.

**Parameters** `obs` : array\_like, shape (n, n\_features)

Sequence of n\_features-dimensional data points. Each row corresponds to a single point in the sequence.

**Returns** `logprob` : float

Log likelihood of the sequence `obs`

**posteriors:** array\_like, shape (n, n\_components) :

Posterior probabilities of each state for each observation

**See Also:**

**score** Compute the log probability under the model

**decode** Find most likely state sequence corresponding to a `obs`

**fit** (`obs`, \*\*`kwargs`)

Estimate model parameters.

An 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 ``. Likewise, if you would like just to do an initialization, call this method with `n_iter=0`.

**Parameters** `obs` : list

List of array-like observation sequences (shape (n\_i, n\_features)).

**Notes**

In general, `logprob` should be non-decreasing unless aggressive pruning is used. Decreasing `logprob` is generally a sign of overfitting (e.g. a covariance parameter getting too small). You can fix this by getting more training data, or decreasing `covars_prior`.

**Please note that setting parameters in the ‘fit’ method is deprecated and will be removed in the next release. Set it on initialization instead.**

**get\_params** (`deep=True`)

Get parameters for the estimator

**Parameters** `deep`: boolean, optional :

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

**means\_**

Mean parameters for each state.

**predict** (`obs`, `algorithm='viterbi'`)

Find most likely state sequence corresponding to `obs`.

**Parameters** `obs` : array\_like, shape (n, n\_features)

List of n\_features-dimensional data points. Each row corresponds to a single data point.

**Returns** `state_sequence` : array\_like, shape (n,)

Index of the most likely states for each observation

**predict\_proba** (`obs`)

Compute the posterior probability for each state in the model

**Parameters** `obs` : array\_like, shape (n, n\_features)

List of n\_features-dimensional data points. Each row corresponds to a single data point.

**Returns** `T` : array-like, shape (n, n\_components)

Returns the probability of the sample for each state in the model.

**sample** (`n=1, random_state=None`)

Generate random samples from the model.

**Parameters** `n` : int

Number of samples to generate.

**random\_state: RandomState or an int seed (0 by default)** :

A random number generator instance. If None is given, the object's random\_state is used

**Returns (obs, hidden\_states) :**

`obs` : array\_like, length  $n$  List of samples

`hidden_states` : array\_like, length  $n$  List of hidden states

**score** (`obs`)

Compute the log probability under the model.

**Parameters** `obs` : array\_like, shape (n, n\_features)

Sequence of n\_features-dimensional data points. Each row corresponds to a single data point.

**Returns** `logprob` : float

Log likelihood of the `obs`

**See Also:**

**eval**Compute the log probability under the model and posteriors

**decode**Find most likely state sequence corresponding to a `obs`

**set\_params** (\*\*`params`)

Set the parameters of the 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` :

**startprob\_**

Mixing startprob for each state.

**transmat\_**

Matrix of transition probabilities.

## sklearn.hmm.MultinomialHMM

```
class sklearn.hmm.MultinomialHMM(n_components=1,          startprob=None,          transmat=None,
                                    startprob_prior=None,      transmat_prior=None,      algo-
                                    rithm='viterbi', random_state=None, n_iter=10, thresh=0.01,
                                    params='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ',
                                    init_params='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

Hidden Markov Model with multinomial (discrete) emissions

See Also:

[GaussianHMMHMM](#) with Gaussian emissions

### Examples

```
>>> from sklearn.hmm import MultinomialHMM
>>> MultinomialHMM(n_components=2)
...
MultinomialHMM(algorithm='viterbi', ...)
```

### Attributes

n_components	int	Number of states in the model.
n_symbols	int	Number of possible symbols emitted by the model (in the observations).
transmat	array, shape (n_components, n_components)	Matrix of transition probabilities between states.
startprob	array, shape (‘n_components’,)	Initial state occupation distribution.
emissionprob	array, shape (‘n_components’, ‘n_symbols’)	Probability of emitting a given symbol when in each state.
random_state: RandomState or an int seed (0 by default)		A random number generator instance
n_iter	int, optional	Number of iterations to perform.
thresh	float, optional	Convergence threshold.
params	string, optional	Controls which parameters are updated in the training process. Can contain any combination of ‘s’ for startprob, ‘t’ for transmat, ‘m’ for means, and ‘c’ for covars, etc. Defaults to all parameters.
init_params	string, optional	Controls which parameters are initialized prior to training. Can contain any combination of ‘s’ for startprob, ‘t’ for transmat, ‘m’ for means, and ‘c’ for covars, etc. Defaults to all parameters.

### Methods

<code>decode(obs[, algorithm])</code>	Find most likely state sequence corresponding to <i>obs</i> .
<code>eval(obs)</code>	Compute the log probability under the model and compute posteriors
<code>fit(obs, **kwargs)</code>	
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(obs[, algorithm])</code>	Find most likely state sequence corresponding to <i>obs</i> .
<code>predict_proba(obs)</code>	Compute the posterior probability for each state in the model
<code>sample([n, random_state])</code>	Generate random samples from the model.
<code>score(obs)</code>	Compute the log probability under the model.
<code>set_params(**params)</code>	Set the parameters of the estimator.

**`__init__(n_components=1, startprob=None, transmat=None, startprob_prior=None, transmat_prior=None, algorithm='viterbi', random_state=None, n_iter=10, thresh=0.01, params='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ', init_params='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ')`**  
Create a hidden Markov model with multinomial emissions.

**Parameters** `n_components` : int

Number of states.

**algorithm**

decoder algorithm

**decode(obs, algorithm='viterbi')**

Find most likely state sequence corresponding to *obs*. Uses the selected algorithm for decoding.

**Parameters** `obs` : array\_like, shape (n, n\_features)

List of n\_features-dimensional data points. Each row corresponds to a single data point.

**algorithm** : string, one of the *decoder\_algorithms*

decoder algorithm to be used

**Returns** `logprob` : float

Log probability of the maximum likelihood path through the HMM

**state\_sequence** : array\_like, shape (n,)

Index of the most likely states for each observation

**See Also:**

**eval** Compute the log probability under the model and posteriors

**score** Compute the log probability under the model

**emissionprob**

Emission probability distribution for each state.

**eval(obs)**

Compute the log probability under the model and compute posteriors

Implements rank and beam pruning in the forward-backward algorithm to speed up inference in large models.

**Parameters** `obs` : array\_like, shape (n, n\_features)

Sequence of n\_features-dimensional data points. Each row corresponds to a single point in the sequence.

**Returns logprob** : float

Log likelihood of the sequence *obs*

**posteriors: array\_like, shape (n, n\_components)** :

Posterior probabilities of each state for each observation

**See Also:**

**score**Compute the log probability under the model

**decode**Find most likely state sequence corresponding to a *obs*

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional** :

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

**predict (obs, algorithm='viterbi')**

Find most likely state sequence corresponding to *obs*.

**Parameters obs** : array\_like, shape (n, n\_features)

List of n\_features-dimensional data points. Each row corresponds to a single data point.

**Returns state\_sequence** : array\_like, shape (n,)

Index of the most likely states for each observation

**predict\_proba (obs)**

Compute the posterior probability for each state in the model

**Parameters obs** : array\_like, shape (n, n\_features)

List of n\_features-dimensional data points. Each row corresponds to a single data point.

**Returns T** : array-like, shape (n, n\_components)

Returns the probability of the sample for each state in the model.

**sample (n=1, random\_state=None)**

Generate random samples from the model.

**Parameters n** : int

Number of samples to generate.

**random\_state: RandomState or an int seed (0 by default)** :

A random number generator instance. If None is given, the object's random\_state is used

**Returns (obs, hidden\_states)** :

**obs** : array\_like, length *n* List of samples

**hidden\_states** : array\_like, length *n* List of hidden states

**score (obs)**

Compute the log probability under the model.

**Parameters obs** : array\_like, shape (n, n\_features)

Sequence of n\_features-dimensional data points. Each row corresponds to a single data point.

**Returns logprob** : float

Log likelihood of the *obs*

**See Also:**

**eval**Compute the log probability under the model and posteriors

**decode**Find most likely state sequence corresponding to a *obs*

**set\_params** (\*\*params)

Set the parameters of the 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** :

**startprob\_**

Mixing startprob for each state.

**transmat\_**

Matrix of transition probabilities.

## sklearn.hmm.GMMHMM

```
class sklearn.hmm.GMMHMM(n_components=1,           n_mix=1,           startprob=None,          trans-
                           mat=None,          startprob_prior=None,    transmat_prior=None,     al-
                           gorithm='viterbi',   gmms=None,          covariance_type='diag',  co-
                           vars_prior=0.01,    random_state=None,   n_iter=10,          thresh=0.01,
                           params='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ',
                           init_params='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

Hidden Markov Model with Gaussian mixture emissions

**See Also:**

**GaussianHMMHMM** with Gaussian emissions

### Examples

```
>>> from sklearn.hmm import GMMHMM
>>> GMMHMM(n_components=2, n_mix=10, covariance_type='diag')
...
GMMHMM(algorithm='viterbi', covariance_type='diag', ...)
```

**Attributes**

init_params	string, optional	Controls which parameters are initialized prior to training. Can contain any combination of ‘s’ for startprob, ‘t’ for transmat, ‘m’ for means, and ‘c’ for covars, etc. Defaults to all parameters.
params	string, optional	Controls which parameters are updated in the training process. Can contain any combination of ‘s’ for startprob, ‘t’ for transmat, ‘m’ for means, and ‘c’ for covars, etc. Defaults to all parameters.
n_components		Number of states in the model.
transmat	array, shape (n_components, n_components)	Matrix of transition probabilities between states.
startprob	array, shape (‘n_components’,)	Initial state occupation distribution.
gmms	array of GMM objects, length n_components	GMM emission distributions for each state.
random_state	RandomState or an int seed (0 by default)	A random number generator instance
n_iter	int, optional	Number of iterations to perform.
thresh	float, optional	Convergence threshold.

**Methods**

decode(obs[, algorithm])	Find most likely state sequence corresponding to <i>obs</i> .
eval(obs)	Compute the log probability under the model and compute posteriors
fit(obs, **kwargs)	Estimate model parameters.
get_params([deep])	Get parameters for the estimator
predict(obs[, algorithm])	Find most likely state sequence corresponding to <i>obs</i> .
predict_proba(obs)	Compute the posterior probability for each state in the model
sample([n, random_state])	Generate random samples from the model.
score(obs)	Compute the log probability under the model.
set_params(**params)	Set the parameters of the estimator.

```
__init__(n_components=1,      n_mix=1,      startprob=None,      transmat=None,      start-
        prob_prior=None,  transmat_prior=None,  algorithm='viterbi', gmms=None, covari-
        ance_type='diag', covars_prior=0.01, random_state=None, n_iter=10, thresh=0.01,
        params='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ',
        init_params='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

Create a hidden Markov model with GMM emissions.

**Parameters** `n_components` : int

Number of states.

**algorithm**

decoder algorithm

**covariance\_type**

Covariance type of the model.

Must be one of ‘spherical’, ‘tied’, ‘diag’, ‘full’.

**decode**(*obs*, *algorithm*=‘viterbi’)

Find most likely state sequence corresponding to *obs*. Uses the selected algorithm for decoding.

**Parameters** **obs** : array\_like, shape (n, n\_features)

List of n\_features-dimensional data points. Each row corresponds to a single data point.

**algorithm** : string, one of the *decoder\_algorithms*

decoder algorithm to be used

**Returns** **logprob** : float

Log probability of the maximum likelihood path through the HMM

**state\_sequence** : array\_like, shape (n,)

Index of the most likely states for each observation

**See Also:**

**eval**Compute the log probability under the model and posteriors

**score**Compute the log probability under the model

**eval**(*obs*)

Compute the log probability under the model and compute posteriors

Implements rank and beam pruning in the forward-backward algorithm to speed up inference in large models.

**Parameters** **obs** : array\_like, shape (n, n\_features)

Sequence of n\_features-dimensional data points. Each row corresponds to a single point in the sequence.

**Returns** **logprob** : float

Log likelihood of the sequence *obs*

**posteriors: array\_like, shape (n, n\_components) :**

Posterior probabilities of each state for each observation

**See Also:**

**score**Compute the log probability under the model

**decode**Find most likely state sequence corresponding to a *obs*

**fit**(*obs*, \*\**kwargs*)

Estimate model parameters.

An 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 ‘’. Likewise, if you would like just to do an initialization, call this method with n\_iter=0.

**Parameters** **obs** : list

List of array-like observation sequences (shape (n\_i, n\_features)).

## Notes

In general, *logprob* should be non-decreasing unless aggressive pruning is used. Decreasing *logprob* is generally a sign of overfitting (e.g. a covariance parameter getting too small). You can fix this by getting more training data, or decreasing *covars\_prior*.

**Please note that setting parameters in the ‘fit’ method is deprecated and will be removed in the next release. Set it on initialization instead.**

### `get_params(deep=True)`

Get parameters for the estimator

#### **Parameters** `deep: boolean, optional` :

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

### `predict(obs, algorithm='viterbi')`

Find most likely state sequence corresponding to *obs*.

#### **Parameters** `obs : array_like, shape (n, n_features)`

List of n\_features-dimensional data points. Each row corresponds to a single data point.

#### **Returns** `state_sequence : array_like, shape (n,)`

Index of the most likely states for each observation

### `predict_proba(obs)`

Compute the posterior probability for each state in the model

#### **Parameters** `obs : array_like, shape (n, n_features)`

List of n\_features-dimensional data points. Each row corresponds to a single data point.

#### **Returns** `T : array-like, shape (n, n_components)`

Returns the probability of the sample for each state in the model.

### `sample(n=1, random_state=None)`

Generate random samples from the model.

#### **Parameters** `n : int`

Number of samples to generate.

#### **random\_state: RandomState or an int seed (0 by default) :**

A random number generator instance. If None is given, the object’s random\_state is used

#### **Returns (obs, hidden\_states) :**

`obs : array_like, length n` List of samples

`hidden_states : array_like, length n` List of hidden states

### `score(obs)`

Compute the log probability under the model.

#### **Parameters** `obs : array_like, shape (n, n_features)`

Sequence of n\_features-dimensional data points. Each row corresponds to a single data point.

#### **Returns** `logprob : float`

Log likelihood of the *obs*

**See Also:**

**eval**Compute the log probability under the model and posteriors

**decode**Find most likely state sequence corresponding to a *obs*

**set\_params** (*\*\*params*)

Set the parameters of the 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 :**

**startprob\_**

Mixing startprob for each state.

**transmat\_**

Matrix of transition probabilities.

## 1.8.13 `sklearn.isotonic`: Isotonic regression

**User guide:** See the *Isotonic regression* section for further details.

---

`isotonic.IsotonicRegression([y_min, y_max])` Isotonic regression model.

---

### `sklearn.isotonic.IsotonicRegression`

**class** `sklearn.isotonic.IsotonicRegression` (*y\_min=None*, *y\_max=None*)  
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*.

## References

Isotonic Median Regression: A Linear Programming Approach Nilotpal Chakravarti Mathematics of Operations Research Vol. 14, No. 2 (May, 1989), pp. 303-308

## Attributes

<code>X_</code>	ndarray (n_samples, )	A copy of the input X.
<code>y_</code>	ndarray (n_samples, )	Isotonic fit of y.

## Methods

<code>fit(X, y[, weight])</code>	Fit the model using X, y as training data.
<code>fit_transform(X, y[, weight])</code>	Fit model and transform y by linear interpolation.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(T)</code>	Predict new data by linear interpolation.
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(T)</code>	Transform new data by linear interpolation

`__init__ (y_min=None, y_max=None)`

`fit (X, y, weight=None)`

Fit the model using X, y as training data.

**Parameters** `X` : array-like, shape=(n\_samples,)

Training data.

`y` : array-like, shape=(n\_samples,)

Training target.

`weight` : array-like, shape=(n\_samples,), optional, default: None

Weights. If set to None, all weights will be set to 1 (equal weights).

**Returns** `self` : object

Returns an instance of self.

## Notes

X is stored for future use, as `transform` needs X to interpolate new input data.

`fit_transform (X, y, weight=None)`

Fit model and transform y by linear interpolation.

**Parameters** `X` : array-like, shape=(n\_samples,)

Training data.

`y` : array-like, shape=(n\_samples,)

Training target.

**weight** : array-like, shape=(n\_samples,), optional, default: None

Weights. If set to None, all weights will be equal to 1 (equal weights).

**Returns ‘y\_‘** : array, shape=(n\_samples,)

The transformed data.

## Notes

X doesn't influence the result of *fit\_transform*. It is however stored for future use, as *transform* needs X to interpolate new input data.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**predict (T)**

Predict new data by linear interpolation.

**Parameters T** : array-like, shape=(n\_samples,)

Data to transform.

**Returns ‘T\_‘** : array, shape=(n\_samples,)

Transformed data.

**score (X, y)**

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  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}.mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params (\*\*params)**

Set the parameters of the 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 (T)**

Transform new data by linear interpolation

**Parameters T** : array-like, shape=(n\_samples,)

Data to transform.

**Returns ‘T\_‘** : array, shape=(n\_samples,)

The transformed data

---

isotonic.isotonic_regression(y[, weight, ...])	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
--	---

## **sklearn.isotonic.isotonic\_regression**

`sklearn.isotonic.isotonic_regression(y, weight=None, y_min=None, y_max=None)`

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)

**Parameters** `y` : iterable of floating-point values

The data.

`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  $y_{\max}$ .

**Returns** ‘`y`’ : 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.

## 1.8.14 `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.

---

<code>kernel_approximation.AdditiveChi2Sampler([...])</code>	Approximate feature map for additive chi <sup>2</sup> kernel.
<code>kernel_approximation.Nystroem([kernel, ...])</code>	Approximate a kernel map using a subset of the training data.
<code>kernel_approximation.RBFSampler([gamma, ...])</code>	Approximates feature map of an RBF kernel by Monte Carlo approx.
<code>kernel_approximation.SkewedChi2Sampler([...])</code>	Approximates feature map of the “skewed chi-squared” kernel by

## sklearn.kernel\_approximation.AdditiveChi2Sampler

```
class sklearn.kernel_approximation.AdditiveChi2Sampler(sample_steps=2, sample_interval=None)
```

Approximate feature map for additive chi<sup>2</sup> 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  $2 \times \text{sample\_steps} + 1$  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.

**Parameters** `sample_steps` : int, optional

Gives the number of (complex) sampling points.

`sample_interval` : float, optional

Sampling interval. Must be specified when `sample_steps` not in {1,2,3}.

**See Also:**

`SkewedChi2Sampler`A Fourier-approximation to a non-additive variant of the chi squared kernel.

`sklearn.metrics.chi2_kernel`The exact chi squared kernel.

`sklearn.metrics.additive_chi2_kernel`The exact additive chi squared kernel.

### Notes

This estimator approximates a slightly different version of the additive chi squared kernel than `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

<code>fit(X[, y])</code>	Set parameters.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, y])</code>	Apply approximate feature map to X.

`__init__(sample_steps=2, sample_interval=None)`

`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.

**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 the estimator

**Parameters** **deep**: boolean, optional :

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

**set\_params** (\*\**params*)

Set the parameters of the 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, y=None)

Apply approximate feature map to X.

**Parameters** **X**: {array-like, sparse matrix}, **shape** = (n\_samples, n\_features) :

**Returns** **X\_new**: {array, sparse matrix}, **shape** = (n\_samples, n\_features × (2×sample\_steps + 1)) :

Whether the return value is an array of sparse matrix depends on the type of the input X.

## sklearn.kernel\_approximation.Nystroem

```
class sklearn.kernel_approximation.Nystroem(kernel='rbf', gamma=None, coef0=1,
                                             degree=3, 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.

**Parameters** **kernel** : string or callable, default="rbf"

Kernel map to be approximated.

**n\_components** : int

Number of features to construct. How many data points will be used to construct the mapping.

**gamma** : float, default=1/n\_features.

Parameter for the RBF kernel.

**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:

**RBF Sampler** An approximation to the RBF kernel using random Fourier features.

**sklearn.metrics.pairwise.kernel\_metrics** List of build-in kernels.

#### References

- Williams, C.K.I. and Seeger, M. “Using the Nystrom 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

#### Attributes

<code>components_</code>	array, shape (n_components, n_features)	Subset of training points used to construct the feature map.
<code>component_indices_</code>	array, shape (n_components)	Indices of <code>components_</code> in the training set.
<code>normalization_</code>	array, shape (n_components, n_components)	Normalization matrix needed for embedding. Square root of the kernel matrix on <code>components_</code> .

#### Methods

<code>fit(X[, y])</code>	Fit estimator to data.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Apply feature map to X.

**`__init__(kernel='rbf', gamma=None, coef0=1, degree=3, n_components=100, random_state=None)`**

**`fit(X, y=None)`**

Fit estimator to data.

Samples a subset of training points, computes kernel on these and computes normalization matrix.

**Parameters** `X` : array-like, 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`.

**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 the estimator

**Parameters deep: boolean, optional :**

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

**set\_params** (\*\*params)

Set the parameters of the 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)

Apply feature map to X.

Computes an approximate feature map using the kernel between some training points and X.

**Parameters X** : array-like, shape=(n\_samples, n\_features)

Data to transform.

**Returns X\_transformed** : array, shape=(n\_samples, n\_components)

Transformed data.

## sklearn.kernel\_approximation.RBFSampler

```
class sklearn.kernel_approximation.RBFSampler(gamma=1.0, n_components=100.0, random_state=None)
```

Approximates feature map of an RBF kernel by Monte Carlo approximation of its Fourier transform.

**Parameters gamma: float :**

parameter of RBF kernel:  $\exp(-\gamma * x^2)$

**n\_components: int :**

number of Monte Carlo samples per original feature. Equals the dimensionality of the computed feature space.

**random\_state** : {int, RandomState}, 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.

## Methods

<code>fit(X[, y])</code>	Fit the model with X.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, y])</code>	Apply the approximate feature map to X.

`__init__(gamma=1.0, n_components=100.0, random_state=None)`

**fit** (*X*, *y*=*None*)

Fit the model with *X*.

Samples random projection according to *n\_features*.

**Parameters** *X*: {array-like, sparse matrix}, shape (*n\_samples*, *n\_features*) :

Training data, where *n\_samples* in the number of samples and *n\_features* is the number of features.

**Returns self** : object

Returns the transformer.

**fit\_transform** (*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 the estimator

**Parameters** *deep*: boolean, optional :

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

**set\_params** (\*\**params*)

Set the parameters of the 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*, *y*=*None*)

Apply the approximate feature map to *X*.

**Parameters** *X*: {array-like, sparse matrix}, shape (*n\_samples*, *n\_features*) :

New data, where n\_samples is the number of samples and n\_features is the number of features.

**Returns X\_new: array-like, shape (n\_samples, n\_components) :**

### sklearn.kernel\_approximation.SkewedChi2Sampler

```
class sklearn.kernel_approximation.SkewedChi2Sampler(skewedness=1.0,
                                                       n_components=100,           ran-
                                                       dom_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.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

<code>fit(X[, y])</code>	Fit the model with X.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, y])</code>	Apply the approximate feature map to X.

**\_\_init\_\_(skewedness=1.0, n\_components=100, random\_state=None)**

**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 is 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.

**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 the estimator

**Parameters deep: boolean, optional :**

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

**set\_params**(\*\*params)

Set the parameters of the 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, y=None)

Apply the approximate feature map to X.

**Parameters X: array-like, shape (n\_samples, n\_features) :**

New data, where n\_samples in the number of samples and n\_features is the number of features.

**Returns X\_new: array-like, shape (n\_samples, n\_components) :**

## 1.8.15 `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

**sklearn.semi\_supervised.LabelPropagation**

```
class sklearn.semi_supervised.LabelPropagation(kernel='rbf', gamma=20, n_neighbors=7,  

                                              alpha=1, max_iter=30, tol=0.001,  

                                              max_iters=None)
```

Label Propagation classifier

**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

change maximum number of iterations allowed

**tol** : float

Convergence tolerance: threshold to consider the system at steady state

**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**

<code>fit(X, y)</code>	Fit a semi-supervised label propagation model based
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Performs inductive inference across the model.
<code>predict_proba(X)</code>	Predict probability for each possible outcome.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(kernel='rbf', gamma=20, n_neighbors=7, alpha=1, max_iter=30, tol=0.001, max_iters=None)`

**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.

**Parameters** **X** : 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

**Returns self** : returns an instance of self.

**get\_params(deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**predict**(X)

Performs inductive inference across the model.

**Parameters** **X** : array\_like, shape = [n\_samples, n\_features]

**Returns y** : 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).

**Parameters** **X** : array\_like, shape = [n\_samples, n\_features]

**Returns probabilities** : array, shape = [n\_samples, n\_classes]

Normalized probability distributions across class labels

**score**(X, y)

Returns the mean accuracy on the given test data and labels.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns z** : float

**set\_params** (\*\*params)

Set the parameters of the 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.semi\_supervised.LabelSpreading

```
class sklearn.semi_supervised.LabelSpreading(kernel='rbf', gamma=20, n_neighbors=7,
                                              alpha=0.2, max_iter=30, tol=0.001,
                                              max_iters=None)
```

LabelSpreading model for semi-supervised learning

This model is similar to the basic Label Propagation algorithm, but uses affinity matrix based on the normalized graph Laplacian and soft clamping across the labels.

**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

**See Also:**

**LabelPropagation** Unregularized graph based semi-supervised learning

## References

Dengyong Zhou, Olivier Bousquet, Thomas Navin Lal, Jason Weston, Bernhard Schölkopf. 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

<code>fit(X, y)</code>	Fit a semi-supervised label propagation model based
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Performs inductive inference across the model.
<code>predict_proba(X)</code>	Predict probability for each possible outcome.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(kernel='rbf', gamma=20, n_neighbors=7, alpha=0.2, max_iter=30, tol=0.001, max_iters=None)`

**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.

**Parameters** **X** : 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

**Returns self** : returns an instance of self.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters** **deep**: boolean, optional :

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

**predict (X)**

Performs inductive inference across the model.

**Parameters** **X** : array\_like, shape = [n\_samples, n\_features]

**Returns y** : 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).

**Parameters** *X* : array-like, shape = [n\_samples, n\_features]

**Returns** **probabilities** : array, shape = [n\_samples, n\_classes]

Normalized probability distributions across class labels

**score**(*X*, *y*)

Returns the mean accuracy on the given test data and labels.

**Parameters** *X* : array-like, shape = [n\_samples, n\_features]

Training set.

*y* : array-like, shape = [n\_samples]

Labels for *X*.

**Returns** *z* : float

**set\_params**(\*\**params*)

Set the parameters of the 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 :

## 1.8.16 `sklearn.lda`: Linear Discriminant Analysis

The `sklearn.lda` module implements Linear Discriminant Analysis (LDA).

**User guide:** See the *Linear and Quadratic Discriminant Analysis* section for further details.

---

`lda.LDA([n_components, priors])` Linear Discriminant Analysis (LDA)

---

### `sklearn.lda.LDA`

**class** `sklearn.lda.LDA`(*n\_components=None*, *priors=None*)

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.

**Parameters** *n\_components*: int :

Number of components (< n\_classes - 1) for dimensionality reduction

*priors* : array, optional, shape = [n\_classes]

Priors on classes

**See Also:****sklearn.qda.QDA**Quadratic discriminant analysis**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)
>>> print(clf.predict([[ -0.8, -1]]))
[1]
```

**Attributes**

<i>coef_</i>	array-like, shape = [rank, n_classes - 1]	Coefficients of the features in the linear decision function. rank is min(rank_features, n_classes) where rank_features is the dimensionality of the spaces spanned by the features (i.e. n_features excluding redundant features).
<i>co-variance_</i>	array-like, shape = [n_features, n_features]	Covariance matrix (shared by all classes).
<i>means</i>	array-like, shape = [n_classes, n_features]	Class means.
<i>priors_</i>	array-like, shape = [n_classes]	Class priors (sum to 1).
<i>scalings_</i>	array-like, shape = [rank, n_classes - 1]	Scaling of the features in the space spanned by the class centroids.
<i>xbar_</i>	float, shape = [n_features]	Overall mean.

**Methods**

<code>decision_function(X)</code>	This function return the decision function values related to each
<code>fit(X, y[, store_covariance, tol])</code>	Fit the LDA model according to the given training data and parameters.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	This function does classification on an array of test vectors X.
<code>predict_log_proba(X)</code>	This function return posterior log-probabilities of classification
<code>predict_proba(X)</code>	This function return posterior probabilities of classification
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Project the data so as to maximize class separation (large separation between projected class mea

```
__init__(n_components=None, priors=None)
decision_function(X)
    This function return the decision function values related to each class on an array of test vectors X.

    Parameters X : array-like, shape = [n_samples, n_features]

    Returns C : 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.

fit(X, y, store_covariance=False, tol=0.0001)
    Fit the LDA model according to the given training data and parameters.

    Parameters X : array-like, shape = [n_samples, n_features]

        Training vector, where n_samples in the number of samples and n_features is the number of features.

    y : array, shape = [n_samples]

        Target values (integers)

    store_covariance : boolean

        If True the covariance matrix (shared by all classes) is computed and stored in self.covariance_ attribute.

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 the estimator

    Parameters deep: boolean, optional :

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

predict(X)
    This function does classification on an array of test vectors X.

    The predicted class C for each sample in X is returned.

    Parameters X : array-like, shape = [n_samples, n_features]

    Returns C : array, shape = [n_samples]

predict_log_proba(X)
    This function return posterior log-probabilities of classification according to each class on an array of test vectors X.

    Parameters X : array-like, shape = [n_samples, n_features]
```

**Returns C** : array, shape = [n\_samples, n\_classes]

**predict\_proba**(X)

This function return posterior probabilities of classification according to each class on an array of test vectors X.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

**Returns C** : array, shape = [n\_samples, n\_classes]

**score**(X, y)

Returns the mean accuracy on the given test data and labels.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns z** : float

**set\_params**(\*\*params)

Set the parameters of the 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)

Project the data so as to maximize class separation (large separation between projected class means and small variance within each class).

**Parameters X** : array-like, shape = [n\_samples, n\_features]

**Returns X\_new** : array, shape = [n\_samples, n\_components]

## 1.8.17 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.

<code>linear_model.ARDRegression([n_iter, tol, ...])</code>	Bayesian ARD regression.
<code>linear_model.BayesianRidge([n_iter, tol, ...])</code>	Bayesian ridge regression
<code>linear_model.ElasticNet([alpha, l1_ratio, ...])</code>	Linear Model trained with L1 and L2 prior as regularizer
<code>linear_model.ElasticNetCV([l1_ratio, eps, ...])</code>	Elastic Net model with iterative fitting along a regularization path
<code>linear_model.Lars([fit_intercept, verbose, ...])</code>	Least Angle Regression model a.k.a. LAR
<code>linear_model.LarsCV([fit_intercept, ...])</code>	Cross-validated Least Angle Regression model
<code>linear_model.Lasso([alpha, fit_intercept, ...])</code>	Linear Model trained with L1 prior as regularizer (aka the Lasso)
<code>linear_model.LassoCV([eps, n_alphas, ...])</code>	Lasso linear model with iterative fitting along a regularization path
<code>linear_model.LassoLars([alpha, ...])</code>	Lasso model fit with Least Angle Regression a.k.a. Lars
<code>linear_model.LassoLarsCV([fit_intercept, ...])</code>	Cross-validated Lasso, using the LARS algorithm
<code>linear_model.LassoLarsIC([criterion, ...])</code>	Lasso model fit with Lars using BIC or AIC for model selection

Continued on next page

**Table 1.119 – continued from previous page**

linear_model.LinearRegression([...])	Ordinary least squares Linear Regression.
linear_model.LogisticRegression([penalty, ...])	Logistic Regression (aka logit, MaxEnt) classifier.
linear_model.MultiTaskLasso([alpha, ...])	Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.
linear_model.MultiTaskElasticNet([alpha, ...])	Multi-task ElasticNet model trained with L1/L2 mixed-norm as regularizer.
linear_model.OrthogonalMatchingPursuit([...])	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.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 model fitted by minimizing a regularized empirical loss function.
linear_model.SGDRegressor([loss, penalty, ...])	Linear model fitted by minimizing a regularized empirical loss function.

**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)

**Parameters** **X** : array, shape = (n\_samples, n\_features)

Training vectors.

**y** : array, shape = (n\_samples)

Target values for training vectors

**n\_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

wether 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

If True, the regressors X are normalized

**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.

## 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.])
```

## Attributes

<i>coef_</i>	array, shape = (n_features)	Coefficients of the regression model (mean of distribution)
<i>alpha_</i>	float	estimated precision of the noise.
<i>lambda_</i>	array, shape = (n_features)	estimated precisions of the weights.
<i>sigma_</i>	array, shape = (n_features, n_features)	estimated variance-covariance matrix of the weights
<i>scores_</i>	float	if computed, value of the objective function (to be maximized)

## Methods

<code>decision_function(X)</code>	Decision function of the linear model
<code>fit(X, y)</code>	Fit the ARDRegression model according to the given training data
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__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

**Parameters** `X` : numpy array of shape [n\_samples, n\_features]

**Returns** `C` : array, shape = [n\_samples]

Returns predicted values.

### `fit(X, y)`

Fit the ARDRegression model according to the given training data and parameters.

Iterative procedure to maximize the evidence

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Training vector, where n\_samples in the number of samples and n\_features is the number of features.

`y` : array, shape = [n\_samples]

Target values (integers)

**Returns** `self` : returns an instance of self.

### `get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep: boolean, optional` :

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

### `predict(X)`

Predict using the linear model

**Parameters** `X` : numpy array of shape [n\_samples, n\_features]

**Returns** `C` : array, shape = [n\_samples]

Returns predicted values.

### `score(X, y)`

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  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params** (\*\*params)

Set the parameters of the 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.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).

**Parameters** **X** : array, shape = (n\_samples, n\_features)

Training vectors.

**y** : array, shape = (length)

Target values for training vectors

**n\_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

wether 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 are normalized

**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.

## 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.])
```

## Attributes

<i>coef_</i>	array, shape = (n_features)	Coefficients of the regression model (mean of distribution)
<i>alpha_</i>	float	estimated precision of the noise.
<i>lambda_</i>	array, shape = (n_features)	estimated precisions of the weights.
<i>scores_</i>	float	if computed, value of the objective function (to be maximized)

## Methods

<code>decision_function(X)</code>	Decision function of the linear model
<code>fit(X, y)</code>	Fit the model
<code>get_params(deep=True)</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

---

`__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

**Parameters** **X** : numpy array of shape [n\_samples, n\_features]

**Returns** **C** : array, shape = [n\_samples]

Returns predicted values.

**fit(X, y)**

Fit the model

**Parameters** **X** : numpy array of shape [n\_samples,n\_features]

Training data

**y** : numpy array of shape [n\_samples]

Target values

**Returns self** : returns an instance of self.

**get\_params(deep=True)**

Get parameters for the estimator

**Parameters** **deep: boolean, optional** :

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

**predict(X)**

Predict using the linear model

**Parameters** **X** : numpy array of shape [n\_samples, n\_features]

**Returns** **C** : array, shape = [n\_samples]

Returns predicted values.

**score(X, y)**

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as  $(1 - \frac{u}{v})$ , where u is the regression sum of squares  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params(\*\*params)**

Set the parameters of the 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.ElasticNet

```
class sklearn.linear_model.ElasticNet(alpha=1.0, l1_ratio=0.5, fit_intercept=True, normalize=False, precompute='auto', max_iter=1000, copy_X=True, tol=0.0001, warm_start=False, positive=False, rho=None)
```

Linear Model trained with L1 and L2 prior as regularizer

Minimizes the objective function:

$$\frac{1}{2 * n\_samples} * ||y - Xw||^2_2 + \alpha * l1\_ratio * ||w||_1 + 0.5 * \alpha * (1 - l1\_ratio) * ||w||^2_2$$

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:

$$\alpha = a + b \text{ and } l1\_ratio = a / (a + b)$$

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`.

### Parameters

#### `alpha` : 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 in the scikit. 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  $0 \leq l1\_ratio \leq 1$ . 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

If True, the regressors `X` are normalized

#### `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.

#### `max_iter`: int, optional :

The maximum number of iterations

#### `copy_X` : boolean, optional, default False

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.

## Notes

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

## Attributes

<i>coef_</i>	array, shape = (n_features,)	parameter vector (w in the cost function formula)
<i>sparse_coef_</i>	scipy.sparse matrix, shape = (n_features, 1)	<i>sparse_coef_</i> is a readonly property derived from <i>coef_</i>
<i>intercept_</i>	float   array, shape = (n_targets,)	independent term in decision function.
<i>dual_gap_</i>	float	the current fit is guaranteed to be epsilon-suboptimal with epsilon := <i>dual_gap_</i>
<i>eps_</i>	float	<i>eps_</i> is used to check if the fit converged to the requested <i>tol</i>

## Methods

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

**\_\_init\_\_**(alpha=1.0, l1\_ratio=0.5, fit\_intercept=True, normalize=False, precompute='auto', max\_iter=1000, copy\_X=True, tol=0.0001, warm\_start=False, positive=False, rho=None)

**decision\_function**(X)

Decision function of the linear model

**Parameters** **X** : numpy array or scipy.sparse matrix of shape (n\_samples, n\_features)

**Returns** **T** : array, shape = (n\_samples,)

The predicted decision function

**fit**(X, y, Xy=None, coef\_init=None)

Fit model with coordinate descent

**Parameters** **X**: ndarray or scipy.sparse matrix, (n\_samples, n\_features) :

Data

**y: ndarray, shape = (n\_samples,) or (n\_samples, n\_targets) :**

Target

**Xy** : array-like, optional

$Xy = np.dot(X.T, y)$  that can be precomputed. It is useful only when the Gram matrix is precomputed.

**coef\_init: ndarray of shape n\_features or (n\_targets, n\_features) :**

The initial coefficients to warm-start the optimization

## 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 the estimator

**Parameters deep: boolean, optional :**

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

**predict (X)**

Predict using the linear model

**Parameters X** : numpy array of shape [n\_samples, n\_features]

**Returns C** : array, shape = [n\_samples]

Returns predicted values.

**score (X, y)**

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  $((y - y_{pred})^2).sum()$  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]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params (\*\*params)**

Set the parameters of the 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** :

**sparse\_coef\_**

sparse representation of the fitted coef

**sklearn.linear\_model.ElasticNetCV**

```
class sklearn.linear_model.ElasticNetCV(l1_ratio=0.5,    eps=0.001,    n_alphas=100,    al-
                                         phas=None,    fit_intercept=True,    normalize=False,
                                         precompute='auto',    max_iter=1000,    tol=0.0001,
                                         cv=None,    copy_X=True,    verbose=0,    n_jobs=1,
                                         rho=None)
```

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

**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 crossvalidation generator, optional

If an integer is passed, it is the number of fold (default 3). Specific crossvalidation objects can be passed, see 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.

**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:

$\text{alpha} = a + b$  and  $\text{l1\_ratio} = a / (a + b)$ .

## Attributes

<i>alpha_</i>	float	The amount of penalization choosen by cross validation
<i>l1_ratio_</i>	float	The compromise between l1 and l2 penalization choosen by cross validation
<i>coef_</i>	array, shape = (n_features,)	Parameter vector (w in the cost function formula),
<i>intercept_</i>	float	Independent term in the decision function.
<i>mse_path</i>	array, shape = (n_l1_ratio, n_alpha, n_folds)	Mean square error for the test set on each fold, varying l1_ratio and alpha.

## Methods

<code>decision_function(X)</code>	Decision function of the linear model
<code>fit(X, y)</code>	Fit linear model with coordinate descent
<code>get_params([deep])</code>	Get parameters for the estimator
<code>path(X, y[, l1_ratio, eps, n_alphas, ...])</code>	Compute Elastic-Net path with coordinate descent
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

```
__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, rho=None)
```

### `decision_function(X)`

Decision function of the linear model

**Parameters** `X` : numpy array of shape [n\_samples, n\_features]

**Returns** `C` : 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.

**Parameters** X : array-like, shape (n\_samples, n\_features)

Training data. Pass directly as fortran contiguous data to avoid unnecessary memory duplication

## y : ndarray, shape (n\_samples,) or (n\_samples, n\_targets)

Target values

**get\_params**(deep=True)

Get parameters for the estimator

**Parameters** deep: boolean, optional :

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

**static path**(X, y, l1\_ratio=0.5, eps=0.001, n\_alphas=100, alphas=None, precompute='auto', Xy=None, fit\_intercept=True, normalize=False, copy\_X=True, verbose=False, rho=None, \*\*params)

Compute Elastic-Net path with coordinate descent

The Elastic Net optimization function is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2 +
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

**Parameters** X : ndarray, shape = (n\_samples, n\_features)

Training data. Pass directly as fortran contiguous data to avoid unnecessary memory duplication

## y : ndarray, shape = (n\_samples,)

Target values

## l1\_ratio : float, optional

float between 0 and 1 passed to ElasticNet (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

$Xy = \text{np.dot}(X.T, y)$  that can be precomputed. It is useful only when the Gram matrix is precomputed.

**fit\_intercept** : bool

Fit or not an intercept

**normalize** : boolean, optional

If True, the regressors X are normalized

**copy\_X** : boolean, optional, default True

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

**verbose** : bool or integer

Amount of verbosity

**params** : kwargs

keyword arguments passed to the Lasso objects

**Returns** **models** : a list of models along the regularization path

#### See Also:

ElasticNet, ElasticNetCV

#### Notes

See examples/plot\_lasso\_coordinate\_descent\_path.py for an example.

**predict** ( $X$ )

Predict using the linear model

**Parameters**  $X$  : numpy array of shape [n\_samples, n\_features]

**Returns**  $C$  : array, shape = [n\_samples]

Returns predicted values.

**rho**

DEPRECATED: rho was renamed to `l1_ratio` and will be removed in 0.15

**score** ( $X, y$ )

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  $((y - y_{\text{pred}})^2).sum()$  and  $v$  is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters**  $X$  : array-like, shape = [n\_samples, n\_features]

Training set.

$y$  : array-like, shape = [n\_samples]

**Returns**  $z$  : float

**set\_params** (\*\**params*)

Set the parameters of the 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.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

**Parameters** `n_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

If True, the regressors X are normalized

**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.

**See Also:**

`lars_path`, `LarsCV`, `sklearn.decomposition.sparse_encode`

[http://en.wikipedia.org/wiki/Least\\_angle\\_regression](http://en.wikipedia.org/wiki/Least_angle_regression)

**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...]
```

## Attributes

<i>coef_path_</i>	array, shape = [n_features, n_alpha]	The varying values of the coefficients along the path. It is not present if the fit_path parameter is False.
<i>coef_</i>	array, shape = [n_features]	Parameter vector (w in the formulation formula).
<i>intercept_</i>	float	Independent term in decision function.

## Methods

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

**`__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

**Parameters** `X` : numpy array of shape [n\_samples, n\_features]

**Returns** `C` : array, shape = [n\_samples]

Returns predicted values.

**`fit(X, y, Xy=None)`**

Fit the model using X, y as training data.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Training data.

`y` : array-like, shape = [n\_samples] or [n\_samples, n\_targets]

Target values.

`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.

**Returns self** : object

returns an instance of self.

**`get_params(deep=True)`**

Get parameters for the estimator

**Parameters** `deep`: boolean, optional :

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

**predict**(*X*)

Predict using the linear model

**Parameters** **X** : numpy array of shape [n\_samples, n\_features]

**Returns** **C** : array, shape = [n\_samples]

Returns predicted values.

**score**(*X*, *y*)

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  $((y - y_{\text{pred}})^2).sum()$  and  $v$  is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns** **z** : float

**set\_params**(\*\*params)

Set the parameters of the 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.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

**Parameters** **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

If True, the regressors X are normalized

**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 : crossvalidation 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.

**See Also:**

`lars_path`, `LassoLars`, `LassoLarsCV`

**Attributes**

<code>coef_</code>	array, shape = [n_features]	parameter vector (w in the formulation formula)
<code>intercept_</code>	float	independent term in decision function
<code>coef_path_</code> : array, shape = [n_features, n_alpha]		the varying values of the coefficients along the path
<code>alpha_</code> : float		the estimated regularization parameter alpha
<code>alphas_</code> : array, shape = [n_alpha]		the different values of alpha along the path
<code>cv_alphas_</code> : array, shape = [n_cv_alphas]		all the values of alpha along the path for the different folds
<code>cv_mse_path_</code> : 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)

**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 the estimator

---

`predict(X)` Predict using the linear model

---

`score(X, y)` Returns the coefficient of determination R^2 of the prediction.

---

`set_params(**params)` Set the parameters of the 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` : numpy array of shape [n\_samples, n\_features]

**Returns C** : array, shape = [n\_samples]

Returns predicted values.

**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 the estimator

**Parameters deep: boolean, optional :**

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

**predict** (*X*)

Predict using the linear model

**Parameters X** : numpy array of shape [n\_samples, n\_features]

**Returns C** : array, shape = [n\_samples]

Returns predicted values.

**score** (*X*, *y*)

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  $((y - y_{\text{pred}})^2).sum()$  and *v* is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params** (\*\**params*)

Set the parameters of the 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.Lasso**

```
class sklearn.linear_model.Lasso(alpha=1.0, fit_intercept=True, normalize=False, precompute='auto', copy_X=True, max_iter=1000, tol=0.0001, warm_start=False, positive=False)
```

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).

**Parameters alpha** : float, optional

Constant that multiplies the L1 term. Defaults to 1.0 alpha = 0 is equivalent to an ordinary least square, solved by the LinearRegression object in the scikit. 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

If True, the regressors X are normalized

**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.

**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.

**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='auto', tol=0.0001,
      warm_start=False)
>>> print(clf.coef_)
[ 0.85  0.   ]
>>> print(clf.intercept_)
0.15
```

## Attributes

<i>coef_</i>	array, shape = (n_features,)	parameter vector ( $w$ in the cost function formula)
<i>sparse_coef_</i>	scipy.sparse matrix, shape = (n_features, 1)	<i>sparse_coef_</i> is a readonly property derived from <i>coef_</i>
<i>intercept_</i>	float	independent term in decision function.
<i>dual_gap_</i>	float	the current fit is guaranteed to be epsilon-suboptimal with epsilon := <i>dual_gap_</i>
<i>eps_</i>	float	<i>eps_</i> is used to check if the fit converged to the requested <i>tol</i>

## Methods

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

```
__init__(alpha=1.0, fit_intercept=True, normalize=False, precompute='auto', copy_X=True,
        max_iter=1000, tol=0.0001, warm_start=False, positive=False)
```

`decision_function(X)`

Decision function of the linear model

**Parameters** `X` : numpy array or scipy.sparse matrix of shape (n\_samples, n\_features)

**Returns** `T` : array, shape = (n\_samples,)

The predicted decision function

**fit** ( $X, y, Xy=None, coef\_init=None$ )

Fit model with coordinate descent

**Parameters X: ndarray or scipy.sparse matrix, (n\_samples, n\_features) :**

Data

**y: ndarray, shape = (n\_samples,) or (n\_samples, n\_targets) :**

Target

**Xy** : array-like, optional

$Xy = np.dot(X.T, y)$  that can be precomputed. It is useful only when the Gram matrix is precomputed.

**coef\_init: ndarray of shape n\_features or (n\_targets, n\_features) :**

The initial coefficients to warm-start the optimization

## 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 the estimator

**Parameters deep: boolean, optional :**

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

**predict (X)**

Predict using the linear model

**Parameters X :** numpy array of shape [n\_samples, n\_features]

**Returns C :** array, shape = [n\_samples]

Returns predicted values.

**score (X, y)**

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  $((y - y_{\text{pred}})^2).sum()$  and  $v$  is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters X :** array-like, shape = [n\_samples, n\_features]

Training set.

**y :** array-like, shape = [n\_samples]

**Returns z :** float

**set\_params (\*\*params)**

Set the parameters of the 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 :**

**sparse\_coef\_**  
sparse representation of the fitted coef

## 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)
```

Lasso linear model with iterative fitting along a regularization path

The best model is selected by cross-validation.

The optimization objective for Lasso is:

$$(1 / (2 * n\_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1$$

**Parameters** **eps** : float, optional

Length of the path. eps=1e-3 means that alpha\_min / alpha\_max = 1e-3.

**n\_alphas** : int, optional

Number of alphas along the regularization path

**alphas** : numpy array, optional

List of alphas where to compute the models. If None alphas are set automatically

**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 crossvalidation generator, optional

If an integer is passed, it is the number of fold (default 3). Specific crossvalidation objects can be passed, see sklearn.cross\_validation module for the list of possible objects

**verbose** : bool or integer

amount of verbosity

**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.

## Attributes

<code>alpha_</code> : float		The amount of penalization choosen by cross validation
<code>coef_</code>	array, shape = (n_features,)	parameter vector (w in the cost function formula)
<code>intercept_</code>	float	independent term in decision function.
<code>mse_path_</code> : array, shape = (n_alphas, n_folds)		mean square error for the test set on each fold, varying alpha
<code>alphas_</code> : numpy array		The grid of alphas used for fitting

## Methods

<code>decision_function(X)</code>	Decision function of the linear model
<code>fit(X, y)</code>	Fit linear model with coordinate descent
<code>get_params([deep])</code>	Get parameters for the estimator
<code>path(X, y[, eps, n_alphas, alphas, ...])</code>	Compute Lasso path with coordinate descent
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__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)`

**decision\_function(X)**

Decision function of the linear model

**Parameters X** : numpy array of shape [n\_samples, n\_features]

**Returns C** : 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.

**Parameters X** : array-like, shape (n\_samples, n\_features)

Training data. Pass directly as fortran contiguous data to avoid unnecessary memory duplication

**y** : ndarray, shape (n\_samples,) or (n\_samples, n\_targets)

Target values

**get\_params(deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

`static path(X, y, eps=0.001, n_alphas=100, alphas=None, precompute='auto', Xy=None, fit_intercept=True, normalize=False, copy_X=True, verbose=False, **params)`

Compute Lasso path with coordinate descent

The optimization objective for Lasso is:

$$(1 / (2 * n\_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1$$

**Parameters** **X** : ndarray, shape = (n\_samples, n\_features)

Training data. Pass directly as fortran contiguous data to avoid unnecessary memory duplication

**y** : ndarray, shape = (n\_samples, )

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.

**fit\_intercept** : bool

Fit or not an intercept

**normalize** : boolean, optional

If True, the regressors X are normalized

**copy\_X** : boolean, optional, default True

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

**verbose** : bool or integer

Amount of verbosity

**params** : kwargs

keyword arguments passed to the Lasso objects

**Returns** **models** : a list of models along the regularization path

**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.

### **predict**(X)

Predict using the linear model

**Parameters** **X** : numpy array of shape [n\_samples, n\_features]

**Returns** **C** : array, shape = [n\_samples]

Returns predicted values.

### **score**(X, y)

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  $((y - y_{\text{pred}})^2 \cdot \text{sum}()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2 \cdot \text{sum}()$ . Best possible score is 1.0, lower values are worse.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns** **z** : float

### **set\_params**(\*\*params)

Set the parameters of the 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.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

Constant that multiplies the penalty term. Defaults to 1.0. alpha = 0 is equivalent to an ordinary least square, solved by the LinearRegression object in the scikit. 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

If True, the regressors X are normalized

**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.

## See Also:

`lars_path`, `lasso_path`, `Lasso`, `LassoCV`, `LassoLarsCV`, `sklearn.decomposition.sparse_encode`

[http://en.wikipedia.org/wiki/Least\\_angle\\_regression](http://en.wikipedia.org/wiki/Least_angle_regression)

## 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...]
```

## Attributes

<code>coef_path_</code>	<code>array, shape = [n_features, n_alpha]</code>	The varying values of the coefficients along the path. It is not present if fit_path parameter is False.
<code>coef_</code>	<code>array, shape = [n_features]</code>	Parameter vector ( $w$ in the formulation formula).
<code>intercept_</code>	<code>float</code>	Independent term in decision function.

## Methods

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

<code>__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)</code>	
<b>decision_function(<math>X</math>)</b>	Decision function of the linear model
<b>Parameters</b> $X$ : numpy array of shape [n_samples, n_features]	
<b>Returns</b> $C$ : array, shape = [n_samples]	
Returns predicted values.	
<b>fit(<math>X, y, Xy=None</math>)</b>	
Fit the model using $X, y$ as training data.	
<b>Parameters</b> $X$ : array-like, shape = [n_samples, n_features]	
Training data.	
<b>y</b> : array-like, shape = [n_samples] or [n_samples, n_targets]	
Target values.	
<b>Xy</b> : 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.	
<b>Returns self</b> : object	
returns an instance of self.	
<b>get_params(deep=True)</b>	
Get parameters for the estimator	
<b>Parameters deep: boolean, optional :</b>	
If True, will return the parameters for this estimator and contained subobjects that are estimators.	
<b>predict(<math>X</math>)</b>	
Predict using the linear model	

**Parameters** **X** : numpy array of shape [n\_samples, n\_features]

**Returns** **C** : array, shape = [n\_samples]

Returns predicted values.

**score** (*X*, *y*)

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  $((y - y_{\text{pred}})^2 \cdot \text{sum}()$  and *v* is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2 \cdot \text{sum}()$ . Best possible score is 1.0, lower values are worse.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns** **z** : float

**set\_params** (\*\**params*)

Set the parameters of the 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.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
```

**Parameters** **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

If True, the regressors X are normalized

**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** : crossvalidation 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.

#### 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.

#### Attributes

<code>coef_</code>	array, shape = [n_features]	parameter vector (w in the formulation formula)
<code>intercept_</code>	float	independent term in decision function.
<code>coef_path_</code> : array, shape = [n_features, n_alpha]		the varying values of the coefficients along the path
<code>alpha_</code> : float		the estimated regularization parameter alpha
<code>alphas_</code> : array, shape = [n_alpha]		the different values of alpha along the path
<code>cv_alphas_</code> : array, shape = [n_cv_alphas]		all the values of alpha along the path for the different folds
<code>cv_mse_path_</code> : 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)

#### Methods

---

<code>decision_function(X)</code>	Decision function of the linear model
-----------------------------------	---------------------------------------

---

<code>fit(X, y)</code>	Fit the model using X, y as training data.
------------------------	--

---

Continued on next page
------------------------

**Table 1.129 – continued from previous page**

<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the 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 :** numpy array of shape [n\_samples, n\_features]

**Returns C :** array, shape = [n\_samples]

Returns predicted values.

**`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 the estimator

**Parameters deep: boolean, optional :**

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

**`predict(X)`**

Predict using the linear model

**Parameters X :** numpy array of shape [n\_samples, n\_features]

**Returns C :** array, shape = [n\_samples]

Returns predicted values.

**`score(X, y)`**

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  $((y - y_{\text{pred}})^2 \cdot \text{sum}()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2 \cdot \text{sum}()$ . Best possible score is 1.0, lower values are worse.

**Parameters X :** array-like, shape = [n\_samples, n\_features]

Training set.

**y :** array-like, shape = [n\_samples]

**Returns z :** float

**set\_params** (\*\*params)

Set the parameters of the 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.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.

**Parameters****criterion: ‘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

If True, the regressors X are normalized

**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.

**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/Akaike_information_criterion) [http://en.wikipedia.org/wiki/Bayesian\\_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...]
```

## Attributes

<i>coef_</i>	array, shape = [n_features]	parameter vector ( $w$ in the formulation formula)
<i>intercept_</i>	float	independent term in decision function.
<i>alpha_</i>	float	the alpha parameter chosen by the information criterion

## Methods

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

`__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

**Parameters** `X` : numpy array of shape [n\_samples, n\_features]

**Returns** `C` : array, shape = [n\_samples]

Returns predicted values.

`fit(X, y, copy_X=True)`

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 the estimator

**Parameters deep: boolean, optional :**

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

**predict (X)**

Predict using the linear model

**Parameters X** : numpy array of shape [n\_samples, n\_features]

**Returns C** : array, shape = [n\_samples]

Returns predicted values.

**score (X, y)**

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  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}.mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params (\*\*params)**

Set the parameters of the 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.LinearRegression

```
class sklearn.linear_model.LinearRegression(fit_intercept=True,           normalize=False,
                                            copy_X=True)
```

Ordinary least squares Linear Regression.

**Parameters fit\_intercept** : boolean, optional

wether 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 True, the regressors X are normalized

## Notes

From the implementation point of view, this is just plain Ordinary Least Squares (numpy.linalg.lstsq) wrapped as a predictor object.

## Attributes

<code>coef_</code>	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.
<code>intercept_</code>	array	Independent term in the linear model.

## Methods

<code>decision_function(X)</code>	Decision function of the linear model
<code>fit(X, y[, n_jobs])</code>	Fit linear model.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

**`__init__(fit_intercept=True, normalize=False, copy_X=True)`**

**`decision_function(X)`**

Decision function of the linear model

**Parameters** `X` : numpy array of shape [n\_samples, n\_features]

**Returns** `C` : array, shape = [n\_samples]

Returns predicted values.

**`fit(X, y, n_jobs=1)`**

Fit linear model.

**Parameters** `X` : numpy array or sparse matrix of shape [n\_samples,n\_features]

Training data

`y` : numpy array of shape [n\_samples, n\_targets]

Target values

`n_jobs` : 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

**Returns self** : returns an instance of self.

**`get_params(deep=True)`**

Get parameters for the estimator

**Parameters** `deep: boolean, optional :`

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

**predict**(*X*)

Predict using the linear model

**Parameters** **X** : numpy array of shape [n\_samples, n\_features]

**Returns** **C** : array, shape = [n\_samples]

Returns predicted values.

**score**(*X*, *y*)

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  $((y - y_{\text{pred}})^2).sum()$  and  $v$  is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns** **z** : float

**set\_params**(\*\*params)

Set the parameters of the 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.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)
```

Logistic Regression (aka logit, MaxEnt) classifier.

In the multiclass case, the training algorithm uses a one-vs.-all (OvA) scheme, rather than the “true” multinomial LR.

This class implements L1 and L2 regularized logistic regression using the *liblinear* library. 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).

**Parameters** **penalty** : string, ‘l1’ or ‘l2’

Used to specify the norm used in the penalization.

**dual** : boolean

Dual or primal formulation. Dual formulation is only implemented for l2 penalty. Prefer dual=False when n\_samples > n\_features.

**C** : float, optional (default=1.0)

Specifies the strength of the regularization. The smaller it is the bigger is the 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

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 11/12 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.

**tol: float, optional :**

Tolerance for stopping criteria.

#### See Also:

LinearSVC

#### 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.

References:

**LIBLINEAR – A Library for Large Linear Classification** <http://www.csie.ntu.edu.tw/~cjlin/liblinear/>

**Hsiang-Fu Yu, Fang-Lan Huang, Chih-Jen Lin (2011). Dual coordinate descent methods for logistic regression and maximum entropy models.** Machine Learning 85(1-2):41-75. [http://www.csie.ntu.edu.tw/~cjlin/papers/maxent\\_dual.pdf](http://www.csie.ntu.edu.tw/~cjlin/papers/maxent_dual.pdf)

#### Attributes

<i>coef_</i>	array, shape = [n_classes-1, n_features]	Coefficient of the features in the decision function. <i>coef_</i> is readonly property derived from <i>raw_coef_</i> that follows the internal memory layout of liblinear.
<i>intercept_</i>	array, shape = [n_classes-1]	Intercept (a.k.a. bias) added to the decision function. It is available only when parameter intercept is set to True.

#### Methods

`decision_function(X)` Predict confidence scores for samples.

`fit(X, y)` Fit the model according to the given training data.

`fit_transform(X[, y])` Fit to data, then transform it

Continued on next page

**Table 1.132 – continued from previous page**

<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class labels for samples in X.
<code>predict_log_proba(X)</code>	Log of probability estimates.
<code>predict_proba(X)</code>	Probability estimates.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

`__init__(penalty='l2', dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=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.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Samples.

**Returns** array, shape = [n\_samples] if n\_classes == 2 else [n\_samples,n\_classes] :

Confidence scores per (sample, class) combination. In the binary case, confidence score for the “positive” class.

**fit(X, y)**

Fit the model according to the given training data.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training vector, where n\_samples in the number of samples and n\_features is the number of features.

`y` : array-like, shape = [n\_samples]

Target vector relative to X

`class_weight` : {dict, ‘auto’}, optional

Weights associated with classes. If not given, all classes are supposed to have weight one.

**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.

**Returns** `X_new` : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**label\_**

DEPRECATED: The `labels_` attribute has been renamed to `classes_` for consistency and will be removed in 0.15.

**predict (X)**

Predict class labels for samples in X.

**Parameters X : {array-like, sparse matrix}, shape = [n\_samples, n\_features]**

Samples.

**Returns C : 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.

**Parameters X : array-like, shape = [n\_samples, n\_features]****Returns T : array-like, shape = [n\_samples, n\_classes]**

Returns the log-probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

**predict\_proba (X)**

Probability estimates.

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

**Parameters X : array-like, shape = [n\_samples, n\_features]****Returns T : 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)**

Returns the mean accuracy on the given test data and labels.

**Parameters X : array-like, shape = [n\_samples, n\_features]**

Training set.

**y : array-like, shape = [n\_samples]**

Labels for X.

**Returns z : float****set\_params (\*\*params)**

Set the parameters of the 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.

**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.

## 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)
```

Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer

The optimization objective for Lasso is:

$$(1 / (2 * n\_samples)) * ||Y - XW||^2\_Fro + alpha * ||W||\_21$$

Where:

$$||W||\_21 = \sum_i \sqrt{\sum_j w_{ij}^2}$$

i.e. the sum of norm of each row.

**Parameters** **alpha** : float, optional

Constant that multiplies the L1/L2 term. Defaults to 1.0

**fit\_intercept** : boolean

whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

**normalize** : boolean, optional

If True, the regressors X are normalized

**copy\_X** : boolean, optional, default True

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

**max\_iter** : int, optional

The maximum number of iterations

**tol** : float, optional

The tolerance for the optimization: if the updates are smaller than ‘tol’, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.

**warm\_start** : bool, optional

When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.

#### See 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, tol=0.0001, warm_start=False)
>>> print clf.coef_
[[ 0.89393398  0.        ]
 [ 0.89393398  0.        ]]
>>> print clf.intercept_
[ 0.10606602  0.10606602]
```

#### Attributes

<code>coef_</code>	array, shape = (n_tasks, n_features)	parameter vector (W in the cost function formula)
<code>intercept_</code>	array, shape = (n_tasks,)	independent term in decision function.

#### Methods

<code>decision_function(X)</code>	Decision function of the linear model
<code>fit(X, y[, Xy, coef_init])</code>	Fit MultiTaskLasso model with coordinate descent
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(alpha=1.0, fit_intercept=True, normalize=False, copy_X=True, max_iter=1000, tol=0.0001, warm_start=False)`

`decision_function(X)`

Decision function of the linear model

**Parameters** `X` : numpy array or scipy.sparse matrix of shape (n\_samples, n\_features)

**Returns** `T` : array, shape = (n\_samples,)

The predicted decision function

---

**fit** ( $X, y, Xy=None, coef\_init=None$ )  
Fit MultiTaskLasso model with coordinate descent

**Parameters X:** ndarray, shape = (n\_samples, n\_features) :  
Data

**y:** ndarray, shape = (n\_samples, n\_tasks) :  
Target

**coef\_init:** ndarray of shape n\_features :  
The initial coefficients to warm-start the optimization

**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 the estimator

**Parameters deep:** boolean, optional :

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

**predict** ( $X$ )  
Predict using the linear model

**Parameters X** : numpy array of shape [n\_samples, n\_features]

**Returns C** : array, shape = [n\_samples]

Returns predicted values.

**score** ( $X, y$ )  
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  $((y - y_{\text{pred}})^2).sum()$  and  $v$  is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}.mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params** (\*\*params)  
Set the parameters of the 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** :

**sparse\_coef\_**  
sparse representation of the fitted coef

**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, rho=None)
```

Multi-task ElasticNet model trained with L1/L2 mixed-norm as regularizer

The optimization objective for Lasso 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.

**Parameters** **alpha** : float, optional

Constant that multiplies the L1/L2 term. Defaults to 1.0

**l1\_ratio** : float

The ElasticNet mixing parameter, with  $0 < l1\_ratio \leq 1$ . 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

If True, the regressors X are normalized

**copy\_X** : boolean, optional, default True

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

**max\_iter** : int, optional

The maximum number of iterations

**tol** : float, optional

The tolerance for the optimization: if the updates are smaller than ‘tol’, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.

**warm\_start** : bool, optional

When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.

**See 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, rho=None, tol=0.0001,
                     warm_start=False)
>>> print clf.coef_
[[ 0.45663524  0.45612256]
 [ 0.45663524  0.45612256]]
>>> print clf.intercept_
[ 0.0872422  0.0872422]
```

## Attributes

<i>intercept_</i>	array, shape = (n_tasks,)	Independent term in decision function.
<i>coef_</i>	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), <i>coef_</i> is then a 1D array

## Methods

<code>decision_function(X)</code>	Decision function of the linear model
<code>fit(X, y[, Xy, coef_init])</code>	Fit MultiTaskLasso model with coordinate descent
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__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, rho=None)`

`decision_function(X)`

Decision function of the linear model

**Parameters** `X` : numpy array or scipy.sparse matrix of shape (n\_samples, n\_features)

**Returns** `T` : array, shape = (n\_samples,)

The predicted decision function

`fit(X, y, Xy=None, coef_init=None)`

Fit MultiTaskLasso model with coordinate descent

**Parameters X: ndarray, shape = (n\_samples, n\_features) :**

    Data

**y: ndarray, shape = (n\_samples, n\_tasks) :**

    Target

**coef\_init: ndarray of shape n\_features :**

    The initial coefficients to warm-start the optimization

## 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 the estimator

**Parameters deep: boolean, optional :**

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

**predict (X)**

    Predict using the linear model

**Parameters X : numpy array of shape [n\_samples, n\_features]**

**Returns C : array, shape = [n\_samples]**

    Returns predicted values.

**score (X, y)**

    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  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters X : array-like, shape = [n\_samples, n\_features]**

    Training set.

**y : array-like, shape = [n\_samples]**

**Returns z : float**

**set\_params (\*\*params)**

    Set the parameters of the 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 :**

**sparse\_coef\_**

    sparse representation of the fitted coef

**sklearn.linear\_model.OrthogonalMatchingPursuit**

```
class sklearn.linear_model.OrthogonalMatchingPursuit(copy_X=True, copy_Gram=True,
                                                    copy_Xy=True,
                                                    n_nonzero_coefs=None,
                                                    tol=None, fit_intercept=True,
                                                    normalize=True, precompute_gram=False)
```

Orthogonal Matching Pursuit model (OMP)

**Parameters**

**n\_nonzero\_coefs** : int, optional

Desired number of non-zero entries in the solution. If None (by default) this value is set to 10% of n\_features.

**tol** : float, optional

Maximum norm of the residual. If not None, overrides n\_nonzero\_coefs.

**fit\_intercept** : boolean, optional

whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

**normalize** : boolean, optional

If False, the regressors X are assumed to be already normalized.

**precompute\_gram** : {True, False, ‘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.

**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.

**copy\_Gram** : bool, optional

Whether the gram matrix must be copied by the algorithm. A false value is only helpful if X 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.

**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>)

## Attributes

<code>coef_</code>	array, shape = (n_features,) or (n_features, n_targets)	parameter vector (w in the formulation formula)
<code>intercept_</code>	float or array, shape =(n_targets,)	independent term in decision function.

## Methods

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

**`__init__(copy_X=True, copy_Gram=True, copy_Xy=True, n_nonzero_coefs=None, tol=None, fit_intercept=True, normalize=True, precompute_gram=False)`**

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

**Parameters** `X` : numpy array of shape [n\_samples, n\_features]

**Returns** `C` : array, shape = [n\_samples]  
Returns predicted values.

**`fit(X, y, Gram=None, Xy=None)`**  
Fit the model using X, y as training data.

**Parameters** `X`: array-like, shape = (n\_samples, n\_features) :  
Training data.

**y**: array-like, shape = (n\_samples,) or (n\_samples, n\_targets) :  
Target values.

**Gram**: array-like, shape = (n\_features, n\_features) (optional) :  
Gram matrix of the input data: `X.T * X`

**Xy**: array-like, shape = (n\_features,) or (n\_features, n\_targets) :  
(optional) Input targets multiplied by X: `X.T * y`

**Returns self: object** :  
returns an instance of self.

**`get_params(deep=True)`**  
Get parameters for the estimator

**Parameters** `deep`: boolean, optional :  
If True, will return the parameters for this estimator and contained subobjects that are estimators.

**`predict(X)`**  
Predict using the linear model

**Parameters** `X` : numpy array of shape [n\_samples, n\_features]

**Returns C :** array, shape = [n\_samples]

Returns predicted values.

**score (X, y)**

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as  $(1 - \text{u/v})$ , where u is the regression sum of squares  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters X :** array-like, shape = [n\_samples, n\_features]

Training set.

**y :** array-like, shape = [n\_samples]

**Returns z :** float

**set\_params (\*\*params)**

Set the parameters of the 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.PasSiveAggressiveClassifier

```
class sklearn.linear_model.PasSiveAggressiveClassifier(C=1.0, fit_intercept=True,
                                                       n_iter=5, shuffle=False,
                                                       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 :

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 :

Whether or not the training data should be shuffled after each epoch. Defaults to False.

**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.

#### See Also:

`SGDClassifier`, `Perceptron`

#### References

Online Passive-Aggressive Algorithms <<http://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06a.pdf>>  
K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, Y. Singer - JMLR (2006)

#### Attributes

<code>coef_</code>	array, shape = [1, n_features] if n_classes == 2 else [n_classes, n_features]	
<code>n_features</code>		Weights assigned to the features.
<code>intercept_</code>	array, shape = [1] if n_classes == 2 else [n_classes]	Constants in decision function.

#### Methods

<code>decision_function(X)</code>	Predict confidence scores for samples.
<code>fit(X, y[, coef_init, intercept_init])</code>	Fit linear model with Passive Aggressive algorithm.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>partial_fit(X, y[, classes])</code>	Fit linear model with Passive Aggressive algorithm.
<code>predict(X)</code>	Predict class labels for samples in X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(*args, **kwargs)</code>	

`__init__(C=1.0, fit_intercept=True, n_iter=5, shuffle=False, 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.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Samples.

**Returns** array, shape = [n\_samples] if n\_classes == 2 else [n\_samples,n\_classes] :

Confidence scores per (sample, class) combination. In the binary case, confidence score for the “positive” class.

**fit** ( $X, y, \text{coef\_init}=\text{None}, \text{intercept\_init}=\text{None}$ )

Fit linear model with Passive Aggressive algorithm.

**Parameters**  $X$  : {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 coeffients 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.

**Returns self** : returns an instance of self.

**get\_params** ( $deep=True$ )

Get parameters for the estimator

**Parameters**  $deep$ : boolean, optional :

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

**partial\_fit** ( $X, y, \text{classes}=\text{None}$ )

Fit linear model with Passive Aggressive algorithm.

**Parameters**  $X$  : {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  $classes$ .

**Returns self** : returns an instance of self.

**predict** ( $X$ )

Predict class labels for samples in  $X$ .

**Parameters**  $X$  : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Samples.

**Returns C** : array, shape = [n\_samples]

Predicted class label per sample.

**score** ( $X, y$ )

Returns the mean accuracy on the given test data and labels.

**Parameters**  $X$  : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns z** : float

**seed**

DEPRECATED: Parameter ‘seed’ was renamed to ‘random\_state’ for consistency and will be removed in 0.15

## `sklearn.linear_model.PASSIVEAGGRESSIVEREGRESSOR`

```
class sklearn.linear_model.PASSIVEAGGRESSIVEREGRESSOR(C=1.0,      fit_intercept=True,
                                                       n_iter=5,          shuffle=False,
                                                       fit=False,         verbose=0,
                                                       loss='epsilon_insensitive', epsilon=0.1, random_state=None,
                                                       class_weight=None, warm_start=False)
```

Passive Aggressive Regressor

**Parameters C** : float

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, optional :

Whether or not the training data should be shuffled after each epoch. Defaults to False.

**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.

**See Also:**

SGDRegressor

**References**

Online Passive-Aggressive Algorithms <<http://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06a.pdf>>  
 K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, Y. Singer - JMLR (2006)

**Attributes**

<i>coef_</i>	array, shape = [1, n_features] if n_classes == 2 else [n_classes, n_features]	
<i>n_features</i>		Weights assigned to the features.
<i>intercept_</i>	array, shape = [1] if n_classes == 2 else [n_classes]	Constants in decision function.

**Methods**

<code>decision_function(X)</code>	Predict using the linear model
<code>fit(X, y[, coef_init, intercept_init])</code>	Fit linear model with Passive Aggressive algorithm.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>partial_fit(X, y)</code>	Fit linear model with Passive Aggressive algorithm.
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(*args, **kwargs)</code>	

**`__init__(C=1.0, fit_intercept=True, n_iter=5, shuffle=False, 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

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

**Returns** array, shape = [n\_samples] :

Predicted target values per element in X.

**`fit(X, y, coef_init=None, intercept_init=None)`**

Fit linear model with Passive Aggressive algorithm.

**Parameters** `X` : {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.

**Returns** self : returns an instance of self.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**partial\_fit (X, y)**

Fit linear model with Passive Aggressive algorithm.

**Parameters X : {array-like, sparse matrix}, shape = [n\_samples, n\_features]**

Subset of training data

**y : numpy array of shape [n\_samples]**

Subset of target values

**Returns self** : returns an instance of self.**predict (X)**

Predict using the linear model

**Parameters X : {array-like, sparse matrix}, shape = [n\_samples, n\_features]****Returns array, shape = [n\_samples] :**

Predicted target values per element in X.

**score (X, y)**

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  $((y - y_{\text{pred}})^2 \cdot \text{sum}())$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2 \cdot \text{sum}())$ . Best possible score is 1.0, lower values are worse.**Parameters X : array-like, shape = [n\_samples, n\_features]**

Training set.

**y : array-like, shape = [n\_samples]****Returns z : float**

## sklearn.linear\_model.Perceptron

```
class sklearn.linear_model.Perceptron(penalty=None, alpha=0.0001, fit_intercept=True,
                                       n_iter=5, shuffle=False, verbose=0, eta0=1.0,
                                       n_jobs=1, random_state=0, class_weight=None,
                                       warm_start=False, seed=None)
```

Perceptron

**Parameters penalty : 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 :**

Whether or not the training data should be shuffled after each epoch. Defaults to False.

**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**

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.

**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.

**Attributes**

<code>coef_</code>	array, shape = [1, n_features] if n_classes == 2 else [n_classes,	
<code>n_features_</code>		Weights assigned to the features.
<code>intercept_</code>	array, shape = [1] if n_classes == 2 else [n_classes]	Constants in decision function.

## Methods

<code>decision_function(X)</code>	Predict confidence scores for samples.
<code>fit(X, y[, coef_init, intercept_init, ...])</code>	Fit linear model with Stochastic Gradient Descent.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>partial_fit(X, y[, classes, sample_weight])</code>	Fit linear model with Stochastic Gradient Descent.
<code>predict(X)</code>	Predict class labels for samples in X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(*args, **kwargs)</code>	
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

**`__init__(penalty=None, alpha=0.0001, fit_intercept=True, n_iter=5, shuffle=False, verbose=0, eta0=1.0, n_jobs=1, random_state=0, class_weight=None, warm_start=False, seed=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.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Samples.

**Returns array, shape = [n\_samples] if n\_classes == 2 else [n\_samples,n\_classes] :**

Confidence scores per (sample, class) combination. In the binary case, confidence score for the “positive” class.

**`fit(X, y, coef_init=None, intercept_init=None, class_weight=None, sample_weight=None)`**

Fit linear model with Stochastic Gradient Descent.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training data

`y` : 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.

`sample_weight` : array-like, shape = [n\_samples], optional

Weights applied to individual samples. If not provided, uniform weights are assumed.

**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 the estimator

**Parameters deep: boolean, optional :**  
If True, will return the parameters for this estimator and contained subobjects that are estimators.

**partial\_fit (X, y, classes=None, sample\_weight=None)**  
Fit linear model with Stochastic Gradient Descent.

**Parameters X** : {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`.

**sample\_weight** : array-like, shape = [n\_samples], optional  
Weights applied to individual samples. If not provided, uniform weights are assumed.

**Returns self** : returns an instance of self.

**predict (X)**  
Predict class labels for samples in X.

**Parameters X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]  
Samples.

**Returns C** : array, shape = [n\_samples]  
Predicted class label per sample.

**score (X, y)**  
Returns the mean accuracy on the given test data and labels.

**Parameters X** : array-like, shape = [n\_samples, n\_features]  
Training set.

**y** : array-like, shape = [n\_samples]  
Labels for X.

**Returns z** : float

**seed**  
DEPRECATED: Parameter ‘seed’ was renamed to ‘random\_state’ for consistency and will be removed in 0.15

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

Reduce X to its most important features.

**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.

**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=2.2204460492503131e-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.

**Parameters** **alpha** : float, ‘aic’, or ‘bic’

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

The alpha parameter in the stability selection article used to randomly scale the features. Should be between 0 and 1.

**sample\_fraction** : float

The fraction of samples to be used in each randomized design. Should be between 0 and 1. If 1, all samples are used.

**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

If True, the regressors X are normalized

**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.

**See Also:**

[RandomizedLogisticRegression](#), [LogisticRegression](#)

**Notes**

See examples/linear\_model/plot\_sparse\_recovery.py for an example.

**References**

Stability selection Nicolai Meinshausen, Peter Bühlmann 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()
```

## Attributes

<code>scores_</code>	array, shape = [n_features]	Feature scores between 0 and 1.
<code>all_scores_</code>	array, shape = [n_features, n_reg_parameter]	Feature scores between 0 and 1 for all values of the regularization parameter. The reference article suggests <code>scores_</code> is the max of <code>all_scores_</code> .

## Methods

<code>fit(X, y)</code>	Fit the model using X, y as training data.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>get_support([indices])</code>	Return a mask, or list, of the features/indices selected.
<code>inverse_transform(X)</code>	Transform a new matrix using the selected features
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Transform a new matrix using the selected features

  
| `__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.

**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.

`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 the estimator

**Parameters deep: boolean, optional :**

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

**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 the 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)**

Transform a new matrix using the selected features

## 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**

The regularization parameter C in the LogisticRegression.

**scaling : float**

The alpha parameter in the stability selection article used to randomly scale the features.  
Should be between 0 and 1.

**sample\_fraction : float**

The fraction of samples to be used in each randomized design. Should be between 0 and 1. If 1, all samples are used.

**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

If True, the regressors X are normalized

**tol** : float, optional

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.

#### See Also:

[RandomizedLasso](#), [Lasso](#), [ElasticNet](#)

#### Notes

See examples/linear\_model/plot\_randomized\_lasso.py for an example.

#### References

Stability selection Nicolai Meinshausen, Peter Bühlmann 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()
```

## Attributes

<code>scores_</code>	array, shape = [n_features]	Feature scores between 0 and 1.
<code>all_scores_</code>	array, shape = [n_features, n_reg_parameter]	Feature scores between 0 and 1 for all values of the regularization parameter. The reference article suggests <code>scores_</code> is the max of <code>all_scores_</code> .

## Methods

<code>fit(X, y)</code>	Fit the model using X, y as training data.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>get_support([indices])</code>	Return a mask, or list, of the features/indices selected.
<code>inverse_transform(X)</code>	Transform a new matrix using the selected features
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Transform a new matrix using the selected features

`__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.

**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.

**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 the estimator

**Parameters deep: boolean, optional :**

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

**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 the 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)**

Transform a new matrix using the selected features

## sklearn.linear\_model.Ridge

```
class sklearn.linear_model.Ridge(alpha=1.0, fit_intercept=True, normalize=False, copy_X=True,
                                  max_iter=None, tol=0.001, solver='auto')
```

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]).

**Parameters alpha : 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.

**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**

If True, the regressors X are normalized

**solver : {'auto', 'dense\_cholesky', 'lsqr', 'sparse\_cg'}**

Solver to use in the computational routines:

- ‘auto’ chooses the solver automatically based on the type of data.
- ‘dense\_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 ‘dense\_cholesky’ for large-scale data (possibility to set `tol` and `max_iter`).
- ‘lsqr’ uses the dedicated regularized least-squares routine `scipy.sparse.linalg.lsqr`. It is the fastest 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.

#### See Also:

`RidgeClassifier`, `RidgeCV`

#### 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)
```

#### Attributes

<code>coef_</code>	array, shape = [n_features] or [n_targets, n_features]	Weight vector(s).
--------------------	--	-------------------

#### Methods

<code>decision_function(X)</code>	Decision function of the linear model
<code>fit(X, y[, sample_weight, solver])</code>	Fit Ridge regression model
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

```
__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

**Parameters** **X** : numpy array of shape [n\_samples, n\_features]

**Returns** **C** : array, shape = [n\_samples]

Returns predicted values.

**fit** (*X*, *y*, *sample\_weight*=1.0, *solver*=None)

Fit Ridge regression model

**Parameters** **X** : {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

**Returns self** : returns an instance of self.

**get\_params** (*deep*=True)

Get parameters for the estimator

**Parameters** **deep**: boolean, optional :

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

**predict** (*X*)

Predict using the linear model

**Parameters** **X** : numpy array of shape [n\_samples, n\_features]

**Returns** **C** : array, shape = [n\_samples]

Returns predicted values.

**score** (*X*, *y*)

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  $((y - y_{\text{pred}})^2).sum()$  and  $v$  is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}.mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns** **z** : float

**set\_params** (\*\*params)

Set the parameters of the 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.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.

**Parameters alpha** : 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

If True, the regressors X are normalized

**solver** : {'auto', 'dense\_cholesky', 'lsqr', 'sparse\_cg'}

Solver to use in the computational routines. ‘dense\_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.

**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.

**Attributes**

<i>coef_</i>	array, shape = [n_features] or [n_classes, n_features]	Weight vector(s).
--------------	--	-------------------

## Methods

<code>decision_function(X)</code>	Predict confidence scores for samples.
<code>fit(X, y[, solver])</code>	Fit Ridge regression model.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class labels for samples in X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__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.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Samples.

**Returns array, shape = [n\_samples] if n\_classes == 2 else [n\_samples,n\_classes] :**

Confidence scores per (sample, class) combination. In the binary case, confidence score for the “positive” class.

### `fit(X, y, solver=None)`

Fit Ridge regression model.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples,n\_features]

Training data

`y` : array-like, shape = [n\_samples]

Target values

**Returns self** : returns an instance of self.

### `get_params(deep=True)`

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

### `predict(X)`

Predict class labels for samples in X.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Samples.

**Returns C** : array, shape = [n\_samples]

Predicted class label per sample.

### `score(X, y)`

Returns the mean accuracy on the given test data and labels.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns z** : float

**set\_params** (\*\*params)

Set the parameters of the 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.RidgeClassifierCV

```
class sklearn.linear_model.RidgeClassifierCV(alphas=array([ 0.1,    1.,   10.        ]),
                                             fit_intercept=True,           normalize=False,
                                             score_func=None, loss_func=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

If True, the regressors X are normalized

**score\_func: callable, optional** :

function that takes 2 arguments and compares them in order to evaluate the performance of prediction (big is good) if None is passed, the score of the estimator is maximized

**loss\_func: callable, optional** :

function that takes 2 arguments and compares them in order to evaluate the performance of prediction (small is good) if None is passed, the score of the estimator is maximized

**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.

**See Also:**

**Ridge**Ridge regression

**RidgeClassifier**Ridge classifier

**RidgeCV**Ridge regression with built-in cross validation

## Notes

For multi-class classification, n\_class classifiers are trained in a one-versus-all approach. Concretely, this is implemented by taking advantage of the multi-variate response support in Ridge.

## Attributes

<i>cv_values_</i>	array, shape = [n_samples, n_alphas] or shape = [n_samples, n_responses, n_alphas], optional	Cross-validation values for each alpha (if <i>store_cv_values=True</i> and
<i>cv=None</i> ). After <i>fit()</i> has been called, this attribute will contain the mean squared errors (by default) or the values of the <i>{loss,score}_func</i> function (if provided in the constructor).		
<i>coef_</i>	array, shape = [n_features] or [n_targets, n_features]	Weight vector(s).
<i>alpha_</i>	float	Estimated regularization parameter

## Methods

<code>decision_function(X)</code>	Predict confidence scores for samples.
<code>fit(X, y[, sample_weight, class_weight])</code>	Fit the ridge classifier.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class labels for samples in X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

**`__init__(alphas=array([ 0.1, 1., 10.]), fit_intercept=True, normalize=False, score_func=None, loss_func=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.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Samples.

**Returns** array, shape = [n\_samples] if n\_classes == 2 else [n\_samples,n\_classes] :

Confidence scores per (sample, class) combination. In the binary case, confidence score for the “positive” class.

**`fit(X, y, sample_weight=1.0, class_weight=None)`**

Fit the ridge classifier.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features.

**y** : array-like, shape = [n\_samples]

Target values.

**sample\_weight** : float or numpy array of shape [n\_samples]

Sample weight

**Returns self** : object

Returns self.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters** **deep**: boolean, optional :

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

**predict (X)**

Predict class labels for samples in X.

**Parameters** **X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Samples.

**Returns** **C** : array, shape = [n\_samples]

Predicted class label per sample.

**score (X, y)**

Returns the mean accuracy on the given test data and labels.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns** **z** : float

**set\_params (\*\*params)**

Set the parameters of the 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.RidgeCV

```
class sklearn.linear_model.RidgeCV(alphas=array([ 0.1,  1.,  10.]), fit_intercept=True, normalize=False, score_func=None, loss_func=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.

**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  $(2 \times C)^{-1}$  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`

If True, the regressors X are normalized

`score_func: callable, optional`:

function that takes 2 arguments and compares them in order to evaluate the performance of prediction (big is good) if None is passed, the score of the estimator is maximized

`loss_func: callable, optional`:

function that takes 2 arguments and compares them in order to evaluate the performance of prediction (small is good) if None is passed, the score of the estimator is maximized

`cv: cross-validation generator, optional`

If None, Generalized Cross-Validation (efficient Leave-One-Out) will be used.

`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, otherwise use eigen  
'svd' : force computation via singular value decomposition of X  
'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 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).

**See Also:**

`Ridge`Ridge regression

`RidgeClassifier`Ridge classifier

`RidgeClassifierCV`Ridge classifier with built-in cross validation

## Attributes

<code>cv_valuarray</code> , shape = [n_samples, n_alphas] or shape = [n_samples, n_targets, n_alphas], optional	Cross-validation values for each alpha (if <code>store_cv_values=True</code> and <code>cv=None</code> ). After <code>fit()</code> has been called, this attribute will contain the mean squared errors (by default) or the values of the <code>{loss,score}_func</code> function (if provided in the constructor).
<code>coef_</code>	array, shape = [n_features] or [n_targets, n_features]
<code>al-pha_</code>	float

## Methods

<code>decision_function(X)</code>	Decision function of the linear model
<code>fit(X, y[, sample_weight])</code>	Fit Ridge regression model
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(alphas=array([-0.1, 1., 10.]), fit_intercept=True, normalize=False, score_func=None, loss_func=None, cv=None, gcv_mode=None, store_cv_values=False)`

### `decision_function(X)`

Decision function of the linear model

**Parameters** `X` : numpy array of shape [n\_samples, n\_features]

**Returns** `C` : array, shape = [n\_samples]

Returns predicted values.

### `fit(X, y, sample_weight=1.0)`

Fit Ridge regression model

**Parameters** `X` : 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

**Returns self** : Returns self.

### `get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep`: boolean, optional :

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

**predict**(*X*)

Predict using the linear model

**Parameters** **X** : numpy array of shape [n\_samples, n\_features]

**Returns** **C** : array, shape = [n\_samples]

Returns predicted values.

**score**(*X*, *y*)

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as  $(1 - \frac{u}{v})$ , where u is the regression sum of squares  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}.mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns** **z** : float

**set\_params**(\*\*params)

Set the parameters of the 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.SGDClassifier**

```
class sklearn.linear_model.SGDClassifier(loss='hinge',      penalty='l2',      alpha=0.0001,
                                         l1_ratio=0.15,    fit_intercept=True,  n_iter=5,   shuffle=False,
                                         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, rho=None,    seed=None)
```

Linear model fitted by minimizing a regularized empirical loss with SGD.

SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

This implementation works with data represented as dense or sparse arrays of floating point values for the features.

**Parameters** **loss** : str, ‘hinge’ or ‘log’ or ‘modified\_huber’

The loss function to be used. Defaults to ‘hinge’. The hinge loss is a margin loss used by standard linear SVM models. The ‘log’ loss is the loss of logistic regression models and can be used for probability estimation in binary classifiers. ‘modified\_huber’ is another smooth loss that brings tolerance to outliers.

**penalty** : str, ‘l2’ or ‘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  $0 \leq l1\_ratio \leq 1$ .  $l1\_ratio=0$  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). Defaults to 5.

**shuffle: bool, optional** :

Whether or not the training data should be shuffled after each epoch. Defaults to False.

**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  $loss='huber'$  or  $loss='epsilon\_insensitive'$ . If the difference between the current prediction and the correct label is below this threshold, the model is not updated.

**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: constant: eta = eta0 optimal: eta =  $1.0/(t+t_0)$  [default] invscaling: eta = eta0 / pow(t, power\_t)

**eta0** : double

The initial learning rate [default 0.01].

**power\_t** : double

The exponent for inverse scaling learning rate [default 0.5].

**class\_weight** : dict, {class\_label}

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.

**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, 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, rho=None, shuffle=False,
    verbose=0, warm_start=False)
>>> print(clf.predict([[[-0.8, -1]]]))
[1]
```

**Attributes**

<i>coef_</i>	array, shape = [1, n_features] if n_classes == 2 else [n_classes, n_features]	Weights assigned to the features.
<i>intercept_</i>	array, shape = [1] if n_classes == 2 else [n_classes]	Constants in decision function.

**Methods**

<code>decision_function(X)</code>	Predict confidence scores for samples.
<code>fit(X, y[, coef_init, intercept_init, ...])</code>	Fit linear model with Stochastic Gradient Descent.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>partial_fit(X, y[, classes, sample_weight])</code>	Fit linear model with Stochastic Gradient Descent.
<code>predict(X)</code>	Predict class labels for samples in X.
<code>predict_log_proba(X)</code>	Log of probability estimates.
<code>predict_proba(X)</code>	Probability estimates.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(*args, **kwargs)</code>	
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

```
__init__(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, n_iter=5,
        shuffle=False, 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,
        rho=None, seed=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.

**Parameters** **X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Samples.

**Returns array, shape = [n\_samples] if n\_classes == 2 else [n\_samples,n\_classes] :**

Confidence scores per (sample, class) combination. In the binary case, confidence score for the “positive” class.

**fit** (*X*, *y*, *coef\_init=None*, *intercept\_init=None*, *class\_weight=None*, *sample\_weight=None*)  
Fit linear model with Stochastic Gradient Descent.

**Parameters** **X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training data

**y** : 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.

**sample\_weight** : array-like, shape = [n\_samples], optional

Weights applied to individual samples. If not provided, uniform weights are assumed.

**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 the estimator

**Parameters** **deep: boolean, optional** :

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

**partial\_fit** (*X*, *y*, *classes=None*, *sample\_weight=None*)

Fit linear model with Stochastic Gradient Descent.

**Parameters** **X** : {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`.

**sample\_weight** : array-like, shape = [n\_samples], optional

Weights applied to individual samples. If not provided, uniform weights are assumed.

**Returns self** : returns an instance of self.

**predict** (*X*)

Predict class labels for samples in *X*.

**Parameters** *X* : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Samples.

**Returns C** : array, shape = [n\_samples]

Predicted class label per sample.

**predict\_log\_proba** (*X*)

Log of probability estimates.

Log probability estimates are only supported for binary classification.

**Parameters** *X* : array-like, shape = [n\_samples, n\_features]

**Returns T** : array-like, shape = [n\_samples, n\_classes]

Returns the log-probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

**predict\_proba** (*X*)

Probability estimates.

Probability estimates are only supported for binary classification.

**Parameters** *X* : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

**Returns array, shape = [n\_samples, n\_classes] :**

Returns the probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

## References

The justification for the formula in the `loss="modified_huber"` case is in the appendix B in: <http://jmlr.csail.mit.edu/papers/volume2/zhang02c/zhang02c.pdf>

**score** (*X, y*)

Returns the mean accuracy on the given test data and labels.

**Parameters** *X* : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns z** : float

**seed**

DEPRECATED: Parameter ‘seed’ was renamed to ‘random\_state’ for consistency and will be removed in 0.15

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

Reduce X to its most important features.

**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.

## 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=False,
                                         verbose=0, epsilon=0.1, p=None, random_state=None,
                                         learning_rate='invscaling', eta0=0.01, power_t=0.25, warm_start=False,
                                         rho=None)
```

Linear model fitted by minimizing a regularized empirical loss with SGD

SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

This implementation works with data represented as dense numpy arrays of floating point values for the features.

**Parameters** loss : str, ‘squared\_loss’ or ‘huber’

The loss function to be used. Defaults to ‘squared\_loss’ which refers to the ordinary least squares fit. ‘huber’ is an epsilon insensitive loss function for robust regression.

**penalty** : str, ‘l2’ or ‘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  $0 \leq l1\_ratio \leq 1$ .  $l1\_ratio=0$  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). Defaults to 5.

**shuffle: bool, optional :**

Whether or not the training data should be shuffled after each epoch. Defaults to False.

**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  $loss=='huber'$  or  $loss='epsilon\_insensitive'$ . If the difference between the current prediction and the correct label is below this threshold, the model is not updated.

**learning\_rate : string, optional**

The learning rate: constant: eta = eta0 optimal: eta =  $1.0/(t+t_0)$  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.

**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, epsilon=0.1, eta0=0.01, fit_intercept=True,
             l1_ratio=0.15, learning_rate='invscaling', loss='squared_loss',
```

---

```
n_iter=5, p=None, penalty='l2', power_t=0.25, random_state=None,
rho=None, shuffle=False, verbose=0, warm_start=False)
```

## Attributes

<code>coef_</code>	array, shape = [n_features]	Weights assigned to the features.
<code>intercept_</code>	array, shape = [1]	The intercept term.

## Methods

<code>decision_function(X)</code>	Predict using the linear model
<code>fit(X, y[, coef_init, intercept_init, ...])</code>	Fit linear model with Stochastic Gradient Descent.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>partial_fit(X, y[, sample_weight])</code>	Fit linear model with Stochastic Gradient Descent.
<code>predict(X)</code>	Predict using the linear model
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(*args, **kwargs)</code>	
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

`__init__(loss='squared_loss', penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, n_iter=5, shuffle=False, verbose=0, epsilon=0.1, p=None, random_state=None, learning_rate='invscaling', eta0=0.01, power_t=0.25, warm_start=False, rho=None)`

### `decision_function(X)`

Predict using the linear model

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

**Returns array, shape = [n\_samples] :**

Predicted target values per element in X.

`fit(X, y, coef_init=None, intercept_init=None, sample_weight=None)`

Fit linear model with Stochastic Gradient Descent.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training data

`y` : 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.

`sample_weight` : array-like, shape = [n\_samples], optional

Weights applied to individual samples (1. for unweighted).

**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 the estimator

**Parameters** **deep**: boolean, optional :

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

**partial\_fit**(*X*, *y*, *sample\_weight=None*)

Fit linear model with Stochastic Gradient Descent.

**Parameters** **X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Subset of training data

**y** : numpy array of shape [n\_samples]

Subset of target values

**sample\_weight** : array-like, shape = [n\_samples], optional

Weights applied to individual samples. If not provided, uniform weights are assumed.

**Returns** **self** : returns an instance of self.**predict**(*X*)

Predict using the linear model

**Parameters** **X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]**Returns** array, shape = [n\_samples] :Predicted target values per element in *X*.**score**(*X*, *y*)

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 - 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]

Training set.

**y** : array-like, shape = [n\_samples]**Returns** **z** : float**transform**(*X*, *threshold=None*)Reduce *X* to its most important features.

**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.

<code>linear_model.lars_path(X, y[, Xy, Gram, ...])</code>	Compute Least Angle Regression and Lasso path
<code>linear_model.lasso_path(X, y[, eps, ...])</code>	Compute Lasso path with coordinate descent
<code>linear_model.lasso_stability_path(X, y[, ...])</code>	Stability path based on randomized Lasso estimates
<code>linear_model.orthogonal_mp(X, y[, ...])</code>	Orthogonal Matching Pursuit (OMP)
<code>linear_model.orthogonal_mp_gram(Gram, Xy[, ...])</code>	Gram Orthogonal Matching Pursuit (OMP)

## 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=2.2204460492503131e-16,
                           copy_Gram=True, verbose=0, return_path=True)
```

Compute Least Angle Regression and Lasso path

The optimization objective for Lasso is:

$$(1 / (2 * n\_samples)) * ||y - Xw||^2_2 + \alpha * ||w||_1$$

**Parameters** **X** : array, shape: (n\_samples, n\_features)

Input data.

**y** : array, shape: (n\_samples)

Input targets.

**max\_iter** : integer, optional

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 ( $X^T * X$ ), if ‘auto’, the Gram matrix is precomputed from the given `X`, if there are more samples than features.

**alpha\_min** : float, optional

Minimum correlation along the path. It corresponds to the regularization parameter `alpha` parameter in the Lasso.

**method** : {‘lar’, ‘lasso’}

Specifies the returned model. Select ‘lar’ for Least Angle Regression, ‘lasso’ for the Lasso.

**eps** : float, optional

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

**copy\_X** : bool

If False, X is overwritten.

**copy\_Gram** : bool

If False, Gram is overwritten.

**verbose** : int (default=0)

Controls output verbosity.

**Returns alphas:** array, shape: (max\_features + 1,):

Maximum of covariances (in absolute value) at each iteration.

**active:** array, shape (max\_features,):

Indices of active variables at the end of the path.

**coefs:** array, shape (n\_features, max\_features + 1):

Coefficients along the path

#### See Also:

lasso\_path, LassoLars, Lars, LassoLarsCV, LarsCV, sklearn.decomposition.sparse\_encode

#### Notes

- [http://en.wikipedia.org/wiki/Least-angle\\_regression](http://en.wikipedia.org/wiki/Least-angle_regression)
- [http://en.wikipedia.org/wiki/Lasso\\_\(statistics\)#LASSO\\_method](http://en.wikipedia.org/wiki/Lasso_(statistics)#LASSO_method)

### sklearn.linear\_model.lasso\_path

```
sklearn.linear_model.lasso_path(X, y, eps=0.001, n_alphas=100, alphas=None, precompute='auto', Xy=None, fit_intercept=True, normalize=False, copy_X=True, verbose=False, **params)
```

Compute Lasso path with coordinate descent

The optimization objective for Lasso is:

$$(1 / (2 * n\_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1$$

**Parameters** **X** : ndarray, shape = (n\_samples, n\_features)

Training data. Pass directly as fortran contiguous data to avoid unnecessary memory duplication

**y** : ndarray, shape = (n\_samples,)

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.

**fit\_intercept** : bool

Fit or not an intercept

**normalize** : boolean, optional

If True, the regressors X are normalized

**copy\_X** : boolean, optional, default True

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

**verbose** : bool or integer

Amount of verbosity

**params** : kwargs

keyword arguments passed to the Lasso objects

**Returns** **models** : a list of models along the regularization path

#### 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.

### `sklearn.linear_model.lasso_stability_path`

```
sklearn.linear_model.lasso_stability_path(X, y, scaling=0.5, ran-
                                          dom_state=None, n_resampling=200,
                                          n_grid=100, sample_fraction=0.75,
                                          eps=8.8817841970012523e-16, n_jobs=1,
                                          verbose=False)
```

Stability path based on randomized Lasso estimates

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

training data.

**y** : array-like, shape = [n\_samples]

target values.

**scaling** : float

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.RandomState, optional

The generator used to randomize the design.

**n\_resampling** : int

Number of randomized models.

**n\_grid** : int

Number of grid points. The path is linearly reinterpolated on a grid between 0 and 1 before computing the scores.

**sample\_fraction** : float

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

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

**Returns** **alphas\_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\_randomized\_lasso.py for an example.

## sklearn.linear\_model.orthogonal\_mp

```
sklearn.linear_model.orthogonal_mp(X, y, n_nonzero_coefs=None, tol=None, precompute_gram=False, copy_X=True)
```

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*:  $\operatorname{argmin} \|y - X\gamma\|^2$  subject to  $\|\gamma\|_0 \leq n_{\text{nonzero coefs}}$

When parametrized by error using the parameter *tol*:  $\operatorname{argmin} \|\gamma\|_0$  subject to  $\|y - X\gamma\|^2 \leq \text{tol}$

**Parameters** **X**: array, shape = (n\_samples, n\_features) :

Input data. Columns are assumed to have unit norm.

**y: array, shape = (n\_samples,) or (n\_samples, n\_targets) :**  
Input targets

**n\_nonzero\_coefs: int :**  
Desired number of non-zero entries in the solution. If None (by default) this value is set to 10% of n\_features.

**tol: float :**  
Maximum norm of the residual. If not None, overrides n\_nonzero\_coefs.

**precompute\_gram: {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.

**Returns coef: array, shape = (n\_features,) or (n\_features, n\_targets) :**

Coefficients of the OMP solution

#### 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>

## 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)
```

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.

**Parameters Gram: array, shape = (n\_features, n\_features) :**

Gram matrix of the input data: X.T \* X

**Xy: array, shape = (n\_features,) or (n\_features, n\_targets) :**

Input targets multiplied by X: X.T \* y

**n\_nonzero\_coefs: int :**

Desired number of non-zero entries in the solution. If None (by default) this value is set to 10% of n\_features.

**tol: float :**

Maximum norm of the residual. If not None, overrides n\_nonzero\_coefs.

**norms\_squared: array-like, shape = (n\_targets,) :**

Squared L2 norms of the lines of y. Required if tol is not None.

**copy\_Gram: bool, optional :**

Whether the gram matrix must be copied by the algorithm. A false value is only helpful if it is already Fortran-ordered, otherwise a copy is made anyway.

**copy\_Xy: bool, optional :**

Whether the covariance vector Xy must be copied by the algorithm. If False, it may be overwritten.

**Returns coef: array, shape = (n\_features,) or (n\_features, n\_targets) :**

Coefficients of the OMP solution

**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>

## 1.8.18 sklearn.manifold: Manifold Learning

The `sklearn.manifold` module implements data embedding techniques.

**User guide:** See the *Manifold learning* section for further details.

<code>manifold.LocallyLinearEmbedding([...])</code>	Locally Linear Embedding
<code>manifold.Isomap([n_neighbors, n_components, ...])</code>	Isomap Embedding
<code>manifold.MDS([n_components, metric, n_init, ...])</code>	Multidimensional scaling
<code>manifold.SpectralEmbedding([n_components, ...])</code>	Spectral Embedding for Non-linear Dimensionality Reduction.

## 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

### Parameters

**n\_neighbors** : integer

number of neighbors to consider for each point.

**n\_components** : integer

number of coordinates for the manifold

**reg** : float

regularization constant, multiplies the trace of the local covariance matrix of the distances.

**eigen\_solver** : string, {‘auto’, ‘arpack’, ‘dense’}

**auto** : algorithm will attempt to choose the best method for input data

**arpack**[use arnoldi iteration in shift-invert mode.] For this method, M may be a dense matrix, sparse matrix, or general linear operator. Warning: ARPACK can be unstable for some problems. It is best to try several random seeds in order to check results.

**dense**[use standard dense matrix operations for the eigenvalue] decomposition. For this method, M must be an array or matrix type. This method should be avoided for large problems.

**tol** : float, optional

Tolerance for ‘arpack’ method Not used if eigen\_solver==‘dense’.

**max\_iter** : integer

maximum number of iterations for the arpack solver. Not used if eigen\_solver==‘dense’.

**method** : string [‘standard’ | ‘hessian’ | ‘modified’ | ‘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.

## References

[R88], [R89], [R90], [R91]

## Attributes

<i>embedding_vectors_</i>	array-like, shape [n_components, n_samples]	Stores the embedding vectors
<i>reconstruction_error_</i>	float	Reconstruction error associated with <i>embedding_vectors_</i>
<i>nbrs_</i>	NearestNeighbors object	Stores nearest neighbors instance, including BallTree or KDtree if applicable.

## Methods

<code>fit(X[, y])</code>	Compute the embedding vectors for data X
<code>fit_transform(X[, y])</code>	Compute the embedding vectors for data X and transform X.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Transform new points into embedding space.

`__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)`

**fit (X, y=None)**

Compute the embedding vectors for data X

**Parameters** **X** : array-like of shape [n\_samples, n\_features]

training set.

**Returns** **self** : returns an instance of self.

**fit\_transform (X, y=None)**

Compute the embedding vectors for data X and transform X.

**Parameters** **X** : array-like of shape [n\_samples, n\_features]

training set.

**Returns** **X\_new: array-like, shape (n\_samples, n\_components) :**

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters** **deep: boolean, optional :**

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

**set\_params**(*\*\*params*)

Set the parameters of the 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*)

Transform new points into embedding space.

**Parameters** *X* : array-like, shape = [n\_samples, n\_features]

**Returns** *X\_new* : array, shape = [n\_samples, n\_components]

**Notes**

Because of scaling performed by this method, it is discouraged to use it together with methods that are not scale-invariant (like SVMs)

**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

**Parameters** *n\_neighbors* : integer

number of neighbors to consider for each point.

**n\_components** : integer

number of coordinates for the manifold

**eigen\_solver** : ['auto'|'arpack'|'dense']

'auto'[Attempt to choose the most efficient solver] for the given problem.

'arpack'[Use Arnoldi decomposition to find the eigenvalues] and eigenvectors.

'dense'[Use a direct solver (i.e. LAPACK)] for the eigenvalue decomposition.

**tol** : float

Convergence tolerance passed to arpack or lobpcg. not used if eigen\_solver == 'dense'.

**max\_iter** : integer

Maximum number of iterations for the arpack solver. not used if eigen\_solver == 'dense'.

**path\_method** : string ['auto'|'FW'|'D']

Method to use in finding shortest path. 'auto' : attempt to choose the best algorithm automatically 'FW' : Floyd-Warshall algorithm 'D' : Dijkstra algorithm with Fibonacci Heaps

**neighbors\_algorithm** : string ['auto'|'brute'|'kd\_tree'|'ball\_tree']

Algorithm to use for nearest neighbors search, passed to neighbors.NearestNeighbors instance.

## References

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

## Attributes

<code>embedding_</code>	array-like, shape (n_samples, n_components)	Stores the embedding vectors.
<code>kernel_pca_</code>	object	<i>KernelPCA</i> object used to implement the embedding.
<code>training_data_</code>	array-like, shape (n_samples, n_features)	Stores the training data.
<code>nbs_</code>	sklearn.neighbors.NearestNeighbors instance	Stores nearest neighbors instance, including BallTree or KDtree if applicable.
<code>dist_matrix_</code>	array-like, shape (n_samples, n_samples)	Stores the geodesic distance matrix of training data.

## Methods

<code>fit(X[, y])</code>	Compute the embedding vectors for data X
<code>fit_transform(X[, y])</code>	Fit the model from data in X and transform X.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>reconstruction_error()</code>	Compute the reconstruction error for the embedding.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Transform X.

<code>__init__(n_neighbors=5, n_components=2, eigen_solver='auto', tol=0, max_iter=None, path_method='auto', neighbors_algorithm='auto')</code>	
<code>fit(X, y=None)</code>	Compute the embedding vectors for data X

**Parameters** `X` : {array-like, sparse matrix, BallTree, cKDTree, NearestNeighbors}

Sample data, shape = (n\_samples, n\_features), in the form of a numpy array, precomputed tree, or NearestNeighbors object.

**Returns** `self` : returns an instance of self.

<code>fit_transform(X, y=None)</code>	
	Fit the model from data in X and transform X.

**Parameters** `X`: {array-like, sparse matrix, BallTree, cKDTree} :

Training vector, where n\_samples in the number of samples and n\_features is the number of features.

**Returns** `X_new`: array-like, shape (n\_samples, n\_components) :

<code>get_params(deep=True)</code>	
	Get parameters for the estimator

**Parameters** `deep`: boolean, optional :

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

#### **reconstruction\_error()**

Compute the reconstruction error for the embedding.

**Returns** `reconstruction_error` : float

#### Notes

The cost function of an isomap embedding is

```
E = frobenius_norm[K(D) - K(D_fit)] / n_samples
```

Where D is the matrix of distances for the input data X, D\_fit is the matrix of distances for the output embedding X\_fit, and K is the isomap kernel:

```
K(D) = -0.5 * (I - 1/n_samples) * D^2 * (I - 1/n_samples)
```

#### **set\_params(\*\*params)**

Set the parameters of the 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)**

Transform X.

This is implemented by linking the points X into the graph of geodesic distances of the training data. First the *n\_neighbors* nearest neighbors of X are found in the training data, and from these the shortest geodesic distances from each point in X to each point in the training data are computed in order to construct the kernel. The embedding of X is the projection of this kernel onto the embedding vectors of the training set.

**Parameters** `X`: array-like, shape (n\_samples, n\_features) :

**Returns** `X_new`: array-like, shape (n\_samples, n\_components) :

## 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

**Parameters** `metric` : 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.

`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’.

## 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)

## Attributes

embedding	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)

## Methods

<code>fit(X[, init, y])</code>	Computes the position of the points in the embedding space
<code>fit_transform(X[, init, y])</code>	Fit the data from X, and returns the embedded coordinates
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__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')`

`fit(X, init=None, y=None)`

Computes the position of the points in the embedding space

**Parameters X** : array, shape=[n\_samples, n\_features]

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.

**fit\_transform**(X, init=None, y=None)

Fit the data from X, and returns the embedded coordinates

**Parameters X** : array, shape=[n\_samples, n\_features]

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 the estimator

**Parameters deep: boolean, optional** :

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

**set\_params**(\*\*params)

Set the parameters of the 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.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.

**Parameters n\_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.

**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.

### 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>

### Attributes

<i>embedding_</i>	array, shape = (n_samples, n_components)	Spectral embedding of the training matrix.
<i>affinity_matrix_</i>	array, shape = (n_samples, n_samples)	Affinity_matrix constructed from samples or precomputed.

### Methods

<code>fit(X[, y])</code>	Fit the model from data in X.
<code>fit_transform(X[, y])</code>	Fit the model from data in X and transform X.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(n_components=2, affinity='nearest_neighbors', gamma=None, random_state=None, eigen_solver=None, n_neighbors=None)`

**fit**(X, y=None)

Fit the model from data in X.

**Parameters** **X** : array-like, shape (n\_samples, n\_features)

Training vector, where n\_samples in the number of samples and n\_features is the number of features.

If affinity is “precomputed” X : array-like, shape (n\_samples, n\_samples), Interpret X as precomputed adjacency graph computed from samples.

**Returns self** : object

Returns the instance itself.

**fit\_transform**(X, y=None)

Fit the model from data in X and transform X.

**Parameters X: array-like, shape (n\_samples, n\_features) :**

Training vector, where n\_samples is the number of samples and n\_features is the number of features.

If affinity is “precomputed” X : array-like, shape (n\_samples, n\_samples), Interpret X as precomputed adjacency graph computed from samples.

**Returns X\_new: array-like, shape (n\_samples, n\_components) :**

**get\_params**(deep=True)

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**set\_params**(\*\*params)

Set the parameters of the 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 :**

---

<code>manifold.locally_linear_embedding(X, ...[, ...])</code>	Perform a Locally Linear Embedding analysis on the data.
---	--

---

<code>manifold.spectral_embedding(adjacency[, ...])</code>	Project the sample on the first eigen vectors of the graph Laplacian.
--	---

---

## 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.

**Parameters X** : {array-like, sparse matrix, BallTree, cKDTree, NearestNeighbors}

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

**arpac**[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 ‘arpac’ 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 [R92]

**hessian**[use the Hessian eigenmap method. This method requires] n\_neighbors > n\_components \* (1 + (n\_components + 1) / 2. see reference [R93]

**modified**[use the modified locally linear embedding algorithm.] see reference [R94]

**ltsa**[use local tangent space alignment algorithm] see reference [R95]

**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.

**Returns** **Y** : array-like, shape [n\_samples, n\_components]

Embedding vectors.

**squared\_error** : float

Reconstruction error for the embedding vectors. Equivalent to `norm(Y - W * Y, 'fro') ** 2`, where W are the reconstruction weights.

## References

[R92], [R93], [R94], [R95]

## 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, mode=None)
```

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.

**Parameters** `adjacency` : array-like or sparse matrix, shape: (n\_samples, n\_samples)

The adjacency matrix of the graph to embed.

`n_components` : integer, optional

The dimension of the projection subspace.

`eigen_solver` : {None, ‘arpack’, ‘lobpcg’, or ‘amg’}

The eigenvalue decomposition strategy to use. AMG requires pyamg to be installed. It can be faster on very large, sparse problems, but may also lead to instabilities.

`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.

**Returns** `embedding` : 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>

## 1.8.19 `sklearn.metrics`: Metrics

See the *Model evaluation* section and the *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.

### Classification metrics

<code>metrics.accuracy_score(y_true, y_pred)</code>	Accuracy classification score
<code>metrics.auc(x, y[, reorder])</code>	Compute Area Under the Curve (AUC) using the trapezoidal rule
<code>metrics.auc_score(y_true, y_score)</code>	Compute Area Under the Curve (AUC) from prediction scores
<code>metrics.average_precision_score(y_true, y_score)</code>	Compute average precision (AP) from prediction scores
<code>metrics.classification_report(y_true, y_pred)</code>	Build a text report showing the main classification metrics
<code>metrics.confusion_matrix(y_true, y_pred[, ...])</code>	Compute confusion matrix to evaluate the accuracy of a classification
<code>metrics.f1_score(y_true, y_pred[, labels, ...])</code>	Compute the F1 score, also known as balanced F-score or F-measure
<code>metrics.fbeta_score(y_true, y_pred, beta[, ...])</code>	Compute the F-beta score
<code>metrics.hinge_loss(y_true, pred_decision[, ...])</code>	Average hinge loss (non-regularized)
<code>metrics.matthews_corrcoef(y_true, y_pred)</code>	Compute the Matthews correlation coefficient (MCC) for binary classification
<code>metrics.precision_recall_curve(y_true, ...)</code>	Compute precision-recall pairs for different probability thresholds
<code>metrics.precision_recall_fscore_support(...)</code>	Compute precision, recall, F-measure and support for each class
<code>metrics.precision_score(y_true, y_pred[, ...])</code>	Compute the precision
<code>metrics.recall_score(y_true, y_pred[, ...])</code>	Compute the recall
<code>metrics.roc_curve(y_true, y_score[, pos_label])</code>	Compute Receiver operating characteristic (ROC)
<code>metrics.zero_one_loss(y_true, y_pred[, ...])</code>	Zero-One classification loss

### `sklearn.metrics.accuracy_score`

`sklearn.metrics.accuracy_score(y_true, y_pred)`  
Accuracy classification score

**Parameters** `y_true` : array-like, shape = n\_samples

Ground truth (correct) labels.

`y_pred` : array-like, shape = n\_samples

Predicted labels, as returned by a classifier.

**Returns** `score` : float

The fraction of correct predictions in `y_pred`. The best performance is 1.

**See Also:**

`zero_one_loss`

### Examples

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]
>>> accuracy_score(y_true, y_pred)
0.5
```

## 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 `auc_score`.

**Parameters** `x` : array, shape = [n]

x coordinates.

`y` : array, shape = [n]

y coordinates.

`reorder` : boolean, optional

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.

**Returns** `auc` : float

**See Also:**

`auc_score`

## 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
```

## sklearn.metrics.auc\_score

```
sklearn.metrics.auc_score(y_true, y_score)
```

Compute Area Under the Curve (AUC) from prediction scores

Note: this implementation is restricted to the binary classification task.

**Parameters** `y_true` : array, shape = [n\_samples]

True binary labels.

`y_score` : array, shape = [n\_samples]

Target scores, can either be probability estimates of the positive class, confidence values, or binary decisions.

**Returns** `auc` : float

**See Also:**

`average_precision_score`Area under the precision-recall curve

## References

[http://en.wikipedia.org/wiki/Receiver\\_operating\\_characteristic](http://en.wikipedia.org/wiki/Receiver_operating_characteristic)

## Examples

```
>>> import numpy as np
>>> from sklearn.metrics import auc_score
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> auc_score(y_true, y_scores)
0.75
```

## sklearn.metrics.average\_precision\_score

`sklearn.metrics.average_precision_score(y_true, y_score)`

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.

**Parameters** `y_true` : array, shape = [n\_samples]

True binary labels.

`y_score` : array, shape = [n\_samples]

Target scores, can either be probability estimates of the positive class, confidence values, or binary decisions.

**Returns** `average_precision` : float

**See Also:**

`auc_score` Area under the ROC curve

## References

[http://en.wikipedia.org/wiki/Information\\_retrieval#Average\\_precision](http://en.wikipedia.org/wiki/Information_retrieval#Average_precision)

## 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...
```

**sklearn.metrics.classification\_report**

```
sklearn.metrics.classification_report(y_true, y_pred, labels=None, target_names=None)
```

Build a text report showing the main classification metrics

**Parameters** `y_true` : array, shape = [n\_samples]

Ground truth (correct) target values.

`y_pred` : array, shape = [n\_samples]

Estimated targets as returned by a classifier.

`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).

**Returns** `report` : 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, 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
```

**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  $j$ .

**Parameters** `y_true` : array, shape = [n\_samples]

Ground truth (correct) target values.

`y_pred` : array, shape = [n\_samples]

Estimated targets as returned by a classifier.

`labels` : array, shape = [n\_classes]

List of all labels occurring in the dataset. If none is given, those that appear at least once in `y_true` or `y_pred` are used.

**Returns** `C` : array, shape = [n\_classes, n\_classes]

Confusion matrix

## References

[http://en.wikipedia.org/wiki/Confusion\\_matrix](http://en.wikipedia.org/wiki/Confusion_matrix)

## 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]])
```

## sklearn.metrics.f1\_score

`sklearn.metrics.f1_score(y_true, y_pred, labels=None, pos_label=1, average='weighted')`

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 case, this is the weighted average of the F1 score of each class.

**Parameters** `y_true` : array, shape = [n\_samples]

Ground truth (correct) target values.

`y_pred` : array, shape = [n\_samples]

Estimated targets as returned by a classifier.

`labels` : array

Integer array of labels.

`pos_label` : int

In the binary classification case, give the label of the positive class (default is 1). Everything else but `pos_label` is considered to belong to the negative class. Set to `None` in the case of multiclass classification.

`average` : string, [None, ‘micro’, ‘macro’, ‘weighted’ (default)]

In the multiclass classification case, this determines the type of averaging performed on the data.

**None**:Do not perform any averaging, return the score for each class.

**‘macro’**:Average over classes (does not take imbalance into account).

**‘micro’**:Average over instances (takes imbalance into account). This implies that `precision == recall == F1`.

**'weighted'**: Average weighted by support (takes imbalance into account). Can result in F-score that is not between precision and recall.

**Returns f1\_score** : float or array of float, shape = [n\_unique\_labels]

F1 score of the positive class in binary classification or weighted average of the F1 scores of each class for the multiclass task.

## References

[http://en.wikipedia.org/wiki/F1\\_score](http://en.wikipedia.org/wiki/F1_score)

## Examples

In the binary case:

```
>>> from sklearn.metrics import f1_score
>>> y_pred = [0, 1, 0, 0]
>>> y_true = [0, 1, 0, 1]
>>> f1_score(y_true, y_pred)
0.666...
```

In the multiclass case:

```
>>> 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. ])
```

## sklearn.metrics.fbeta\_score

`sklearn.metrics.fbeta_score(y_true, y_pred, beta, labels=None, pos_label=1, average='weighted')`

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 precision (*beta* == 0 considers only precision, *beta* == inf only recall).

**Parameters** `y_true` : array, shape = [n\_samples]

Ground truth (correct) target values.

`y_pred` : array, shape = [n\_samples]

Estimated targets as returned by a classifier.

`beta`: float :

Weight of precision in harmonic mean.

**labels** : array

Integer array of labels.

**pos\_label** : int

In the binary classification case, give the label of the positive class (default is 1). Everything else but `pos_label` is considered to belong to the negative class. Set to `None` in the case of multiclass classification.

**average** : string, [None, ‘micro’, ‘macro’, ‘weighted’ (default)]

In the multiclass classification case, this determines the type of averaging performed on the data.

**None**:Do not perform any averaging, return the scores for each class.

**‘macro’**:Average over classes (does not take imbalance into account).

**‘micro’**:Average over instances (takes imbalance into account). This implies that `precision == recall == F1`.

**‘weighted’**:Average weighted by support (takes imbalance into account). Can result in F-score that is not between precision and recall. Do not perform any averaging, return the score for each class.

**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

R. Baeza-Yates and B. Ribeiro-Neto (2011). Modern Information Retrieval. Addison Wesley, pp. 327-328.

[http://en.wikipedia.org/wiki/F1\\_score](http://en.wikipedia.org/wiki/F1_score)

## Examples

In the binary case:

```
>>> from sklearn.metrics import fbeta_score
>>> y_pred = [0, 1, 0, 0]
>>> y_true = [0, 1, 0, 1]
>>> fbeta_score(y_true, y_pred, beta=0.5)
0.83...
>>> fbeta_score(y_true, y_pred, beta=1)
0.66...
>>> fbeta_score(y_true, y_pred, beta=2)
0.55...
```

In the multiclass case:

```
>>> 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.hinge\_loss

`sklearn.metrics.hinge_loss(y_true, pred_decision, pos_label=1, neg_label=-1)`  
 Average hinge loss (non-regularized)

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.

**Parameters** `y_true` : array, shape = [n\_samples]

True target (integers).

`pred_decision` : array, shape = [n\_samples] or [n\_samples, n\_classes]

Predicted decisions, as output by `decision_function` (floats).

**Returns** `loss` : float

## References

[http://en.wikipedia.org/wiki/Hinge\\_loss](http://en.wikipedia.org/wiki/Hinge_loss)

## 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='l2', 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...

```

## 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.

**Parameters** `y_true` : array, shape = [n\_samples]

Ground truth (correct) target values.

`y_pred` : array, shape = [n\_samples]

Estimated targets as returned by a classifier.

**Returns** `mcc` : float

The Matthews correlation coefficient (+1 represents a perfect prediction, 0 an average random prediction and -1 and inverse prediction).

## References

[R100], [R101]

## 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)`

Compute precision-recall pairs for different probability thresholds

Note: this implementation is restricted to the binary classification task.

The precision is the ratio  $\text{tp} / (\text{tp} + \text{fp})$  where  $\text{tp}$  is the number of true positives and  $\text{fp}$  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  $\text{tp} / (\text{tp} + \text{fn})$  where  $\text{tp}$  is the number of true positives and  $\text{fn}$  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.

**Parameters** `y_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.

**Returns** `precision` : array, shape = [n + 1]

Precision values.

**recall** : array, shape = [n + 1]

Recall values.

**thresholds** : array, shape = [n]

Thresholds on y\_score 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, 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 ])
```

## 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)
```

Compute precision, recall, F-measure and support for each class

The precision is the ratio  $\frac{tp}{tp + fp}$  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  $\frac{tp}{tp + fn}$  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, this function returns the average precision, recall and F-measure if average is one of 'micro', 'macro', 'weighted'.

**Parameters** **y\_true** : array, shape = [n\_samples]

Ground truth (correct) target values.

**y\_pred** : array, shape = [n\_samples]

Estimated targets as returned by a classifier.

**beta** : float, 1.0 by default

The strength of recall versus precision in the F-score.

**labels** : array

Integer array of labels.

**pos\_label** : int

In the binary classification case, give the label of the positive class (default is 1). Everything else but `pos_label` is considered to belong to the negative class. Set to `None` in the case of multiclass classification.

**average** : string, [`None` (default), ‘micro’, ‘macro’, ‘weighted’]

In the multiclass classification case, this determines the type of averaging performed on the data.

**None**:Do not perform any averaging, return the scores for each class.

**‘macro’**:Average over classes (does not take imbalance into account).

**‘micro’**:Average over instances (takes imbalance into account). This implies that `precision == recall == F1`.

**‘weighted’**:Average weighted by support (takes imbalance into account). Can result in F-score that is not between precision and recall.

**Returns** `precision`: float (if average is not `None`) or array of float, shape = [n\_unique\_labels] :

`recall`: float (if average is not `None`) or array of float, , shape = [n\_unique\_labels] :

`f1_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] :

## References

[http://en.wikipedia.org/wiki/Precision\\_and\\_recall](http://en.wikipedia.org/wiki/Precision_and_recall)

## Examples

In the binary case:

```
>>> from sklearn.metrics import precision_recall_fscore_support
>>> y_pred = [0, 1, 0, 0]
>>> y_true = [0, 1, 0, 1]
>>> p, r, f, s = precision_recall_fscore_support(y_true, y_pred, beta=0.5)
>>> p
array([ 0.66...,  1.        ])
>>> r
array([ 1. ,  0.5])
>>> f
array([ 0.71...,  0.83...])
>>> s
array([2, 2]...)
```

In the multiclass case:

```
>>> 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='weighted')`

Compute the precision

The precision is the ratio  $\frac{tp}{tp + fp}$  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.

**Parameters** `y_true` : array, shape = [n\_samples]

Ground truth (correct) target values.

`y_pred` : array, shape = [n\_samples]

Estimated targets as returned by a classifier.

`labels` : array

Integer array of labels.

`pos_label` : int

In the binary classification case, give the label of the positive class (default is 1). Everything else but `pos_label` is considered to belong to the negative class. Set to `None` in the case of multiclass classification.

`average` : string, [None, ‘micro’, ‘macro’, ‘weighted’ (default)]

In the multiclass classification case, this determines the type of averaging performed on the data.

**None**:Do not perform any averaging, return the scores for each class.

**‘macro’**:Average over classes (does not take imbalance into account).

**‘micro’**:Average over instances (takes imbalance into account). This implies that `precision == recall == F1`.

**‘weighted’**:Average weighted by support (takes imbalance into account). Can result in F-score that is not between precision and recall.

**Returns** `precision` : float (if average is not `None`) or array of float, shape = [n\_unique\_labels]

Precision of the positive class in binary classification or weighted average of the precision of each class for the multiclass task.

## Examples

In the binary case:

```
>>> from sklearn.metrics import precision_score
>>> y_pred = [0, 1, 0, 0]
>>> y_true = [0, 1, 0, 1]
>>> precision_score(y_true, y_pred)
1.0
```

In the multiclass case:

```
>>> 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.        ])
```

## sklearn.metrics.recall\_score

sklearn.metrics.**recall\_score**(y\_true, y\_pred, labels=None, pos\_label=1, average='weighted')

Compute the recall

The recall is the ratio  $\frac{tp}{(tp + fn)}$  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.

**Parameters** **y\_true** : array, shape = [n\_samples]

Ground truth (correct) target values.

**y\_pred** : array, shape = [n\_samples]

Estimated targets as returned by a classifier.

**labels** : array

Integer array of labels.

**pos\_label** : int

In the binary classification case, give the label of the positive class (default is 1). Everything else but **pos\_label** is considered to belong to the negative class. Set to None in the case of multiclass classification.

**average** : string, [None, ‘micro’, ‘macro’, ‘weighted’ (default)]

In the multiclass classification case, this determines the type of averaging performed on the data.

**None**:Do not perform any averaging, return the scores for each class.

**‘macro’**:Average over classes (does not take imbalance into account).

**‘micro’**:Average over instances (takes imbalance into account). This implies that  $precision == recall == F1$ .

**‘weighted’**:Average weighted by support (takes imbalance into account). Can result in F-score that is not between precision and recall.

**Returns recall** : float (if average is not None) or array of float, shape = [n\_unique\_labels]

Recall of the positive class in binary classification or weighted average of the recall of each class for the multiclass task.

## Examples

In the binary case:

```
>>> from sklearn.metrics import recall_score
>>> y_pred = [0, 1, 0, 0]
>>> y_true = [0, 1, 0, 1]
>>> recall_score(y_true, y_pred)
0.5
```

In the multiclass case:

```
>>> 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.])
```

## sklearn.metrics.roc\_curve

`sklearn.metrics.roc_curve(y_true, y_score, pos_label=None)`

Compute Receiver operating characteristic (ROC)

Note: this implementation is restricted to the binary classification task.

**Parameters** `y_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, confidence values, or binary decisions.

`pos_label` : int

Label considered as positive and others are considered negative.

**Returns** `fpr` : array, shape = [>2]

False Positive Rates.

`tpr` : array, shape = [>2]

True Positive Rates.

`thresholds` : array, shape = [>2]

Thresholds on `y_score` used to compute `fpr` and `tpr`.

## 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

[http://en.wikipedia.org/wiki/Receiver\\_operating\\_characteristic](http://en.wikipedia.org/wiki/Receiver_operating_characteristic)

## 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. ])
```

## sklearn.metrics.zero\_one\_loss

`sklearn.metrics.zero_one_loss` (`y_true`, `y_pred`, `normalize=True`)

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.

**Parameters** `y_true` : array-like

`y_pred` : array-like

`normalize` : bool, optional

If `False` (default), return the number of misclassifications. Otherwise, return the fraction of misclassifications.

**Returns** `loss` : float or int,

If `normalize == True`, return the fraction of misclassifications (float), else it returns the number of misclassifications (int).

## 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
```

## Regression metrics

See the *Regression metrics* section of the user guide for further details.

<code>metrics.explained_variance_score(y_true, y_pred)</code>	Explained variance regression score function
<code>metrics.mean_absolute_error(y_true, y_pred)</code>	Mean absolute error regression loss
<code>metrics.mean_squared_error(y_true, y_pred)</code>	Mean squared error regression loss
<code>metrics.r2_score(y_true, y_pred)</code>	R^2 (coefficient of determination) regression score function

### sklearn.metrics.explained\_variance\_score

`sklearn.metrics.explained_variance_score(y_true, y_pred)`

Explained variance regression score function

Best possible score is 1.0, lower values are worse.

**Parameters** `y_true` : array-like

Ground truth (correct) target values.

`y_pred` : array-like

Estimated target values.

**Returns** `score` : 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)`

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.

**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*)

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.

**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...
```

## sklearn.metrics.r2\_score

sklearn.metrics.**r2\_score**(*y\_true*, *y\_pred*)

R<sup>2</sup> (coefficient of determination) regression score function

Best possible score is 1.0, lower values are worse.

**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.

**Returns** *z* : float

The R<sup>2</sup> score

## Notes

This is not a symmetric function.

## References

[http://en.wikipedia.org/wiki/Coefficient\\_of\\_determination](http://en.wikipedia.org/wiki/Coefficient_of_determination)

## 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...
```

## 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.

<code>metrics.adjusted_mutual_info_score(...)</code>	Adjusted Mutual Information between two clusterings
<code>metrics.adjusted_rand_score(labels_true, ...)</code>	Rand index adjusted for chance
<code>metrics.completeness_score(labels_true, ...)</code>	Completeness metric of a cluster labeling given a ground truth
<code>metrics.homogeneity_completeness_v_measure(...)</code>	Compute the homogeneity and completeness and V-Measure score
<code>metrics.homogeneity_score(labels_true, ...)</code>	Homogeneity metric of a cluster labeling given a ground truth
<code>metrics.mutual_info_score(labels_true, ...)</code>	Mutual Information between two clusterings
<code>metrics.normalized_mutual_info_score(...)</code>	Normalized Mutual Information between two clusterings
<code>metrics.silhouette_score(X, labels[, ...])</code>	Compute the mean Silhouette Coefficient of all samples.
<code>metrics.silhouette_samples(X, labels[, metric])</code>	Compute the Silhouette Coefficient for each sample.
<code>metrics.v_measure_score(labels_true, labels_pred)</code>	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:

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

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.

**Parameters** `labels_true` : int array, shape = [n\_samples]

A clustering of the data into disjoint subsets.

`labels_pred` : array, shape = [n\_samples]

A clustering of the data into disjoint subsets.

**Returns** `ami`: float :

score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

**See Also:**

`adjusted_rand_score`Adjusted Rand Index

`mutual_information_score`Mutual Information (not adjusted for chance)

## References

[R96], [R97]

## 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 splitted 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
```

## sklearn.metrics.adjusted\_rand\_score

`sklearn.metrics.adjusted_rand_score`(`labels_true`, `labels_pred`)

Rand index adjusted for chance

The Rand Index computes a similarity measure between two clusterings by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusterings.

The raw RI score is then “adjusted for chance” into the ARI score using the following scheme:

```
ARI = (RI - Expected_RI) / (max(RI) - Expected_RI)
```

The adjusted Rand index is thus ensured to have a value close to 0.0 for random labeling independently of the number of clusters and samples and exactly 1.0 when the clusterings are identical (up to a permutation).

ARI is a symmetric measure:

```
adjusted_rand_score(a, b) == adjusted_rand_score(b, a)
```

**Parameters** `labels_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

**Returns** `ari: float` :

Similarity score between -1.0 and 1.0. Random labelings have an ARI close to 0.0. 1.0 stands for perfect match.

**See Also:**

`adjusted_mutual_info_score` Adjusted Mutual Information

## References

[Hubert1985], [wk]

## Examples

Perfectly matching labelings have a score of 1 even

```
>>> from sklearn.metrics.cluster import adjusted_rand_score
>>> adjusted_rand_score([0, 0, 1, 1], [0, 0, 1, 1])
1.0
>>> adjusted_rand_score([0, 0, 1, 1], [1, 1, 0, 0])
1.0
```

Labelings that assign all classes members to the same clusters are complete but 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
```

## sklearn.metrics.completeness\_score

`sklearn.metrics.completeness_score(labels_true, labels_pred)`

Completeness metric of a cluster labeling given a ground truth

A clustering result satisfies completeness if all the data points that are members of a given class are elements of the same cluster.

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is not symmetric: switching `label_true` with `label_pred` will return the `homogeneity_score` which will be different in general.

**Parameters** `labels_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

**Returns completeness:** float :

score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

**See Also:**

`homogeneity_score`, `v_measure_score`

## References

[R98]

## 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 splitted 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
```

**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.

V-Measure is furthermore symmetric: swapping `labels_true` and `label_pred` will give the same score. This does not hold for homogeneity and completeness.

**Parameters** `labels_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

**Returns** `homogeneity`: float :

score between 0.0 and 1.0. 1.0 stands for perfectly homogeneous labeling

`completeness`: float :

score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

`v_measure`: float :

harmonic mean of the first two

**See Also:**

`homogeneity_score`, `completeness_score`, `v_measure_score`

**sklearn.metrics.homogeneity\_score**

```
sklearn.metrics.homogeneity_score(labels_true, labels_pred)
```

Homogeneity metric of a cluster labeling given a ground truth

A clustering result satisfies homogeneity if all of its clusters contain only data points which are members of a single class.

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is not symmetric: switching `label_true` with `label_pred` will return the `completeness_score` which will be different in general.

**Parameters** `labels_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

**Returns homogeneity: float :**

score between 0.0 and 1.0. 1.0 stands for perfectly homogeneous labeling

**See Also:**

`completeness_score`, `v_measure_score`

## References

[R99]

## Examples

Perfect labelings are homegenous:

```
>>> from sklearn.metrics.cluster import homogeneity_score
>>> homogeneity_score([0, 0, 1, 1], [1, 1, 0, 0])
1.0
```

Non-pefect labelings that futher 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...
```

## 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.

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.

**Parameters** `labels_true` : int array, shape = [n\_samples]

A clustering of the data into disjoint subsets.

`labels_pred` : array, shape = [n\_samples]

A clustering of the data into disjoint subsets.

**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.

**Returns mi:** float :

Mutual information, a non-negative value

**See Also:**

`adjusted_mutual_info_score` Adjusted against chance Mutual Information

`normalized_mutual_info_score` Normalized Mutual Information

### `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 a normalization of the Mutual Information (MI) score to scale the results between 0 (no mutual information) and 1 (perfect correlation). In this function, mutual information is normalized by  $\sqrt{H(\text{labels\_true}) * H(\text{labels\_pred})}$

This measure is not adjusted for chance. Therefore `adjusted_mutual_info_score` might be preferred.

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is furthermore symmetric: switching `label_true` with `label_pred` will return the same score value. This can be useful to measure the agreement of two independent label assignments strategies on the same dataset when the real ground truth is not known.

**Parameters** `labels_true` : int array, shape = [n\_samples]

A clustering of the data into disjoint subsets.

`labels_pred` : array, shape = [n\_samples]

A clustering of the data into disjoint subsets.

**Returns nmi:** float :

score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

**See Also:**

`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 splitted 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.

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.

**Parameters** `X` : array [n\_samples\_a, n\_samples\_a] if metric == “precomputed”, or, [n\_samples\_a, n\_features] otherwise

Array of pairwise distances between samples, or a feature array.

`labels` : array, shape = [n\_samples]

label values for each sample

`metric` : string, or callable

The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by `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.

**Returns silhouette** : float

Mean Silhouette Coefficient for all samples.

## References

[R104], [R105]

### `sklearn.metrics.silhouette_samples`

`sklearn.metrics.silhouette_samples(X, labels, metric='euclidean', **kwds)`

Compute the Silhouette Coefficient for each sample.

The Silhouette Coefficient is a measure of how well samples are clustered with samples that are similar to themselves. Clustering models with a high Silhouette Coefficient are said to be dense, where samples in the same cluster are similar to each other, and well separated, where samples in different clusters are not very similar to each other.

The Silhouette Coefficient is calculated using the mean intra-cluster distance ( $a$ ) and the mean nearest-cluster distance ( $b$ ) for each sample. The Silhouette Coefficient for a sample is  $(b - a) / \max(a, b)$ .

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.

**Parameters** `X` : array [n\_samples\_a, n\_samples\_a] if metric == “precomputed”, or, [n\_samples\_a, n\_features] otherwise

Array of pairwise distances between samples, or a feature array.

`labels` : array, shape = [n\_samples]

label values for each sample

`metric` : string, or callable

The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by `sklearn.metrics.pairwise.pairwise_distances`. If X is the distance array itself, use “precomputed” as the metric.

`**kwds` : optional keyword parameters

Any further parameters are passed directly to the distance function. If using a `scipy.spatial.distance` metric, the parameters are still metric dependent. See the `scipy` docs for usage examples.

**Returns silhouette** : array, shape = [n\_samples]

Silhouette Coefficient for each samples.

## References

[R102], [R103]

## 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.

**Parameters** `labels_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

**Returns** `completeness`: float :

score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

**See Also:**

`homogeneity_score`, `completeness_score`

## References

[R106]

## 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 but 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 splitted 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...
```

## Pairwise metrics

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 a function  $d(a, b)$  such that  $d(a, b) < d(a, c)$  if objects  $a$  and  $b$  are considered “more similar” to 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.  $d(a, b) \geq 0$ , 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.  $d(a, c) \leq d(a, b) + d(b, c)$ , 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” to 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  $1 / \text{num\_features}$
2.  $S = 1. / (D / np.max(D))$

<code>metrics.pairwise.additive_chi2_kernel(X[, Y])</code>	Computes the additive chi-squared kernel between observations in X and Y.
<code>metrics.pairwise.chi2_kernel(X[, Y, gamma])</code>	Computes the exponential chi-squared kernel X and Y.
<code>metrics.pairwise.distance_metrics()</code>	Valid metrics for pairwise_distances
<code>metrics.pairwise.euclidean_distances(X[, Y, ...])</code>	Considering the rows of X (and Y=X) as vectors, compute the Euclidean distance between them.
<code>metrics.pairwise.kernel_metrics()</code>	Valid metrics for pairwise_kernels
<code>metrics.pairwise.linear_kernel(X[, Y])</code>	Compute the linear kernel between X and Y.
<code>metrics.pairwise.manhattan_distances(X[, Y, ...])</code>	Compute the L1 distances between the vectors in X and Y.
<code>metrics.pairwise.pairwise_distances(X[, Y, ...])</code>	Compute the distance matrix from a vector array X and optional Y.
<code>metrics.pairwise.pairwise_kernels(X[, Y, ...])</code>	Compute the kernel between arrays X and optional array Y.
<code>metrics.pairwise.polynomial_kernel(X[, Y, ...])</code>	Compute the polynomial kernel between X and Y.

Continued on next page

**Table 1.157 – continued from previous page**

<code>metrics.pairwise.rbf_kernel(X[, Y, gamma])</code>	Compute the rbf (gaussian) kernel between X and Y:
---	--

### `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:

$$k(x, y) = -\sum_i (x[i] - y[i]) ** 2 / (x[i] + y[i])$$

It can be interpreted as a weighted difference per entry.

**Parameters** `X` : array-like of shape (n\_samples\_X, n\_features)

`Y` : array of shape (n\_samples\_Y, n\_features)

**Returns** `kernel_matrix` : array of shape (n\_samples\_X, n\_samples\_Y)

**See Also:**

`chi2_kernel` The exponentiated version of the kernel, which is usually preferable.

`sklearn.kernel_approximation.AdditiveChi2Sampler` A Fourier approximation to this kernel.

### Notes

As the negative of a distance, this kernel is only conditionally positive definite.

### References

- Zhang, J. and Marszalek, M. and Lazebnik, S. and Schmid, C. Local features and kernels for classification of texture and object categories: A comprehensive study International Journal of Computer Vision 2007 <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:

$$k(x, y) = \exp(-gamma * \sum_i (x[i] - y[i]) ** 2 / (x[i] + y[i]))$$

It can be interpreted as a weighted difference per entry.

**Parameters** `X` : array-like of shape (n\_samples\_X, n\_features)

`Y` : array of shape (n\_samples\_Y, n\_features)

**gamma** : float, default=1.

Scaling parameter of the chi2 kernel.

**Returns** `kernel_matrix` : array of shape (n\_samples\_X, n\_samples\_Y)

**See Also:**

`additive_chi2_kernel` The additive version of this kernel

`sklearn.kernel_approximation.AdditiveChi2Sampler` A Fourier approximation to the additive version of this kernel.

## References

- Zhang, J. and Marszalek, M. and Lazebnik, S. and Schmid, C. Local features and kernels for classification of texture and object categories: A comprehensive study International Journal of Computer Vision 2007 <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, 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
'cityblock'	<code>sklearn.pairwise.manhattan_distances</code>
'euclidean'	<code>sklearn.pairwise.euclidean_distances</code>
'l1'	<code>sklearn.pairwise.manhattan_distances</code>
'l2'	<code>sklearn.pairwise.euclidean_distances</code>
'manhattan'	<code>sklearn.pairwise.manhattan_distances</code>

## 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 main advantages. 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.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples\_1, n\_features]

`Y` : {array-like, sparse matrix}, shape = [n\_samples\_2, n\_features]

`Y_norm_squared` : array-like, shape = [n\_samples\_2], optional

Pre-computed dot-products of vectors in Y (e.g., `(Y**2).sum(axis=1)`)

`squared` : boolean, optional

Return squared Euclidean distances.

**Returns distances** : {array, sparse matrix}, shape = [n\_samples\_1, n\_samples\_2]

## 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.

**Parameters** `X` : array of shape (n\_samples\_1, n\_features)

`Y` : array of shape (n\_samples\_2, n\_features)

**Returns Gram matrix** : array of shape (n\_samples\_1, n\_samples\_2)

## sklearn.metrics.pairwise.manhattan\_distances

`sklearn.metrics.pairwise.manhattan_distances(X, Y=None, sum_over_features=True)`

Compute the L1 distances between the vectors in X and Y.

With sum\_over\_features equal to False it returns the componentwise distances.

**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.

**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.

Please note that support for sparse matrices is currently limited to those metrics listed in pairwise.pairwise\_distance\_functions.

Valid values for metric are:

- from scikit-learn: ['euclidean', 'l2', 'l1', 'manhattan', 'cityblock']

•from scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'cosine', 'dice', 'hamming', 'jaccard', 'kulnsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russell-rao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'squeuclidean', 'yule'] See the documentation for `scipy.spatial.distance` for details on these metrics.

Note in the case of ‘euclidean’ and ‘cityblock’ (which are valid `scipy.spatial.distance` metrics), the values will use the scikit-learn implementation, which is faster and has support for sparse matrices. For a verbose description of the metrics from scikit-learn, see the `__doc__` of the `sklearn.pairwise.distance_metrics` function.

**Parameters** `X` : array [n\_samples\_a, n\_samples\_a] if metric == “precomputed”, or, [n\_samples\_a, n\_features] otherwise

    Array of pairwise distances between samples, or a feature array.

`Y` : array [n\_samples\_b, n\_features]

    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.

**Returns** `D` : array [n\_samples\_a, n\_samples\_a] or [n\_samples\_a, n\_samples\_b]

    A distance matrix `D` such that  $D_{i,j}$  is the distance between the  $i$ th and  $j$ th vectors of the given matrix `X`, if `Y` is None. If `Y` is not None, then  $D_{i,j}$  is the distance between the  $i$ th array from `X` and the  $j$ th array from `Y`.

## sklearn.metrics.pairwise.pairwise\_kernels

```
sklearn.metrics.pairwise.pairwise_kernels(X, Y=None, metric='linear', fil-  
ter_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']

**Parameters** **X** : array [n\_samples\_a, n\_samples\_a] if metric == “precomputed”, or, [n\_samples\_a, n\_features] otherwise

Array of pairwise kernels between samples, or a feature array.

**Y** : array [n\_samples\_b, n\_features]

A second feature array only if X has shape [n\_samples\_a, n\_features].

**metric** : string, or callable

The metric to use when calculating kernel between instances in a feature array. If metric is a string, it must be one of the metrics in pairwise.pairwise\_kernel\_functions. If metric is “precomputed”, X is assumed to be a kernel matrix. Alternatively, if metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two 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.

**Returns** **K** : array [n\_samples\_a, n\_samples\_a] or [n\_samples\_a, n\_samples\_b]

A kernel matrix K such that  $K_{\{i, j\}}$  is the kernel between the ith and jth vectors of the given matrix X, if Y is None. If Y is not None, then  $K_{\{i, j\}}$  is the kernel between the ith array from X and the jth array from Y.

## Notes

If metric is ‘precomputed’, Y is ignored and X is returned.

### `sklearn.metrics.pairwise.polynomial_kernel`

```
sklearn.metrics.pairwise.polynomial_kernel(X, Y=None, degree=3, gamma=None,
                                           coef0=1)
```

Compute the polynomial kernel between X and Y:

```
K(X, Y) = (gamma <X, Y> + coef0)^degree
```

**Parameters** **X** : array of shape (n\_samples\_1, n\_features)  
**Y** : array of shape (n\_samples\_2, n\_features)  
**degree** : int  
**Returns** **Gram matrix** : array of shape (n\_samples\_1, n\_samples\_2)

#### sklearn.metrics.pairwise.rbf\_kernel

`sklearn.metrics.pairwise.rbf_kernel(X, Y=None, gamma=None)`  
Compute the rbf (gaussian) kernel between X and Y:

$K(x, y) = \exp(-\gamma \|x-y\|^2)$

for each pair of rows x in X and y in Y.

**Parameters** **X** : array of shape (n\_samples\_X, n\_features)  
**Y** : array of shape (n\_samples\_Y, n\_features)  
**gamma** : float  
**Returns** **kernel\_matrix** : array of shape (n\_samples\_X, n\_samples\_Y)

## 1.8.20 sklearn.mixture: Gaussian Mixture Models

The `sklearn.mixture` module implements mixture modeling algorithms.

**User guide:** See the *Gaussian mixture models* section for further details.

<code>mixture.GMM([n_components, covariance_type, ...])</code>	Gaussian Mixture Model
<code>mixture.DPGMM([n_components, ...])</code>	Variational Inference for the Infinite Gaussian Mixture Model.
<code>mixture.VBGMM([n_components, ...])</code>	Variational Inference for the Gaussian Mixture Model

#### sklearn.mixture.GMM

`class sklearn.mixture.GMM(n_components=1, covariance_type='diag', random_state=None, thresh=0.01, 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.

**Parameters** **n\_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 (0 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.

**thresh** : float, optional

Convergence threshold.

**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’.

## 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=0.01)
>>> 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=0.01)
>>> np.round(g.weights_, 2)
array([ 0.5,  0.5])
```

## Attributes

<code>weights_</code>	array, shape ( <i>n_components</i> ,)	This attribute stores the mixing weights for each mixture component.
<code>means_</code>	array, shape ( <i>n_components</i> , <i>n_features</i> )	Mean parameters for each mixture component.
<code>covars_</code>	array	Covariance parameters for each mixture component. The shape depends on <code>covariance_type</code> :  ( <i>n_components</i> , <i>n_features</i> ) ( <i>n_features</i> , <i>n_features</i> ) ( <i>n_components</i> , <i>n_features</i> ) ( <i>n_components</i> , <i>n_features</i> , <i>n_features</i> )
<code>converged_</code>	bool	True when convergence was reached in fit(), False otherwise.

## Methods

<code>aic(X)</code>	Akaike information criterion for the current model fit
<code>bic(X)</code>	Bayesian information criterion for the current model fit
<code>eval(X)</code>	Evaluate the model on data
<code>fit(X)</code>	Estimate model parameters with the expectation-maximization algorithm.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict label for data.
<code>predict_proba(X)</code>	Predict posterior probability of data under each Gaussian
<code>sample([n_samples, random_state])</code>	Generate random samples from the model.
<code>score(X)</code>	Compute the log probability under the model.
<code>set_params(**params)</code>	Set the parameters of the estimator.

```
__init__(n_components=1, covariance_type='diag', random_state=None, thresh=0.01,
        min_covar=0.001, n_iter=100, n_init=1, params='wmc', init_params='wmc')
```

**aic**(*X*)

Akaike information criterion for the current model fit and the proposed data

**Parameters** **X** : array of shape(*n\_samples*, *n\_dimensions*)

**Returns** **aic**: float (the lower the better) :

**bic**(*X*)

Bayesian information criterion for the current model fit and the proposed data

**Parameters** **X** : array of shape(*n\_samples*, *n\_dimensions*)

**Returns bic: float (the lower the better) :**

**eval (X)**

Evaluate the model on data

Compute the log probability of X under the model and return the posterior distribution (responsibilities) of each mixture component for each element of X.

**Parameters X: array\_like, shape (n\_samples, n\_features) :**

List of n\_features-dimensional data points. Each row corresponds to a single data point.

**Returns logprob: 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

**fit (X)**

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.

**Parameters X : 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 the estimator

**Parameters deep: boolean, optional :**

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

**predict (X)**

Predict label for data.

**Parameters X : array-like, shape = [n\_samples, n\_features]**

**Returns C : array, shape = (n\_samples,)**

**predict\_proba (X)**

Predict posterior probability of data under each Gaussian in the model.

**Parameters X : array-like, shape = [n\_samples, n\_features]**

**Returns responsibilities : 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.

**Parameters n\_samples : int, optional**

Number of samples to generate. Defaults to 1.

**Returns X : array\_like, shape (n\_samples, n\_features)**

List of samples

**score (X)**

Compute the log probability under the model.

**Parameters** **X** : array\_like, shape (n\_samples, n\_features)

List of n\_features-dimensional data points. Each row corresponds to a single data point.

**Returns logprob** : array\_like, shape (n\_samples,)

Log probabilities of each data point in X

**set\_params** (\*\*params)

Set the parameters of the 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.mixture.DPGMM

```
class sklearn.mixture.DPGMM(n_components=1, covariance_type='diag', alpha=1.0, random_state=None, thresh=0.01, 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.

**Parameters** **n\_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’.

**alpha**: float, optional :

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  $\alpha \log(N)$ . Defaults to 1.

**thresh** : float, optional

Convergence threshold.

**n\_iter** : int, optional

Maximum number of iterations to perform before convergence.

**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’.

See Also:

**GMM**Finite Gaussian mixture model fit with EM

**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.

## Attributes

covariance_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 ( <i>n_components</i> ,)	Mixing weights for each mixture component.
means_	array, shape ( <i>n_components</i> , <i>n_features</i> )	Mean parameters for each mixture component.
precisions_	array	Precision (inverse covariance) parameters for each mixture component. The shape depends on <i>covariance_type</i> : (‘n_components’, ‘n_features’) (‘n_features’, ‘n_features’) (‘n_components’, ‘n_features’) (‘n_components’, ‘n_features’, ‘n_features’)
converged_	bool	True when convergence was reached in fit(), False otherwise.

## Methods

aic(X)	Akaike information criterion for the current model fit
bic(X)	Bayesian information criterion for the current model fit
eval(X)	Evaluate the model on data
fit(X, **kwargs)	Estimate model parameters with the variational algorithm.
get_params([deep])	Get parameters for the 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
sample([n_samples, random_state])	Generate random samples from the model.
score(X)	Compute the log probability under the model.
set_params(**params)	Set the parameters of the estimator.

**\_\_init\_\_** (*n\_components*=1, *covariance\_type*=‘diag’, *alpha*=1.0, *random\_state*=None, *thresh*=0.01, *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

**Parameters** X : array of shape(n\_samples, n\_dimensions)

**Returns aic:** float (the lower the better) :

**bic**(X)

Bayesian information criterion for the current model fit and the proposed data

**Parameters** X : array of shape(n\_samples, n\_dimensions)

**Returns bic:** float (the lower the better) :

**eval**(X)

Evaluate the model on data

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.

**Parameters** X : array\_like, shape (n\_samples, n\_features)

List of n\_features-dimensional data points. Each row corresponds to a single data point.

**Returns logprob :** 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

**fit**(X, \*\*kwargs)

Estimate model parameters with the variational algorithm.

For a full derivation and description of the algorithm see doc/dp-derivation/dp-derivation.tex

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.

**Parameters** X : 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 the estimator

**Parameters deep:** boolean, optional :

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

**lower\_bound**(X, z)

returns a lower bound on model evidence based on X and membership

**predict**(X)

Predict label for data.

**Parameters** X : array-like, shape = [n\_samples, n\_features]

**Returns C :** array, shape = (n\_samples,)

**predict\_proba**(X)

Predict posterior probability of data under each Gaussian in the model.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

**Returns responsibilities** : 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.

**Parameters n\_samples** : int, optional

Number of samples to generate. Defaults to 1.

**Returns X** : array\_like, shape (n\_samples, n\_features)

List of samples

**score** (X)

Compute the log probability under the model.

**Parameters X** : array\_like, shape (n\_samples, n\_features)

List of n\_features-dimensional data points. Each row corresponds to a single data point.

**Returns logprob** : array\_like, shape (n\_samples,)

Log probabilities of each data point in X

**set\_params** (\*\*params)

Set the parameters of the 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.mixture.VBGMM

```
class sklearn.mixture.VBGMM(n_components=1, covariance_type='diag', alpha=1.0, random_state=None, thresh=0.01, 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.

**Parameters n\_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’.

**alpha: float, optional** :

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. Defaults to 1.

**See Also:****GMM**Finite Gaussian mixture model fit with EM**DPGMM**Infinite Gaussian mixture model, using the dirichlet process, fit with a variational algorithm**Attributes**

covariance_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.
precisions_	array	Precision (inverse covariance) parameters for each mixture component. The shape depends on covariance_type: (‘n_components’, ‘n_features’) (‘n_features’, ‘n_features’) (‘n_components’, ‘n_features’) (‘n_components’, ‘n_features’, ‘n_features’)
converged_	bool	True when convergence was reached in fit(), False otherwise.

**Methods**

aic(X)	Akaike information criterion for the current model fit
bic(X)	Bayesian information criterion for the current model fit
eval(X)	Evaluate the model on data
fit(X, **kwargs)	Estimate model parameters with the variational algorithm.
get_params([deep])	Get parameters for the 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
sample([n_samples, random_state])	Generate random samples from the model.
score(X)	Compute the log probability under the model.
set_params(**params)	Set the parameters of the estimator.

**\_\_init\_\_** (n\_components=1, covariance\_type='diag', alpha=1.0, random\_state=None, thresh=0.01, 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

**Parameters** X : array of shape(n\_samples, n\_dimensions)

**Returns aic: float (the lower the better) :**

**bic**(*X*)

Bayesian information criterion for the current model fit and the proposed data

**Parameters** *X* : array of shape(n\_samples, n\_dimensions)

**Returns bic: float (the lower the better) :**

**eval**(*X*)

Evaluate the model on data

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.

**Parameters** *X* : array\_like, shape (n\_samples, n\_features)

List of n\_features-dimensional data points. Each row corresponds to a single data point.

**Returns logprob** : 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

**fit**(*X*, \*\*kwargs)

Estimate model parameters with the variational algorithm.

For a full derivation and description of the algorithm see doc/dp-derivation/dp-derivation.tex

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 object. Likewise, if you would like just to do an initialization, set n\_iter=0.

**Parameters** *X* : 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 the estimator

**Parameters deep: boolean, optional :**

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

**lower\_bound**(*X*, *z*)

returns a lower bound on model evidence based on *X* and membership

**predict**(*X*)

Predict label for data.

**Parameters** *X* : array-like, shape = [n\_samples, n\_features]

**Returns C** : array, shape = (n\_samples,)

**predict\_proba**(*X*)

Predict posterior probability of data under each Gaussian in the model.

**Parameters** *X* : array-like, shape = [n\_samples, n\_features]

**Returns responsibilities** : 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.

**Parameters** *n\_samples* : int, optional

Number of samples to generate. Defaults to 1.

**Returns** *X* : array\_like, shape (*n\_samples*, *n\_features*)

List of samples

**score** (*X*)

Compute the log probability under the model.

**Parameters** *X* : array\_like, shape (*n\_samples*, *n\_features*)

List of *n\_features*-dimensional data points. Each row corresponds to a single data point.

**Returns** *logprob* : array\_like, shape (*n\_samples*,

Log probabilities of each data point in *X*

**set\_params** (\*\**params*)

Set the parameters of the 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 :

## 1.8.21 sklearn.multiclass: Multiclass and multilabel classification

### 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.

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.

<code>multiclass.OneVsRestClassifier(estimator[, ...])</code>	One-vs-the-rest (OvR) multiclass/multilabel strategy
<code>multiclass.OneVsOneClassifier(estimator[, ...])</code>	One-vs-one multiclass strategy
<code>multiclass.OutputCodeClassifier(estimator[, ...])</code>	(Error-Correcting) Output-Code multiclass strategy

## 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 sequence of sequences of labels (e.g., a list of tuples) rather than a single target vector. For multilabel learning, the number of classes must be at least three, since otherwise OvR reduces to binary classification.

In the multilabel learning literature, OvR is also known as the binary relevance method.

**Parameters estimator** : 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.

### Attributes

<i>estimators_</i>	list of $n_{classes}$ estimators	Estimators used for predictions.
<i>classes_</i>	array, shape = [ $n_{classes}$ ]	Class labels.
<i>label_binarizer_</i>	LabelBinarizer object	Object used to transform multiclass labels to binary labels and vice-versa.
<i>multilabel_</i>	boolean	Whether a OneVsRestClassifier is a multilabel classifier.

### Methods

<b>fit(X, y)</b>	Fit underlying estimators.
<b>get_params([deep])</b>	Get parameters for the estimator
<b>predict(X)</b>	Predict multi-class targets using underlying estimators.
<b>predict_proba(X)</b>	Probability estimates.
<b>score(X, y)</b>	
<b>set_params(**params)</b>	Set the parameters of the estimator.

**\_\_init\_\_(estimator, n\_jobs=1)**

**fit(X, y)**

Fit underlying estimators.

**Parameters X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Data.

**y** : array-like, shape = [n\_samples]  
**or sequence of sequences, len = n\_samples** Multi-class targets. A sequence of sequences turns on multilabel classification.

**Returns self :**

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**multilabel\_**

Whether this is a multilabel classifier

**predict (X)**

Predict multi-class targets using underlying estimators.

**Parameters X: {array-like, sparse matrix}, shape = [n\_samples, n\_features] :**

Data.

**Returns y : array-like, shape = [n\_samples]**

Predicted multi-class targets.

**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.

**Parameters X : array-like, shape = [n\_samples, n\_features]**

**Returns T : 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\_*.

**set\_params (\*\*params)**

Set the parameters of the 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.multiclass.OneVsOneClassifier

**class sklearn.multiclass.OneVsOneClassifier (estimator, n\_jobs=1)**  
One-vs-one multiclass strategy

This strategy consists in fitting one classifier per class pair. At prediction time, the class which received the most votes is selected. Since it requires to fit  $n_{\text{classes}} * (n_{\text{classes}} - 1) / 2$  classifiers, this method is usually slower than one-vs-the-rest, due to its  $O(n_{\text{classes}}^2)$  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.

**Parameters estimator** : estimator object

An estimator object implementing *fit* and *predict*.

**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.

## Attributes

<i>estimators_</i>	list of $n_{classes} * (n_{classes} - 1) / 2$ estimators	Estimators used for predictions.
<i>classes_</i>	numpy array of shape [n_classes]	Array containing labels.

## Methods

<code>fit(X, y)</code>	Fit underlying estimators.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict multi-class targets using underlying estimators.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

**\_\_init\_\_**(estimator, n\_jobs=1)

**fit**(X, y)

Fit underlying estimators.

**Parameters X**: {array-like, sparse matrix}, shape = [n\_samples, n\_features] :

Data.

**y** : numpy array of shape [n\_samples]

Multi-class targets.

**Returns self** :

**get\_params**(deep=True)

Get parameters for the estimator

**Parameters deep**: boolean, optional :

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

**predict**(X)

Predict multi-class targets using underlying estimators.

**Parameters X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Data.

**Returns y** : numpy array of shape [n\_samples]

Predicted multi-class targets.

**score** ( $X, y$ )

Returns the mean accuracy on the given test data and labels.

**Parameters**  $X$  : array-like, shape = [n\_samples, n\_features]

Training set.

$y$  : array-like, shape = [n\_samples]

Labels for  $X$ .

**Returns**  $z$  : float

**set\_params** (\*\*params)

Set the parameters of the 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.multiclass.OutputCodeClassifier

**class** sklearn.multiclass.**OutputCodeClassifier** (*estimator*, code\_size=1.5, ran-

dom\_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 ( $0 < \text{code\_size} < 1$ ) or for making the model more robust to errors ( $\text{code\_size} > 1$ ). See the documentation for more details.

**Parameters** **estimator** : estimator object

An estimator object implementing *fit* and one of *decision\_function* or *predict\_proba*.

**code\_size** : float

Percentage of the number of classes to be used to create the code book. A number between 0 and 1 will require fewer classifiers than one-vs-the-rest. A number greater than 1 will require more classifiers than one-vs-the-rest.

**random\_state** : 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, ( $n_{\text{cpus}} + 1 + n_{\text{jobs}}$ ) are used. Thus for n\_jobs = -2, all CPUs but one are used.

## References

[R107], [R108], [R109]

## Attributes

<code>estimators_</code>	list of <code>int(n_classes * code_size)</code> estimators	Estimators used for predictions.
<code>classes_</code>	numpy array of shape [n_classes]	Array containing labels.
<code>code_book_</code>	numpy array of shape [n_classes, code_size]	Binary array containing the code of each class.

## Methods

<code>fit(X, y)</code>	Fit underlying estimators.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict multi-class targets using underlying estimators.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(estimator, code_size=1.5, random_state=None, n_jobs=1)`

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

Fit underlying estimators.

**Parameters X:** {array-like, sparse matrix}, shape = [n\_samples, n\_features] :

Data.

*y* : numpy array of shape [n\_samples]

Multi-class targets.

**Returns self :**

**get\_params** (*deep=True*)

Get parameters for the estimator

**Parameters deep:** boolean, optional :

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

**predict** (*X*)

Predict multi-class targets using underlying estimators.

**Parameters X:** {array-like, sparse matrix}, shape = [n\_samples, n\_features] :

Data.

**Returns y** : numpy array of shape [n\_samples]

Predicted multi-class targets.

**score** (*X*, *y*)

Returns the mean accuracy on the given test data and labels.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

*y* : array-like, shape = [n\_samples]

Labels for *X*.

**Returns z** : float

**set\_params** (*\*\*params*)

Set the parameters of the 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 :**

<code>multiclass.fit_ovr(estimator, X, y[, n_jobs])</code>	Fit a one-vs-the-rest strategy.
<code>multiclass.predict_ovr(estimators, ...)</code>	Make predictions using the one-vs-the-rest strategy.
<code>multiclass.fit_ovo(estimator, X, y[, n_jobs])</code>	Fit a one-vs-one strategy.
<code>multiclass.predict_ovo(estimators, classes, X)</code>	Make predictions using the one-vs-one strategy.
<code>multiclass.fit_ecoc(estimator, X, y[, ...])</code>	Fit an error-correcting output-code strategy.
<code>multiclass.predict_ecoc(estimators, classes, ...)</code>	Make predictions using the error-correcting output-code strategy.

**sklearn.multiclass.fit\_ovr**

```
sklearn.multiclass.fit_ovr(estimator, X, y, n_jobs=1)
```

Fit a one-vs-the-rest strategy.

**sklearn.multiclass.predict\_ovr**

```
sklearn.multiclass.predict_ovr(estimators, label_binarizer, X)
```

Make predictions using the one-vs-the-rest strategy.

**sklearn.multiclass.fit\_ovo**

```
sklearn.multiclass.fit_ovo(estimator, X, y, n_jobs=1)
```

Fit a one-vs-one strategy.

**sklearn.multiclass.predict\_ovo**

```
sklearn.multiclass.predict_ovo(estimators, classes, X)
```

Make predictions using the one-vs-one strategy.

**sklearn.multiclass.fit\_ecoc**

```
sklearn.multiclass.fit_ecoc(estimator, X, y, code_size=1.5, random_state=None, n_jobs=1)
```

Fit an error-correcting output-code strategy.

**Parameters estimator** : estimator object

An estimator object implementing *fit* and one of *decision\_function* or *predict\_proba*.

**code\_size: float, optional** :

Percentage of the number of classes to be used to create the code book.

**random\_state: numpy.RandomState, optional** :

The generator used to initialize the codebook. Defaults to numpy.random.

**Returns estimators** : list of *int(n\_classes \* code\_size)* estimators

Estimators used for predictions.

**classes** : numpy array of shape [n\_classes]

Array containing labels.

**'code\_book\_'**: numpy array of shape [n\_classes, code\_size] :

Binary array containing the code of each class.

## sklearn.multiclass.predict\_ecoc

`sklearn.multiclass.predict_ecoc(estimators, classes, code_book, X)`

Make predictions using the error-correcting output-code strategy.

## 1.8.22 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.

<code>naive_bayes.GaussianNB</code>	Gaussian Naive Bayes (GaussianNB)
<code>naive_bayes.MultinomialNB([alpha, ...])</code>	Naive Bayes classifier for multinomial models
<code>naive_bayes.BernoulliNB([alpha, binarize, ...])</code>	Naive Bayes classifier for multivariate Bernoulli models.

### sklearn.naive\_bayes.GaussianNB

`class sklearn.naive_bayes.GaussianNB`  
Gaussian Naive Bayes (GaussianNB)

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Training vector, where n\_samples in the number of samples and n\_features is the number of features.

`y` : array, shape = [n\_samples]

Target vector relative to `X`

### 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]
```

## Attributes

<code>class_prior_</code>	array, shape = [n_classes]	probability of each class.
<code>theta_</code>	array, shape = [n_classes, n_features]	mean of each feature per class
<code>sigma_</code>	array, shape = [n_classes, n_features]	variance of each feature per class

## Methods

<code>fit(X, y)</code>	Fit Gaussian Naive Bayes according to X, y
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Perform classification on an array of test vectors X.
<code>predict_log_proba(X)</code>	Return log-probability estimates for the test vector X.
<code>predict_proba(X)</code>	Return probability estimates for the test vector X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

### `__init__()`

x.`__init__`(...) initializes x; see help(type(x)) for signature

### `fit(X, y)`

Fit Gaussian Naive Bayes according to X, y

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features.

`y` : array-like, shape = [n\_samples]

Target values.

**Returns** `self` : object

Returns self.

### `get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep`: boolean, optional :

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

### `predict(X)`

Perform classification on an array of test vectors X.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

**Returns** `C` : array, shape = [n\_samples]

Predicted target values for X

### `predict_log_proba(X)`

Return log-probability estimates for the test vector X.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

**Returns** `C` : array-like, shape = [n\_samples, n\_classes]

Returns the log-probability of the sample for each class in the model, where classes are ordered arithmetically.

**predict\_proba (X)**

Return probability estimates for the test vector X.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

**Returns** **C** : array-like, shape = [n\_samples, n\_classes]

Returns the probability of the sample for each class in the model, where classes are ordered arithmetically.

**score (X, y)**

Returns the mean accuracy on the given test data and labels.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns** **z** : float

**set\_params (\*\*params)**

Set the parameters of the 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.naive\_bayes.MultinomialNB**

```
class sklearn.naive_bayes.MultinomialNB(alpha=1.0, fit_prior=True, class_prior=None)
Naive Bayes classifier for multinomial models
```

The multinomial Naive Bayes classifier is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf may also work.

**Parameters** **alpha** : 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.

**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.

## 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]
```

## Attributes

<code>intercept_</code> , <code>class_log_prior_[n_classes]</code>	array, shape = [n_classes]	Smoothed empirical log probability for each class.
<code>feature_log_prob_</code> , <code>coef_</code>	array, shape = [n_classes, n_features]	Empirical log probability of features given a class, $P(x_i y)$ . ( <code>intercept_</code> and <code>coef_</code> are properties referring to <code>class_log_prior_</code> and <code>feature_log_prob_</code> , respectively.)

## Methods

<code>fit(X, y[, sample_weight, class_prior])</code>	Fit Naive Bayes classifier according to X, y
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Perform classification on an array of test vectors X.
<code>predict_log_proba(X)</code>	Return log-probability estimates for the test vector X.
<code>predict_proba(X)</code>	Return probability estimates for the test vector X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(alpha=1.0, fit_prior=True, class_prior=None)`

`fit(X, y, sample_weight=None, class_prior=None)`

Fit Naive Bayes classifier according to X, y

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features.

`y` : array-like, shape = [n\_samples]

Target values.

`sample_weight` : array-like, shape = [n\_samples], optional

Weights applied to individual samples (1. for unweighted).

**Returns self** : object

Returns self.

`get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep`: boolean, optional :

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

**predict**(*X*)

Perform classification on an array of test vectors *X*.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

**Returns** **C** : array, shape = [n\_samples]

Predicted target values for *X*

**predict\_log\_proba**(*X*)

Return log-probability estimates for the test vector *X*.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

**Returns** **C** : array-like, shape = [n\_samples, n\_classes]

Returns the log-probability of the sample for each class in the model, where classes are ordered arithmetically.

**predict\_proba**(*X*)

Return probability estimates for the test vector *X*.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

**Returns** **C** : array-like, shape = [n\_samples, n\_classes]

Returns the probability of the sample for each class in the model, where classes are ordered arithmetically.

**score**(*X*, *y*)

Returns the mean accuracy on the given test data and labels.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for *X*.

**Returns** **z** : float

**set\_params**(*\*\*params*)

Set the parameters of the 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.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.

**Parameters** **alpha** : float, optional (default=1.0)

Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

**binarize** : float or None, optional

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.

## 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. Palouras (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]
```

## Attributes

<code>class_log_prior_</code>	array, shape = [n_classes]	Log probability of each class (smoothed).
<code>feature_log_prob_</code>	array, shape = [n_classes, n_features]	Empirical log probability of features given a class, $P(x_i y)$ .

## Methods

<code>fit(X, y[, sample_weight, class_prior])</code>	Fit Naive Bayes classifier according to X, y
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Perform classification on an array of test vectors X.
<code>predict_log_proba(X)</code>	Return log-probability estimates for the test vector X.
<code>predict_proba(X)</code>	Return probability estimates for the test vector X.

Continued on next page

**Table 1.170 – continued from previous page**

<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

**`__init__(alpha=1.0, binarize=0.0, fit_prior=True, class_prior=None)`**

**`fit(X, y, sample_weight=None, class_prior=None)`**

Fit Naive Bayes classifier according to X, y

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features.

`y` : array-like, shape = [n\_samples]

Target values.

`sample_weight` : array-like, shape = [n\_samples], optional

Weights applied to individual samples (1. for unweighted).

**Returns** `self` : object

Returns self.

**`get_params(deep=True)`**

Get parameters for the estimator

**Parameters** `deep`: boolean, optional :

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

**`predict(X)`**

Perform classification on an array of test vectors X.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

**Returns** `C` : array, shape = [n\_samples]

Predicted target values for X

**`predict_log_proba(X)`**

Return log-probability estimates for the test vector X.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

**Returns** `C` : array-like, shape = [n\_samples, n\_classes]

Returns the log-probability of the sample for each class in the model, where classes are ordered arithmetically.

**`predict_proba(X)`**

Return probability estimates for the test vector X.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

**Returns** `C` : array-like, shape = [n\_samples, n\_classes]

Returns the probability of the sample for each class in the model, where classes are ordered arithmetically.

**`score(X, y)`**

Returns the mean accuracy on the given test data and labels.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns** **z** : float

**set\_params** (\*\*params)

Set the parameters of the 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** :

## 1.8.23 sklearn.neighbors: Nearest Neighbors

The `sklearn.neighbors` module implements the k-nearest neighbors algorithm.

**User guide:** See the *Nearest Neighbors* section for further details.

<code>neighbors.NearestNeighbors([n_neighbors, ...])</code>	Unsupervised learner for implementing neighbor searches.
<code>neighbors.KNeighborsClassifier([...])</code>	Classifier implementing the k-nearest neighbors vote.
<code>neighbors.RadiusNeighborsClassifier([...])</code>	Classifier implementing a vote among neighbors within a given radius.
<code>neighbors.KNeighborsRegressor([n_neighbors, ...])</code>	Regression based on k-nearest neighbors.
<code>neighbors.RadiusNeighborsRegressor([radius, ...])</code>	Regression based on neighbors within a fixed radius.
<code>neighbors.BallTree</code>	Ball Tree for fast nearest-neighbor searches :
<code>neighbors.NearestCentroid([metric, ...])</code>	Nearest centroid classifier.

### sklearn.neighbors.NearestNeighbors

```
class sklearn.neighbors.NearestNeighbors(n_neighbors=5, radius=1.0, algorithm='auto', leaf_size=30, warn_on_equidistant=True, p=2)
```

Unsupervised learner for implementing neighbor searches.

**Parameters** **n\_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 `scipy.spatial.cKDtree`
- ‘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 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.

**warn\_on\_equidistant** : boolean, optional. Defaults to True.

Generate a warning if equidistant neighbors are discarded. For classification or regression based on k-neighbors, if neighbor  $k$  and neighbor  $k+1$  have identical distances but different labels, then the result will be dependent on the ordering of the training data. If the fit method is 'kd\_tree', no warnings will be generated.

**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.

### 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](http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm)

### Examples

```
>>> 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]]...)

>>> neigh.radius_neighbors([0, 0, 1.3], 0.4, return_distance=False)
array([[2]])
```

### Methods

<code>fit(X[, y])</code>	Fit the model using X as training data
<code>get_params([deep])</code>	Get parameters for the estimator
<code>kneighbors(X[, n_neighbors, return_distance])</code>	Finds the K-neighbors of a point.
<code>kneighbors_graph(X[, n_neighbors, mode])</code>	Computes the (weighted) graph of k-Neighbors for points in X
<code>radius_neighbors(X[, radius, return_distance])</code>	Finds the neighbors within a given radius of a point or points.

Continued on next page

**Table 1.172 – continued from previous page**

<code>radius_neighbors_graph(X[, radius, mode])</code>	Computes the (weighted) graph of Neighbors for points in X
<code>set_params(**params)</code>	Set the parameters of the estimator.

**`__init__(n_neighbors=5, radius=1.0, algorithm='auto', leaf_size=30, warn_on_equidistant=True, p=2)`**

**`fit(X, y=None)`**

Fit the model using X as training data

**Parameters X :** {array-like, sparse matrix, BallTree, cKDTree}

Training data. If array or matrix, shape = [n\_samples, n\_features]

**`get_params(deep=True)`**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**`kneighbors(X, n_neighbors=None, return_distance=True)`**

Finds the K-neighbors of a point.

Returns distance

**Parameters X :** array-like, last dimension same as that of fit data

The new point.

**n\_neighbors :** int

Number of neighbors to get (default is the value passed to the constructor).

**return\_distance :** boolean, optional. Defaults to True.

If False, distances will not be returned

**Returns dist :** array

Array representing the lengths to point, 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*, *n\_neighbors=None*, *mode='connectivity'*)  
Computes the (weighted) graph of k-Neighbors for points in *X*

**Parameters** *X* : array-like, shape = [n\_samples, n\_features]  
Sample data  
*n\_neighbors* : int  
Number of neighbors for each sample. (default is value passed to the constructor).  
*mode* : {‘connectivity’, ‘distance’}, optional  
Type of returned matrix: ‘connectivity’ will return the connectivity matrix with ones and zeros, in ‘distance’ the edges are Euclidean distance between points.

**Returns** *A* : sparse 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.todense()
matrix([[ 1.,  0.,  1.],
       [ 0.,  1.,  1.],
       [ 1.,  0.,  1.]])
```

**radius\_neighbors**(*X*, *radius=None*, *return\_distance=True*)

Finds the neighbors within a given radius of a point or points.

Returns indices of and distances to the neighbors of each point.

**Parameters** *X* : array-like, last dimension same as that of fit data

The new point or points

**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

**Returns** *dist* : arrayArray representing the euclidean distances to each point, only present if *return\_distance=True*.

**ind** : array

Indices of the nearest points in the population matrix.

## 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]

```
>>> 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, ...)
>>> print(neigh.radius_neighbors([1., 1., 1.]))
(array([[ 1.5,  0.5]]...), array([[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*, *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]

Sample data

**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.todense()
matrix([[ 1.,  0.,  1.],
       [ 0.,  1.,  0.],
       [ 1.,  0.,  1.]])
```

**set\_params**(\**params*)

Set the parameters of the 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.neighbors.KNeighborsClassifier**

```
class sklearn.neighbors.KNeighborsClassifier(n_neighbors=5,           weights='uniform',
                                             algorithm='auto',      leaf_size=30,
                                             warn_on_equidistant=True, p=2)
```

Classifier implementing the k-nearest neighbors vote.

**Parameters** **n\_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 `scipy.spatial.cKDtree`
- ‘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 `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.

**warn\_on\_equidistant** : boolean, optional. Defaults to True.

Generate a warning if equidistant neighbors are discarded. For classification or regression based on k-neighbors, if neighbor  $k$  and neighbor  $k+1$  have identical distances but different labels, then the result will be dependent on the ordering of the training data. If the fit method is 'kd\_tree', no warnings will be generated.

**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.

### 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`.

[http://en.wikipedia.org/wiki/K-nearest\\_neighbor\\_algorithm](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

<code>fit(X, y)</code>	Fit the model using $X$ as training data and $y$ as target values
<code>get_params([deep])</code>	Get parameters for the estimator
<code>kneighbors(X[, n_neighbors, return_distance])</code>	Finds the K-neighbors of a point.
<code>kneighbors_graph(X[, n_neighbors, mode])</code>	Computes the (weighted) graph of k-Neighbors for points in $X$
<code>predict(X)</code>	Predict the class labels for the provided data
<code>predict_proba(X)</code>	Return probability estimates for the test data $X$ .
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

```
__init__(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30,
        warn_on_equidistant=True, p=2)
```

**fit (X, y)**

Fit the model using  $X$  as training data and  $y$  as target values

**Parameters**  $\mathbf{X}$  : {array-like, sparse matrix, BallTree, cKDTree}

Training data. If array or matrix, then the shape is [n\_samples, n\_features]

**y** : {array-like, sparse matrix}, shape = [n\_samples]

Target values, array of integer values.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**kneighbors (X, n\_neighbors=None, return\_distance=True)**

Finds the K-neighbors of a point.

Returns distance

**Parameters X** : array-like, last dimension same as that of fit data

The new point.

**n\_neighbors** : int

Number of neighbors to get (default is the value passed to the constructor).

**return\_distance** : boolean, optional. Defaults to True.

If False, distances will not be returned

**Returns dist** : array

Array representing the lengths to point, 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, n\_neighbors=None, mode='connectivity')**

Computes the (weighted) graph of k-Neighbors for points in X

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Sample data

**n\_neighbors** : int

Number of neighbors for each sample. (default is value passed to the constructor).

**mode** : {‘connectivity’, ‘distance’}, optional

Type of returned matrix: ‘connectivity’ will return the connectivity matrix with ones and zeros, in ‘distance’ the edges are Euclidean distance between points.

**Returns A** : sparse 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.todense()
matrix([[ 1.,  0.,  1.],
       [ 0.,  1.,  1.],
       [ 1.,  0.,  1.]])
```

**predict (X)**

Predict the class labels for the provided data

**Parameters X: array :**

A 2-D array representing the test points.

**Returns labels: array :**

List of class labels (one for each data sample).

**predict\_proba (X)**

Return probability estimates for the test data X.

**Parameters X: array, shape = (n\_samples, n\_features) :**

A 2-D array representing the test points.

**Returns probabilities : array, shape = [n\_samples, n\_classes]**

Probabilities of the samples for each class in the model, where classes are ordered arithmetically.

**score (X, y)**

Returns the mean accuracy on the given test data and labels.

**Parameters X : array-like, shape = [n\_samples, n\_features]**

Training set.

**y : array-like, shape = [n\_samples]**

Labels for X.

**Returns z :** float

**set\_params** (\*\*params)

Set the parameters of the 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.neighbors.RadiusNeighborsClassifier

```
class sklearn.neighbors.RadiusNeighborsClassifier(radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, p=2, outlier_label=None)
```

Classifier implementing a vote among neighbors within a given radius

**Parameters radius :** 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 `scipy.spatial.cKDtree`
- ‘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 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: 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.

**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.

#### 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](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

<code>fit(X, y)</code>	Fit the model using X as training data and y as target values
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict the class labels for the provided data
<code>radius_neighbors(X[, radius, return_distance])</code>	Finds the neighbors within a given radius of a point or points.
<code>radius_neighbors_graph(X[, radius, mode])</code>	Computes the (weighted) graph of Neighbors for points in X
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, p=2, outlier_label=None)`

`fit(X, y)`

Fit the model using X as training data and y as target values

**Parameters** `X` : {array-like, sparse matrix, BallTree, cKDTree}

Training data. If array or matrix, then the shape is [n\_samples, n\_features]

`y` : {array-like, sparse matrix}, shape = [n\_samples]

Target values, array of integer values.

`get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep`: boolean, optional :

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

**`predict(X)`**

Predict the class labels for the provided data

**Parameters X: array :**

A 2-D array representing the test points.

**Returns labels: array :**

List of class labels (one for each data sample).

**`radius_neighbors(X, radius=None, return_distance=True)`**

Finds the neighbors within a given radius of a point or points.

Returns indices of and distances to the neighbors of each point.

**Parameters X : array-like, last dimension same as that of fit data**

The new point or points

**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

**Returns dist : array**

Array representing the euclidean distances to each point, only present if `return_distance=True`.

**ind : array**

Indices of the nearest points in the population matrix.

## 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]

```
>>> 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, ...)
>>> print(neigh.radius_neighbors([1., 1., 1.]))
(array([[ 1.5,  0.5]]...), array([[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, 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]

Sample data

**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.todense()
matrix([[ 1.,  0.,  1.],
       [ 0.,  1.,  0.],
       [ 1.,  0.,  1.]])
```

**score**(X, y)

Returns the mean accuracy on the given test data and labels.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns** **z** : float

**set\_params**(\**params*)

Set the parameters of the 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.neighbors.KNeighborsRegressor**

```
class sklearn.neighbors.KNeighborsRegressor(n_neighbors=5,           weights='uniform',
                                            algorithm='auto',      leaf_size=30,
                                            warn_on_equidistant=True, p=2)
```

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.

**Parameters** `n_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 `scipy.spatial.cKDtree`
- ‘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 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.

`warn_on_equidistant` : boolean, optional. Defaults to True.

Generate a warning if equidistant neighbors are discarded. For classification or regression based on k-neighbors, if neighbor  $k$  and neighbor  $k+1$  have identical distances but different labels, then the result will be dependent on the ordering of the training data. If the fit method is ‘kd\_tree’, no warnings will be generated.

`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.

**See Also:**

<code>NearestNeighbors</code> ,	<code>RadiusNeighborsRegressor</code> ,	<code>KNeighborsClassifier</code> ,
<code>RadiusNeighborsClassifier</code>		

## 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](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

<code>fit(X, y)</code>	Fit the model using X as training data and y as target values
<code>get_params([deep])</code>	Get parameters for the estimator
<code>kneighbors(X[, n_neighbors, return_distance])</code>	Finds the K-neighbors of a point.
<code>kneighbors_graph(X[, n_neighbors, mode])</code>	Computes the (weighted) graph of k-Neighbors for points in X
<code>predict(X)</code>	Predict the target for the provided data
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, warn_on_equidistant=True, p=2)`

`fit(X, y)`

Fit the model using X as training data and y as target values

**Parameters** `X` : {array-like, sparse matrix, BallTree, cKDTree}

Training data. If array or matrix, then the shape is [n\_samples, n\_features]

`y` : {array-like, sparse matrix}, shape = [n\_samples]

Target values, array of float values.

`get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep: boolean, optional` :

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

`kneighbors(X, n_neighbors=None, return_distance=True)`

Finds the K-neighbors of a point.

Returns distance

**Parameters** `X` : array-like, last dimension same as that of fit data

The new point.

**n\_neighbors** : int

Number of neighbors to get (default is the value passed to the constructor).

**return\_distance** : boolean, optional. Defaults to True.

If False, distances will not be returned

**Returns** **dist** : array

Array representing the lengths to point, 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*, *n\_neighbors=None*, *mode='connectivity'*)

Computes the (weighted) graph of k-Neighbors for points in *X*

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Sample data

**n\_neighbors** : int

Number of neighbors for each sample. (default is value passed to the constructor).

**mode** : {‘connectivity’, ‘distance’}, optional

Type of returned matrix: ‘connectivity’ will return the connectivity matrix with ones and zeros, in ‘distance’ the edges are Euclidean distance between points.

**Returns** **A** : sparse 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.todense()
matrix([[ 1.,  0.,  1.],
       [ 0.,  1.,  1.],
       [ 1.,  0.,  1.]])
```

### `predict(X)`

Predict the target for the provided data

#### Parameters `X` : array

A 2-D array representing the test data.

#### Returns `y`: array :

List of target values (one for each data sample).

### `score(X, y)`

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  $((y - y_{\text{pred}})^2).sum()$  and  $v$  is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

#### Parameters `X` : array-like, shape = [n\_samples, n\_features]

Training set.

#### `y` : array-like, shape = [n\_samples]

#### Returns `z` : float

### `set_params(**params)`

Set the parameters of the 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.neighbors.RadiusNeighborsRegressor

```
class sklearn.neighbors.RadiusNeighborsRegressor(radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, p=2)
```

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.

#### Parameters `radius` : 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 `scipy.spatial.cKDtree`
- ‘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 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: 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.

## See Also:

<code>NearestNeighbors</code> ,	<code>KNeighborsRegressor</code> ,	<code>KNeighborsClassifier</code> ,
<code>RadiusNeighborsClassifier</code>		

## 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](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

<code>fit(X, y)</code>	Fit the model using X as training data and y as target values
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict the target for the provided data
<code>radius_neighbors(X[, radius, return_distance])</code>	Finds the neighbors within a given radius of a point or points.
<code>radius_neighbors_graph(X[, radius, mode])</code>	Computes the (weighted) graph of Neighbors for points in X
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, p=2)`

`fit(X, y)`

Fit the model using X as training data and y as target values

**Parameters** `X` : {array-like, sparse matrix, BallTree, cKDTree}

Training data. If array or matrix, then the shape is [n\_samples, n\_features]

`y` : {array-like, sparse matrix}, shape = [n\_samples]

Target values, array of float values.

`get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep: boolean, optional` :

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

`predict(X)`

Predict the target for the provided data

**Parameters** `X` : array

A 2-D array representing the test data.

**Returns y: array** :

List of target values (one for each data sample).

`radius_neighbors(X, radius=None, return_distance=True)`

Finds the neighbors within a given radius of a point or points.

Returns indices of and distances to the neighbors of each point.

**Parameters** `X` : array-like, last dimension same as that of fit data

The new point or points

`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

**Returns dist** : array

Array representing the euclidean distances to each point, only present if `return_distance=True`.

**ind** : array

Indices of the nearest points in the population matrix.

**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]

```
>>> 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, ...)
>>> print(neigh.radius_neighbors([1., 1., 1.]))
(array([[ 1.5,  0.5]]...), array([[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*, *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]

Sample data

**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.todense()
matrix([[ 1.,  0.,  1.],
       [ 0.,  1.,  0.],
       [ 1.,  0.,  1.]])
```

**score**(X, y)

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as  $(1 - \frac{u}{v})$ , where u is the regression sum of squares  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters** X : array-like, shape = [n\_samples, n\_features]

Training set.

y : array-like, shape = [n\_samples]

**Returns** z : float

**set\_params**(\*\*params)

Set the parameters of the 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.neighbors.BallTree****class sklearn.neighbors.BallTree**

Ball Tree for fast nearest-neighbor searches :

BallTree(X, leaf\_size=20, p=2.0)

**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 built ball tree. The amount of memory needed to store the tree scales as  $2^{(1 + \text{floor}(\log_2((n_{\text{samples}} - 1) / \text{leaf\_size}))) - 1}$ . For a specified leaf\_size, a leaf node is guaranteed to satisfy  $\text{leaf\_size} \leq n_{\text{points}} \leq 2 * \text{leaf\_size}$ , except in the case that  $n_{\text{samples}} < \text{leaf\_size}$ .

**p** : distance metric for the BallTree. p encodes the Minkowski

p-distance:

```
D = sum((X[i] - X[j]) ** p) ** (1. / p)
```

p must be greater than or equal to 1, so that the triangle inequality will hold. If p == np.inf, then the distance is equivalent to:

---

```
D = max(X[i] - X[j])
```

## 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
>>> ball_tree = BallTree(X, leaf_size=2)
>>> dist, ind = ball_tree.query(X[0], n_neighbors=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 ball tree (using protocol = 2). Note that the state of the tree is saved in the pickle operation: the tree is not rebuilt on un-pickling

```
>>> import numpy as np
>>> import pickle
>>> np.random.seed(0)
>>> X = np.random.random((10,3)) # 10 points in 3 dimensions
>>> ball_tree = BallTree(X, leaf_size=2)
>>> s = pickle.dumps(ball_tree, protocol=2)
>>> ball_tree_copy = pickle.loads(s)
>>> dist, ind = ball_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]
```

## Attributes

---

<code>data</code>	
<code>warning_flag</code>	

---

## Methods

---

<code>query(X[, k, return_distance])</code>	query the Ball Tree for the k nearest neighbors
<code>query_radius</code>	<code>query_radius(self, X, r, count_only = False):</code>

---

`__init__()`  
x.`__init__`(...) initializes x; see help(type(x)) for signature

`query (X, k=1, return_distance=True)`  
query the Ball Tree for the k nearest neighbors

**Parameters** `X` : array-like, last dimension self.dim

An array of points to query

**k** : integer (default = 1)  
The number of nearest neighbors to return

**return\_distance** : boolean (default = True)  
if True, return a tuple (d,i) if False, return array i

**Returns** **i** : if return\_distance == False  
**(d,i)** : if return\_distance == True  
**d** : array of doubles - shape: x.shape[:-1] + (k,)  
each entry gives the list of distances to the neighbors of the corresponding point (note that distances are not sorted)

**i** : array of integers - shape: x.shape[:-1] + (k,)  
each entry gives the list of indices of neighbors of the corresponding point (note that neighbors are not sorted)

## 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
>>> ball_tree = BallTree(X, leaf_size=2)
>>> dist, ind = ball_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]

query_radius()
query_radius(self, X, r, count_only = False):
    query the Ball Tree for neighbors within a ball of size r

Parameters X : array-like, last dimension self.dim
    An array of points to query
r : distance within which neighbors are returned
    r can be a single value, or an array of values of shape x.shape[:-1] if different radii are desired for each point.
return_distance : boolean (default = False)
    if True, return distances to neighbors of each point if False, return only neighbors Note that unlike BallTree.query(), setting return_distance=True adds to the computation time. Not all distances need to be calculated explicitly for return_distance=False. Results are not sorted by default: see sort_results keyword.
count_only : boolean (default = False)
    if True, return only the count of points within distance r if False, return the indices of all points within distance r If return_distance==True, setting count_only=True will result in an error.
sort_results : boolean (default = False)
```

if True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. If return\_distance == False, setting sort\_results = True will result in an error.

**Returns**

- count** : if count\_only == True
- ind** : if count\_only == False and return\_distance == False
- (ind, dist)** : if count\_only == False and return\_distance == True
- count** : array of integers, shape = X.shape[:-1]  
each entry gives the number of neighbors within a distance r of the corresponding point.
- ind** : array of objects, shape = X.shape[:-1]  
each element is a numpy integer array listing the indices of neighbors of the corresponding point. Note that unlike the results of BallTree.query(), the returned neighbors are not sorted by distance
- dist** : array of objects, shape = X.shape[:-1]  
each element is a numpy double array listing the distances corresponding to indices in i.

## Examples

Query for neighbors in a given radius

```
>>> import numpy as np
>>> np.random.seed(0)
>>> X = np.random.random((10,3)) # 10 points in 3 dimensions
>>> ball_tree = BallTree(X, leaf_size=2)
>>> print ball_tree.query_radius(X[0], r=0.3, count_only=True)
3
>>> ind = ball_tree.query_radius(X[0], r=0.3)
>>> print ind # indices of neighbors within distance 0.3
[3 0 1]
```

## 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.

**Parameters metric: string, or callable :**

The metric to use when calculating distance between instances in a feature array.  
If metric is a string or callable, it must be one of the options allowed by metrics.pairwise.pairwise\_distances for its metric parameter.

**shrink\_threshold** : float, optional (default = None)

Threshold for shrinking centroids to remove features.

**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]
```

## Attributes

<i>centroids_</i>	array-like, shape = [n_classes, n_features]	Centroid of each class
-------------------	---	------------------------

## Methods

<code>fit(X, y)</code>	Fit the NearestCentroid model according to the given training data.
<code>get_params(deep=False)</code>	Get parameters for the estimator
<code>predict(X)</code>	Perform classification on an array of test vectors X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(metric='euclidean', shrink_threshold=None)`

`fit(X, y)`

Fit the NearestCentroid model according to the given training data.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training vector, where n\_samples is the number of samples and n\_features is the number of features. 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 the estimator

**Parameters** `deep: boolean, optional :`

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

### **`predict(X)`**

Perform classification on an array of test vectors X.

The predicted class C for each sample in X is returned.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

**Returns** `C` : 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)`**

Returns the mean accuracy on the given test data and labels.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Training set.

`y` : array-like, shape = [n\_samples]

Labels for X.

**Returns** `z` : float

### **`set_params(**params)`**

Set the parameters of the 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` :

<code>neighbors.kneighbors_graph(X, n_neighbors[, ...])</code>	Computes the (weighted) graph of k-Neighbors for points in X
<code>neighbors.radius_neighbors_graph(X, radius)</code>	Computes the (weighted) graph of Neighbors for points in X

## **sklearn.neighbors.kneighbors\_graph**

`sklearn.neighbors.kneighbors_graph(X, n_neighbors, mode='connectivity')`

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.

**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:**

`radius_neighbors_graph`

**Examples**

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import kneighbors_graph
>>> A = kneighbors_graph(X, 2)
>>> A.todense()
matrix([[ 1.,  0.,  1.],
       [ 0.,  1.,  1.],
       [ 1.,  0.,  1.]])
```

**sklearn.neighbors.radius\_neighbors\_graph**

`sklearn.neighbors.radius_neighbors_graph(X, radius, 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 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.

**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.todense()
matrix([[ 1.,  0.,  1.],
       [ 0.,  1.,  0.],
       [ 1.,  0.,  1.]])
```

## 1.8.24 sklearn.pls: Partial Least Squares

The `sklearn.pls` module implements Partial Least Squares (PLS).

**User guide:** See the *Partial Least Squares* section for further details.

<code>pls.PLSRegression([n_components, scale, ...])</code>	PLS regression
<code>pls.PLSCanonical([n_components, scale, ...])</code>	PLSCanonical implements the 2 blocks canonical PLS of the original Wold
<code>pls.CCA([n_components, scale, max_iter, ...])</code>	CCA Canonical Correlation Analysis. CCA inherits from PLS with
<code>pls.PLSSVD([n_components, scale, copy])</code>	Partial Least Square SVD

### sklearn.pls.PLSRegression

```
class sklearn.pls.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** `X` : 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]

Training vectors, where n\_samples in the number of samples and q is the number of response variables.

`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

### Notes

For each component k, find weights u, v that optimizes:  $\max \text{corr}(X_k u, Y_k v) * \text{var}(X_k u)$   $\text{var}(Y_k u)$ , such that  $|u| = 1$

Note that it maximizes both the correlations between the scores and the intra-block variances.

The residual matrix of X ( $X_{k+1}$ ) block is obtained by the deflation on the current X score: `x_score`.

The residual matrix of Y ( $Y_{k+1}$ ) 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.pls import PLSCanonical, PLSRegression, CCA
>>> 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)
```

## Attributes

<code>x_weights_</code>	array, [p, n_components]	X block weights vectors.
<code>y_weights_</code>	array, [q, n_components]	Y block weights vectors.
<code>x_loadings_</code>	array, [p, n_components]	X block loadings vectors.
<code>y_loadings_</code>	array, [q, n_components]	Y block loadings vectors.
<code>x_scores_</code>	array, [n_samples, n_components]	X scores.
<code>y_scores_</code>	array, [n_samples, n_components]	Y scores.
<code>x_rotations_</code>	array, [p, n_components]	X block to latents rotations.
<code>y_rotations_</code>	array, [q, n_components]	Y block to latents rotations.
<code>coefs:</code> array, [p, q]		The coefficients of the linear model: $Y = X \text{ coefs} + \text{Err}$

## Methods

<code>fit(X, Y)</code>	
<code>fit_transform(X[, y])</code>	Learn and apply the dimension reduction on the train data.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X[, copy])</code>	Apply the dimension reduction learned on the train data.
<code>score(X, y)</code>	Returns the coefficient of determination $R^2$ of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

Continued on next page

**Table 1.182 – continued from previous page**


---

<code>transform(X[, Y, copy])</code>	Apply the dimension reduction learned on the train data.
--------------------------------------	--

---

**`__init__(n_components=2, scale=True, max_iter=500, tol=1e-06, copy=True)`**  
**`fit_transform(X, y=None, **fit_params)`**  
     Learn and apply the dimension reduction on the train data.

**Parameters X :** 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  
     Whether to copy X and Y, or perform in-place normalization.

**Returns x\_scores if Y is not given, (x\_scores, y\_scores) otherwise. :**

**`get_params(deep=True)`**  
     Get parameters for the estimator

**Parameters deep: boolean, optional :**  
     If True, will return the parameters for this estimator and contained subobjects that are estimators.

**`predict(X, copy=True)`**  
     Apply the dimension reduction learned on the train data.

**Parameters X :** 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  
     Whether to copy X and Y, or perform in-place normalization.

## Notes

This call require the estimation of a p x q matrix, which may be an issue in high dimensional space.

**`score(X, y)`**  
     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 - 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]  
     Training set.

**y :** array-like, shape = [n\_samples]

**Returns z :** float

**set\_params** (\*\**params*)

Set the parameters of the 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, Y=None, copy=True)

Apply the dimension reduction learned on the train data.

**Parameters** X : 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

Whether to copy X and Y, or perform in-place normalization.

**Returns** x\_scores if Y is not given, (x\_scores, y\_scores) otherwise. :

**sklearn.pls.PLSCanonical**

**class** sklearn.pls.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.

**Parameters** X : 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]

Training vectors, where n\_samples in the number of samples and q is the number of response variables.

**n\_components** : int, number of components to keep. (default 2).

**scale** : 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

**See Also:**

CCA, PLSSVD

**Notes**

For each component k, find weights u, v that optimize::  $\max \text{corr}(X_k u, Y_k v) * \text{var}(X_k u) \text{ var}(Y_k v)$ , 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 ( $X_{k+1}$ ) block is obtained by the deflation on the current X score: `x_score`.

The residual matrix of Y ( $Y_{k+1}$ ) 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 mode 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 colinear 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.pls import PLSCanonical, PLSRegression, CCA
>>> 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)
```

## Attributes

<code>x_weights_</code>	array, shape = [p, n_components]	X block weights vectors.
<code>y_weights_</code>	array, shape = [q, n_components]	Y block weights vectors.
<code>x_loadings_</code>	array, shape = [p, n_components]	X block loadings vectors.
<code>y_loadings_</code>	array, shape = [q, n_components]	Y block loadings vectors.
<code>x_scores_</code>	array, shape = [n_samples, n_components]	X scores.
<code>y_scores_</code>	array, shape = [n_samples, n_components]	Y scores.
<code>x_rotations_</code>	array, shape = [p, n_components]	X block to latents rotations.
<code>y_rotations_</code>	array, shape = [q, n_components]	Y block to latents rotations.

## Methods

`fit(X, Y)`

`fit_transform(X[, y])` Learn and apply the dimension reduction on the train data.

`get_params([deep])` Get parameters for the estimator

`predict(X[, copy])` Apply the dimension reduction learned on the train data.

`score(X, y)` Returns the coefficient of determination R^2 of the prediction.

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

`transform(X[, Y, copy])` Apply the dimension reduction learned on the train data.

`__init__(n_components=2, scale=True, algorithm='nipals', max_iter=500, tol=1e-06, copy=True)`

`fit_transform(X, y=None, **fit_params)`

Learn and apply the dimension reduction on the train data.

**Parameters** `X` : 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

Whether to copy X and Y, or perform in-place normalization.

**Returns** `x_scores` if `Y` is not given, (`x_scores`, `y_scores`) otherwise. :

`get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep: boolean, optional` :

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

`predict(X, copy=True)`

Apply the dimension reduction learned on the train data.

**Parameters** `X` : 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

Whether to copy X and Y, or perform in-place normalization.

## Notes

This call require the estimation of a p x q matrix, which may be an issue in high dimensional space.

**score**(X, y)

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  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns** **z** : float

**set\_params**(\*\*params)

Set the parameters of the 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, Y=None, copy=True)

Apply the dimension reduction learned on the train data.

**Parameters** **X** : 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

Whether to copy X and Y, or perform in-place normalization.

**Returns** **x\_scores if Y is not given, (x\_scores, y\_scores) otherwise.** :

## sklearn.pls.CCA

**class** **sklearn.pls.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".

**Parameters** **X** : 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]

Training vectors, where n\_samples in the number of samples and q is the number of response variables.

**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, default 1e-06.

the tolerance used in the iterative algorithm

**copy** : boolean

Whether the deflation be done on a copy. Let the default value to True unless you don't care about side effects

#### See Also:

PLSCanonical, PLSSVD

#### Notes

For each component k, find the weights u, v that maximizes  $\max \text{corr}(X_k u, Y_k v)$ , such that  $|u| = |v| = 1$

Note that it maximizes only the correlations between the scores.

The residual matrix of X ( $X_{k+1}$ ) block is obtained by the deflation on the current X score: x\_score.

The residual matrix of Y ( $Y_{k+1}$ ) 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.pls import PLSCanonical, PLSRegression, 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)
```

## Attributes

<code>x_weights_</code>	array, [p, n_components]	X block weights vectors.
<code>y_weights_</code>	array, [q, n_components]	Y block weights vectors.
<code>x_loadings_</code>	array, [p, n_components]	X block loadings vectors.
<code>y_loadings_</code>	array, [q, n_components]	Y block loadings vectors.
<code>x_scores_</code>	array, [n_samples, n_components]	X scores.
<code>y_scores_</code>	array, [n_samples, n_components]	Y scores.
<code>x_rotations_</code>	array, [p, n_components]	X block to latents rotations.
<code>y_rotations_</code>	array, [q, n_components]	Y block to latents rotations.

## Methods

`fit(X, Y)`

`fit_transform(X[, y])` Learn and apply the dimension reduction on the train data.

`get_params([deep])` Get parameters for the estimator

`predict(X[, copy])` Apply the dimension reduction learned on the train data.

`score(X, y)` Returns the coefficient of determination R^2 of the prediction.

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

`transform(X[, Y, copy])` Apply the dimension reduction learned on the train data.

`__init__(n_components=2, scale=True, max_iter=500, tol=1e-06, copy=True)`

`fit_transform(X, y=None, **fit_params)`

Learn and apply the dimension reduction on the train data.

**Parameters** `X` : 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

Whether to copy X and Y, or perform in-place normalization.

**Returns** `x_scores` if `Y` is not given, (`x_scores`, `y_scores`) otherwise. :

`get_params(deep=True)`

Get parameters for the estimator

**Parameters** `deep: boolean, optional` :

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

`predict(X, copy=True)`

Apply the dimension reduction learned on the train data.

**Parameters** `X` : 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

Whether to copy X and Y, or perform in-place normalization.

## Notes

This call require the estimation of a p x q matrix, which may be an issue in high dimensional space.

**score**(X, y)

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  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]**Returns** **z** : float**set\_params**(\*\*params)

Set the parameters of the 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, Y=None, copy=True)

Apply the dimension reduction learned on the train data.

**Parameters** **X** : 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

Whether to copy X and Y, or perform in-place normalization.

**Returns** **x\_scores if Y is not given, (x\_scores, y\_scores) otherwise.** :

## sklearn.pls.PLSSVD

**class** **sklearn.pls.PLSSVD**(n\_components=2, scale=True, copy=True)

Partial Least Square SVD

Simply perform a svd on the crosscovariance matrix: X'Y The are no iterative deflation here.

**Parameters** **X** : array-like of predictors, shape = [n\_samples, p]

Training vector, where n\_samples in the number of samples and p is the number of predictors. X will be centered before any analysis.

**Y** : array-like of response, shape = [n\_samples, q]

Training vector, where n\_samples in the number of samples and q is the number of response variables. X will be centered before any analysis.

**n\_components** : int, (default 2).

number of components to keep.

**scale** : boolean, (default True)

scale X and Y

#### See Also:

PLSCanonical, CCA

#### Attributes

<i>x_weights_</i>	array, [p, n_components]	X block weights vectors.
<i>y_weights_</i>	array, [q, n_components]	Y block weights vectors.
<i>x_scores_</i>	array, [n_samples, n_components]	X scores.
<i>y_scores_</i>	array, [n_samples, n_components]	Y scores.

#### Methods

**fit(X, Y)**

**fit\_transform(X[, y])** Learn and apply the dimension reduction on the train data.

**get\_params([deep])** Get parameters for the estimator

**set\_params(\*\*params)** Set the parameters of the estimator.

**transform(X[, Y])** Apply the dimension reduction learned on the train data.

**\_\_init\_\_(n\_components=2, scale=True, copy=True)**

**fit\_transform(X, y=None, \*\*fit\_params)**

Learn and apply the dimension reduction on the train data.

**Parameters** **X** : 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.

**Returns** **x\_scores** if **Y** is not given, (**x\_scores**, **y\_scores**) otherwise. :

**get\_params(deep=True)**

Get parameters for the estimator

**Parameters** **deep: boolean, optional** :

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

**set\_params(\*\*params)**

Set the parameters of the 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, Y=None)

Apply the dimension reduction learned on the train data.

## 1.8.25 sklearn.pipeline: Pipeline

The `sklearn.pipeline` module implements utilites to build a composite estimator, as a chain of transforms and estimators.

<code>pipeline.Pipeline(steps)</code>	Pipeline of transforms with a final estimator.
<code>pipeline.FeatureUnion(transformer_list[, ...])</code>	Concatenates results of multiple transformer objects.

### 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 needs only implements 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.

**Parameters steps: list :**

List of (name, transform) tuples (implementing fit/transform) that are chained, in the order in which they are chained, with the last object an estimator.

### 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 svn
```

---

```
>>> 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)
0.75
```

## Methods

<code>decision_function(X)</code>	Applies transforms to the data, and the decision_function method of the final estimator.
<code>fit(X[, y])</code>	Fit all the transforms one after the other and transform the
<code>fit_transform(X[, y])</code>	Fit all the transforms one after the other and transform the
<code>get_params([deep])</code>	
<code>inverse_transform(X)</code>	
<code>predict(X)</code>	Applies transforms to the data, and the predict method of the final estimator.
<code>predict_log_proba(X)</code>	
<code>predict_proba(X)</code>	Applies transforms to the data, and the predict_proba method of the final estimator.
<code>score(X[, y])</code>	Applies transforms to the data, and the score method of the final estimator.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Applies transforms to the data, and the transform method of the final estimator.

`__init__(steps)`

`decision_function(X)`

Applies transforms to the data, and the decision\_function method of the final estimator. Valid only if the final estimator implements decision\_function.

`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.

`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.

`predict(X)`

Applies transforms to the data, and the predict method of the final estimator. Valid only if the final estimator implements predict.

`predict_proba(X)`

Applies transforms to the data, and the predict\_proba method of the final estimator. Valid only if the final estimator implements predict\_proba.

`score(X, y=None)`

Applies transforms to the data, and the score method of the final estimator. Valid only if the final estimator implements score.

`set_params(**params)`

Set the parameters of the 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)**

Applies transforms to the data, and the transform method of the final estimator. Valid only if the final estimator implements transform.

**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.

**Parameters** `transformers: list of (name, transformer)` :

    List of transformer objects to be applied to the data.

`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 tranformers using X, transform the data and concatenate

---

`get_feature_names()`                Get feature names from all transformers.

---

`get_params([deep])`

---

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

---

`transform(X)`                        Transform X separately by each transformer, concatenate results.

---

`__init__(transformer_list, n_jobs=1, transformer_weights=None)`

**fit** (`X, y=None`)

Fit all transformers using X.

**Parameters** `X` : array-like or sparse matrix, shape (n\_samples, n\_features)

    Input data, used to fit transformers.

**fit\_transform** (`X, y=None, **fit_params`)

Fit all tranformers using X, transform the data and concatenate results.

**Parameters** `X` : array-like or sparse matrix, shape (n\_samples, n\_features)

    Input data to be transformed.

**Returns** `X_t` : array-like or sparse matrix, shape (n\_samples, sum\_n\_components)

    hstack of results of transformers. sum\_n\_components is the sum of n\_components (output dimension) over transformers.

**get\_feature\_names()**

Get feature names from all transformers.

**Returns** `feature_names` : list of strings

Names of the features produced by transform.

**set\_params** (\*\*params)

Set the parameters of the 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)

Transform X separately by each transformer, concatenate results.

**Parameters** X : array-like or sparse matrix, shape (n\_samples, n\_features)

Input data to be transformed.

**Returns** X\_t : array-like or sparse matrix, shape (n\_samples, sum\_n\_components)

hstack of results of transformers. sum\_n\_components is the sum of n\_components (output dimension) over transformers.

## 1.8.26 sklearn.preprocessing: Preprocessing and Normalization

**User guide:** See the *Preprocessing data* section for further details.

<code>preprocessing.Binarizer([threshold, copy])</code>	Binarize data (set feature values to 0 or 1) according to a threshold
<code>preprocessing.KernelCenterer</code>	Center a kernel matrix
<code>preprocessing.LabelBinarizer([neg_label, ...])</code>	Binarize labels in a one-vs-all fashion
<code>preprocessing.LabelEncoder</code>	Encode labels with value between 0 and n_classes-1.
<code>preprocessing.MinMaxScaler([feature_range, copy])</code>	Standardizes features by scaling each feature to a given range.
<code>preprocessing.Normalizer([norm, copy])</code>	Normalize samples individually to unit norm
<code>preprocessing.OneHotEncoder([n_values, dtype])</code>	Encode categorical integer features using a one-hot aka one-of-K scheme
<code>preprocessing.StandardScaler([copy, ...])</code>	Standardize features by removing the mean and scaling to unit variance

### 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

The default threshold is 0.0 so that any non-zero values are set to 1.0 and zeros are left untouched.

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. modeled using the Bernoulli distribution in a Bayesian setting).

**Parameters** **threshold** : float, optional (0.0 by default)

The lower bound that triggers feature values to be replaced by 1.0.

**copy** : boolean, optional, default is 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

<code>fit(X[, y])</code>	Do nothing and return the estimator unchanged
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, y, copy])</code>	Binarize each element of X

`__init__(threshold=0.0, copy=True)`

`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.

**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 the estimator

**Parameters** `deep: boolean, optional` :

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

`set_params(**params)`

Set the parameters of the 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, y=None, copy=None)`

Binarize each element of X

**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 format to avoid an un-necessary copy.

## sklearn.preprocessing.KernelCenterer

```
class sklearn.preprocessing.KernelCenterer
    Center a kernel matrix
```

Let  $K(x_i, x_j)$  be a kernel defined by  $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ , where  $\phi(x)$  is a function mapping  $x$  to a hilbert space. KernelCenterer is a class to center (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

<code>fit(K[, y])</code>	Fit KernelCenterer
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(K[, y, copy])</code>	Center kernel

`__init__()`

`x.__init__(...)` initializes `x`; see `help(type(x))` for signature

`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 the estimator

**Parameters** `deep: boolean, optional :`

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

**set\_params** (\*\**params*)

Set the parameters of the 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**(*K*, *y*=None, *copy*=True)

Center kernel

**Parameters** **K** : numpy array of shape [n\_samples1, n\_samples2]

Kernel matrix.

**Returns** **K\_new** : numpy array of shape [n\_samples1, n\_samples2]

## sklearn.preprocessing.LabelBinarizer

**class** sklearn.preprocessing.LabelBinarizer(*neg\_label*=0, *pos\_label*=1)

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.

**Parameters** **neg\_label**: int (default: 0) :

Value with which negative labels must be encoded.

**pos\_label**: int (default: 1) :

Value with which positive labels must be encoded.

## Examples

```
>>> from sklearn import preprocessing
>>> lb = preprocessing.LabelBinarizer()
>>> lb.fit([1, 2, 6, 4, 2])
LabelBinarizer(neg_label=0, pos_label=1)
>>> lb.classes_
array([1, 2, 4, 6])
>>> lb.transform([1, 6])
array([[1, 0, 0, 0],
       [0, 0, 0, 1]])

>>> lb.fit_transform([(1, 2), (3,)])
array([[1, 1, 0],
       [0, 0, 1]])
>>> lb.classes_
array([1, 2, 3])
```

## Attributes

<code>classes_</code> : array of shape [n_class]	Holds the label for each class.
--	---------------------------------

## Methods

<code>fit(y)</code>	Fit label binarizer
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>inverse_transform(Y[, threshold])</code>	Transform binary labels back to multi-class labels
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(y)</code>	Transform multi-class labels to binary labels

`__init__(neg_label=0, pos_label=1)`

`fit(y)`

Fit label binarizer

**Parameters** `y` : numpy array of shape [n\_samples] or sequence of sequences

Target values. In the multilabel case the nested sequences can have variable lengths.

**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 the estimator

**Parameters** `deep: boolean, optional` :

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

`inverse_transform(Y, threshold=None)`

Transform binary labels back to multi-class labels

**Parameters** `Y` : numpy array of shape [n\_samples, n\_classes]

Target values.

`threshold` : float or None

Threshold used in the binary and multi-label cases.

**Use 0 when:**

- Y contains the output of decision\_function (classifier)

**Use 0.5 when:**

- Y contains the output of predict\_proba

If None, the threshold is assumed to be half way between neg\_label and pos\_label.

**Returns y :** numpy array of shape [n\_samples] or sequence of sequences

Target values. In the multilabel case the nested sequences can have variable lengths.

**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 the 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(y)**

Transform multi-class labels to binary labels

The output of transform is sometimes referred to by some authors as the 1-of-K coding scheme.

**Parameters y :** numpy array of shape [n\_samples] or sequence of sequences

Target values. In the multilabel case the nested sequences can have variable lengths.

**Returns Y :** numpy array of shape [n\_samples, n\_classes]

## sklearn.preprocessing.LabelEncoder

**class sklearn.preprocessing.LabelEncoder**

Encode labels with value between 0 and n\_classes-1.

**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']
```

## Attributes

<i>classes_</i> :	array of shape [n_class]	Holds the label for each class.
-------------------	--------------------------	---------------------------------

## Methods

<b>fit(y)</b>	Fit label encoder
<b>fit_transform(y)</b>	Fit label encoder and return encoded labels
<b>get_params([deep])</b>	Get parameters for the estimator
<b>inverse_transform(y)</b>	Transform labels back to original encoding.
<b>set_params(**params)</b>	Set the parameters of the estimator.
<b>transform(y)</b>	Transform labels to normalized encoding.

**\_\_init\_\_()**  
x.\_\_init\_\_(...) initializes x; see help(type(x)) for signature

**fit(y)**  
Fit label encoder

**Parameters** **y** : array-like of shape [n\_samples]

Target values.

**Returns** **self** : returns an instance of self.

**fit\_transform(y)**  
Fit label encoder and return encoded labels

**Parameters** **y** : array-like of shape [n\_samples]

Target values.

**Returns** **y** : array-like of shape [n\_samples]

**get\_params(deep=True)**  
Get parameters for the estimator

**Parameters** **deep: boolean, optional** :

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

**inverse\_transform(y)**  
Transform labels back to original encoding.

**Parameters** `y` : numpy array of shape [n\_samples]

Target values.

**Returns** `y` : numpy array of shape [n\_samples]

**set\_params** (`**params`)

Set the parameters of the 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** (`y`)

Transform labels to normalized encoding.

**Parameters** `y` : array-like of shape [n\_samples]

Target values.

**Returns** `y` : array-like of shape [n\_samples]

## 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:**  $X_{\text{std}} = (X - X.\min(\text{axis}=0)) / (X.\max(\text{axis}=0) - X.\min(\text{axis}=0))$   $X_{\text{scaled}} = X_{\text{std}} / (\max - \min) + \min$

where  $\min, \max = \text{feature\_range}$ .

This standardization is often used as an alternative to zero mean, unit variance scaling.

**Parameters** `feature_range: tuple (min, max), default=(0, 1)` :

Desired range of transformed data.

`copy` : boolean, optional, default is True

Set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array).

### Attributes

<code>min_</code>	ndarray, shape (n_features,)	Per feature adjustment for minimum.
<code>scale_</code>	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 the estimator

Continued on next page

**Table 1.194 – continued from previous page**

<code>inverse_transform(X)</code>	Undo the scaling of X according to feature_range.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Scaling features of X according to feature_range.

**`__init__(feature_range=(0, 1), copy=True)`**

**`fit(X, y=None)`**

Compute the minimum and maximum to be used for later scaling.

**Parameters X :** 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.

**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 the estimator

**Parameters deep: boolean, optional :**

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

**`inverse_transform(X)`**

Undo the scaling of X according to feature\_range.

**Parameters X :** array-like with shape [n\_samples, n\_features]

Input data that will be transformed.

**`set_params(**params)`**

Set the parameters of the 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)`**

Scaling features of X according to feature\_range.

**Parameters X :** array-like with shape [n\_samples, n\_features]

Input data that will be transformed.

## 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.

**Parameters** `norm` : ‘l1’ or ‘l2’, optional (‘l2’ by default)

The norm to use to normalize each non zero sample.

`copy` : boolean, optional, default is True

set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR matrix).

### 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

<code>fit(X[, y])</code>	Do nothing and return the estimator unchanged
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, y, copy])</code>	Scale each non zero row of X to unit norm

`__init__(norm='l2', copy=True)`

`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.

**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 the estimator

**Parameters deep: boolean, optional :**

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

**set\_params (\*\*params)**

Set the parameters of the 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, y=None, copy=None)**

Scale each non zero row of X to unit norm

**Parameters X** : 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.

## sklearn.preprocessing.OneHotEncoder

**class sklearn.preprocessing.OneHotEncoder (n\_values='auto', dtype=<type 'float'>)**

Encode categorical integer features using a one-hot aka one-of-K scheme.

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 were 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 scikit-learn estimators.

**Parameters n\_values** : ‘auto’, int or array of int

Number of values per feature. ‘auto’ : determine value range from training data. int : maximum value for all features. array : maximum value per feature.

**dtype** : number type, default=np.float

Desired dtype of output.

**See Also:**

**LabelEncoder** performs a one-hot encoding on arbitrary class labels.

**sklearn.feature\_extraction.DictVectorizer** performs a one-hot encoding of dictionary items (also handles string-valued features).

## 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(dtype=<type 'float'>, n_values='auto')
>>> 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.]])
```

## Attributes

<i>active_features_</i>	array	Indices for active features, meaning values that actually occur in the training set. Only available when <i>n_values</i> is 'auto'.
<i>feature_indices_</i>	array of shape (n_features,)	Indices to feature ranges. Feature <i>i</i> in the original data is mapped to features from <i>feature_indices_[i]</i> to <i>feature_indices_[i+1]</i> (and then potentially masked by <i>active_features_</i> afterwards)
<i>n_values</i>	array of shape (n_features,)	Maximum number of values per feature.

## Methods

<code>fit(X[, y])</code>	Fit OneHotEncoder to X.
<code>fit_transform(X[, y])</code>	Fit OneHotEncoder to X, then transform X.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X)</code>	Transform X using one-hot encoding.

`__init__(n_values='auto', dtype=<type 'float'>)`

`fit(X, y=None)`

Fit OneHotEncoder to X.

**Parameters** `X` : array-like, shape=(n\_samples, n\_feature)

Input array of type int.

**Returns self :**

`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 the estimator

**Parameters deep: boolean, optional :**

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

**set\_params (\*\*params)**

Set the parameters of the 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)**

Transform X using one-hot encoding.

**Parameters X : array-like, shape=(n\_samples, feature\_indices\_[-1])**

Input array of type int.

**Returns X\_out : sparse matrix, dtype=int**

Transformed input.

**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 than others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

**Parameters 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 is True**

Set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR matrix and if axis is 1).

**See Also:**

`sklearn.preprocessing.scale`, `scaling`, `sklearn.decomposition.RandomizedPCA`, to

## Attributes

<code>mean_</code>	array of floats with shape [n_features]	The mean value for each feature in the training set.
<code>std_</code>	array of floats with shape [n_features]	The standard deviation for each feature in the training set.

## Methods

<code>fit(X[, y])</code>	Compute the mean and std to be used for later scaling.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>inverse_transform(X[, copy])</code>	Scale back the data to the original representation
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, y, copy])</code>	Perform standardization by centering and scaling

`__init__(copy=True, with_mean=True, with_std=True)`

`fit(X, y=None)`

Compute the mean and std to be used for later scaling.

**Parameters** `X` : 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.

**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 the estimator

**Parameters** `deep: boolean, optional` :

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

`inverse_transform(X, copy=None)`

Scale back the data to the original representation

**Parameters** `X` : array-like with shape [n\_samples, n\_features]

The data used to scale along the features axis.

`set_params(**params)`

Set the parameters of the 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*, *y=None*, *copy=None*)

Perform standardization by centering and scaling

**Parameters X** : array-like with shape [n\_samples, n\_features]

The data used to scale along the features axis.

---

<code>preprocessing.add_dummy_feature(X[, value])</code>	Augment dataset with an additional dummy feature.
<code>preprocessing.binarize(X[, threshold, copy])</code>	Boolean thresholding of array-like or <code>scipy.sparse</code> matrix
<code>preprocessing.normalize(X[, norm, axis, copy])</code>	Normalize a dataset along any axis
<code>preprocessing.scale(X[, axis, with_mean, ...])</code>	Standardize a dataset along any axis

---

## sklearn.preprocessing.add\_dummy\_feature

`sklearn.preprocessing.add_dummy_feature(X, value=1.0)`

Augment dataset with an additional dummy feature.

This is useful for fitting an intercept term with implementations which cannot otherwise fit it directly.

**Parameters X** : array or `scipy.sparse` matrix with shape [n\_samples, n\_features]

Data.

**value** : float

Value to use for the dummy feature.

**Returns X** : array or `scipy.sparse` matrix with shape [n\_samples, n\_features + 1]

Same data with dummy feature added as first column.

## Examples

```
>>> from sklearn.preprocessing import add_dummy_feature
>>> add_dummy_feature([[0, 1], [1, 0]])
array([[ 1.,  0.,  1.],
       [ 1.,  1.,  0.]])
```

## 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)

The lower bound that triggers feature values to be replaced by 1.0.

**copy** : boolean, optional, default is 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`

## `sklearn.preprocessing.normalize`

`sklearn.preprocessing.normalize(X, norm='l2', axis=1, copy=True)`

Normalize a dataset along any axis

**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 is True

set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a `scipy.sparse` CSR matrix and if axis is 1).

**See Also:**

`sklearn.preprocessing.Normalizer`, using, `sklearn.pipeline.Pipeline`

## `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.

**Parameters** `X` : 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 is True

set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a `scipy.sparse` CSR matrix and if axis is 1).

**See Also:**

`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.

## 1.8.27 `sklearn.qda`: Quadratic Discriminant Analysis

Quadratic Discriminant Analysis

**User guide:** See the *Linear and Quadratic Discriminant Analysis* section for further details.

---

`qda.QDA([priors])`    Quadratic Discriminant Analysis (QDA)

---

### `sklearn.qda.QDA`

**class** `sklearn.qda.QDA(priors=None)`  
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.

**Parameters** `priors` : array, optional, shape = [n\_classes]

Priors on classes

**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)
>>> print(clf.predict([[ -0.8, -1]]))
[1]
```

## Attributes

<i>co-vari-ances_</i>	list of array-like, shape = [n_features, n_features]	Covariance matrices of each class.
<i>means_</i>	array-like, shape = [n_classes, n_features]	Class means.
<i>pri-ors_</i>	array-like, shape = [n_classes]	Class priors (sum to 1).
<i>rota-tions_</i>	list of arrays	For each class an array of shape [n_samples, n_samples], the rotation of the Gaussian distribution, i.e. its principal axis.
<i>scal-ings_</i>	array-like, shape = [n_classes, n_features]	Contains the scaling of the Gaussian distributions along the principal axes for each class, i.e. the variance in the rotated coordinate system.

## Methods

<code>decision_function(X)</code>	Apply decision function to an array of samples.
<code>fit(X, y[, store_covariances, tol])</code>	Fit the QDA model according to the given training data and parameters.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Perform classification on an array of test vectors X.
<code>predict_log_proba(X)</code>	Return posterior probabilities of classification.
<code>predict_proba(X)</code>	Return posterior probabilities of classification.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

**`__init__(priors=None)`**

**`decision_function(X)`**

Apply decision function to an array of samples.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Array of samples (test vectors).

**Returns** `C` : 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.

**`fit(X, y, store_covariances=False, tol=0.0001)`**

Fit the QDA model according to the given training data and parameters.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

Training vector, where n\_samples is the number of samples and n\_features is the number of features.

`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.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**predict (X)**

Perform classification on an array of test vectors X.

The predicted class C for each sample in X is returned.

**Parameters X : array-like, shape = [n\_samples, n\_features]****Returns C : array, shape = [n\_samples]****predict\_log\_proba (X)**

Return posterior probabilities of classification.

**Parameters X : array-like, shape = [n\_samples, n\_features]**

Array of samples/test vectors.

**Returns C : array, shape = [n\_samples, n\_classes]**

Posterior log-probabilities of classification per class.

**predict\_proba (X)**

Return posterior probabilities of classification.

**Parameters X : array-like, shape = [n\_samples, n\_features]**

Array of samples/test vectors.

**Returns C : array, shape = [n\_samples, n\_classes]**

Posterior probabilities of classification per class.

**score (X, y)**

Returns the mean accuracy on the given test data and labels.

**Parameters X : array-like, shape = [n\_samples, n\_features]**

Training set.

**y : array-like, shape = [n\_samples]**

Labels for X.

**Returns z : float****set\_params (\*\*params)**

Set the parameters of the 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 :**

## 1.8.28 sklearn.random\_projection: Random projection

Random Projection transformers

Random Projections are a simple and computationally efficient way to reduce the dimensionality of the data by trading a controlled amount of accuracy (as additional variance) for faster processing times and smaller model sizes.

The dimensions and distribution of Random Projections matrices are controlled so as to preserve the pairwise distances between any two samples of the dataset.

The main theoretical result behind the efficiency of random projection is the Johnson-Lindenstrauss lemma (quoting Wikipedia):

In mathematics, the Johnson-Lindenstrauss lemma is a result concerning low-distortion embeddings of points from high-dimensional into low-dimensional Euclidean space. The lemma states that a small set of points in a high-dimensional space can be embedded into a space of much lower dimension in such a way that distances between the points are nearly preserved. The map used for the embedding is at least Lipschitz, and can even be taken to be an orthogonal projection.

**User guide:** See the *Random Projection* section for further details.

---

<code>random_projection.GaussianRandomProjection([...])</code>	Reduce dimensionality through Gaussian random projection
<code>random_projection.SparseRandomProjection([...])</code>	Reduce dimensionality through sparse random projection

---

## 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  $N(0, 1 / n_{\text{components}})$ .

**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.

**See Also:**

`SparseRandomProjection`

## Attributes

n_components	<code>int</code>	Concrete number of components computed when n_components="auto".
components	<code>numpy array of shape [n_components, n_features]</code>	Random matrix used for the projection.

## Methods

<code>fit(X[, y])</code>	Generate a sparse random projection matrix
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, y])</code>	Project the data by using matrix product with the random matrix

`__init__(n_components='auto', eps=0.1, random_state=None)`

**fit** (*X*, *y*=*None*)

Generate a sparse random projection matrix

**Parameters** *X* : numpy array or scipy.sparse of shape [n\_samples, n\_features]

Training set: only the shape is used to find optimal random matrix dimensions based on the theory referenced in the afore mentioned papers.

*y* : is not used: placeholder to allow for usage in a Pipeline.

**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.

**Returns** *X\_new* : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

`get_params(deep=True)`

Get parameters for the estimator

**Parameters** *deep*: boolean, optional :

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

`set_params(**params)`

Set the parameters of the 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, y=None)

Project the data by using matrix product with the random matrix

**Parameters X** : numpy array or scipy.sparse of shape [n\_samples, n\_features]

The input data to project into a smaller dimensional space.

**y** : is not used: placeholder to allow for usage in a Pipeline.

**Returns X\_new** : numpy array or scipy sparse of shape [n\_samples, n\_components]

Projected array.

## 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  $s = 1 / \text{density}$  the components of the random matrix are drawn from:

- $-\sqrt{s} / \sqrt{n_{\text{components}}}$  with probability  $1 / 2s$
- 0 with probability  $1 - 1 / s$
- $+\sqrt{s} / \sqrt{n_{\text{components}}}$  with probability  $1 / 2s$

**Parameters n\_components** : int or ‘auto’, optional (default = ‘auto’)

Dimensionality of the target projection space.

n\_components can be automatically adjusted according to the number of samples in the dataset and the bound given by the Johnson-Lindenstrauss lemma. In that case the quality of the embedding is controlled by the eps parameter.

It should be noted that Johnson-Lindenstrauss lemma can yield very conservative estimated of the required number of components as it makes no assumption on the structure of the dataset.

**density** : float in range ]0, 1], optional (default=’auto’)

Ratio of non-zero component in the random projection matrix.

If density = ‘auto’, the value is set to the minimum density as recommended by Ping Li et al.:  $1 / \sqrt{n_{\text{features}}}$ .

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.

### See Also:

GaussianRandomProjection

### References

[R110], [R111]

### Attributes

n_components	int	Concrete number of components computed when n_components="auto".
components_	CSR matrix with shape [n_components, n_features]	Random matrix used for the projection.
density_	float in range 0.0 - 1.0	Concrete density computed from when density = "auto".

### Methods

<code>fit(X[, y])</code>	Generate a sparse random projection matrix
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, y])</code>	Project the data by using matrix product with the random matrix

`__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

**Parameters** X : numpy array or scipy.sparse of shape [n\_samples, n\_features]

Training set: only the shape is used to find optimal random matrix dimensions based on the theory referenced in the afore mentioned papers.

y : is not used: placeholder to allow for usage in a Pipeline.

**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.

**Returns** **X\_new** : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params** (*deep=True*)

Get parameters for the estimator

**Parameters** **deep**: boolean, optional :

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

**set\_params** (\*\**params*)

Set the parameters of the 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*, *y=None*)

Project the data by using matrix product with the random matrix

**Parameters** **X** : numpy array or scipy.sparse of shape [n\_samples, n\_features]

The input data to project into a smaller dimensional space.

**y** : is not used: placeholder to allow for usage in a Pipeline.

**Returns** **X\_new** : numpy array or scipy sparse of shape [n\_samples, n\_components]

Projected array.

`random_projection.johnson_lindenstrauss_min_dim(...)` Find a ‘safe’ number of components to randomly project

---

## 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  $(1 \pm \text{eps})$  in an euclidean space with good probability. The projection  $p$  is an eps-embedding as defined by:

$$(1 - \text{eps}) \|u - v\|^2 < \|p(u) - p(v)\|^2 < (1 + \text{eps}) \|u - v\|^2$$

Where  $u$  and  $v$  are any rows taken from a dataset of shape [n\_samples, n\_features],  $\text{eps}$  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:

$$\text{n\_components} \geq 4 \log(n_{\text{samples}}) / (\text{eps}^2 / 2 - \text{eps}^3 / 3)$$

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.

**Parameters n\_samples** : int or numpy array of int greater than 0,

Number of samples. If an array is given, it will compute a safe number of components array-wise.

**eps** : float or numpy array of float in ]0,1[, optional (default=0.1)

Maximum distortion rate as defined by the Johnson-Lindenstrauss lemma. If an array is given, it will compute a safe number of components array-wise.

**Returns n\_components** : int or numpy array of int,

The minimal number of components to guarantee with good probability an eps-embedding with n\_samples.

## References

[R112], [R113]

## 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])
```

## 1.8.29 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.

## Estimators

<code>svm.SVC([C, kernel, degree, gamma, coef0, ...])</code>	C-Support Vector Classification.
<code>svm.LinearSVC([penalty, loss, dual, tol, C, ...])</code>	Linear Support Vector Classification.
<code>svm.NuSVC([nu, kernel, degree, gamma, ...])</code>	Nu-Support Vector Classification.
<code>svm.SVR([kernel, degree, gamma, coef0, tol, ...])</code>	epsilon-Support Vector Regression.
<code>svm.NuSVR([nu, C, kernel, degree, gamma, ...])</code>	Nu Support Vector Regression.
<code>svm.OneClassSVM([kernel, degree, gamma, ...])</code>	Unsupervised Outliers 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)
```

C-Support Vector Classification.

The implementations is a 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, 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 kernel function. It is significant only in ‘poly’ and ‘sigmoid’.

**gamma** : float, optional (default=0.0)

Kernel coefficient for ‘rbf’ and ‘poly’. 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 predict\_proba.

**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

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.

**See Also:**

**SVR**Support Vector Machine for Regression implemented using libsvm.

**LinearSVC**Scalable Linear Support Vector Machine for classification implemented using liblinear. Check the See also section of LinearSVC for more comparison element.

## 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,
     shrinking=True, tol=0.001, verbose=False)
>>> print(clf.predict([[[-0.8, -1]]]))
[1]
```

## Attributes

<i>sup-port_</i>	array-like, shape = [n_SV]	Index of support vectors.
<i>sup-port_vectors</i>	array-like, shape [n_SV, n_features]	Support vectors.
<i>n_support</i>	array-like, dtype=int32, shape = [n_class]	number of support vector for each class.
<i>dual_coef_</i>	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.
<i>coef_</i>	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. <i>coef_</i> is readonly property derived from <i>dual_coef_</i> and <i>support_vectors_</i>
<i>intercept_</i>	array, shape = [n_class * (n_class-1) / 2]	Constants in decision function.

## Methods

<i>decision_function</i> (X)	Distance of the samples X to the separating hyperplane.
<i>fit</i> (X, y[, sample_weight])	Fit the SVM model according to the given training data.
<i>get_params</i> ([deep])	Get parameters for the estimator
<i>predict</i> (X)	Perform classification or regression samples in X.
<i>predict_log_proba</i> (X)	Compute log probabilities of possible outcomes for samples in X.
<i>predict_proba</i> (X)	Compute probabilities of possible outcomes for samples in X.
<i>score</i> (X, y)	Returns the mean accuracy on the given test data and labels.
<i>set_params</i> (**params)	Set the parameters of the estimator.

```
__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)
```

***decision\_function*(X)**

Distance of the samples X to the separating hyperplane.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

**Returns** **X** : 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.

**Parameters** **X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features.

**y** : array-like, shape = [n\_samples]

Target values (integers in classification, real numbers in regression)

**sample\_weight** : array-like, shape = [n\_samples], optional

Weights applied to individual samples (1. for unweighted).

**Returns self** : 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 the estimator

**Parameters** **deep**: boolean, optional :

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

**label\_**

DEPRECATED: The `labels_` attribute has been renamed to `classes_` for consistency and will be removed in 0.15.

**predict** (*X*)

Perform classification or regression samples in X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the function value of X calculated is returned.

For an one-class model, +1 or -1 is returned.

**Parameters** **X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

**Returns** **y\_pred** : array, shape = [n\_samples]

**predict\_log\_proba** (*X*)

Compute log probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute `probability` set to True.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

**Returns X** : array-like, shape = [n\_samples, n\_classes]

Returns the log-probabilities of the sample for each class in the model, where classes are ordered by arithmetical order.

## Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will produce meaningless results on very small datasets.

**predict\_proba (X)**

Compute probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute *probability* set to True.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

**Returns X** : array-like, shape = [n\_samples, n\_classes]

Returns the probability of the sample for each class in the model, where classes are ordered by arithmetical order.

## Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will produce meaningless results on very small datasets.

**score (X, y)**

Returns the mean accuracy on the given test data and labels.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns z** : float

**set\_params (\*\*params)**

Set the parameters of the 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.svm.LinearSVC

```
class sklearn.svm.LinearSVC(penalty='l2', loss='l2', 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)
```

Linear Support Vector Classification.

Similar to SVC with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better (to large numbers of samples).

This class supports both dense and sparse input and the multiclass support is handled according to a one-vs-the-rest scheme.

**Parameters** **C** : float, optional (default=1.0)

Penalty parameter C of the error term.

**loss** : string, 'l1' or 'l2' (default='l2')

Specifies the loss function. 'l1' is the hinge loss (standard SVM) while 'l2' is the squared 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.

**See Also:**

**SVC** Implementation of Support Vector Machine classifier using libsvm: the kernel can be non-linear but its SMO algorithm does not scale to large number of samples as LinearSVC does. Furthermore SVC multi-class mode is implemented using one vs one scheme while LinearSVC uses one vs the rest. It is possible to implement one vs the rest with SVC by using the `sklearn.multiclass.OneVsRestClassifier` wrapper. Finally SVC can fit dense data without memory copy if the input is C-contiguous. Sparse data will still incur memory copy though.

**sklearn.linear\_model.SGDClassifier** SGDClassifier can optimize the same cost function as LinearSVC by adjusting the penalty and loss parameters. Furthermore SGDClassifier is scalable to large number of samples as it uses a Stochastic Gradient Descent optimizer. Finally SGDClassifier can fit both dense and sparse data without memory copy if the input is C-contiguous or CSR.

## 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.

**References:** [LIBLINEAR: A Library for Large Linear Classification](#)

## Attributes

<code>coef_</code>	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. <code>coef_</code> is readonly property derived from <code>raw_coef_</code> that follows the internal memory layout of liblinear.
<code>intercept_</code>	array, shape = [1] if n_classes == 2 else [n_classes]	Constants in decision function.

## Methods

<code>decision_function(X)</code>	Predict confidence scores for samples.
<code>fit(X, y)</code>	Fit the model according to the given training data.
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class labels for samples in X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

`__init__(penalty='l2', loss='l2', 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)`

`decision_function(X)`

Predict confidence scores for samples.

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

**Parameters** **X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Samples.

**Returns** array, shape = [n\_samples] if n\_classes == 2 else [n\_samples,n\_classes] :

Confidence scores per (sample, class) combination. In the binary case, confidence score for the “positive” class.

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

Fit the model according to the given training data.

**Parameters** **X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training vector, where n\_samples in the number of samples and n\_features is the number of features.

**y** : array-like, shape = [n\_samples]

Target vector relative to X

**class\_weight** : {dict, ‘auto’}, optional

Weights associated with classes. If not given, all classes are supposed to have weight one.

**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.

**Returns** **X\_new** : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params** (*deep=True*)

Get parameters for the estimator

**Parameters** **deep**: boolean, optional :

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

**label\_**

DEPRECATED: The `labels_` attribute has been renamed to `classes_` for consistency and will be removed in 0.15.

**predict** (*X*)

Predict class labels for samples in X.

**Parameters** **X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Samples.

**Returns** **C** : array, shape = [n\_samples]

Predicted class label per sample.

**score**(*X*, *y*)

Returns the mean accuracy on the given test data and labels.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns** **z** : float

**set\_params**(\**params*)

Set the parameters of the 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.

**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.

**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)
```

Nu-Support Vector Classification.

Similar to SVC but uses a parameter to control the number of support vectors.

The implementation is based on libsvm.

**Parameters** **nu** : float, optional (default=0.5)

An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1].

**kernel** : string, optional (default='rbf')

Specifies the kernel type to be used in the algorithm. It must be one of ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’ or a callable. If none is given, ‘rbf’ will be used. If a callable is given it is used to precompute the kernel matrix.

**degree** : int, optional (default=3)

degree of 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 predict\_proba.

**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.

#### 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, shrinking=True, tol=0.001,
      verbose=False)
>>> print(clf.predict([[ -0.8, -1]]))
[1]
```

## Attributes

<code>sup-port_</code>	array-like, shape = [n_SV]	Index of support vectors.
<code>sup-port_vectors</code>	array-like, shape [n_SV, n_features]	Support vectors.
<code>n_support</code>	array-like, dtype=int32, shape = [n_class]	number of support vector for each class.
<code>dual_coef_</code>	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.
<code>coef_</code>	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. <code>coef_</code> is readonly property derived from <code>dual_coef_</code> and <code>support_vectors_</code> .
<code>intercept_</code>	array, shape = [n_class * (n_class-1) / 2]	Constants in decision function.

## Methods

<code>decision_function(X)</code>	Distance of the samples X to the separating hyperplane.
<code>fit(X, y[, sample_weight])</code>	Fit the SVM model according to the given training data.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Perform classification or regression samples in X.
<code>predict_log_proba(X)</code>	Compute log probabilities of possible outcomes for samples in X.
<code>predict_proba(X)</code>	Compute probabilities of possible outcomes for samples in X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.

**`__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)`**

**`decision_function(X)`**

Distance of the samples X to the separating hyperplane.

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

**Returns** `X` : 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.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features.

`y` : array-like, shape = [n\_samples]

Target values (integers in classification, real numbers in regression)

**sample\_weight** : array-like, shape = [n\_samples], optional

Weights applied to individual samples (1. for unweighted).

**Returns self** : 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 the estimator

**Parameters deep: boolean, optional :**

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

**label\_**

DEPRECATED: The `labels_` attribute has been renamed to `classes_` for consistency and will be removed in 0.15.

**predict (X)**

Perform classification or regression samples in X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the function value of X calculated is returned.

For an one-class model, +1 or -1 is returned.

**Parameters X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

**Returns y\_pred** : array, shape = [n\_samples]

**predict\_log\_proba (X)**

Compute log probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute *probability* set to True.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

**Returns X** : array-like, shape = [n\_samples, n\_classes]

Returns the log-probabilities of the sample for each class in the model, where classes are ordered by arithmetical order.

## Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will produce meaningless results on very small datasets.

**predict\_proba (X)**

Compute probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute *probability* set to True.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

**Returns** **X** : array-like, shape = [n\_samples, n\_classes]

Returns the probability of the sample for each class in the model, where classes are ordered by arithmetical order.

## Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will produce meaningless results on very small datasets.

**score** (*X*, *y*)

Returns the mean accuracy on the given test data and labels.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for *X*.

**Returns** **z** : float

**set\_params** (\*\**params*)

Set the parameters of the 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.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, probability=False, cache_size=200, verbose=False, max_iter=-1)
```

epsilon-Support Vector Regression.

The free parameters in the model are *C* and *epsilon*.

The implementations is a based on libsvm.

**Parameters** **C** : 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.

**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 predict\_proba.

**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.

## See Also:

**NuSVR** Support Vector Machine for regression implemented using libsvm using a parameter to control the number of support vectors.

## 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, probability=False, shrinking=True, tol=0.001,
     verbose=False)
```

## Attributes

<i>sup-port_</i>	array-like, shape = [n_SV]	Index of support vectors.
<i>sup-port_vectors</i>	array-like, shape [nSV, n_features]	Support vectors.
<i>dual_coef_</i>	array, shape = [n_classes-1, n_SV]	Coefficients of the support vector in the decision function.
<i>coef_</i>	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. <i>coef_</i> is readonly property derived from <i>dual_coef_</i> and <i>support_vectors_</i>
<i>intercept_</i>	array, shape = [n_class * (n_class-1) / 2]	Constants in decision function.

## Methods

<code>decision_function(X)</code>	Distance of the samples X to the separating hyperplane.
<code>fit(X, y[, sample_weight])</code>	Fit the SVM model according to the given training data.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Perform classification or regression samples in X.
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__init__(kernel='rbf', degree=3, gamma=0.0, coef0=0.0, tol=0.001, C=1.0, epsilon=0.1, shrinking=True, probability=False, 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]

**Returns** **X** : 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.

**Parameters** **X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features.

**y** : array-like, shape = [n\_samples]

Target values (integers in classification, real numbers in regression)

**sample\_weight** : array-like, shape = [n\_samples], optional

Weights applied to individual samples (1. for unweighted).

**Returns self** : 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 the estimator

#### **Parameters** `deep: boolean, optional :`

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

### `label_`

DEPRECATED: The `labels_` attribute has been renamed to `classes_` for consistency and will be removed in 0.15.

### `predict(X)`

Perform classification or regression samples in X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the function value of X calculated is returned.

For an one-class model, +1 or -1 is returned.

**Parameters** `X : {array-like, sparse matrix}, shape = [n_samples, n_features]`

**Returns** `y_pred : array, shape = [n_samples]`

### `score(X, y)`

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  $((y - y_{\text{pred}})^2 \cdot \text{sum}())$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2 \cdot \text{sum}())$ . Best possible score is 1.0, lower values are worse.

**Parameters** `X : array-like, shape = [n_samples, n_features]`

Training set.

`y : array-like, shape = [n_samples]`

**Returns** `z : float`

### `set_params(**params)`

Set the parameters of the 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.svm.NuSVR`

```
class sklearn.svm.NuSVR(nu=0.5, C=1.0, 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)
```

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 with the parameter epsilon of SVR.

The implementation is based on libsvm.

**Parameters** **C** : 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. Only available if `impl='nu_svc'`.

**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 `predict_proba`.

**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.

## See Also:

**NuSVC** Support Vector Machine for classification implemented with libsvm with a parameter to control the number of support vectors.

**SVR** Epsilon Support Vector Machine for regression implemented with libsvm.

## 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, probability=False, shrinking=True, tol=0.001,
       verbose=False)
```

## Attributes

<i>sup-port_</i>	array-like, shape = [n_SV]	Index of support vectors.
<i>sup-port_vectors</i>	[nSV, n_features]	Support vectors.
<i>dual_coef_</i>	array, shape = [n_classes-1, n_SV]	Coefficients of the support vector in the decision function.
<i>coef_</i>	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. <i>coef_</i> is readonly property derived from <i>dual_coef_</i> and <i>support_vectors_</i> .
<i>intercept_</i>	array, shape = [n_class * (n_class-1) / 2]	Constants in decision function.

## Methods

<i>decision_function</i> (X)	Distance of the samples X to the separating hyperplane.
<i>fit</i> (X, y[, sample_weight])	Fit the SVM model according to the given training data.
<i>get_params</i> ([deep])	Get parameters for the estimator
<i>predict</i> (X)	Perform classification or regression samples in X.
<i>score</i> (X, y)	Returns the coefficient of determination R^2 of the prediction.
<i>set_params</i> (**params)	Set the parameters of the estimator.

***\_\_init\_\_***(nu=0.5, C=1.0, 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)

***decision\_function***(X)

Distance of the samples X to the separating hyperplane.

**Parameters** X : array-like, shape = [n\_samples, n\_features]

**Returns** X : 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.

**Parameters** **X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features.

**y** : array-like, shape = [n\_samples]

Target values (integers in classification, real numbers in regression)

**sample\_weight** : array-like, shape = [n\_samples], optional

Weights applied to individual samples (1. for unweighted).

**Returns self** : 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 the estimator

**Parameters deep: boolean, optional :**

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

**label\_**

DEPRECATED: The `labels_` attribute has been renamed to `classes_` for consistency and will be removed in 0.15.

**predict**(*X*)

Perform classification or regression samples in X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the function value of X calculated is returned.

For an one-class model, +1 or -1 is returned.

**Parameters** **X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

**Returns y\_pred** : array, shape = [n\_samples]

**score**(*X*, *y*)

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  $((y - y_{\text{pred}})^2 \cdot \text{sum}())$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2 \cdot \text{sum}())$ . Best possible score is 1.0, lower values are worse.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params** (\*\**params*)

Set the parameters of the 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.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)
```

Unsupervised Outliers Detection.

Estimate the support of a high-dimensional distribution.

The implementation is based on libsvm.

**Parameters** **kernel** : string, optional (default='rbf')

Specifies the kernel type to be used in the algorithm. It must be one of ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’ or a callable. If none is given, ‘rbf’ will be used. If a callable is given it is used to precompute the kernel matrix.

**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

Degree of kernel function. 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

Independent term in kernel function. It is only significant in poly/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.

## Attributes

<i>sup-port_</i>	array-like, shape = [n_SV]	Index of support vectors.
<i>sup-port_vectors</i>	array-like, shape [nSV, n_features]	Support vectors.
<i>dual_coef_</i>	array, shape = [n_classes-1, n_SV]	Coefficient of the support vector in the decision function.
<i>coef_</i>	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. <i>coef_</i> is readonly property derived from <i>dual_coef_</i> and <i>support_vectors_</i>
<i>intercept_</i>	array, shape = [n_classes-1]	Constants in decision function.

## Methods

<code>decision_function(X)</code>	Distance of the samples X to the separating hyperplane.
<code>fit(X[, sample_weight])</code>	Detects the soft boundary of the set of samples X.
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Perform classification or regression samples in X.
<code>set_params(**params)</code>	Set the parameters of the estimator.

`__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)`

**decision\_function(X)**

Distance of the samples X to the separating hyperplane.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

**Returns** **X** : 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, sample\_weight=None, \*\*params)**

Detects the soft boundary of the set of samples X.

**Parameters** **X** : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

Set of samples, where n\_samples is the number of samples and n\_features is the number of features.

**Returns self** : object

Returns self.

## Notes

If X is not a C-ordered contiguous array it is copied.

**get\_params(deep=True)**

Get parameters for the estimator

**Parameters** **deep**: boolean, optional :

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

**label\_**

DEPRECATED: The `labels_` attribute has been renamed to `classes_` for consistency and will be removed in 0.15.

**predict (X)**

Perform classification or regression samples in X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the function value of X calculated is returned.

For an one-class model, +1 or -1 is returned.

**Parameters** `X` : {array-like, sparse matrix}, shape = [n\_samples, n\_features]

**Returns** `y_pred` : array, shape = [n\_samples]

**set\_params (\*\*params)**

Set the parameters of the 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 :**

---

`svm.l1_min_c(X, y[, loss, fit_intercept, ...])` Return the lowest bound for C such that for C in (l1\_min\_C, infinity)

---

**sklearn.svm.l1\_min\_c**

`sklearn.svm.l1_min_c(X, y, loss='l2', fit_intercept=True, intercept_scaling=1.0)`

Return the lowest bound for C such that for C in (l1\_min\_C, infinity) the model is guaranteed not to be empty. This applies to l1 penalized classifiers, such as LinearSVC with `penalty='l1'` and `linear_model.LogisticRegression` with `penalty='l1'`.

This value is valid if `class_weight` parameter in `fit()` is not set.

**Parameters** `X` : array-like or sparse matrix, 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` : {'l2', 'log'}, default to 'l2'

Specifies the loss function. With 'l2' it is the l2 loss (a.k.a. squared hinge loss). With 'log' it is the loss of logistic regression models.

`fit_intercept` : bool, default: True

Specifies if the intercept should be fitted by the model. It must match the `fit()` method parameter.

`intercept_scaling` : float, default: 1

when `fit_intercept` is True, instance vector x becomes [x, `intercept_scaling`], i.e. a "synthetic" feature with constant value equals to `intercept_scaling` is appended to the instance vector. It must match the `fit()` method parameter.

**Returns l1\_min\_c: float :**

minimum value for C

## Low-level methods

<code>svm.libsvm.fit</code>	Train the model using libsvm (low-level method)
<code>svm.libsvm.decision_function</code>	Predict margin (libsvm name for this is predict_values)
<code>svm.libsvm.predict</code>	Predict target values of X given a model (low-level method)
<code>svm.libsvm.predict_proba</code>	Predict probabilities svm_model stores all parameters needed to predict a given value.
<code>svm.libsvm.cross_validation</code>	Binding of the cross-validation routine (low-level routine)

### sklearn.svm.libsvm.fit

`sklearn.svm.libsvm.fit()`

Train the model using libsvm (low-level method)

**Parameters X: array-like, dtype=float64, size=[n\_samples, n\_features] :**

**Y: array, dtype=float64, size=[n\_samples] :**

target vector

**svm\_type : {0, 1, 2, 3, 4}**

Type of SVM: C\_SVC, NuSVC, OneClassSVM, EpsilonSVR or NuSVR respectevely.

**kernel : {‘linear’, ‘rbf’, ‘poly’, ‘sigmoid’, ‘precomputed’}**

Kernel to use in the model: linear, polynomial, RBF, sigmoid or precomputed.

**degree : int32**

Degree of the polynomial kernel (only relevant if kernel is set to polynomial)

**gamma : float64**

Gamma parameter in RBF kernel (only relevant if kernel is set to RBF)

**coef0 : float64**

Independent parameter in poly/sigmoid kernel.

**tol : float64**

Numeric stopping criterion (WRITEME).

**C : float64**

C parameter in C-Support Vector Classification

**nu : float64**

**cache\_size : float64**

Cache size for gram matrix columns (in megabytes)

**max\_iter: int (-1 for no limit) :**

Stop solver after this many iterations regardless of accuracy (XXX Currently there is no API to know whether this kicked in.)

**Returns support : array, shape=[n\_support]**

index of support vectors

**support\_vectors** : array, shape=[n\_support, n\_features]

support vectors (equivalent to X[support]). Will return an empty array in the case of precomputed kernel.

**n\_class\_SV** : array

number of support vectors in each class.

**sv\_coef** : array

coefficients of support vectors in decision function.

**intercept** : array

intercept in decision function

**label** : labels for different classes (only relevant in classification).

**probA, probB** : array

probability estimates, empty array for probability=False

### **sklearn.svm.libsvm.decision\_function**

`sklearn.svm.libsvm.decision_function()`

Predict margin (libsvm name for this is predict\_values)

We have to reconstruct model and parameters to make sure we stay in sync with the python object.

### **sklearn.svm.libsvm.predict**

`sklearn.svm.libsvm.predict()`

Predict target values of X given a model (low-level method)

**Parameters X: array-like, dtype=float, size=[n\_samples, n\_features] :**

**svm\_type** : {0, 1, 2, 3, 4}

Type of SVM: C SVC, nu SVC, one class, epsilon SVR, nu SVR

**kernel** : {'linear', 'rbf', 'poly', 'sigmoid', 'precomputed'}

Kernel to use in the model: linear, polynomial, RBF, sigmoid or precomputed.

**degree** : int

Degree of the polynomial kernel (only relevant if kernel is set to polynomial)

**gamma** : float

Gamma parameter in RBF kernel (only relevant if kernel is set to RBF)

**coef0** : float

Independent parameter in poly/sigmoid kernel.

**eps** : float

Stopping criteria.

**C** : float

C parameter in C-Support Vector Classification

**Returns** `dec_values` : array

predicted values.

**TODO:** probably there's no point in setting some parameters, like :

`cache_size` or `weights`. :

### `sklearn.svm.libsvm.predict_proba`

`sklearn.svm.libsvm.predict_proba()`

Predict probabilities

`svm_model` stores all parameters needed to predict a given value.

For speed, all real work is done at the C level in function `copy_predict` (`libsvm_helper.c`).

We have to reconstruct model and parameters to make sure we stay in sync with the python object.

See `sklearn.svm.predict` for a complete list of parameters.

**Parameters** `X`: array-like, `dtype=float` :

**Y**: array :

target vector

`kernel` : {‘linear’, ‘rbf’, ‘poly’, ‘sigmoid’, ‘precomputed’}

**Returns** `dec_values` : array

predicted values.

### `sklearn.svm.libsvm.cross_validation`

`sklearn.svm.libsvm.cross_validation()`

Binding of the cross-validation routine (low-level routine)

**Parameters** `X`: array-like, `dtype=float`, `size=[n_samples, n_features]` :

**Y**: array, `dtype=float`, `size=[n_samples]` :

target vector

`svm_type` : {0, 1, 2, 3, 4}

Type of SVM: C SVC, nu SVC, one class, epsilon SVR, nu SVR

`kernel` : {‘linear’, ‘rbf’, ‘poly’, ‘sigmoid’, ‘precomputed’}

Kernel to use in the model: linear, polynomial, RBF, sigmoid or precomputed.

`degree` : int

Degree of the polynomial kernel (only relevant if kernel is set to polynomial)

`gamma` : float

Gamma parameter in RBF kernel (only relevant if kernel is set to RBF)

`coef0` : float

Independent parameter in poly/sigmoid kernel.

`tol` : float

Stopping criteria.

**C** : float

C parameter in C-Support Vector Classification

**nu** : float

**cache\_size** : float

**Returns** **target** : array, float

### 1.8.30 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.

<code>tree.DecisionTreeClassifier([criterion, ...])</code>	A decision tree classifier.
<code>tree.DecisionTreeRegressor([criterion, ...])</code>	A tree regressor.
<code>tree.ExtraTreeClassifier([criterion, ...])</code>	An extremely randomized tree classifier.
<code>tree.ExtraTreeRegressor([criterion, ...])</code>	An extremely randomized tree regressor.

#### sklearn.tree.DecisionTreeClassifier

```
class sklearn.tree.DecisionTreeClassifier(criterion='gini', max_depth=None,
                                         min_samples_split=2, min_samples_leaf=1,
                                         min_density=0.1, max_features=None, compute_importances=False, random_state=None)
```

A decision tree classifier.

**Parameters** **criterion** : string, optional (default="gini")

The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

**max\_features** : int, string or None, optional (default=None)

**The number of features to consider when looking for the best split:**

- If "auto", then `max_features=sqrt(n_features)` on classification tasks and `max_features=n_features` on regression problems.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

**max\_depth** : integer or None, optional (default=None)

The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than `min_samples_split` samples.

**min\_samples\_split** : 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\_density** : float, optional (default=0.1)

This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the *sample\_mask* (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If *min\_density* equals to one, the partitions are always represented as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks).

**compute\_importances** : boolean, optional (default=False)

Whether feature importances are computed and stored into the *feature\_importances\_* attribute when calling fit.

**random\_state** : int, RandomState instance or None, optional (default=None)

If int, *random\_state* is the seed used by the random number generator; If RandomState instance, *random\_state* is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

## See Also:

[DecisionTreeRegressor](#)

## References

[R114], [R115], [R116], [R117]

## 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...,  0.93...,  1.        ,  0.93...,  1.        ])
```

## Attributes

<i>tree_</i>	Tree object	The underlying Tree object.
<i>classes_</i>	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).
<i>n_classes_</i>	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).
<i>feature_importances_</i>	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 error brought by that feature. It is also known as the Gini importance [R117].

## Methods

<code>fit(X, y[, sample_mask, X_argsorted, ...])</code>	Build a decision tree from the training set (X, y).
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class or regression value for X.
<code>predict_log_proba(X)</code>	Predict class log-probabilities of the input samples X.
<code>predict_proba(X)</code>	Predict class probabilities of the input samples X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

`__init__(criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_density=0.1, max_features=None, compute_importances=False, random_state=None)`

`fit(X, y, sample_mask=None, X_argsorted=None, check_input=True, sample_weight=None)`  
Build a decision tree from the training set (X, y).

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

The training input samples. Use `dtype=np.float32` and `order='F'` for maximum efficiency.

`y` : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]

The target values (integers that correspond to classes in classification, real numbers in regression). Use `dtype=np.float64` and `order='C'` for maximum efficiency.

`sample_mask` : array-like, shape = [n\_samples], dtype = bool or None

A bit mask that encodes the rows of `X` that should be used to build the decision tree. It can be used for bagging without the need to create of copy of `X`. If `None` a mask will be created that includes all samples.

`X_argsorted` : array-like, shape = [n\_samples, n\_features] or `None`

Each column of `X_argsorted` holds the row indices of `X` sorted according to the value of the corresponding feature in ascending order. I.e.  $X[X_{\text{argsorted}}[i, k], k] \leq X[X_{\text{argsorted}}[j, k], k]$  for each  $j > i$ . If `None`, `X_argsorted` is computed internally. The argument is supported to enable multiple decision trees to share the data structure and to avoid re-computation in tree ensembles. For maximum efficiency use `dtype np.int32`.

`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.

**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.

**Returns** **X\_new** : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params**(deep=True)

Get parameters for the estimator

**Parameters** **deep**: boolean, optional :

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

**predict**(X)

Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

**Parameters** **X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns** **y** : 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.

**Parameters** **X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns** **p** : 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. Classes are ordered by arithmetical order.

**predict\_proba**(X)

Predict class probabilities of the input samples X.

**Parameters** **X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**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. Classes are ordered by arithmetical order.

**score**(X, y)

Returns the mean accuracy on the given test data and labels.

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

Labels for X.

**Returns z** : float

**set\_params** (\*\*params)

Set the parameters of the 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.

**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.

## sklearn.tree.DecisionTreeRegressor

```
class sklearn.tree.DecisionTreeRegressor(criterion='mse', max_depth=None,
                                         min_samples_split=2, min_samples_leaf=1,
                                         min_density=0.1, max_features=None, compute_importances=False, random_state=None)
```

A tree regressor.

**Parameters 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.

**max\_features** : int, string or None, optional (default=None)

**The number of features to consider when looking for the best split:**

- If “auto”, then  $\text{max\_features}=\sqrt{n\_features}$  on classification tasks and  $\text{max\_features}=n\_features$  on regression problems.
- If “sqrt”, then  $\text{max\_features}=\sqrt{n\_features}$ .
- If “log2”, then  $\text{max\_features}=\log_2(n\_features)$ .
- If None, then  $\text{max\_features}=n\_features$ .

**max\_depth** : integer or None, optional (default=None)

The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min\_samples\_split samples.

**min\_samples\_split** : 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\_density** : float, optional (default=0.1)

This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the *sample\_mask* (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If *min\_density* equals to one, the partitions are always represented as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks).

**compute\_importances** : boolean, optional (default=True)

Whether feature importances are computed and stored into the `feature_importances_` attribute when calling fit.

**random\_state** : int, RandomState instance or None, optional (default=None)

If int, random\_state is the seed used by the random number generator; If RandomState instance, random\_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

## See Also:

`DecisionTreeClassifier`

## References

[R118], [R119], [R120], [R121]

## 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)
```

R2 scores (a.k.a. coefficient of determination) over 10-folds CV:

```
>>> 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...])
```

## Attributes

<code>tree_</code>	Tree object	The underlying Tree object.
<code>feature_importances_</code>	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 error brought by that feature. It is also known as the Gini importance [R121].

## Methods

<code>fit(X, y[, sample_mask, X_argsorted, ...])</code>	Build a decision tree from the training set (X, y).
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class or regression value for X.
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

`__init__(criterion='mse', max_depth=None, min_samples_split=2, min_samples_leaf=1,`

`min_density=0.1, max_features=None, compute_importances=False, random_state=None)`

`fit(X, y, sample_mask=None, X_argsorted=None, check_input=True, sample_weight=None)`

Build a decision tree from the training set (X, y).

**Parameters** `X` : array-like, shape = [n\_samples, n\_features]

The training input samples. Use `dtype=np.float32` and `order='F'` for maximum efficiency.

`y` : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]

The target values (integers that correspond to classes in classification, real numbers in regression). Use `dtype=np.float64` and `order='C'` for maximum efficiency.

`sample_mask` : array-like, shape = [n\_samples], `dtype` = bool or None

A bit mask that encodes the rows of `X` that should be used to build the decision tree. It can be used for bagging without the need to create of copy of `X`. If None a mask will be created that includes all samples.

`X_argsorted` : array-like, shape = [n\_samples, n\_features] or None

Each column of `X_argsorted` holds the row indices of `X` sorted according to the value of the corresponding feature in ascending order. I.e.  $X[X_{\text{argsorted}}[i, k], k] \leq X[X_{\text{argsorted}}[j, k], k]$  for each  $j > i$ . If None, `X_argsorted` is computed internally. The argument is supported to enable multiple decision trees to share the data structure and to avoid re-computation in tree ensembles. For maximum efficiency use `dtype np.int32`.

`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.

**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.

**Returns X\_new** : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params**(deep=True)

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**predict**(X)

Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

**Parameters X** : array-like of shape = [n\_samples, n\_features]

The input samples.

**Returns y** : array of shape = [n\_samples] or [n\_samples, n\_outputs]

The predicted classes, or the predict values.

**score**(X, y)

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  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters X** : array-like, shape = [n\_samples, n\_features]

Training set.

**y** : array-like, shape = [n\_samples]

**Returns z** : float

**set\_params**(\*\*params)

Set the parameters of the 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.

**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.

## sklearn.tree.ExtraTreeClassifier

```
class sklearn.tree.ExtraTreeClassifier(criterion='gini', max_depth=None,
                                         min_samples_split=2, min_samples_leaf=1,
                                         min_density=0.1, max_features='auto', compute_importances=False, random_state=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.

**See Also:**

[ExtraTreeRegressor](#), [ExtraTreesClassifier](#), [ExtraTreesRegressor](#)

## References

[R122]

## Methods

<code>fit(X, y[, sample_mask, X_argsorted, ...])</code>	Build a decision tree from the training set (X, y).
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class or regression value for X.
<code>predict_log_proba(X)</code>	Predict class log-probabilities of the input samples X.
<code>predict_proba(X)</code>	Predict class probabilities of the input samples X.
<code>score(X, y)</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

---

```
__init__(criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1,
        min_density=0.1, max_features='auto', compute_importances=False, random_state=None)
```

**fit(X, y, sample\_mask=None, X\_argsorted=None, check\_input=True, sample\_weight=None)**

Build a decision tree from the training set (X, y).

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

The training input samples. Use `dtype=np.float32` and `order='F'` for maximum efficiency.

**y** : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]

The target values (integers that correspond to classes in classification, real numbers in regression). Use `dtype=np.float64` and `order='C'` for maximum efficiency.

**sample\_mask** : array-like, shape = [n\_samples], dtype = bool or None

A bit mask that encodes the rows of X that should be used to build the decision tree. It can be used for bagging without the need to create of copy of X. If None a mask will be created that includes all samples.

**X\_argsorted** : array-like, shape = [n\_samples, n\_features] or None

Each column of `X_argsorted` holds the row indices of X sorted according to the value of the corresponding feature in ascending order. I.e. `X[X_argsorted[i, k], k] <= X[X_argsorted[j, k], k]` for each `j > i`. If None, `X_argsorted` is computed internally. The argument is supported to enable multiple decision trees to share the data structure and to avoid re-computation in tree ensembles. For maximum efficiency use `dtype np.int32`.

**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.

**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.

**Returns** **X\_new** : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**predict (X)**

Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

**Parameters X : array-like of shape = [n\_samples, n\_features]**

The input samples.

**Returns y : 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.

**Parameters X : array-like of shape = [n\_samples, n\_features]**

The input samples.

**Returns p : 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. Classes are ordered by arithmetical order.

**predict\_proba (X)**

Predict class probabilities of the input samples X.

**Parameters X : array-like of shape = [n\_samples, n\_features]**

The input samples.

**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. Classes are ordered by arithmetical order.

**score (X, y)**

Returns the mean accuracy on the given test data and labels.

**Parameters X : array-like, shape = [n\_samples, n\_features]**

Training set.

**y : array-like, shape = [n\_samples]**

Labels for X.

**Returns z : float****set\_params (\*\*params)**

Set the parameters of the 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.

**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.

**sklearn.tree.ExtraTreeRegressor**

```
class sklearn.tree.ExtraTreeRegressor(criterion='mse', max_depth=None,
                                      min_samples_split=2, min_samples_leaf=1,
                                      min_density=0.1, max_features='auto', com-
                                      pute_importances=False, random_state=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**A classifier base on extremely randomized trees

**sklearn.ensemble.ExtraTreesClassifier**An ensemble of extra-trees for classification

**sklearn.ensemble.ExtraTreesRegressor**An ensemble of extra-trees for regression

**References**

[R123]

**Methods**

<code>fit(X, y[, sample_mask, X_argsorted, ...])</code>	Build a decision tree from the training set (X, y).
<code>fit_transform(X[, y])</code>	Fit to data, then transform it
<code>get_params([deep])</code>	Get parameters for the estimator
<code>predict(X)</code>	Predict class or regression value for X.
<code>score(X, y)</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(**params)</code>	Set the parameters of the estimator.
<code>transform(X[, threshold])</code>	Reduce X to its most important features.

```
__init__(criterion='mse', max_depth=None, min_samples_split=2, min_samples_leaf=1,
        min_density=0.1, max_features='auto', compute_importances=False, random_state=None)
```

```
fit(X, y, sample_mask=None, X_argsorted=None, check_input=True, sample_weight=None)
```

Build a decision tree from the training set (X, y).

**Parameters** **X** : array-like, shape = [n\_samples, n\_features]

The training input samples. Use `dtype=np.float32` and `order='F'` for maximum efficiency.

**y** : array-like, shape = [n\_samples] or [n\_samples, n\_outputs]

The target values (integers that correspond to classes in classification, real numbers in regression). Use `dtype=np.float64` and `order='C'` for maximum efficiency.

**sample\_mask** : array-like, shape = [n\_samples], dtype = bool or None

A bit mask that encodes the rows of X that should be used to build the decision tree. It can be used for bagging without the need to create of copy of X. If None a mask will be created that includes all samples.

**X\_argsorted** : array-like, shape = [n\_samples, n\_features] or None

Each column of `X_argsorted` holds the row indices of X sorted according to the value of the corresponding feature in ascending order. I.e. `X[X_argsorted[i, k], k] <= X[X_argsorted[j, k], k]` for each `j > i`. If None, `X_argsorted` is computed internally. The argument is supported to enable multiple decision trees to share the data structure and to avoid re-computation in tree ensembles. For maximum efficiency use `dtype np.int32`.

**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.

**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.

**Returns X\_new** : numpy array of shape [n\_samples, n\_features\_new]

Transformed array.

**get\_params (deep=True)**

Get parameters for the estimator

**Parameters deep: boolean, optional :**

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

**predict (X)**

Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

**Parameters X : array-like of shape = [n\_samples, n\_features]**

The input samples.

**Returns y : array of shape = [n\_samples] or [n\_samples, n\_outputs]**

The predicted classes, or the predict values.

**score (X, y)**

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  $((y - y_{\text{pred}})^2).sum()$  and v is the residual sum of squares  $((y_{\text{true}} - y_{\text{true}}.\text{mean()})^2).sum()$ . Best possible score is 1.0, lower values are worse.

**Parameters X : array-like, shape = [n\_samples, n\_features]**

Training set.

**y : array-like, shape = [n\_samples]****Returns z : float****set\_params (\*\*params)**

Set the parameters of the 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.

**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.

---

`tree.export_graphviz(decision_tree[, ...])` Export a decision tree in DOT format.

---

## sklearn.tree.export\_graphviz

`sklearn.tree.export_graphviz(decision_tree, out_file=None, feature_names=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)
```

**Parameters** `decision_tree` : decision tree classifier

The decision tree to be exported to graphviz.

`out` : file object or string, optional (default=None)

Handle or name of the output file.

`feature_names` : list of strings, optional (default=None)

Names of each of the features.

**Returns** `out_file` : file object

The file object to which the tree was exported. The user is expected to `close()` this object when done with it.

## Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn import tree

>>> clf = tree.DecisionTreeClassifier()
>>> iris = load_iris()

>>> clf = clf.fit(iris.data, iris.target)
>>> import tempfile
>>> out_file = tree.export_graphviz(clf, out_file=tempfile.TemporaryFile())
>>> out_file.close()
```

## 1.8.31 sklearn.utils: Utilities

The `sklearn.utils` module includes various utilites.

**Developer guide:** See the *Utilities for Developers* page for further details.

---

<code>utils.check_random_state(seed)</code>	Turn seed into a <code>np.random.RandomState</code> instance
<code>utils.resample(*arrays, **options)</code>	Resample arrays or sparse matrices in a consistent way
<code>utils.shuffle(*arrays, **options)</code>	Shuffle arrays or sparse matrices in a consistent way

---

**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.

**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 arrays or scipy.sparse matrices with same shape[0]

`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.

**Returns** 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 = [[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.],
       [ 0.,  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])
```

## 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 arrays or `scipy.sparse` matrices with same shape[0]

`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.

**Returns** 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 = [[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])
```



# EXAMPLE GALLERY

## 2.1 Examples

### 2.1.1 General examples

General-purpose and introductory examples for the scikit.

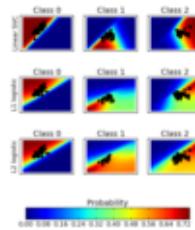
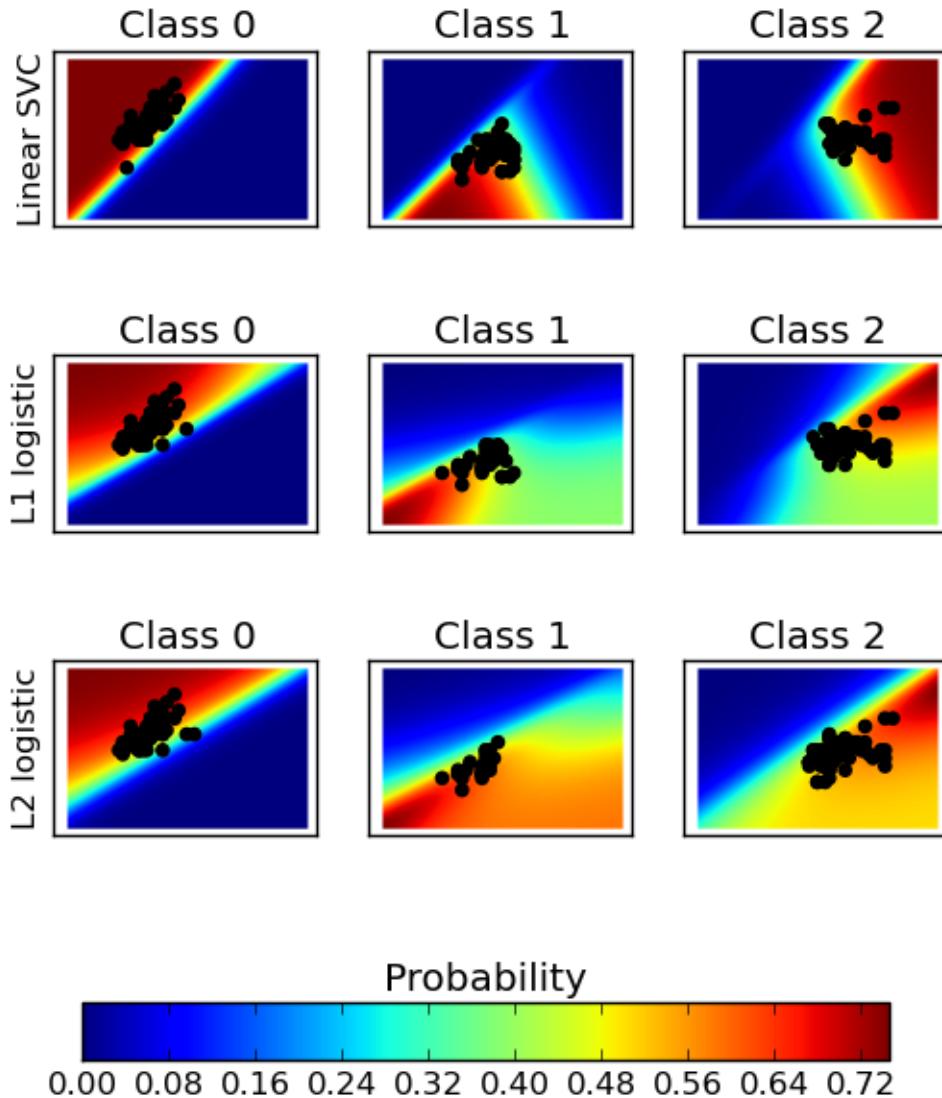


Figure 2.1: Plot classification probability

#### 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, as well as L1 and L2 penalized logistic regression.

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 Linear SVC : 82.000000
classif_rate for L1 logistic : 79.333333
classif_rate for L2 logistic : 76.666667
```

**Python source code:** [plot\\_classification\\_probability.py](#)

```
print __doc__

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD Style.

import pylab as pl
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': LogisticRegression(C=C, penalty='l2'),
               'Linear SVC': SVC(kernel='linear', C=C, probability=True)}

n_classifiers = len(classifiers)

pl.figure(figsize=(3 * 2, n_classifiers * 2))
pl.subplots_adjust(bottom=.2, top=.95)

for index, (name, classifier) in enumerate(classifiers.iteritems()):
    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))

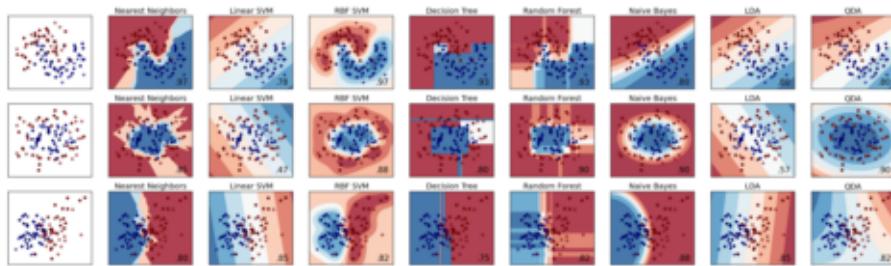
    # View probabilities=
    xx = np.linspace(3, 9, 100)
    yy = np.linspace(1, 5, 100).T
    xx, yy = np.meshgrid(xx, yy)
    Xfull = np.c_[xx.ravel(), yy.ravel()]
    probas = classifier.predict_proba(Xfull)
    n_classes = np.unique(y_pred).size
    for k in range(n_classes):
        pl.subplot(n_classifiers, n_classes, index * n_classes + k + 1)
        pl.title("Class %d" % k)
        if k == 0:
            pl.ylabel(name)
        imshow_handle = pl.imshow(probas[:, k].reshape((100, 100)),
                                  extent=(3, 9, 1, 5), origin='lower')
        pl.xticks(())
        pl.yticks(())
        idx = (y_pred == k)
        if idx.any():
            pl.scatter(X[idx, 0], X[idx, 1], marker='o', c='k')

ax = pl.axes([0.15, 0.04, 0.7, 0.05])
pl.title("Probability")
pl.colorbar(imshow_handle, cax=ax, orientation='horizontal')

pl.show()

```

**Total running time of the example:** 4.09 seconds

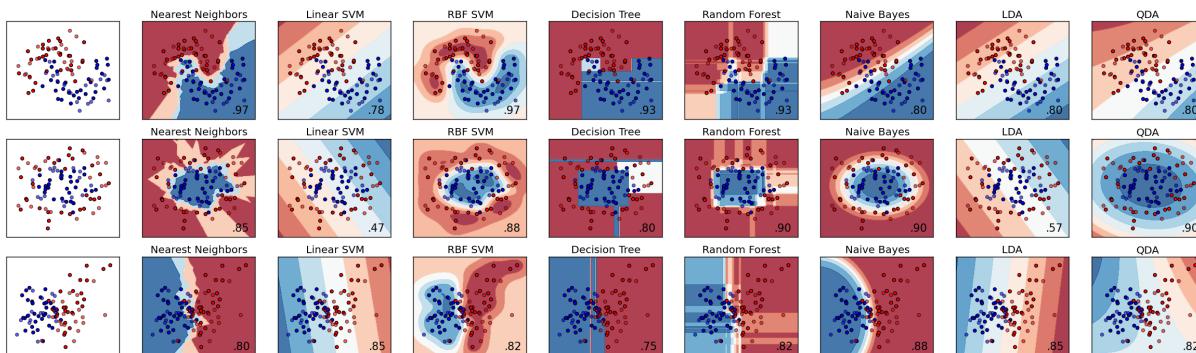
Figure 2.2: *Classifiers Comparison*

## Classifiers 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.

In particular 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.

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: Gael Varoquaux
#              Andreas Mueller
# Modified for Documentation merge by Jaques Grobler
# License: BSD

import numpy as np
import pylab as pl
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
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", "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),
    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 = pl.figure(figsize=(24, 8))
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 = pl.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])
    ax = pl.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 = pl.subplot(len(datasets), len(classifiers) + 1, i)
    clf.fit(X_train, y_train)
    score = clf.score(X_test, y_test)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    if hasattr(clf, "decision_function"):
        Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
    else:
        Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    ax.contourf(xx, yy, Z, cmap=cm, alpha=.8)

    # Plot also the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
               # 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)
pl.show()
```

**Total running time of the example:** 3.60 seconds

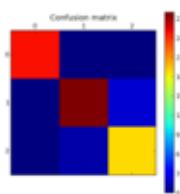
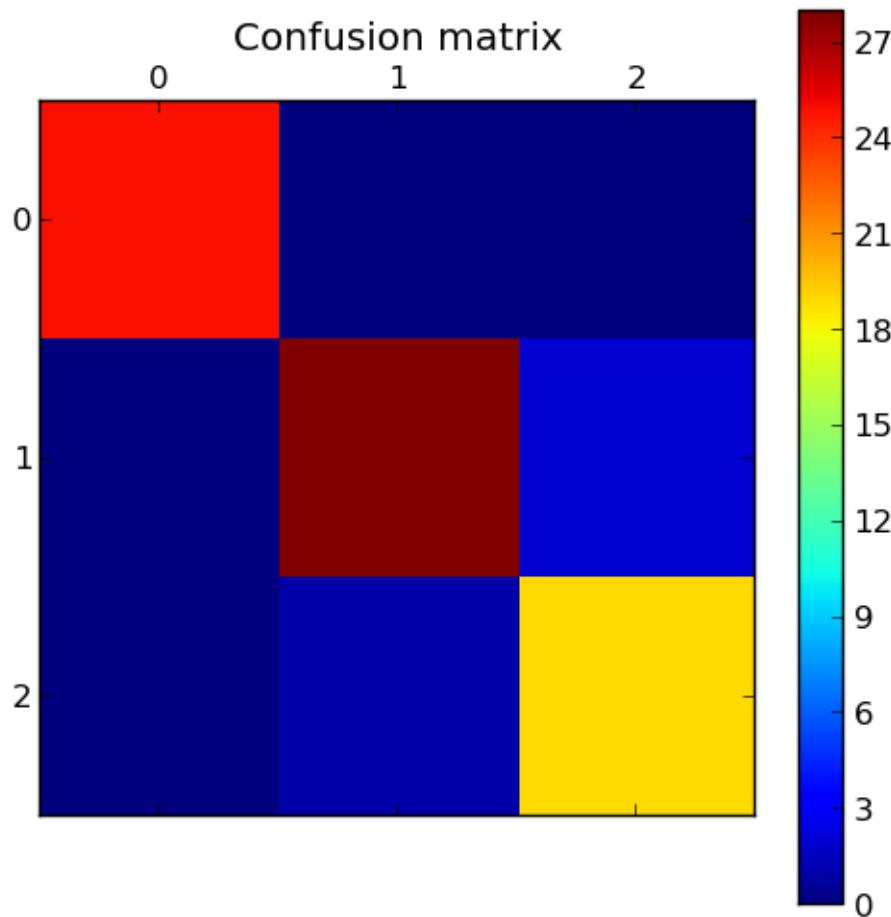


Figure 2.3: *Confusion matrix*

## Confusion matrix

Example of confusion matrix usage to evaluate the quality of the output of a classifier.

**Script output:**

```
[ [25  0  0]
 [ 0 28  2]
 [ 0  1 19]]
```

**Python source code:** [plot\\_confusion\\_matrix.py](#)

```
print __doc__

import random
import pylab as pl
from sklearn import svm, datasets
from sklearn.metrics import confusion_matrix

# import some data to play with
iris = datasets.load_iris()
X = iris.data
y = iris.target
```

```
n_samples, n_features = X.shape
p = range(n_samples)
random.seed(0)
random.shuffle(p)
X, y = X[p], y[p]
half = int(n_samples / 2)

# Run classifier
classifier = svm.SVC(kernel='linear')
y_ = classifier.fit(X[:half], y[:half]).predict(X[half:])

# Compute confusion matrix
cm = confusion_matrix(y[half:], y_)

print cm

# Show confusion matrix
pl.matshow(cm)
pl.title('Confusion matrix')
pl.colorbar()
pl.show()
```

**Total running time of the example:** 0.24 seconds

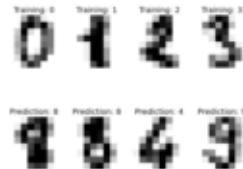
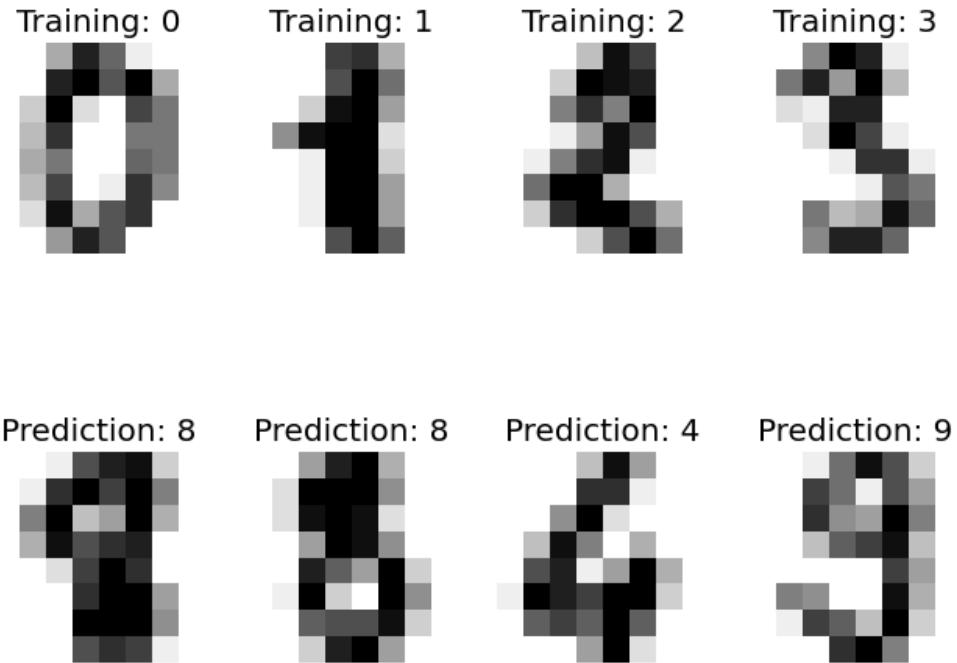


Figure 2.4: Recognizing hand-written digits

## 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*.

**Script output:**

```
Classification report for classifier SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
gamma=0.001, kernel=rbf, max_iter=-1, probability=False, 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](#)

```
print __doc__

# Author: Gael Varoquaux <gael dot varoquaux at normalesup dot org>
# License: Simplified BSD

# Standard scientific Python imports
import pylab as pl

# 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. For these images know which
# digit they represent: it is given in the 'target' of the dataset.
for index, (image, label) in enumerate(zip(digits.images, digits.target)[:4]):
    pl.subplot(2, 4, index + 1)
    pl.axis('off')
    pl.imshow(image, cmap=pl.cm.gray_r, interpolation='nearest')
    pl.title('Training: %i' % label)

# 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)
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)

for index, (image, prediction) in enumerate(
    zip(digits.images[n_samples / 2:], predicted)[:4]):
    pl.subplot(2, 4, index + 5)
    pl.axis('off')
    pl.imshow(image, cmap=pl.cm.gray_r, interpolation='nearest')
```

```

    pl.title('Prediction: %i' % prediction)
    pl.show()

```

**Total running time of the example:** 0.80 seconds

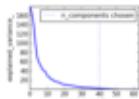
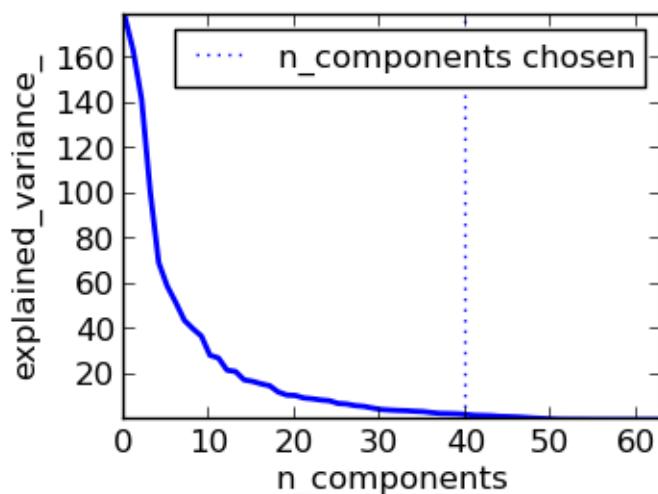


Figure 2.5: Pipelining: chaining a PCA and a logistic regression

### 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



**Python source code:** [plot\\_digits\\_pipe.py](#)

```

print __doc__

# Code source: Gael Varoquaux
# Modified for Documentation merge by Jaques Grobler
# License: BSD

import numpy as np
import pylab as pl

from sklearn import linear_model, decomposition, datasets

logistic = linear_model.LogisticRegression()

```

```
pca = decomposition.PCA()
from sklearn.pipeline import Pipeline
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)

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

#####
# Prediction

from sklearn.grid_search import GridSearchCV

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)

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

**Total running time of the example:** 8.95 seconds

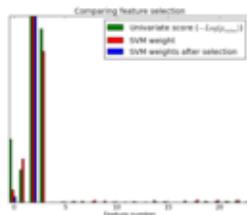


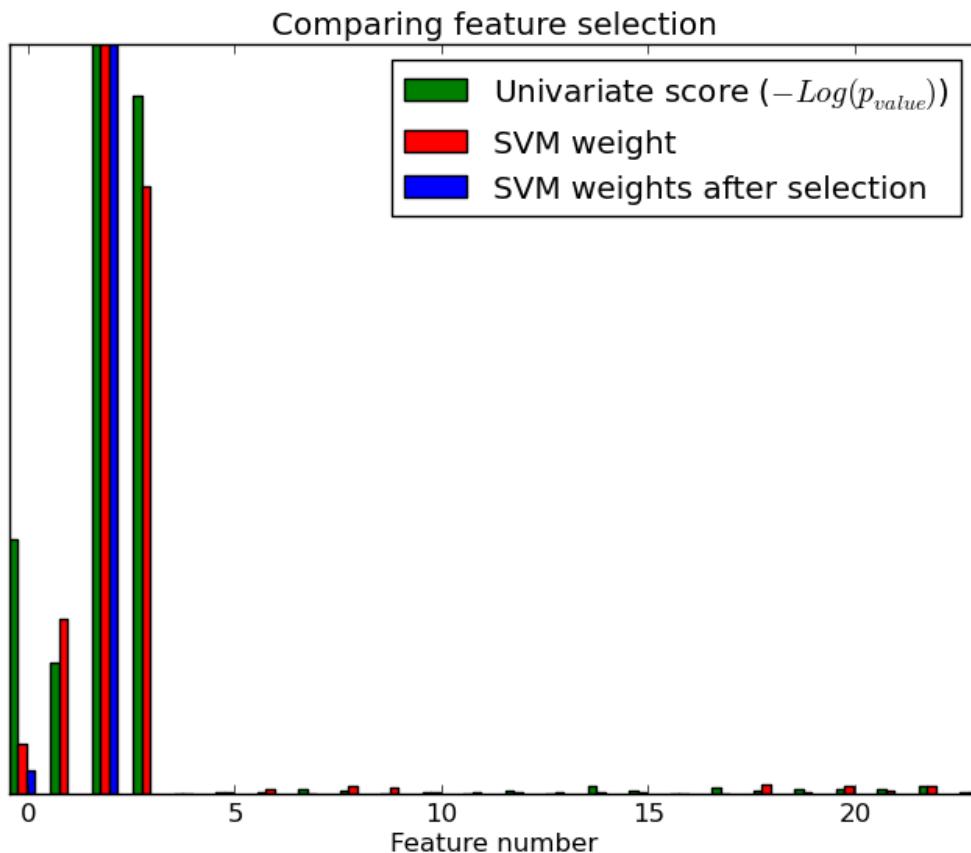
Figure 2.6: *Univariate Feature Selection*

## 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.



**Python source code:** [plot\\_feature\\_selection.py](#)

```
print __doc__

import numpy as np
import pylab as pl

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

#####
pl.figure(1)
pl.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()
pl.bar(X_indices - .45, scores, width=.2,
       label=r'Univariate score ($-\text{Log}(p_{\text{value}})$)', 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()

pl.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()

pl.bar(X_indices[selector.get_support()] - .05, svm_weights_selected, width=.2,
       label='SVM weights after selection', color='b')

pl.title("Comparing feature selection")
pl.xlabel('Feature number')
pl.yticks(())
pl.axis('tight')
pl.legend(loc='upper right')
pl.show()
```

**Total running time of the example:** 0.16 seconds

## Demonstration of sampling from HMM

This script shows how to sample points from a Hidden Markov Model (HMM): we use a 4-components with specified mean and covariance.

The plot show the sequence of observations generated with the transitions between them. We can see that, as specified

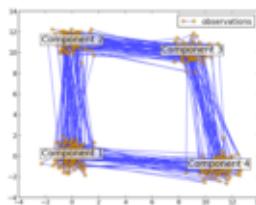
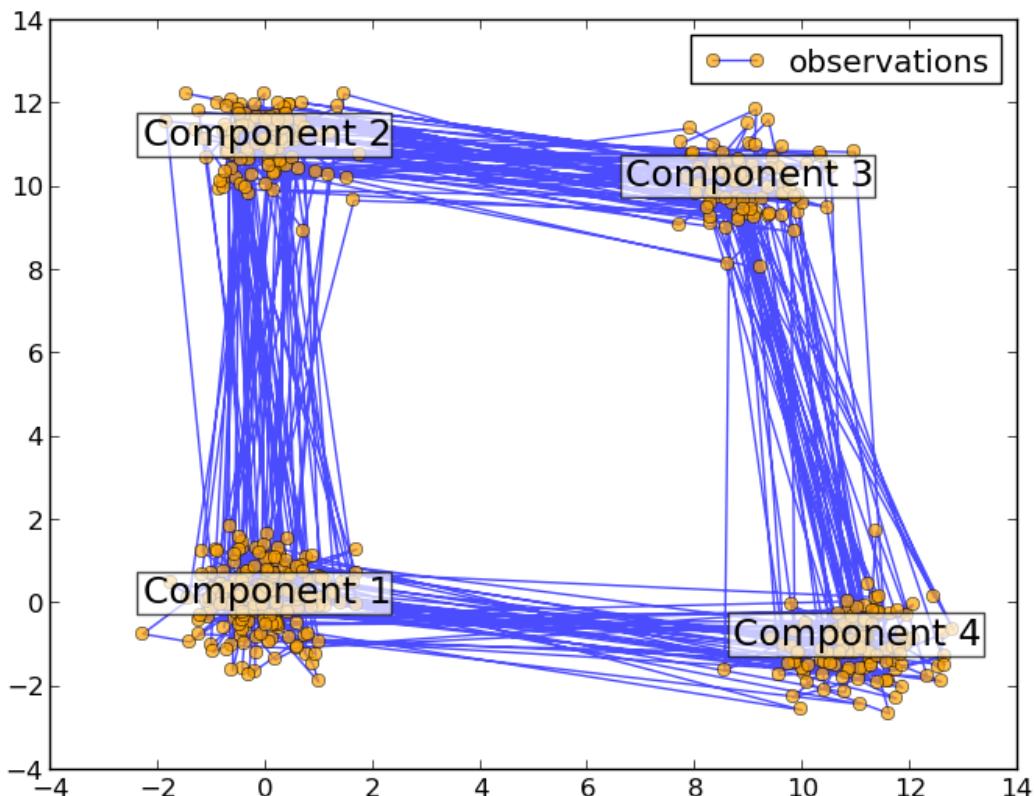


Figure 2.7: Demonstration of sampling from HMM

by our transition matrix, there are no transition between component 1 and 3.



**Python source code:** [plot\\_hmm\\_sampling.py](#)

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn import hmm

#####
# Prepare parameters for a 3-components HMM
# Initial population probability
start_prob = np.array([0.6, 0.3, 0.1, 0.0])
# The transition matrix, note that there are no transitions possible
```

```
# between component 1 and 4
trans_mat = np.array([[0.7, 0.2, 0.0, 0.1],
                     [0.3, 0.5, 0.2, 0.0],
                     [0.0, 0.3, 0.5, 0.2],
                     [0.2, 0.0, 0.2, 0.6]])
# The means of each component
means = np.array([[0.0, 0.0],
                  [0.0, 11.0],
                  [9.0, 10.0],
                  [11.0, -1.0],
                  []])
# The covariance of each component
covars = .5 * np.tile(np.identity(2), (4, 1, 1))

# Build an HMM instance and set parameters
model = hmm.GaussianHMM(4, "full", start_prob, trans_mat,
                        random_state=42)

# Instead of fitting it from the data, we directly set the estimated
# parameters, the means and covariance of the components
model.means_ = means
model.covars_ = covars
#####
# Generate samples
X, Z = model.sample(500)

# Plot the sampled data
plt.plot(X[:, 0], X[:, 1], "-o", label="observations", ms=6,
         mfc="orange", alpha=0.7)

# Indicate the component numbers
for i, m in enumerate(means):
    plt.text(m[0], m[1], 'Component %i' % (i + 1),
             size=17, horizontalalignment='center',
             bbox=dict(alpha=.7, facecolor='w'))
```

Total running time of the example: 0.49 seconds

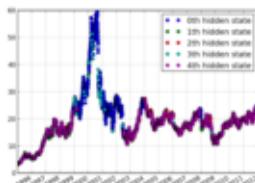
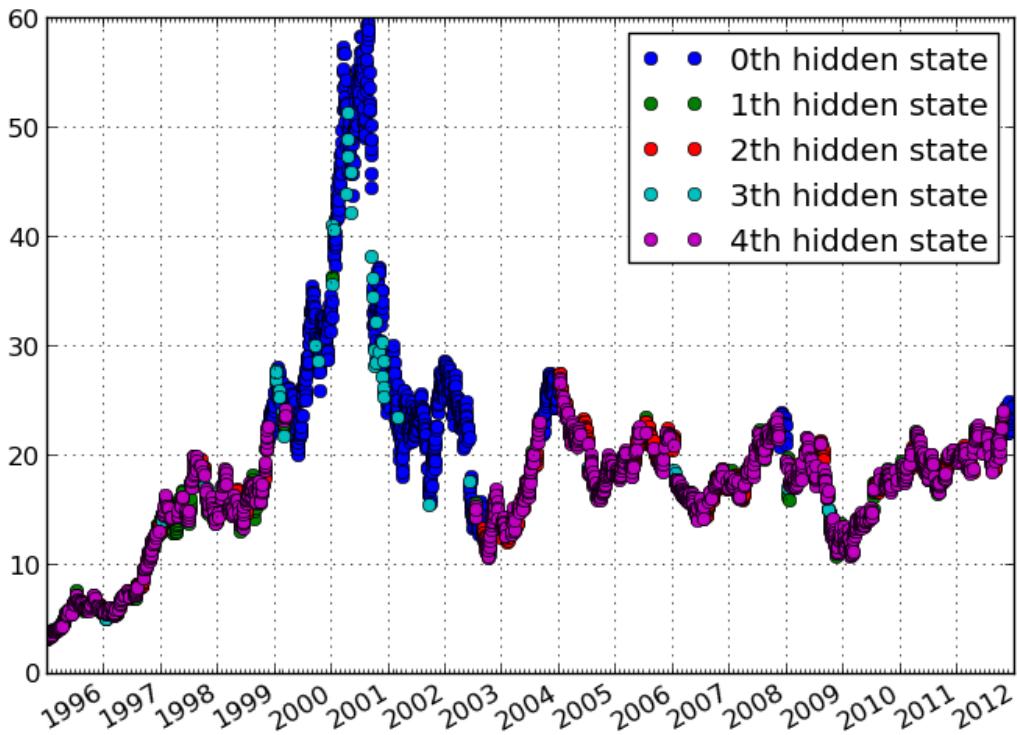


Figure 2.8: Gaussian HMM of stock data

## Gaussian HMM of stock data

This script shows how to use Gaussian HMM. It uses stock price data, which can be obtained from yahoo finance. For more information on how to get stock prices with matplotlib, please refer to date\_demo1.py of matplotlib.

**Script output:**

```

fitting to HMM and decoding ... done

Transition matrix
[[ 9.76678321e-01   1.63094197e-16   2.38275244e-03   2.09389209e-02
  5.92841578e-09]
 [ 4.64499065e-15   6.26669324e-01   3.31030650e-02   2.39509822e-02
  3.16276629e-01]
 [ 7.87921469e-04   2.91935585e-02   8.20942488e-01   9.67147966e-06
  1.49066361e-01]
 [ 2.63641047e-01   3.22002362e-01   3.60014275e-18   4.14356591e-01
  1.69427371e-16]
 [ 3.99740296e-03   1.18545066e-01   1.54490761e-01   3.57721691e-03
  7.19389553e-011]

means and vars of each hidden state
0th hidden state
mean = [ 2.80181977e-02   4.96955040e+07]
var = [ 8.85469963e-01   2.50139810e+14]

1th hidden state
mean = [ 3.73479997e-02   1.10541973e+08]
var = [ 1.97431875e-01   8.82226016e+14]

2th hidden state

```

```
mean = [ 6.33545732e-03   4.91489505e+07]
var = [ 5.06333239e-02   1.09824688e+14]
```

```
3th hidden state
mean = [ -7.71496971e-01   1.48871576e+08]
var = [ 6.17257739e+00   1.02346891e+16]
```

```
4th hidden state
mean = [ 1.16344782e-02   6.99631659e+07]
var = [ 1.24178968e-01   1.53072855e+14]
```

**Python source code:** [plot\\_hmm\\_stock\\_analysis.py](#)

```
print __doc__

import datetime
import numpy as np
import pylab as pl
from matplotlib.finance import quotes_historical_yahoo
from matplotlib.dates import YearLocator, MonthLocator, DateFormatter
from sklearn.hmm import GaussianHMM

#####
# Downloading the data
date1 = datetime.date(1995, 1, 1) # start date
date2 = datetime.date(2012, 1, 6) # end date
# get quotes from yahoo finance
quotes = quotes_historical_yahoo("INTC", date1, date2)
if len(quotes) == 0:
    raise SystemExit

# unpack quotes
dates = np.array([q[0] for q in quotes], dtype=int)
close_v = np.array([q[2] for q in quotes])
volume = np.array([q[5] for q in quotes])[1:]

# take diff of close value
# this makes len(diff) = len(close_t) - 1
# therefore, others quantity also need to be shifted
diff = close_v[1:] - close_v[:-1]
dates = dates[1:]
close_v = close_v[1:]

# pack diff and volume for training
X = np.column_stack([diff, volume])

#####
# Run Gaussian HMM
print "fitting to HMM and decoding ...",
n_components = 5

# make an HMM instance and execute fit
model = GaussianHMM(n_components, covariance_type="diag", n_iter=1000)

model.fit([X])

# predict the optimal sequence of internal hidden state
hidden_states = model.predict(X)
```

```

print "done\n"

#####
# print trained parameters and plot
print "Transition matrix"
print model.transmat_
print ""

print "means and vars of each hidden state"
for i in xrange(n_components):
    print "%dth hidden state" % i
    print "mean = ", model.means_[i]
    print "var = ", np.diag(model.covars_[i])
    print ""

years = YearLocator()      # every year
months = MonthLocator()    # every month
yearsFmt = DateFormatter('%Y')
fig = pl.figure()
ax = fig.add_subplot(111)

for i in xrange(n_components):
    # use fancy indexing to plot data in each state
    idx = (hidden_states == i)
    ax.plot_date(dates[idx], close_v[idx], 'o', label="%dth hidden state" % i)
ax.legend()

# format the ticks
ax.xaxis.set_major_locator(years)
ax.xaxis.set_major_formatter(yearsFmt)
ax.xaxis.set_minor_locator(months)
ax.autoscale_view()

# format the coords message box
ax.fmt_xdata = DateFormatter('%Y-%m-%d')
ax.fmt_ydata = lambda x: '$%1.2f' % x
ax.grid(True)

fig.autofmt_xdate()
pl.show()

```

**Total running time of the example:** 9.22 seconds

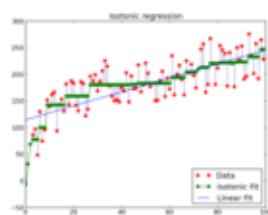
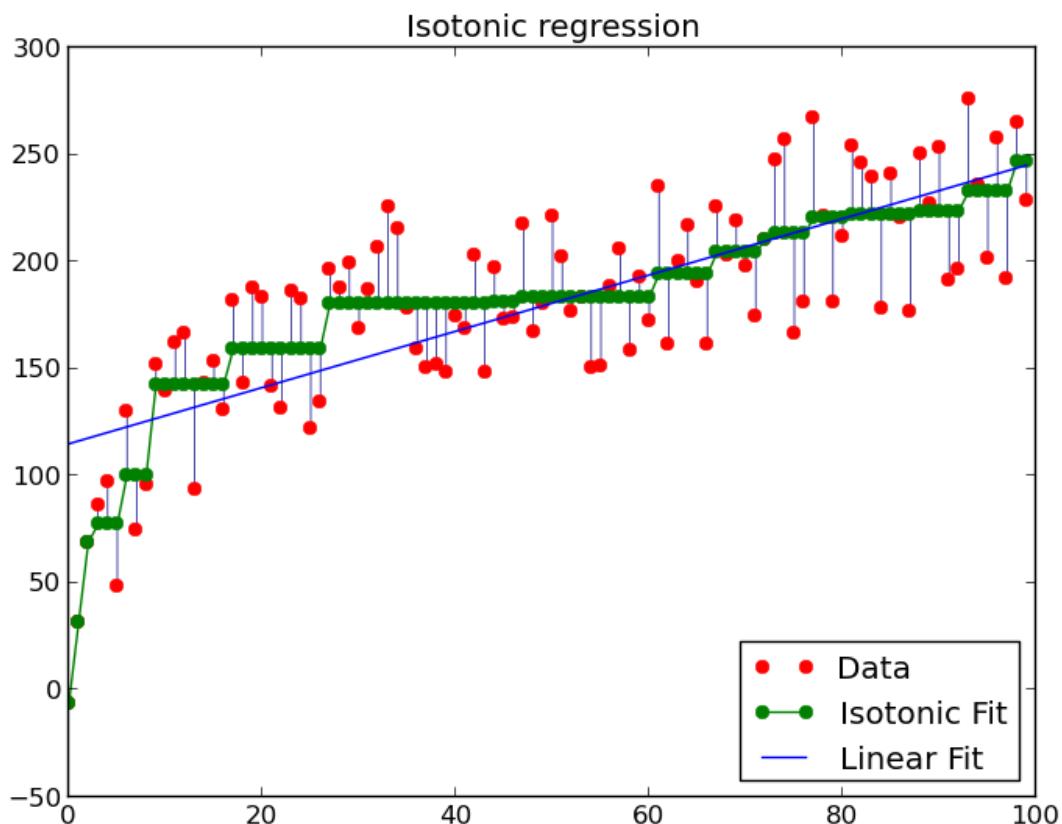


Figure 2.9: Isotonic Regression

## 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](#)

```
# Author: Nelle Varoquaux <nelle.varoquaux@gmail.com>
#         Alexandre Gramfort <alexandre.gramfort@inria.fr>
# Licence: BSD

import numpy as np
import pylab as pl
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 = pl.figure()
pl.plot(x, y, 'r.', markersize=12)
pl.plot(x, y_, 'g.-', markersize=12)
pl.plot(x, lr.predict(x[:, np.newaxis]), 'b-')
pl.gca().add_collection(lc)
pl.legend(['Data', 'Isotonic Fit', 'Linear Fit'], loc='lower right')
pl.title('Isotonic regression')
pl.show()
```

**Total running time of the example:** 0.15 seconds

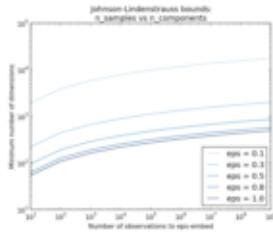


Figure 2.10: The Johnson-Lindenstrauss bound for embedding with random projections

## 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  $\epsilon$ -embedding with good probability as defined by:

$$(1 - \epsilon) \|u - v\|^2 \leq \|p(u) - p(v)\|^2 \leq (1 + \epsilon) \|u - v\|^2$$

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  $\epsilon$ -embedding is given by:

$$n_{\text{components}} \geq 4 \log(n_{\text{samples}}) / (\epsilon^2 / 2 - \epsilon^3 / 3)$$

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 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).

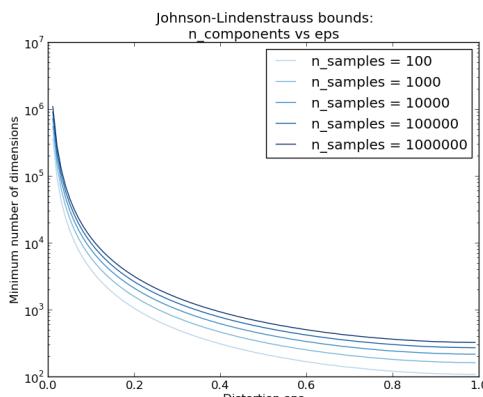
We can see that for low values of `n_components` the distribution is wide with many distorted pairs and a skewed distribution (due to the hard limit of zero ratio on the left as distances are always positives) while for larger values of `n_components` the distortion is controlled and the distances are well preserved by the random projection.

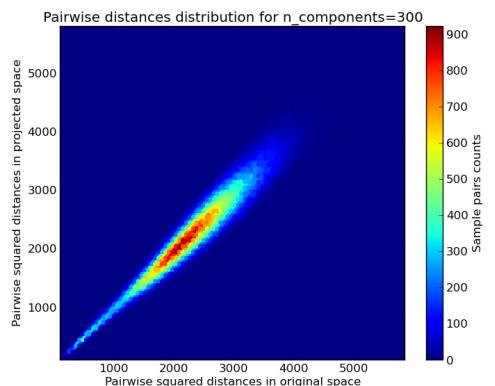
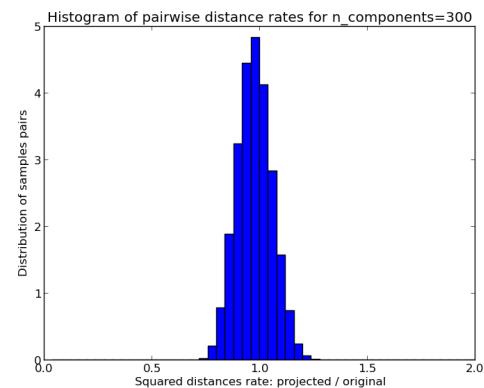
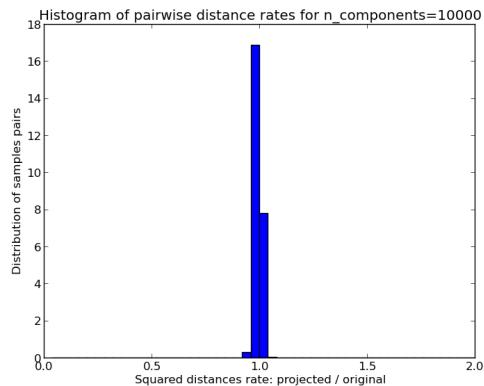
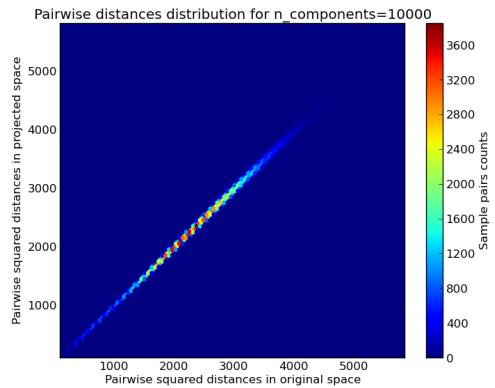
### Remarks

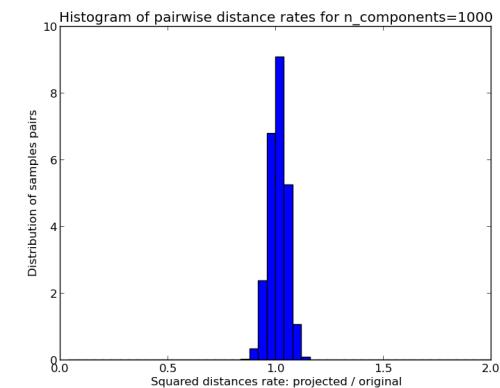
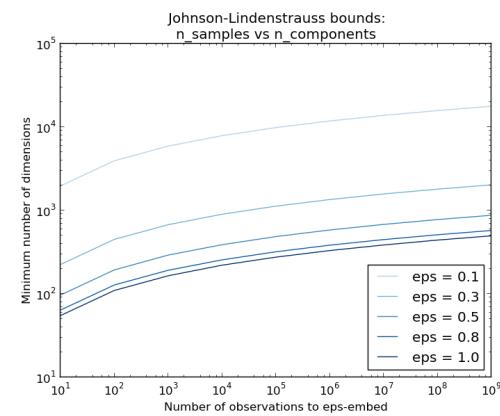
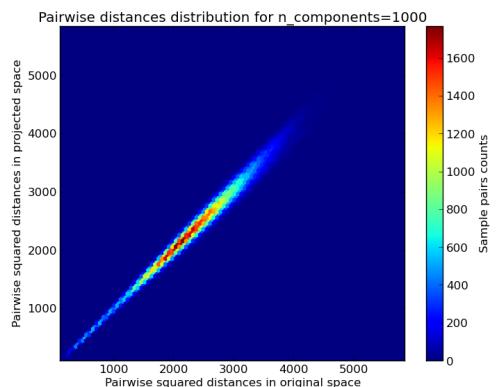
According to the JL lemma, projecting 500 samples without too much distortion will require at least several thousands dimensions, irrespectively of the number of features of the original dataset.

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.







### Script output:

```
Embedding 500 samples with dim 64 using various random projections
Projected 500 samples from 64 to 300 in 0.023s
Random matrix with size: 0.029MB
Mean distances rate: 0.98 (0.08)
Projected 500 samples from 64 to 1000 in 0.014s
Random matrix with size: 0.095MB
Mean distances rate: 1.01 (0.04)
Projected 500 samples from 64 to 10000 in 0.135s
Random matrix with size: 0.954MB
Mean distances rate: 0.99 (0.02)
```

**Python source code:** [plot\\_johnson\\_lindenstrauss\\_bound.py](#)

```

import sys
from time import time
import numpy as np
import pylab as pl
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 = pl.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)

pl.figure()
for eps, color in zip(eps_range, colors):
    min_n_components = johnson_lindenstrauss_min_dim(n_samples_range, eps=eps)
    pl.loglog(n_samples_range, min_n_components, color=color)

pl.legend(["eps = %0.1f" % eps for eps in eps_range], loc="lower right")
pl.xlabel("Number of observations to eps-embed")
pl.ylabel("Minimum number of dimensions")
pl.title("Johnson-Lindenstrauss bounds:\\nn_samples vs n_components")
pl.show()

# 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 = pl.cm.Blues(np.linspace(0.3, 1.0, len(n_samples_range)))

pl.figure()
for n_samples, color in zip(n_samples_range, colors):
    min_n_components = johnson_lindenstrauss_min_dim(n_samples, eps=eps_range)
    pl.semilogy(eps_range, min_n_components, color=color)

pl.legend(["n_samples = %d" % n for n in n_samples_range], loc="upper right")
pl.xlabel("Distortion eps")
pl.ylabel("Minimum number of dimensions")
pl.title("Johnson-Lindenstrauss bounds:\\nn_components vs eps")
pl.show()

# 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 %.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: %.3fMB" % (
            n_bytes / 1e6)

    projected_dists = euclidean_distances(
        projected_data, squared=True).ravel()[nonzero]

    pl.figure()
    pl.hexbin(dists, projected_dists, gridsize=100)
    pl.xlabel("Pairwise squared distances in original space")
    pl.ylabel("Pairwise squared distances in projected space")
    pl.title("Pairwise distances distribution for n_components=%d" %
              n_components)
    cb = pl.colorbar()
    cb.set_label('Sample pairs counts')

    rates = projected_dists / dists
    print "Mean distances rate: %.2f (%.2f)" % (
        np.mean(rates), np.std(rates))

    pl.figure()
    pl.hist(rates, bins=50, normed=True, range=(0., 2.))
    pl.xlabel("Squared distances rate: projected / original")
    pl.ylabel("Distribution of samples pairs")
    pl.title("Histogram of pairwise distance rates for n_components=%d" %
              n_components)
    pl.show()

# TODO: compute the expected value of eps and add them to the previous plot
# as vertical lines / region
```

**Total running time of the example:** 8.28 seconds

## Explicit feature map approximation for RBF kernels

An example shows how to use RBFSampler and Nystrom to appoximate 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

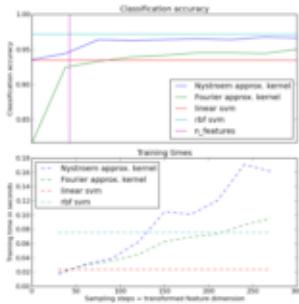


Figure 2.11: *Explicit feature map approximation for RBF kernels*

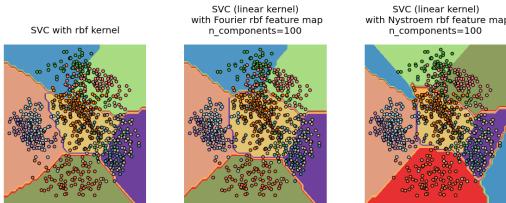
Monte Carlo samplings (in the case of `RBFSampler`, which uses random Fourier features) and different sized subsets of the training set (for `Nystroem`) for the approximate mapping are shown.

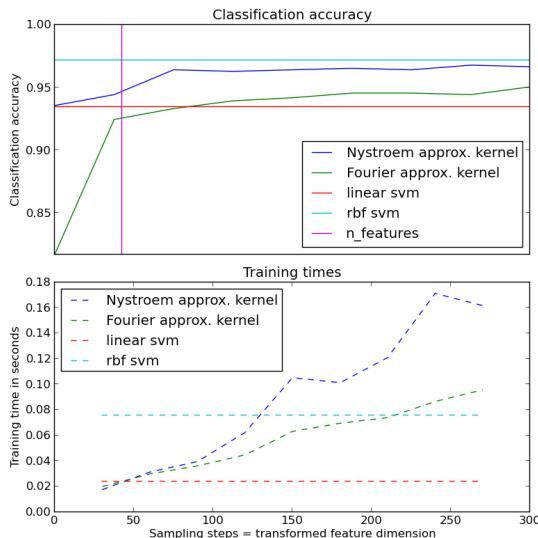
Please note that the dataset here is not large enough to show the benefits of kernel approximation, as the exact SVM is still reasonably fast.

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.

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: Simplified BSD

# Standard scientific Python imports
import pylab as pl
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([('feature_map', feature_map_fourier),
                               ('svm', svm.LinearSVC()))])

nystroem_approx_svm = pipeline.Pipeline([('feature_map', feature_map_nystroem),
                                         ('svm', svm.LinearSVC())])

# fit and predict using linear and kernel svm:

kernel_svm_time = time()
kernel_svm.fit(data_train, targets_train)
kernel_svm_score = kernel_svm.score(data_test, targets_test)
kernel_svm_time = time() - kernel_svm_time

linear_svm_time = time()
linear_svm.fit(data_train, targets_train)
linear_svm_score = linear_svm.score(data_test, targets_test)
linear_svm_time = time() - linear_svm_time

sample_sizes = 30 * np.arange(1, 10)
fourier_scores = []
nystroem_scores = []
fourier_times = []
nystroem_times = []

for D in sample_sizes:
    fourier_approx_svm.set_params(feature_map__n_components=D)
    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:
pl.figure(figsize=(8, 8))
accuracy = pl.subplot(211)
# second y axis for timeings
timescale = pl.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')

# 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)

# Generate grid along first two principal components
multiples = np.arange(-2, 2, 0.1)
# steps along first component
first = multiples[:, np.newaxis] * pca.components_[0, :]
# steps along second component
second = multiples[:, np.newaxis] * pca.components_[1, :]
# combine
grid = first[np.newaxis, :, :] + second[:, np.newaxis, :]
flat_grid = grid.reshape(-1, data.shape[1])

# title for the plots
titles = ['SVC with rbf kernel',
          'SVC (linear kernel)\n with Fourier rbf feature map\n'
          'n_components=100',
          'SVC (linear kernel)\n with Nystroem rbf feature map\n'
          'n_components=100']

pl.tight_layout()
pl.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, x_max]x[y_min, y_max].
    pl.subplot(1, 3, i + 1)
```

```
Z = clf.predict(flat_grid)

# Put the result into a color plot
Z = Z.reshape(grid.shape[:-1])
pl.contourf(multiples, multiples, Z, cmap=pl.cm.Paired)
pl.axis('off')

# Plot also the training points
pl.scatter(X[:, 0], X[:, 1], c=targets_train, cmap=pl.cm.Paired)

pl.title(titles[i])
pl.tight_layout()
pl.show()
```

**Total running time of the example:** 2.50 seconds

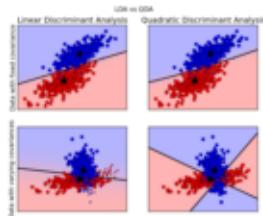
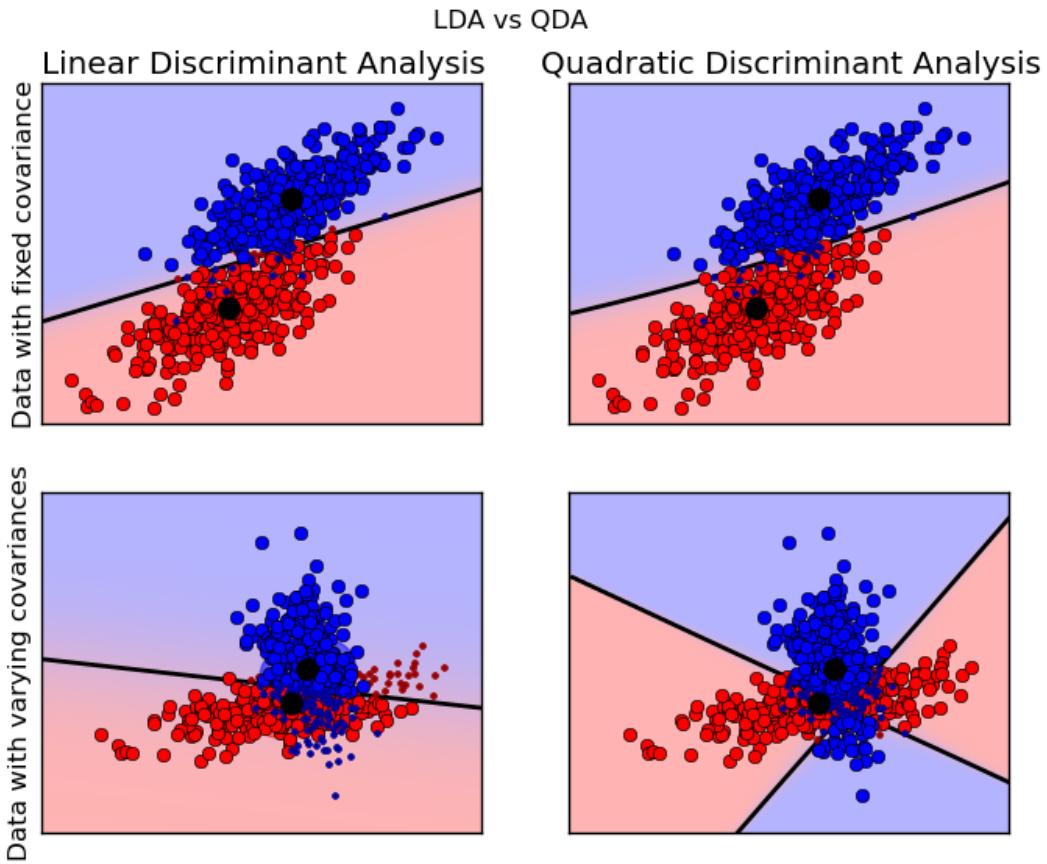


Figure 2.12: *Linear and Quadratic Discriminant Analysis with confidence ellipsoid*

### 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 pylab as pl
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)]})
pl.cm.register_cmap(cmap)

#####
# generate datasets
def dataset_fixed_cov():

    #####
    # Create two classes of points
    n_class = 2
    n_per_c = 500
    np.random.seed(1)
    X = np.concatenate((np.random.multivariate_normal([0, 0], [[1, 0], [0, 1]], n_per_c),
                       np.random.multivariate_normal([2, 2], [[1, 0], [0, 1]], n_per_c)))
    y = np.concatenate((np.repeat(0, n_per_c), np.repeat(1, n_per_c)))
    return X, y

    #####
    # Create two classes of points
    n_class = 2
    n_per_c = 500
    np.random.seed(1)
    X = np.concatenate((np.random.multivariate_normal([0, 0], [[1, 0], [0, 1]], n_per_c),
                       np.random.multivariate_normal([2, 2], [[1, 0.2], [0.2, 1]], n_per_c)))
    y = np.concatenate((np.repeat(0, n_per_c), np.repeat(1, n_per_c)))
    return X, y
```

```

'''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 = pl.subplot(2, 2, fig_index)
    if fig_index == 1:
        pl.title('Linear Discriminant Analysis')
        pl.ylabel('Data with fixed covariance')
    elif fig_index == 2:
        pl.title('Quadratic Discriminant Analysis')
    elif fig_index == 3:
        pl.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
    pl.plot(X0_tp[:, 0], X0_tp[:, 1], 'o', color='red')
    pl.plot(X0_fp[:, 0], X0_fp[:, 1], '.', color='#990000') # dark red

    # class 1: dots
    pl.plot(X1_tp[:, 0], X1_tp[:, 1], 'o', color='blue')
    pl.plot(X1_fp[:, 0], X1_fp[:, 1], '.', color='#000099') # dark blue

    # class 0 and 1 : areas
    nx, ny = 200, 100
    x_min, x_max = pl.xlim()
    y_min, y_max = pl.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)
    pl.pcolormesh(xx, yy, Z, cmap='red_blue_classes',

```

```
norm=colors.Normalize(0., 1.))
pl.contour(xx, yy, Z, linewidths=2., colors='k')

# means
pl.plot(lda.means_[0][0], lda.means_[0][1],
        'o', color='black', markersize=10)
pl.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()
    y_pred = lda.fit(X, y, store_covariance=True).predict(X)
    splot = plot_data(lda, X, y, y_pred, fig_index=2 * i + 1)
    plot_lda_cov(lda, splot)
    pl.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)
    pl.axis('tight')
pl.suptitle('LDA vs QDA')
pl.show()
```

Total running time of the example: 0.34 seconds

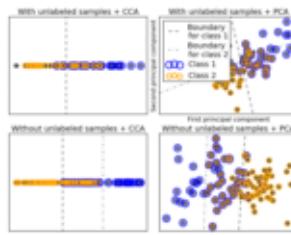


Figure 2.13: *Multilabel classification*

## Multilabel classification

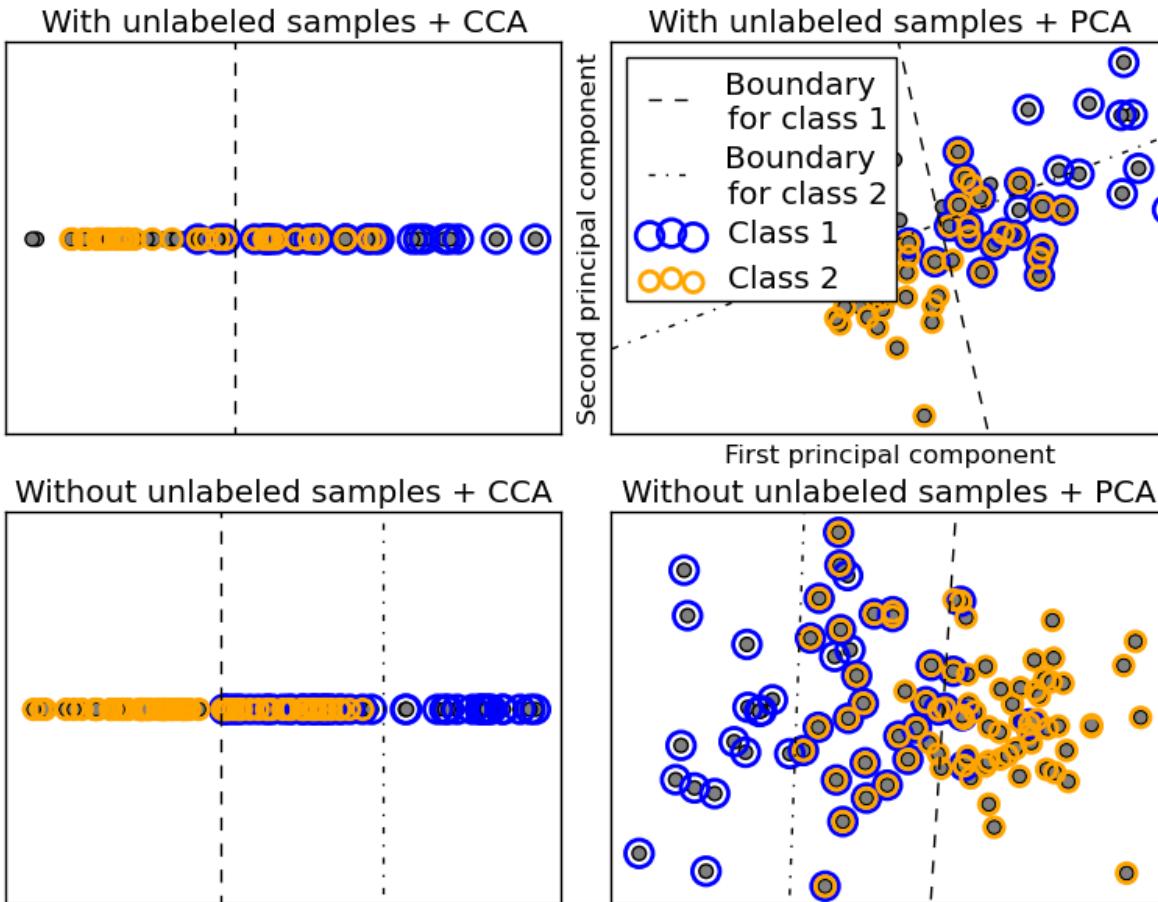
This example simulates a multi-label document classification problem. The dataset is generated randomly based on the following process:

- pick the number of labels:  $n \sim \text{Poisson}(n\_labels)$
- $n$  times, choose a class  $c$ :  $c \sim \text{Multinomial}(\theta)$
- pick the document length:  $k \sim \text{Poisson}(\text{length})$
- $k$  times, choose a word:  $w \sim \text{Multinomial}(\theta_c)$

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 pl

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.pls 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]
    pl.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":
    # Convert list of tuples to a class indicator matrix first
    Y_indicator = LabelBinarizer().fit(Y).transform(Y)
    X = CCA(n_components=2).fit(X, Y_indicator).transform(X)
else:
    raise ValueError

min_x = np.min(X[:, 0])
max_x = np.max(X[:, 0])

classif = OneVsRestClassifier(SVC(kernel='linear'))
classif.fit(X, Y)

pl.subplot(2, 2, subplot)
pl.title(title)

zero_class = np.where([0 in y for y in Y])
one_class = np.where([1 in y for y in Y])
pl.scatter(X[:, 0], X[:, 1], s=40, c='gray')
pl.scatter(X[zero_class, 0], X[zero_class, 1], s=160, edgecolors='b',
           facecolors='none', linewidths=2, label='Class 1')
pl.scatter(X[one_class, 0], X[one_class, 1], s=80, edgecolors='orange',
           facecolors='none', linewidths=2, label='Class 2')
pl.axis('tight')

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')
pl.xticks(())
pl.yticks(())

if subplot == 2:
    pl.xlim(min_x - 5, max_x)
    pl.xlabel('First principal component')
    pl.ylabel('Second principal component')
    pl.legend(loc="upper left")

pl.figure(figsize=(8, 6))

X, Y = make_multilabel_classification(n_classes=2, n_labels=1,
                                       allow_unlabeled=True,
                                       random_state=1)

plot_subfigure(X, Y, 1, "With unlabeled samples + CCA", "cca")
plot_subfigure(X, Y, 2, "With unlabeled samples + PCA", "pca")

X, Y = make_multilabel_classification(n_classes=2, n_labels=1,
                                       allow_unlabeled=False,
                                       random_state=1)

plot_subfigure(X, Y, 3, "Without unlabeled samples + CCA", "cca")
plot_subfigure(X, Y, 4, "Without unlabeled samples + PCA", "pca")

pl.subplots_adjust(.04, .02, .97, .94, .09, .2)
pl.show()

```

Total running time of the example: 0.41 seconds

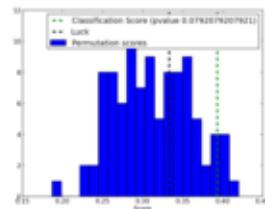
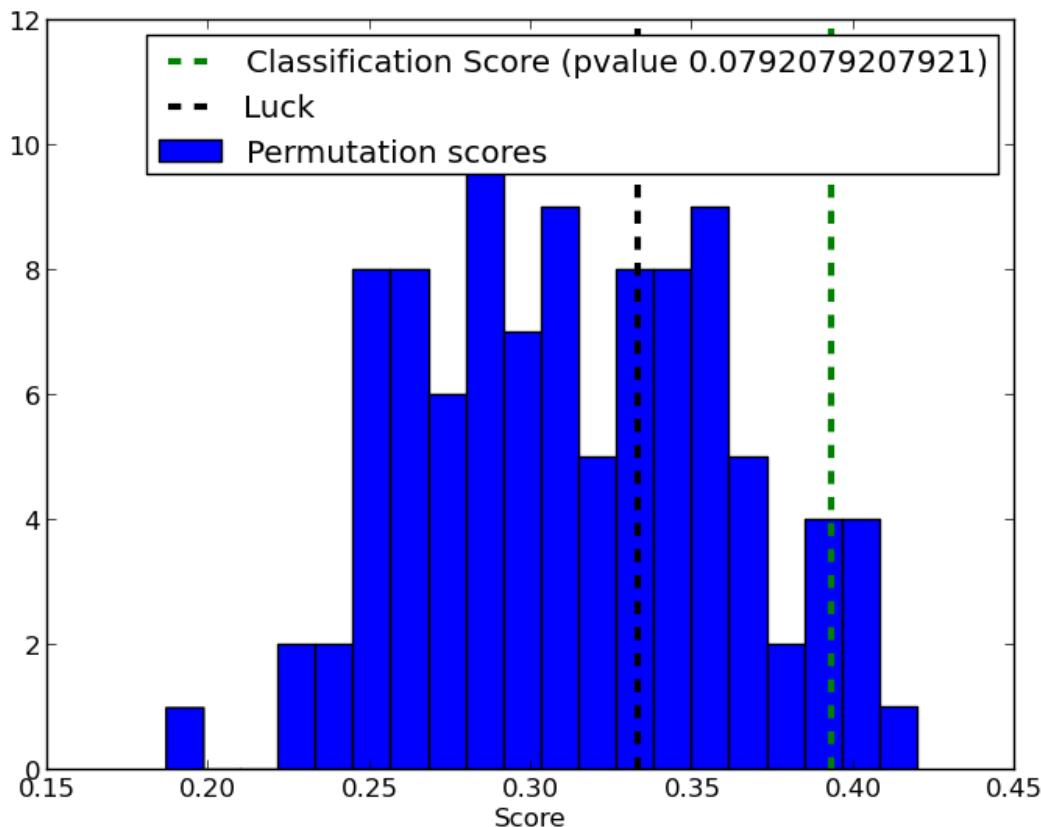


Figure 2.14: Test with permutations the significance of a classification score

### 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 0.393333333333 (pvalue : 0.0792079207921)
```

**Python source code:** [plot\\_permutation\\_test\\_for\\_classification.py](#)

```

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD

print __doc__

import numpy as np
import pylab as pl

from sklearn.svm import SVC
from sklearn.cross_validation import StratifiedKFold, permutation_test_score
from sklearn import datasets
from sklearn.metrics import accuracy_score

#####
# 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, accuracy_score, cv=cv, n_permutations=100, n_jobs=1)

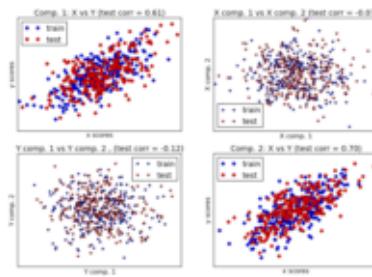
print "Classification score %s (pvalue : %s)" % (score, pvalue)

#####
# View histogram of permutation scores
pl.hist(permutation_scores, 20, label='Permutation scores')
ylim = pl.ylim()
# BUG: vlines(..., linestyle='--') fails on older versions of matplotlib
#pl.vlines(score, ylim[0], ylim[1], linestyle='--',
#           color='g', linewidth=3, label='Classification Score'
#           '(pvalue %s)' % pvalue)
#pl.vlines(1.0 / n_classes, ylim[0], ylim[1], linestyle='--',
#           color='k', linewidth=3, label='Luck')
pl.plot(2 * [score], ylim, '--g', linewidth=3,
        label='Classification Score'
        '(pvalue %s)' % pvalue)
pl.plot(2 * [1. / n_classes], ylim, '--k', linewidth=3, label='Luck')

pl.ylim(ylim)
pl.legend()
pl.xlabel('Score')
pl.show()

```

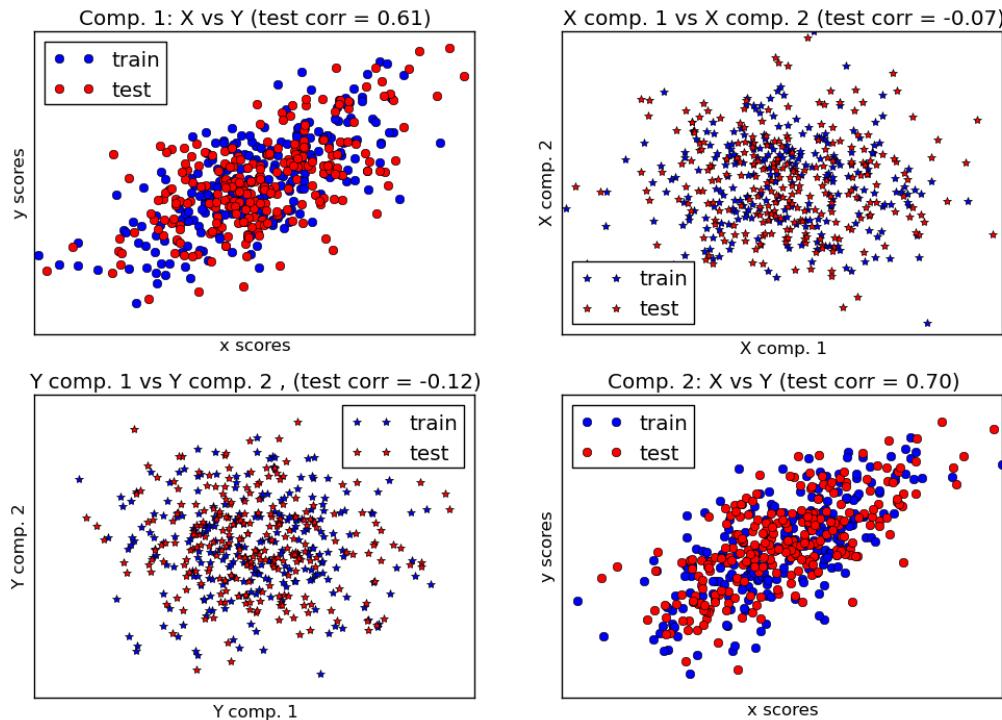
**Total running time of the example:** 9.15 seconds

Figure 2.15: *PLS Partial Least Squares*

## PLS Partial Least Squares

Simple usage of various PLS flavor: - 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.      0.5    -0.07   0.04]]
```

```
[ 0.5   1.    0.07  0.06]
[-0.07  0.07  1.    0.5 ]
[ 0.04  0.06  0.5   1.  ]]
Corr(Y)
[[ 1.    0.46 -0.04  0.01]
[ 0.46  1.    -0.04 -0.02]
[-0.04 -0.04  1.    0.54]
[ 0.01 -0.02  0.54  1.  ]]
True B (such that: Y = XB + Err)
[[1 1 1]
 [2 2 2]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]]
Estimated B
[[ 1.    1.    1.  ]
 [ 2.    2.1   2.  ]
 [ 0.   -0.    0.  ]
 [-0.    0.   -0.  ]
 [-0.1   0.   -0.  ]
 [ 0.   -0.    0.  ]
 [ 0.    0.   -0.  ]
 [ 0.    0.    0.  ]
 [-0.    0.    0.  ]
 [-0.   -0.   -0.  ]]
Estimated betas
[[ 1.]
 [ 2.]
 [ 0.]
 [-0.]
 [-0.]
 [-0.]
 [-0.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

**Python source code:** [plot\\_pls.py](#)

```
print __doc__

import numpy as np
import pylab as pl
from sklearn.pls 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([11, 11, 12, 12]).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 (symetric) 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
pl.figure(figsize=(12, 8))
pl.subplot(221)
pl.plot(X_train_r[:, 0], Y_train_r[:, 0], "ob", label="train")
pl.plot(X_test_r[:, 0], Y_test_r[:, 0], "or", label="test")
pl.xlabel("x scores")
pl.ylabel("y scores")
pl.title('Comp. 1: X vs Y (test corr = %.2f)' %
          np.corrcoef(X_test_r[:, 0], Y_test_r[:, 0])[0, 1])
pl.xticks(())
pl.yticks(())
pl.legend(loc="best")

pl.subplot(224)
pl.plot(X_train_r[:, 1], Y_train_r[:, 1], "ob", label="train")
pl.plot(X_test_r[:, 1], Y_test_r[:, 1], "or", label="test")
pl.xlabel("x scores")
pl.ylabel("y scores")
pl.title('Comp. 2: X vs Y (test corr = %.2f)' %
          np.corrcoef(X_test_r[:, 1], Y_test_r[:, 1])[0, 1])
pl.xticks(())
pl.yticks(())
pl.legend(loc="best")

# 2) Off diagonal plot components 1 vs 2 for X and Y
pl.subplot(222)
pl.plot(X_train_r[:, 0], X_train_r[:, 1], "*b", label="train")
pl.plot(X_test_r[:, 0], X_test_r[:, 1], "*r", label="test")
pl.xlabel("X comp. 1")
pl.ylabel("X comp. 2")
pl.title('X comp. 1 vs X comp. 2 (test corr = %.2f)' %
          np.corrcoef(X_test_r[:, 0], X_test_r[:, 1])[0, 1])
```

```

pl.legend(loc="best")
pl.xticks(())
pl.yticks(())

pl.subplot(223)
pl.plot(Y_train_r[:, 0], Y_train_r[:, 1], "*b", label="train")
pl.plot(Y_test_r[:, 0], Y_test_r[:, 1], "*r", label="test")
pl.xlabel("Y comp. 1")
pl.ylabel("Y comp. 2")
pl.title('Y comp. 1 vs Y comp. 2 , (test corr = %.2f)' % np.corrcoef(Y_test_r[:, 0], Y_test_r[:, 1])[0, 1])
pl.legend(loc="best")
pl.xticks(())
pl.yticks(())
pl.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

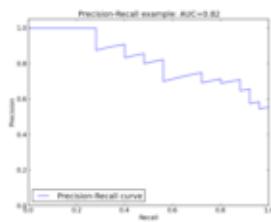
n = 1000
p = 10
X = np.random.normal(size=n * p).reshape((n, p))
y = X[:, 0] + 2 * X[:, 1] + np.random.normal(size=n * 1) + 5
pls1 = PLSRegression(n_components=3)
pls1.fit(X, y)
# note that the number of components exceeds 1 (the dimension of y)
print("Estimated betas")
print(np.round(pls1.coefs, 1))

#####
# 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)

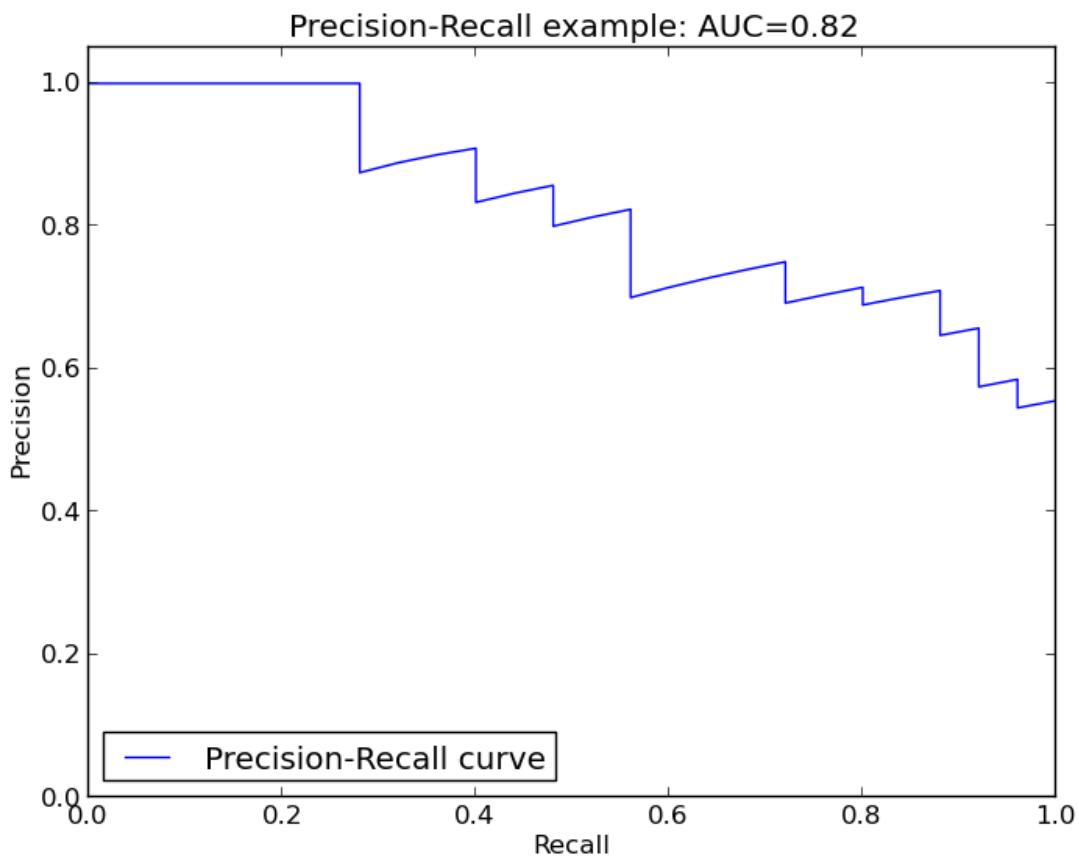
```

**Total running time of the example:** 0.32 seconds

Figure 2.16: *Precision-Recall*

### Precision-Recall

Example of Precision-Recall metric to evaluate the quality of the output of a classifier.



#### Script output:

Area Under Curve: 0.82

**Python source code:** [plot\\_precision\\_recall.py](#)

```
print __doc__  
  
import random  
import pylab as pl  
import numpy as np
```

```

from sklearn import svm, datasets
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import auc

# import some data to play with
iris = datasets.load_iris()
X = iris.data
y = iris.target
X, y = X[y != 2], y[y != 2] # Keep also 2 classes (0 and 1)
n_samples, n_features = X.shape
p = range(n_samples) # Shuffle samples
random.seed(0)
random.shuffle(p)
X, y = X[p], y[p]
half = int(n_samples / 2)

# Add noisy features
np.random.seed(0)
X = np.c_[X, np.random.randn(n_samples, 200 * n_features)]

# Run classifier
classifier = svm.SVC(kernel='linear', probability=True)
probas_ = classifier.fit(X[:half], y[:half]).predict_proba(X[half:])

# Compute Precision-Recall and plot curve
precision, recall, thresholds = precision_recall_curve(y[half:], probas_[:, 1])
area = auc(recall, precision)
print "Area Under Curve: %0.2f" % area

pl.clf()
pl.plot(recall, precision, label='Precision-Recall curve')
pl.xlabel('Recall')
pl.ylabel('Precision')
pl.ylim([0.0, 1.05])
pl.xlim([0.0, 1.0])
pl.title('Precision-Recall example: AUC=%0.2f' % area)
pl.legend(loc="lower left")
pl.show()

```

**Total running time of the example:** 0.14 seconds

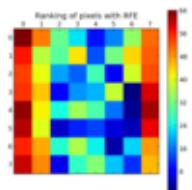
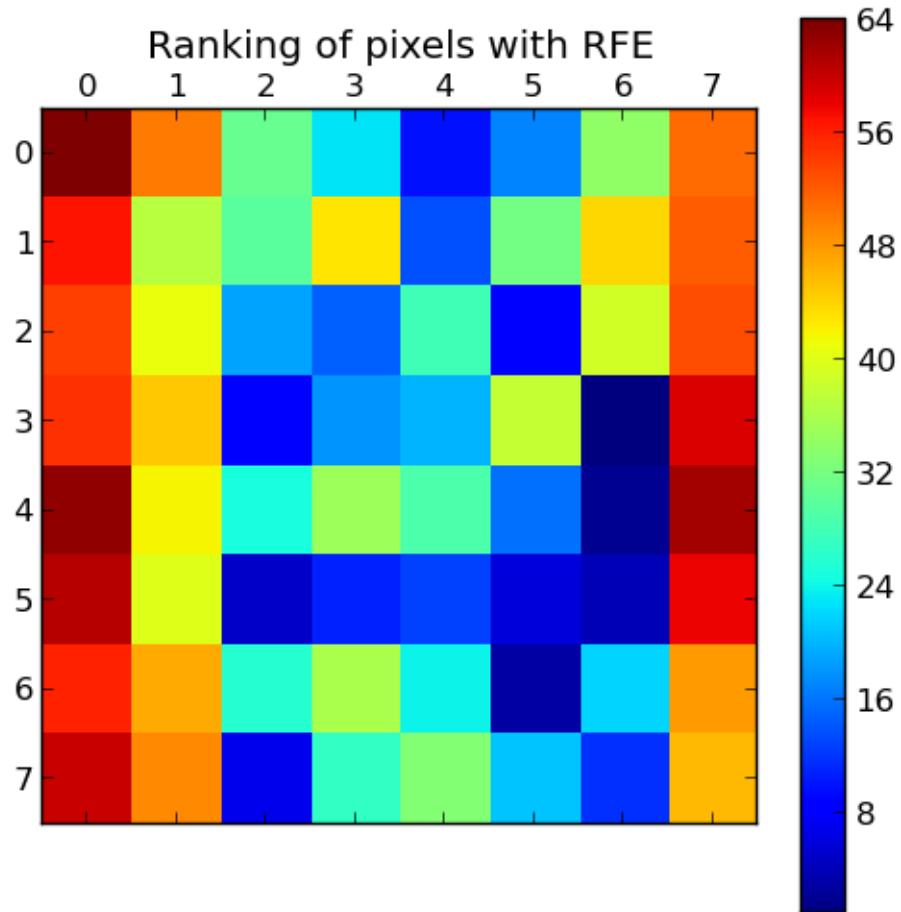


Figure 2.17: Recursive feature elimination

## Recursive feature elimination

A recursive feature elimination example showing the relevance of pixels in a digit classification task.



**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

# 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
import pylab as pl
pl.matshow(ranking)
pl.colorbar()
pl.title("Ranking of pixels with RFE")
pl.show()
```

Total running time of the example: 5.28 seconds

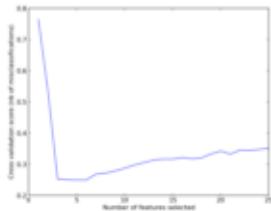
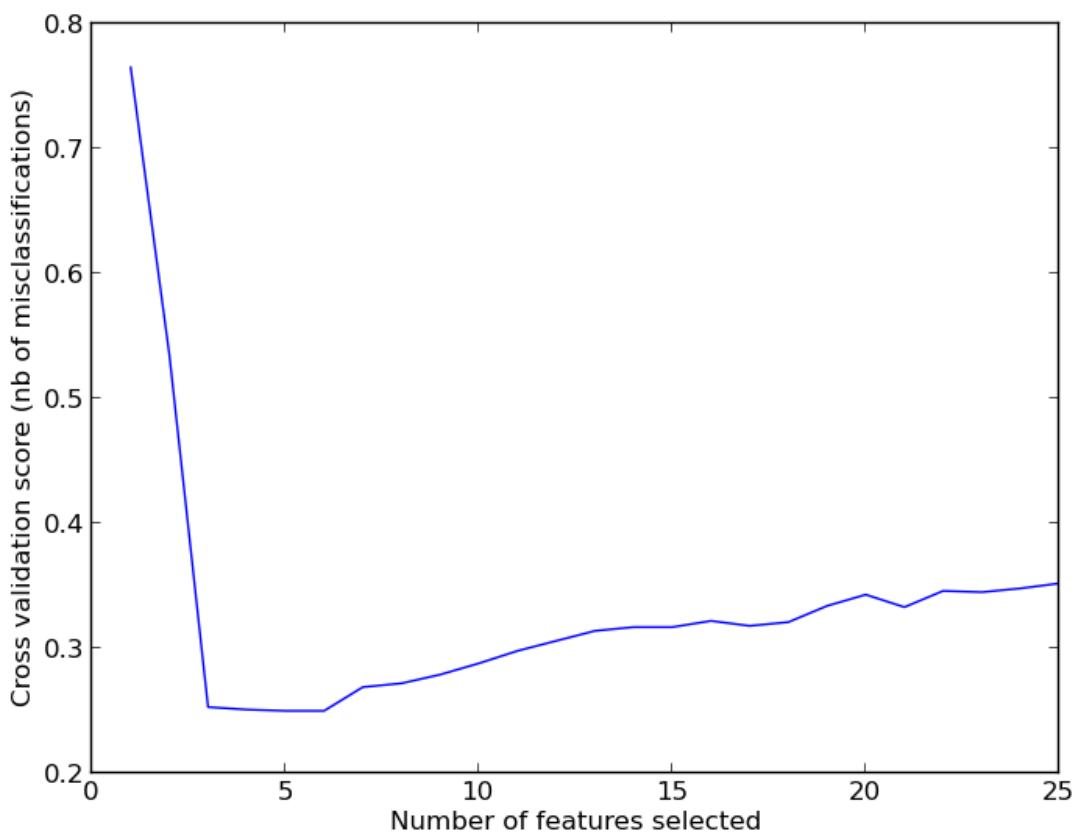


Figure 2.18: Recursive feature elimination with cross-validation

### 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 : 5
```

**Python source code:** [plot\\_rfe\\_with\\_cross\\_validation.py](#)

```
print __doc__\n\nfrom sklearn.svm import SVC\nfrom sklearn.cross_validation import StratifiedKFold\nfrom sklearn.feature_selection import RFECV\nfrom sklearn.datasets import make_classification\nfrom sklearn.metrics import zero_one_loss\n\n# Build a classification task using 3 informative features\nX, y = make_classification(n_samples=1000, n_features=25, n_informative=3,\n                           n_redundant=2, n_repeated=0, n_classes=8,\n                           n_clusters_per_class=1, random_state=0)\n\n# Create the RFE object and compute a cross-validated score.\nsvc = SVC(kernel="linear")\nrfecv = RFECV(estimator=svc, step=1, cv=StratifiedKFold(y, 2),\n              loss_func=zero_one_loss)\nrfecv.fit(X, y)\n\nprint "Optimal number of features : %d" % rfecv.n_features_\n\n# Plot number of features VS. cross-validation scores\nimport pylab as pl\npl.figure()\npl.xlabel("Number of features selected")\npl.ylabel("Cross validation score (nb of misclassifications)")\npl.plot(xrange(1, len(rfecv.cv_scores_) + 1), rfecv.cv_scores_)\npl.show()
```

**Total running time of the example:** 3.33 seconds

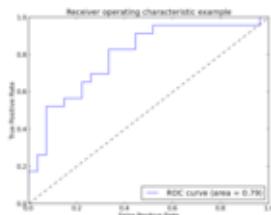
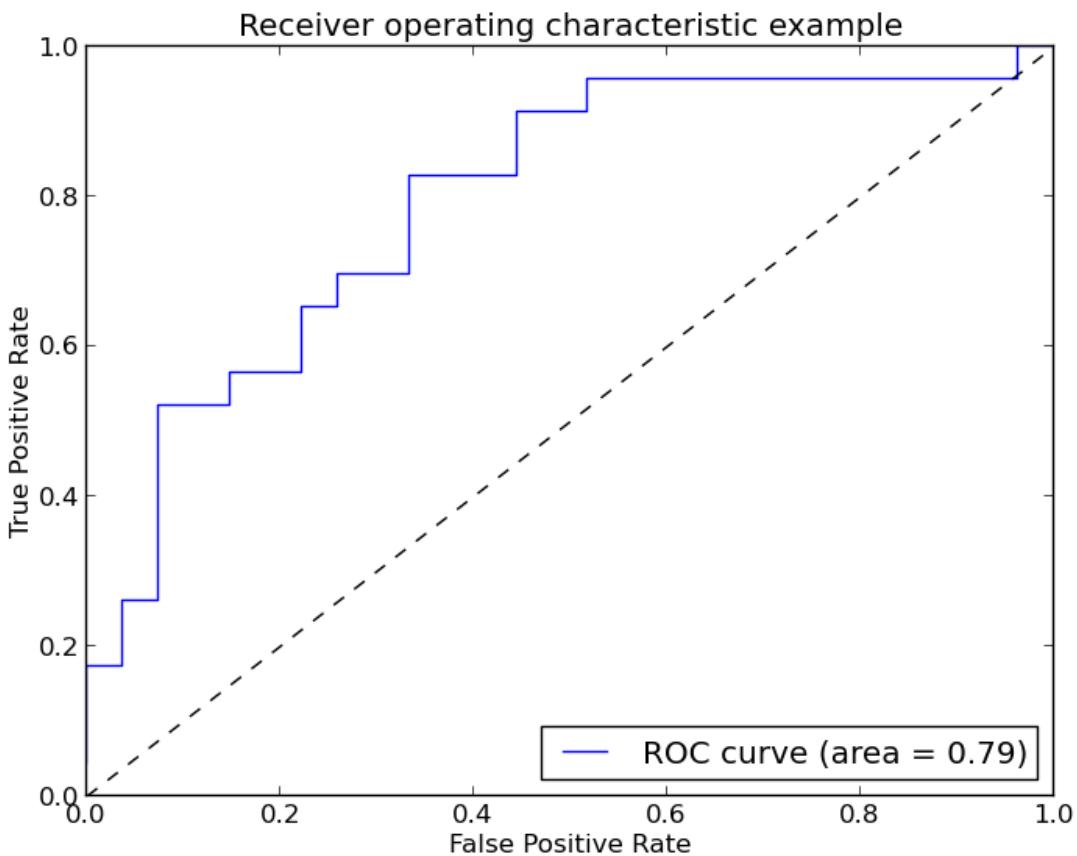


Figure 2.19: Receiver operating characteristic (ROC)

## Receiver operating characteristic (ROC)

Example of Receiver operating characteristic (ROC) metric to evaluate the quality of the output of a classifier.



### Script output:

```
Area under the ROC curve : 0.793881
```

### Python source code: plot\_roc.py

```
print __doc__

import numpy as np
import pylab as pl
from sklearn import svm, datasets
from sklearn.utils import shuffle
from sklearn.metrics import roc_curve, auc

random_state = np.random.RandomState(0)

# Import some data to play with
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Make it a binary classification problem by removing the third class
X, y = X[y != 2], y[y != 2]
n_samples, n_features = X.shape

# Add noisy features to make the problem harder
X = np.c_[X, random_state.randn(n_samples, 200 * n_features)]
```

```
# shuffle and split training and test sets
X, y = shuffle(X, y, random_state=random_state)
half = int(n_samples / 2)
X_train, X_test = X[:half], X[half:]
y_train, y_test = y[:half], y[half:]

# Run classifier
classifier = svm.SVC(kernel='linear', probability=True)
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])
roc_auc = auc(fpr, tpr)
print "Area under the ROC curve : %f" % roc_auc

# Plot ROC curve
pl.clf()
pl.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
pl.plot([0, 1], [0, 1], 'k--')
pl.xlim([0.0, 1.0])
pl.ylim([0.0, 1.0])
pl.xlabel('False Positive Rate')
pl.ylabel('True Positive Rate')
pl.title('Receiver operating characteristic example')
pl.legend(loc="lower right")
pl.show()
```

**Total running time of the example:** 0.14 seconds

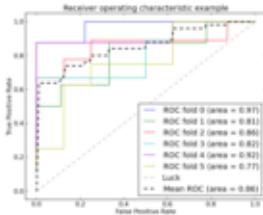
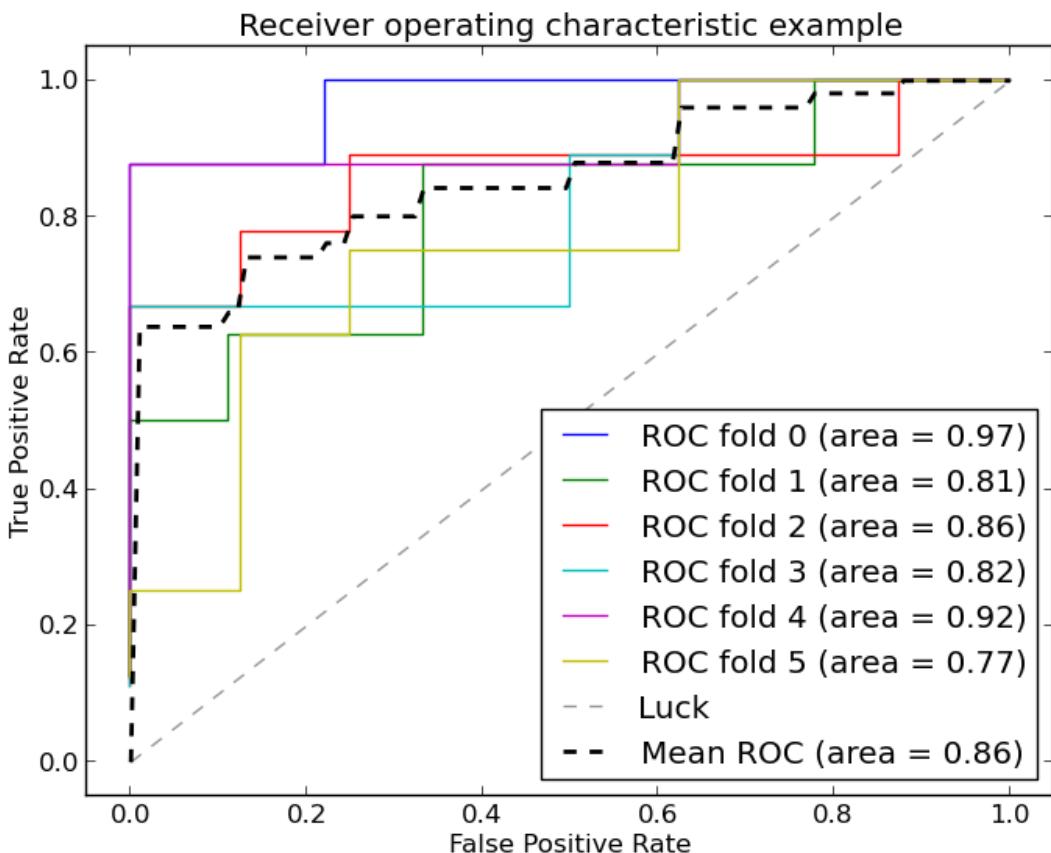


Figure 2.20: Receiver operating characteristic (ROC) with cross validation

### Receiver operating characteristic (ROC) with cross validation

Example of Receiver operating characteristic (ROC) metric to evaluate the quality of the output of a classifier using cross-validation.



**Python source code:** [plot\\_roc\\_crossval.py](#)

```
print __doc__

import numpy as np
from scipy import interp
import pylab as pl

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
X = np.c_[X, np.random.randn(n_samples, 200 * n_features)]

#####
```

```
# Classification and ROC analysis

# Run classifier with crossvalidation and plot ROC curves
cv = StratifiedKFold(y, n_folds=6)
classifier = svm.SVC(kernel='linear', probability=True)

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)
    pl.plot(fpr, tpr, lw=1, label='ROC fold %d (area = %0.2f)' % (i, roc_auc))

pl.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)
pl.plot(mean_fpr, mean_tpr, 'k--',
        label='Mean ROC (area = %0.2f)' % mean_auc, lw=2)

pl.xlim([-0.05, 1.05])
pl.ylim([-0.05, 1.05])
pl.xlabel('False Positive Rate')
pl.ylabel('True Positive Rate')
pl.title('Receiver operating characteristic example')
pl.legend(loc="lower right")
pl.show()
```

**Total running time of the example:** 0.35 seconds

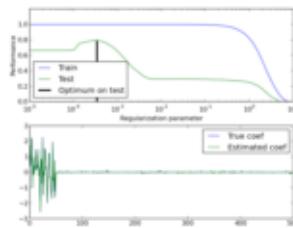
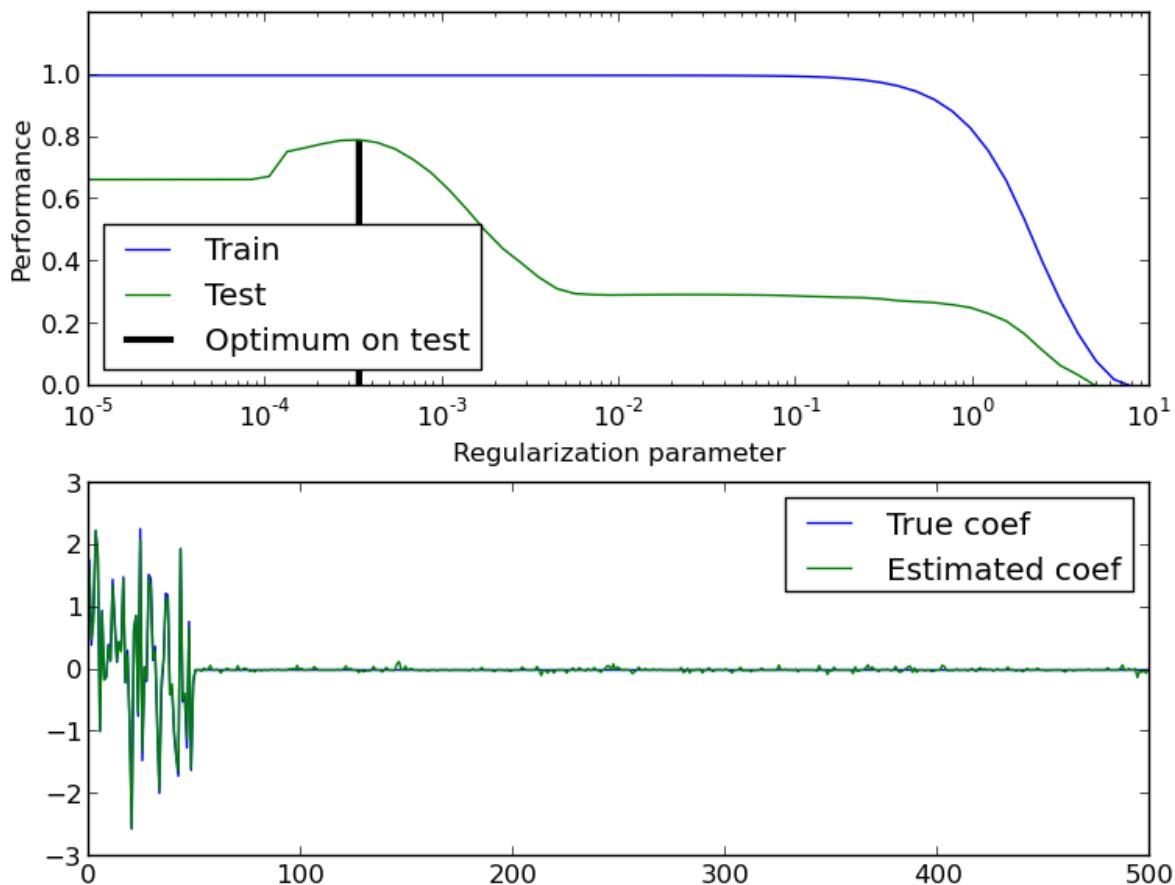


Figure 2.21: Train error vs Test error

### 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<sup>2</sup>.

**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 Style.

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 pylab as pl
pl.subplot(2, 1, 1)
pl.semilogx(alphas, train_errors, label='Train')
pl.semilogx(alphas, test_errors, label='Test')
pl.vlines(alpha_optim, pl.ylim()[0], np.max(test_errors), color='k',
           linewidth=3, label='Optimum on test')
pl.legend(loc='lower left')
pl.ylim([0, 1.2])
pl.xlabel('Regularization parameter')
pl.ylabel('Performance')

# Show estimated coef_ vs true coef
pl.subplot(2, 1, 2)
pl.plot(coef, label='True coef')
pl.plot(coef_, label='Estimated coef')
pl.legend()
pl.subplots_adjust(0.09, 0.04, 0.94, 0.94, 0.26, 0.26)
pl.show()
```

**Total running time of the example:** 4.69 seconds

Figure 2.22: Classification of text documents using sparse features

## Classification of text documents using sparse features

This is an example showing how scikit-learn can be used to classify documents by topics using a bag-of-words approach. This example uses a `scipy.sparse` matrix to store the features and demonstrates various classifiers that can efficiently handle sparse matrices.

The dataset used in this example is the 20 newsgroups dataset. It will be automatically downloaded, then cached.

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: Simplified BSD

import logging
import numpy as np
from optparse import OptionParser
import sys
from time import time
import pylab as pl

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.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.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")

(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',
    ]

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)

data_test = fetch_20newsgroups(subset='test', categories=categories,
                               shuffle=True, random_state=42)
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 - %.3fMB (training set) % (
    len(data_train.data), data_train_size_mb)
print "%d documents - %.3fMB (training 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 dataset 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 dataset 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

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)
    print "done in %fs" % (time() - t0)
    print

def trim(s):
    """Trim string to fit on terminal (assuming 80-column display)"""
    return s if len(s) <= 80 else s[:77] + "..."

# mapping from integer feature name to original token string
if opts.use_hashing:
    feature_names = None
else:
    feature_names = np.asarray(vectorizer.get_feature_names())

#####
# 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.f1_score(y_test, pred)
print "f1-score: %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")):
    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)))

class L1LinearSVC(LinearSVC):

    def fit(self, X, y):
        # The smaller C, the stronger the regularization.
        # The more regularization, the more sparsity.
        self.transformer_ = LinearSVC(penalty="l1",
                                      dual=False, tol=1e-3)
        X = self.transformer_.fit_transform(X, y)
        return LinearSVC.fit(self, X, y)

    def predict(self, X):
        X = self.transformer_.transform(X)
        return LinearSVC.predict(self, X)

print 80 * '='
print "LinearSVC with L1-based feature selection"
results.append(benchmark(L1LinearSVC()))

# make some plots

indices = np.arange(len(results))

results = [[x[i] for x in results] for i in xrange(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)

pl.title("Score")
pl.barch(indices, score, .2, label="score", color='r')
pl.barch(indices + .3, training_time, .2, label="training time", color='g')
pl.barch(indices + .6, test_time, .2, label="test time", color='b')
pl.yticks(())
pl.legend(loc='best')
pl.subplots_adjust(left=.25)

for i, c in zip(indices, clf_names):
    pl.text(-.3, i, c)

```

```
pl.show()
```

Figure 2.23: Clustering text documents using k-means

## 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 phenomenom 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 assignements 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: Simplified BSD

from sklearn.datasets import fetch_20newsgroups
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 Pipeline
```

```

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("--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.")

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 = Pipeline((
            ('hasher', hasher),
            ('tf_idf', 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,
                                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

#####
# 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=1)
else:
    km = KMeans(n_clusters=true_k, init='k-means++', max_iter=100, n_init=1,
                verbose=1)

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, labels, sample_size=1000)
```

```
print
```

Figure 2.24: Pipeline Anova SVM

## 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 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 = Pipeline([('anova', anova_filter), ('svm', clf)])  
anova_svm.fit(X, y)  
anova_svm.predict(X)
```

Figure 2.25: Concatenating multiple feature extraction methods

## 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)

# Classify:
svm = SVC(kernel="linear")
svm.fit(X_features, y)

# 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 2.26: Parameter estimation using grid search with a nested cross-validation

### Parameter estimation using grid search with a nested cross-validation

The classifier is optimized by “nested” cross-validation 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: setting estimator parameters*.

**Python source code:** `grid_search_digits.py`

```
print __doc__

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.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.svm import SVC

# 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', precision_score),
    ('recall', recall_score),
]

for score_name, score_func in scores:
```

```
print "# Tuning hyper-parameters for %s" % score_name
print

clf = GridSearchCV(SVC(C=1), tuned_parameters, score_func=score_func)
clf.fit(X_train, y_train, cv=5)

print "Best parameters set found on development set:"
print
print clf.best_estimator_
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 2.27: Sample pipeline for text feature extraction and evaluation

### 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 name 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': (1.000000000000001e-05, 9.99999999999995e-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: 9.99999999999995e-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](#)

```

print __doc__

# Author: Olivier Grisel <olivier.grisel@ensta.org>
#         Peter Prettenhofer <peter.prettenhofer@gmail.com>
#         Mathieu Blondel <mathieu@mblondel.org>
# License: Simplified BSD

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

# 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()),
])
parameters = {
    # uncommenting more parameters will give better exploring power but will
    # increase processing time in a combinatorial way
    'vect__max_df': (0.5, 0.75, 1.0),
    #'vect__max_features': (None, 5000, 10000, 50000),
    'vect__max_n': (1, 2), # words 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])
```

## Comparison of hashing-based and dictionary based text vectorization

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

Figure 2.28: Comparison of hashing-based and dictionary based text vectorization

to hash collisions.

**Python source code:** [hashing\\_vs\\_dict\\_vectorizer.py](#)

```
# Author: Lars Buitinck <L.J.Buitinck@uva.nl>
# License: 3-clause BSD

from __future__ import print_function
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))
```

## 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.

Figure 2.29: Classification of text documents: using a MLComp dataset

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`

```
print __doc__

# Author: Olivier Grisel <olivier.grisel@ensta.org>
# License: Simplified BSD

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

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(charset='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()

```

## 2.1.2 Examples based on real world datasets

Applications to real world problems with some medium sized datasets or interactive user interface.

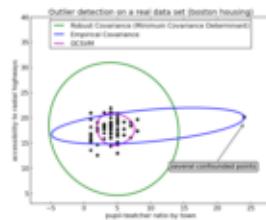


Figure 2.30: *Outlier detection on a real data set*

### 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.

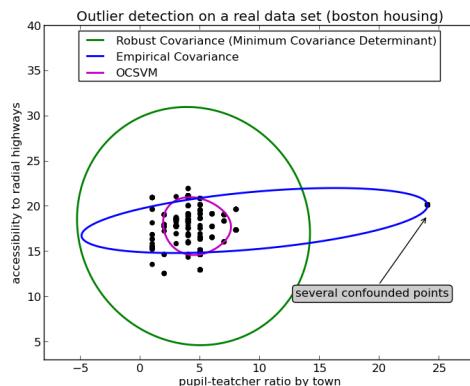
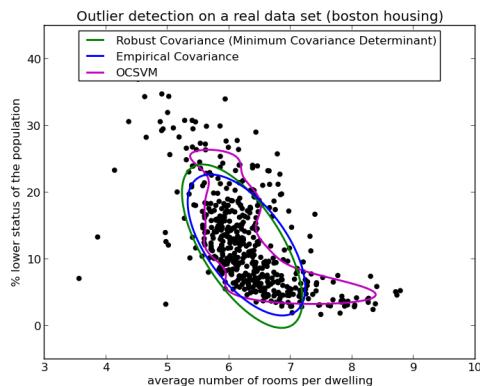
In both examples below, the main result is that the empirical covariance estimate, as a non-robust one, is highly influenced by the heterogeneous structure of the observations. Although the robust covariance estimate is able to focus on the main mode of the data distribution, it sticks to the assumption 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  
  
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])

# 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()[0].collections[0],
            legend1.values()[1].collections[0],
            legend1.values()[2].collections[0]),
            (legend1.keys()[0], legend1.keys()[1], legend1.keys()[2]),
            loc="upper center",
            prop=matplotlib.font_manager.FontProperties(size=12))
plt.ylabel("accessibility to radial highways")
plt.xlabel("pupil-teatcher ratio by town")

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()[0].collections[0],
            legend2.values()[1].collections[0],
            legend2.values()[2].collections[0]),
            (legend2.keys()[0], legend2.keys()[1], legend2.keys()[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:** 6.47 seconds

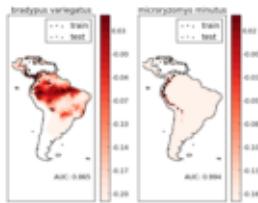


Figure 2.31: *Species distribution modeling*

## Species distribution modeling

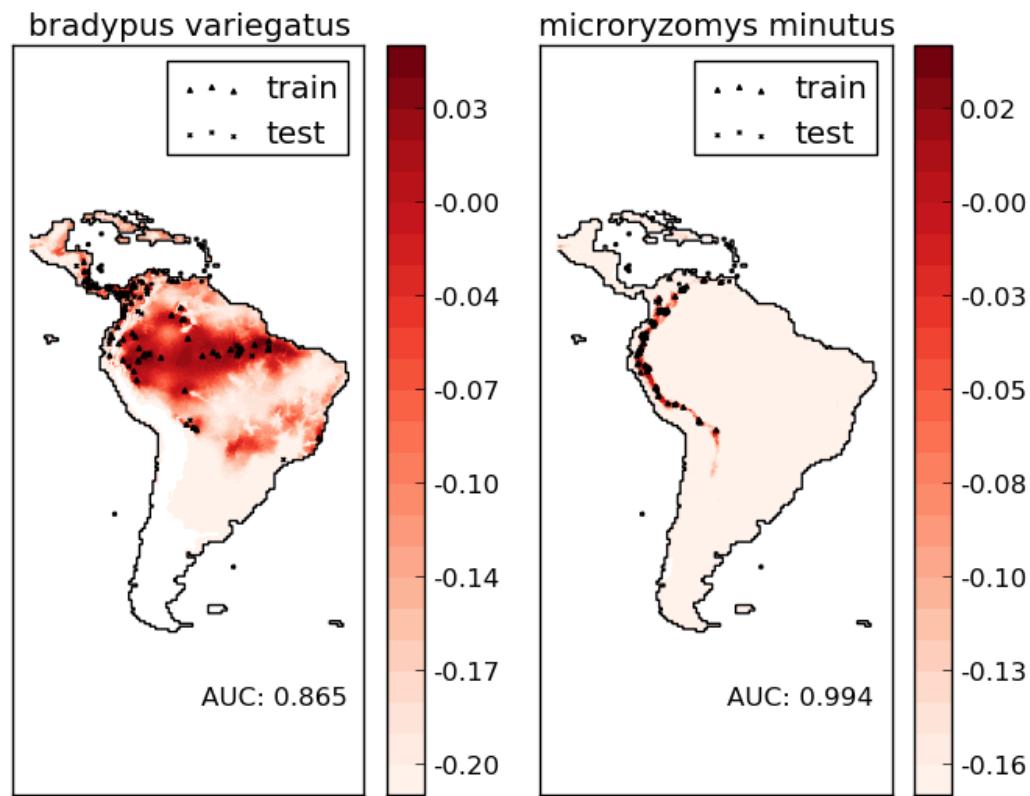
Modeling species' geographic distributions is an important problem in conservation biology. In this example we model the geographic distribution of two south american mammals given past observations and 14 environmental variables. Since we have only positive examples (there are no unsuccessful observations), 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:

- “*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 : 0.865318

---

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: 6.61s

**Python source code:** [plot\\_species\\_distribution\\_modeling.py](#)

```
# Authors: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#          Jake Vanderplas <vanderplas@astro.washington.edu>
#
# License: BSD Style.
```

```
from time import time

import numpy as np
import pylab as pl

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]))

    points = dict(test=test, train=train)

    for label, pts in points.iteritems():
        # 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 ... ",
    clf = svm.OneClassSVM(nu=0.1, kernel="rbf", gamma=0.5)
    clf.fit(train_cover_std)
    print "done. "

    # Plot map of South America
    pl.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"
        pl.contour(X, Y, land_reference,
                   levels=[-9999], colors="k",
                   linestyles="solid")

```

```
    pl.xticks([])
    pl.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
pl.contourf(X, Y, Z, levels=levels, cmap=pl.cm.Reds)
pl.colorbar(format='%.2f')

    # scatter training/testing points
pl.scatter(species.pts_train['dd long'], species.pts_train['dd lat'],
           s=2 ** 2, c='black',
           marker='^', label='train')
pl.scatter(species.pts_test['dd long'], species.pts_test['dd lat'],
           s=2 ** 2, c='black',
           marker='x', label='test')
pl.legend()
pl.title(species.name)
pl.axis('equal')

    # Compute AUC w.r.t. 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)
pl.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()
pl.show()
```

**Total running time of the example:** 6.67 seconds

## Visualizing the stock market structure

This example employs several unsupervised learning techniques to extract the stock market structure from variations in historical quotes.



Figure 2.32: Visualizing the stock market structure

The quantity that we use is the daily variation in quote price: quotes that are linked tend to cofluctuate during a day.

## Learning a graph structure

We use sparse inverse covariance estimation to find which quotes are correlated conditionally on the others. Specifically, sparse inverse covariance gives us a graph, that is a list of connection. For each symbol, the symbols that it is connected too are those useful to explain its fluctuations.

## Clustering

We use clustering to group together quotes that behave similarly. Here, amongst the *various clustering techniques* available in the scikit-learn, we use *Affinity Propagation* as it does not enforce equal-size clusters, and it can choose automatically the number of clusters from the data.

Note that this gives us a different indication than the graph, as the graph reflects conditional relations between variables, while the clustering reflects marginal properties: variables clustered together can be considered as having a similar impact at the level of the full stock market.

## Embedding in 2D space

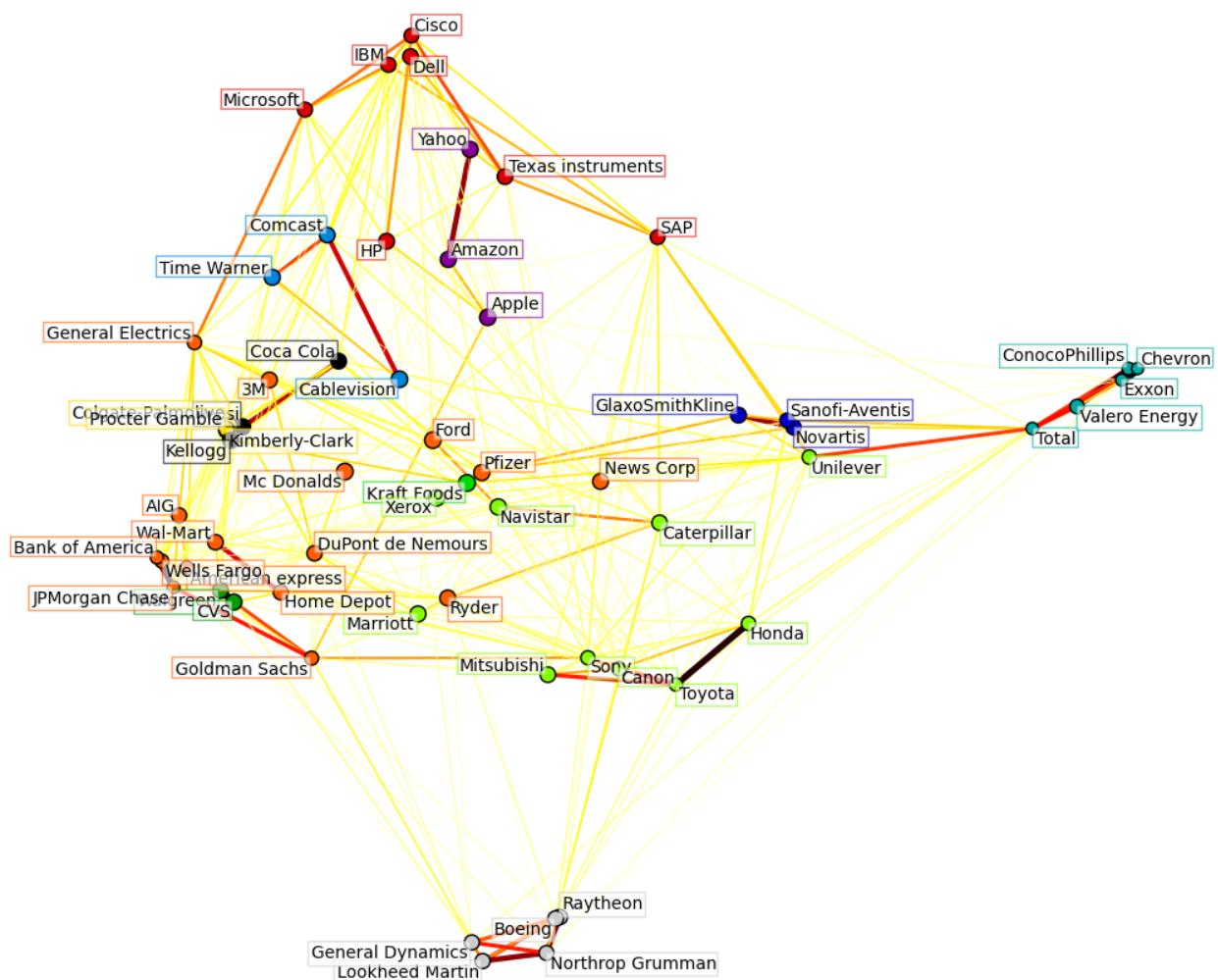
For visualization purposes, we need to lay out the different symbols on a 2D canvas. For this we use *Manifold learning* techniques to retrieve 2D embedding.

## Visualization

The output of the 3 models are combined in a 2D graph where nodes represents the stocks and edges the:

- cluster labels are used to define the color of the nodes
  - the sparse covariance model is used to display the strength of the edges
  - the 2D embedding is used to position the nodes in the plan

This example has a fair amount of visualization-related code, as visualization is crucial here to display the graph. One of the challenges is to position the labels minimizing overlap. For this we use a 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: Walgreen, CVS
Cluster 7: Kraft Foods
Cluster 8: Navistar, Sony, Marriott, Caterpillar, Canon, Toyota, Honda, Mitsubishi, Xerox, Unilever
Cluster 9: Kimberly-Clark, Colgate-Palmolive, Procter Gamble
Cluster 10: American express, Ryder, Goldman Sachs, Wal-Mart, General Electrics, Pfizer, 3M, Wells F
Cluster 11: Microsoft, SAP, IBM, Texas instruments, HP, Dell, Cisco
Cluster 12: Raytheon, Boeing, Lockheed Martin, General Dynamics, Northrop Grumman

```

**Python source code:** plot\_stock\_market.py

```

print __doc__

# Author: Gael Varoquaux gael.varoquaux@normalesup.org
# License: BSD

import datetime

```

```
import numpy as np
import pylab as pl
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, 01, 01)
d2 = datetime.datetime(2008, 01, 01)

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',
    'KFT': 'Kraft Foods',
    'K': 'Kellogg',
    'UN': 'Unilever',
    'MAR': 'Marriott',
    'PG': 'Procter Gamble',
    'CL': 'Colgate-Palmolive',
    'NWS': 'News Corp',
    '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',
'WAG': '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(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
pl.figure(1, facecolor='w', figsize=(10, 8))
pl.clf()
ax = pl.axes([0., 0., 1., 1.])
pl.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
pl.scatter(embedding[0], embedding[1], s=100 * d ** 2, c=labels,
           cmap=pl.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=pl.cm.hot_r,
                     norm=pl.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
pl.text(x, y, name, size=10,
        horizontalalignment=horizontalalignment,
        verticalalignment=verticalalignment,
        bbox=dict(facecolor='w',
                  edgecolor=pl.cm.spectral(label / float(n_labels)),
                  alpha=.6))

pl.xlim(embedding[0].min() - .15 * embedding[0].ptp(),
        embedding[0].max() + .10 * embedding[0].ptp())
pl.ylim(embedding[1].min() - .03 * embedding[1].ptp(),
        embedding[1].max() + .03 * embedding[1].ptp())

pl.show()
```

**Total running time of the example:** 5.48 seconds

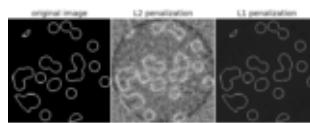


Figure 2.33: *Compressive sensing: tomography reconstruction with L1 prior (Lasso)*

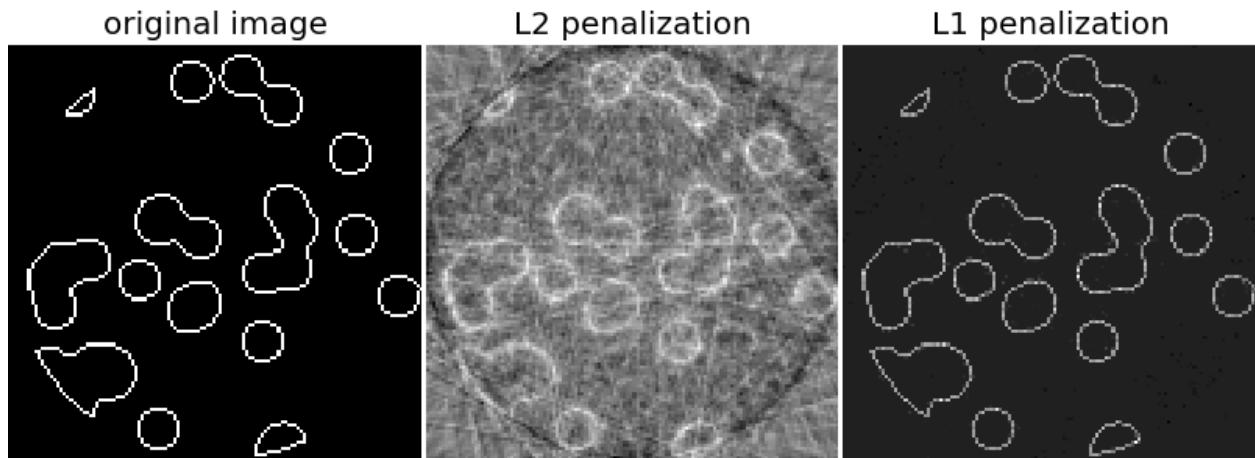
### Compressive sensing: tomography reconstruction with L1 prior (Lasso)

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  $1/7$  projections are acquired, therefore it is necessary to use prior information available on the sample (its sparsity): this is an example of **compressive sensing**.

The tomography projection operation is a linear transformation. In addition to the data-fidelity term corresponding to a linear regression, we penalize the L1 norm of the image to account for its sparsity. The resulting optimization problem is called the *Lasso*. We use the class `sklearn.linear_model.Lasso`, that uses the coordinate descent algorithm. Importantly, this implementation is more computationally efficient on a sparse matrix, than the projection operator used here.

The reconstruction with L1 penalization gives a result with zero error (all pixels are successfully labeled with 0 or 1), even if noise was added to the projections. In comparison, an L2 penalization (`sklearn.linear_model.Ridge`) produces a large number of labeling errors for the pixels. Important artifacts are observed on the reconstructed image, contrary to the L1 penalization. Note in particular the circular artifact separating the pixels in the corners, that have contributed to fewer projections than the central disk.



**Python source code:** [plot\\_tomography\\_l1\\_reconstruction.py](#)

```
print __doc__

# Author: Emmanuelle Gouillart <emmanuelle.gouillart@nsup.org>
# License: Simplified BSD

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[0:1, 0:1]
    mask_outer = (x - 1 / 2) ** 2 + (y - 1 / 2) ** 2 < (1 / 2) ** 2
    mask = np.zeros((1, 1))
    points = 1 * rs.rand(2, n_pts)
    mask[(points[0]).astype(np.int), (points[1]).astype(np.int)] = 1
    mask = ndimage.gaussian_filter(mask, sigma=1 / 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_12, cmap=plt.cm.gray, interpolation='nearest')
plt.title('L2 penalization')
plt.axis('off')
plt.subplot(133)
plt.imshow(rec_11, 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:** 11.68 seconds

Figure 2.34: *Faces recognition example using eigenfaces and SVMs*

## 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`

```

print __doc__

from time import time
import logging
import pylab as pl

```

```
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

# 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 the people names on the testing 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"""
    pl.figure(figsize=(1.8 * n_col, 2.4 * n_row))
    pl.subplots_adjust(bottom=0, left=.01, right=.99, top=.9, hspace=.35)
    for i in range(n_row * n_col):
        pl.subplot(n_row, n_col, i + 1)
        pl.imshow(images[i].reshape((h, w)), cmap=pl.cm.gray)
        pl.title(titles[i], size=12)
        pl.xticks(())
        pl.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)

pl.show()
```

Figure 2.35: Libsvm GUI

## 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 __doc__

# Author: Peter Prettenhoer <peter.prettenhofer@gmail.com>
#
# License: BSD Style.

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

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)

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.controller = controller
        self.f = f
        self.ax = ax
        self.canvas = canvas
```

```

        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 ControlBar(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 2.36: Topics extraction with Non-Negative Matrix Factorization

## 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 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 be ware than the time complexity is polynomial.

Here are some sample extracted topics that look quite good:

Topic #0: god people bible israel jesus christian true moral think christians believe don say human israeli church life children jewish

Topic #1: drive windows card drivers video scsi software pc thanks vga graphics help disk uni dos file ide controller work

Topic #2: game team nhl games ca hockey players buffalo edu cc year play university teams baseball columbia league player toronto

Topic #3: window manager application mit motif size display widget program xlib windows user color event information use events x11r5 values

Topic #4: pitt gordon banks cs science pittsburgh univ computer soon disease edu reply pain health david article medical medicine 16

**Python source code:** `topics_extraction_with_nmf.py`

```
# Author: Olivier Grisel <olivier.grisel@ensta.org>
# License: Simplified BSD

from time import time
from sklearn.feature_extraction import text
from sklearn import decomposition
from sklearn import datasets

n_samples = 1000
n_features = 1000
n_topics = 10
n_top_words = 20

# Load the 20 newsgroups dataset and vectorize it using the most common word
# frequency with TF-IDF weighting (without top 5% stop words)

t0 = time()
print "Loading dataset and extracting TF-IDF features..."
dataset = datasets.fetch_20newsgroups(shuffle=True, random_state=1)

vectorizer = text.CountVectorizer(max_df=0.95, max_features=n_features)
```

```

counts = vectorizer.fit_transform(dataset.data[:n_samples])
tfidf = text.TfidfTransformer().fit_transform(counts)
print "done in %0.3fs." % (time() - t0)

# Fit the NMF model
print "Fitting the NMF model on with n_samples=%d and n_features=%d..." % (
    n_samples, n_features)
nmf = decomposition.NMF(n_components=n_topics).fit(tfidf)
print "done in %0.3fs." % (time() - t0)

# Inverse the vectorizer vocabulary to be able
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 2.37: *Wikipedia principal eigenvector*

## 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](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](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](#)

```

print __doc__

# Author: Olivier Grisel <olivier.grisel@ensta.org>
# License: Simplified BSD

```

```
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.utils.extmath import randomized_svd
from sklearn.externals.joblib import Memory

#####
# 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 %.3fs" % (time() - t0)

# print the names of the wikipedia related strongest components 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:]])

```

### 2.1.3 Clustering

Examples concerning the `sklearn.cluster` package.

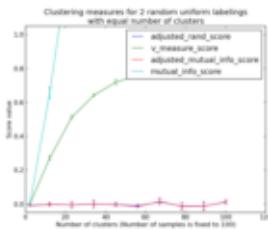


Figure 2.38: Adjustment for chance in clustering performance evaluation

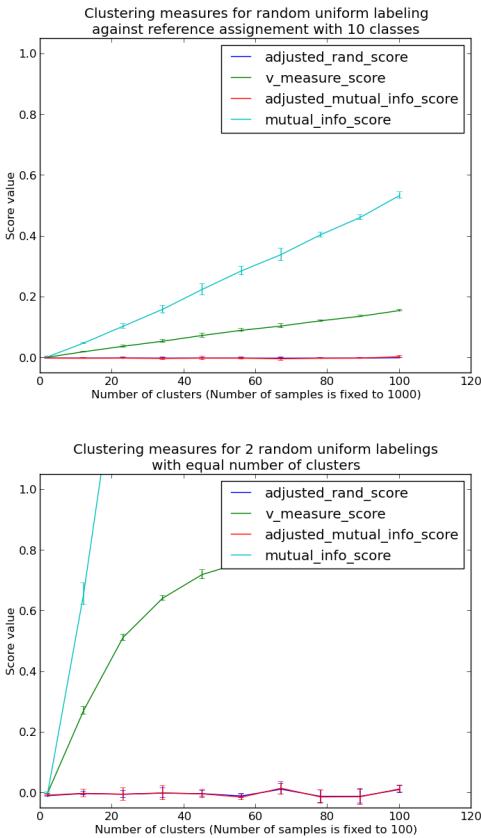
#### Adjustment for chance in clustering performance evaluation

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.



### Script output:

```
Computing adjusted_rand_score for 10 values of n_clusters and n_samples=100
done in 0.155s
Computing v_measure_score for 10 values of n_clusters and n_samples=100
done in 0.025s
Computing adjusted_mutual_info_score for 10 values of n_clusters and n_samples=100
done in 0.719s
Computing mutual_info_score for 10 values of n_clusters and n_samples=100
done in 0.017s
Computing adjusted_rand_score for 10 values of n_clusters and n_samples=1000
done in 0.154s
Computing v_measure_score for 10 values of n_clusters and n_samples=1000
done in 0.046s
Computing adjusted_mutual_info_score for 10 values of n_clusters and n_samples=1000
done in 0.311s
Computing mutual_info_score for 10 values of n_clusters and n_samples=1000
done in 0.029s
```

**Python source code:** [plot\\_adjusted\\_for\\_chance\\_measures.py](#)

```
print __doc__

# Author: Olivier Grisel <olivier.grisel@ensta.org>
# License: Simplified BSD

import numpy as np
import pylab as pl
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 assignement 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)

pl.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 %.3fs" % (time() - t0)
    plots.append(pl.errorbar(
        n_clusters_range, np.median(scores, axis=1), scores.std(axis=1))[0])
    names.append(score_func.__name__)

pl.title("Clustering measures for 2 random uniform labelings\n"
         "with equal number of clusters")
pl.xlabel('Number of clusters (Number of samples is fixed to %d)' % n_samples)

```

```
pl.ylabel('Score value')
pl.legend(plots, names)
pl.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

pl.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(pl.errorbar(
        n_clusters_range, scores.mean(axis=1), scores.std(axis=1))[0])
    names.append(score_func.__name__)

pl.title("Clustering measures for random uniform labeling\n"
         "against reference assignement with %d classes" % n_classes)
pl.xlabel('Number of clusters (Number of samples is fixed to %d)' % n_samples)
pl.ylabel('Score value')
pl.ylim(ymin=-0.05, ymax=1.05)
pl.legend(plots, names)
pl.show()
```

**Total running time of the example:** 1.64 seconds

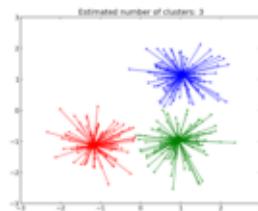
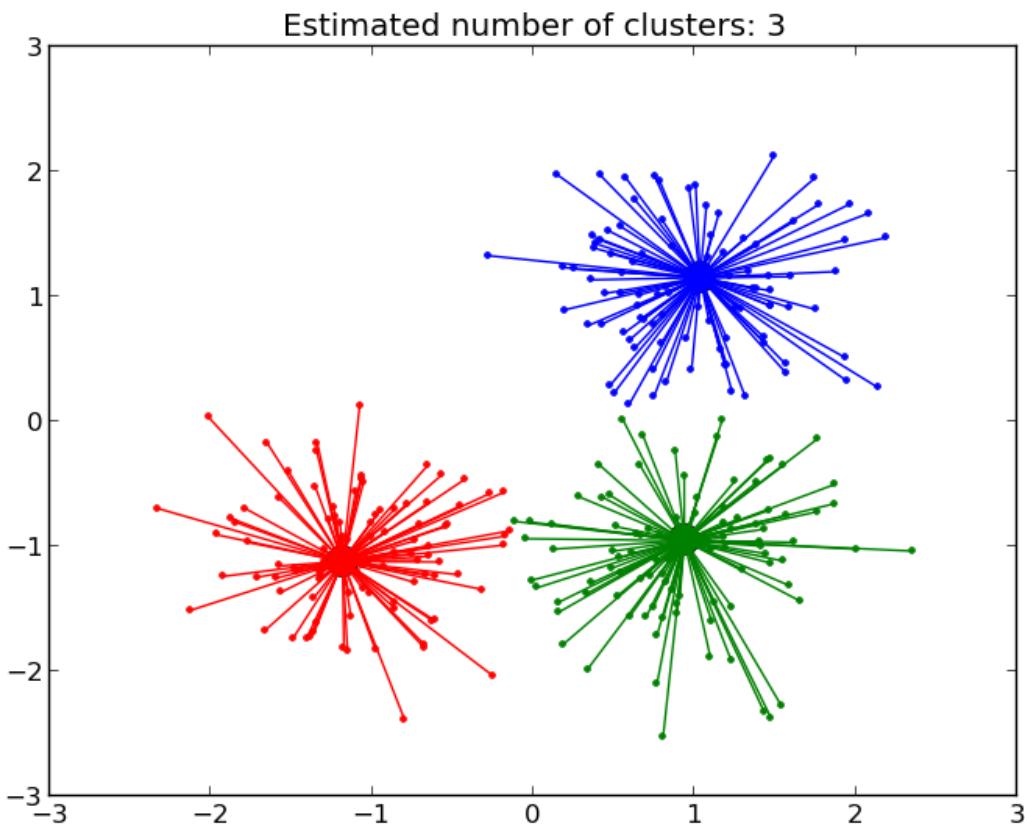


Figure 2.39: Demo of affinity propagation clustering algorithm

## Demo of affinity propagation clustering algorithm

Reference: Brendan J. Frey and Delbert Dueck, “Clustering by Passing Messages Between Data Points”, Science Feb. 2007

**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](#)

```
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 pylab as pl
from itertools import cycle

pl.close('all')
pl.figure(1)
pl.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    class_members = labels == k
    cluster_center = X[cluster_centers_indices[k]]
    pl.plot(X[class_members, 0], X[class_members, 1], col + '.')
    pl.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
            markeredgecolor='k', markersize=14)
    for x in X[class_members]:
        pl.plot([cluster_center[0], x[0]], [cluster_center[1], x[1]], col)

pl.title('Estimated number of clusters: %d' % n_clusters_)
pl.show()
```

**Total running time of the example:** 0.66 seconds

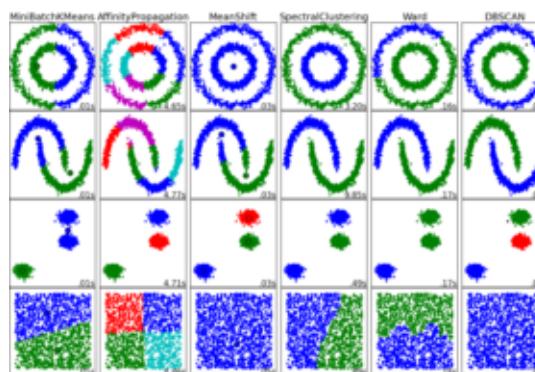


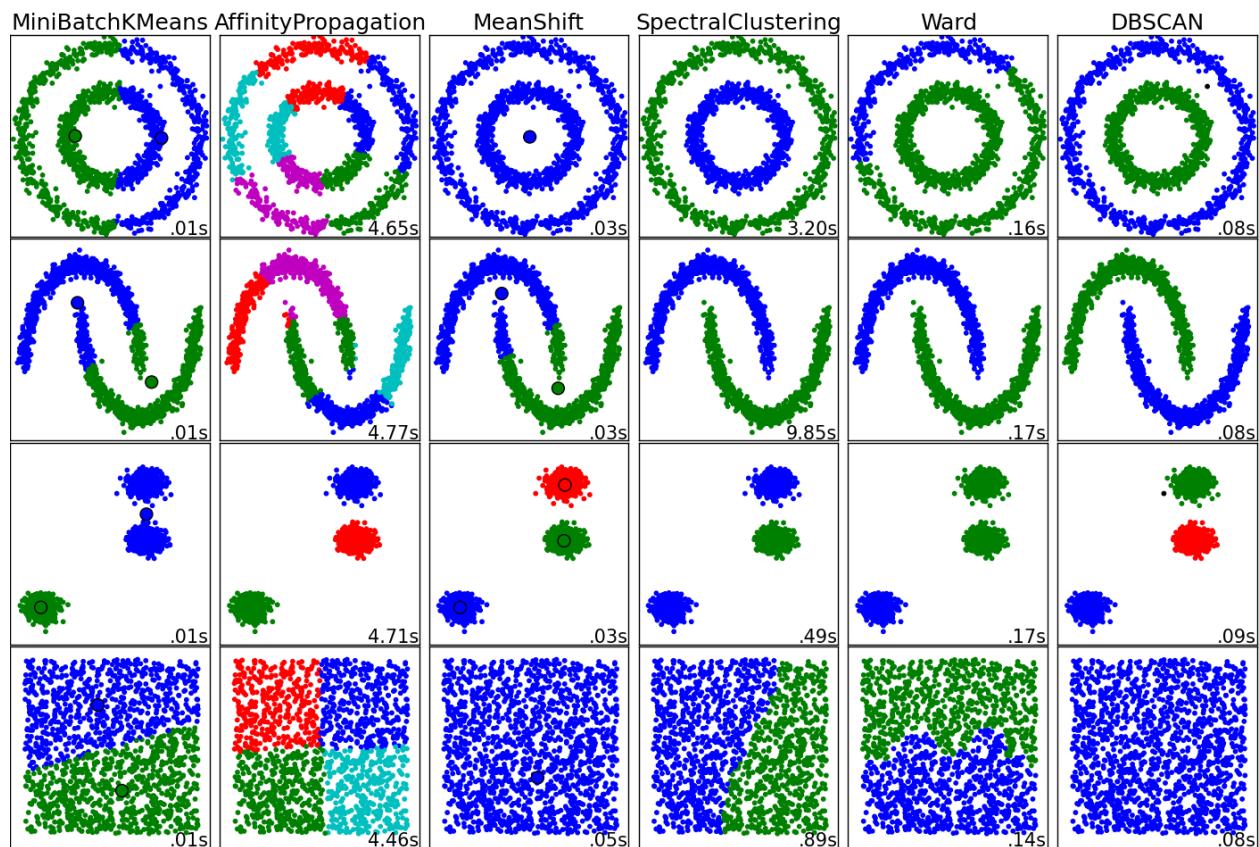
Figure 2.40: Comparing different clustering algorithms on toy datasets

## 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 mitigate this behavior.



**Python source code:** [plot\\_cluster\\_comparison.py](#)

```
print __doc__

import time

import numpy as np
import pylab as pl

from sklearn import cluster, datasets
from sklearn.metrics import euclidean_distances
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)

pl.figure(figsize=(14, 9.5))
pl.subplots_adjust(left=.01, right=.999, bottom=.001, top=.96, wspace=.05,
                   hspace=.01)

plot_num = 1
for i_dataset, dataset in enumerate([noisy_circles, noisy_moons, blobs,
                                       no_structure]):
    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)
    # make connectivity symmetric
    connectivity = 0.5 * (connectivity + connectivity.T)

    # Compute distances
    #distances = np.exp(-euclidean_distances(X))
    distances = euclidean_distances(X)

    # create clustering estimators
    ms = cluster.MeanShift(bandwidth=bandwidth, bin_seeding=True)
    two_means = cluster.MiniBatchKMeans(n_clusters=2)
    ward_five = cluster.Ward(n_clusters=2, 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)

    for algorithm in [two_means, affinity_propagation, ms, spectral,
                      ward_five, dbscan]:
        # 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
```

```

pl.subplot(4, 6, plot_num)
if i_dataset == 0:
    pl.title(str(algorithm).split(' ')[0], size=18)
pl.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)]
    pl.scatter(centers[:, 0], centers[:, 1], s=100, c=center_colors)
pl.xlim(-2, 2)
pl.ylim(-2, 2)
pl.xticks(())
pl.yticks(())
pl.text(.99, .01, ('%.2fs' % (t1 - t0)).lstrip('0'),
        transform=pl.gca().transAxes, size=15,
        horizontalalignment='right')
plot_num += 1

pl.show()

```

**Total running time of the example:** 36.96 seconds

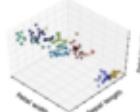
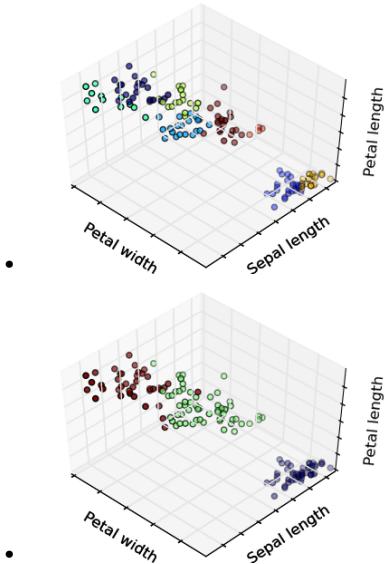
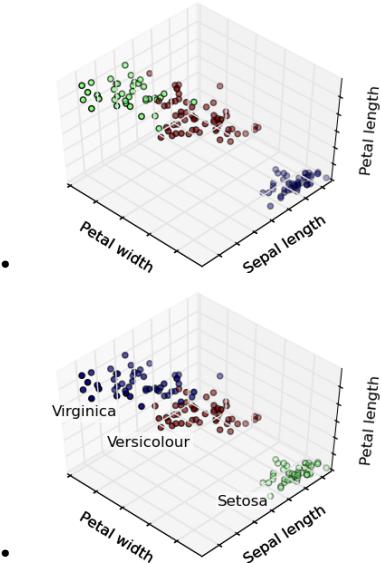


Figure 2.41: *K-means Clustering*

## 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.





**Python source code:** [plot\\_cluster\\_iris.py](#)

```
print __doc__  
  
# Code source: Gael Varoquaux  
# Modified for Documentation merge by Jaques Grobler  
# License: BSD  
  
import numpy as np  
import pylab as pl  
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 = pl.figure(fignum, figsize=(4, 3))  
    pl.clf()  
    ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)  
  
    pl.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 = pl.figure(fignum, figsize=(4, 3))
pl.clf()
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

pl.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')
pl.show()

```

**Total running time of the example:** 0.44 seconds



Figure 2.42: *Color Quantization using K-Means*

## 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.



#### Script output:

Fitting estimator on a small sub-sample of the data  
done in 0.819s.

Predicting color indices on the full image (k-means)  
done in 0.393s.

Predicting color indices on the full image (random)  
done in 0.305s.

**Python source code:** [plot\\_color\\_quantization.py](#)

```
# Authors: Robert Layton <robertlayton@gmail.com>
#          Olivier Grisel <olivier.grisel@ensta.org>
#          Mathieu Blondel <mathieu@mblondel.org>
#
# License: BSD

print __doc__
import numpy as np
import pylab as pl
from sklearn.cluster import KMeans
from sklearn.metrics import euclidean_distances
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 pl.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 estimator 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()
dist = euclidean_distances(codebook_random, image_array, squared=True)
labels_random = dist.argmin(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[random[labels[label_idx]]]
            label_idx += 1
    return image
```

```
for j in range(h):
    image[i][j] = codebook[labels[label_idx]]
    label_idx += 1
return image

# Display all results, alongside original image
pl.figure(1)
pl.clf()
ax = pl.axes([0, 0, 1, 1])
pl.axis('off')
pl.title('Original image (96,615 colors)')
pl.imshow(china)

pl.figure(2)
pl.clf()
ax = pl.axes([0, 0, 1, 1])
pl.axis('off')
pl.title('Quantized image (64 colors, K-Means)')
pl.imshow(recreate_image(kmeans.cluster_centers_, labels, w, h))

pl.figure(3)
pl.clf()
ax = pl.axes([0, 0, 1, 1])
pl.axis('off')
pl.title('Quantized image (64 colors, Random)')
pl.imshow(recreate_image(codebook_random, labels_random, w, h))
pl.show()
```

**Total running time of the example:** 2.87 seconds

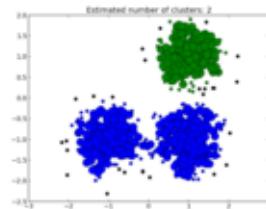
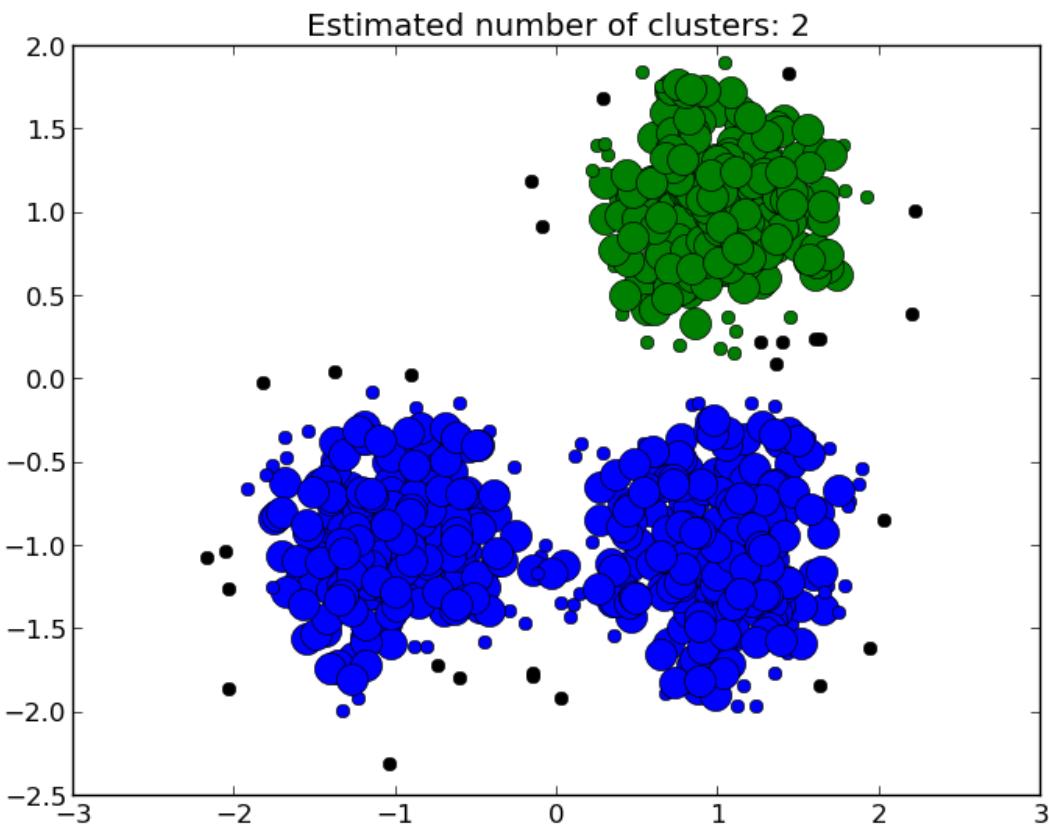


Figure 2.43: Demo of DBSCAN clustering algorithm

### Demo of DBSCAN clustering algorithm

Finds core samples of high density and expands clusters from them.

**Script output:**

```
Estimated number of clusters: 2
Homogeneity: 0.560
Completeness: 0.802
V-measure: 0.659
Adjusted Rand Index: 0.541
Adjusted Mutual Information: 0.559
Silhouette Coefficient: 0.417
```

**Python source code:** [plot\\_dbSCAN.py](#)

```
print __doc__

import numpy as np
from scipy.spatial import distance
from sklearn.cluster import DBSCAN
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=750, centers=centers, cluster_std=0.4)

#####
```

```
# Compute similarities
D = distance.squareform(distance.pdist(X))
S = 1 - (D / np.max(D))

#####
# Compute DBSCAN
db = DBSCAN(eps=0.95, min_samples=10).fit(S)
core_samples_ = db.core_sample_indices_
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(D, labels, metric='precomputed'))

#####
# Plot result
import pylab as pl
from itertools import cycle

pl.close('all')
pl.figure(1)
pl.clf()

# Black removed and is used for noise instead.
colors = cycle('bgrcmybgrcmybgrcmybgrcmy')
for k, col in zip(set(labels), colors):
    if k == -1:
        # Black used for noise.
        col = 'k'
        markersize = 6
    class_members = [index[0] for index in np.argwhere(labels == k)]
    cluster_core_samples = [index for index in core_samples_
                            if labels[index] == k]
    for index in class_members:
        x = X[index]
        if index in core_samples and k != -1:
            markersize = 14
        else:
            markersize = 6
        pl.plot(x[0], x[1], 'o', markerfacecolor=col,
                markeredgecolor='k', markersize=markersize)

pl.title('Estimated number of clusters: %d' % n_clusters_)
pl.show()
```

Total running time of the example: 1.07 seconds



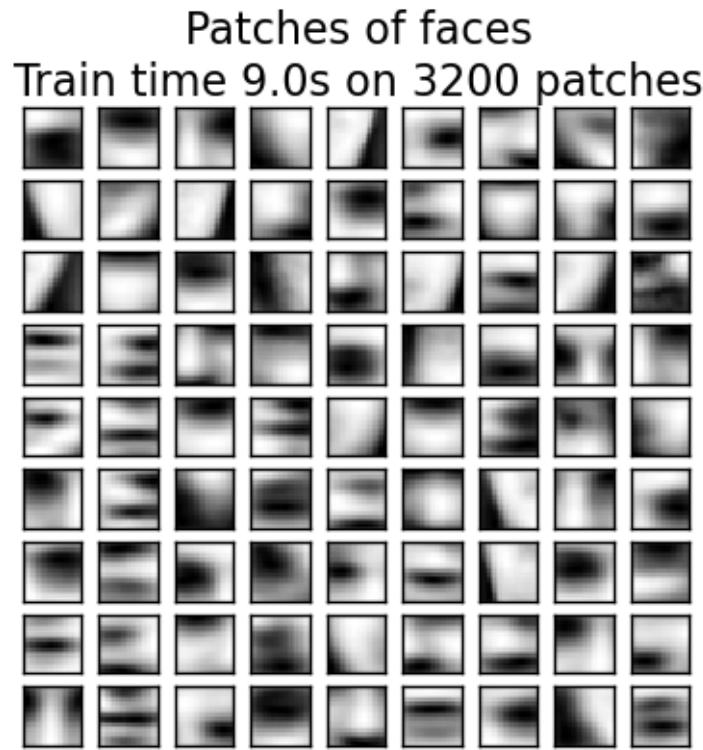
Figure 2.44: Online learning of a dictionary of parts of faces

### 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 15 patches from this image. Once we have accumulated 750 of these patches (using 50 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...
[_mini_batch_step] Reassigning 6 cluster centers.
Partial fit of 100 out of 2400
Partial fit of 200 out of 2400
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
[_mini_batch_step] Reassigning 2 cluster centers.
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 9.03s.
```

**Python source code:** [plot\\_dict\\_face\\_patches.py](#)

```
print __doc__

import time

import pylab as pl
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 4 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
pl.figure(figsize=(4.2, 4))
for i, patch in enumerate(kmeans.cluster_centers_):
    pl.subplot(9, 9, i + 1)
    pl.imshow(patch.reshape(patch_size), cmap=pl.cm.gray,
              interpolation='nearest')
    pl.xticks(())
    pl.yticks(())

pl.suptitle('Patches of faces\nTrain time %.1fs on %d patches' %
            (dt, 8 * len(faces.images)), fontsize=16)
pl.subplots_adjust(0.08, 0.02, 0.92, 0.85, 0.08, 0.23)

pl.show()

```

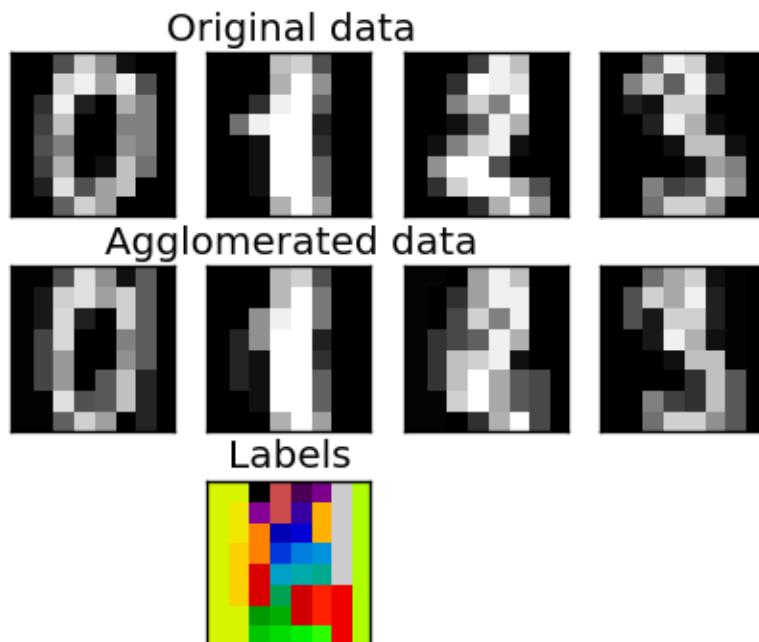
**Total running time of the example:** 12.84 seconds



Figure 2.45: *Feature agglomeration*

## Feature agglomeration

These images show similar features are merged together using feature agglomeration.



**Python source code:** [plot\\_digits\\_agglomeration.py](#)

```
print __doc__

# Code source: Gael Varoquaux
# Modified for Documentation merge by Jaques Grobler
# License: BSD

import numpy as np
import pylab as pl

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.WardAgglomeration(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)
pl.figure(1, figsize=(4, 3.5))
pl.clf()
pl.subplots_adjust(left=.01, right=.99, bottom=.01, top=.91)
for i in range(4):
    pl.subplot(3, 4, i + 1)
    pl.imshow(images[i], cmap=pl.cm.gray, vmax=16, interpolation='nearest')
    pl.xticks(())
    pl.yticks(())
```

```

if i == 1:
    pl.title('Original data')
pl.subplot(3, 4, 4 + i + 1)
pl.imshow(images_restored[i], cmap=pl.cm.gray, vmax=16,
          interpolation='nearest')
if i == 1:
    pl.title('Agglomerated data')
pl.xticks(())
pl.yticks(())
pl.show()

pl.subplot(3, 4, 10)
pl.imshow(np.reshape(aglo.labels_, images[0].shape),
          interpolation='nearest', cmap=pl.cm.spectral)
pl.xticks(())
pl.yticks(())
pl.title('Labels')
pl.show()

```

**Total running time of the example:** 0.69 seconds

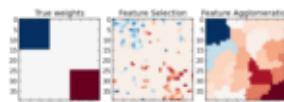


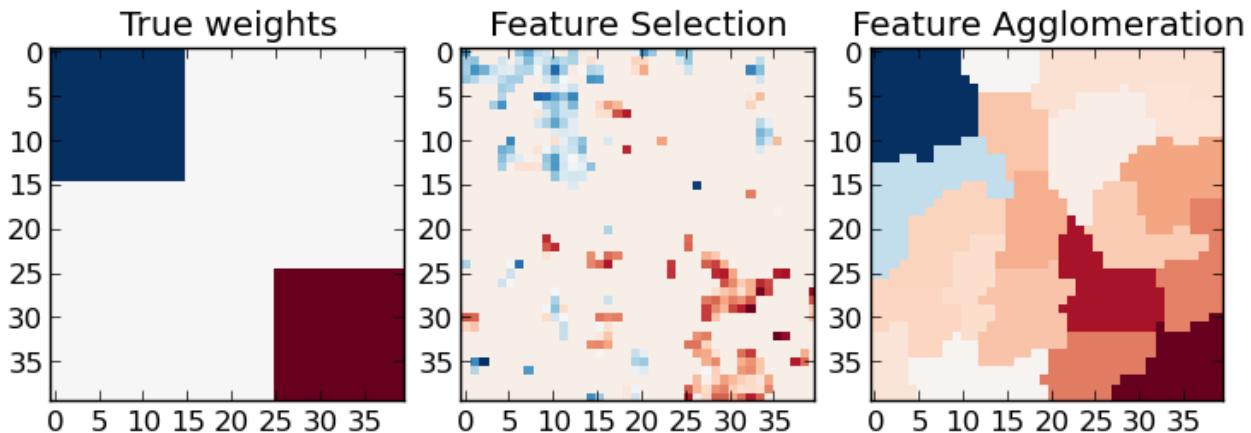
Figure 2.46: Feature agglomeration vs. univariate selection

## 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>, copy=True, n_components=1, n_clusters=10  
ward_tree - 0.2s, 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>, copy=True, n_components=1, n_clusters=10  
ward_tree - 0.1s, 0.0min
```

---

```
[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>, copy=True, n_components=1, n_clusters=20  
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>, copy=True, n_components=1, n_clusters=20  
ward_tree - 0.1s, 0.0min
```

---

```
[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>, copy=True, n_components=1, n_clusters=30  
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>, copy=True, n_components=1, n_clusters=30  
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>, copy=True, n_components=1, n_clusters=20  
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]))
```

```
[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 Style.
```

```
print __doc__
```

```
import shutil  
import tempfile
```

```
import numpy as np  
import pylab as pl
```

```
from sklearn.feature_extraction.image import grid_to_graph
from sklearn import feature_selection
from sklearn.cluster import WardAgglomeration
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
A = grid_to_graph(n_x=size, n_y=size)
ward = WardAgglomeration(n_clusters=10, connectivity=A, memory=mem,
                         n_components=1)
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
pl.close('all')
pl.figure(figsize=(7.3, 2.7))
pl.subplot(1, 3, 1)
pl.imshow(coef, interpolation="nearest", cmap=pl.cm.RdBu_r)
pl.title("True weights")
pl.subplot(1, 3, 2)
pl.imshow(coef_selection_, interpolation="nearest", cmap=pl.cm.RdBu_r)
pl.title("Feature Selection")
pl.subplot(1, 3, 3)
pl.imshow(coef_agglomeration_, interpolation="nearest", cmap=pl.cm.RdBu_r)
pl.title("Feature Agglomeration")
pl.subplots_adjust(0.04, 0.0, 0.98, 0.94, 0.16, 0.26)
pl.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:** 2.50 seconds



Figure 2.47: A demo of K-Means clustering on the handwritten digits data

### A demo of K-Means clustering on the handwritten digits data

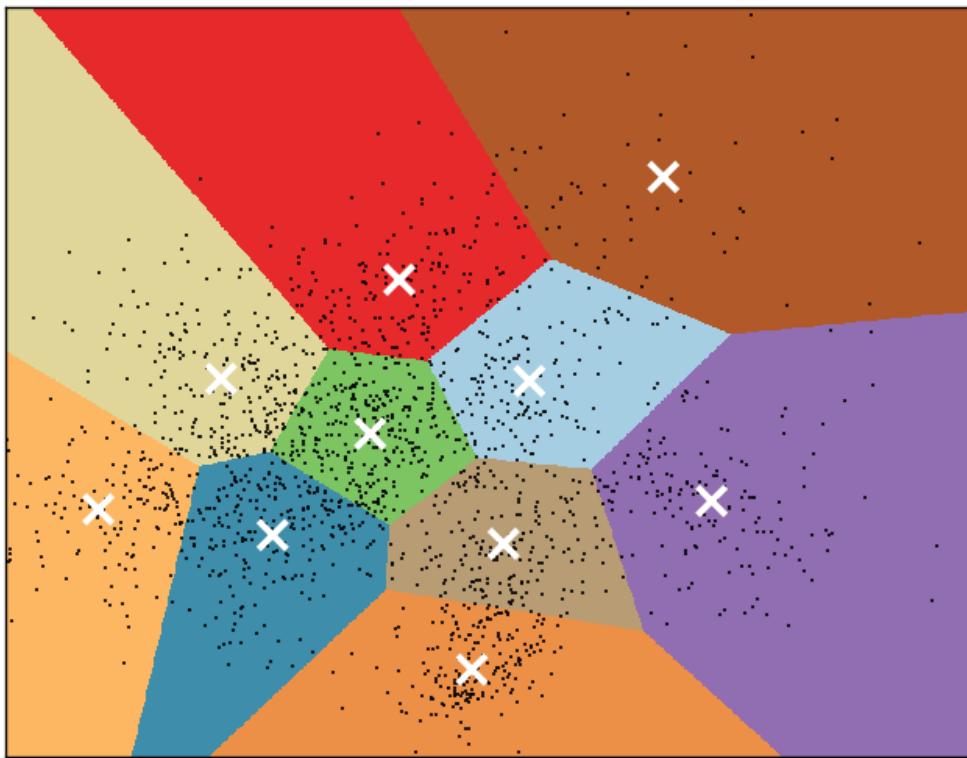
In this example we compare the various initialization strategies for K-means in terms of runtime and quality of the results.

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

K-means clustering on the digits dataset (PCA-reduced data)  
Centroids are marked with white cross



#### Script output:

```
n_digits: 10,           n_samples 1797,           n_features 64
```

	init	time	inertia	homo	compl	v-meas	ARI	AMI	silhouette
k-means++		0.87s	69432	0.602	0.650	0.625	0.465	0.598	0.146
random		1.08s	69694	0.669	0.710	0.689	0.553	0.666	0.147
PCA-based		0.06s	71820	0.673	0.715	0.693	0.567	0.670	0.150

**Python source code:** [plot\\_kmeans\\_digits.py](#)

```
print __doc__\n\nfrom time import time\nimport numpy as np\nimport pylab as pl\n\nfrom sklearn import metrics\nfrom sklearn.cluster import KMeans\nfrom sklearn.datasets import load_digits\nfrom sklearn.decomposition import PCA\nfrom sklearn.preprocessing import scale\n\nnp.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')
      % (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, x_max]x[y_min, y_max].
```

# Plot the decision boundary. For that, we will assign a color to each

x\_min, x\_max = reduced\_data[:, 0].min() + 1, reduced\_data[:, 0].max() - 1

```
y_min, y_max = reduced_data[:, 1].min() + 1, reduced_data[:, 1].max() - 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

# Obtain labels for each point in mesh. Use last trained model.
Z = kmeans.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
pl.figure(1)
pl.clf()
pl.imshow(Z, interpolation='nearest',
           extent=(xx.min(), xx.max(), yy.min(), yy.max()),
           cmap=pl.cm.Paired,
           aspect='auto', origin='lower')

pl.plot(reduced_data[:, 0], reduced_data[:, 1], 'k.', markersize=2)
# Plot the centroids as a white X
centroids = kmeans.cluster_centers_
pl.scatter(centroids[:, 0], centroids[:, 1],
           marker='x', s=169, linewidths=3,
           color='w', zorder=10)
pl.title('K-means clustering on the digits dataset (PCA-reduced data)\n'
          'Centroids are marked with white cross')
pl.xlim(x_min, x_max)
pl.ylim(y_min, y_max)
pl.xticks(())
pl.yticks(())
pl.show()
```

**Total running time of the example:** 2.94 seconds

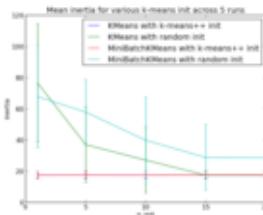


Figure 2.48: Empirical evaluation of the impact of k-means initialization

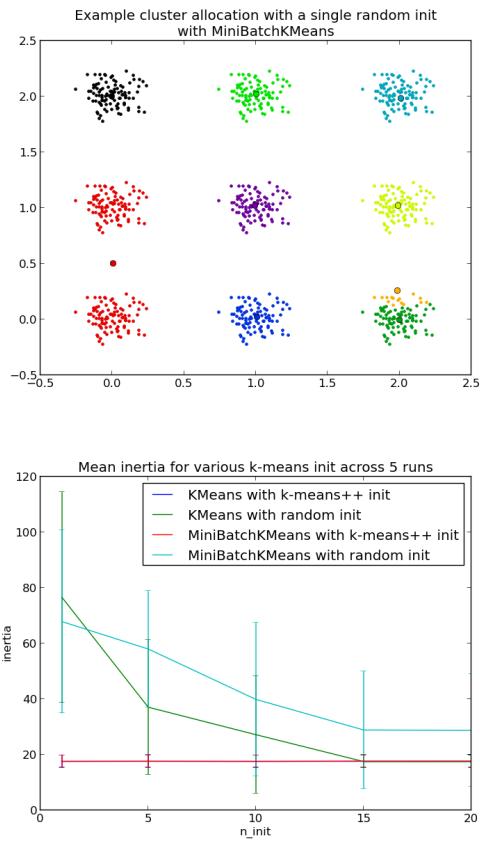
## Empirical evaluation of the impact of k-means initialization

Evaluate the ability of k-means initializations strategies to make the algorithm converge 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 between stucked 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: Simplified BSD

import numpy as np
import pylab as pl
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 init 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 = pl.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 = pl.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))

pl.xlabel('n_init')
pl.ylabel('inertia')
pl.legend(plots, legends)
pl.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 = pl.figure()
for k in range(n_clusters):
    my_members = km.labels_ == k
    color = cm.spectral(float(k) / n_clusters, 1)
    pl.plot(X[my_members, 0], X[my_members, 1], 'o', marker='.', c=color)
    cluster_center = km.cluster_centers_[k]
    pl.plot(cluster_center[0], cluster_center[1], 'o',
            markerfacecolor=color, markeredgecolor='k', markersize=6)
pl.title("Example cluster allocation with a single random init\n"
          "with MiniBatchKMeans")

pl.show()
```

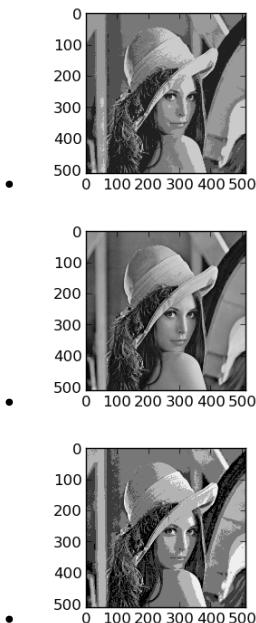
**Total running time of the example:** 4.30 seconds

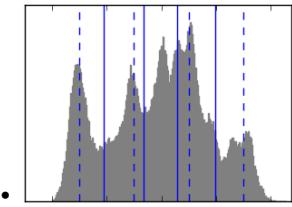


Figure 2.49: *Vector Quantization Example*

## 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__\n\n# Code source: Gael Varoquaux\n# Modified for Documentation merge by Jaques Grobler\n# License: BSD\n\nimport numpy as np\nimport scipy as sp\nimport pylab as pl\n\nfrom sklearn import cluster\n\nn_clusters = 5\nnp.random.seed(0)\n\ntry:\n    lena = sp.lena()\nexcept AttributeError:\n    # Newer versions of scipy have lena in misc\n    from scipy import misc\n    lena = misc.lena()\nX = lena.reshape((-1, 1))\n\nk_means = cluster.KMeans(n_clusters=n_clusters, n_init=4)\nk_means.fit(X)\nvalues = k_means.cluster_centers_.squeeze()\nlabeled = k_means.labels_\n\n# create an array from labels and values\nlena_compressed = np.choose(labeled, values)\nlena_compressed.shape = lena.shape\n\nvmin = lena.min()\nvmax = lena.max()\n\n# original lena\npl.figure(1, figsize=(3, 2.2))\npl.imshow(lena, cmap=pl.cm.gray, vmin=vmin, vmax=256)\n\n# compressed lena\npl.figure(2, figsize=(3, 2.2))\npl.imshow(lena_compressed, cmap=pl.cm.gray, vmin=vmin, vmax=vmax)\n\n# equal bins lena\nregular_values = np.linspace(0, 256, n_clusters + 1)\nregular_labels = np.searchsorted(regular_values, lena) - 1\nregular_values = .5 * (regular_values[1:] + regular_values[:-1])\nregular_lena = np.choose(regular_labels.ravel(), regular_values)\nregular_lena.shape = lena.shape
```

```

pl.figure(3, figsize=(3, 2.2))
pl.imshow(regular_lena, cmap=pl.cm.gray, vmin=vmin, vmax=vmax)

# histogram
pl.figure(4, figsize=(3, 2.2))
pl.clf()
pl.axes([.01, .01, .98, .98])
pl.hist(X, bins=256, color='5', edgecolor='5')
pl.yticks(())
pl.xticks(regular_values)
values = np.sort(values)
for center_1, center_2 in zip(values[:-1], values[1:]):
    pl.axvline(.5 * (center_1 + center_2), color='b')

for center_1, center_2 in zip(regular_values[:-1], regular_values[1:]):
    pl.axvline(.5 * (center_1 + center_2), color='b', linestyle='--')

pl.show()

```

**Total running time of the example:** 1.84 seconds



Figure 2.50: Segmenting the picture of Lena in regions

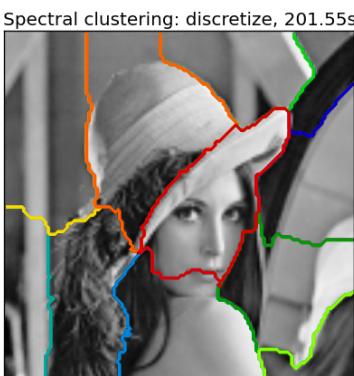
### Segmenting the picture of Lena in regions

This example uses *Spectral clustering* on a graph created from voxel-to-voxel difference on an image to break this image into multiple partly-homogenous 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.



**Python source code:** [plot\\_lena\\_segmentation.py](#)

```
print __doc__

# Author: Gael Varoquaux <gael.varoquaux@normalesup.org>, Brian Cheung
# License: BSD

import time

import numpy as np
import scipy as sp
import pylab as pl

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)

    pl.figure(figsize=(5, 5))
    pl.imshow(lena, cmap=pl.cm.gray)
    for l in range(N_REGIONS):
        pl.contour(labels == l, contours=1,
                   colors=[pl.cm.spectral(l / float(N_REGIONS)), ])
    pl.xticks(())
    pl.yticks(())
    pl.title('Spectral clustering: %s, %.2fs' % (assign_labels, (t1 - t0)))

pl.show()

```

**Total running time of the example:** 378.51 seconds



Figure 2.51: A demo of structured Ward hierarchical clustering on Lena image

### 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: 8.80133795738
Number of pixels: 65536
Number of clusters: 15
```

**Python source code:** [plot\\_lena\\_ward\\_segmentation.py](#)

```
# Author : Vincent Michel, 2010
#          Alexandre Gramfort, 2011
# License: BSD Style.

print __doc__

import time as time
import numpy as np
import scipy as sp
import pylab as pl
from sklearn.feature_extraction.image import grid_to_graph
from sklearn.cluster import Ward

#####
# 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 = Ward(n_clusters=n_clusters, 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
pl.figure(figsize=(5, 5))
pl.imshow(lena, cmap=pl.cm.gray)
for l in range(n_clusters):
    pl.contour(label == l, contours=1,
               colors=[pl.cm.spectral(l / float(n_clusters)), ])
pl.xticks(())
pl.yticks(())
pl.show()

```

**Total running time of the example:** 9.17 seconds

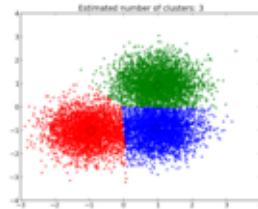
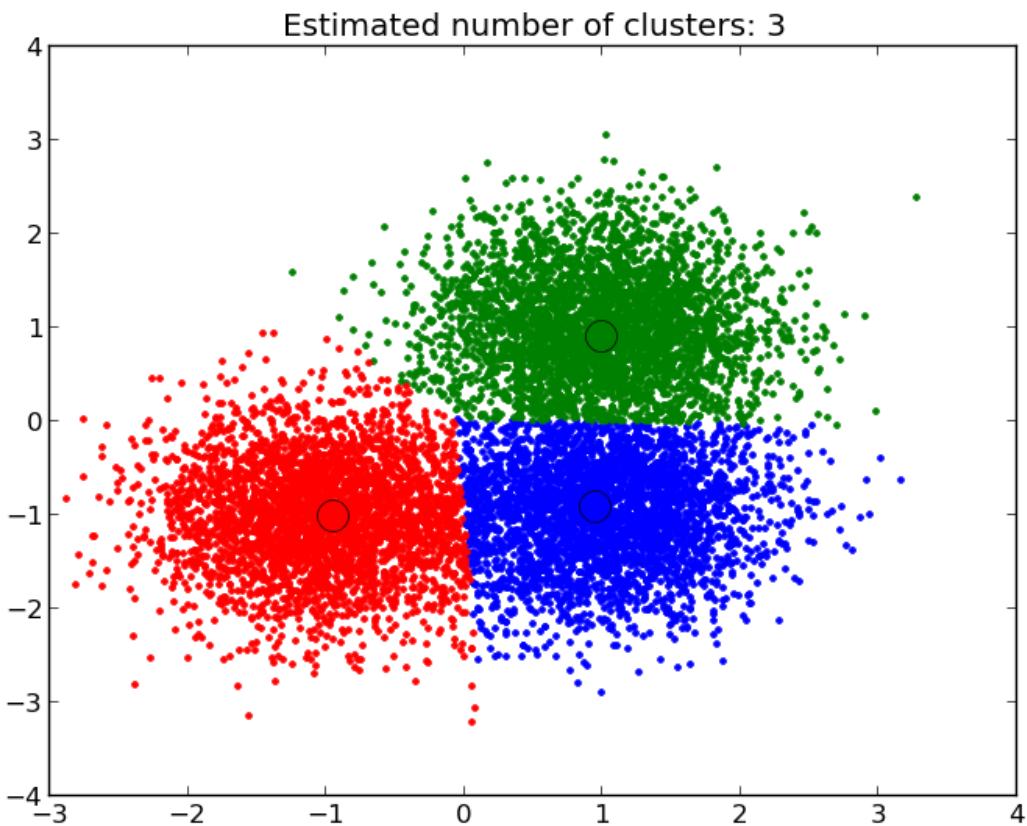


Figure 2.52: A demo of the mean-shift clustering algorithm

## 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 pylab as pl
from itertools import cycle

pl.figure(1)
pl.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    my_members = labels == k
    cluster_center = cluster_centers[k]
    pl.plot(X[my_members, 0], X[my_members, 1], col + '.')
    pl.plot(cluster_center[0], cluster_center[1], 'o',
            markerfacecolor=col,
            markeredgecolor='k', markersize=14)
pl.title('Estimated number of clusters: %d' % n_clusters_)
pl.show()

```

**Total running time of the example:** 0.55 seconds

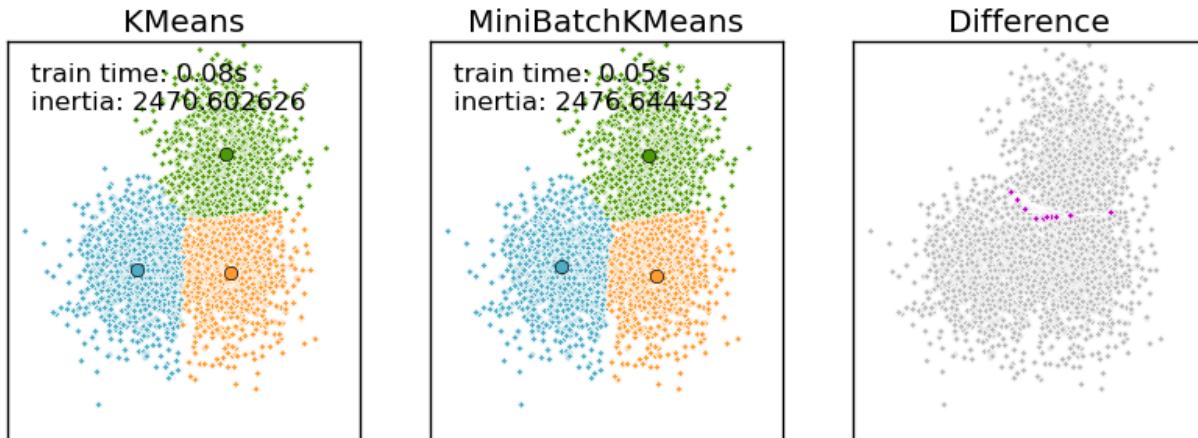


Figure 2.53: A demo of the K Means clustering algorithm

### A demo of the K Means clustering algorithm

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__\n\nimport time\n\nimport numpy as np\nimport pylab as pl\n\nfrom sklearn.cluster import MiniBatchKMeans, KMeans\nfrom sklearn.metrics.pairwise import euclidean_distances\nfrom sklearn.datasets.samples_generator import make_blobs\n\n#####\n# Generate sample data\nnp.random.seed(0)\n\nbatch_size = 45\ncenters = [[1, 1], [-1, -1], [1, -1]]\nn_clusters = len(centers)\nX, labels_true = make_blobs(n_samples=3000, centers=centers, cluster_std=0.7)\n\n#####\n# Compute clustering with Means\n\nk_means = KMeans(init='k-means++', n_clusters=3, n_init=10)\nt0 = time.time()\nk_means.fit(X)\nt_batch = time.time() - t0\nk_means_labels = k_means.labels_\nk_means_cluster_centers = k_means.cluster_centers_\nk_means_labels_unique = np.unique(k_means_labels)\n\n#####\n# Compute clustering with MiniBatchKMeans\n\nmbk = MiniBatchKMeans(init='k-means++', n_clusters=3, batch_size=batch_size,\n                      n_init=10, max_no_improvement=10, verbose=0)\nt0 = time.time()\nmbk.fit(X)\nt_mini_batch = time.time() - t0\nmbk_means_labels = mbk.labels_\nmbk_means_cluster_centers = mbk.cluster_centers_\nmbk_means_labels_unique = np.unique(mbk_means_labels)\n\n#####\n# Plot result\n\nfig = pl.figure(figsize=(8, 3))\nfig.subplots_adjust(left=0.02, right=0.98, bottom=0.05, top=0.9)\ncolors = ['#4EACC5', '#FF9C34', '#4E9A06']\n\n# We want to have the same colors for the same cluster from the\n# MiniBatchKMeans and the KMeans algorithm. Let's pair the cluster centers per\n# closest one.\n\ndistance = euclidean_distances(k_means_cluster_centers,\n                                mbk_means_cluster_centers,\n                                squared=True)
```

```

order = distance.argmin(axis=1)

# 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(())
pl.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(())
pl.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(())

pl.show()

```

**Total running time of the example:** 0.35 seconds

## 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 known 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

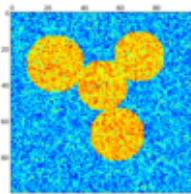


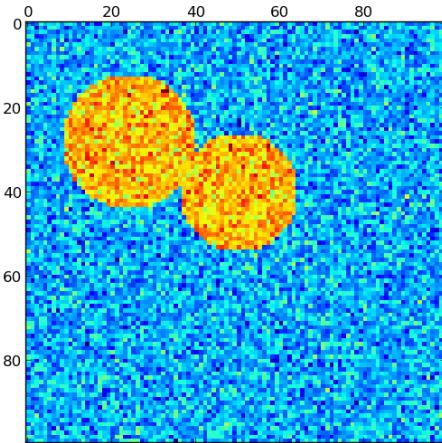
Figure 2.54: Spectral clustering for image segmentation

regions while minimizing the ratio of the gradient along the cut, and the volume of the region.

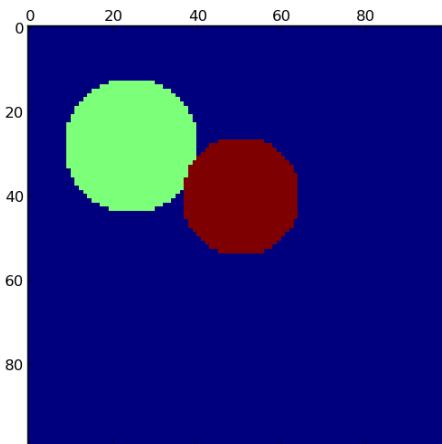
As the algorithm tries to balance the volume (ie balance the region sizes), if we take circles with different sizes, the segmentation fails.

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.

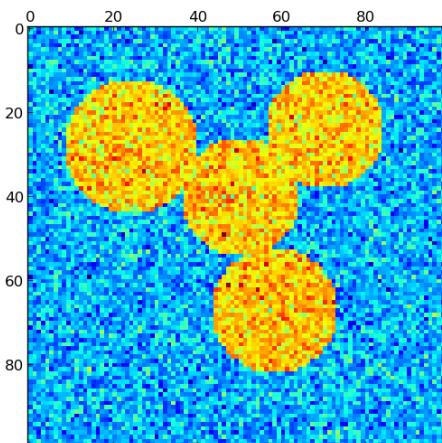
In addition, we use the mask of the objects to restrict the graph to the outline of the objects. In this example, we are interested in separating the objects one from the other, and not from the background.



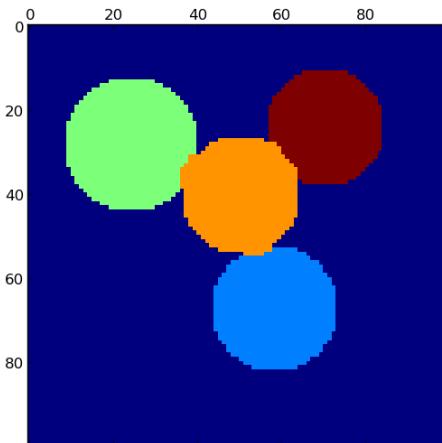
•



•



•



•

**Python source code:** [plot\\_segmentation\\_toy.py](#)

```
print __doc__

# Authors: Emmanuelle Gouillart <emmanuelle.gouillart@normalesup.org>
#          Gael Varoquaux <gael.varoquaux@normalesup.org>
# License: BSD

import numpy as np
import pylab as pl

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
# dependant 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

pl.matshow(img)
pl.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

pl.matshow(img)
pl.matshow(label_im)

pl.show()

```

**Total running time of the example:** 1.67 seconds

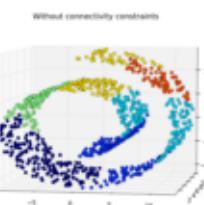


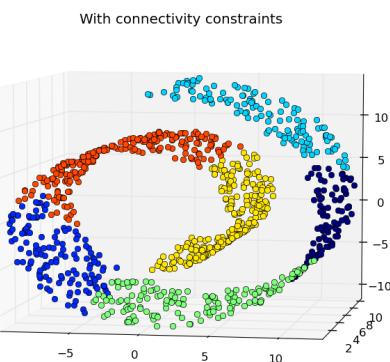
Figure 2.55: *Hierarchical clustering: structured vs unstructured ward*

### Hierarchical clustering: structured vs unstructured ward

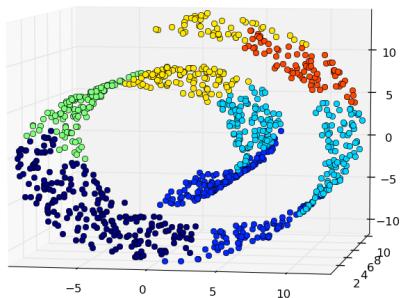
Example builds a swiss roll dataset and runs *Hierarchical clustering* on their position.

In a first step, the hierarchical clustering without connectivity constraints on structure, solely based on distance, whereas in a second step clustering restricted to the k-Nearest Neighbors graph: it's a hierarchical clustering with structure prior.

Some of the clusters learned without connectivity constraints do not respect the structure of the swiss roll and extend across different folds of the manifolds. On the opposite, when opposing connectivity constraints, the clusters form a nice parcellation of the swiss roll.



Without connectivity constraints



•

**Script output:**

```
Compute unstructured hierarchical clustering...
Elapsed time: 0.800763130188
Number of points: 1000
Compute structured hierarchical clustering...
Elapsed time: 0.170639999176
Number of points: 1000
```

**Python source code:** [plot\\_ward\\_structured\\_vs\\_unstructured.py](#)

```
# Authors : Vincent Michel, 2010
#           Alexandre Gramfort, 2010
#           Gael Varoquaux, 2010
# License: BSD

print __doc__

import time as time
import numpy as np
import pylab as pl
import mpl_toolkits.mplot3d.axes3d as p3
from sklearn.cluster import Ward
from sklearn.datasets.samples_generator import make_swiss_roll

#####
# Generate data (swiss roll dataset)
n_samples = 1000
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 = Ward(n_clusters=6).fit(X)
label = ward.labels_
print "Elapsed time: ", time.time() - st
print "Number of points: ", label.size

#####
# Plot result
```

```

fig = pl.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=pl.cm.jet(np.float(l) / np.max(label + 1)))
pl.title('Without connectivity constraints')

#####
# 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)

#####
# Compute clustering
print "Compute structured hierarchical clustering..."
st = time.time()
ward = Ward(n_clusters=6, connectivity=connectivity).fit(X)
label = ward.labels_
print "Elapsed time: ", time.time() - st
print "Number of points: ", label.size

#####
# Plot result
fig = pl.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=pl.cm.jet(float(l) / np.max(label + 1)))
pl.title('With connectivity constraints')

pl.show()

```

**Total running time of the example:** 1.16 seconds

## 2.1.4 Covariance estimation

Examples concerning the `sklearn.covariance` package.

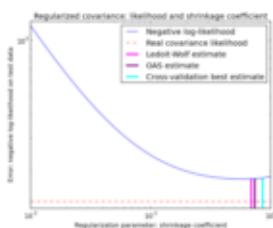


Figure 2.56: Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood

## Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood

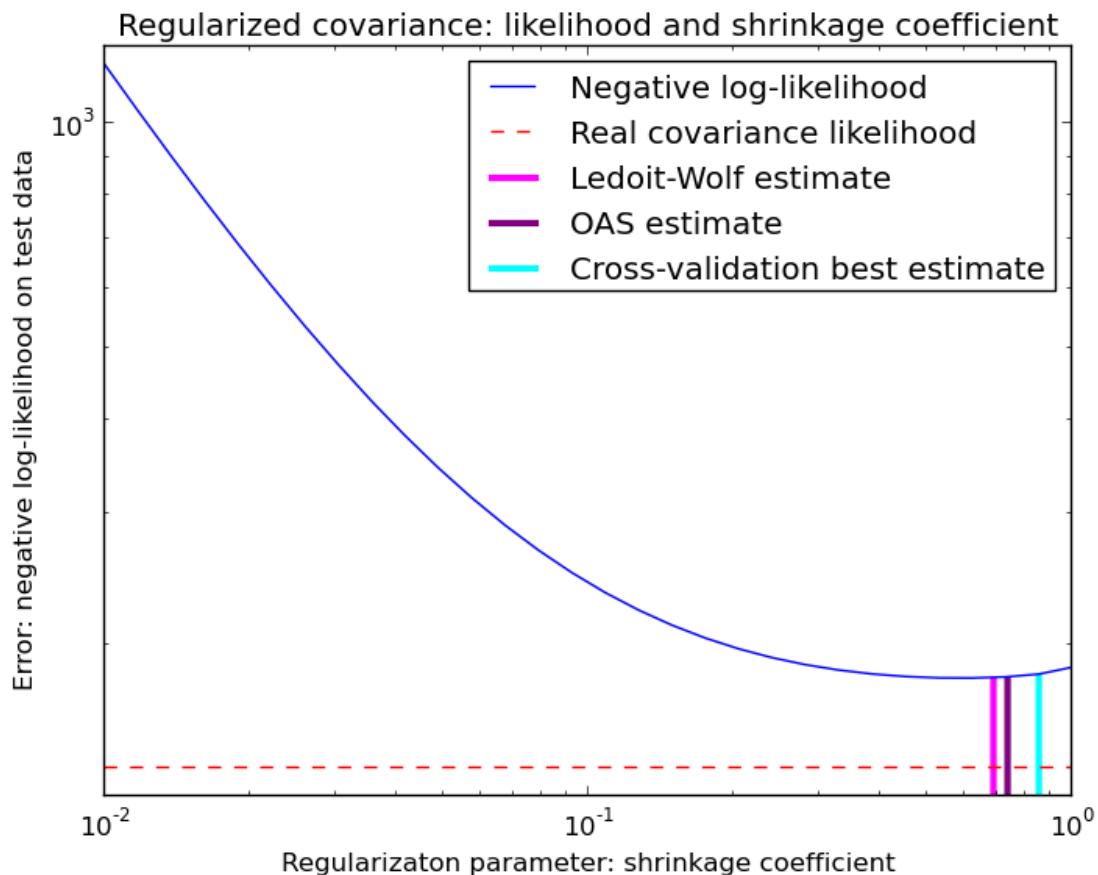
The usual estimator for covariance is the maximum likelihood estimator, `sklearn.covariance.EmpiricalCovariance`. It is unbiased, i.e. it converges to the true (population) covariance when given many observations. However, it can also be beneficial to regularize it, in order to reduce its variance; this, 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 asymptotical 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 pylab as pl
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 = pl.figure()
pl.title("Regularized covariance: likelihood and shrinkage coefficient")
pl.xlabel('Regularization parameter: shrinkage coefficient')
pl.ylabel('Error: negative log-likelihood on test data')
# range shrinkage curve
pl.loglog(shrinkages, negative_logliks, label="Negative log-likelihood")

pl.plot(pl.xlim(), 2 * [loglik_real], '--r',
        label="Real covariance likelihood")

# adjust view
lik_max = npamax(negative_logliks)
lik_min = npamin(negative_logliks)
ymin = lik_min - 6. * np.log((pl.ylim()[1] - pl.ylim()[0]))
ymax = lik_max + 10. * np.log(lik_max - lik_min)
xmin = shrinkages[0]
xmax = shrinkages[-1]
# LW likelihood
pl.vlines(lw.shrinkage_, ymin, -loglik_lw, color='magenta',
           linewidth=3, label='Ledoit-Wolf estimate')
# OAS likelihood
pl.vlines(oa.shrinkage_, ymin, -loglik_oa, color='purple',
           linewidth=3, label='OAS estimate')
# best CV estimator likelihood
pl.vlines(cv.best_estimator_.shrinkage, ymin,
           -cv.best_estimator_.score(X_test), color='cyan',
           linewidth=3, label='Cross-validation best estimate')
```

```

pl.ylim(ymin, ymax)
pl.xlim(xmin, xmax)
pl.legend()

pl.show()

```

**Total running time of the example:** 0.34 seconds

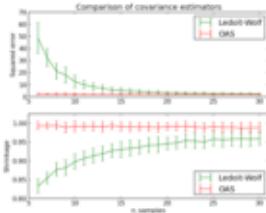


Figure 2.57: *Ledoit-Wolf vs OAS estimation*

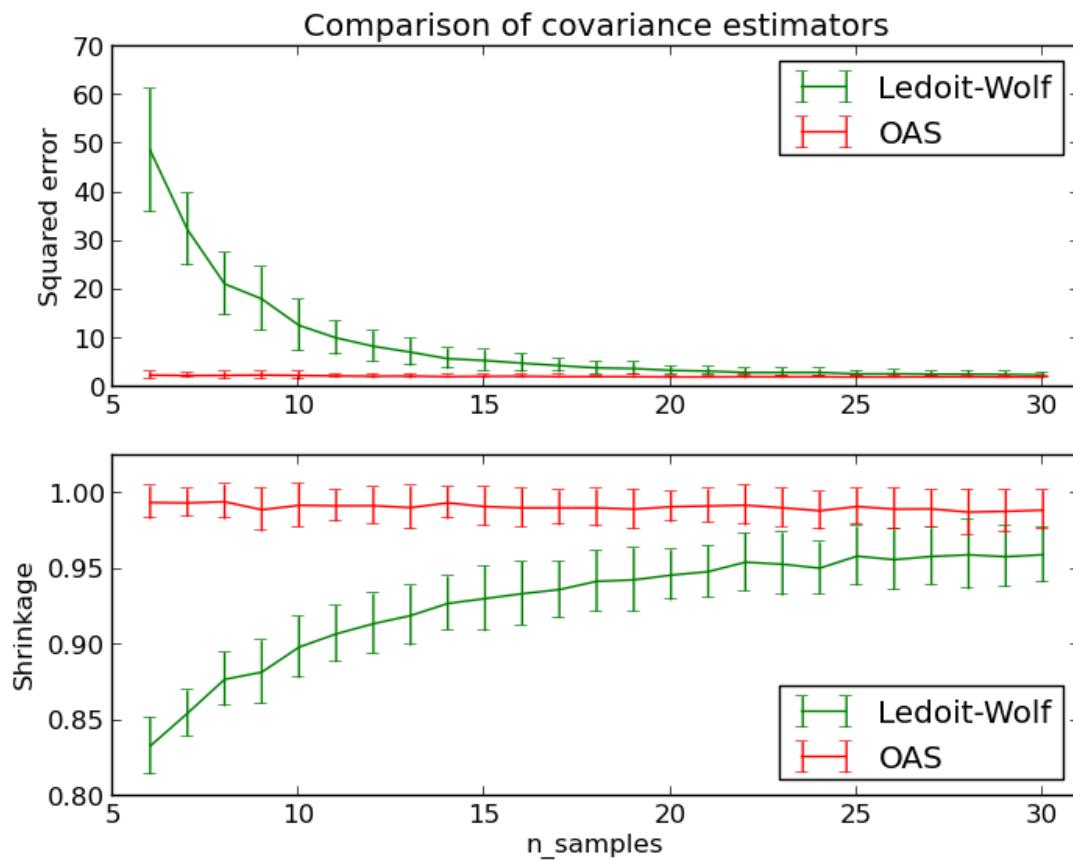
### Ledoit-Wolf vs OAS estimation

The usual covariance maximum likelihood estimate can be regularized using shrinkage. Ledoit and Wolf proposed a close formula to compute the asymptotical 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 pylab as pl
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
pl.subplot(2, 1, 1)
pl.errorbar(n_samples_range, lw_mse.mean(1), yerr=lw_mse.std(1),
            label='Ledoit-Wolf', color='g')
pl.errorbar(n_samples_range, oa_mse.mean(1), yerr=oa_mse.std(1),
            label='OAS', color='r')
pl.ylabel("Squared error")
pl.legend(loc="upper right")
pl.title("Comparison of covariance estimators")
pl.xlim(5, 31)

# plot shrinkage coefficient
pl.subplot(2, 1, 2)
pl.errorbar(n_samples_range, lw_shrinkage.mean(1), yerr=lw_shrinkage.std(1),
            label='Ledoit-Wolf', color='g')
pl.errorbar(n_samples_range, oa_shrinkage.mean(1), yerr=oa_shrinkage.std(1),
            label='OAS', color='r')
pl.xlabel("n_samples")
pl.ylabel("Shrinkage")
pl.legend(loc="lower right")
pl.ylim(pl.ylim()[0], 1. + (pl.ylim()[1] - pl.ylim()[0]) / 10.)
pl.xlim(5, 31)

pl.show()

```

**Total running time of the example:** 6.69 seconds

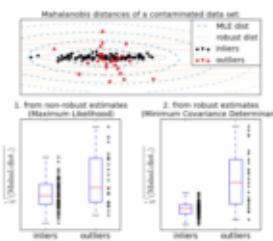


Figure 2.58: Robust covariance estimation and Mahalanobis distances relevance

## Robust covariance estimation and Mahalanobis distances relevance

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  $\mu$  and  $\Sigma$  are the location and the covariance of the underlying gaussian distribution.

In practice,  $\mu$  and  $\Sigma$  are replaced by some estimates. The usual covariance maximum likelihood estimate is very sensitive to the presence of outliers in the data set and therefore, 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.

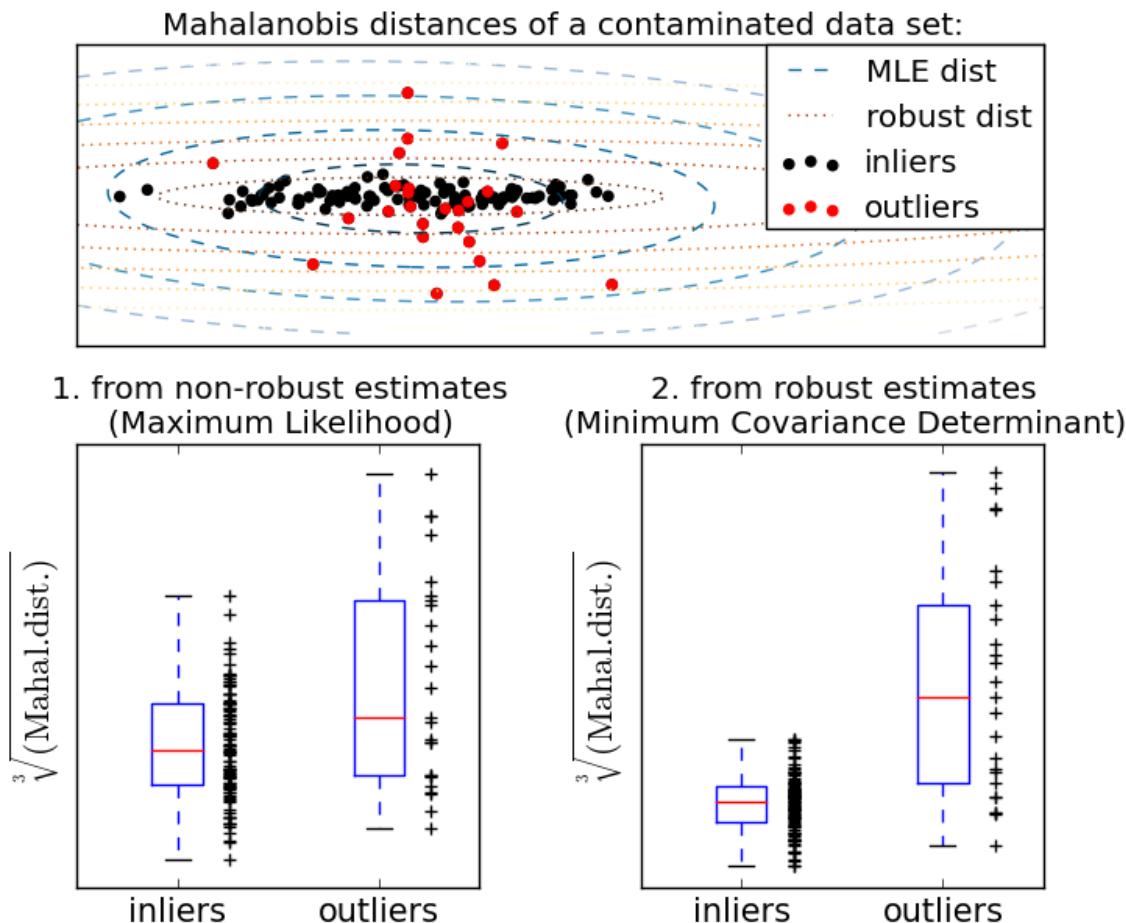
The Minimum Covariance Determinant estimator is a robust, high-breakdown point (i.e. it can be used to estimate the covariance matrix of highly contaminated datasets, up to :math:`\lceil \frac{n\\_samples-n\\_features-1}{2} \rceil` outliers) estimator of covariance. The idea is to find :math:`\lceil \frac{n\\_samples+n\\_features+1}{2} \rceil` observations whose empirical covariance has the smallest determinant, yielding a “pure” subset of observations from which to compute standard estimates of location and covariance.

The Minimum Covariance Determinant estimator (MCD) has been introduced by P.J.Rousseeuw 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 Holferty suggest [2]

[1] P. J. Rousseeuw. Least median of squares regression. *J. Am Stat Ass*, 79:871, 1984.

[2] Wilson, E. B., & Holferty, 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 pylab as pl

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 = pl.figure()
pl.subplots_adjust(hspace=-.1, wspace=.4, top=.95, bottom=.05)

# Show data set
subfig1 = pl.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(pl.xlim()[0], pl.xlim()[1], 100),
                     np.linspace(pl.ylim()[0], pl.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=pl.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=pl.cm.YlOrBr_r, linestyles='dotted')

subfig1.legend([emp_cov_contour.collections[1], robust_contour.collections[1],
               inlier_plot, outlier_plot],
              loc='upper right')

```

```

['MLE dist', 'robust dist', 'inliers', 'outliers'],
loc="upper right", borderaxespad=0)

pl.xticks(())
pl.yticks(())

# Plot the scores for each point
emp_mahal = emp_cov.mahalanobis(X - np.mean(X, 0)) ** (0.33)
subfig2 = pl.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"$\sqrt{3} \{\text{Mahal. dist.}\}$", size=16)
subfig2.set_title("1. from non-robust estimates\n(Maximum Likelihood)")
pl.yticks(())

robust_mahal = robust_cov.mahalanobis(X - robust_cov.location_) ** (0.33)
subfig3 = pl.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"$\sqrt{3} \{\text{Mahal. dist.}\}$", size=16)
subfig3.set_title("2. from robust estimates\n(Minimum Covariance Determinant)")
pl.yticks(())

pl.show()

```

**Total running time of the example:** 0.28 seconds

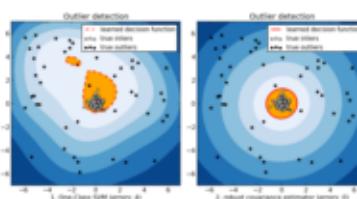


Figure 2.59: Outlier detection with several methods.

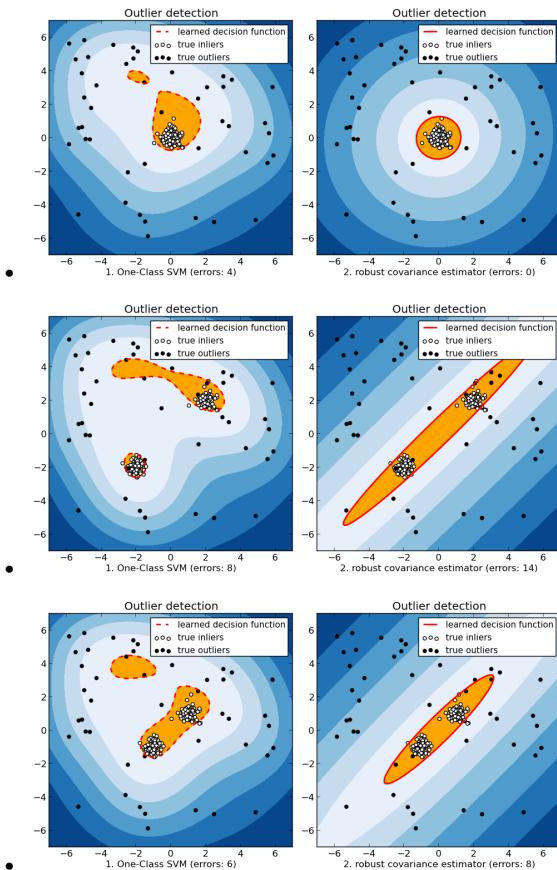
### Outlier detection with several methods.

This example illustrates two ways of performing *Novelty and Outlier Detection* when the amount of contamination is known:

- 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 outliers 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.



**Python source code:** [plot\\_outlier\\_detection.py](#)

```
print __doc__

import numpy as np
import pylab as pl
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
    pl.figure(figsize=(10, 5))
    for i, (clf_name, clf) in enumerate(classifiers.iteritems()):
        # 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 = pl.subplot(1, 2, i + 1)
        subplot.set_title("Outlier detection")
        subplot.contourf(xx, yy, Z, levels=np.linspace(Z.min(), threshold, 7),
                         cmap=pl.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))
    pl.subplots_adjust(0.04, 0.1, 0.96, 0.94, 0.1, 0.26)

pl.show()
```

**Total running time of the example:** 2.16 seconds

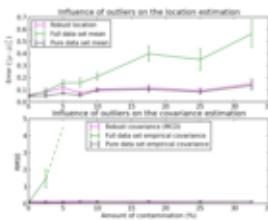


Figure 2.60: Robust vs Empirical covariance estimate

### 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, one would have better to use a robust estimator of covariance to garanty that the estimation is resistant to “erroneous” observations in the data set.

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  $\text{rac}\{\text{n\_samples}-\text{n\_features}-1\}\{2\}$  outliers) estimator of covariance. The idea is to find  $\text{rac}\{\text{n\_samples}+\text{n\_features}+1\}\{2\}$  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 the the estimates were learnt from only a portion of the initial data, we end up with robust estimates of the data set location and covariance.

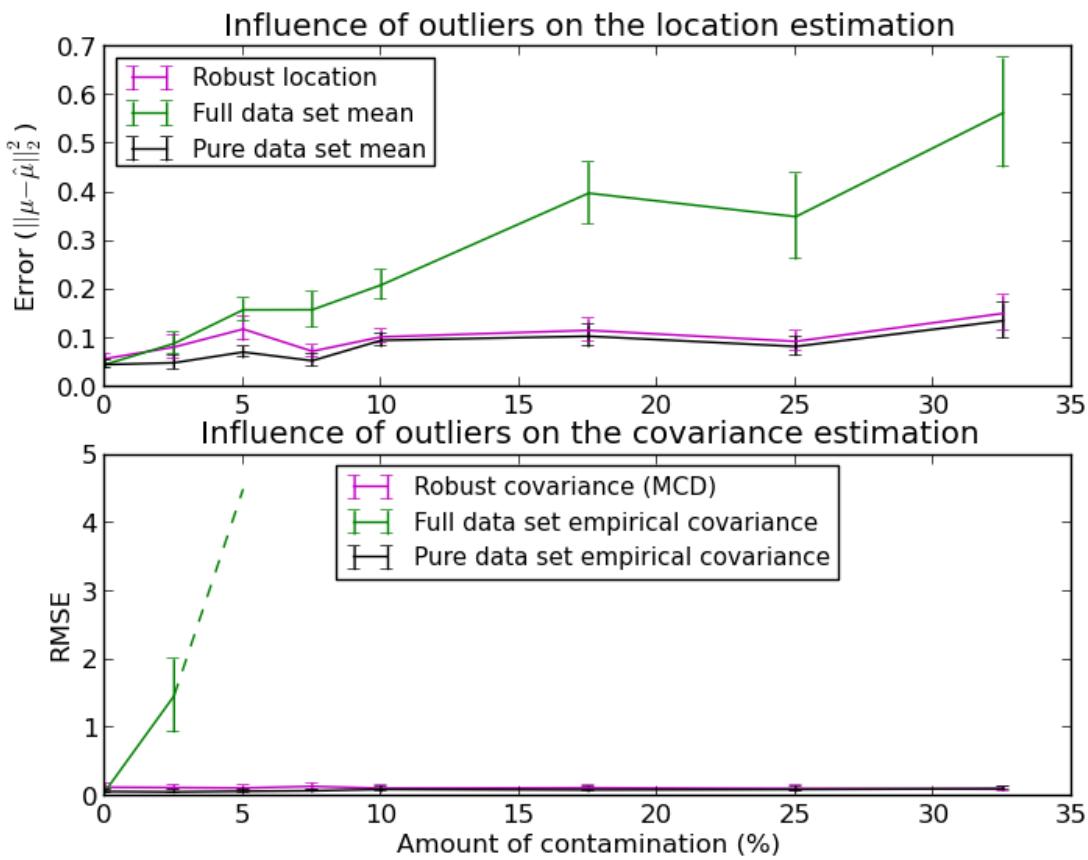
The Minimum Covariance Determinant estimator (MCD) has been introduced by P.J.Rousseeuw in [1].

In this example, we compare the estimation errors that are made when using three 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  $\text{n\_samples} > 5 * \text{n\_features}$
- 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.

[2] Johanna Hardin, David M Rocke. Journal of Computational and Graphical Statistics. December 1, 2005, 14(4): 928-946.



**Python source code:** [plot\\_robust\\_vs\\_empirical\\_covariance.py](#)

```
print __doc__

import numpy as np
import pylab as pl
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):
        # generate data
        X = np.random.randn(n_samples, n_features)
        # add some outliers
        outliers_index = np.random.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
        S = MinCovDet().fit(X)
        # compare raw robust estimates with the true location and covariance
        err_loc_mcd[i, j] = np.sum(S.location_ ** 2)
        err_cov_mcd[i, j] = S.error_norm(np.eye(n_features))
        # compare estimators learnt 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 learnt 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)
pl.subplot(2, 1, 1)
pl.errorbar(range_n_outliers, err_loc_mcd.mean(1),
            yerr=err_loc_mcd.std(1) / np.sqrt(repeat),
            label="Robust location", color='m')
pl.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')
pl.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')
pl.title("Influence of outliers on the location estimation")
pl.ylabel(r"Error ($||\mu - \hat{\mu}||_2^2$)")
pl.legend(loc="upper left", prop=font_prop)

pl.subplot(2, 1, 2)
x_size = range_n_outliers.size
pl.errorbar(range_n_outliers, err_cov_mcd.mean(1),
            yerr=err_cov_mcd.std(1),
            label="Robust covariance (MCD)", color='m')
pl.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')
pl.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='--')

```

```
pl.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')
pl.title("Influence of outliers on the covariance estimation")
pl.xlabel("Amount of contamination (%)")
pl.ylabel("RMSE")
pl.legend(loc="upper center", prop=font_prop)

pl.show()
```

**Total running time of the example:** 3.44 seconds

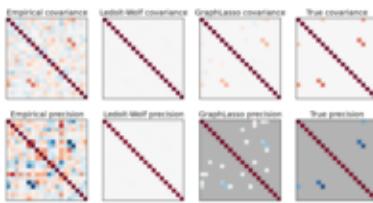


Figure 2.61: *Sparse inverse covariance estimation*

## 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 are 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.

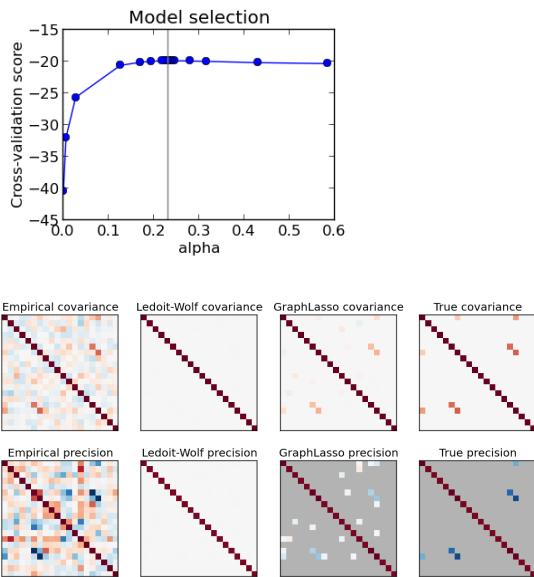
Here, the number of samples is slightly larger than the number of dimensions, thus the empirical covariance is still invertible. However, as the observations are strongly correlated, the empirical covariance matrix is ill-conditioned and as a result its inverse –the empirical precision matrix– is very 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](#)

```
print __doc__
# author: Gael Varoquaux <gael.varoquaux@inria.fr>
# License: BSD Style
# 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 pylab as pl

#####
# 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
pl.figure(figsize=(10, 6))
pl.subplots_adjust(left=0.02, right=0.98)

# plot the covariances
cows = [('Empirical', emp_cov), ('Ledoit-Wolf', lw_cov_),
         ('GraphLasso', cov_), ('True', cov)]
vmax = cov_.max()
for i, (name, this_cov) in enumerate(cows):
    pl.subplot(2, 4, i + 1)
    pl.imshow(this_cov, interpolation='nearest', vmin=-vmax, vmax=vmax,
              cmap=pl.cm.RdBu_r)
    pl.xticks(())
    pl.yticks(())
    pl.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 = pl.subplot(2, 4, i + 5)
    pl.imshow(np.ma.masked_equal(this_prec, 0),
              interpolation='nearest', vmin=-vmax, vmax=vmax,
              cmap=pl.cm.RdBu_r)
    pl.xticks(())
    pl.yticks(())
    pl.title('%s precision' % name)
    ax.set_axis_bgcolor('.7')

# plot the model selection metric
pl.figure(figsize=(4, 3))
pl.axes([.2, .15, .75, .7])
pl.plot(model.cv_alphas_, np.mean(model.cv_scores, axis=1), 'o-')
pl.axvline(model.alpha_, color='.5')
pl.title('Model selection')
pl.ylabel('Cross-validation score')
pl.xlabel('alpha')

pl.show()
```

**Total running time of the example:** 0.86 seconds

## 2.1.5 Dataset examples

Examples concerning the `sklearn.datasets` package.

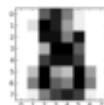
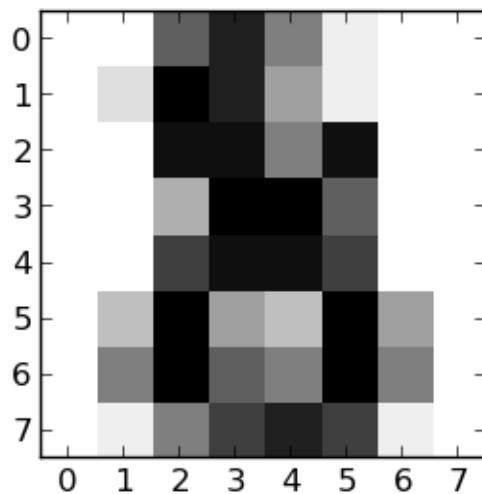


Figure 2.62: The Digit Dataset

## 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 utilise 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.



**Python source code:** [plot\\_digits\\_last\\_image.py](#)

```
print __doc__

# Code source: Gael Varoquaux
# Modified for Documentation merge by Jaques Grobler
# License: BSD

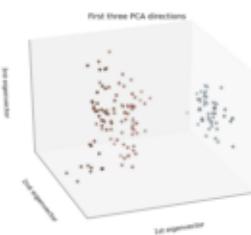
from sklearn import datasets

import pylab as pl

#Load the digits dataset
digits = datasets.load_digits()

#Display the first digit
pl.figure(1, figsize=(3, 3))
pl.imshow(digits.images[-1], cmap=pl.cm.gray_r, interpolation='nearest')
pl.show()
```

**Total running time of the example:** 0.21 seconds

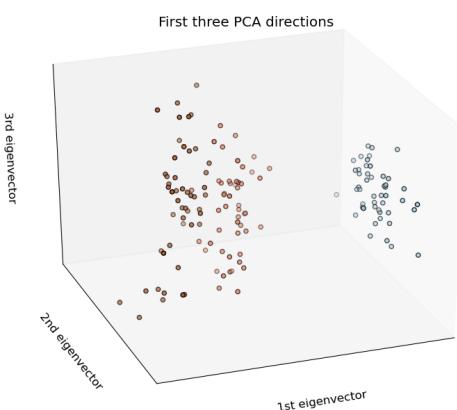
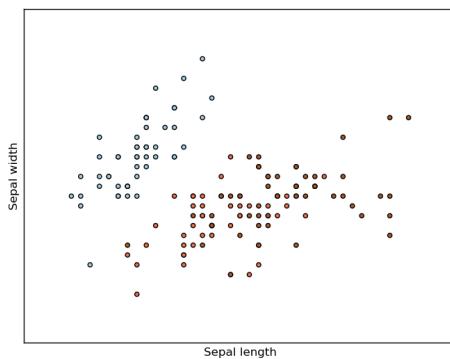
Figure 2.63: *The Iris Dataset*

## The Iris Dataset

This data set 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.



**Python source code:** [plot\\_iris\\_dataset.py](#)

```
print __doc__
```

```
# Code source: Gael Varoquaux
# Modified for Documentation merge by Jaques Grobler
# License: BSD
```

```

import pylab as pl
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

pl.figure(2, figsize=(8, 6))
pl.clf()

# Plot the training points
pl.scatter(X[:, 0], X[:, 1], c=Y, cmap=pl.cm.Paired)
pl.xlabel('Sepal length')
pl.ylabel('Sepal width')

pl.xlim(x_min, x_max)
pl.ylim(y_min, y_max)
pl.xticks(())
pl.yticks(())

# To get a better understanding of interaction of the dimensions
# plot the first three PCA dimensions
fig = pl.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=pl.cm.Paired)
ax.set_title("First three PCA directions")
ax.set_xlabel("1st eigenvector")
ax.set_xticks(())
ax.set_ylabel("2nd eigenvector")
ax.set_yticks(())
ax.set_zlabel("3rd eigenvector")
ax.set_zticks(())

pl.show()

```

**Total running time of the example:** 0.14 seconds

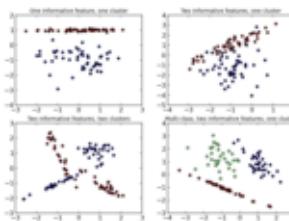
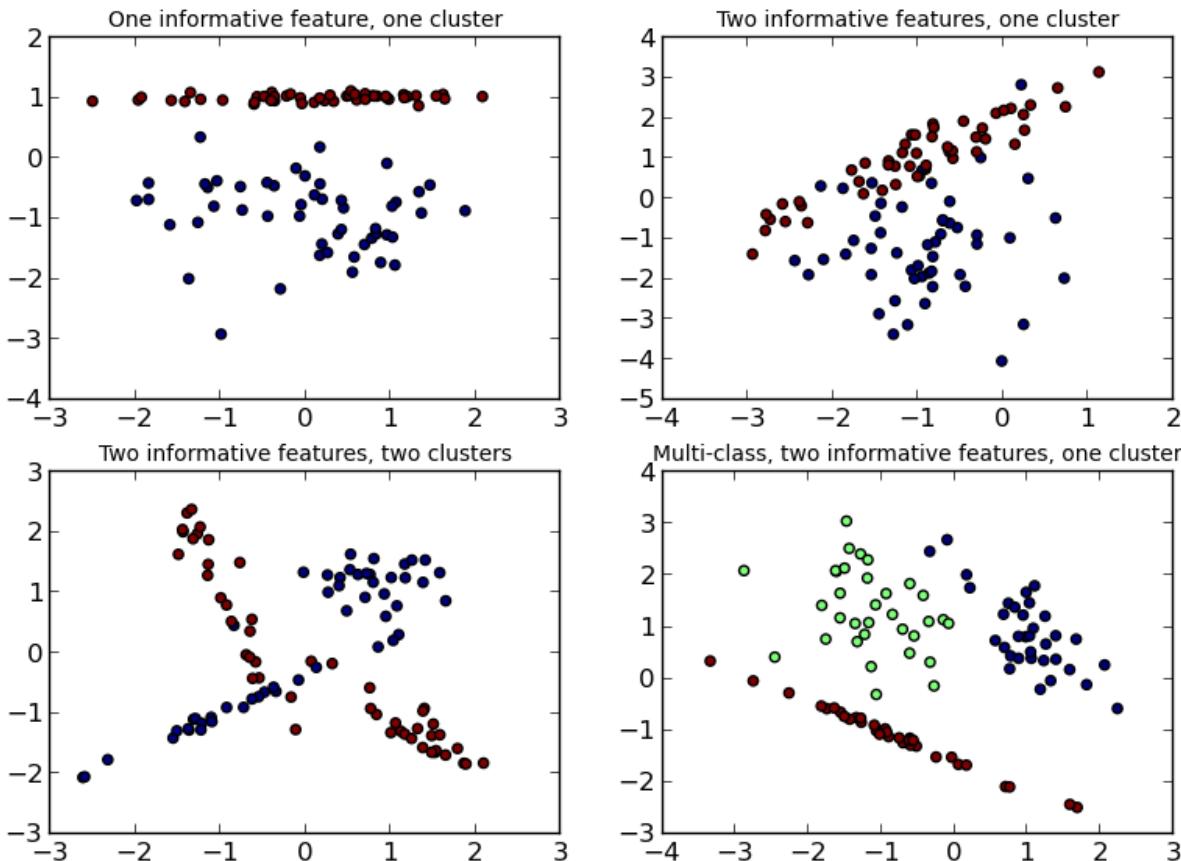


Figure 2.64: Plot randomly generated classification dataset

## Plot randomly generated classification dataset

Plot several randomly generated 2D classification datasets. This example illustrates the `datasets.make_classification` function.

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 pylab as pl

from sklearn.datasets import make_classification

pl.figure(figsize=(8, 6))
pl.subplots_adjust(bottom=.05, top=.9, left=.05, right=.95)

pl.subplot(221)
pl.title("One informative feature, one cluster", fontsize='small')
X1, Y1 = make_classification(n_features=2, n_redundant=0, n_informative=1,
                             n_clusters_per_class=1)
pl.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1)

pl.subplot(222)
```

```

pl.title("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)
pl.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1)

pl.subplot(223)
pl.title("Two informative features, two clusters", fontsize='small')
X2, Y2 = make_classification(n_features=2, n_redundant=0, n_informative=2)
pl.scatter(X2[:, 0], X2[:, 1], marker='o', c=Y2)

pl.subplot(224)
pl.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)
pl.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1)

pl.show()

```

**Total running time of the example:** 0.24 seconds

## 2.1.6 Decomposition

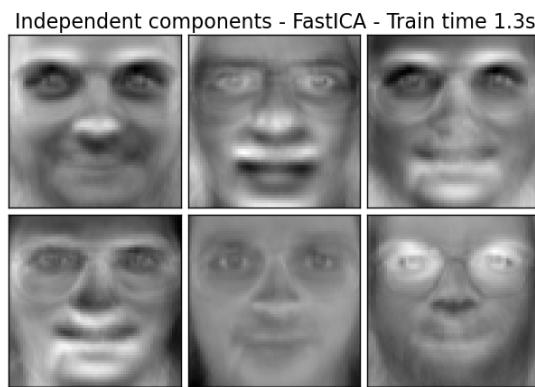
Examples concerning the `sklearn.decomposition` package.



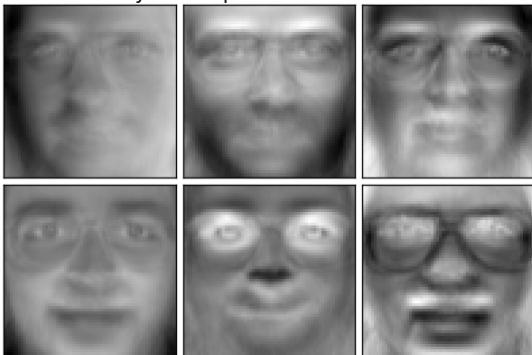
Figure 2.65: *Faces dataset decompositions*

### Faces dataset decompositions

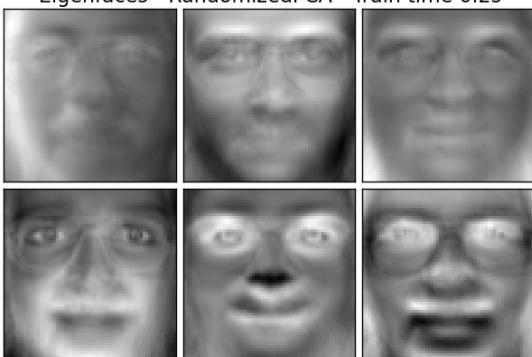
This example applies to `olivetti_faces` different unsupervised matrix decomposition (dimension reduction) methods from the module `sklearn.decomposition` (see the documentation chapter *Decomposing signals in components (matrix factorization problems)*).



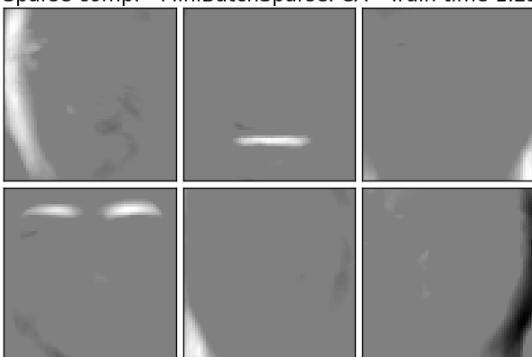
Factor Analysis components - FA - Train time 3.3s



Eigenfaces - RandomizedPCA - Train time 0.2s

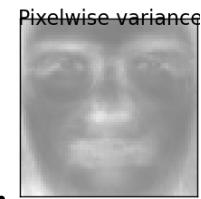


Sparse comp. - MiniBatchSparsePCA - Train time 1.2s



MiniBatchDictionaryLearning - Train time 0.9s

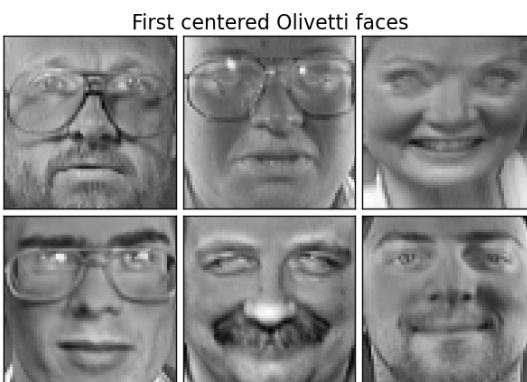




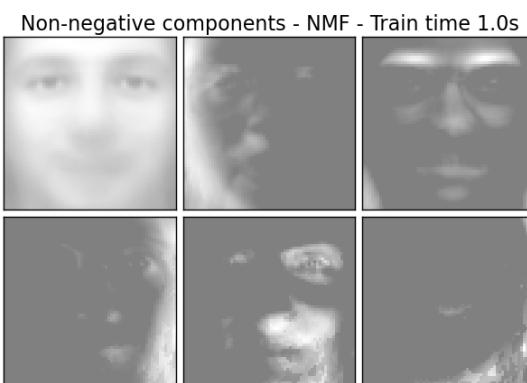
Pixelwise variance



Cluster centers - MiniBatchKMeans - Train time 0.2s



First centered Olivetti faces



Non-negative components - NMF - Train time 1.0s

**Script output:**

```
Dataset consists of 400 faces
Extracting the top 6 Eigenfaces - RandomizedPCA...
done in 0.203s
Extracting the top 6 Non-negative components - NMF...
done in 0.953s
```

```
Extracting the top 6 Independent components - FastICA...
done in 1.290s
Extracting the top 6 Sparse comp. - MiniBatchSparsePCA...
done in 1.172s
Extracting the top 6 MiniBatchDictionaryLearning...
done in 0.946s
Extracting the top 6 Cluster centers - MiniBatchKMeans...
done in 0.201s
Extracting the top 6 Factor Analysis components - FA...
done in 3.274s
```

**Python source code:** [plot\\_faces\\_decomposition.py](#)

```
print __doc__

# Authors: Vlad Niculae, Alexandre Gramfort
# License: BSD

import logging
from time import time

from numpy.random import RandomState
import pylab as pl

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):
    pl.figure(figsize=(2. * n_col, 2.26 * n_row))
    pl.suptitle(title, size=16)
    for i, comp in enumerate(images):
        pl.subplot(n_row, n_col, i + 1)
        vmax = max(comp.max(), -comp.min())
```

```

    pl.imshow(comp.reshape(image_shape), cmap=pl.cm.gray,
              interpolation='nearest',
              vmin=-vmax, vmax=vmax)
    pl.xticks(())
    pl.yticks(())
    pl.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,
                           max_iter=10),
     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])

pl.show()
```

**Total running time of the example:** 10.54 seconds

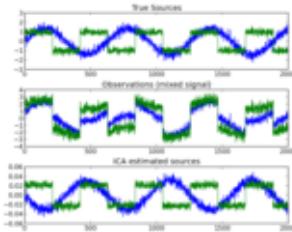
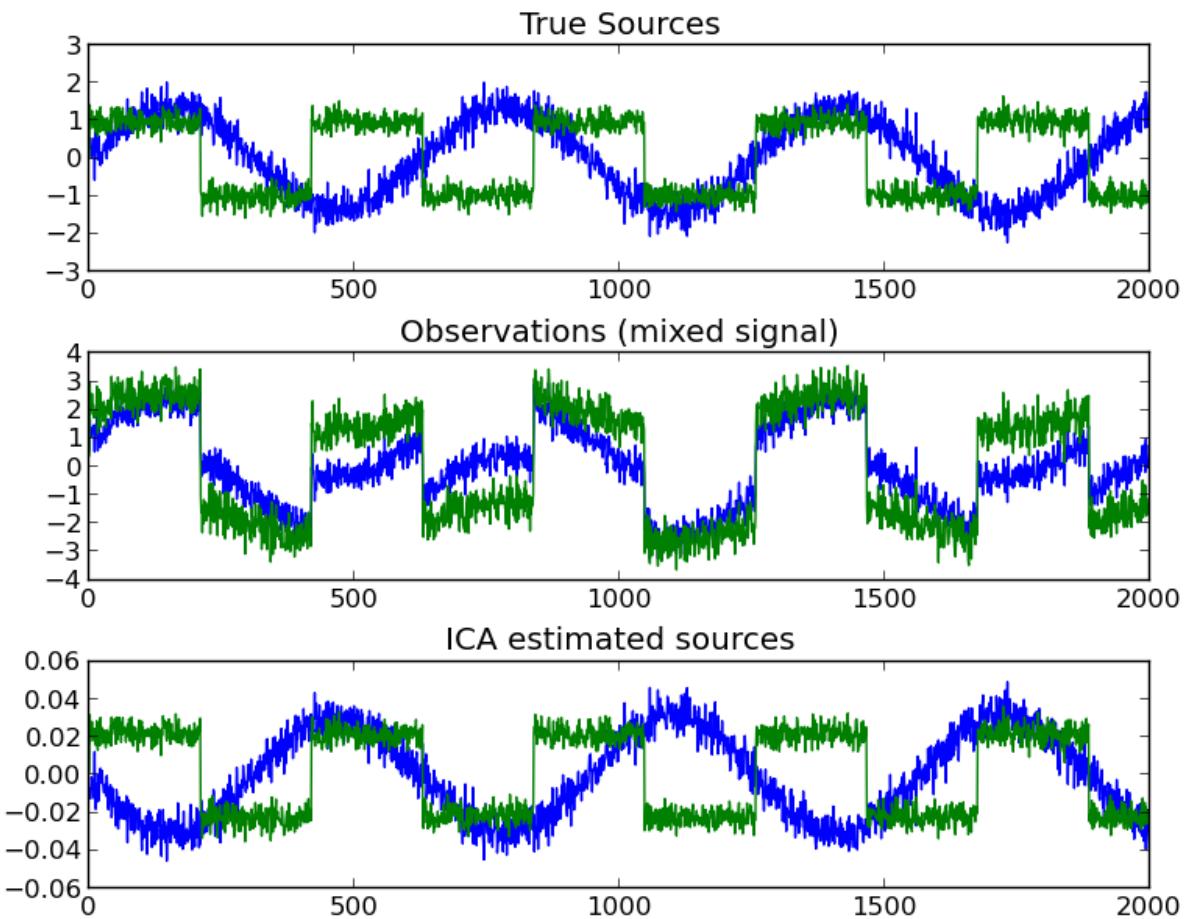


Figure 2.66: *Blind source separation using FastICA*

### Blind source separation using FastICA

*Independent component analysis (ICA)* is used to estimate sources given noisy measurements. Imagine 2 instruments playing simultaneously and 2 microphones recording the mixed signals. ICA is used to recover the sources ie. what is played by each instrument.



**Python source code:** [plot\\_ica\\_blind\\_source\\_separation.py](#)

```
print __doc__

import numpy as np
import pylab as pl
from sklearn.decomposition import FastICA

#####
# Generate sample data
np.random.seed(0)
n_samples = 2000
time = np.linspace(0, 10, n_samples)
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 = FastICA()
S_ = ica.fit(X).transform(X) # Get the estimated sources
A_ = ica.get_mixing_matrix() # Get estimated mixing matrix
```

```
assert np.allclose(X, np.dot(S_, A_.T))

#####
# Plot results
pl.figure()
pl.subplot(3, 1, 1)
pl.plot(S)
pl.title('True Sources')
pl.subplot(3, 1, 2)
pl.plot(X)
pl.title('Observations (mixed signal)')
pl.subplot(3, 1, 3)
pl.plot(S_)
pl.title('ICA estimated sources')
pl.subplots_adjust(0.09, 0.04, 0.94, 0.94, 0.26, 0.36)
pl.show()
```

**Total running time of the example:** 0.33 seconds

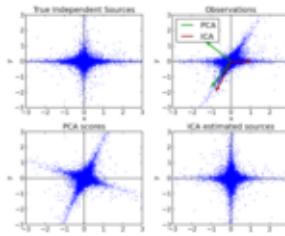


Figure 2.67: *FastICA on 2D point clouds*

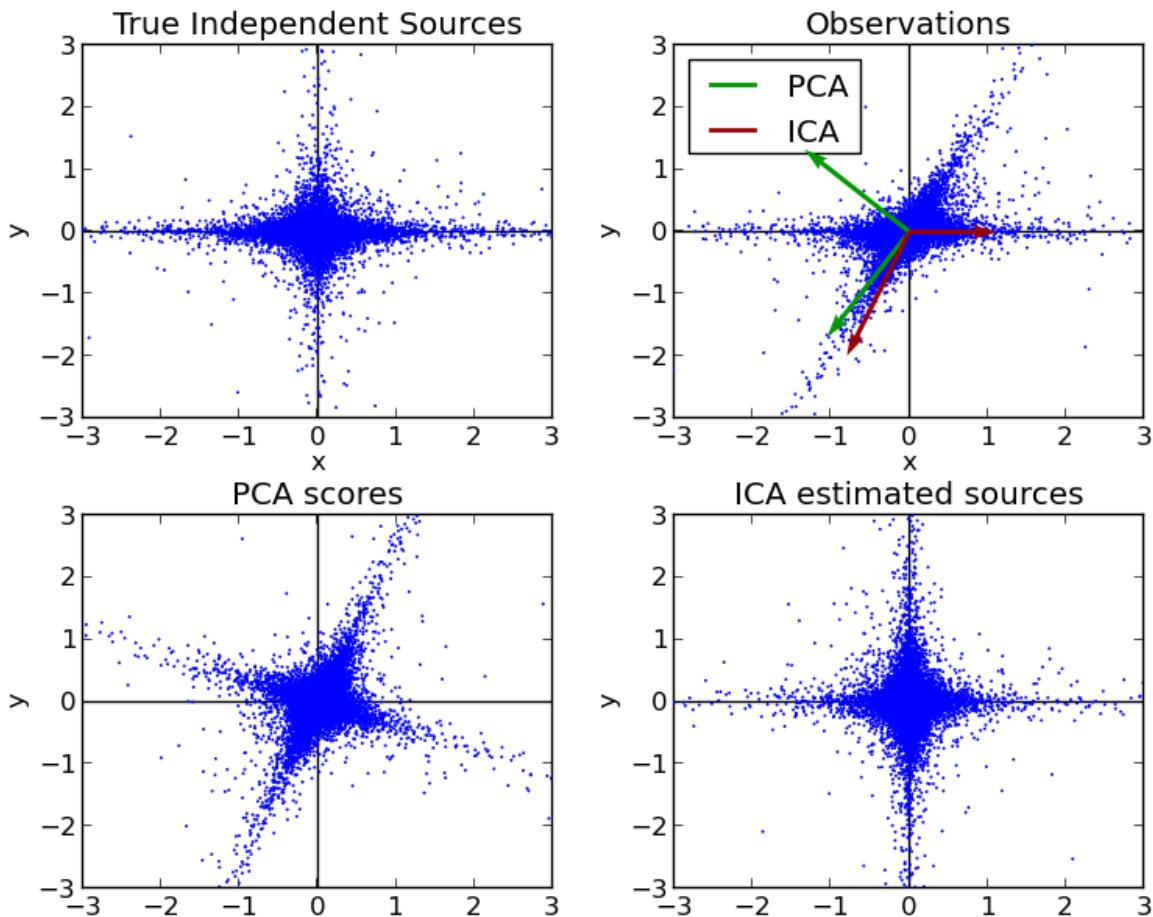
## FastICA on 2D point clouds

Illustrate visually the results of *Independent component analysis (ICA)* vs *Principal component analysis (PCA)* in the feature space.

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 green 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

import numpy as np
import pylab as pl

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):
    pl.scatter(S[:, 0], S[:, 1], s=2, marker='o', linewidths=0, zorder=10)
    if axis_list is not None:
        colors = [(0, 0.6, 0), (0.6, 0, 0)]
        for color, axis in zip(colors, axis_list):
            axis /= axis.std()
            x_axis, y_axis = axis
            # Trick to get legend to work
            pl.plot(0.1 * x_axis, 0.1 * y_axis, linewidth=2, color=color)
            # pl.quiver(x_axis, y_axis, x_axis, y_axis, zorder=11, width=0.01,
            pl.quiver(0, 0, x_axis, y_axis, zorder=11, width=0.01, scale=6,
                      color=color)

    pl.hlines(0, -3, 3)
    pl.vlines(0, -3, 3)
    pl.xlim(-3, 3)
    pl.ylim(-3, 3)
    pl.xlabel('x')
    pl.ylabel('y')

pl.subplot(2, 2, 1)
plot_samples(S / S.std())
pl.title('True Independent Sources')

axis_list = [pca.components_.T, ica.get_mixing_matrix()]
pl.subplot(2, 2, 2)
plot_samples(X / np.std(X), axis_list=axis_list)
pl.legend(['PCA', 'ICA'], loc='upper left')
pl.title('Observations')

pl.subplot(2, 2, 3)
plot_samples(S_pca_ / np.std(S_pca_, axis=0))
pl.title('PCA scores')

pl.subplot(2, 2, 4)
plot_samples(S_ica_ / np.std(S_ica_))
pl.title('ICA estimated sources')

pl.subplots_adjust(0.09, 0.04, 0.94, 0.94, 0.26, 0.26)

pl.show()
```

**Total running time of the example:** 0.79 seconds

## Image denoising using dictionary learning

An example comparing the effect of reconstructing noisy fragments of Lena using online *Dictionary Learning* and various transform methods.

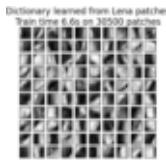


Figure 2.68: *Image denoising using dictionary learning*

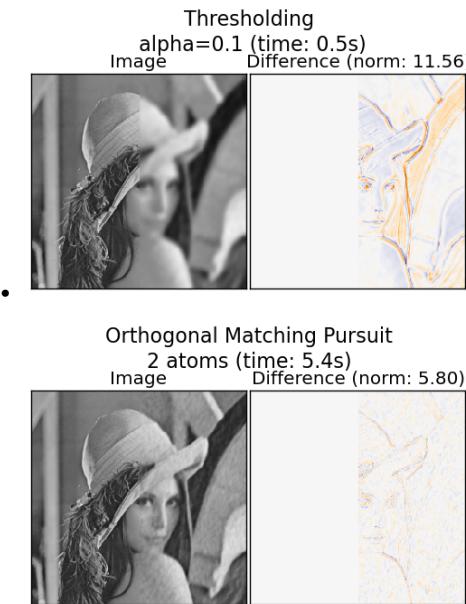
The dictionary is fitted on the distorted left half of the image, and subsequently used to reconstruct the right half. Note that even better performance could be achieved by fitting to an undistorted (i.e. noiseless) image, but here we start from the assumption that it is not available.

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.

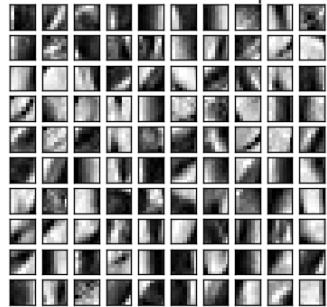
It can be seen from the plots that the results of *Orthogonal Matching Pursuit (OMP)* with two non-zero coefficients is a bit less biased than when keeping only one (the edges look less prominent). It is in addition closer from the ground truth in Frobenius norm.

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 6.6s on 30500 patches



Distorted image



Orthogonal Matching Pursuit

1 atom (time: 3.6s)



Least-angle regression

5 atoms (time: 30.7s)



#### Script output:

```
Distorting image...
Extracting reference patches...
done in 0.05s.
Learning the dictionary...
done in 6.63s.
Extracting noisy patches...
done in 0.01s.
Orthogonal Matching Pursuit
1 atom ...
done in 3.61s.
Orthogonal Matching Pursuit
```

```
2 atoms ...
done in 5.35s.
Least-angle regression
5 atoms ...
done in 30.70s.
Thresholding
alpha=0.1 ...
done in 0.47s.
```

**Python source code:** [plot\\_image\\_denoising.py](#)

```
print __doc__

from time import time

import pylab as pl
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
```

```
pl.figure(figsize=(4.2, 4))
for i, comp in enumerate(V[:100]):
    pl.subplot(10, 10, i + 1)
    pl.imshow(comp.reshape(patch_size), cmap=pl.cm.gray_r,
               interpolation='nearest')
    pl.xticks(())
    pl.yticks(())
pl.suptitle('Dictionary learned from Lena patches\n' +
            'Train time %.1fs on %d patches' % (dt, len(data)),
            fontsize=16)
pl.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"""
    pl.figure(figsize=(5, 3.3))
    pl.subplot(1, 2, 1)
    pl.title('Image')
    pl.imshow(image, vmin=0, vmax=1, cmap=pl.cm.gray, interpolation='nearest')
    pl.xticks(())
    pl.yticks(())
    pl.subplot(1, 2, 2)
    difference = image - reference

    pl.title('Difference (norm: %.2f)' % np.sqrt(np.sum(difference ** 2)))
    pl.imshow(difference, vmin=-0.5, vmax=0.5, cmap=pl.cm.PuOr,
               interpolation='nearest')
    pl.xticks(())
    pl.yticks(())
    pl.suptitle(title, size=16)
    pl.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)

pl.show()

```

**Total running time of the example:** 52.04 seconds

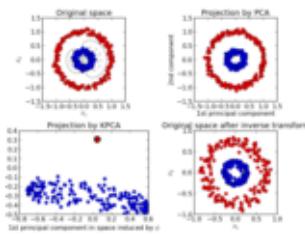
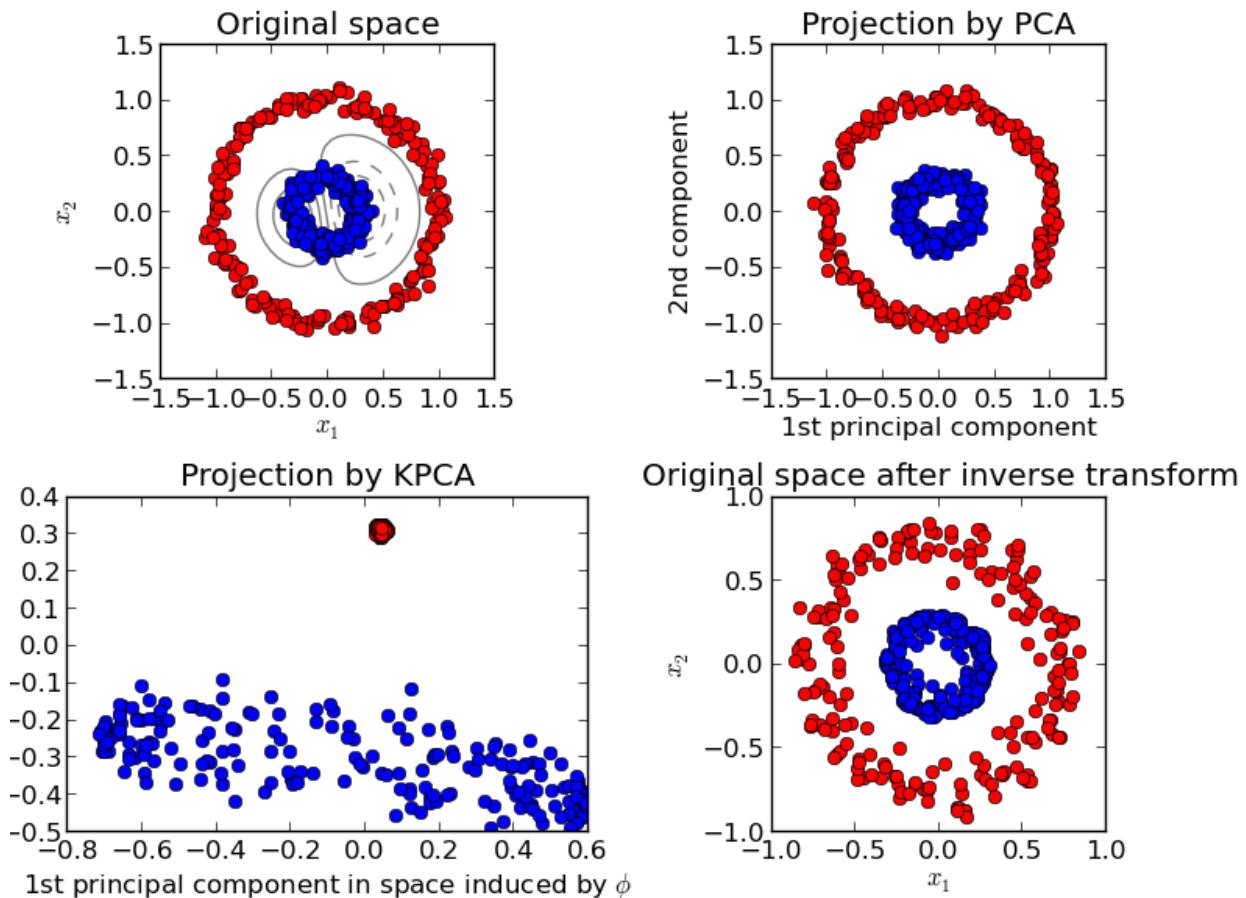


Figure 2.69: *Kernel PCA*

## Kernel PCA

This example shows that Kernel PCA is able to find a projection of the data that makes data linearly separable.



**Python source code:** [plot\\_kernel\\_pca.py](#)

```
print __doc__

# Authors: Mathieu Blondel
#          Andreas Mueller
# License: BSD

import numpy as np
import pylab as pl

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
```

```

pl.figure()
pl.subplot(2, 2, 1, aspect='equal')
pl.title("Original space")
reds = y == 0
blues = y == 1

pl.plot(X[reds, 0], X[reds, 1], "ro")
pl.plot(X[blues, 0], X[blues, 1], "bo")
pl.xlabel("$x_1$")
pl.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)
pl.contour(X1, X2, Z_grid, colors='grey', linewidths=1, origin='lower')

pl.subplot(2, 2, 2, aspect='equal')
pl.plot(X_pca[reds, 0], X_pca[reds, 1], "ro")
pl.plot(X_pca[blues, 0], X_pca[blues, 1], "bo")
pl.title("Projection by PCA")
pl.xlabel("1st principal component")
pl.ylabel("2nd component")

pl.subplot(2, 2, 3, aspect='equal')
pl.plot(X_kpca[reds, 0], X_kpca[reds, 1], "ro")
pl.plot(X_kpca[blues, 0], X_kpca[blues, 1], "bo")
pl.title("Projection by KPCA")
pl.xlabel("1st principal component in space induced by $\phi$")
pl.ylabel("2nd component")

pl.subplot(2, 2, 4, aspect='equal')
pl.plot(X_back[reds, 0], X_back[reds, 1], "ro")
pl.plot(X_back[blues, 0], X_back[blues, 1], "bo")
pl.title("Original space after inverse transform")
pl.xlabel("$x_1$")
pl.ylabel("$x_2$")

pl.subplots_adjust(0.02, 0.10, 0.98, 0.94, 0.04, 0.35)

pl.show()

```

**Total running time of the example:** 0.89 seconds

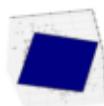
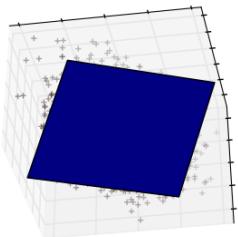
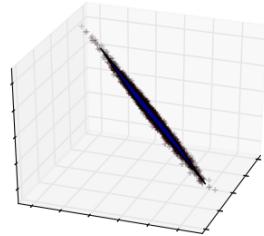


Figure 2.70: Principal Component Analysis

## Principal Component Analysis

These figures aid in illustrating how a the point cloud can be very flat in one direction - which is where PCA would come in to choose a direction that is not flat.



**Python source code:** [plot\\_pca\\_3d.py](#)

```
print __doc__

# Code source: Gael Varoquaux
# Modified for Documentation merge by Jaques Grobler
# License: BSD

import pylab as pl
import numpy as np
from scipy import stats, linalg
from mpl_toolkits.mplot3d import Axes3D

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 = pl.figure(fig_num, figsize=(4, 3))
    pl.clf()
    ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=elev, azim=azim)

    ax.scatter(a[::-10], b[::-10], c[::-10], c=density, marker='+', alpha=.4)
    Y = np.c_[a, b, c]
    U, pca_score, V = linalg.svd(Y, full_matrices=False)
    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)

pl.show()
```

**Total running time of the example:** 0.24 seconds

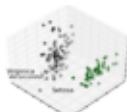
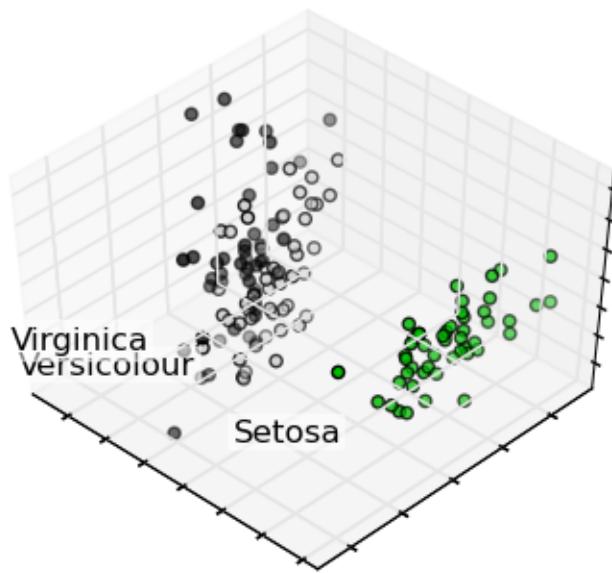


Figure 2.71: PCA example with Iris Data-set

## PCA example with Iris Data-set



**Python source code:** [plot\\_pca\\_iris.py](#)

```
print __doc__\n\n# Code source: Gael Varoquaux\n# License: BSD\n\nimport numpy as np\nimport pylab as pl\nfrom mpl_toolkits.mplot3d import Axes3D\n\n\nfrom sklearn import decomposition\nfrom sklearn import datasets\n\nnp.random.seed(5)\n\ncenters = [[1, 1], [-1, -1], [1, -1]]\niris = datasets.load_iris()\nX = iris.data\ny = iris.target\n\nfig = pl.figure(1, figsize=(4, 3))\npl.clf()\nax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)\n\npl.cla()\npca = decomposition.PCA(n_components=3)\npca.fit(X)\nX = pca.transform(X)\n\nfor name, label in [('Setosa', 0), ('Versicolour', 1), ('Virginica', 2)]:\n    ax.text3D(X[y == label, 0].mean(),\n              X[y == label, 1].mean() + 1.5,\n              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=pl.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([])

pl.show()

```

**Total running time of the example:** 0.16 seconds

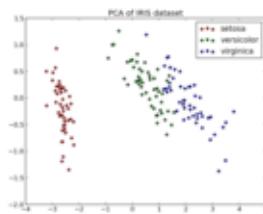


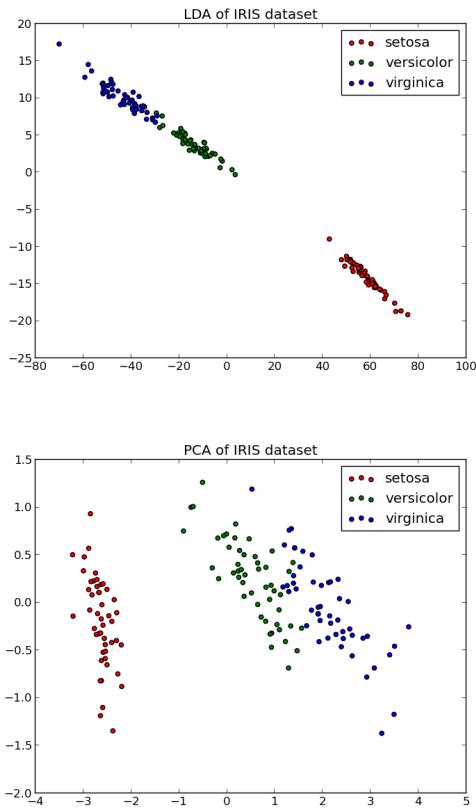
Figure 2.72: Comparison of LDA and PCA 2D projection of Iris dataset

### 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.

**Script output:**

```
explained variance ratio (first two components): [ 0.92461621  0.05301557]
```

**Python source code:** [plot\\_pca\\_vs\\_lda.py](#)

```
print __doc__  
  
import pylab as pl  
  
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):', \  
    pca.explained_variance_ratio_
```

```

pl.figure()
for c, i, target_name in zip("rgb", [0, 1, 2], target_names):
    pl.scatter(X_r[y == i, 0], X_r[y == i, 1], c=c, label=target_name)
pl.legend()
pl.title('PCA of IRIS dataset')

pl.figure()
for c, i, target_name in zip("rgb", [0, 1, 2], target_names):
    pl.scatter(X_r2[y == i, 0], X_r2[y == i, 1], c=c, label=target_name)
pl.legend()
pl.title('LDA of IRIS dataset')

pl.show()

```

**Total running time of the example:** 0.18 seconds

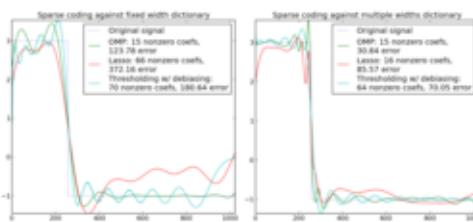
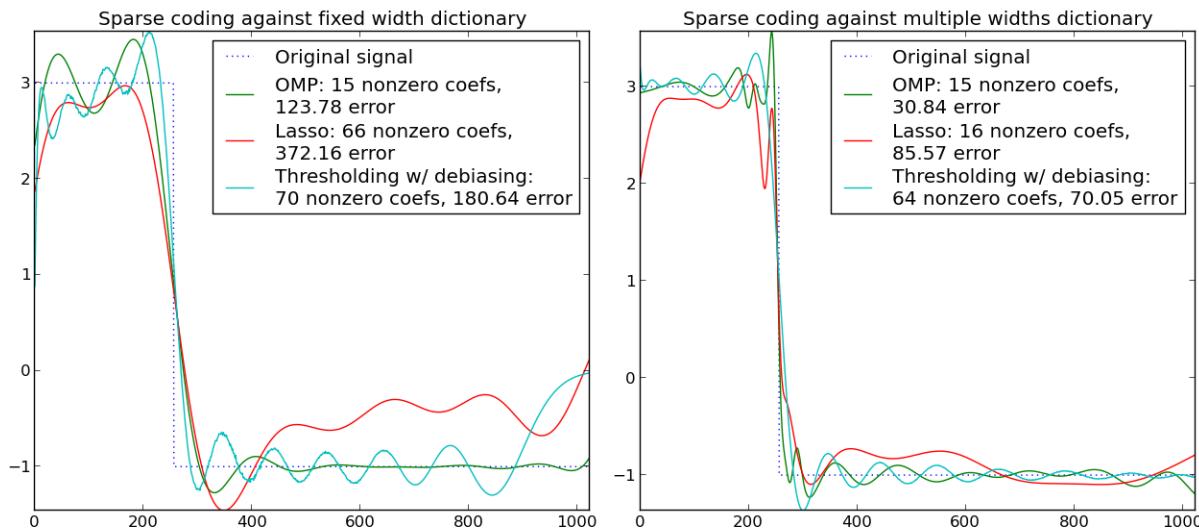


Figure 2.73: *Sparse coding with a precomputed dictionary*

## 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.pyplot 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_nonzero_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, %.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: %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()

```

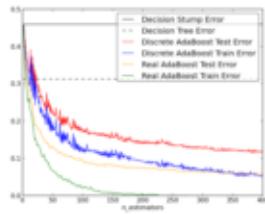
**Total running time of the example:** 0.51 seconds

## 2.1.7 Ensemble methods

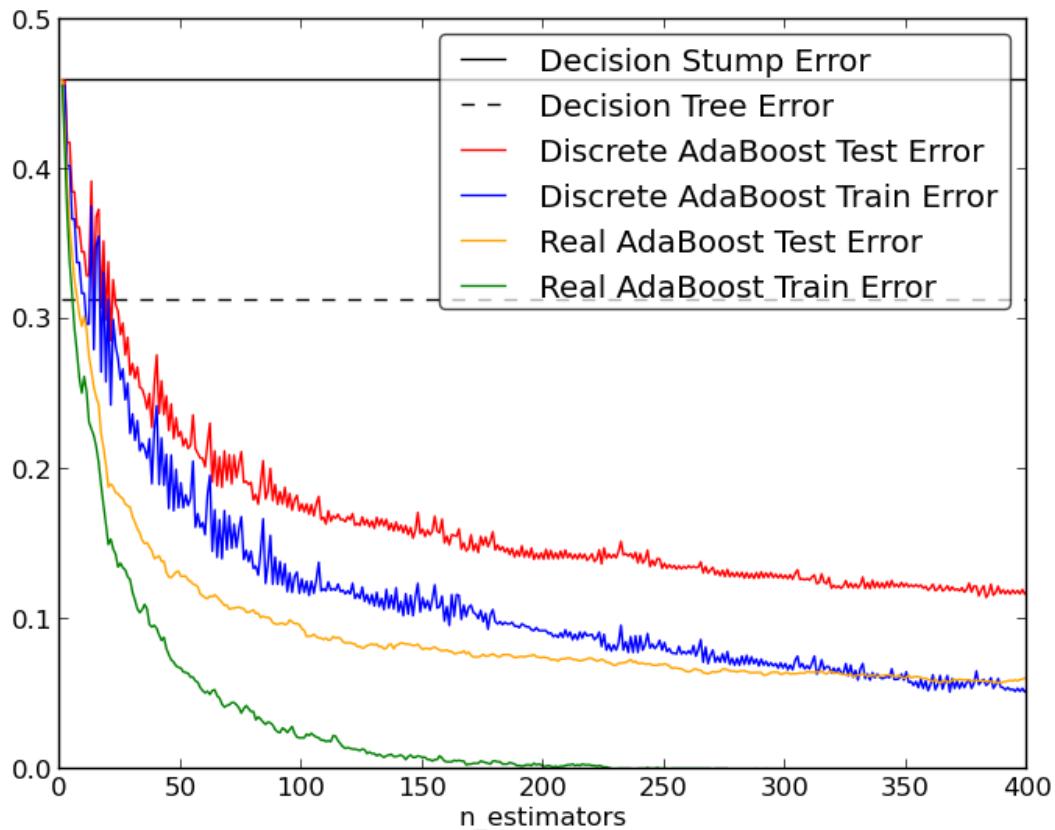
Examples concerning the `sklearn.ensemble` package.

### 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.

Figure 2.74: *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  
  
import numpy as np  
from sklearn import datasets
```

```

from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import zero_one_loss
from sklearn.ensemble import AdaBoostClassifier
import pylab as plt

n_estimators = 400
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_xlim((0.0, 0.5))
ax.set_xlabel('n_estimators')

leg = ax.legend(loc='upper right', fancybox=True)
leg.get_frame().set_alpha(0.7)

plt.show()
```

**Total running time of the example:** 4.84 seconds

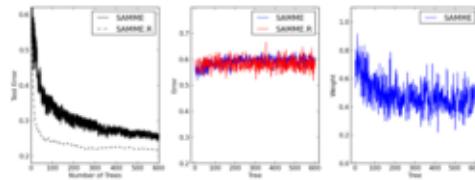
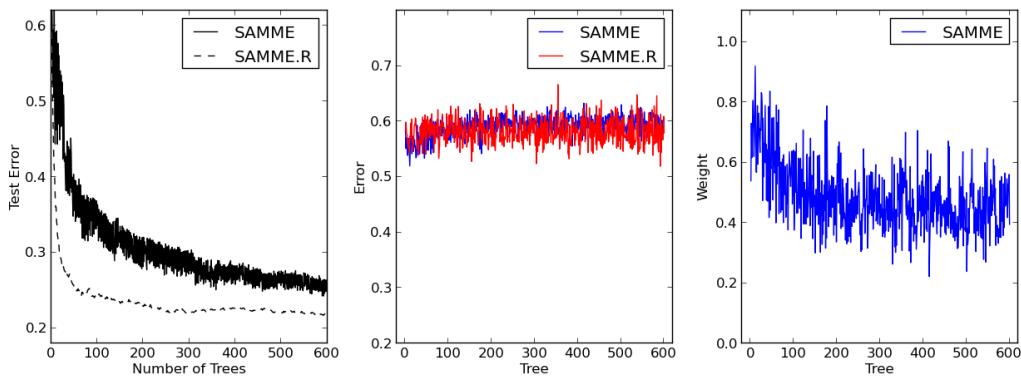


Figure 2.75: Multi-class AdaBoosted Decision Trees

### 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  $\chi^2$  distribution).

The performance of the SAMME and SAMME.R [1] algorithms are compared. 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.



**Python source code:** [plot\\_adaboost\\_multiclass.py](#)

```
print __doc__

# Author: Noel Dawe <noel.dawe@gmail.com>
#
# License: BSD

from itertools import izip

import pylab as pl

from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets.samples_generator import make_gaussian_quantiles
from sklearn.metrics import accuracy_score

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 izip(
```

```
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 = xrange(1, len(bdt_discrete) + 1)

pl.figure(figsize=(15, 5))

pl.subplot(131)
pl.plot(n_trees, discrete_test_errors, c='black', label='SAMME')
pl.plot(n_trees, real_test_errors, c='black',
        linestyle='dashed', label='SAMME.R')
pl.legend()
pl.ylim(0.18, 0.62)
pl.ylabel('Test Error')
pl.xlabel('Number of Trees')

pl.subplot(132)
pl.plot(n_trees, bdt_discrete.errors_, "b", label='SAMME')
pl.plot(n_trees, bdt_real.errors_, "r", label='SAMME.R')
pl.legend()
pl.ylabel('Error')
pl.xlabel('Tree')
pl.ylim((.2, max(bdt_real.errors_.max(), bdt_discrete.errors_.max()) * 1.2))
pl.xlim((-20, len(bdt_discrete) + 20))

pl.subplot(133)
pl.plot(n_trees, bdt_discrete.estimator_weights_, "b", label='SAMME')
pl.legend()
pl.ylabel('Weight')
pl.xlabel('Tree')
pl.ylim((0, bdt_discrete.estimator_weights_.max() * 1.2))
pl.xlim((-20, len(bdt_discrete) + 20))

# prevent overlapping y-axis labels
pl.subplots_adjust(wspace=0.25)
pl.show()
```

**Total running time of the example:** 11.21 seconds

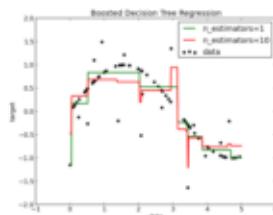
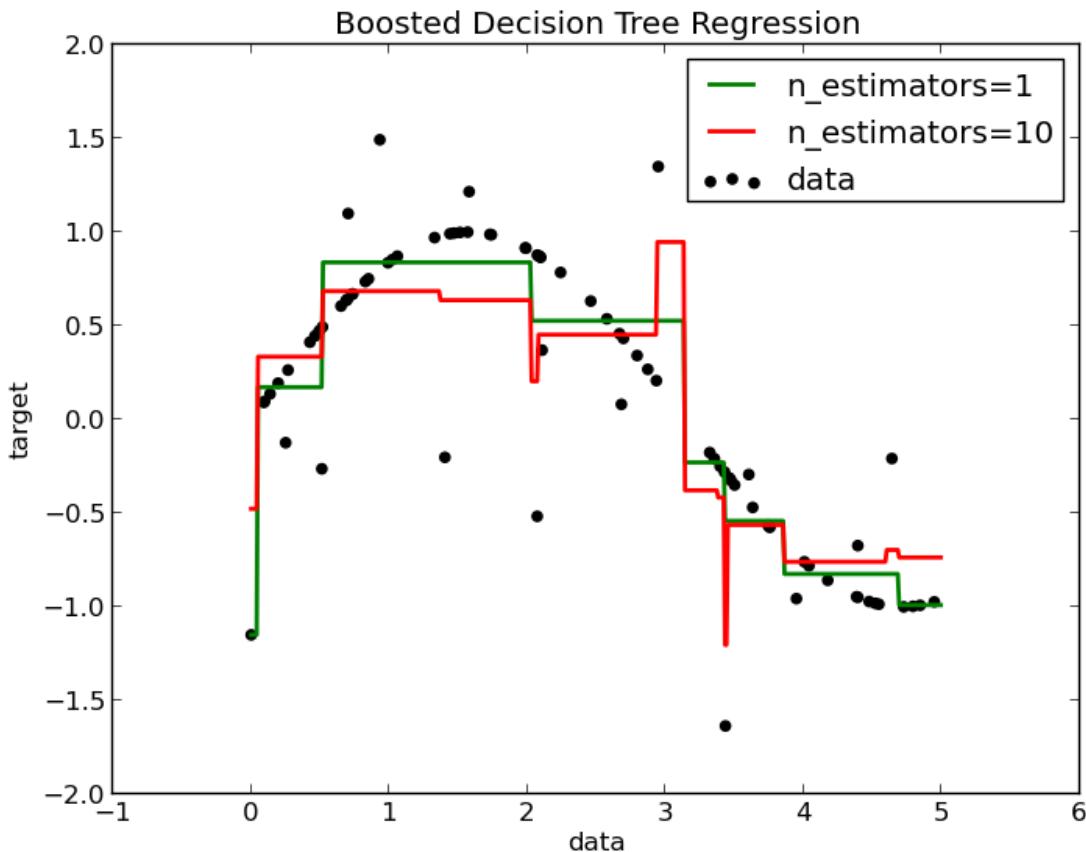


Figure 2.76: Boosted Decision Tree Regression

## Boosted Decision Tree Regression

1D regression with boosted *decision trees*: the decision tree 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 number of boosts (controlled by the `n_estimators` parameter) is set too high, the ensemble learns too fine details of the training data and learn from the noise, i.e. they overfit.



**Python source code:** [plot\\_adaboost\\_regression.py](#)

```
print __doc__

import numpy as np

# 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
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor

clf_1 = AdaBoostRegressor(
    DecisionTreeRegressor(max_depth=3),
```

```
n_estimators=1,
learning_rate=1.)

clf_2 = AdaBoostRegressor(
    DecisionTreeRegressor(max_depth=3),
    n_estimators=100,
    learning_rate=1.)

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
import pylab as pl

pl.figure()
pl.scatter(X, y, c="k", label="data")
pl.plot(X_test, y_1, c="g", label="n_estimators=1", linewidth=2)
pl.plot(X_test, y_2, c="r", label="n_estimators=10", linewidth=2)
pl.xlabel("data")
pl.ylabel("target")
pl.title("Boosted Decision Tree Regression")
pl.legend()
pl.show()
```

Total running time of the example: 0.16 seconds

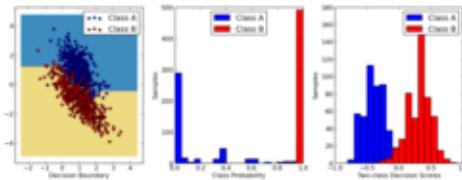
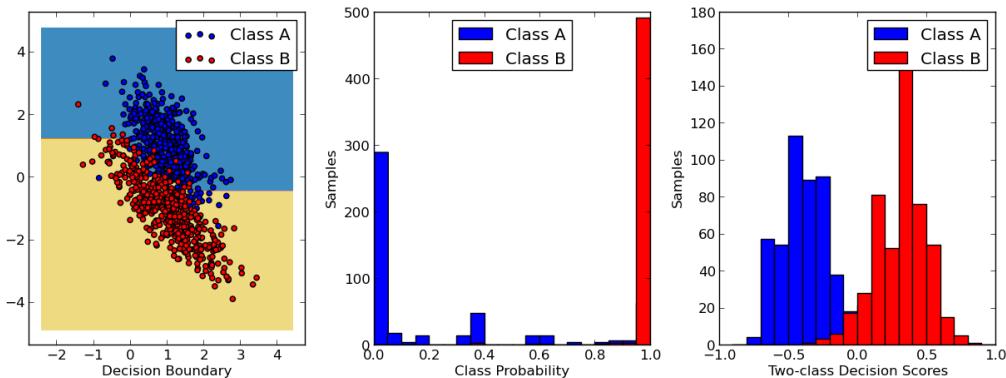


Figure 2.77: Two-class AdaBoost

## Two-class AdaBoost

This example fits an AdaBoosted decision stump on a classification dataset and plots the decision boundary, class probabilities, and two-class decision score.



**Python source code:** [plot\\_adaboost\\_twoclass.py](#)

```
print __doc__

import pylab as pl
import numpy as np

from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_classification

X, y = make_classification(n_samples=1000,
                           n_features=2,
                           n_classes=2,
                           n_clusters_per_class=1,
                           n_informative=2,
                           n_redundant=0,
                           random_state=1)

bdt = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1),
    algorithm="SAMME",
    n_estimators=50)

bdt.fit(X, y)

plot_colors = "br"
plot_step = 0.02
class_names = "AB"

pl.figure(figsize=(15, 5))

# Plot the decision boundaries
pl.subplot(131)
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 = pl.contourf(xx, yy, Z, cmap=pl.cm.Paired)
pl.axis("tight")
```

```
# Plot the training points
for i, n, c in zip(xrange(2), class_names, plot_colors):
    idx = np.where(y == i)
    pl.scatter(X[idx, 0], X[idx, 1],
               c=c, cmap=pl.cm.Paired,
               label="Class %s" % n)
pl.axis("tight")
pl.legend(loc='upper right')
pl.xlabel("Decision Boundary")

# Plot the class probabilities
class_proba = bdt.predict_proba(X)[:, -1]
pl.subplot(132)
for i, n, c in zip(xrange(2), class_names, plot_colors):
    pl.hist(class_proba[y == i],
            bins=20,
            range=(0, 1),
            facecolor=c,
            label='Class %s' % n)
pl.legend(loc='upper center')
pl.ylabel('Samples')
pl.xlabel('Class Probability')

# Plot the two-class decision scores
twoclass_output = bdt.decision_function(X)
pl.subplot(133)
for i, n, c in zip(xrange(2), class_names, plot_colors):
    pl.hist(twoclass_output[y == i],
            bins=20,
            range=(-1, 1),
            facecolor=c,
            label='Class %s' % n)
pl.legend(loc='upper right')
pl.ylabel('Samples')
pl.xlabel('Two-class Decision Scores')

pl.subplots_adjust(wspace=0.25)
pl.show()
```

**Total running time of the example:** 0.76 seconds

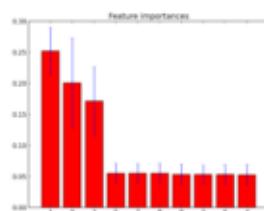
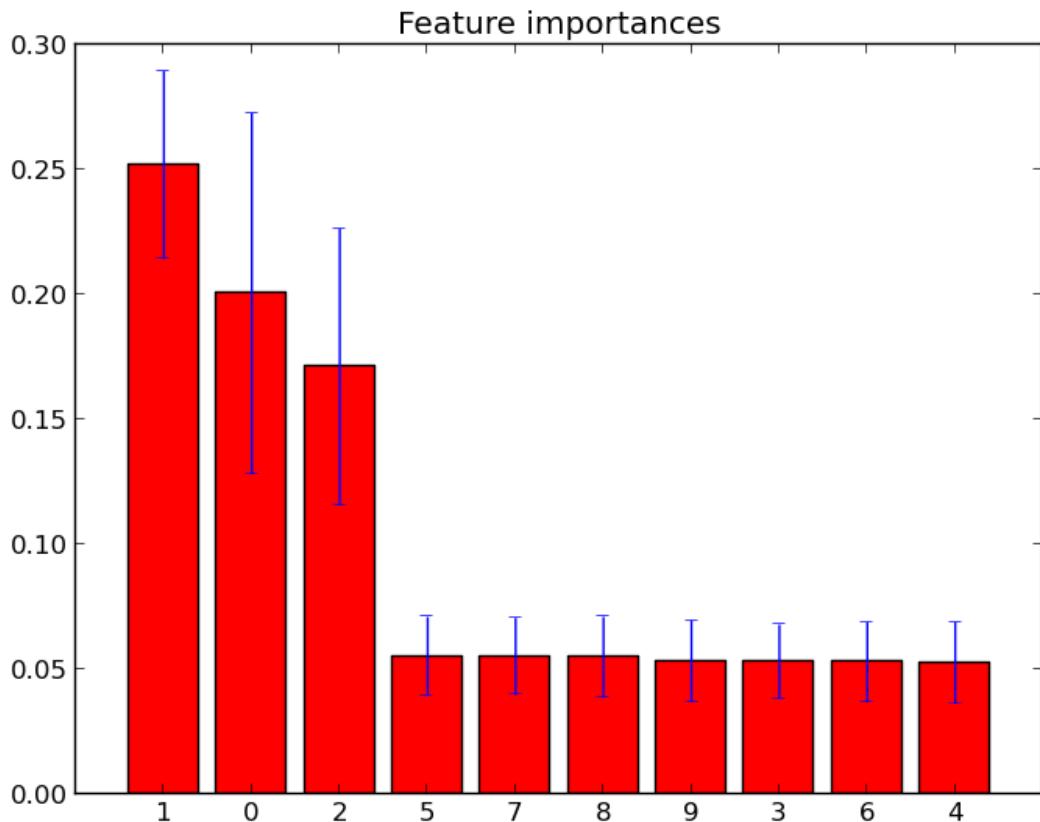


Figure 2.78: *Feature importances with forests of trees*

## Feature importances with forests of trees

This examples shows the use of forests of trees to evaluate the importance of features on an artifical 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 1 (0.251764)
2. feature 0 (0.200486)
3. feature 2 (0.171010)
4. feature 5 (0.055133)
5. feature 7 (0.055120)
6. feature 8 (0.054825)
7. feature 9 (0.053174)
8. feature 3 (0.052988)
9. feature 6 (0.052828)
10. feature 4 (0.052671)
```

**Python source code:** [plot\\_forest\\_importances.py](#)

```
print __doc__

import numpy as np

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,
                             compute_importances=True,
                             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 xrange(10):
    print "%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]])

# Plot the feature importances of the forest
import pylab as pl
pl.figure()
pl.title("Feature importances")
pl.bar(xrange(10), importances[indices],
       color="r", yerr=std[indices], align="center")
pl.xticks(xrange(10), indices)
pl.xlim([-1, 10])
pl.show()
```

**Total running time of the example:** 2.64 seconds

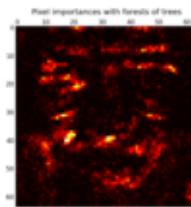
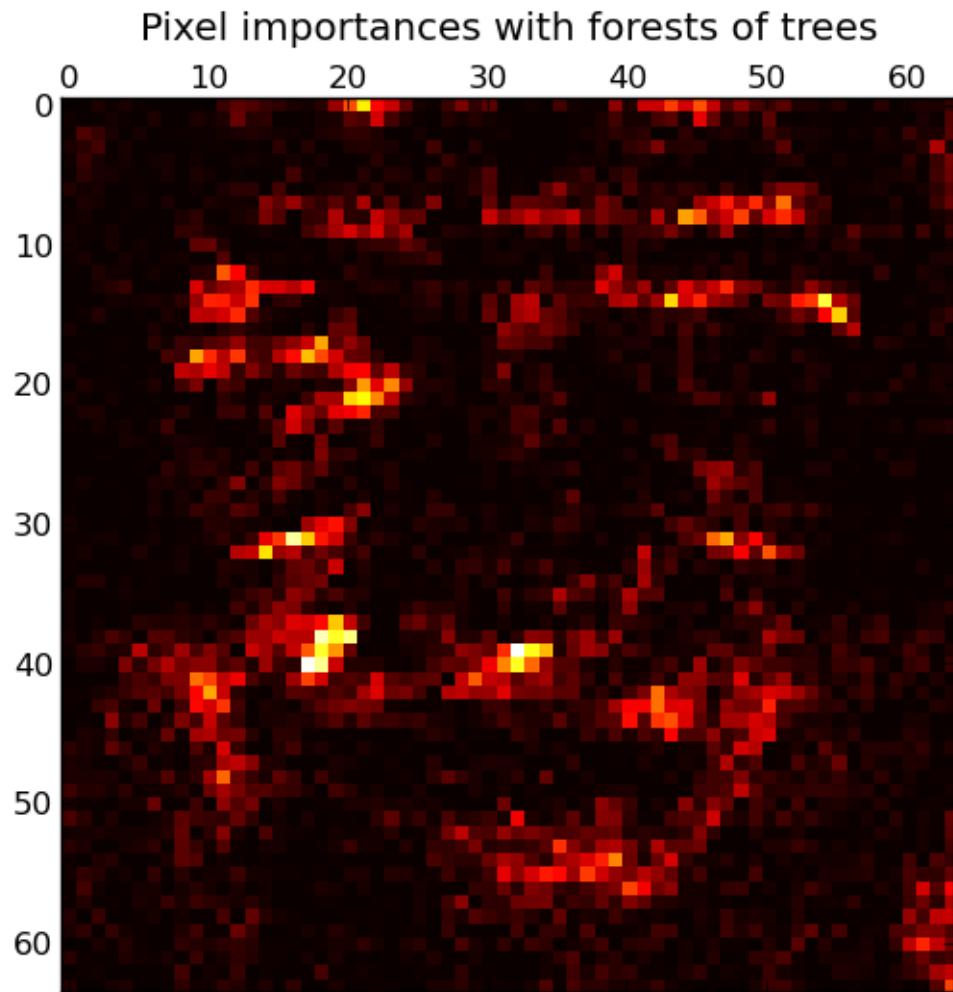


Figure 2.79: Pixel importances with a parallel forest of trees

### 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 9.917s
```

**Python source code:** [plot\\_forest\\_importances\\_faces.py](#)

```
print __doc__

from time import time
import pylab as pl

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,
                              compute_importances=True,
                              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
pl.matshow(importances, cmap=pl.cm.hot)
pl.title("Pixel importances with forests of trees")
pl.show()
```

**Total running time of the example:** 10.05 seconds

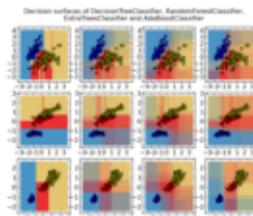


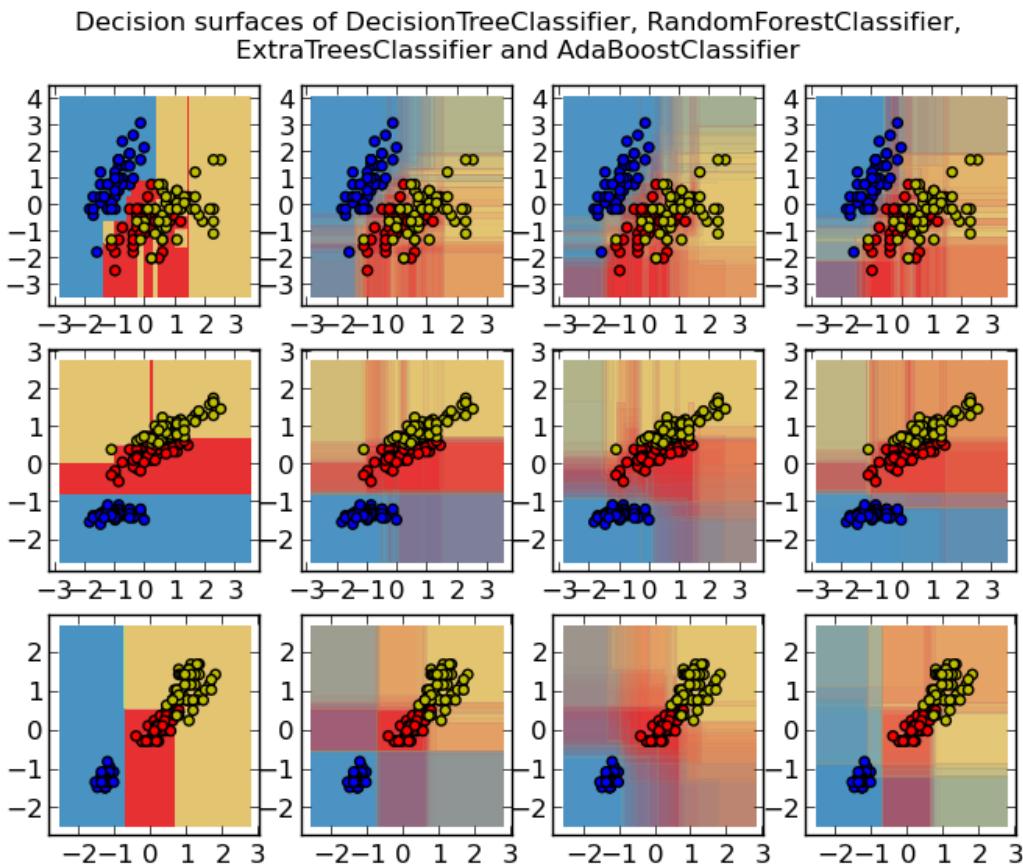
Figure 2.80: Plot the decision surfaces of ensembles of trees on the iris dataset

### 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.



**Python source code:** [plot\\_forest\\_iris.py](#)

```

print __doc__

import numpy as np
import pylab as pl

from sklearn import clone
from sklearn.datasets import load_iris
from sklearn.ensemble import (RandomForestClassifier, ExtraTreesClassifier,
                             AdaBoostClassifier)
from sklearn.tree import DecisionTreeClassifier

# Parameters
n_classes = 3
n_estimators = 30
plot_colors = "bry"
plot_step = 0.02

# Load data
iris = load_iris()

plot_idx = 1

for pair in ([0, 1], [0, 2], [2, 3]):
    for model in (DecisionTreeClassifier(),

```

```
RandomForestClassifier(n_estimators=n_estimators),
ExtraTreesClassifier(n_estimators=n_estimators),
AdaBoostClassifier(DecisionTreeClassifier(),
                   n_estimators=n_estimators)):

# 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 = clone(model)
clf = model.fit(X, y)

# Plot the decision boundary
pl.subplot(3, 4, plot_idx)

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))

if isinstance(model, DecisionTreeClassifier):
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    cs = pl.contourf(xx, yy, Z, cmap=pl.cm.Paired)
else:
    for tree in model.estimators_:
        Z = tree.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)
        cs = pl.contourf(xx, yy, Z, alpha=0.1, cmap=pl.cm.Paired)

pl.axis("tight")

# Plot the training points
for i, c in zip(xrange(n_classes), plot_colors):
    idx = np.where(y == i)
    pl.scatter(X[idx, 0], X[idx, 1], c=c, label=iris.target_names[i],
               cmap=pl.cm.Paired)

pl.axis("tight")

plot_idx += 1

pl.suptitle("Decision surfaces of DecisionTreeClassifier, "
            "RandomForestClassifier,\nExtraTreesClassifier"
            " and AdaBoostClassifier")
pl.show()
```

**Total running time of the example:** 6.88 seconds



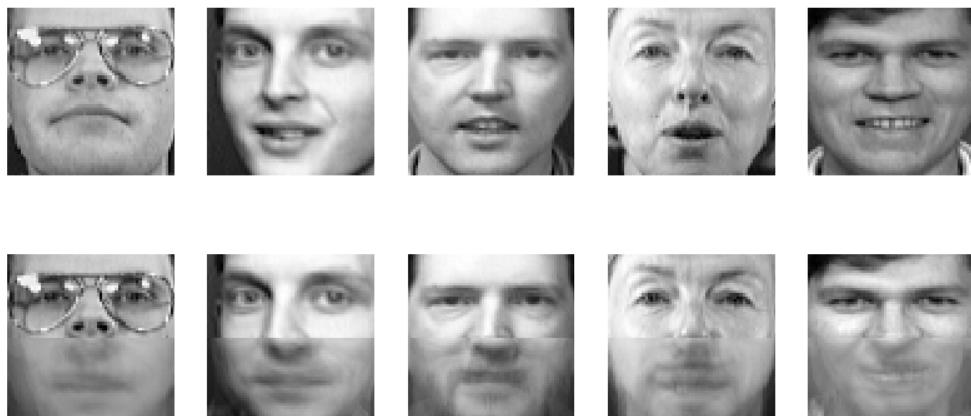
Figure 2.81: *Face completion with multi-output forests*

### Face completion with multi-output forests

This example shows the use of multi-output forests to complete images. The goal is to predict the lower half of a face given its upper half.

The first row of images shows true faces. The second half illustrates how the forest completes the lower half of those faces.

Face completion with multi-output forests



**Python source code:** [plot\\_forest\\_multioutput.py](#)

```
print __doc__

import numpy as np
import pylab as pl

from sklearn.datasets import fetch_olivetti_faces
from sklearn.ensemble import ExtraTreesRegressor

# 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
```

```
n_pixels = data.shape[1]

X_train = train[:, :int(0.5 * n_pixels)] # Upper half of the faces
Y_train = train[:, int(0.5 * n_pixels):] # Lower half of the faces
X_test = test[:, :int(0.5 * n_pixels)]
Y_test = test[:, int(0.5 * n_pixels):]

# Build a multi-output forest
forest = ExtraTreesRegressor(n_estimators=10,
                             max_features=32,
                             random_state=0)

forest.fit(X_train, Y_train)
Y_test_predict = forest.predict(X_test)

# Plot the completed faces
n_faces = 5
image_shape = (64, 64)

pl.figure(figsize=(2. * n_faces, 2.26 * 2))
pl.suptitle("Face completion with multi-output forests", size=16)

for i in xrange(1, 1 + n_faces):
    face_id = np.random.randint(X_test.shape[0])

    true_face = np.hstack((X_test[face_id], Y_test[face_id]))
    completed_face = np.hstack((X_test[face_id], Y_test_predict[face_id]))

    pl.subplot(2, n_faces, i)
    pl.axis("off")
    pl.imshow(true_face.reshape(image_shape),
              cmap=pl.cm.gray,
              interpolation="nearest")

    pl.subplot(2, n_faces, n_faces + i)
    pl.axis("off")
    pl.imshow(completed_face.reshape(image_shape),
              cmap=pl.cm.gray,
              interpolation="nearest")

pl.show()
```

**Total running time of the example:** 13.64 seconds

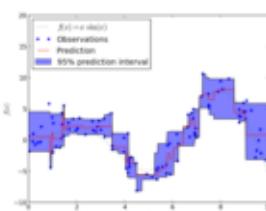
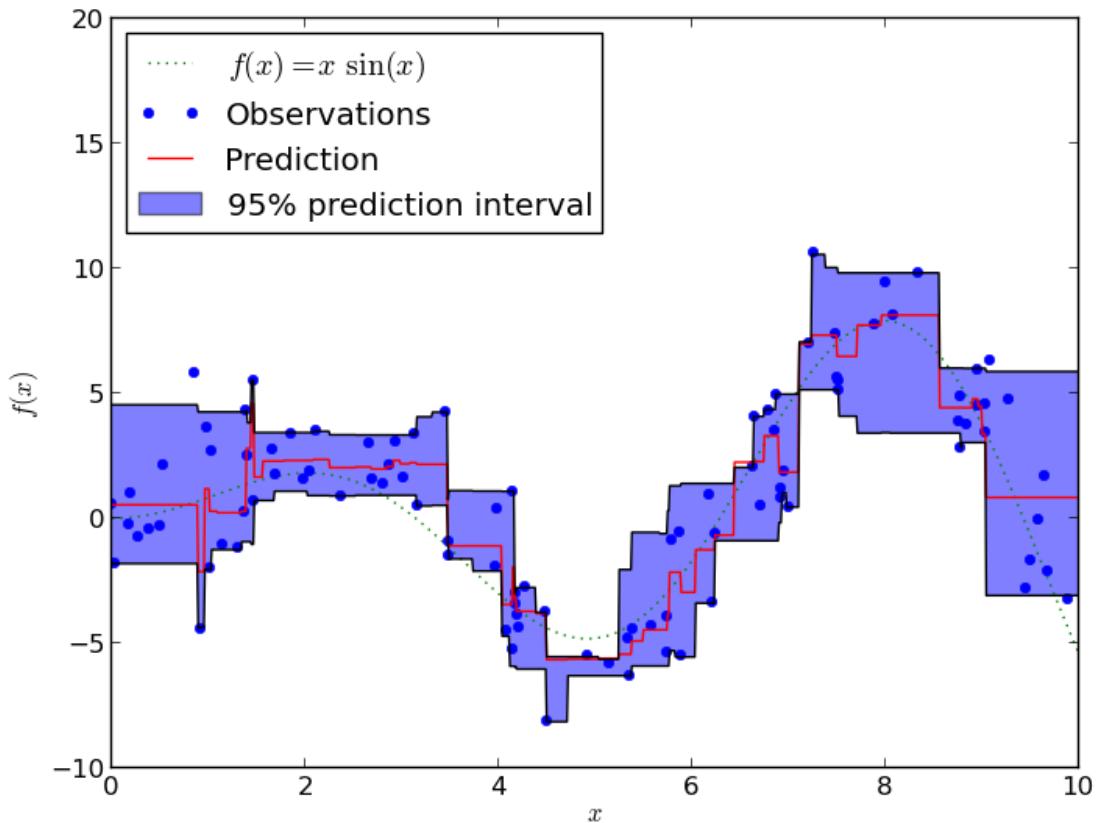


Figure 2.82: *Prediction Intervals for Gradient Boosting Regression*

## 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 pylab as pl
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 95% confidence interval based on
# the MSE
fig = pl.figure()
pl.plot(xx, f(xx), 'g:', label=u'$f(x) = x \cdot \sin(x)$')
pl.plot(X, y, 'b.', markersize=10, label=u'Observations')
pl.plot(xx, y_pred, 'r-', label=u'Prediction')
pl.plot(xx, y_upper, 'k-')
pl.plot(xx, y_lower, 'k-')
pl.fill(np.concatenate([xx, xx[::-1]]),
        np.concatenate([y_upper, y_lower[::-1]]),
        alpha=.5, fc='b', ec='None', label='95% prediction 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:** 0.57 seconds

## 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.

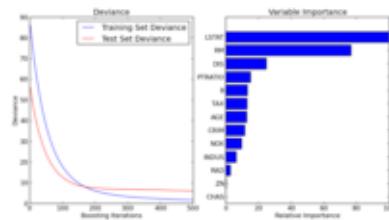
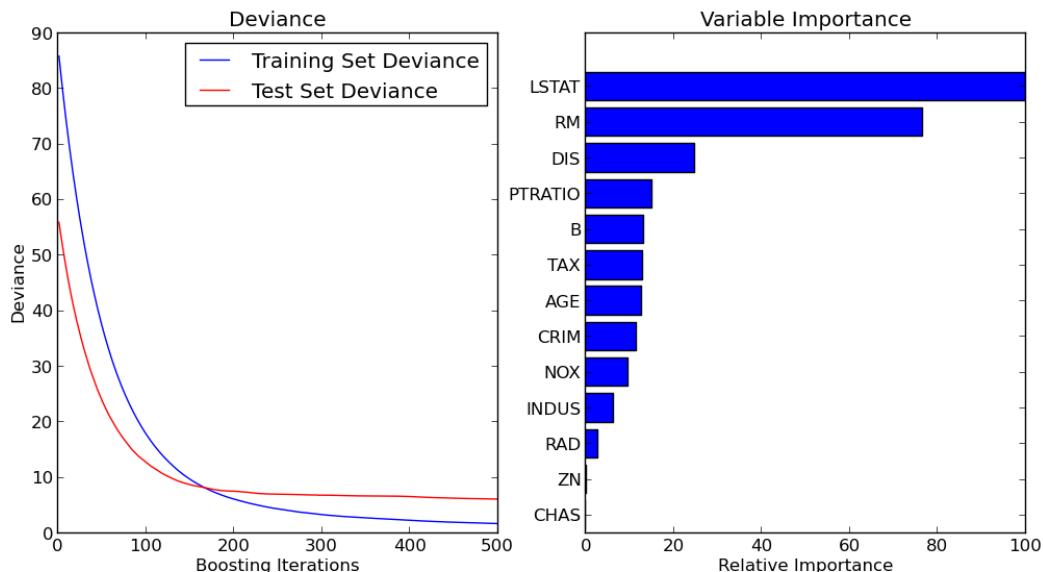


Figure 2.83: Gradient Boosting regression

**Script output:**

MSE: 6.2071

**Python source code:** [plot\\_gradient\\_boosting\\_regression.py](#)

```

print __doc__

# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#
# License: BSD

import numpy as np
import pylab as pl
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)

pl.figure(figsize=(12, 6))
pl.subplot(1, 2, 1)
pl.title('Deviance')
pl.plot(np.arange(params['n_estimators']) + 1, clf.train_score_, 'b-',
        label='Training Set Deviance')
pl.plot(np.arange(params['n_estimators']) + 1, test_score, 'r-',
        label='Test Set Deviance')
pl.legend(loc='upper right')
pl.xlabel('Boosting Iterations')
pl.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
pl.subplot(1, 2, 2)
pl.barh(pos, feature_importance[sorted_idx], align='center')
pl.yticks(pos, boston.feature_names[sorted_idx])
pl.xlabel('Relative Importance')
pl.title('Variable Importance')
pl.show()
```

**Total running time of the example:** 0.79 seconds

## 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

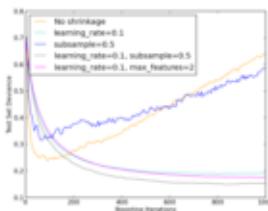
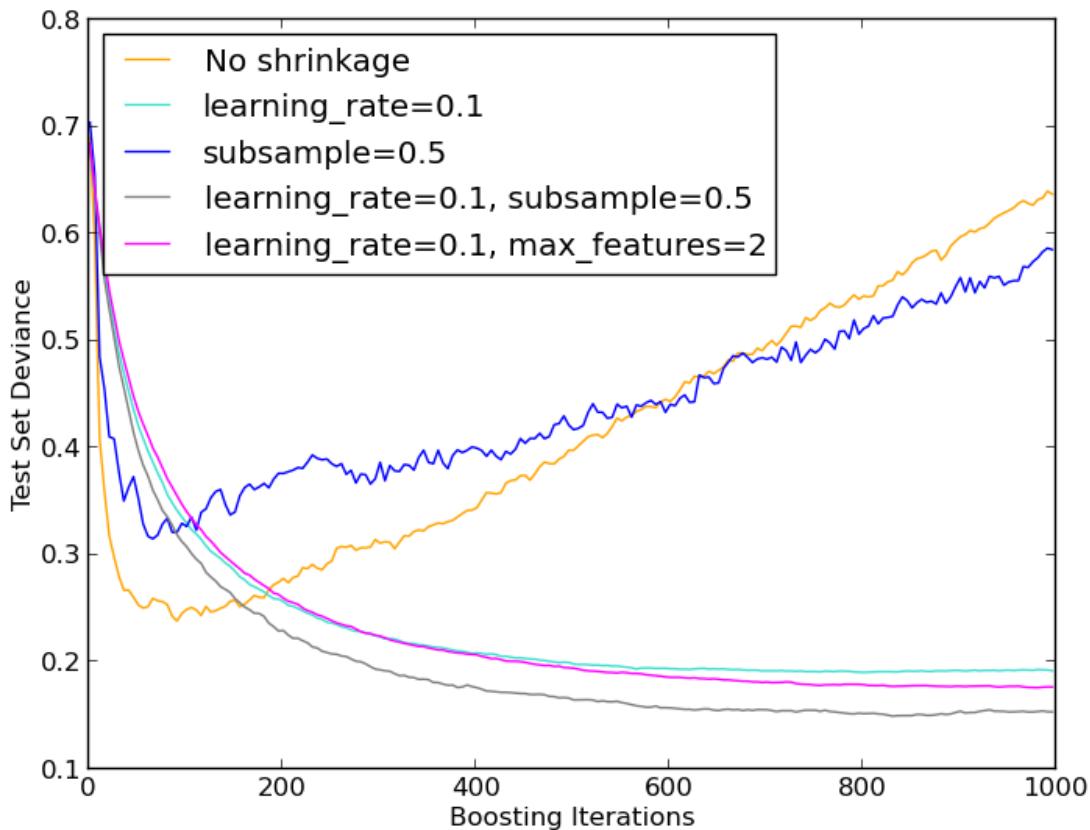


Figure 2.84: Gradient Boosting regularization

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).



**Python source code:** [plot\\_gradient\\_boosting\\_regularization.py](#)

```
print __doc__

# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#
# License: BSD

import numpy as np
import pylab as pl
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)

X_train, X_test = X[:2000], X[2000:]
y_train, y_test = y[:2000], y[2000:]

original_params = {'n_estimators': 1000, 'max_depth': 2, 'random_state': 1,
                   'min_samples_split': 5}

pl.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)):
        test_deviance[i] = clf.loss_(y_test, y_pred)

    pl.plot((np.arange(test_deviance.shape[0]) + 1)[::5], test_deviance[::5],
            '--', color=color, label=label)

pl.legend(loc='upper left')
pl.xlabel('Boosting Iterations')
pl.ylabel('Test Set Deviance')

pl.show()
```

**Total running time of the example:** 11.90 seconds

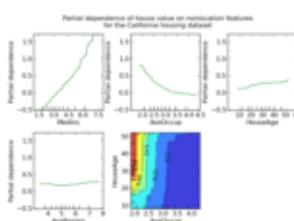


Figure 2.85: Partial Dependence Plots

## Partial Dependence Plots

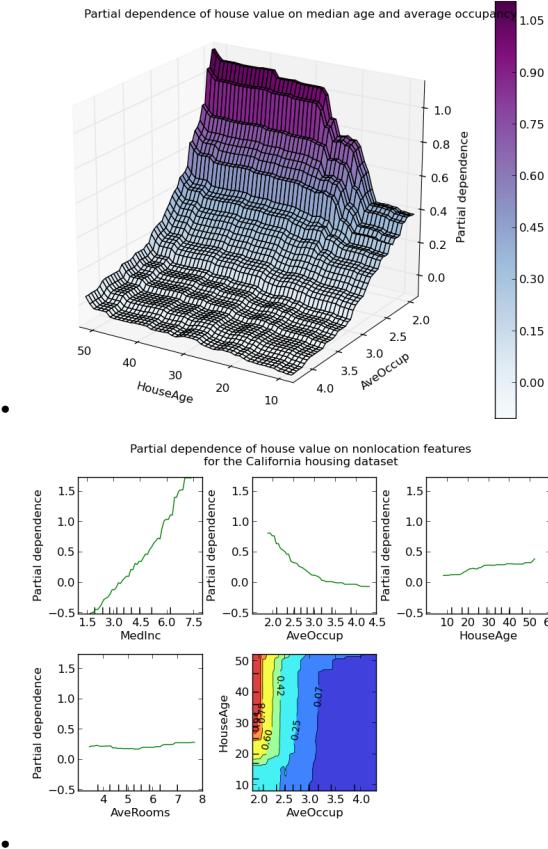
Partial dependence plots show the dependence between the target function <sup>1</sup> and a set of ‘target’ features, marginalizing over the values of all other features (the complement features). Due to the limits of human perception the size of the target feature set must be small (usually, one or 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 house price is nearly independent of the house age, whereas for values less than two there is a strong dependence on age.



### Script output:

<sup>1</sup> For classification you can think of it as the regression score before the link function.

---

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 pylab as pl

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()

# 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')
pl.subplots_adjust(top=0.9) # tight_layout causes overlap with suptitle

print('_' * 80)
print('Custom 3d plot via ``partial_dependence``')
print
fig = pl.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=pl.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)
pl.colorbar(surf)
pl.suptitle('Partial dependence of house value on median age and '
            'average occupancy')
pl.subplots_adjust(top=0.9)

pl.show()

```

**Total running time of the example:** 4.52 seconds

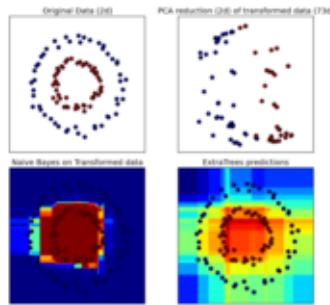


Figure 2.86: *Hashing feature transformation using Totally Random Trees*

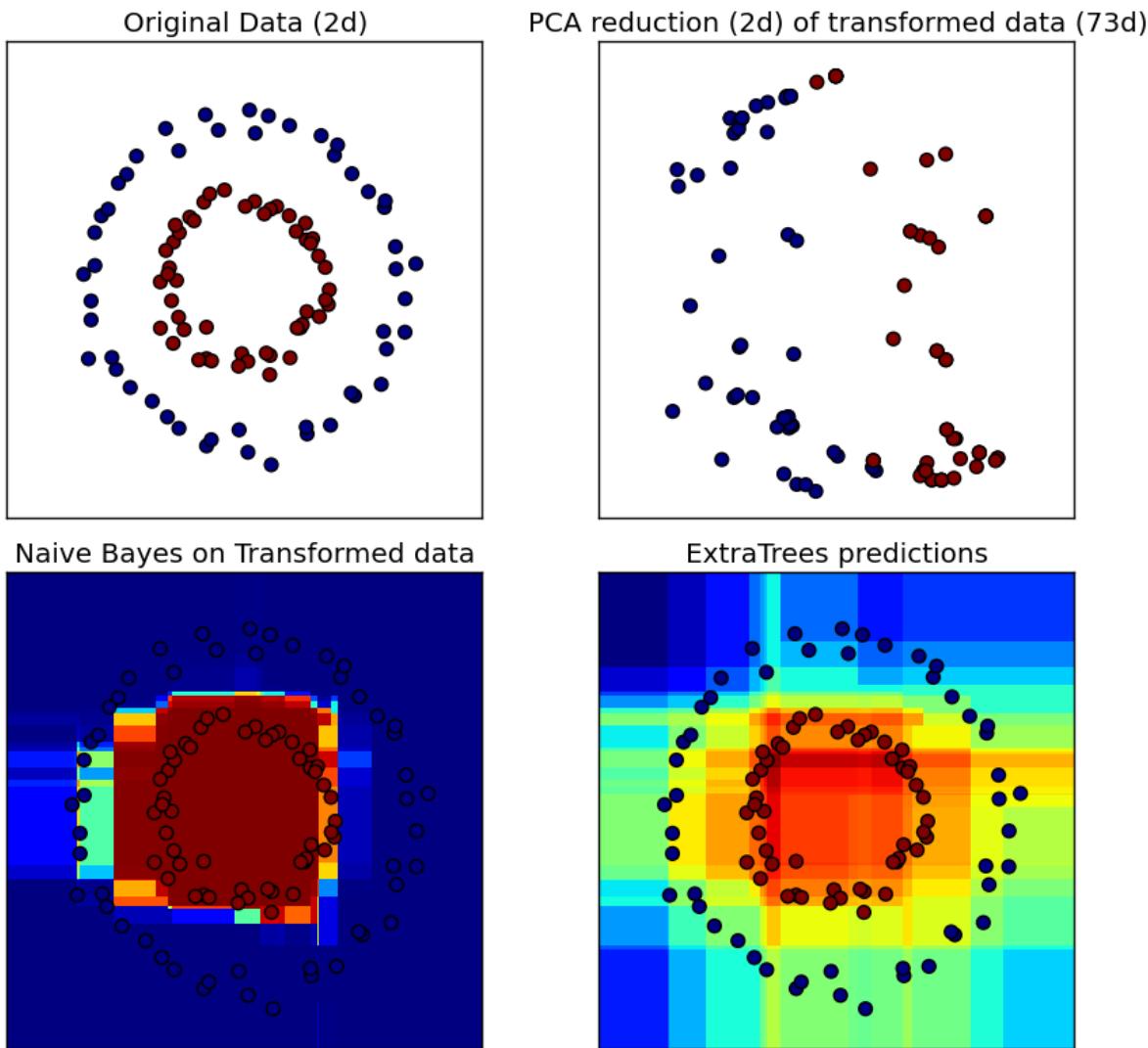
## Hashing feature transformation using Totally Random Trees

RandomTreesEmbedding provides a way to map data to a very high-dimensional, sparse representation, which might be beneficial for classification. The mapping is completely unsupervised and very efficient.

This example visualizes the partitions given by several trees and shows how the transformation can also be used for non-linear dimensionality reduction or non-linear classification.

Points that are neighboring often share the same leaf of a tree and therefore share large parts of their hashed representation. This allows to separate two concentric circles simply based on the principal components of the transformed data.

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 pylab as pl
import numpy as np

from sklearn.datasets import make_circles
from sklearn.ensemble import RandomTreesEmbedding, ExtraTreesClassifier
from sklearn.decomposition import RandomizedPCA
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 = RandomizedPCA(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 = pl.figure(figsize=(9, 8))

ax = pl.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 = pl.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 = pl.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_xlim(-1.4, 1.4)
ax.set_ylim(-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 = pl.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_xlim(-1.4, 1.4)
ax.set_ylim(-1.4, 1.4)
ax.set_xticks(())
ax.set_yticks(())

```

```
pl.tight_layout()  
pl.show()
```

Total running time of the example: 0.54 seconds

## 2.1.8 Tutorial exercises

Exercises for the tutorials

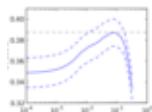
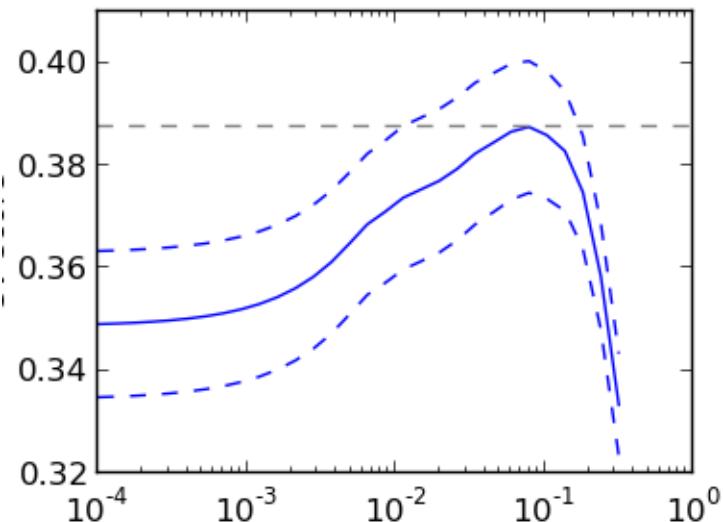


Figure 2.87: Cross-validation on diabetes Dataset Exercise

### 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: 0.16277  
[fold 2] alpha: 0.18139, score: 0.41034
```

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`

```

print __doc__

import numpy as np
import pylab as pl

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))

pl.figure(figsize=(4, 3))
pl.semilogx(alphas, scores)
# plot error lines showing +/- std. errors of the scores
pl.semilogx(alphas, np.array(scores) + np.array(scores_std) / np.sqrt(len(X)),
            'b--')
pl.semilogx(alphas, np.array(scores) - np.array(scores_std) / np.sqrt(len(X)),
            'b--')
pl.ylabel('CV score')
pl.xlabel('alpha')
pl.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"
print "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."
```

```
pl.show()
```

**Total running time of the example:** 3.00 seconds

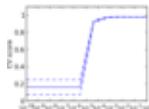
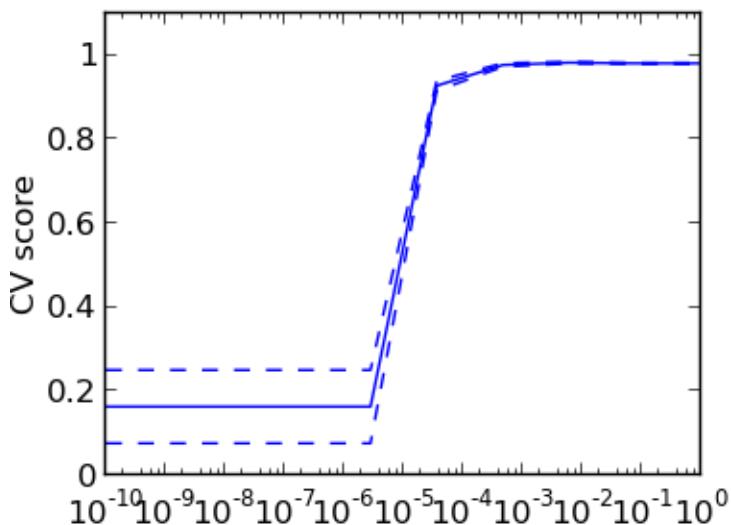


Figure 2.88: *Cross-validation on Digits Dataset Exercise*

### Cross-validation on Digits Dataset Exercise

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*.



**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')  
Cs = 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 pylab as pl
pl.figure(1, figsize=(4, 3))
pl.clf()
pl.semilogx(C_s, scores)
pl.semilogx(C_s, np.array(scores) + np.array(scores_std), 'b--')
pl.semilogx(C_s, np.array(scores) - np.array(scores_std), 'b--')
locs, labels = pl.yticks()
pl.yticks(locs, map(lambda x: "%g" % x, locs))
pl.ylabel('CV score')
pl.xlabel('Parameter C')
pl.ylim(0, 1.1)
pl.show()

```

**Total running time of the example:** 5.84 seconds

Figure 2.89: *Digits Classification Exercise*

## 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*.

### Script output:

```
KNN score: 0.961111
LogisticRegression score: 0.938889
```

**Python source code:** `plot_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))
```

**Total running time of the example:** 0.81 seconds

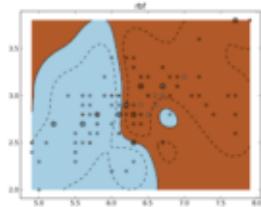
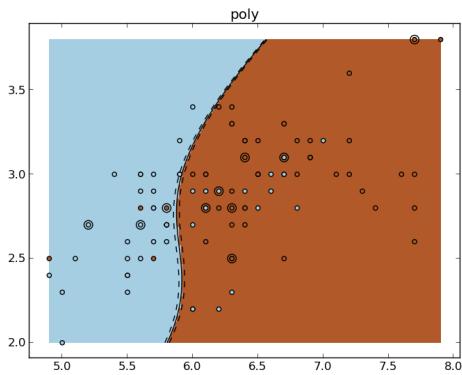
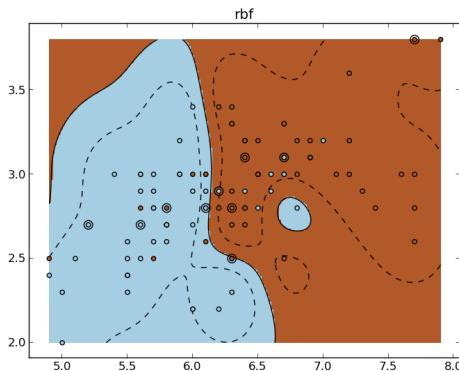


Figure 2.90: SVM Exercise

## SVM Exercise

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 pylab as pl
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)

    pl.figure(fig_num)
    pl.clf()
    pl.scatter(X[:, 0], X[:, 1], c=y, zorder=10, cmap=pl.cm.Paired)

    # Circle out the test data
    pl.scatter(X_test[:, 0], X_test[:, 1], s=80, facecolors='none', zorder=10)

    pl.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)
pl.pcolormesh(XX, YY, Z > 0, cmap=pl.cm.Paired)
pl.contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '--', '--'],
           levels=[-.5, 0, .5])

pl.title(kernel)
pl.show()
```

**Total running time of the example:** 8.12 seconds

## 2.1.9 Gaussian Process for Machine Learning

Examples concerning the `sklearn.gaussian_process` package.

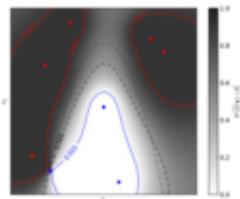
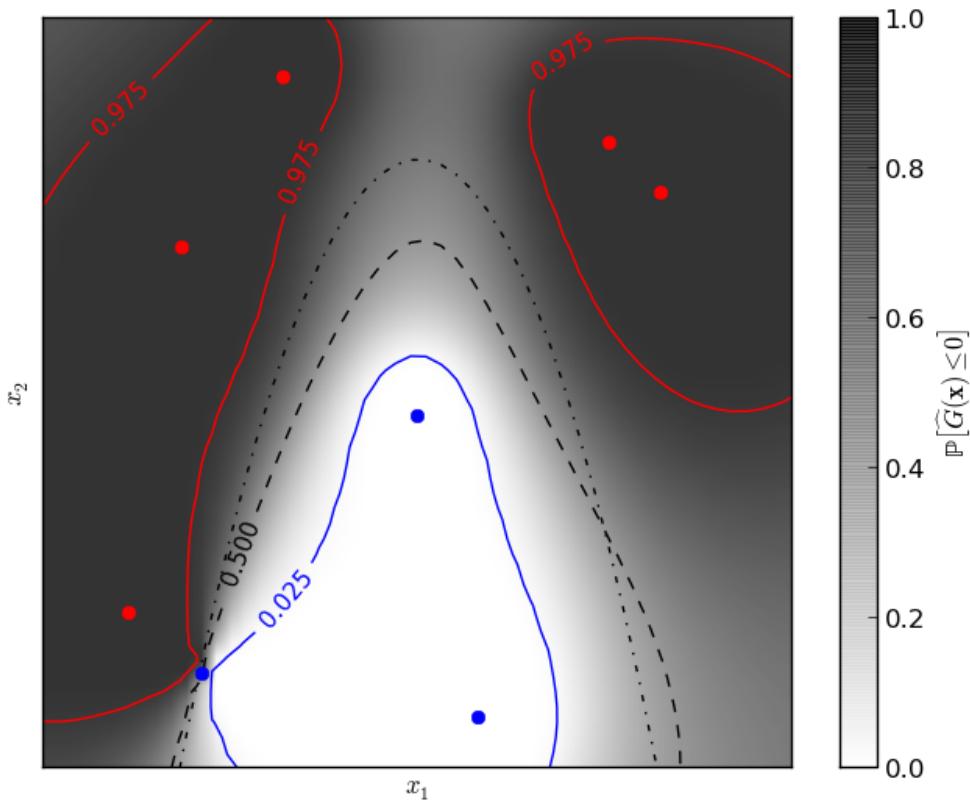


Figure 2.91: *Gaussian Processes classification example: exploiting the probabilistic output*

### 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>
# License: BSD style

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:** 0.21 seconds

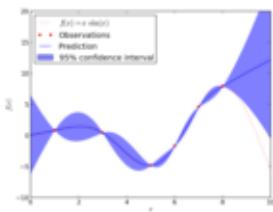


Figure 2.92: *Gaussian Processes regression: basic introductory example*

## Gaussian Processes regression: basic introductory example

A simple one-dimensional regression exercise computed in two different ways:

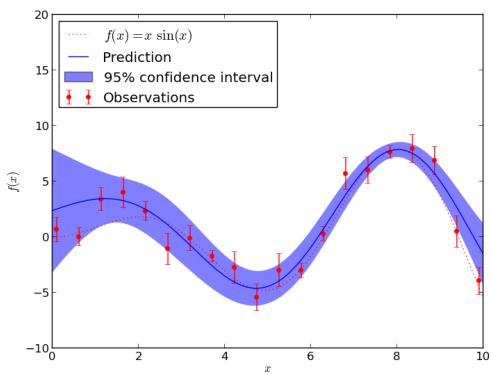
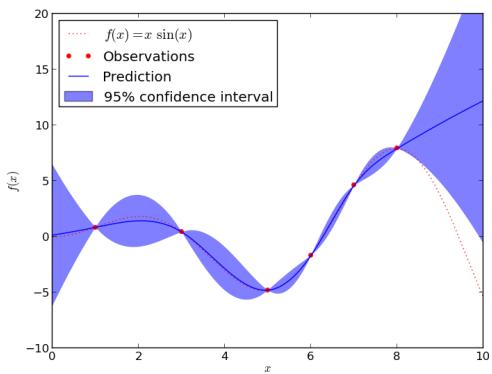
1. A noise-free case with a cubic correlation model
2. A noisy case with a squared Euclidean correlation model

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

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



**Python source code:** [plot\\_gp\\_regression.py](#)

```
print __doc__

# Author: Vincent Dubourg <vincent.dubourg@gmail.com>
#         Jake Vanderplas <vanderplas@astro.washington.edu>
# License: BSD style

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
```

```

# Instantiate 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'$f(x) = x \backslash \sin(x)$')
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

# Instantiate 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'$f(x) = x \backslash \sin(x)$')

```

```
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:** 1.13 seconds

Figure 2.93: Gaussian Processes regression: goodness-of-fit on the ‘diabetes’ dataset

## Gaussian Processes regression: goodness-of-fit on the ‘diabetes’ dataset

This example consists in fitting a Gaussian Process model onto the diabetes dataset.

The correlation parameters are determined by means of maximum likelihood estimation (MLE). An anisotropic squared exponential correlation model with a constant regression model are assumed. We also used a nugget = 1e-2 in order to account for the (strong) noise in the targets.

We compute then 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>
# License: BSD style

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))

```

## 2.1.10 Generalized Linear Models

Examples concerning the `sklearn.linear_model` package.

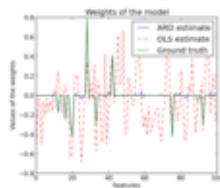


Figure 2.94: Automatic Relevance Determination Regression (ARD)

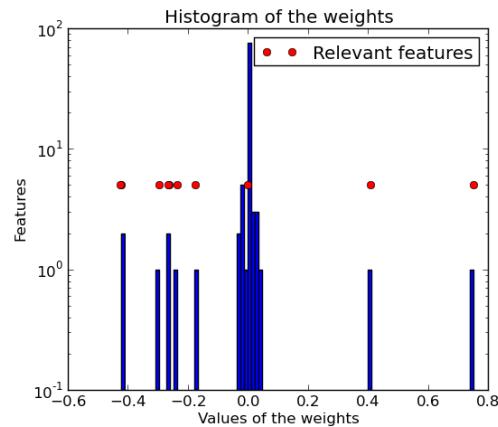
### Automatic Relevance Determination Regression (ARD)

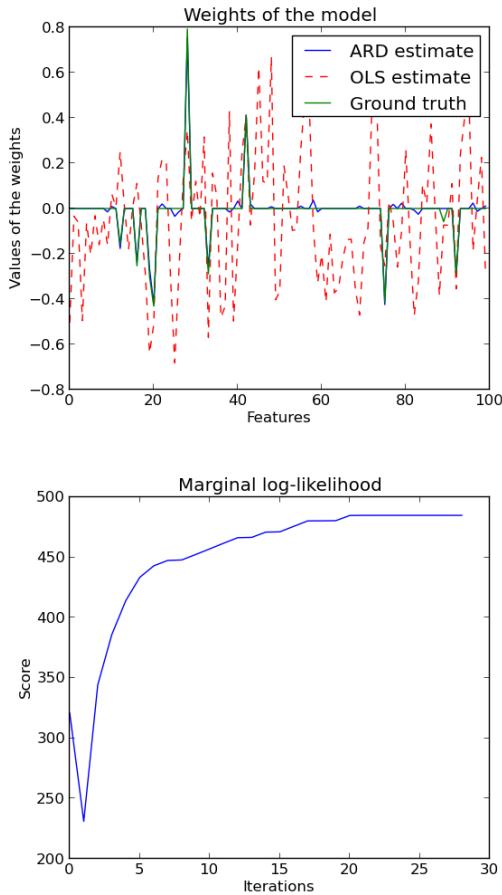
Fit regression model with *Bayesian Ridge Regression*.

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 pylab as pl
from scipy import stats

from sklearn.linear_model import ARDRegression, LinearRegression

#####
# Generating simulated data with Gaussian weights

# Parameters of the example
np.random.seed(0)
n_samples, n_features = 100, 100
# Create gaussian data
X = np.random.randn(n_samples, n_features)
# Create weights with a precision lambda_ of 4.
lambda_ = 4.
w = np.zeros(n_features)
# Only keep 10 weights of interest
relevant_features = np.random.randint(0, n_features, 10)
for i in relevant_features:
    w[i] = stats.norm.rvs(loc=0, scale=1. / np.sqrt(lambda_))
# Create noise with a precision alpha of 50.
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
pl.figure(figsize=(6, 5))
pl.title("Weights of the model")
pl.plot(clf.coef_, 'b-', label="ARD estimate")
pl.plot(ols.coef_, 'r--', label="OLS estimate")
pl.plot(w, 'g-', label="Ground truth")
pl.xlabel("Features")
pl.ylabel("Values of the weights")
pl.legend(loc=1)

pl.figure(figsize=(6, 5))
pl.title("Histogram of the weights")
pl.hist(clf.coef_, bins=n_features, log=True)
pl.plot(clf.coef_[relevant_features], 5 * np.ones(len(relevant_features)),
        'ro', label="Relevant features")
pl.ylabel("Features")
pl.xlabel("Values of the weights")
pl.legend(loc=1)

pl.figure(figsize=(6, 5))
pl.title("Marginal log-likelihood")
pl.plot(clf.scores_)
pl.ylabel("Score")
pl.xlabel("Iterations")
pl.show()

```

**Total running time of the example:** 0.52 seconds

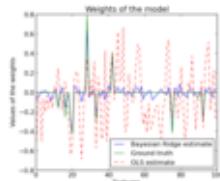


Figure 2.95: *Bayesian Ridge Regression*

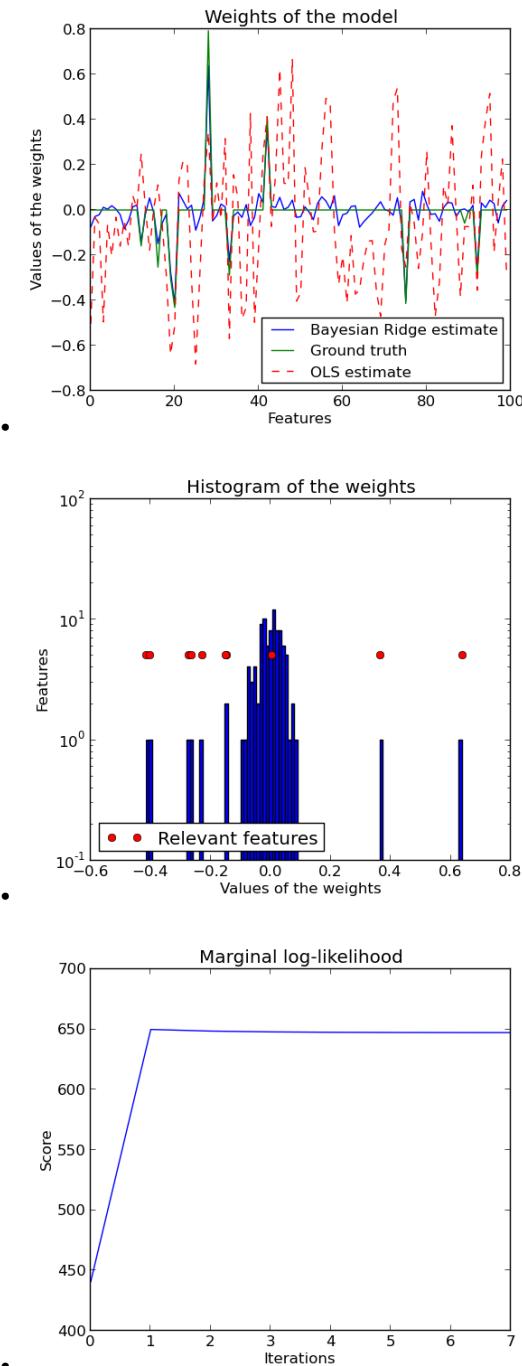
## Bayesian Ridge Regression

Computes a *Bayesian Ridge Regression* on a synthetic dataset.

Compared to the OLS (ordinary least squares) estimator, the coefficient weights are slightly shifted toward zeros, which stabilizes 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.



**Python source code:** [plot\\_bayesian\\_ridge.py](#)

```
print __doc__
```

```

import numpy as np
import pylab as pl
from scipy import stats

from sklearn.linear_model import BayesianRidge, LinearRegression

#####
# Generating simulated data with Gaussian weights
np.random.seed(0)
n_samples, n_features = 100, 100
X = np.random.randn(n_samples, n_features) # Create gaussian data
# Create weights with a precision lambda_ of 4.
lambda_ = 4.
w = np.zeros(n_features)
# Only keep 10 weights of interest
relevant_features = np.random.randint(0, n_features, 10)
for i in relevant_features:
    w[i] = stats.norm.rvs(loc=0, scale=1. / np.sqrt(lambda_))
# Create noise with a precision alpha of 50.
alpha_ = 50.
noise = stats.norm.rvs(loc=0, scale=1. / np.sqrt(alpha_), size=n_samples)
# Create the target
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
pl.figure(figsize=(6, 5))
pl.title("Weights of the model")
pl.plot(clf.coef_, 'b-', label="Bayesian Ridge estimate")
pl.plot(w, 'g-', label="Ground truth")
pl.plot(ols.coef_, 'r--', label="OLS estimate")
pl.xlabel("Features")
pl.ylabel("Values of the weights")
pl.legend(loc="best", prop=dict(size=12))

pl.figure(figsize=(6, 5))
pl.title("Histogram of the weights")
pl.hist(clf.coef_, bins=n_features, log=True)
pl.plot(clf.coef_[relevant_features], 5 * np.ones(len(relevant_features)),
        'ro', label="Relevant features")
pl.xlabel("Features")
pl.ylabel("Values of the weights")
pl.legend(loc="lower left")

pl.figure(figsize=(6, 5))
pl.title("Marginal log-likelihood")
pl.plot(clf.scores_)
pl.ylabel("Score")
pl.xlabel("Iterations")
pl.show()

```

Total running time of the example: 0.31 seconds

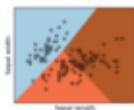
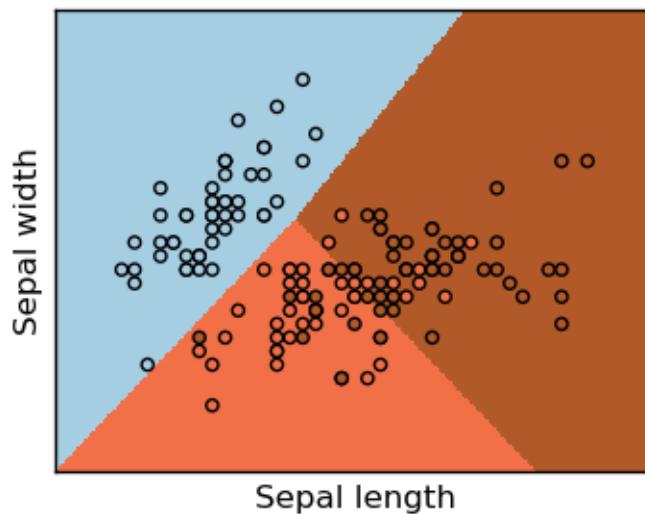


Figure 2.96: Logistic Regression 3-class Classifier

### 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.



**Python source code:** [plot\\_iris\\_logistic.py](#)

```
print __doc__\n\n# Code source: Gael Varoquaux\n# Modified for Documentation merge by Jaques Grobler\n# License: BSD\n\nimport numpy as np\nimport pylab as pl\nfrom sklearn import linear_model, datasets\n\n# import some data to play with\niris = datasets.load_iris()\nX = iris.data[:, :2] # we only take the first two features.\nY = iris.target\n\nh = .02 # step size in the mesh\n\nlogreg = 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)
pl.figure(1, figsize=(4, 3))
pl.pcolormesh(xx, yy, Z, cmap=pl.cm.Paired)

# Plot also the training points
pl.scatter(X[:, 0], X[:, 1], c=Y, edgecolors='k', cmap=pl.cm.Paired)
pl.xlabel('Sepal length')
pl.ylabel('Sepal width')

pl.xlim(xx.min(), xx.max())
pl.ylim(yy.min(), yy.max())
pl.xticks(())
pl.yticks(())

pl.show()
```

**Total running time of the example:** 0.13 seconds

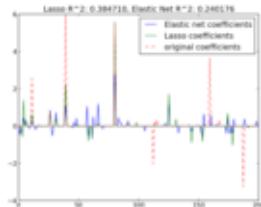
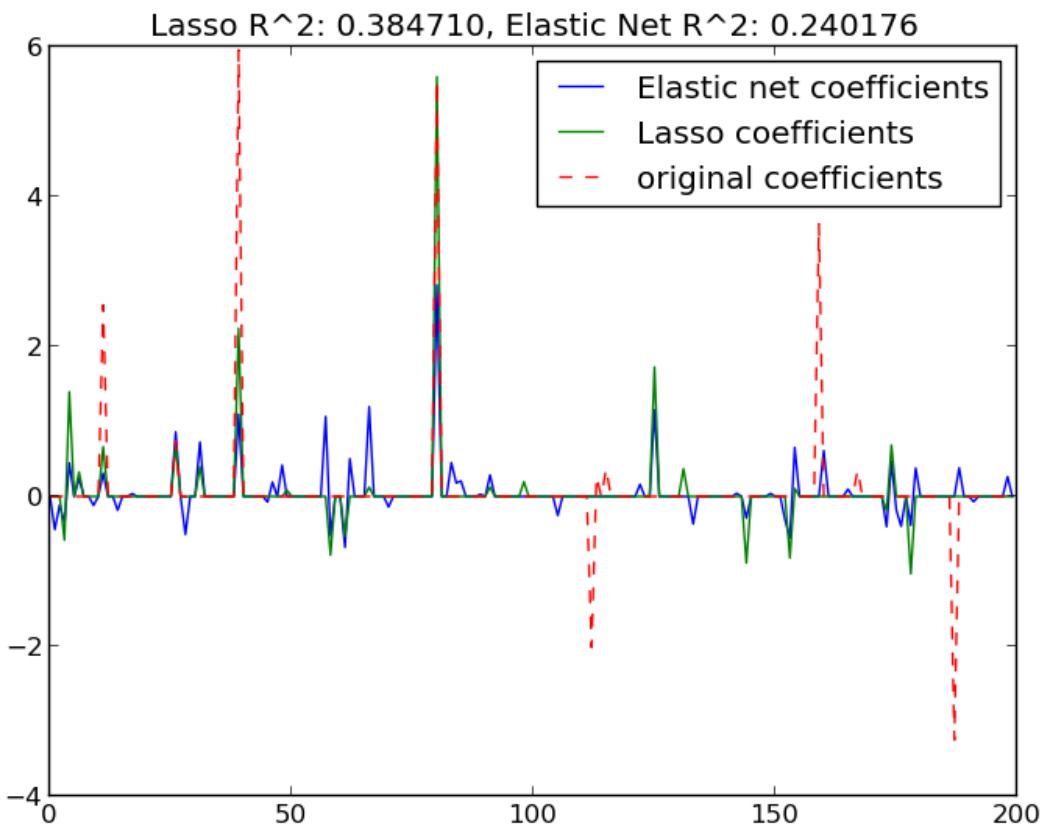


Figure 2.97: Lasso and Elastic Net for Sparse Signals

## 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='auto', 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='auto',
            rho=None, 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 pylab as pl

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

pl.plot(enet.coef_, label='Elastic net coefficients')
pl.plot(lasso.coef_, label='Lasso coefficients')
pl.plot(coef, '--', label='original coefficients')
pl.legend(loc='best')
pl.title("Lasso R^2: %f, Elastic Net R^2: %f"
         % (r2_score_lasso, r2_score_enet))
pl.show()

```

**Total running time of the example:** 0.13 seconds

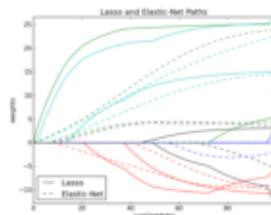
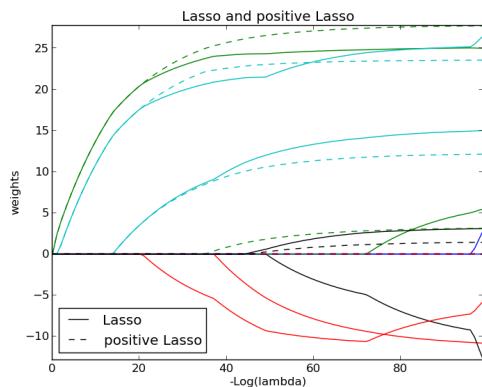
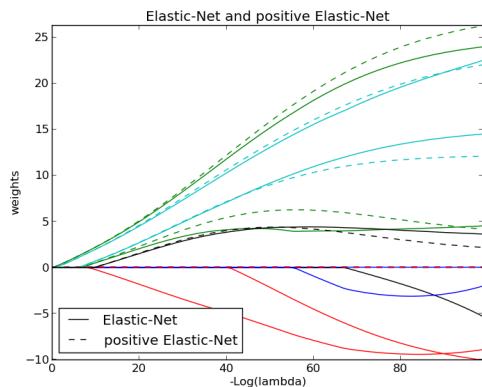
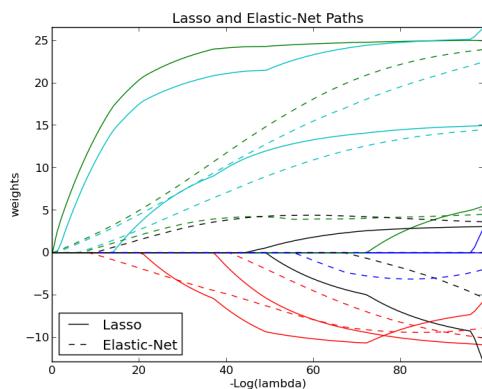


Figure 2.98: *Lasso and Elastic Net*

## 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 positive elastic net...
```

**Python source code:** [plot\\_lasso\\_coordinate\\_descent\\_path.py](#)

```

print __doc__

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD Style.

import numpy as np
import pylab as pl

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(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..."
models = lasso_path(X, y, eps=eps)
alphas_lasso = np.array([model.alpha for model in models])
coefs_lasso = np.array([model.coef_ for model in models])

print "Computing regularization path using the positive lasso..."
models = lasso_path(X, y, eps=eps, positive=True)
alphas_positive_lasso = np.array([model.alpha for model in models])
coefs_positive_lasso = np.array([model.coef_ for model in models])

print "Computing regularization path using the elastic net..."
models = enet_path(X, y, eps=eps, l1_ratio=0.8)
alphas_enet = np.array([model.alpha for model in models])
coefs_enet = np.array([model.coef_ for model in models])

print "Computing regularization path using the positive elastic net..."
models = enet_path(X, y, eps=eps, l1_ratio=0.8, positive=True)
alphas_positive_enet = np.array([model.alpha for model in models])
coefs_positive_enet = np.array([model.coef_ for model in models])

#####
# Display results

pl.figure(1)
ax = pl.gca()
ax.set_color_cycle(2 * ['b', 'r', 'g', 'c', 'k'])
l1 = pl.plot(coefs_lasso)
l2 = pl.plot(coefs_enet, linestyle='--')

pl.xlabel('-Log(lambda)')
pl.ylabel('weights')
pl.title('Lasso and Elastic-Net Paths')
pl.legend((l1[-1], l2[-1]), ('Lasso', 'Elastic-Net'), loc='lower left')
pl.axis('tight')

```

```
pl.figure(2)
ax = pl.gca()
ax.set_color_cycle(2 * ['b', 'r', 'g', 'c', 'k'])
l1 = pl.plot(coefs_lasso)
l2 = pl.plot(coefs_positive_lasso, linestyle='--')

pl.xlabel('-Log(lambda)')
pl.ylabel('weights')
pl.title('Lasso and positive Lasso')
pl.legend((l1[-1], l2[-1]), ('Lasso', 'positive Lasso'), loc='lower left')
pl.axis('tight')

pl.figure(3)
ax = pl.gca()
ax.set_color_cycle(2 * ['b', 'r', 'g', 'c', 'k'])
l1 = pl.plot(coefs_enet)
l2 = pl.plot(coefs_positive_enet, linestyle='--')

pl.xlabel('-Log(lambda)')
pl.ylabel('weights')
pl.title('Elastic-Net and positive Elastic-Net')
pl.legend((l1[-1], l2[-1]), ('Elastic-Net', 'positive Elastic-Net'),
          loc='lower left')
pl.axis('tight')
pl.show()
```

**Total running time of the example:** 0.40 seconds

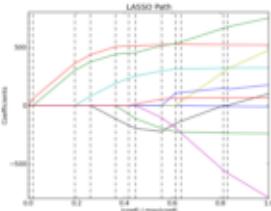
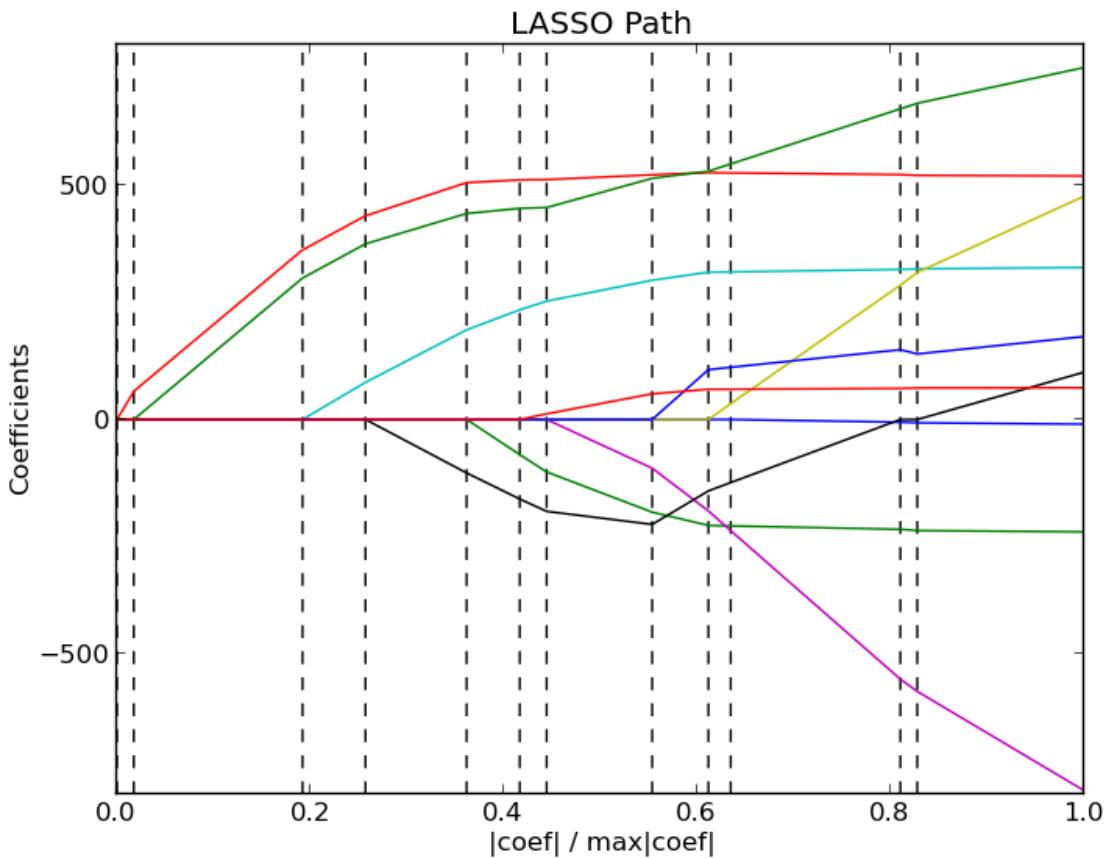


Figure 2.99: *Lasso path using LARS*

### Lasso path using LARS

Computes Lasso Path along the regularization parameter using the LARS algorithm on the diabetest 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 Style.

import numpy as np
import pylab as pl

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]

pl.plot(xx, coefs.T)
ymin, ymax = pl.ylim()
pl.vlines(xx, ymin, ymax, linestyle='dashed')
pl.xlabel('|coef| / max|coef|')
pl.ylabel('Coefficients')
pl.title('LASSO Path')
pl.axis('tight')
pl.show()
```

**Total running time of the example:** 0.13 seconds

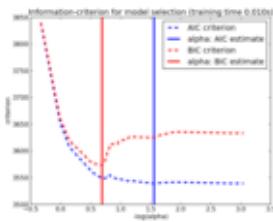


Figure 2.100: *Lasso model selection: Cross-Validation / AIC / BIC*

## 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.

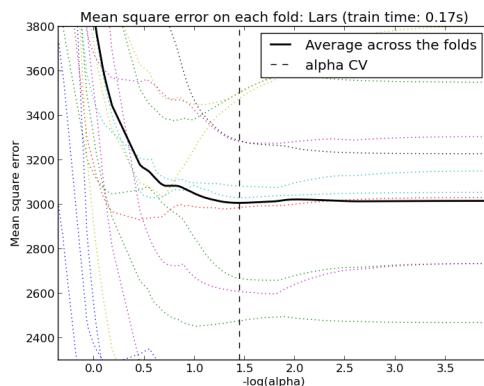
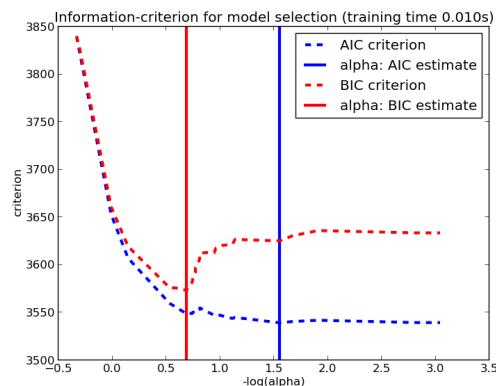
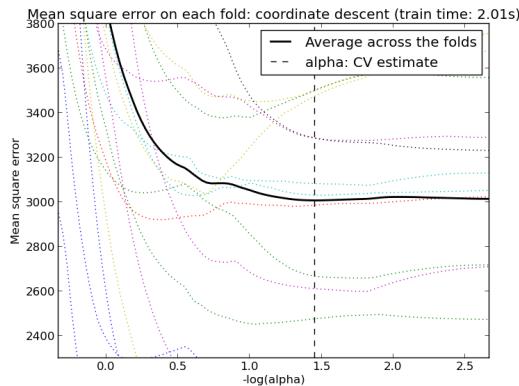
Results obtained with LassoLarsIC are based on AIC/BIC criteria.

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 Style.

import time
```

```
import numpy as np
import pylab as pl

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_
    pl.plot(-np.log10(alphas_), criterion_, '--', color=color,
            linewidth=3, label='%s criterion' % name)
    pl.axvline(-np.log10(alpha_), color=color, linewidth=3,
               label='alpha: %s estimate' % name)
    pl.xlabel('-log(alpha)')
    pl.ylabel('criterion')

    pl.figure()
    plot_ic_criterion(model_aic, 'AIC', 'b')
    plot_ic_criterion(model_bic, 'BIC', 'r')
    pl.legend()
    pl.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_)

pl.figure()
ymin, ymax = 2300, 3800
pl.plot(m_log_alphas, model.mse_path_, ':')
pl.plot(m_log_alphas, model.mse_path_.mean(axis=-1), 'k',
        label='Average across the folds', linewidth=2)
pl.axvline(-np.log10(model.alpha_), linestyle='--', color='k',
           label='alpha: CV estimate')

pl.legend()

pl.xlabel('-log(alpha)')
pl.ylabel('Mean square error')
pl.title('Mean square error on each fold: coordinate descent '
         '(train time: %.2fs)' % t_lasso_cv)
pl.axis('tight')
pl.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_)

pl.figure()
pl.plot(m_log_alphas, model.cv_mse_path_, ':')
pl.plot(m_log_alphas, model.cv_mse_path_.mean(axis=-1), 'k',
        label='Average across the folds', linewidth=2)
pl.axvline(-np.log10(model.alpha_), linestyle='--', color='k',
           label='alpha CV')
pl.legend()

pl.xlabel('-log(alpha)')
pl.ylabel('Mean square error')
pl.title('Mean square error on each fold: Lars (train time: %.2fs)'
         % t_lasso_lars_cv)
pl.axis('tight')
pl.ylim(ymin, ymax)

pl.show()

```

**Total running time of the example:** 2.47 seconds

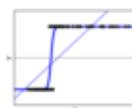
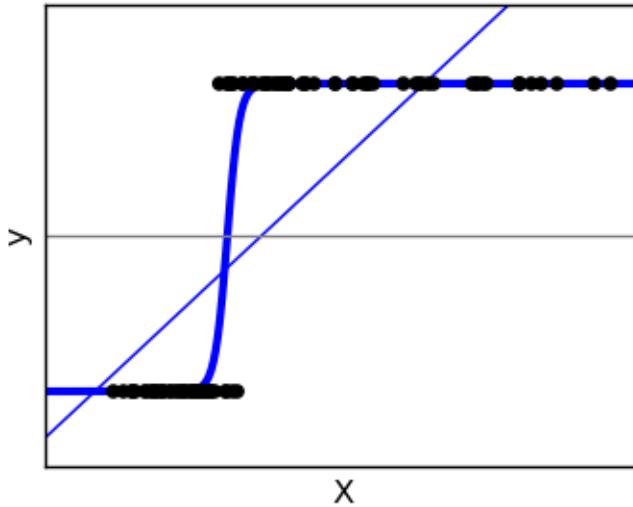


Figure 2.101: *Logit function*

## 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.



**Python source code:** [plot\\_logistic.py](#)

```
print __doc__\n\n# Code source: Gael Varoquaux\n# License: BSD\n\nimport numpy as np\nimport pylab as pl\n\nfrom sklearn import linear_model\n\n# this is our test set, it's just a straight line with some\n# gaussian noise\nxmin, xmax = -5, 5\nn_samples = 100\nnp.random.seed(0)\nX = np.random.normal(size=n_samples)\ny = (X > 0).astype(np.float)\nX[X > 0] *= 4\nX += .3 * np.random.normal(size=n_samples)\n\nX = X[:, np.newaxis]\n# run the classifier\nclf = linear_model.LogisticRegression(C=1e5)\nclf.fit(X, y)\n\n# and plot the result\npl.figure(1, figsize=(4, 3))\npl.clf()\npl.scatter(X.ravel(), y, color='black', zorder=20)\nX_test = np.linspace(-5, 10, 300)
```

```

def model(x):
    return 1 / (1 + np.exp(-x))
loss = model(X_test * clf.coef_ + clf.intercept_).ravel()
pl.plot(X_test, loss, color='blue', linewidth=3)

ols = linear_model.LinearRegression()
ols.fit(X, y)
pl.plot(X_test, ols.coef_ * X_test + ols.intercept_, linewidth=1)
pl.axhline(.5, color='.5')

pl.ylabel('y')
pl.xlabel('X')
pl.xticks(())
pl.yticks(())
pl.ylim(-.25, 1.25)
pl.xlim(-4, 10)

pl.show()

```

**Total running time of the example:** 0.14 seconds

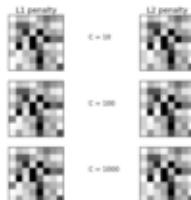
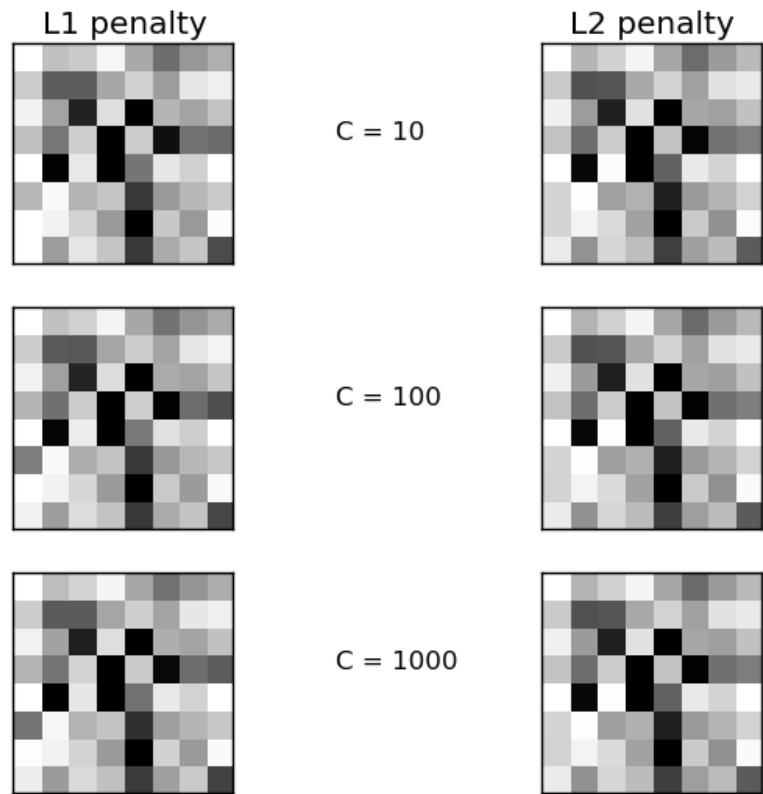


Figure 2.102: *L1 Penalty and Sparsity in Logistic Regression*

## 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=10
Sparsity with L1 penalty: 6.25%
score with L1 penalty: 0.9110
Sparsity with L2 penalty: 4.69%
score with L2 penalty: 0.9093
C=100
Sparsity with L1 penalty: 4.69%
score with L1 penalty: 0.9104
Sparsity with L2 penalty: 4.69%
score with L2 penalty: 0.9098
C=1000
Sparsity with L1 penalty: 4.69%
score with L1 penalty: 0.9093
Sparsity with L2 penalty: 4.69%
score with L2 penalty: 0.9098
```

**Python source code:** [plot\\_logistic\\_l1\\_l2\\_sparsity.py](#)

```
print __doc__

# Authors: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#          Mathieu Blondel <mathieu@mblondel.org>
#          Andreas Mueller <amueller@ais.uni-bonn.de>
# License: BSD Style.
```

```

import numpy as np
import pylab as pl

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(10. ** np.arange(1, 4)):
    # 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=%d" % 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 = pl.subplot(3, 2, 2 * i + 1)
    l2_plot = pl.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)
    pl.text(-8, 3, "C = %d" % C)

    l1_plot.set_xticks(())
    l1_plot.set_yticks(())
    l2_plot.set_xticks(())
    l2_plot.set_yticks(())

pl.show()

```

Total running time of the example: 0.53 seconds

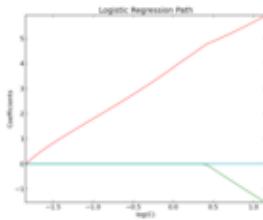
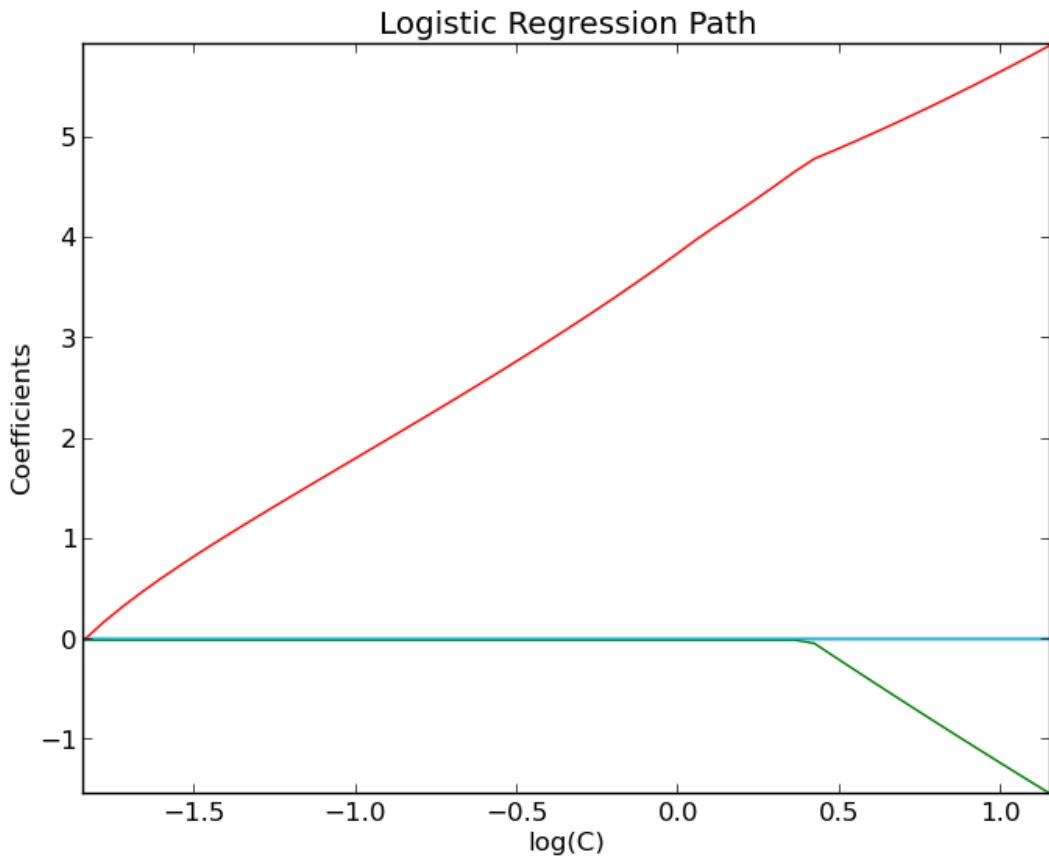


Figure 2.103: Path with L1- Logistic Regression

### Path with L1- Logistic Regression

Computes path on IRIS dataset.



#### Script output:

```
Computing regularization path ...
This took  0:00:00.022005
```

**Python source code:** [plot\\_logistic\\_path.py](#)

```

print __doc__

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD Style.

from datetime import datetime
import numpy as np
import pylab as pl

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_)
pl.plot(np.log10(cs), coefs_)
ymin, ymax = pl.ylim()
pl.xlabel('log(C)')
pl.ylabel('Coefficients')
pl.title('Logistic Regression Path')
pl.axis('tight')
pl.show()

```

**Total running time of the example:** 0.13 seconds

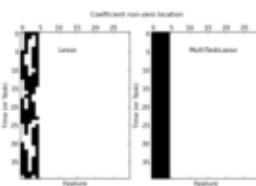
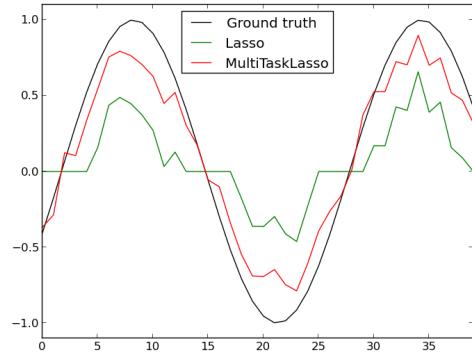
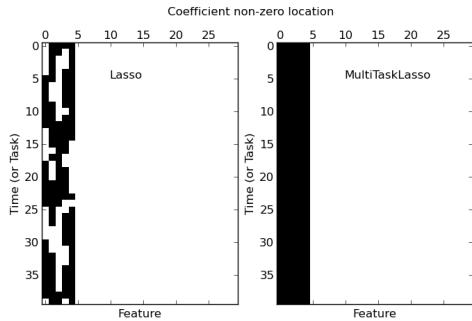


Figure 2.104: *Joint feature selection with multi-task Lasso*

## 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.



**Python source code:** [plot\\_multi\\_task\\_lasso\\_support.py](#)

```
print __doc__

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD Style.

import pylab as pl
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 = pl.figure(figsize=(8, 5))
pl.subplot(1, 2, 1)
pl.spy(coef_lasso_)
pl.xlabel('Feature')
pl.ylabel('Time (or Task)')
pl.text(10, 5, 'Lasso')
pl.subplot(1, 2, 2)
pl.spy(coef_multi_task_lasso_)
pl.xlabel('Feature')
pl.ylabel('Time (or Task)')
pl.text(10, 5, 'MultiTaskLasso')
fig.suptitle('Coefficient non-zero location')

feature_to_plot = 0
pl.figure()
pl.plot(coef[:, feature_to_plot], 'k', label='Ground truth')
pl.plot(coef_lasso_[:, feature_to_plot], 'g', label='Lasso')
pl.plot(coef_multi_task_lasso_[:, feature_to_plot],
        'r', label='MultiTaskLasso')
pl.legend(loc='upper center')
pl.axis('tight')
pl.ylim([-1.1, 1.1])
pl.show()

```

**Total running time of the example:** 0.25 seconds

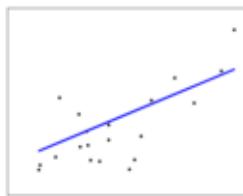
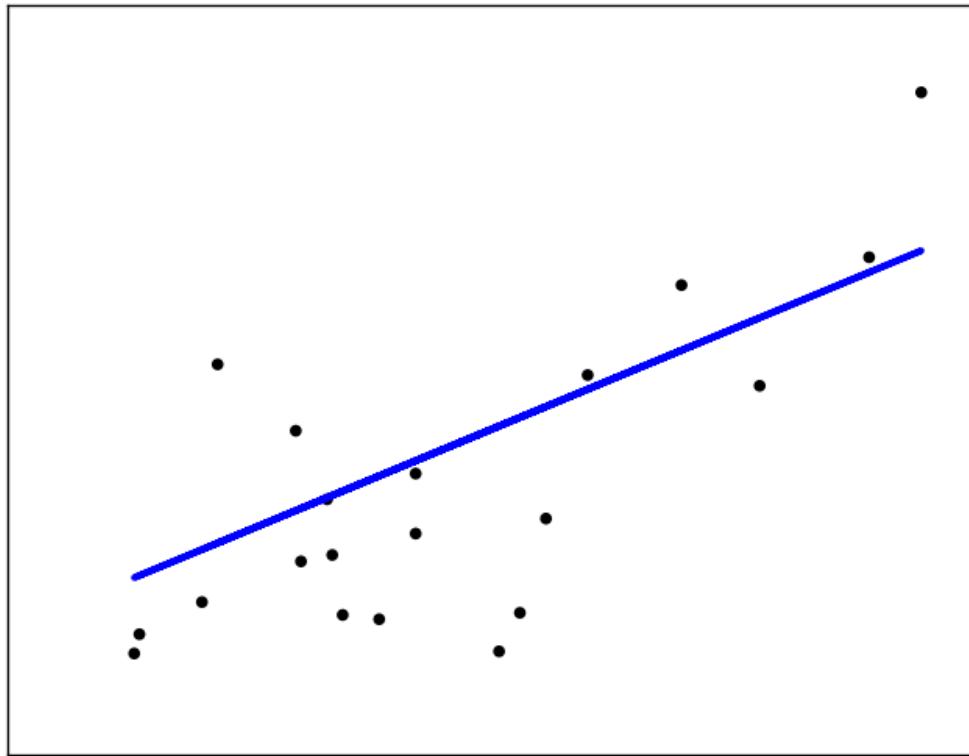


Figure 2.105: *Linear Regression Example*

## 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  
  
import pylab as pl  
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
pl.scatter(diabetes_X_test, diabetes_y_test, color='black')
pl.plot(diabetes_X_test, regr.predict(diabetes_X_test), color='blue',
        linewidth=3)

pl.xticks(())
pl.yticks(())

pl.show()

```

**Total running time of the example:** 0.16 seconds

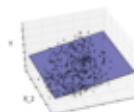
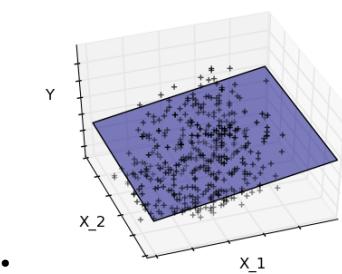
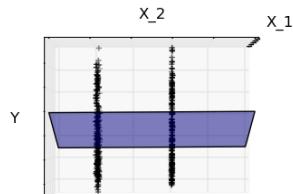
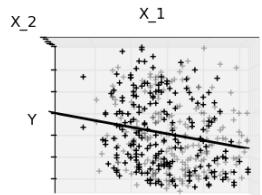


Figure 2.106: Sparsity Example: Fitting only features 1 and 2

### 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





**Python source code:** [plot\\_ols\\_3d.py](#)

```
print __doc__
```

```
# Code source: Gael Varoquaux
# Modified for Documentation merge by Jaques Grobler
# License: BSD

import pylab as pl
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 = pl.figure(fig_num, figsize=(4, 3))
    pl.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)

pl.show()

```

**Total running time of the example:** 0.22 seconds

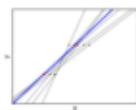
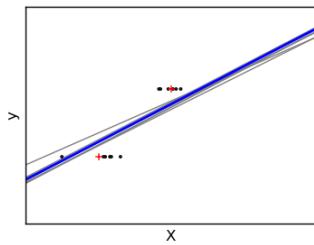


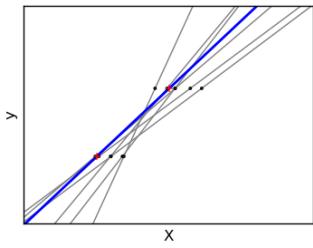
Figure 2.107: Ordinary Least Squares and Ridge Regression Variance

### 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__\n\n# Code source: Gael Varoquaux\n# Modified for Documentation merge by Jaques Grobler\n# License: BSD\n\n\nimport numpy as np\nimport pylab as pl\n\nfrom sklearn import linear_model\n\nX_train = np.c_[.5, 1].T\ny_train = [.5, 1]\nX_test = np.c_[0, 2].T\n\nnp.random.seed(0)\n\nclassifiers = dict(ols=linear_model.LinearRegression(),\n                   ridge=linear_model.Ridge(alpha=.1))\n\nfignum = 1\nfor name, clf in classifiers.iteritems():\n    fig = pl.figure(fignum, figsize=(4, 3))\n    pl.clf()\n    ax = pl.axes([.12, .12, .8, .8])\n\n    for _ in range(6):\n        this_X = .1 * np.random.normal(size=(2, 1)) + X_train\n        clf.fit(this_X, y_train)\n\n        ax.plot(X_test, clf.predict(X_test), color='.5')\n        ax.scatter(this_X, y_train, s=3, c='r', marker='o', zorder=10)\n\n    clf.fit(X_train, y_train)\n    ax.plot(X_test, clf.predict(X_test), linewidth=2, color='blue')\n    ax.scatter(X_train, y_train, s=30, c='r', marker='+', zorder=10)\n\n    ax.set_xticks(())\n    ax.set_yticks(())\n    ax.set_xlim((0, 1.6))\n    ax.set_xlabel('X')\n    ax.set_ylabel('Y')\n    ax.set_ylim(0, 2)\n    fignum += 1
```

```
pl.show()
```

**Total running time of the example:** 0.19 seconds

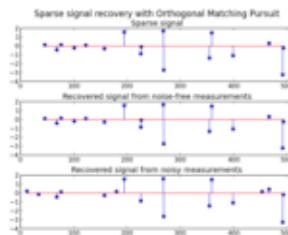
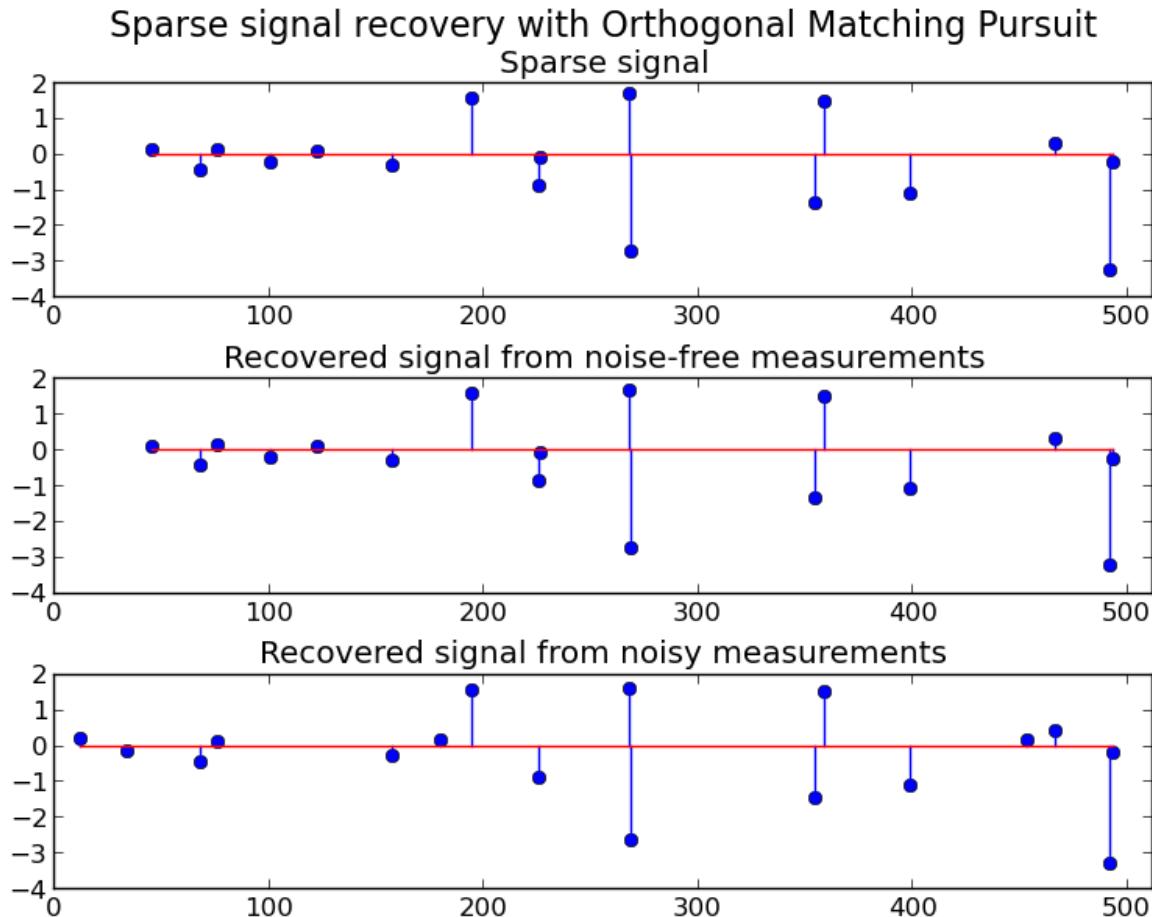


Figure 2.108: *Orthogonal Matching Pursuit*

## Orthogonal Matching Pursuit

Using orthogonal matching pursuit for recovering a sparse signal from a noisy measurement encoded with a dictionary



**Python source code:** [plot\\_omp.py](#)

```
print __doc__

import pylab as pl
```

```
import numpy as np
from sklearn.linear_model import orthogonal_mp
from sklearn.datasets import make_sparse_coded_signal

n_components, n_features = 512, 100
n_nonzero_coefs = 17

# generate the data
#####
#  $y = Dx$ 
#  $|x|_0 = n_{\text{nonzero\_coefs}}$ 

y, D, x = 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_r = x.nonzero()

# distort the clean signal
#####
y_noisy = y + 0.05 * np.random.randn(len(y))

# plot the sparse signal
#####
pl.subplot(3, 1, 1)
pl.xlim(0, 512)
pl.title("Sparse signal")
pl.stem(idx_r, x[idx_r])

# plot the noise-free reconstruction
#####
x_r = orthogonal_mp(D, y, n_nonzero_coefs)
idx_r = x_r.nonzero()
pl.subplot(3, 1, 2)
pl.xlim(0, 512)
pl.title("Recovered signal from noise-free measurements")
pl.stem(idx_r, x_r[idx_r])

# plot the noisy reconstruction
#####
x_r = orthogonal_mp(D, y_noisy, n_nonzero_coefs)
idx_r = x_r.nonzero()
pl.subplot(3, 1, 3)
pl.xlim(0, 512)
pl.title("Recovered signal from noisy measurements")
pl.stem(idx_r, x_r[idx_r])

pl.subplots_adjust(0.06, 0.04, 0.94, 0.90, 0.20, 0.38)
pl.suptitle('Sparse signal recovery with Orthogonal Matching Pursuit',
            fontsize=16)
pl.show()
```

**Total running time of the example:** 0.32 seconds

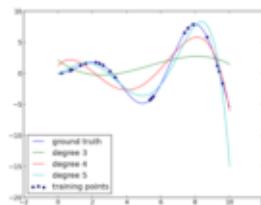


Figure 2.109: *Polynomial interpolation*

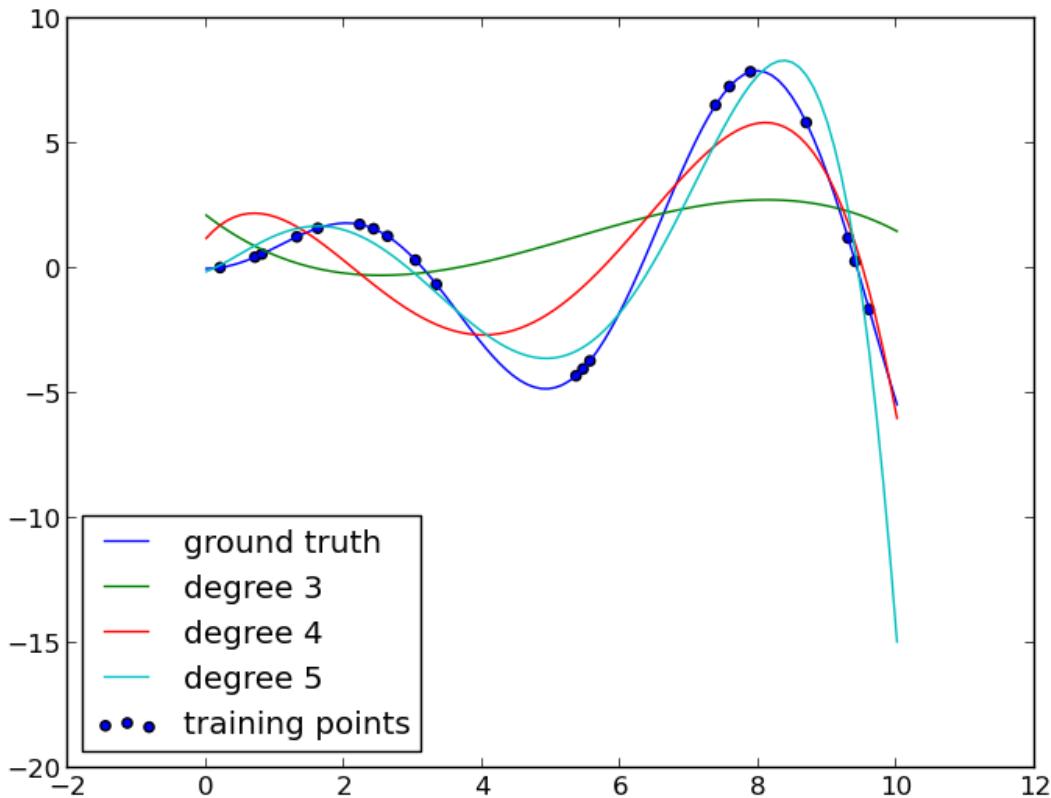
## Polynomial interpolation

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, by manually adding 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
# License: BSD Style.

import numpy as np
import pylab as pl

from sklearn.linear_model import Ridge

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)

pl.plot(x_plot, f(x_plot), label="ground truth")
pl.scatter(x, y, label="training points")

for degree in [3, 4, 5]:
    ridge = Ridge()
    ridge.fit(np.vander(x, degree + 1), y)
    pl.plot(x_plot, ridge.predict(np.vander(x_plot, degree + 1)),
            label="degree %d" % degree)

pl.legend(loc='lower left')
pl.show()
```

**Total running time of the example:** 0.13 seconds

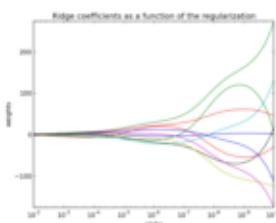
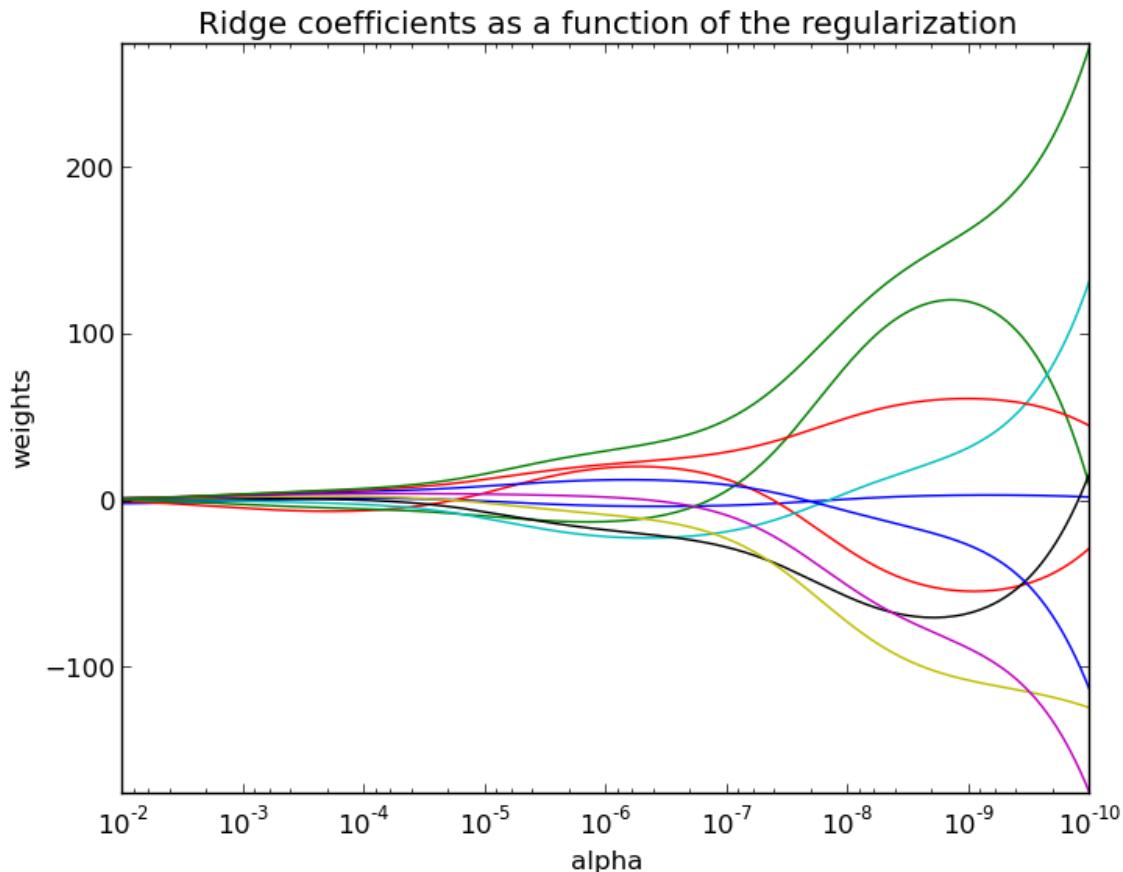


Figure 2.110: Plot Ridge coefficients as a function of the regularization

### Plot Ridge coefficients as a function of the regularization

Shows the effect of collinearity in the coefficients or the Ridge. 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 Style.

print __doc__

import numpy as np
import pylab as pl
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 = pl.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
pl.xlabel('alpha')
pl.ylabel('weights')
pl.title('Ridge coefficients as a function of the regularization')
pl.axis('tight')
pl.show()
```

**Total running time of the example:** 0.22 seconds

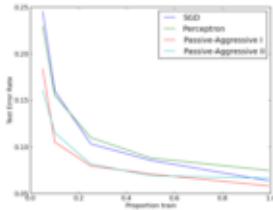
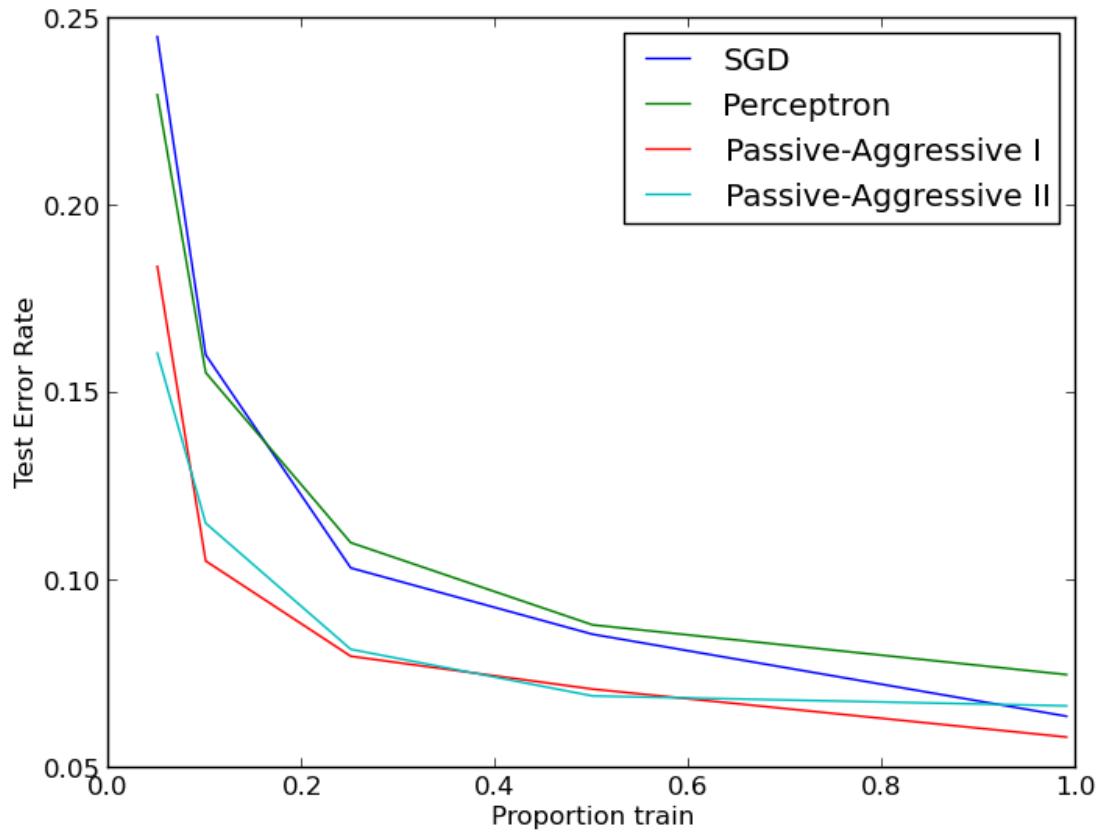


Figure 2.111: Comparing various online solvers

### 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: Simplified BSD

import numpy as np
import pylab as pl
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()

classifiers = [
    ("SGD", SGDClassifier()),
    ("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:
    yy = []
    for i in heldout:
        yy_ = []
        for r in range(rounds):
            X_train, X_test, y_train, y_test = train_test_split(digits.data,
                                                               digits.target,
                                                               test_size=i)
            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_))
    pl.plot(xx, yy, label=name)

pl.legend(loc="upper right")
pl.xlabel("Proportion train")
pl.ylabel("Test Error Rate")
pl.show()
```

**Total running time of the example:** 4.81 seconds

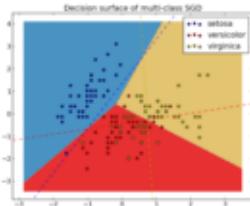
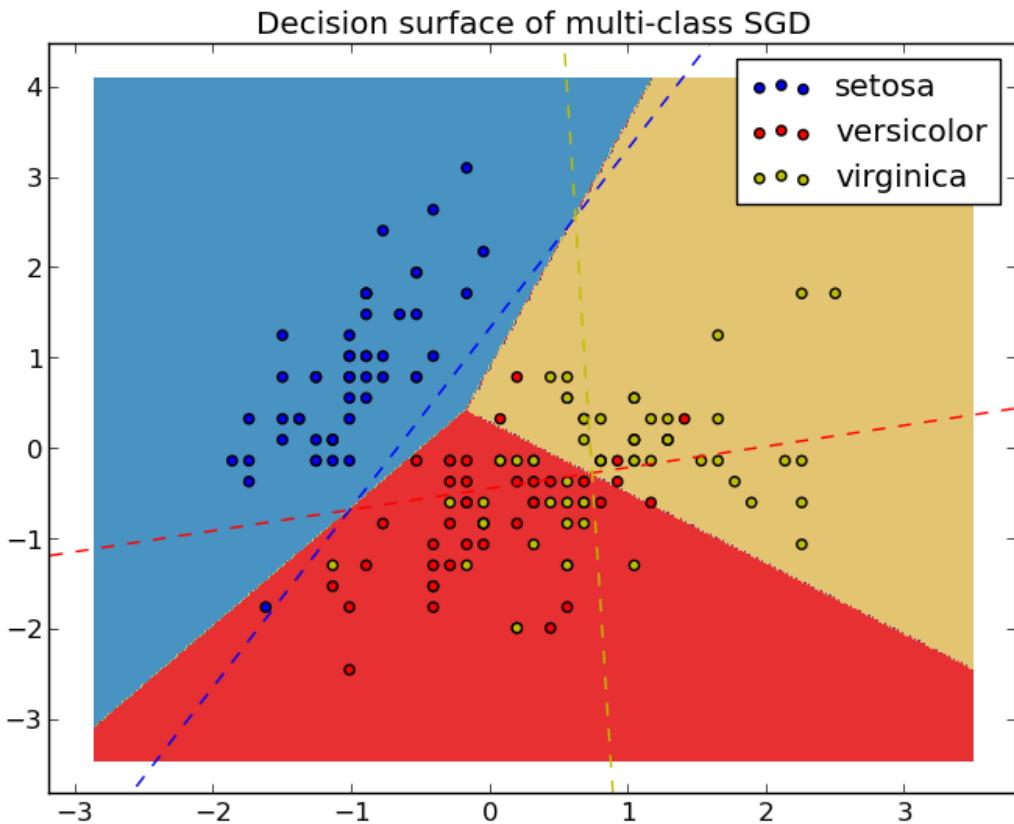


Figure 2.112: Plot multi-class SGD on the iris dataset

### 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 pylab as pl
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 = pl.contourf(xx, yy, Z, cmap=pl.cm.Paired)
pl.axis('tight')

# Plot also the training points
for i, color in zip(clf.classes_, colors):
    idx = np.where(y == i)
    pl.scatter(X[idx, 0], X[idx, 1], c=color, label=iris.target_names[i],
               cmap=pl.cm.Paired)
pl.title("Decision surface of multi-class SGD")
pl.axis('tight')

# Plot the three one-against-all classifiers
xmin, xmax = pl.xlim()
ymin, ymax = pl.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]

    pl.plot([xmin, xmax], [line(xmin), line(xmax)],
            ls="--", color=color)

for i, color in zip(clf.classes_, colors):
    plot_hyperplane(i, color)
pl.legend()
pl.show()
```

**Total running time of the example:** 0.18 seconds

## SGD: Convex Loss Functions

Plot the convex loss functions supported by `sklearn.linear_model.stochastic_gradient`.

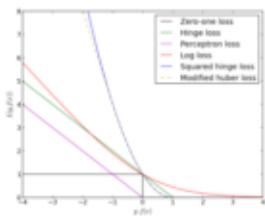
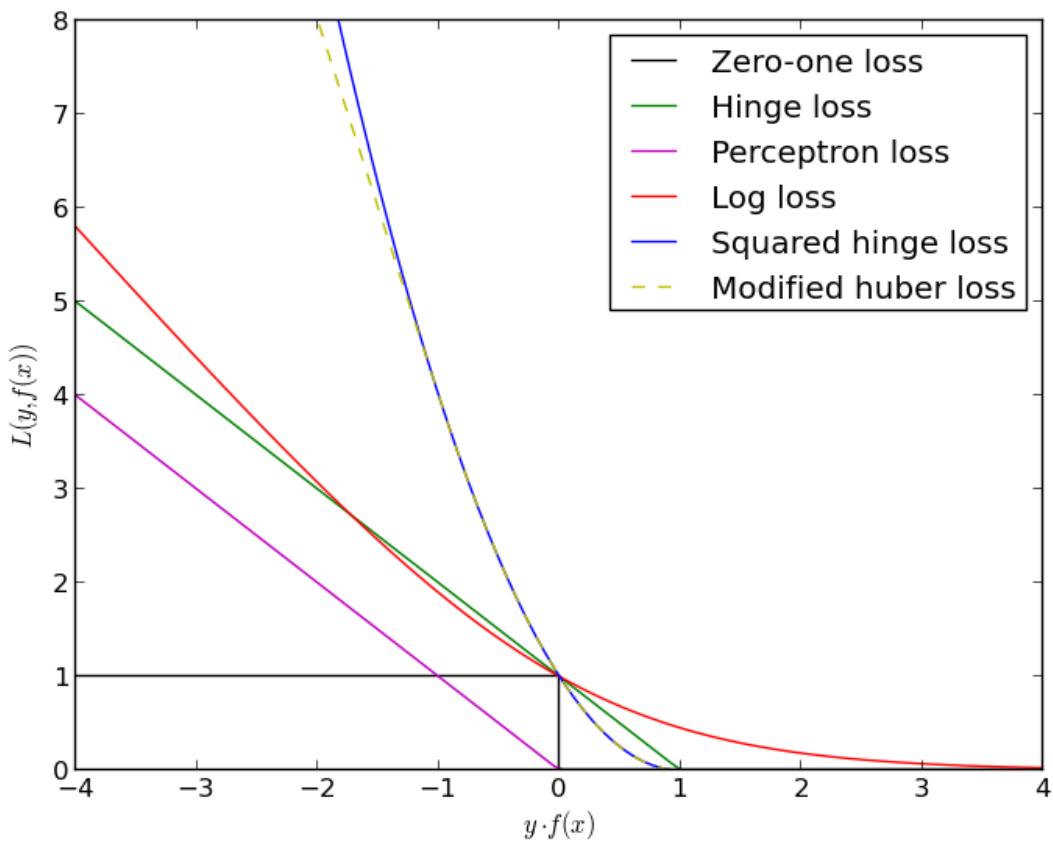


Figure 2.113: SGD: Convex Loss Functions



**Python source code:** [plot\\_sgd\\_loss\\_functions.py](#)

```
print __doc__

import numpy as np
import pylab as pl
from sklearn.linear_model.sgd_fast import SquaredHinge
from sklearn.linear_model.sgd_fast import Hinge
from sklearn.linear_model.sgd_fast import ModifiedHuber
from sklearn.linear_model.sgd_fast import SquaredLoss

#####
# Define loss functions
xmin, xmax = -4, 4
```

```
hinge = Hinge(1)
squared_hinge = SquaredHinge()
perceptron = Hinge(0)
log_loss = lambda z, p: np.log2(1.0 + np.exp(-z))
modified_huber = ModifiedHuber()
squared_loss = SquaredLoss()

#####
# Plot loss functions
xx = np.linspace(xmin, xmax, 100)
pl.plot([xmin, 0, 0, xmax], [1, 1, 0, 0], 'k-',
        label="Zero-one loss")
pl.plot(xx, [hinge.loss(x, 1) for x in xx], 'g-',
        label="Hinge loss")
pl.plot(xx, [perceptron.loss(x, 1) for x in xx], 'm-',
        label="Perceptron loss")
pl.plot(xx, [log_loss(x, 1) for x in xx], 'r-',
        label="Log loss")
#pl.plot(xx, [2 * squared_loss.loss(x, 1) for x in xx], 'c-',
#        label="Squared loss")
pl.plot(xx, [squared_hinge.loss(x, 1) for x in xx], 'b-',
        label="Squared hinge loss")
pl.plot(xx, [modified_huber.loss(x, 1) for x in xx], 'y--',
        label="Modified huber loss")
pl.ylim((0, 8))
pl.legend(loc="upper right")
pl.xlabel(r"$y \cdot f(x)$")
pl.ylabel("$L(y, f(x))$")
pl.show()
```

Total running time of the example: 0.10 seconds

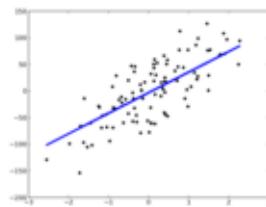
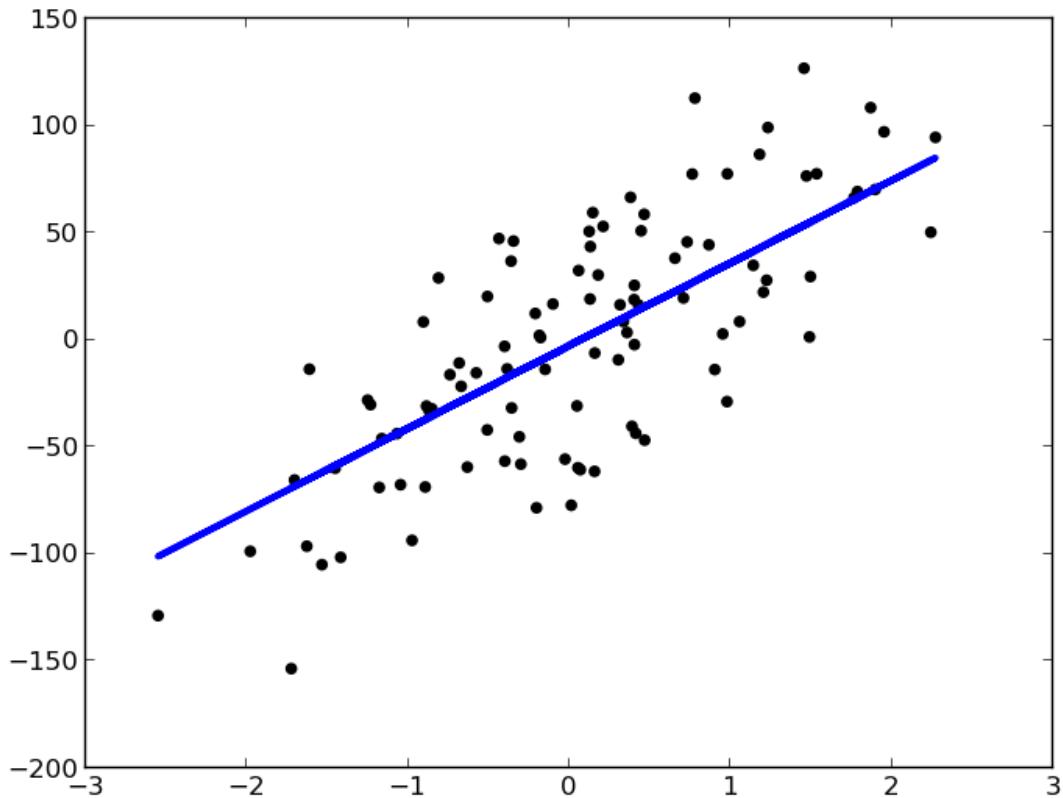


Figure 2.114: Ordinary Least Squares with SGD

### Ordinary Least Squares with SGD

Simple Ordinary Least Squares example with stochastic gradient descent, we draw the linear least squares solution for a random set of points in the plane.



**Python source code:** [plot\\_sgd\\_ols.py](#)

```
print __doc__

import pylab as pl

from sklearn.linear_model import SGDRegressor
from sklearn.datasets.samples_generator import make_regression

# this is our test set, it's just a straight line with some
# gaussian noise
X, Y = make_regression(n_samples=100, n_features=1, n_informative=1,
                       random_state=0, noise=35)

# run the classifier
clf = SGDRegressor(alpha=0.1, n_iter=20)
clf.fit(X, Y)

# and plot the result
pl.scatter(X, Y, color='black')
pl.plot(X, clf.predict(X), color='blue', linewidth=3)
pl.show()
```

**Total running time of the example:** 0.08 seconds

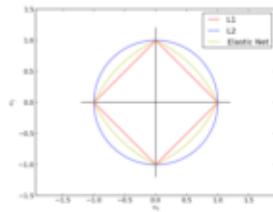
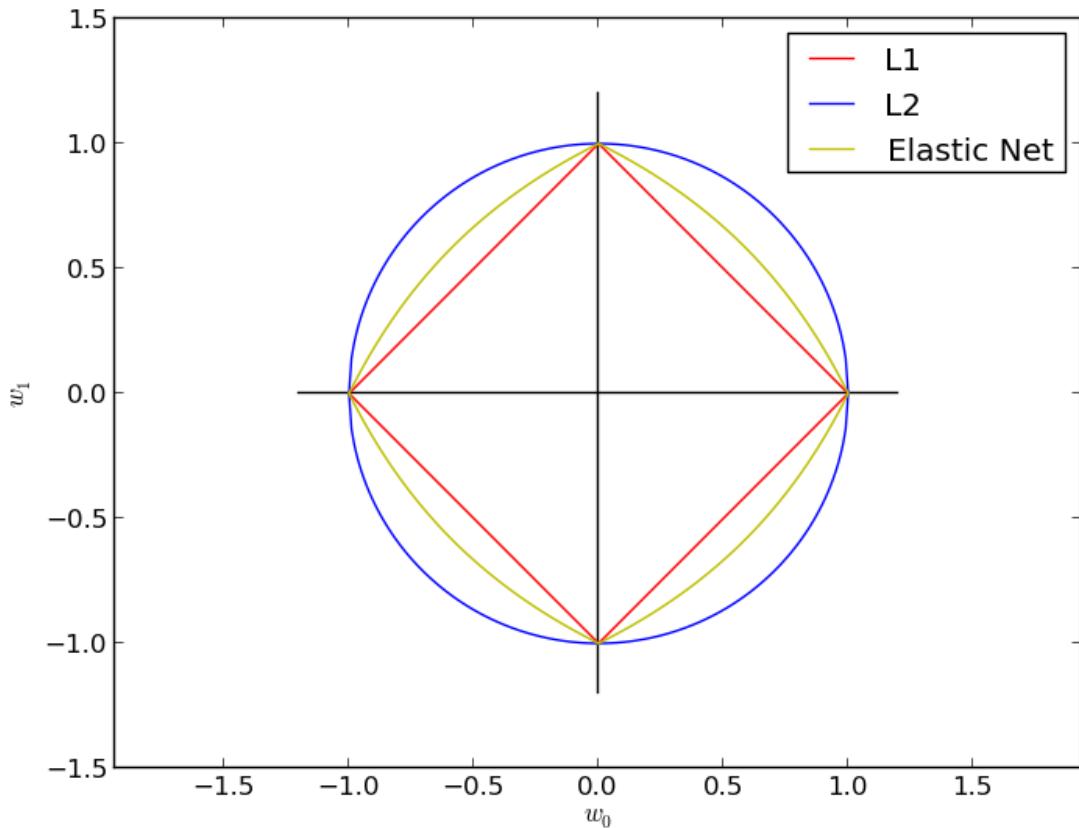


Figure 2.115: SGD: Penalties

### SGD: Penalties

Plot the contours of the three penalties 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 pylab as pl

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):
    pl.plot([-ext, ext], [0, 0], "k-")
    pl.plot([0, 0], [-ext, ext], "k-")

xs = np.linspace(0, 1, 100)

alpha = 0.501 # 0.5 division through zero

cross(1.2)

pl.plot(xs, l1(xs), "r-", label="L1")
pl.plot(xs, -1.0 * l1(xs), "r-")
pl.plot(-1 * xs, l1(xs), "r-")
pl.plot(-1 * xs, -1.0 * l1(xs), "r-")

pl.plot(xs, l2(xs), "b-", label="L2")
pl.plot(xs, -1.0 * l2(xs), "b-")
pl.plot(-1 * xs, l2(xs), "b-")
pl.plot(-1 * xs, -1.0 * l2(xs), "b-")

pl.plot(xs, el(xs, alpha), "y-", label="Elastic Net")
pl.plot(xs, -1.0 * el(xs, alpha), "y-")
pl.plot(-1 * xs, el(xs, alpha), "y-")
pl.plot(-1 * xs, -1.0 * el(xs, alpha), "y-")

pl.xlabel(r"$w_0$")
pl.ylabel(r"$w_1$")
pl.legend()

pl.axis("equal")
pl.show()

```

**Total running time of the example:** 0.13 seconds

## 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.

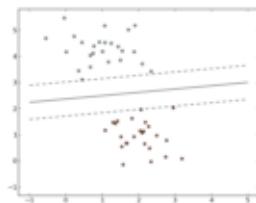
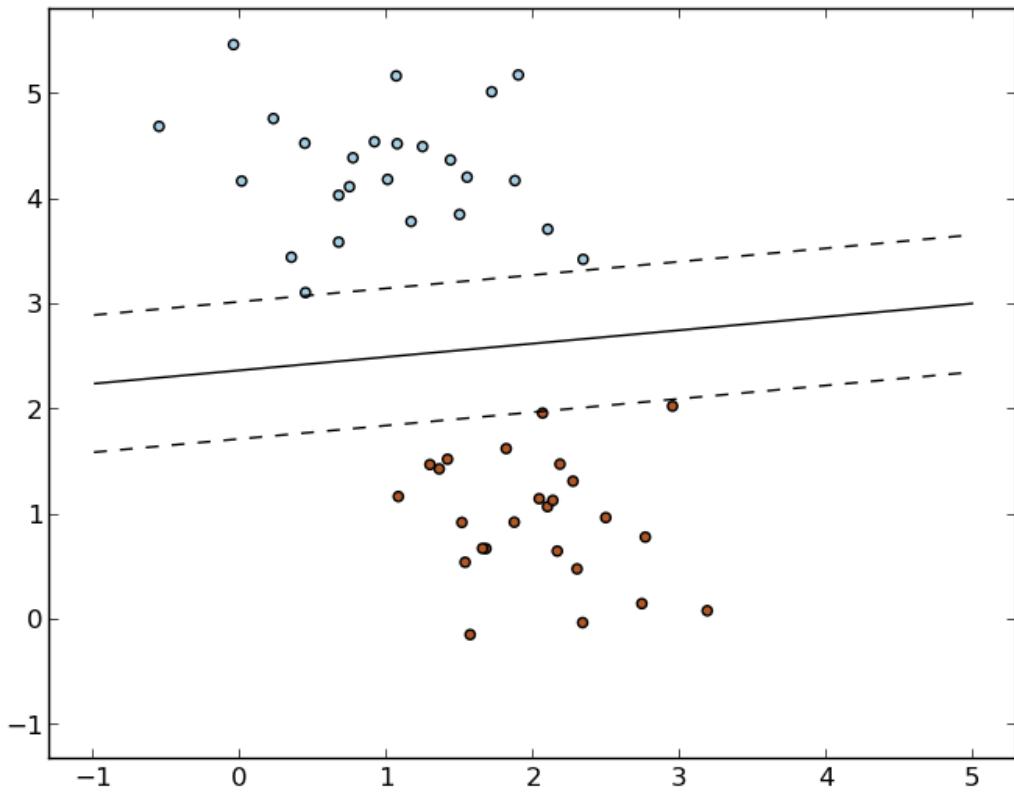


Figure 2.116: SGD: Maximum margin separating hyperplane



**Python source code:** [plot\\_sgd\\_separating\\_hyperplane.py](#)

```
print __doc__

import numpy as np
import pylab as pl
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'
pl.contour(X1, X2, Z, levels, colors=colors, linestyles=linestyles)
pl.scatter(X[:, 0], X[:, 1], c=Y, cmap=pl.cm.Paired)

pl.axis('tight')
pl.show()

```

**Total running time of the example:** 0.11 seconds

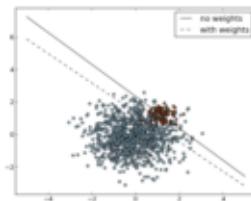
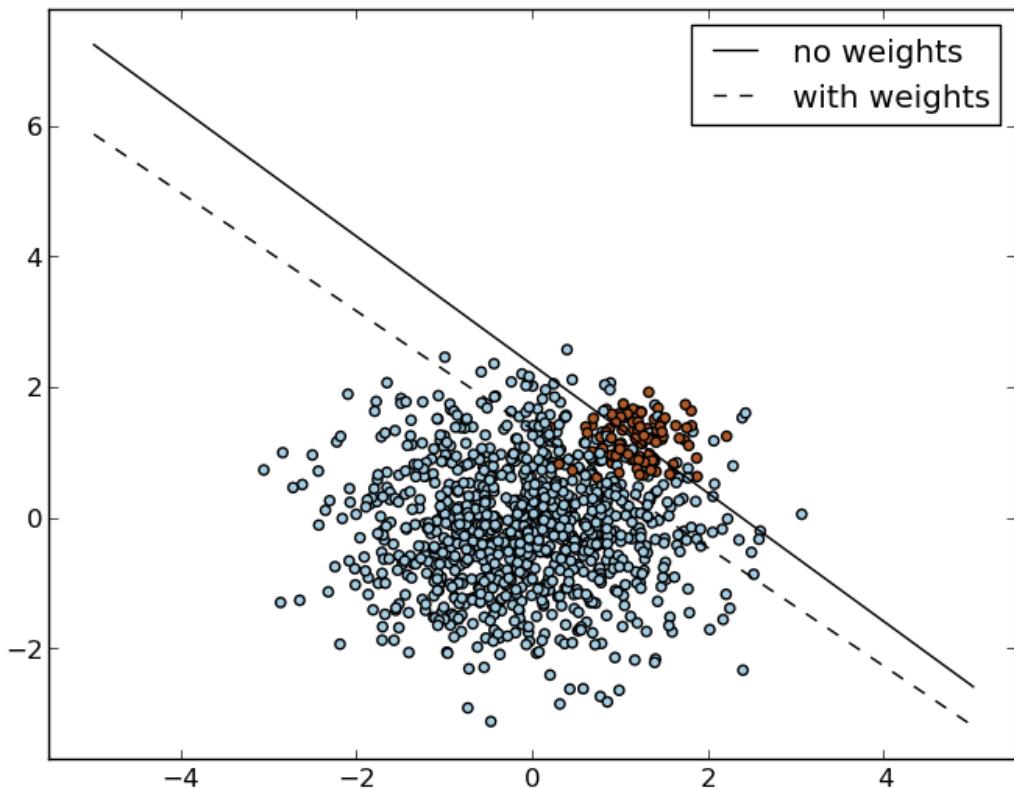


Figure 2.117: SGD: Separating hyperplane with weighted classes

### SGD: Separating hyperplane with weighted classes

Fit linear SVMs with and without class weighting. Allows to handle problems with unbalanced classes.



**Python source code:** [plot\\_sgd\\_weighted\\_classes.py](#)

```
print __doc__

import numpy as np
import pylab as pl
from sklearn.linear_model import SGDClassifier

# we create 40 separable points
np.random.seed(0)
n_samples_1 = 1000
n_samples_2 = 100
X = np.r_[1.5 * np.random.randn(n_samples_1, 2),
          0.5 * np.random.randn(n_samples_2, 2) + [2, 2]]
y = np.array([0] * (n_samples_1) + [1] * (n_samples_2), dtype=np.float64)
idx = np.arange(y.shape[0])
np.random.shuffle(idx)
X = X[idx]
y = y[idx]
mean = X.mean(axis=0)
std = X.std(axis=0)
X = (X - mean) / std

# fit the model and get the separating hyperplane
clf = SGDClassifier(n_iter=100, alpha=0.01)
clf.fit(X, y)
```

```
w = clf.coef_.ravel()
a = -w[0] / w[1]
xx = np.linspace(-5, 5)
yy = a * xx - clf.intercept_ / w[1]

# get the separating hyperplane using weighted classes
wclf = SGDClassifier(n_iter=100, alpha=0.01, class_weight={1: 10})
wclf.fit(X, y)

ww = wclf.coef_.ravel()
wa = -ww[0] / ww[1]
wy = wa * xx - wclf.intercept_ / ww[1]

# plot separating hyperplanes and samples
h0 = pl.plot(xx, yy, 'k-', label='no weights')
h1 = pl.plot(xx, wy, 'k--', label='with weights')
pl.scatter(X[:, 0], X[:, 1], c=y, cmap=pl.cm.Paired)
pl.legend()

pl.axis('tight')
pl.show()
```

**Total running time of the example:** 0.12 seconds

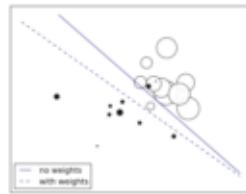
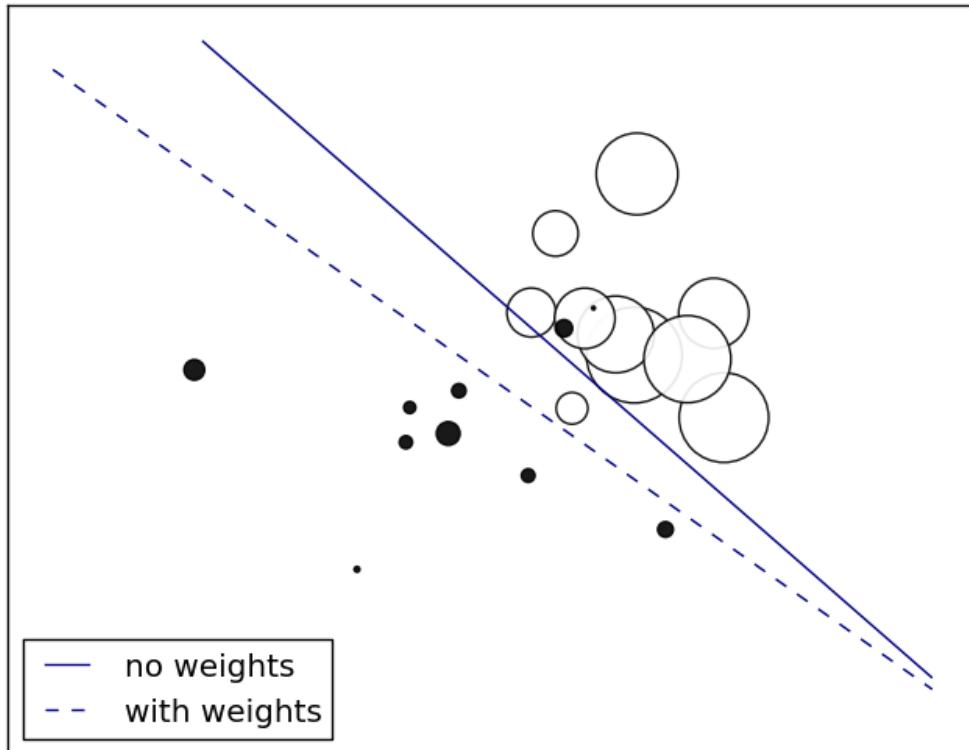


Figure 2.118: SGD: Weighted samples

### 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 pylab as pl
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))
pl.figure()
pl.scatter(X[:, 0], X[:, 1], c=y, s=sample_weight, alpha=0.9,
           cmap=pl.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 = pl.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 = pl.contour(xx, yy, Z, levels=[0], linestyles=['dashed'])

pl.legend([no_weights.collections[0], samples_weights.collections[0]],
          ["no weights", "with weights"], loc="lower left")

pl.xticks(())
pl.yticks(())
pl.show()

```

**Total running time of the example:** 0.14 seconds

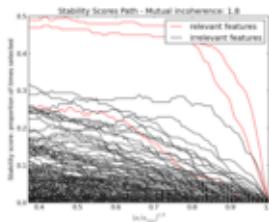


Figure 2.119: Sparse recovery: feature selection for sparse linear models

## Sparse recovery: feature selection for sparse linear models

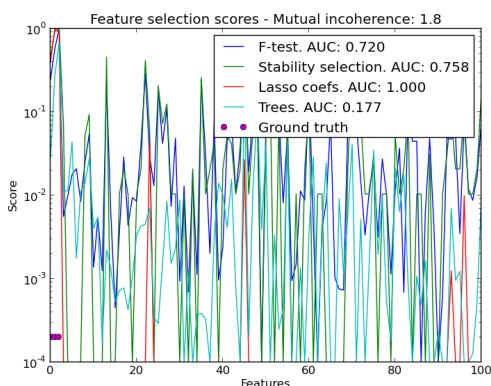
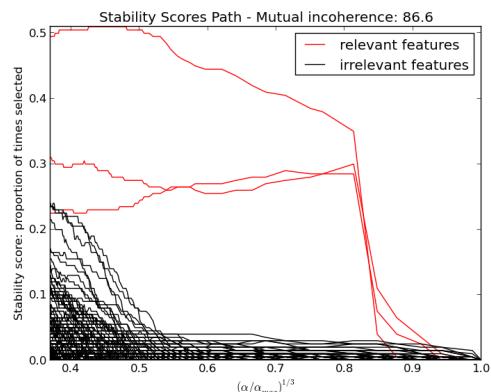
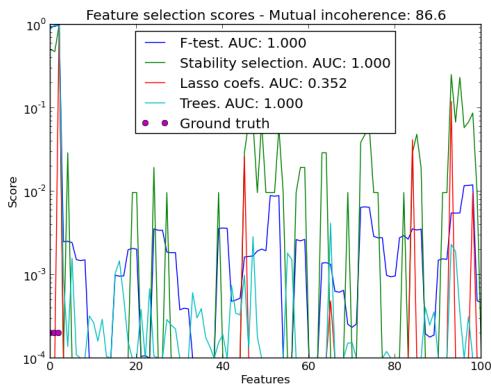
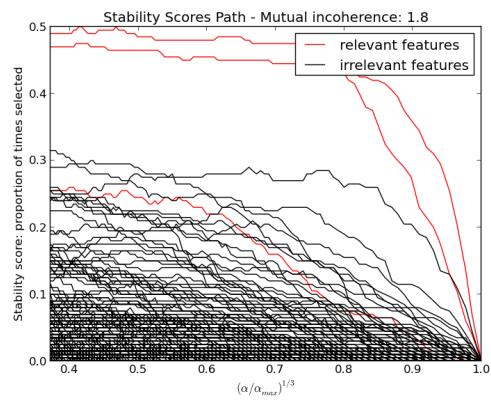
Given a small number of observations, we want to recover which features of  $X$  are relevant to explain  $y$ . For this *sparse linear models* can outperform standard statistical tests if the true model is sparse, i.e. if a small fraction of the features are relevant.

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 conditionning 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 conditionning of the design matrix. For a well-conditionned 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-conditionned 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

import warnings

import pylab as pl
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

def mutual_incoherence(X_relevant, X_irrelevant):
    """Mutual incoherence, as defined by formula (26a) of [Wainwright2006].
    """
    projector = np.dot(np.dot(X_irrelevant.T, X_relevant),
                       pinvh(np.dot(X_relevant.T, X_relevant)))
    return np.max(np.abs(projector).sum(axis=1))

for conditionning in (1, 1e-4):
    ##### Simulate regression data with a correlated design #####
    # 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-conditionned 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 - conditionning
    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)

pl.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 = pl.plot(alpha_grid[1:] ** .333, scores_path[coef != 0].T[1:], 'r')
hb = pl.plot(alpha_grid[1:] ** .333, scores_path[coef == 0].T[1:], 'k')
ymin, ymax = pl.ylim()
pl.xlabel(r'$\alpha / \alpha_{\max})^{1/3}$')
pl.ylabel('Stability score: proportion of times selected')
pl.title('Stability Scores Path - Mutual incoherence: %.1f' % mi)
pl.axis('tight')
pl.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)
    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, compute_importances=True).fit(X, y)
# Compare with F-score
F, _ = f_regression(X, y)

pl.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)
pl.semilogy(np.maximum(score / np.max(score), 1e-4),
            label="%s. AUC: %.3f" % (name, auc(recall, precision)))

pl.plot(np.where(coef != 0)[0], [2e-4] * n_relevant_features, 'mo',
        label="Ground truth")
pl.xlabel("Features")
pl.ylabel("Score")
# Plot only the 100 first coefficients
pl.xlim(0, 100)
pl.legend(loc='best')
pl.title('Feature selection scores - Mutual incoherence: %.1f'
          % mi)

pl.show()

```

**Total running time of the example:** 7.25 seconds

Figure 2.120: *Lasso on dense and sparse data*

## 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.todense(), y)
print "Dense Lasso done in %fs" % (time() - t0)

print("Distance between coefficients : %s"
      % linalg.norm(sparse_lasso.coef_ - dense_lasso.coef_))
```

## 2.1.11 Manifold learning

Examples concerning the `sklearn.manifold` package.

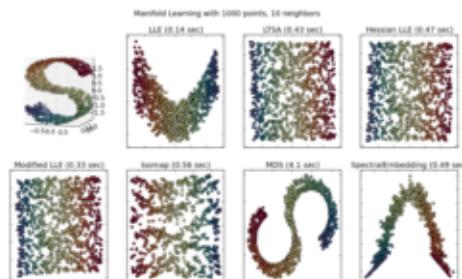


Figure 2.121: Comparison of Manifold Learning methods

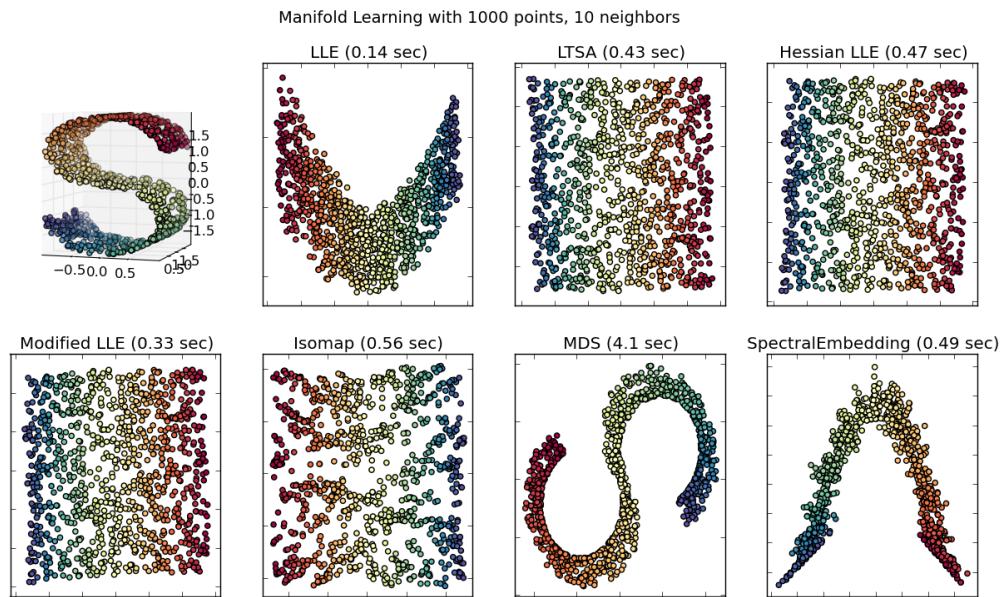
## Comparison of Manifold Learning methods

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 seek an isotropic representation of the data in the low-dimensional space.



### Script output:

```
standard: 0.14 sec
ltsa: 0.43 sec
hessian: 0.47 sec
modified: 0.33 sec
Isomap: 0.56 sec
MDS: 4.1 sec
SpectralEmbedding: 0.49 sec
```

**Python source code:** [plot\\_compare\\_methods.py](#)

# Author: Jake Vanderplas -- <vanderplas@astro.washington.edu>

```
print __doc__

from time import time

import pylab as pl
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 = pl.figure(figsize=(15, 8))
pl.suptitle("Manifold Learning with %i points, %i neighbors"
            % (1000, n_neighbors), fontsize=14)

try:
    # compatibility matplotlib < 1.0
    ax = fig.add_subplot(241, projection='3d')
    ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=pl.cm.Spectral)
    ax.view_init(4, -72)
except:
    ax = fig.add_subplot(241, projection='3d')
    pl.scatter(X[:, 0], X[:, 2], c=color, cmap=pl.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(242 + i)
    pl.scatter(Y[:, 0], Y[:, 1], c=color, cmap=pl.cm.Spectral)
    pl.title("%s (%.2g sec)" % (labels[i], t1 - t0))
    ax.xaxis.set_major_formatter(NullFormatter())
    ax.yaxis.set_major_formatter(NullFormatter())
    pl.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(246)
pl.scatter(Y[:, 0], Y[:, 1], c=color, cmap=pl.cm.Spectral)
pl.title("Isomap (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
pl.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(247)
pl.scatter(Y[:, 0], Y[:, 1], c=color, cmap=pl.cm.Spectral)
```

```

pl.title("MDS (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
pl.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(248)
pl.scatter(Y[:, 0], Y[:, 1], c=color, cmap=pl.cm.Spectral)
pl.title("SpectralEmbedding (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
pl.axis('tight')

pl.show()

```

**Total running time of the example:** 7.13 seconds

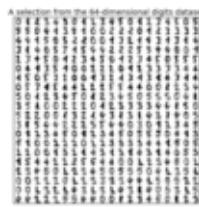
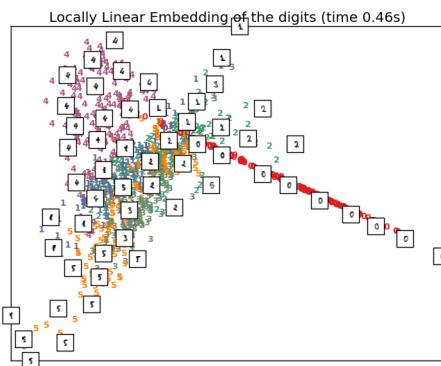


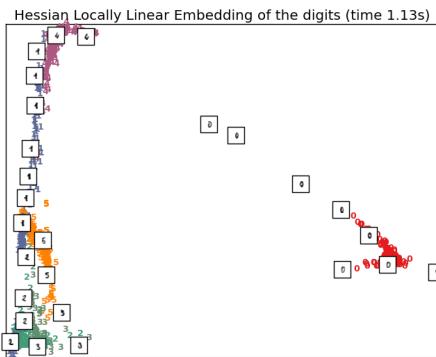
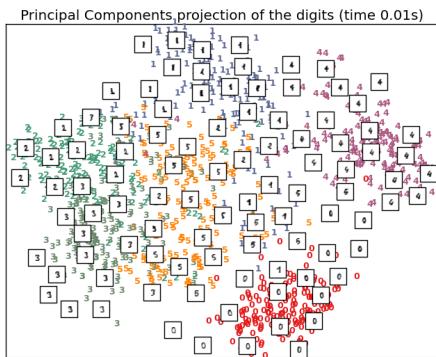
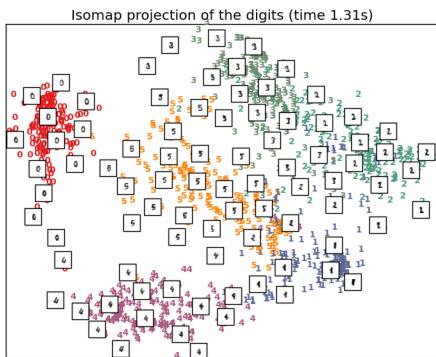
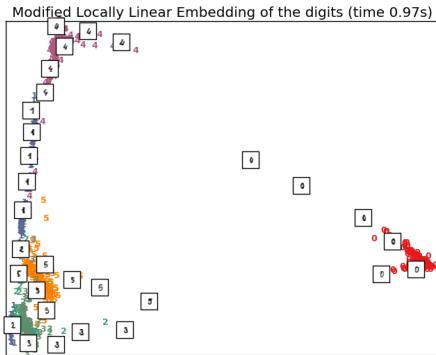
Figure 2.122: *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

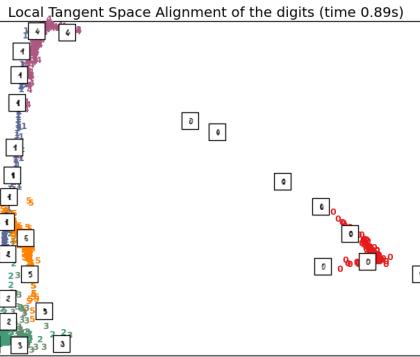
## Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...

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.





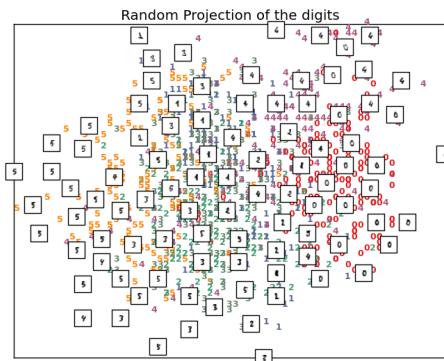
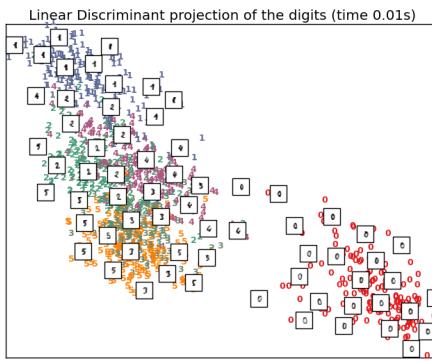


A selection from the 64-dimensional digits dataset

```

0 1 2 3 4 5 0 1 2 3 4 5 0 5
5 5 0 4 1 3 5 1 0 0 2 2 2 0 1 2 3 3 3 3
4 4 1 5 0 5 2 2 0 0 1 3 2 1 4 3 1 3 1 4
3 1 4 0 5 3 1 5 4 5 2 2 2 5 5 4 4 0 0 1
2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 5 5
0 4 1 3 5 4 0 0 2 2 1 0 1 2 3 3 3 3 4 4
1 5 0 5 2 2 0 0 1 3 2 1 3 1 3 4 4 3 4 4
0 5 7 4 5 4 4 1 2 5 5 4 4 0 0 2 2 3 9
5 0 4 2 3 4 5 0 4 2 3 4 5 0 5 5 5 0 4 1
3 5 4 0 0 2 2 2 0 4 2 3 3 3 3 3 4 4 1 5 0
5 2 2 0 0 4 3 2 4 4 3 4 3 1 6 3 1 3 0 5 5
3 8 5 4 4 2 2 2 5 5 4 4 0 3 0 1 2 3 4 5
0 1 2 3 4 5 0 4 2 3 4 5 0 5 5 5 0 4 1 3
5 1 0 0 1 2 2 0 1 3 3 3 3 3 4 4 1 5 0 5
1 2 0 0 1 3 2 1 4 3 1 3 1 4 3 1 4 0 5 3
1 5 4 4 2 2 2 5 5 4 4 0 0 1 2 3 4 5 0 1
2 3 4 5 0 1 2 3 4 5 0 5 5 5 5 0 4 1 3 5 4
0 0 2 2 2 0 1 2 3 3 3 3 4 4 4 5 0 5 5 2 2
0 0 1 3 1 4 3 1 1 4 3 1 4 0 5 5 3 1 5
4 4 1 2 1 5 5 4 4 0 0 1 2 3 4 5 0 1 2 3

```



**Script output:**

```
Computing random projection
Computing PCA projection
Computing LDA projection
Computing Isomap embedding
Done.
Computing LLE embedding
Done. Reconstruction error: 1.28542e-06
Computing modified LLE embedding
Done. Reconstruction error: 0.36
Computing Hessian LLE embedding
Done. Reconstruction error: 0.211995
Computing LTSA embedding
Done. Reconstruction error: 0.212075
Computing MDS embedding
Done. Stress: 141100635.430416
Computing Totally Random Trees embedding
Computing Spectral embedding
```

**Python source code:** plot\_lle\_digits.py

```
# Authors: Fabian Pedregosa <fabian.pedregosa@inria.fr>
#          Olivier Grisel <olivier.grisel@ensta.org>
#          Mathieu Blondel <mathieu@mblondel.org>
#          Gael Varoquaux
# License: BSD, (C) INRIA 2011

print __doc__
from time import time

import numpy as np
import pylab as pl
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)

    pl.figure()
    ax = pl.subplot(111)
    for i in range(X.shape[0]):
        pl.text(X[i, 0], X[i, 1], str(digits.target[i]),
                color=pl.cm.Set1(y[i] / 10.),
                fontdict={'weight': 'bold', 'size': 9})

    if hasattr(offsetbox, 'AnnotationBbox'):
        # only print thumbnails with matplotlib > 1.0
        pass
```

```

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=pl.cm.gray_r),
        X[i])
    ax.add_artist(imagebox)
pl.xticks([]), pl.yticks([])
if title is not None:
    pl.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))

pl.imshow(img, cmap=pl.cm.binary)
pl.xticks([])
pl.yticks([])
pl.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.RandomizedPCA(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.RandomizedPCA(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))

pl.show()

```

**Total running time of the example:** 13.59 seconds

## 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.

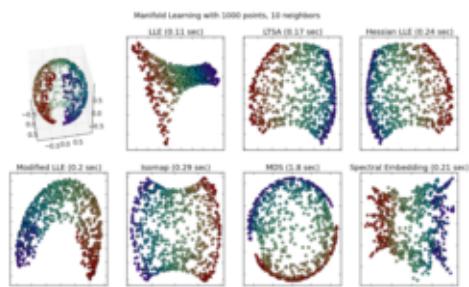
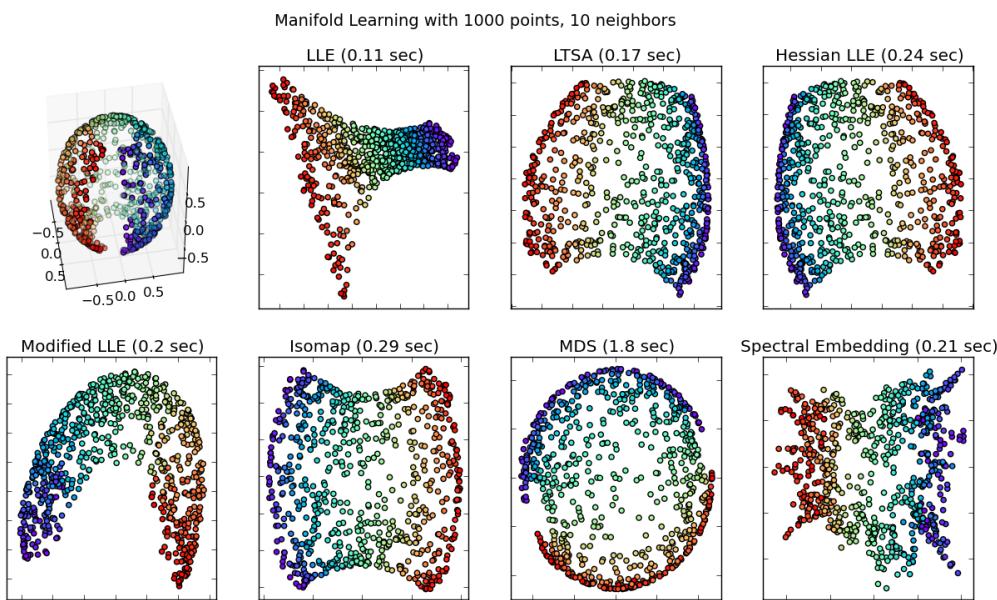


Figure 2.123: *Manifold Learning methods on a severed sphere*

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 seek 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: 0.11 sec
ltsa: 0.17 sec
hessian: 0.24 sec
modified: 0.2 sec
ISO: 0.29 sec
MDS: 1.8 sec
Spectral Embedding: 0.21 sec
```

**Python source code:** `plot_manifold_sphere.py`

```

# Author: Jaques Grobler <jaques.grobler@inria.fr>
# License: BSD

print __doc__

from time import time

import numpy as np
import pylab as pl
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 = pl.figure(figsize=(15, 8))
pl.suptitle("Manifold Learning with %i points, %i neighbors"
            % (1000, n_neighbors), fontsize=14)

ax = fig.add_subplot(241, projection='3d')
ax.scatter(x, y, z, c=p[indices], cmap=pl.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(242 + i)
pl.scatter(trans_data[0], trans_data[1], c=colors, cmap=pl.cm.rainbow)
pl.title("%s (%.2g sec)" % (labels[i], t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
pl.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 "ISO: %.2g sec" % (t1 - t0)

ax = fig.add_subplot(246)
pl.scatter(trans_data[0], trans_data[1], c=colors, cmap=pl.cm.rainbow)
pl.title("%s (%.2g sec)" % ('Isomap', t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
pl.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)

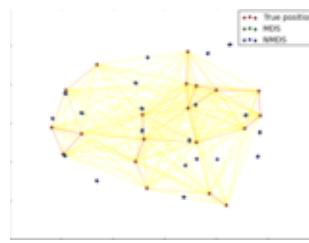
ax = fig.add_subplot(247)
pl.scatter(trans_data[0], trans_data[1], c=colors, cmap=pl.cm.rainbow)
pl.title("MDS (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
pl.axis('tight')

# 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(248)
pl.scatter(trans_data[0], trans_data[1], c=colors, cmap=pl.cm.rainbow)
pl.title("Spectral Embedding (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
pl.axis('tight')

pl.show()
```

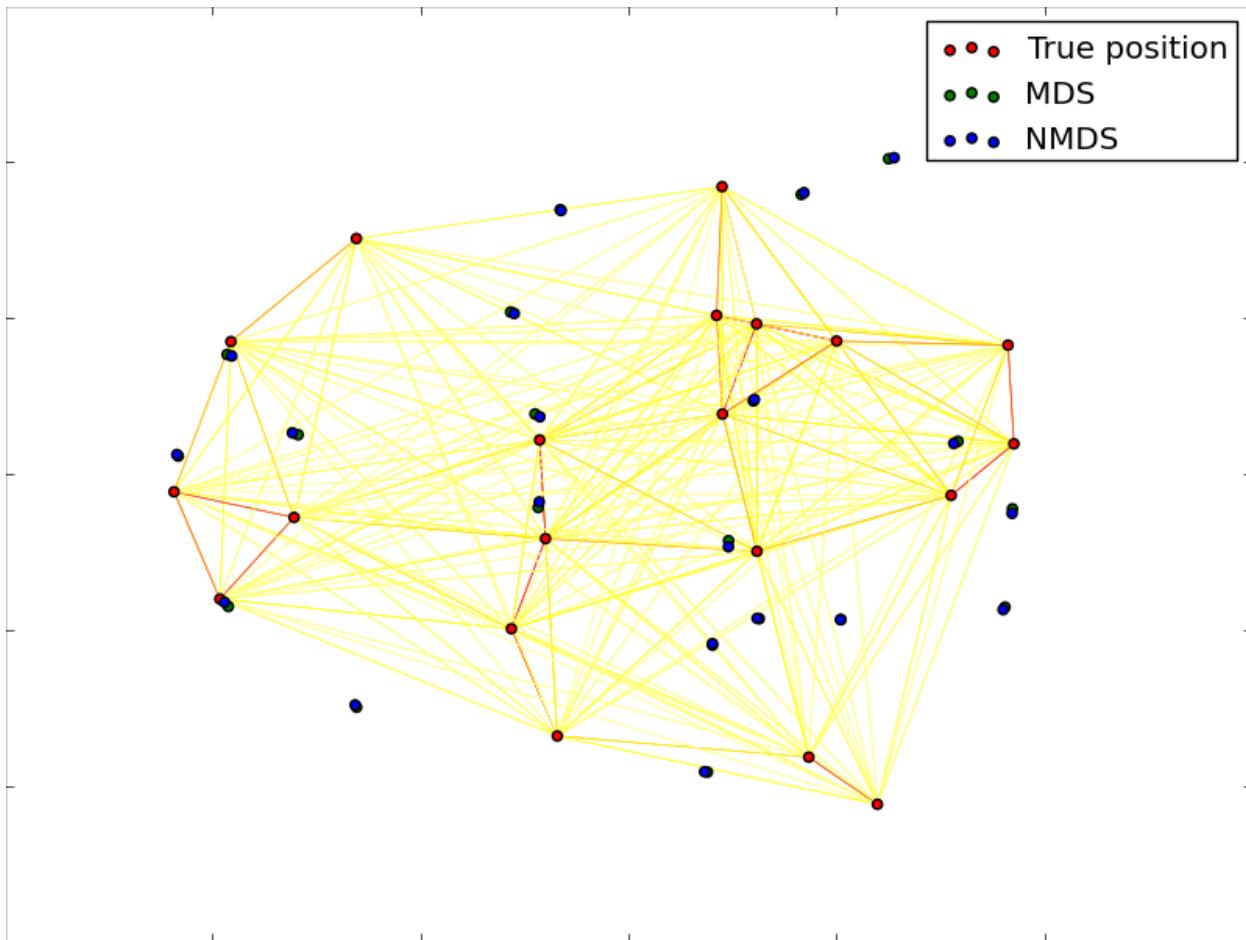
**Total running time of the example:** 3.63 seconds

Figure 2.124: *Multi-dimensional scaling*

### 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_

nmuds = 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 = nmuds.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:** 0.25 seconds

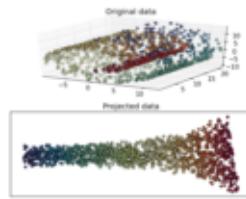
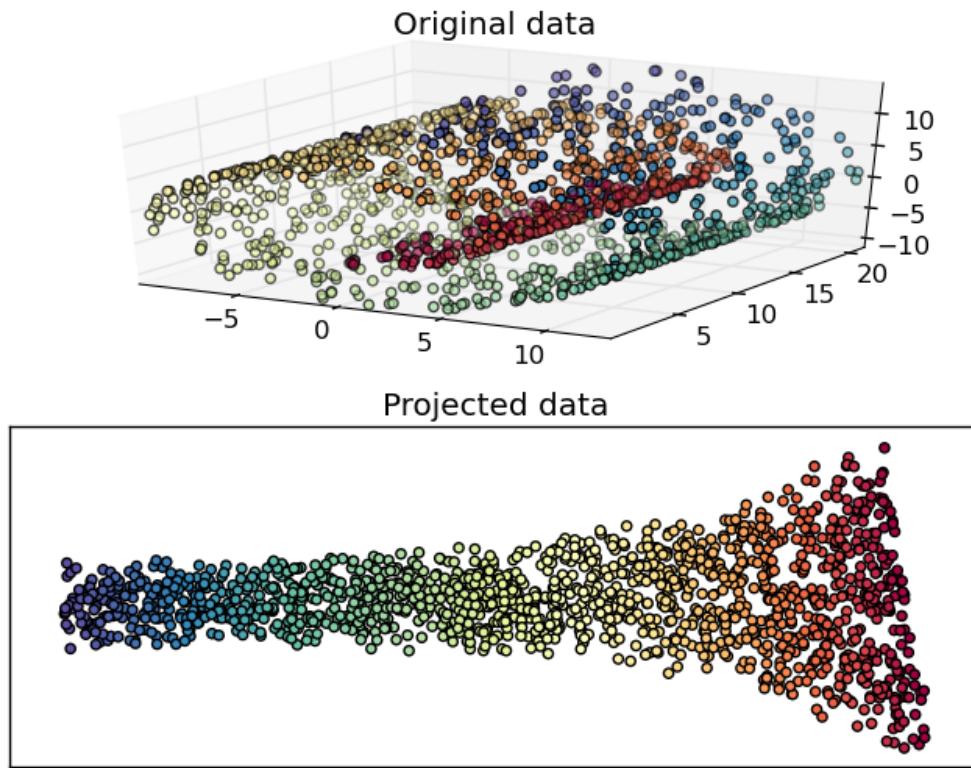


Figure 2.125: *Swiss Roll reduction with LLE*

## Swiss Roll reduction with LLE

An illustration of Swiss Roll reduction with locally linear embedding

**Script output:**

```
Computing LLE embedding
Done. Reconstruction error: 8.67941e-08
```

**Python source code:** [plot\\_swissroll.py](#)

```
# Author: Fabian Pedregosa -- <fabian.pedregosa@inria.fr>
# License: BSD, (C) INRIA 2011
```

```
print __doc__

import pylab as pl

# 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 = pl.figure()
try:
    # compatibility matplotlib < 1.0
    ax = fig.add_subplot(211, projection='3d')
    ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=pl.cm.Spectral)
except:
    ax = fig.add_subplot(211)
    ax.scatter(X[:, 0], X[:, 2], c=color, cmap=pl.cm.Spectral)

ax.set_title("Original data")
ax = fig.add_subplot(212)
ax.scatter(X_r[:, 0], X_r[:, 1], c=color, cmap=pl.cm.Spectral)
pl.axis('tight')
pl.xticks([]), pl.yticks([])
pl.title('Projected data')
pl.show()

```

**Total running time of the example:** 0.40 seconds

## 2.1.12 Gaussian Mixture Models

Examples concerning the `sklearn.mixture` package.

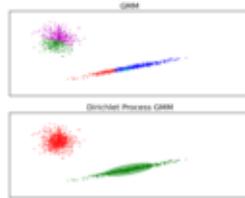


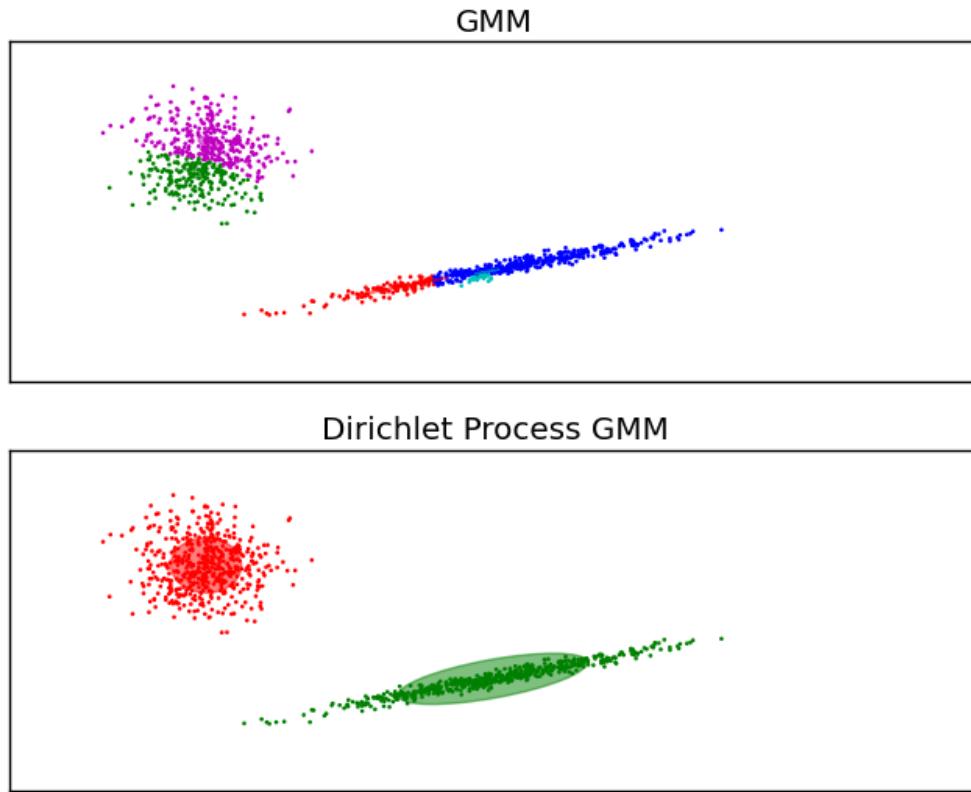
Figure 2.126: Gaussian Mixture Model Ellipsoids

### 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 its 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 pylab as pl
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 = pl.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
        pl.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)

    pl.xlim(-10, 10)
    pl.ylim(-3, 6)
    pl.xticks(())
    pl.yticks(())
    pl.title(title)

pl.show()

```

**Total running time of the example:** 0.81 seconds

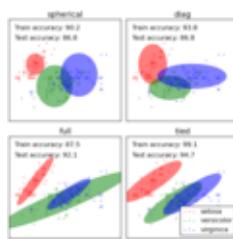


Figure 2.127: GMM classification

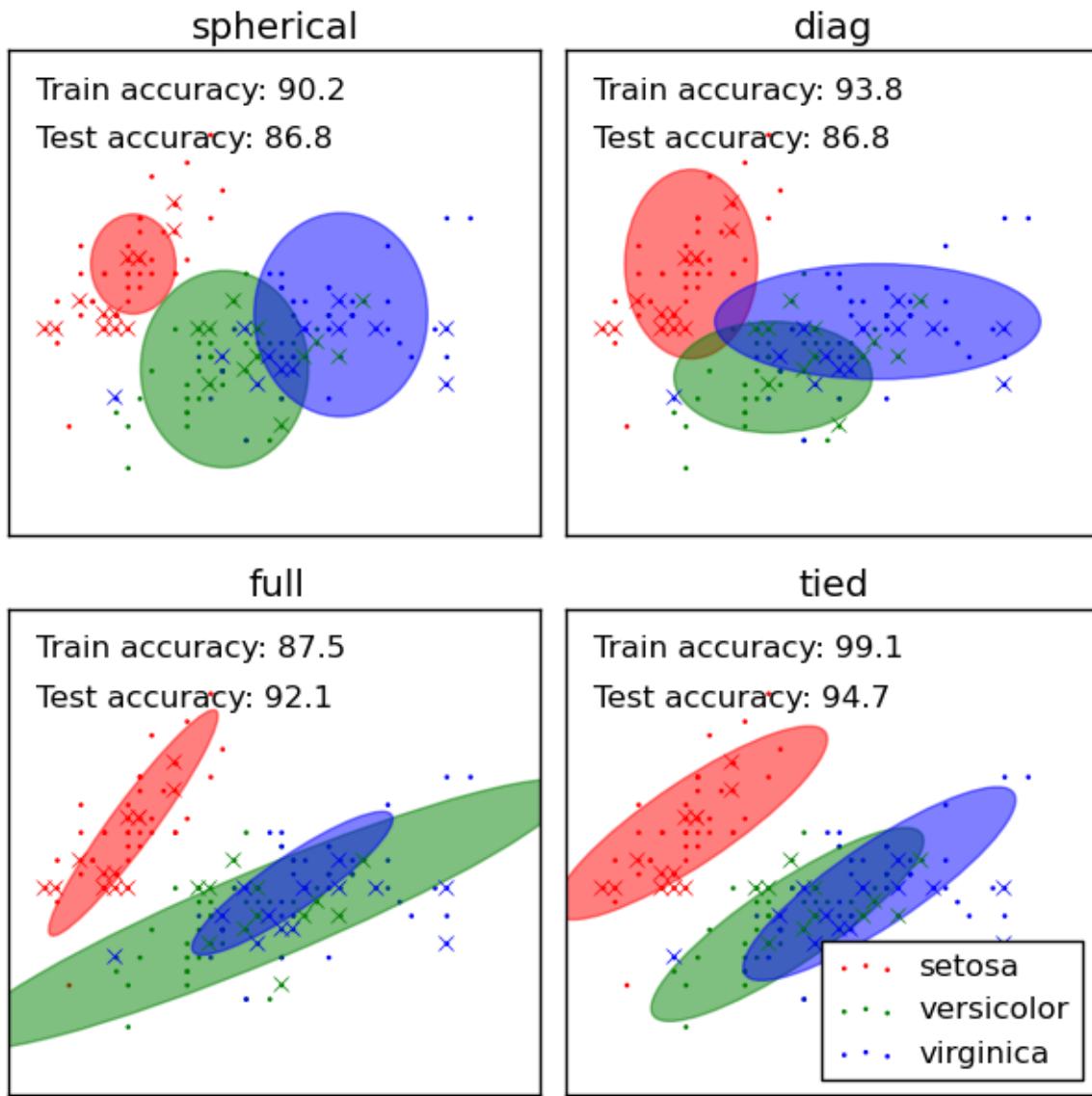
## GMM classification

Demonstration of *Gaussian mixture models* for classification.

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](#)

```
print __doc__

# Author: Ron Weiss <ronweiss@gmail.com>, Gael Varoquaux
# License: BSD Style.

# $Id$

import pylab as pl
import matplotlib as mpl
import numpy as np

from sklearn import datasets
from sklearn.cross_validation import StratifiedKFold
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)

pl.figure(figsize=(3 * n_classifiers / 2, 6))
pl.subplots_adjust(bottom=.01, top=0.95, hspace=.15, wspace=.05,
                   left=.01, right=.99)

for index, (name, classifier) in enumerate(classifiers.iteritems()):
    # 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 = pl.subplot(2, n_classifiers / 2, index + 1)
    make_ellipses(classifier, h)

    for n, color in enumerate('rgb'):
        data = iris.data[iris.target == n]
        pl.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]
    pl.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
pl.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
pl.text(0.05, 0.8, 'Test accuracy: %.1f' % test_accuracy,
        transform=h.transAxes)

pl.xticks(())
pl.yticks(())
pl.title(name)

pl.legend(loc='lower right', prop=dict(size=12))

pl.show()
```

**Total running time of the example:** 0.32 seconds

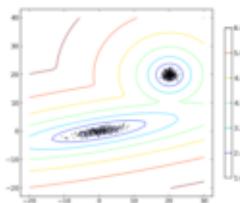
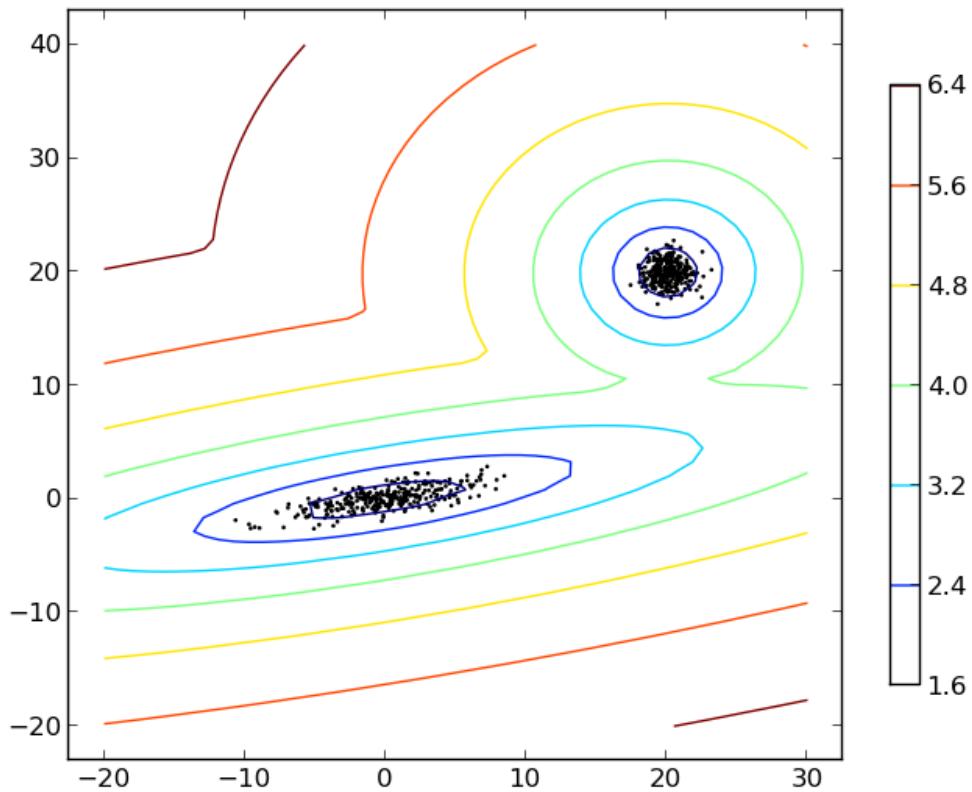


Figure 2.128: Density Estimation for a mixture of Gaussians

### 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 pylab as pl
from sklearn import mixture

n_samples = 300

# generate random sample, two components
np.random.seed(0)
C = np.array([[0., -0.7], [3.5, .7]])
X_train = np.r_[np.dot(np.random.randn(n_samples, 2), C),
               np.random.randn(n_samples, 2) + np.array([20, 20])]

clf = mixture.GMM(n_components=2, covariance_type='full')
clf.fit(X_train)

x = np.linspace(-20.0, 30.0)
y = np.linspace(-20.0, 40.0)
X, Y = np.meshgrid(x, y)
XX = np.c_[X.ravel(), Y.ravel()]
Z = np.log(-clf.eval(XX)[0])
Z = Z.reshape(X.shape)

CS = pl.contour(X, Y, Z)
CB = pl.colorbar(CS, shrink=0.8, extend='both')
```

```
pl.scatter(X_train[:, 0], X_train[:, 1], .8)

pl.axis('tight')
pl.show()
```

**Total running time of the example:** 0.20 seconds

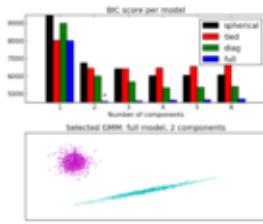
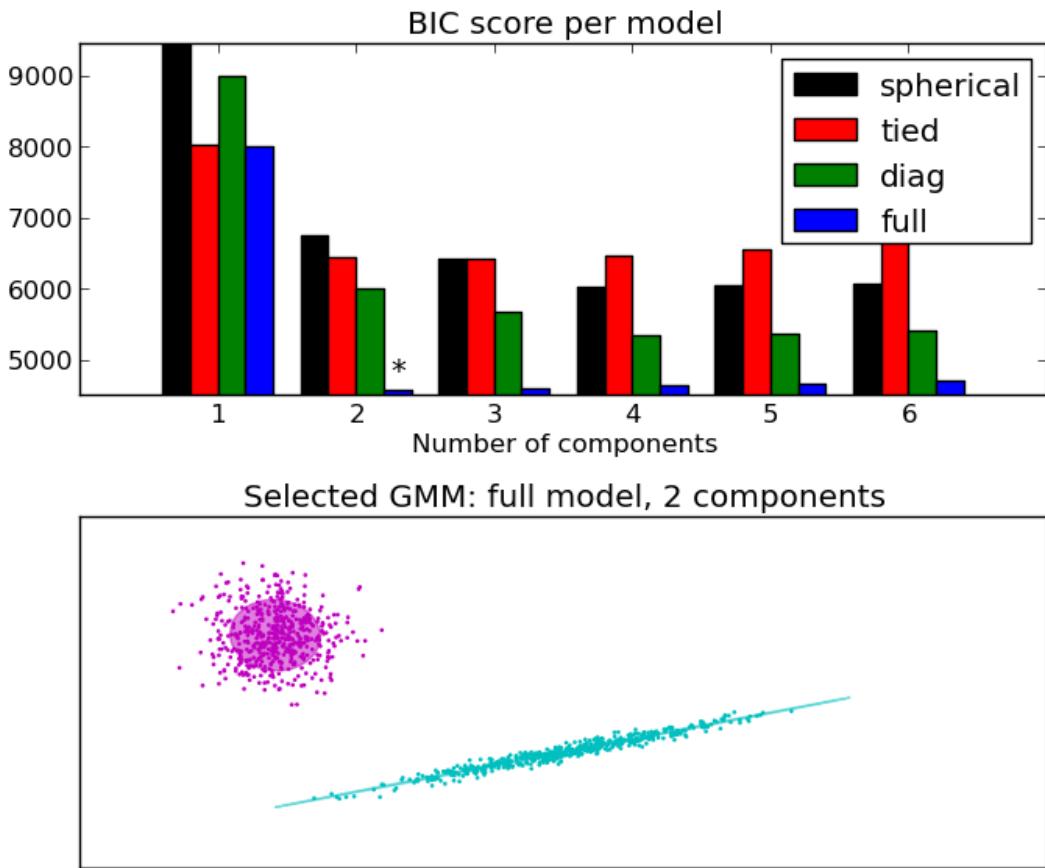


Figure 2.129: *Gaussian Mixture Model Selection*

## 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 pylab as pl
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 = pl.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(pl.bar(xpos, bic[i * len(n_components_range):
                                  (i + 1) * len(n_components_range)],
                        width=.2, color=color))
pl.xticks(n_components_range)
pl.ylim([bic.min() * 1.01 - .01 * bic.max(), bic.max()])
pl.title('BIC score per model')
xpos = np.mod(bic.argmax(), len(n_components_range)) + .65 +\
    .2 * np.floor(bic.argmax() / len(n_components_range))
pl.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 = pl.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
    pl.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)

pl.xlim(-10, 10)
pl.ylim(-3, 6)
pl.xticks(())
pl.yticks(())
pl.title('Selected GMM: full model, 2 components')
pl.subplots_adjust(hspace=.35, bottom=.02)
pl.show()
```

**Total running time of the example:** 3.42 seconds

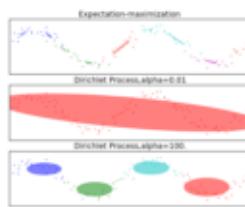
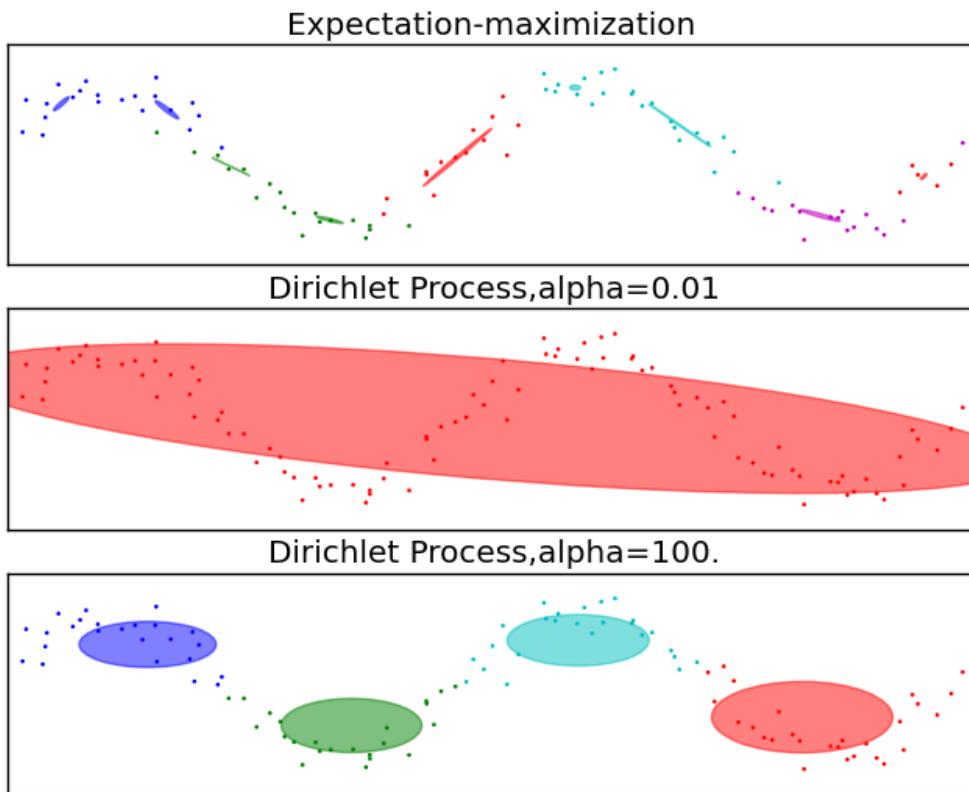


Figure 2.130: *Gaussian Mixture Model Sine Curve*

### 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.



**Python source code:** [plot\\_gmm\\_sin.py](#)

```
import itertools

import numpy as np
from scipy import linalg
import pylab as pl
import matplotlib as mpl

from sklearn import mixture

# 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 = pl.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
        pl.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)
```

```

pl.xlim(-6, 4 * np.pi - 6)
pl.ylim(-5, 5)
pl.title(title)
pl.xticks(())
pl.yticks(())

pl.show()

```

**Total running time of the example:** 0.46 seconds

### 2.1.13 Nearest Neighbors

Examples concerning the `sklearn.neighbors` package.

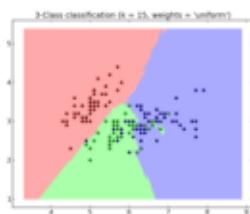
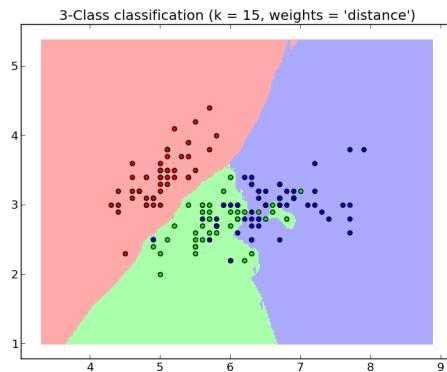
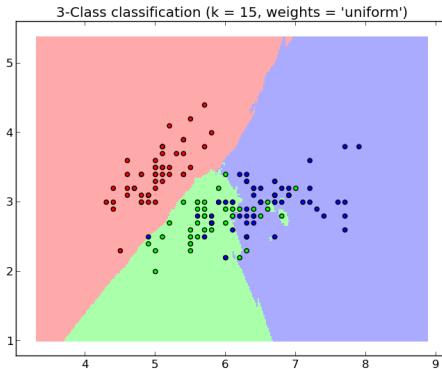


Figure 2.131: *Nearest Neighbors Classification*

#### 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__\n\nimport numpy as np\nimport pylab as pl\nfrom matplotlib.colors import ListedColormap\nfrom sklearn import neighbors, datasets\n\nn_neighbors = 15\n\n# import some data to play with\niris = datasets.load_iris()\nX = iris.data[:, :2] # we only take the first two features. We could\n                     # avoid this ugly slicing by using a two-dim dataset\ny = iris.target\n\nh = .02 # step size in the mesh\n\n# Create color maps\ncmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])\ncmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])\n\nfor weights in ['uniform', 'distance']:\n    # we create an instance of Neighbours Classifier and fit the data.\n    clf = neighbors.KNeighborsClassifier(n_neighbors, weights=weights)\n    clf.fit(X, y)\n\n    # Plot the decision boundary. For that, we will assign a color to each\n    # point in the mesh [x_min, m_max]x[y_min, y_max].\n    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1\n    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1\n    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),\n                         np.arange(y_min, y_max, h))\n    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])\n\n    # Put the result into a color plot\n    Z = Z.reshape(xx.shape)\n    pl.figure()\n    pl.pcolormesh(xx, yy, Z, cmap=cmap_light)\n\n    # Plot also the training points\n    pl.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold)\n    pl.title("3-Class classification (k = %i, weights = '%s')"\n            % (n_neighbors, weights))
```

```
% (n_neighbors, weights))
pl.axis('tight')

pl.show()
```

**Total running time of the example:** 1.82 seconds

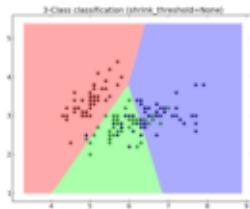
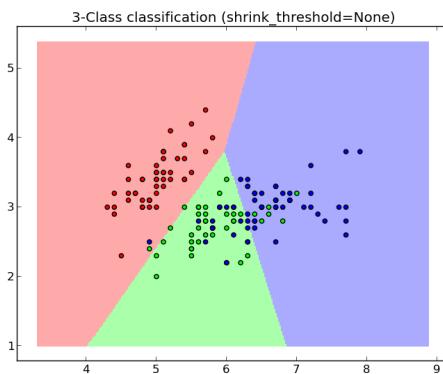


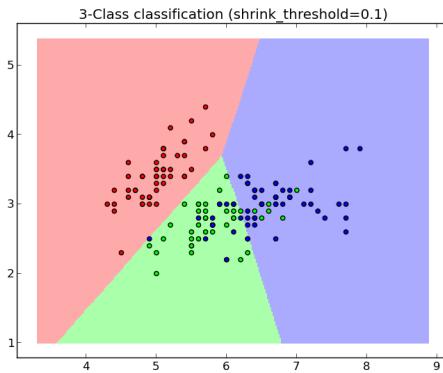
Figure 2.132: *Nearest Centroid Classification*

## 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 0.826666666667
```

**Python source code:** [plot\\_nearest\\_centroid.py](#)

```
print __doc__\n\nimport numpy as np\nimport pylab as pl\nfrom matplotlib.colors import ListedColormap\nfrom sklearn import datasets\nfrom sklearn.neighbors import NearestCentroid\n\nn_neighbors = 15\n\n# import some data to play with\niris = datasets.load_iris()\nX = iris.data[:, :2] # we only take the first two features. We could\n# avoid this ugly slicing by using a two-dim dataset\ny = iris.target\n\nh = .02 # step size in the mesh\n\n# Create color maps\ncmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])\ncmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])\n\nfor shrinkage in [None, 0.1]:\n    # we create an instance of Neighbours Classifier and fit the data.\n    clf = NearestCentroid(shrink_threshold=shrinkage)\n    clf.fit(X, y)\n    y_pred = clf.predict(X)\n    print shrinkage, np.mean(y == y_pred)\n    # Plot the decision boundary. For that, we will assign a color to each\n    # point in the mesh [x_min, m_max]x[y_min, y_max].\n    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1\n    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1\n    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),\n                         np.arange(y_min, y_max, h))\n    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])\n\n    # Put the result into a color plot\n    Z = Z.reshape(xx.shape)\n    pl.figure()\n    pl.pcolormesh(xx, yy, Z, cmap=cmap_light)\n\n    # Plot also the training points\n    pl.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold)\n    pl.title("3-Class classification (shrink_threshold=%r)"\n             % shrinkage)\n    pl.axis('tight')\n\npl.show()
```

**Total running time of the example:** 0.19 seconds

## 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.

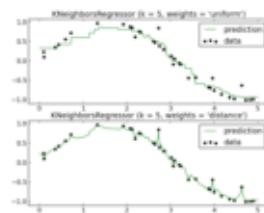
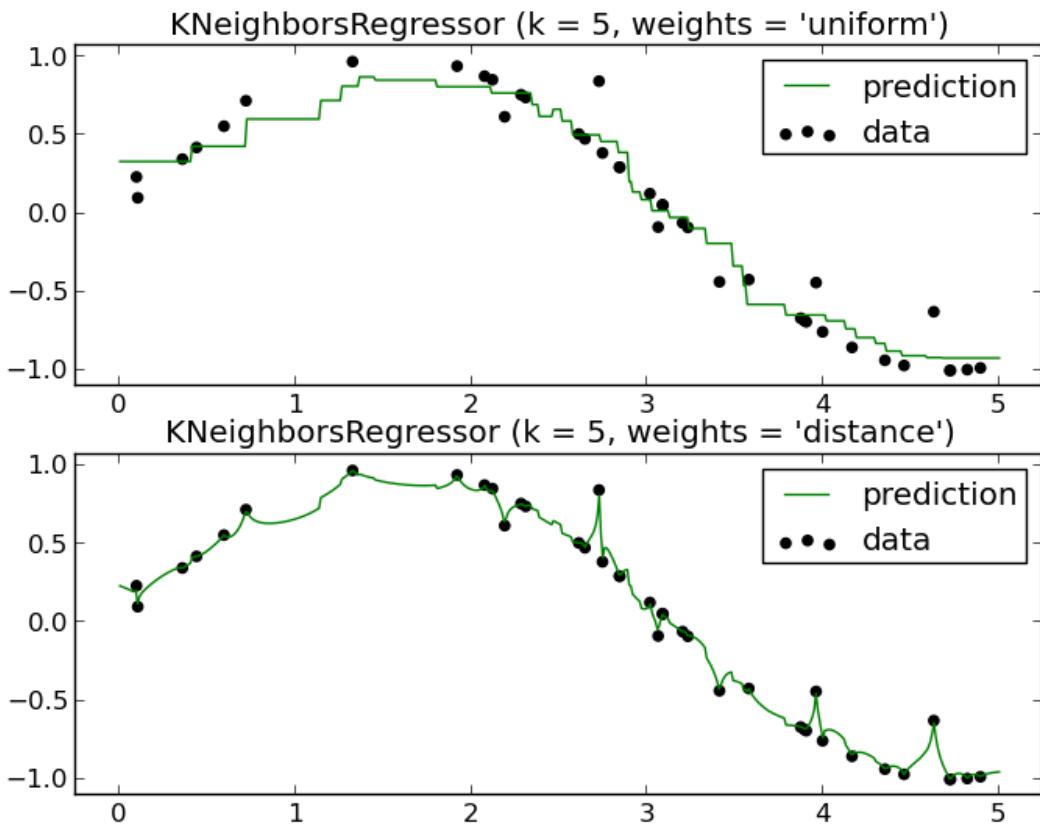


Figure 2.133: Nearest Neighbors regression



**Python source code:** [plot\\_regression.py](#)

```
print __doc__

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#         Fabian Pedregosa <fabian.pedregosa@inria.fr>
#
# License: BSD, (C) INRIA

#####
# Generate sample data
import numpy as np
import pylab as pl
```

```
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)

    pl.subplot(2, 1, i + 1)
    pl.scatter(X, y, c='k', label='data')
    pl.plot(T, y_, c='g', label='prediction')
    pl.axis('tight')
    pl.legend()
    pl.title("KNeighborsRegressor (k = %i, weights = '%s')" % (n_neighbors,
                                                               weights))

pl.show()
```

Total running time of the example: 0.22 seconds

## 2.1.14 Semi Supervised Classification

Examples concerning the `sklearn.semi_supervised` package.

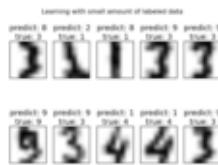


Figure 2.134: Label Propagation digits: Demonstrating performance

### Label Propagation digits: Demonstrating performance

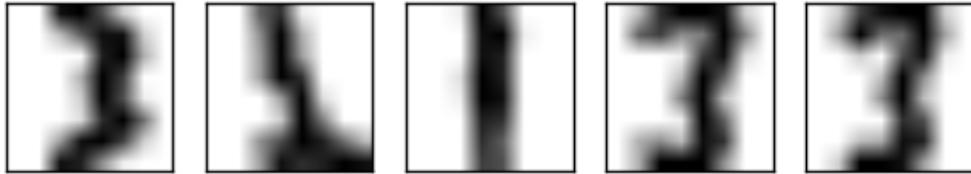
This example demonstrates the power of semisupervised learning by training a Label Spreading model to classify handwritten digits with sets of very few labels.

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.

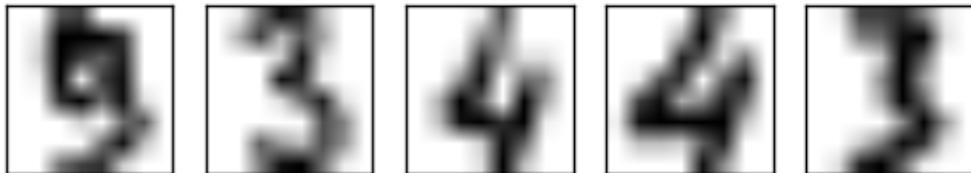
At the end, the top 10 most uncertain predictions will be shown.

### Learning with small amount of labeled data

predict: 8 predict: 2 predict: 8 predict: 9 predict: 9  
true: 3 true: 1 true: 1 true: 3 true: 3



predict: 9 predict: 9 predict: 1 predict: 1 predict: 9  
true: 9 true: 3 true: 4 true: 4 true: 3



#### Script output:

```
Label Spreading model: 30 labeled & 300 unlabeled points (330 total)
      precision    recall   f1-score   support
```

	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 pylab as pl

from scipy import stats

from sklearn import datasets
from sklearn.semi_supervised import label_propagation

from sklearn.metrics import metrics
from sklearn.metrics.metrics import 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 = 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 metrics.classification_report(true_labels, predicted_labels)

print "Confusion matrix"
print cm

# calculate uncertainty values for each transduced distribution
pred_entropies = stats.distributions.entropy(lp_model.label_distributions_.T)

# pick the top 10 most uncertain labels
```

```

uncertainty_index = np.argsort(pred_entropies)[-10:]

#####
# plot
f = pl.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=pl.cm.gray_r)
    pl.xticks([])
    pl.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')
pl.show()

```

**Total running time of the example:** 0.64 seconds

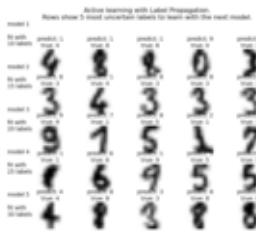


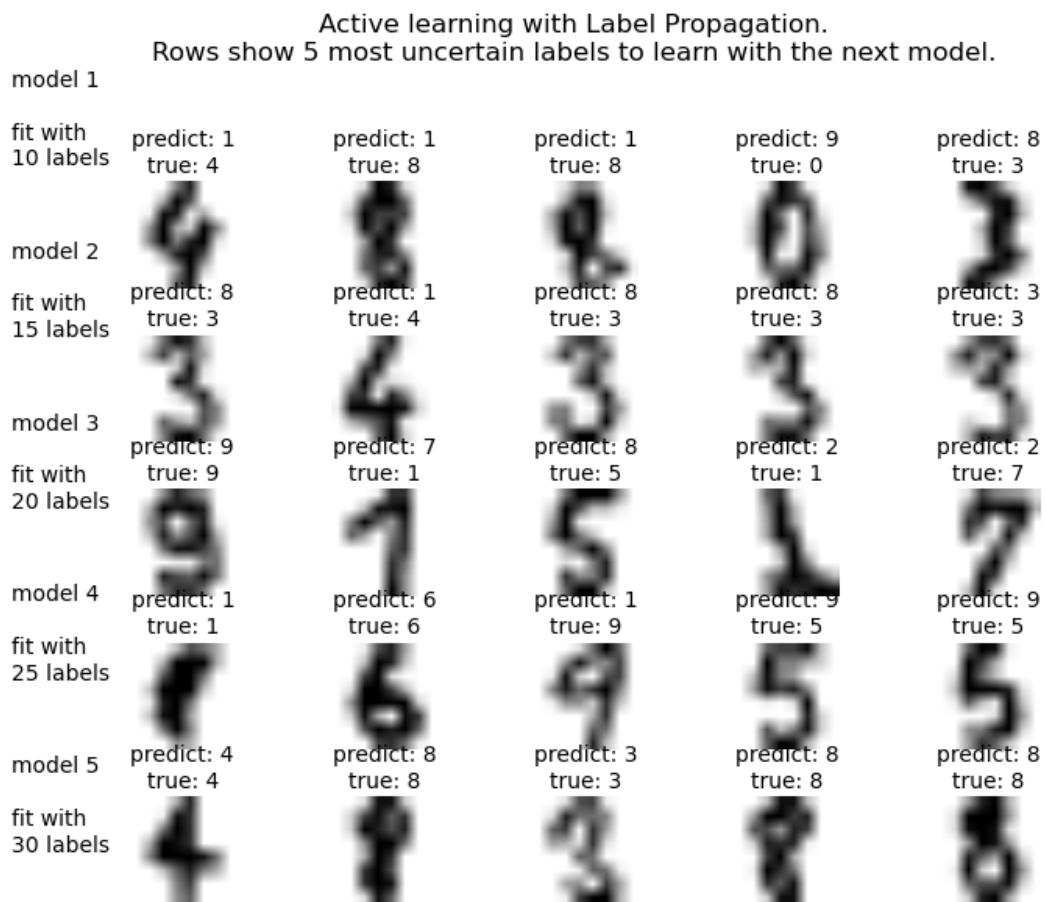
Figure 2.135: *Label Propagation digits active learning*

## 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 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 pylab as pl
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 = pl.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 ' + 70 * '_') % i
    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%fit 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=pl.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.")
pl.subplots_adjust(0.12, 0.03, 0.9, 0.8, 0.2, 0.45)
pl.show()
```

**Total running time of the example:** 1.55 seconds

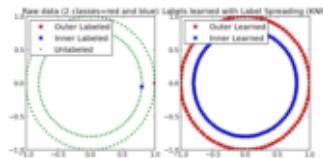
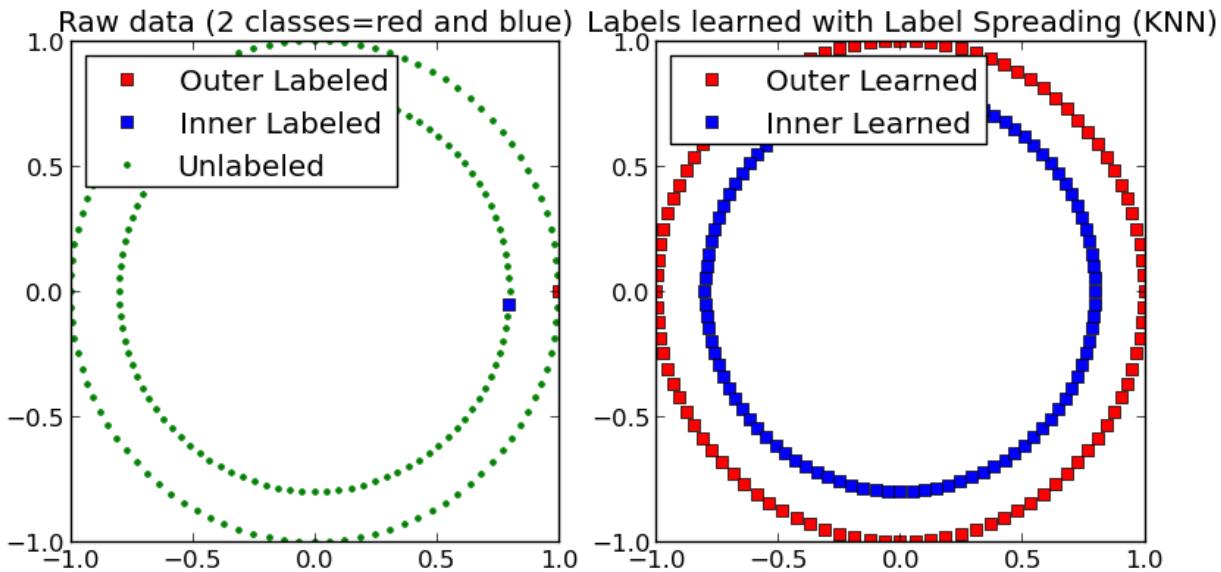


Figure 2.136: *Label Propagation learning a complex structure*

### 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.



**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

import numpy as np
import pylab as pl
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_
pl.figure(figsize=(8.5, 4))
pl.subplot(1, 2, 1)
plot_outer_labeled, = pl.plot(X[labels == outer, 0],
                             X[labels == outer, 1], 'rs')
plot_unlabeled, = pl.plot(X[labels == -1, 0], X[labels == -1, 1], 'g.')
plot_inner_labeled, = pl.plot(X[labels == inner, 0],
                             X[labels == inner, 1], 'bs')
pl.legend((plot_outer_labeled, plot_inner_labeled, plot_unlabeled),
          ('Outer Labeled', 'Inner Labeled', 'Unlabeled'), 'upper left',
```

```
    numpoints=1, shadow=False)
pl.title("Raw data (2 classes=red and blue)")

pl.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, = pl.plot(X[outer_numbers, 0], X[outer_numbers, 1], 'rs')
plot_inner, = pl.plot(X[inner_numbers, 0], X[inner_numbers, 1], 'bs')
pl.legend((plot_outer, plot_inner), ('Outer Learned', 'Inner Learned'),
          'upper left', numpoints=1, shadow=False)
pl.title("Labels learned with Label Spreading (KNN)")

pl.subplots_adjust(left=0.07, bottom=0.07, right=0.93, top=0.92)
pl.show()
```

**Total running time of the example:** 0.27 seconds

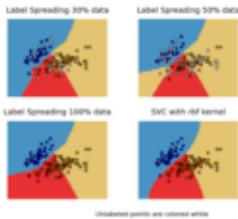


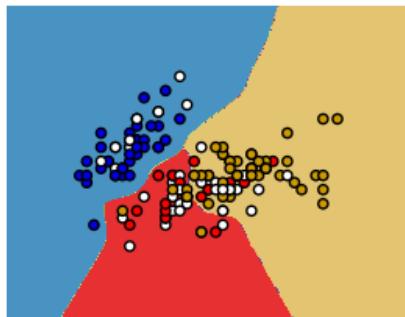
Figure 2.137: Decision boundary of label propagation versus SVM on the Iris dataset

### Decision boundary of label propagation versus SVM on the Iris dataset

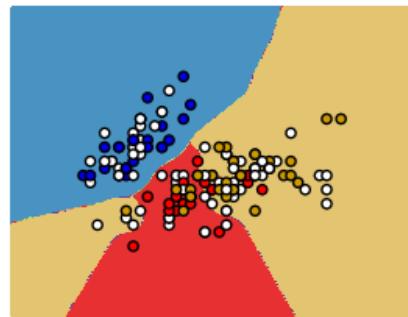
Comparison for decision boundary generated on iris dataset between Label Propagation and SVM.

This demonstrates Label Propagation learning a good boundary even with a small amount of labeled data.

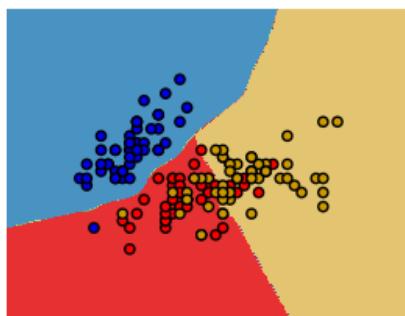
Label Spreading 30% data



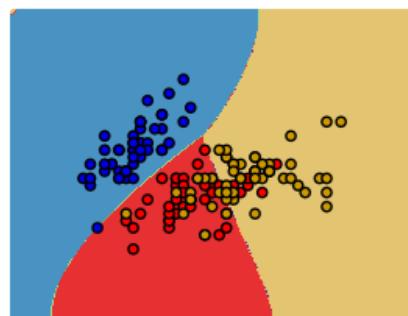
Label Spreading 50% data



Label Spreading 100% data



SVC with rbf kernel



Unlabeled points are colored white

**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 pylab as pl
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].
    pl.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)
    pl.contourf(xx, yy, Z, cmap=pl.cm.Paired)
    pl.axis('off')

    # Plot also the training points
    colors = [color_map[y] for y in y_train]
    pl.scatter(X[:, 0], X[:, 1], c=colors, cmap=pl.cm.Paired)

    pl.title(titles[i])

pl.text(.90, 0, "Unlabeled points are colored white")
pl.show()
```

**Total running time of the example:** 1.99 seconds

## 2.1.15 Support Vector Machines

Examples concerning the `sklearn.svm` package.

### SVM with custom kernel

Simple usage of Support Vector Machines to classify a sample. It will plot the decision surface and the support vectors.

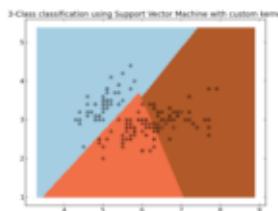
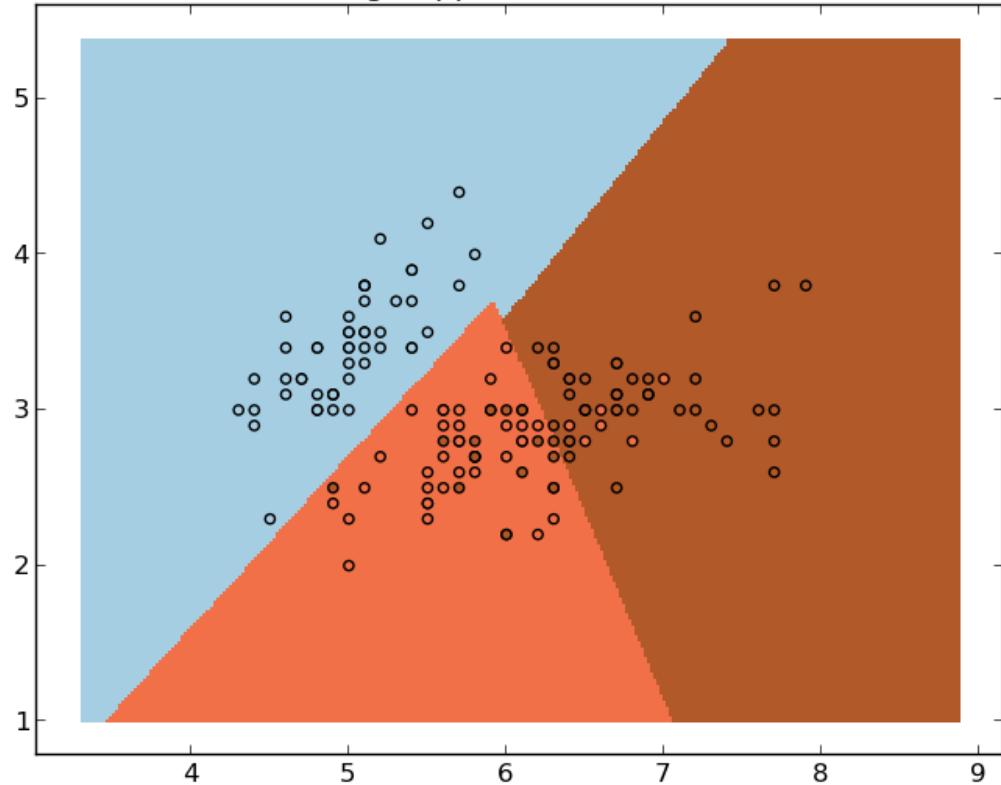


Figure 2.138: SVM with custom kernel

### 3-Class classification using Support Vector Machine with custom kernel



**Python source code:** [plot\\_custom\\_kernel.py](#)

```
print __doc__

import numpy as np
import pylab as pl
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)
pl.pcolormesh(xx, yy, Z, cmap=pl.cm.Paired)

# Plot also the training points
pl.scatter(X[:, 0], X[:, 1], c=Y, cmap=pl.cm.Paired)
pl.title('3-Class classification using Support Vector Machine with custom'
          ' kernel')
pl.axis('tight')
pl.show()
```

**Total running time of the example:** 0.23 seconds

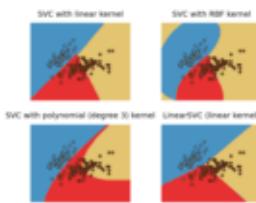
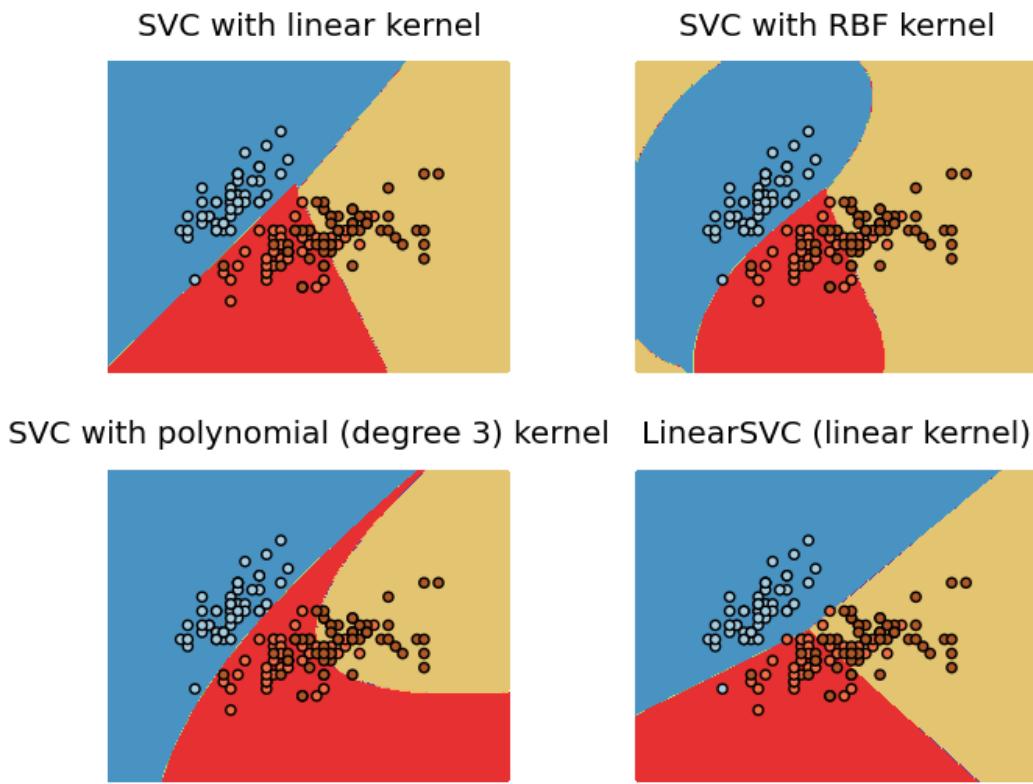


Figure 2.139: Plot different SVM classifiers in the iris dataset

### Plot different SVM classifiers in the iris dataset

Comparison of different linear SVM classifiers on the iris dataset. It will plot the decision surface for four different SVM classifiers.



**Python source code:** [plot\\_iris.py](#)

```
print __doc__

import numpy as np
import pylab as pl
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',
          'SVC with RBF kernel',
          'SVC with polynomial (degree 3) kernel',
          'LinearSVC (linear kernel)']

for i, clf in enumerate((svc, rbf_svc, poly_svc, lin_svc)):
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    pl.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)
    pl.contourf(xx, yy, Z, cmap=pl.cm.Paired)
    pl.axis('off')

    # Plot also the training points
    pl.scatter(X[:, 0], X[:, 1], c=Y, cmap=pl.cm.Paired)

    pl.title(titles[i])

pl.show()
```

**Total running time of the example:** 0.62 seconds

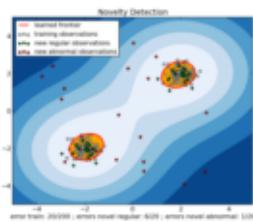
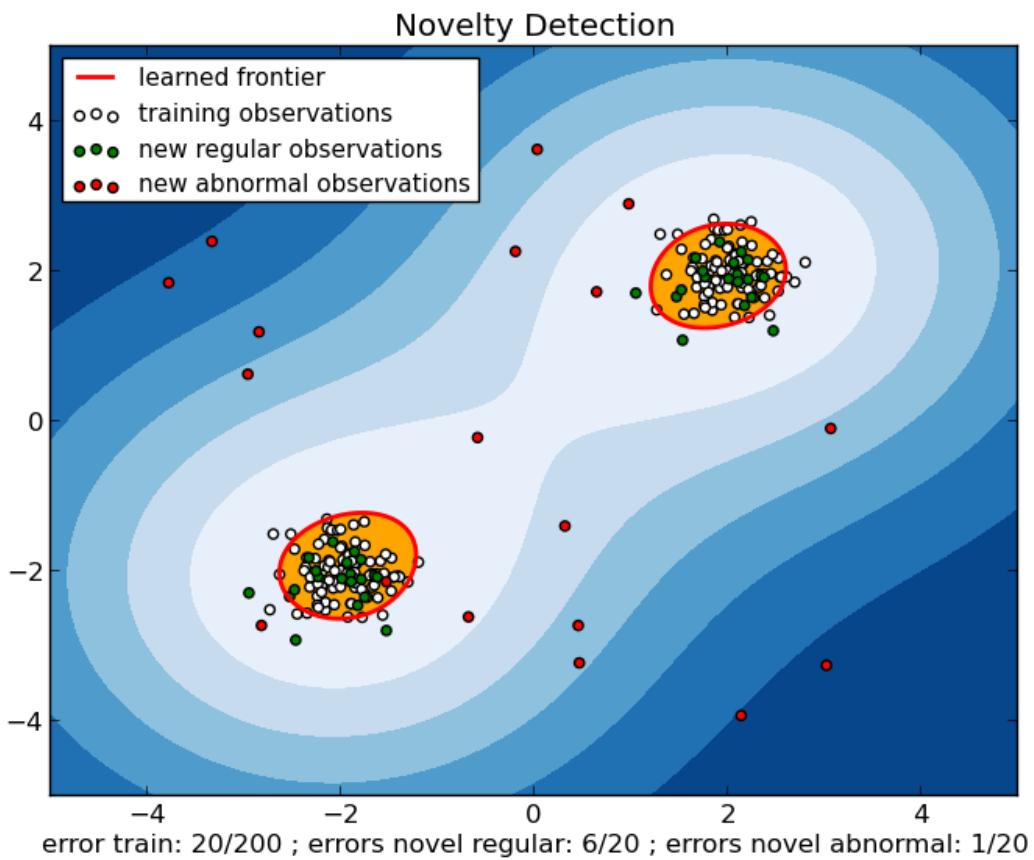


Figure 2.140: One-class SVM with non-linear kernel (RBF)

### One-class SVM with non-linear kernel (RBF)

*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.



**Python source code:** [plot\\_oneclass.py](#)

```
print __doc__

import numpy as np
import pylab as pl
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)

pl.title("Novelty Detection")
pl.contourf(xx, yy, Z, levels=np.linspace(Z.min(), 0, 7), cmap=pl.cm.Blues_r)
a = pl.contour(xx, yy, Z, levels=[0], linewidths=2, colors='red')
pl.contourf(xx, yy, Z, levels=[0, Z.max()], colors='orange')

b1 = pl.scatter(X_train[:, 0], X_train[:, 1], c='white')
b2 = pl.scatter(X_test[:, 0], X_test[:, 1], c='green')
c = pl.scatter(X_outliers[:, 0], X_outliers[:, 1], c='red')
pl.axis('tight')
pl.xlim((-5, 5))
pl.ylim((-5, 5))
pl.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))
pl.xlabel(
    "error train: %d/200 ; errors novel regular: %d/20 ; "
    "errors novel abnormal: %d/20"
    % (n_error_train, n_error_test, n_error_outliers))
pl.show()
```

**Total running time of the example:** 0.31 seconds

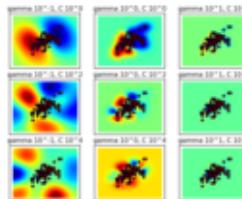


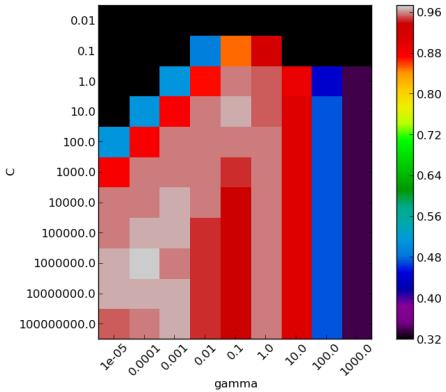
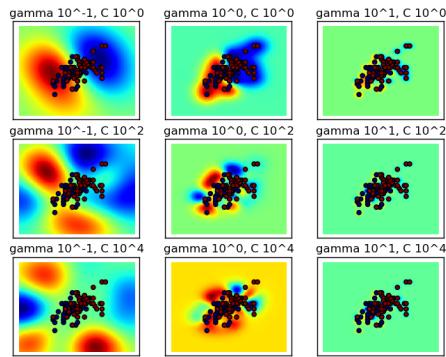
Figure 2.141: *RBF SVM parameters*

## RBF SVM parameters

This example illustrates the effect of the parameters *gamma* and *C* of the 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 *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.

Two plots are generated. The first is a visualization of the decision function for a variety of parameter values, and the second is a heatmap of the classifier’s cross-validation accuracy as a function of *C* and *gamma*.



### Script output:

```
('The best classifier is: ', SVC(C=1000000.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
    gamma=0.0001, kernel='rbf', max_iter=-1, probability=False,
    shrinking=True, tol=0.001, verbose=False))
```

### Python source code: plot\_rbf\_parameters.py

```
print __doc__

import numpy as np
import pylab as pl

from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris
from sklearn.cross_validation import StratifiedKFold
from sklearn.grid_search import GridSearchCV

#####
# Load and prepare data set
#
# dataset for grid search
iris = load_iris()
X = iris.data
Y = iris.target

# dataset for decision function visualization
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 classifier
#
# For an initial search, a logarithmic grid with basis
# 10 is often helpful. Using a basis of 2, a finer
# tuning can be achieved but at a much higher cost.

C_range = 10.0 ** np.arange(-2, 9)
gamma_range = 10.0 ** np.arange(-5, 4)
param_grid = dict(gamma=gamma_range, C=C_range)
cv = StratifiedKFold(y=Y, n_folds=3)
grid = GridSearchCV(SVC(), param_grid=param_grid, cv=cv)
grid.fit(X, Y)

print("The best classifier is: ", grid.best_estimator_)

# 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 = [1, 1e2, 1e4]
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
pl.figure(figsize=(8, 6))
xx, yy = np.meshgrid(np.linspace(-5, 5, 200), np.linspace(-5, 5, 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
    pl.subplot(len(C_2d_range), len(gamma_2d_range), k + 1)
    pl.title("gamma 10^%d, C 10^%d" % (np.log10(gamma), np.log10(C)),
             size='medium')

    # visualize parameter's effect on decision function
```

```

pl.pcolormesh(xx, yy, -Z, cmap=pl.cm.jet)
pl.scatter(X_2d[:, 0], X_2d[:, 1], c=Y_2d, cmap=pl.cm.jet)
pl.xticks(())
pl.yticks(())
pl.axis('tight')

# plot the scores of the grid
# grid_scores_ contains parameter settings and scores
score_dict = grid.grid_scores_

# We extract just the scores
scores = [x[1] for x in score_dict]
scores = np.array(scores).reshape(len(C_range), len(gamma_range))

# draw heatmap of accuracy as a function of gamma and C
pl.figure(figsize=(8, 6))
pl.subplots_adjust(left=0.05, right=0.95, bottom=0.15, top=0.95)
pl.imshow(scores, interpolation='nearest', cmap=pl.cm.spectral)
pl.xlabel('gamma')
pl.ylabel('C')
pl.colorbar()
pl.xticks(np.arange(len(gamma_range)), gamma_range, rotation=45)
pl.yticks(np.arange(len(C_range)), C_range)

pl.show()

```

**Total running time of the example:** 2.17 seconds

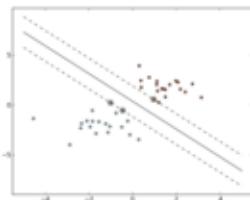
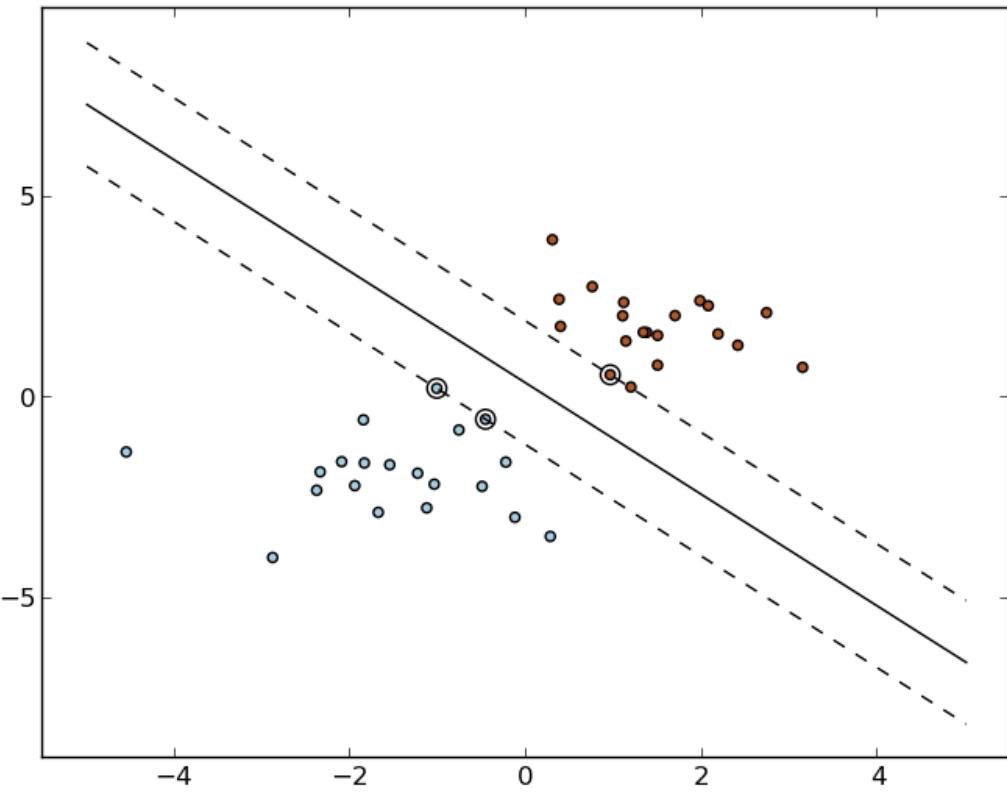


Figure 2.142: SVM: Maximum margin separating hyperplane

### 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.



**Python source code:** [plot\\_separating\\_hyperplane.py](#)

```
print __doc__

import numpy as np
import pylab as pl
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
pl.plot(xx, yy, 'k-')
pl.plot(xx, yy_down, 'k--')
pl.plot(xx, yy_up, 'k--')

pl.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
           s=80, facecolors='none')
pl.scatter(X[:, 0], X[:, 1], c=Y, cmap=pl.cm.Paired)

pl.axis('tight')
pl.show()

```

**Total running time of the example:** 0.12 seconds

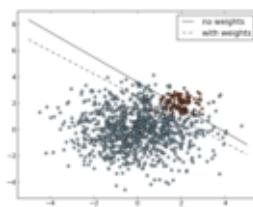
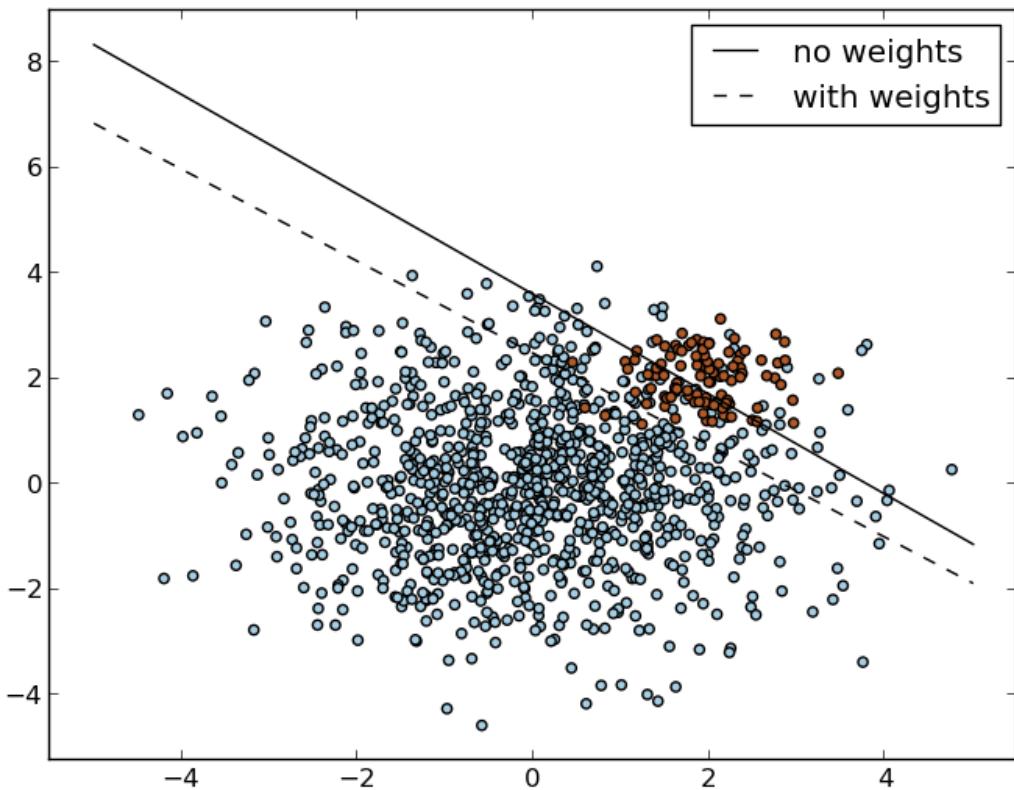


Figure 2.143: SVM: Separating hyperplane for unbalanced classes

### 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.



**Python source code:** [plot\\_separating\\_hyperplane\\_unbalanced.py](#)

```
print __doc__

import numpy as np
import pylab as pl
from sklearn import svm

# 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_ / 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]
wy = wa * xx - wclf.intercept_[0] / ww[1]

# plot separating hyperplanes and samples
h0 = pl.plot(xx, yy, 'k-', label='no weights')
h1 = pl.plot(xx, wy, 'k--', label='with weights')
pl.scatter(X[:, 0], X[:, 1], c=y, cmap=pl.cm.Paired)
pl.legend()

pl.axis('tight')
pl.show()
```

**Total running time of the example:** 0.12 seconds

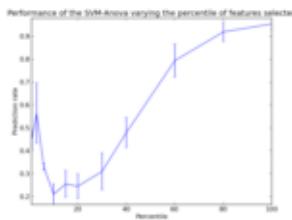
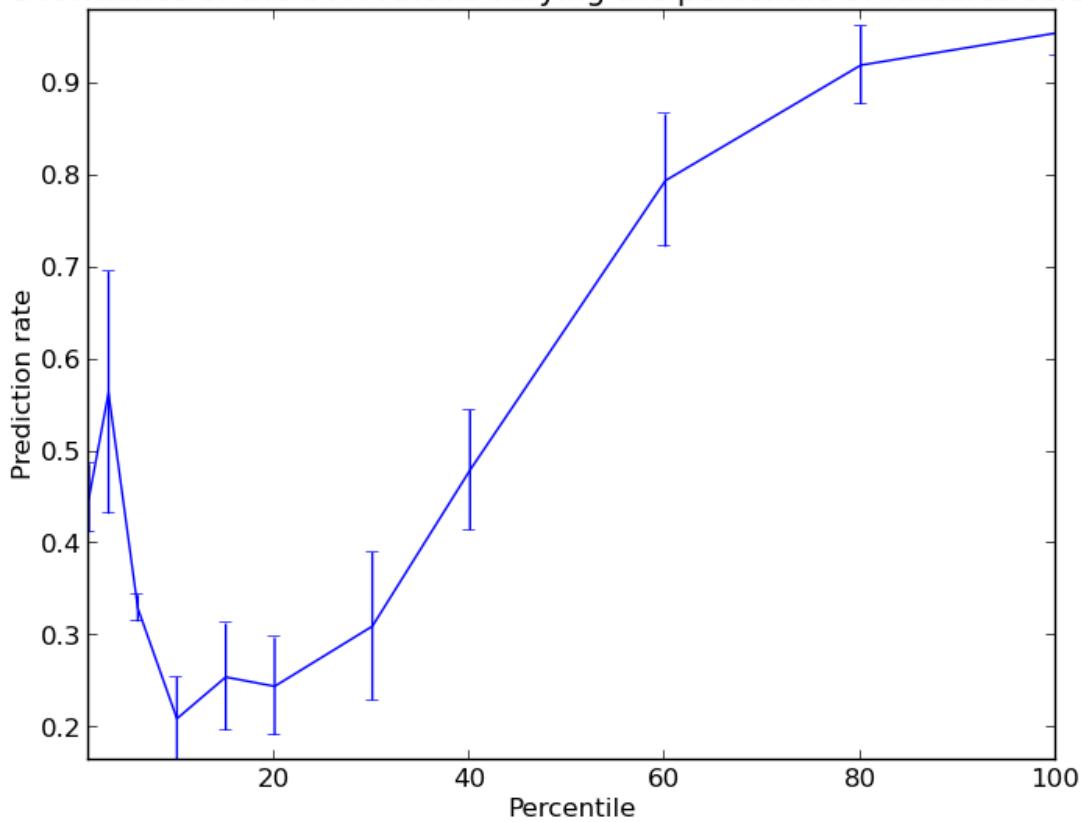


Figure 2.144: *SVM-Anova: SVM with univariate feature selection*

### 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.

### Performance of the SVM-Anova varying the percentile of features selected



**Python source code:** [plot\\_svm\\_anova.py](#)

```
print __doc__\n\nimport numpy as np\nimport pylab as pl\nfrom sklearn import svm, datasets, feature_selection, cross_validation\nfrom sklearn.pipeline import Pipeline\n\n#####\n# Import some data to play with\ndigits = datasets.load_digits()\ny = digits.target\n# Throw away data, to be in the curse of dimension settings\ny = y[:200]\nX = digits.data[:200]\nn_samples = len(y)\nX = X.reshape((n_samples, -1))\n# add 200 non-informative features\nX = np.hstack((X, 2 * np.random.random((n_samples, 200))))\n\n#####\n# Create a feature-selection transform and an instance of SVM that we\n# combine together to have an full-blown estimator\n\ntransform = 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())

pl.errorbar(percentiles, score_means, np.array(score_stds))

pl.title(
    'Performance of the SVM-Anova varying the percentile of features selected')
pl.xlabel('Percentile')
pl.ylabel('Prediction rate')

pl.axis('tight')
pl.show()

```

**Total running time of the example:** 0.61 seconds

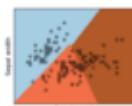
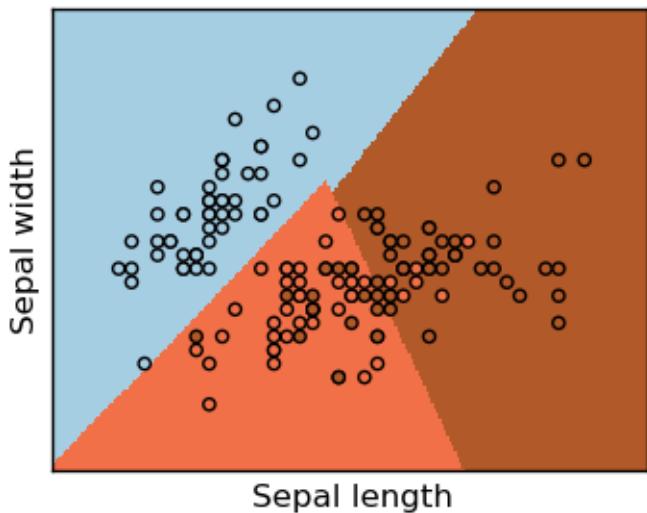


Figure 2.145: SVM-SVC (*Support Vector Classification*)

## SVM-SVC (Support Vector Classification)

The classification application of the SVM is used below. The Iris dataset has been used for this example  
The decision boundaries, are shown with all the points in the training-set.



**Python source code:** [plot\\_svm\\_iris.py](#)

```
print __doc__\n\n# Code source: Gael Varoquaux\n# Modified for Documentation merge by Jaques Grobler\n# License: BSD\n\nimport numpy as np\nimport pylab as pl\nfrom sklearn import svm, datasets\n\n# import some data to play with\niris = datasets.load_iris()\nX = iris.data[:, :2] # we only take the first two features.\nY = iris.target\n\nh = .02 # step size in the mesh\n\nclf = svm.SVC(C=1.0, kernel='linear')\n\n# we create an instance of SVM Classifier and fit the data.\nclf.fit(X, Y)\n\n# Plot the decision boundary. For that, we will assign a color to each\n# point in the mesh [x_min, m_max]x[y_min, y_max].\nx_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5\ny_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5\nxx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))\nZ = clf.predict(np.c_[xx.ravel(), yy.ravel()])\n\n# Put the result into a color plot\nZ = Z.reshape(xx.shape)\npl.figure(1, figsize=(4, 3))\npl.pcolormesh(xx, yy, Z, cmap=pl.cm.Paired)\n\n# Plot also the training points
```

```

pl.scatter(X[:, 0], X[:, 1], c=Y, cmap=pl.cm.Paired)
pl.xlabel('Sepal length')
pl.ylabel('Sepal width')

pl.xlim(xx.min(), xx.max())
pl.ylim(yy.min(), yy.max())
pl.xticks(())
pl.yticks(())

pl.show()

```

**Total running time of the example:** 0.15 seconds

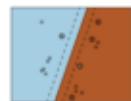
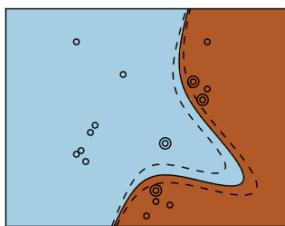


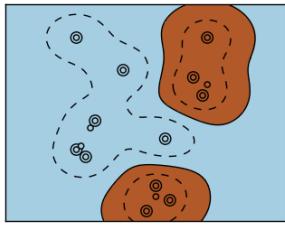
Figure 2.146: SVM-Kernels

## SVM-Kernels

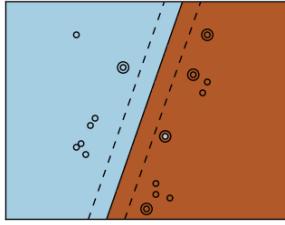
Three different types of SVM-Kernels are displayed below. The polynomial and RBF are especially useful when the data-points are not linearly separable.



•



•



•

**Python source code:** [plot\\_svm\\_kernels.py](#)

```
print __doc__\n\n# Code source: Gael Varoquaux\n# License: BSD\n\nimport numpy as np\nimport pylab as pl\nfrom sklearn import svm\n\n# Our dataset and targets\nX = np.c_[(.4, -.7),\n          (-1.5, -1),\n          (-1.4, -.9),\n          (-1.3, -1.2),\n          (-1.1, -.2),\n          (-1.2, -.4),\n          (-.5, 1.2),\n          (-1.5, 2.1),\n          (1, 1),\n          # --\n          (1.3, .8),\n          (1.2, .5),\n          (.2, -2),\n          (.5, -2.4),\n          (.2, -2.3),\n          (0, -2.7),\n          (1.3, 2.1)].T\nY = [0] * 8 + [1] * 8\n\n# figure number\nfignum = 1\n\n# fit the model\nfor kernel in ('linear', 'poly', 'rbf'):\n    clf = svm.SVC(kernel=kernel, gamma=2)\n    clf.fit(X, Y)\n\n    # plot the line, the points, and the nearest vectors to the plane\n    pl.figure(fignum, figsize=(4, 3))\n    pl.clf()\n\n    pl.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=80,\n               facecolors='none', zorder=10)\n    pl.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=pl.cm.Paired)\n\n    pl.axis('tight')\n    x_min = -3\n    x_max = 3\n    y_min = -3\n    y_max = 3\n\n    XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]\n    Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()])\n\n    # Put the result into a color plot\n    Z = Z.reshape(XX.shape)
```

```

pl.figure(fignum, figsize=(4, 3))
pl.pcolormesh(XX, YY, Z > 0, cmap=pl.cm.Paired)
pl.contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '-.', '--'],
           levels=[-.5, 0, .5])

pl.xlim(x_min, x_max)
pl.ylim(y_min, y_max)

pl.xticks(())
pl.yticks(())
fignum = fignum + 1
pl.show()

```

**Total running time of the example:** 0.27 seconds

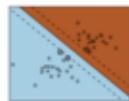
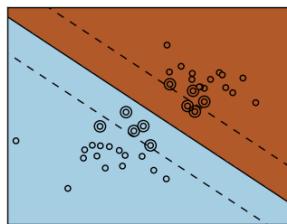


Figure 2.147: SVM Margins Example

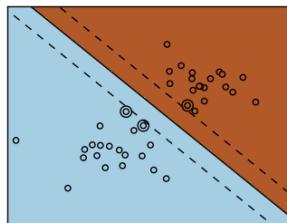
## SVM Margins Example

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: Gael Varoquaux
```

```
# Modified for Documentation merge by Jaques Grobler
# License: BSD

import numpy as np
import pylab as pl
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
    pl.figure(fignum, figsize=(4, 3))
    pl.clf()
    pl.plot(xx, yy, 'k-')
    pl.plot(xx, yy_down, 'k--')
    pl.plot(xx, yy_up, 'k--')

    pl.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=80,
               facecolors='none', zorder=10)
    pl.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=pl.cm.Paired)

    pl.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)
    pl.figure(fignum, figsize=(4, 3))
    pl.pcolormesh(XX, YY, Z, cmap=pl.cm.Paired)
```

```
pl.xlim(x_min, x_max)
pl.ylim(y_min, y_max)

pl.xticks(())
pl.yticks(())
fignum = fignum + 1

pl.show()
```

**Total running time of the example:** 0.21 seconds

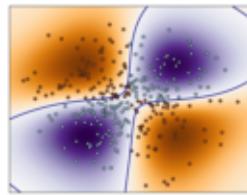
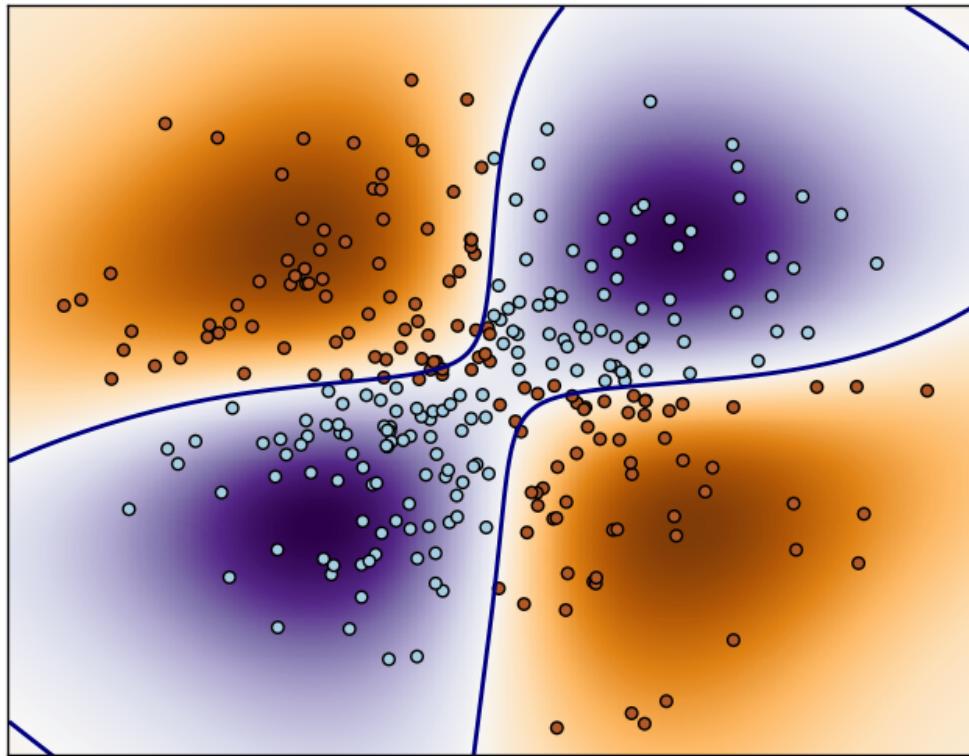


Figure 2.148: *Non-linear SVM*

## 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.



**Python source code:** [plot\\_svm\\_nonlinear.py](#)

```
print __doc__

import numpy as np
import pylab as pl
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)

pl.imshow(Z, interpolation='nearest',
          extent=(xx.min(), xx.max(), yy.min(), yy.max()), aspect='auto',
          origin='lower', cmap=pl.cm.PuOr_r)
contours = pl.contour(xx, yy, Z, levels=[0], linewidths=2,
```

```

linetypes='--')
pl.scatter(X[:, 0], X[:, 1], s=30, c=Y, cmap=pl.cm.Paired)
pl.xticks(())
pl.yticks(())
pl.axis([-3, 3, -3, 3])
pl.show()

```

**Total running time of the example:** 1.26 seconds

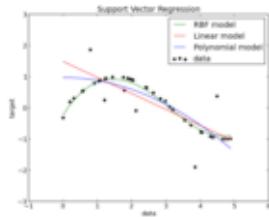
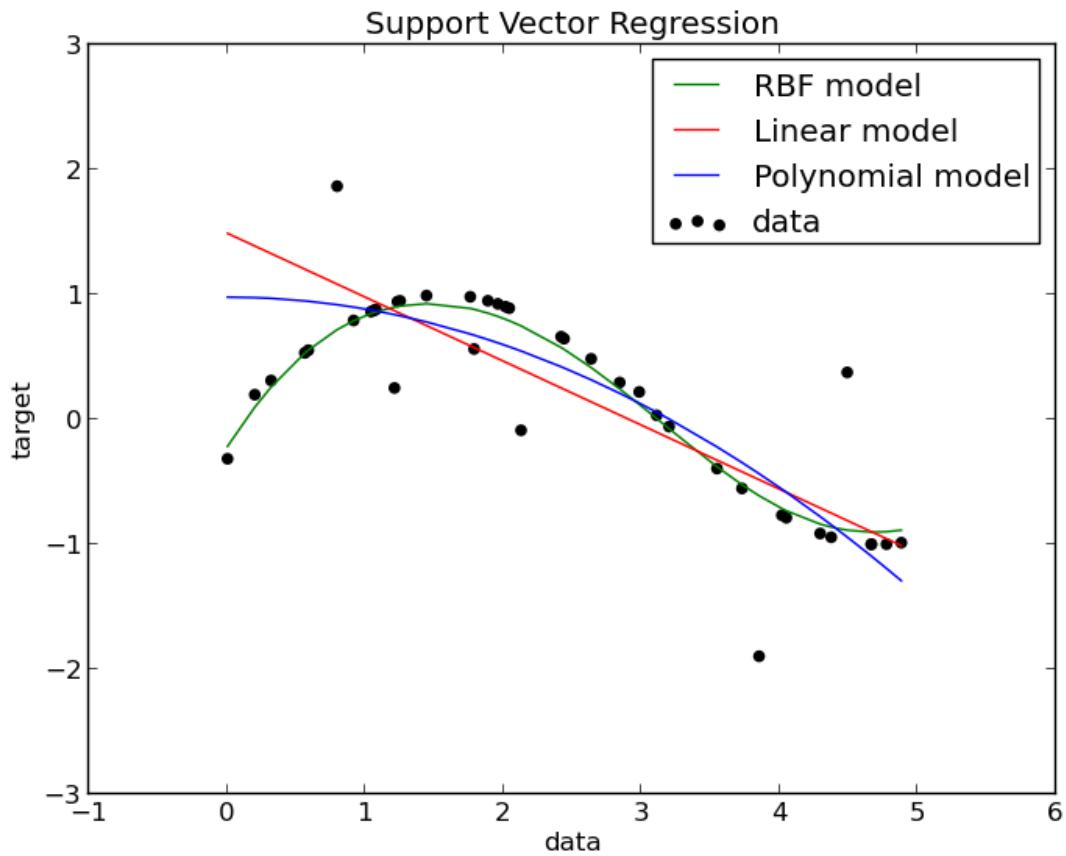


Figure 2.149: Support Vector Regression (SVR) using linear and non-linear kernels

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

Toy example of 1D regression using linear, polynomial and RBF kernels.



**Python source code:** [plot\\_svm\\_regression.py](#)

```
print __doc__\n\n#####\n# Generate sample data\nimport numpy as np\n\nX = np.sort(5 * np.random.rand(40, 1), axis=0)\ny = np.sin(X).ravel()\n\n#####\n# Add noise to targets\ny[::5] += 3 * (0.5 - np.random.rand(8))\n\n#####\n# Fit regression model\nfrom sklearn.svm import SVR\n\nsvr_rbf = SVR(kernel='rbf', C=1e3, gamma=0.1)\nsvr_lin = SVR(kernel='linear', C=1e3)\nsvr_poly = SVR(kernel='poly', C=1e3, degree=2)\ny_rbf = svr_rbf.fit(X, y).predict(X)\ny_lin = svr_lin.fit(X, y).predict(X)\ny_poly = svr_poly.fit(X, y).predict(X)\n\n#####\n# look at the results\nimport pylab as pl\npl.scatter(X, y, c='k', label='data')\npl.hold('on')\npl.plot(X, y_rbf, c='g', label='RBF model')\npl.plot(X, y_lin, c='r', label='Linear model')\npl.plot(X, y_poly, c='b', label='Polynomial model')\npl.xlabel('data')\npl.ylabel('target')\npl.title('Support Vector Regression')\npl.legend()\npl.show()
```

**Total running time of the example:** 4.24 seconds

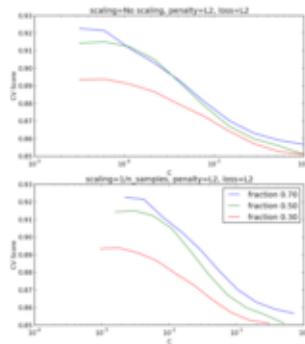


Figure 2.150: *Support Vector Classification (SVC): scaling the regularization parameter*

## Support Vector Classification (SVC): scaling the regularization parameter

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.
- $\Omega$  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 dependant 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

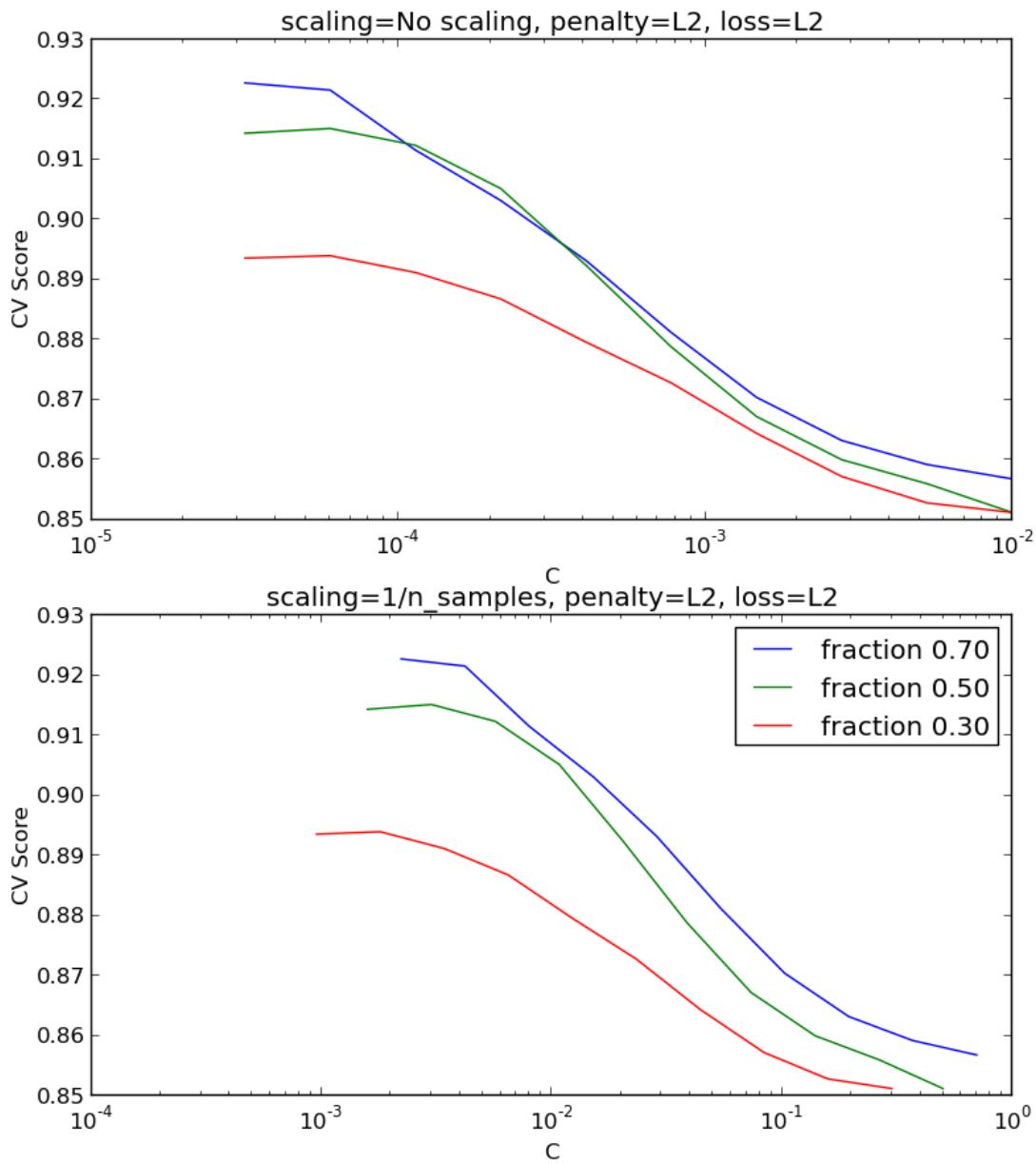
The two figures below plot the values of  $C$  on the *x-axis* and the corresponding cross-validation scores on the *y-axis*, for several different fractions of a generated data-set.

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
```

```
import numpy as np
```

```

import pylab as pl

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='L2', dual=False,
                       tol=1e-3),
              np.logspace(-2.3, -1.3, 10), X_1, y_1),
             (LinearSVC(penalty='L2', loss='L2', 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
    pl.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):
            pl.subplot(2, 1, subplotnum + 1)
            pl.xlabel('C')
            pl.ylabel('CV Score')
            grid_cs = cs * float(scaler) # scale the C's
            pl.semilogx(grid_cs, scores, label="fraction %.2f" %

```

```
    train_size)
pl.title('scaling=%s, penalty=%s, loss=%s' %
          (name, clf.penalty, clf.loss))

pl.legend(loc="best")
pl.show()
```

**Total running time of the example:** 16.56 seconds

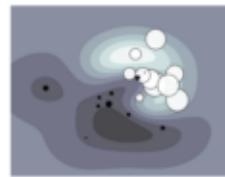
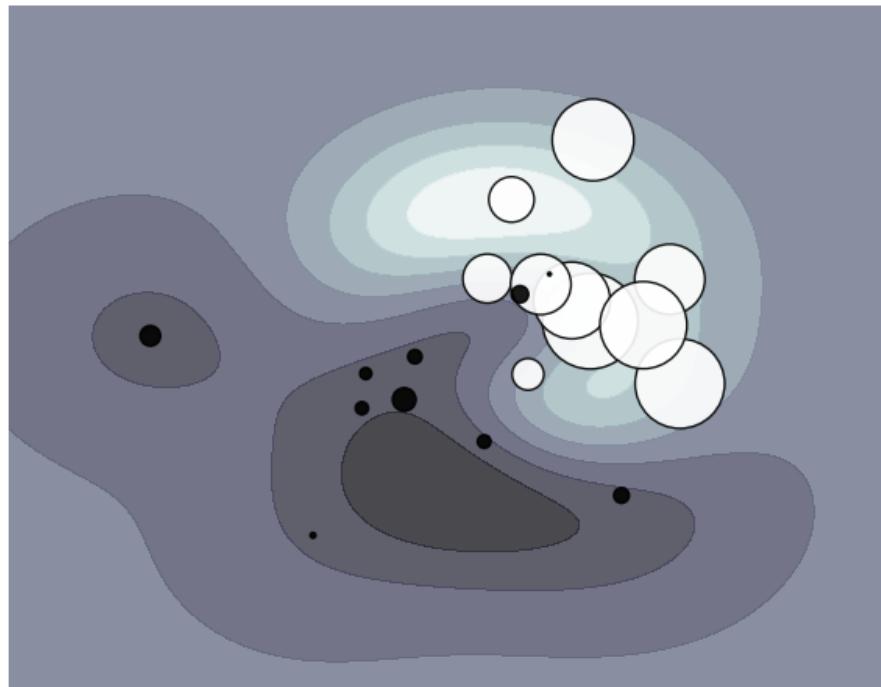


Figure 2.151: SVM: Weighted samples

### SVM: Weighted samples

Plot decision function of a weighted dataset, where the size of points is proportional to its weight.



**Python source code:** [plot\\_weighted\\_samples.py](#)

```
print __doc__

import numpy as np
import pylab as pl
from sklearn import svm

# 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

# # fit the model
clf = svm.SVC()
clf.fit(X, Y, sample_weight=sample_weight)

# plot the decision function
xx, yy = np.meshgrid(np.linspace(-4, 5, 500), np.linspace(-4, 5, 500))

Z = clf.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
pl.contourf(xx, yy, Z, alpha=0.75, cmap=pl.cm.bone)
pl.scatter(X[:, 0], X[:, 1], c=Y, s=sample_weight, alpha=0.9, cmap=pl.cm.bone)

pl.axis('off')
pl.show()
```

**Total running time of the example:** 0.29 seconds

## 2.1.16 Decision Trees

Examples concerning the `sklearn.tree` package.

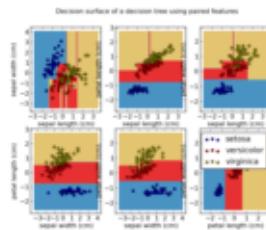


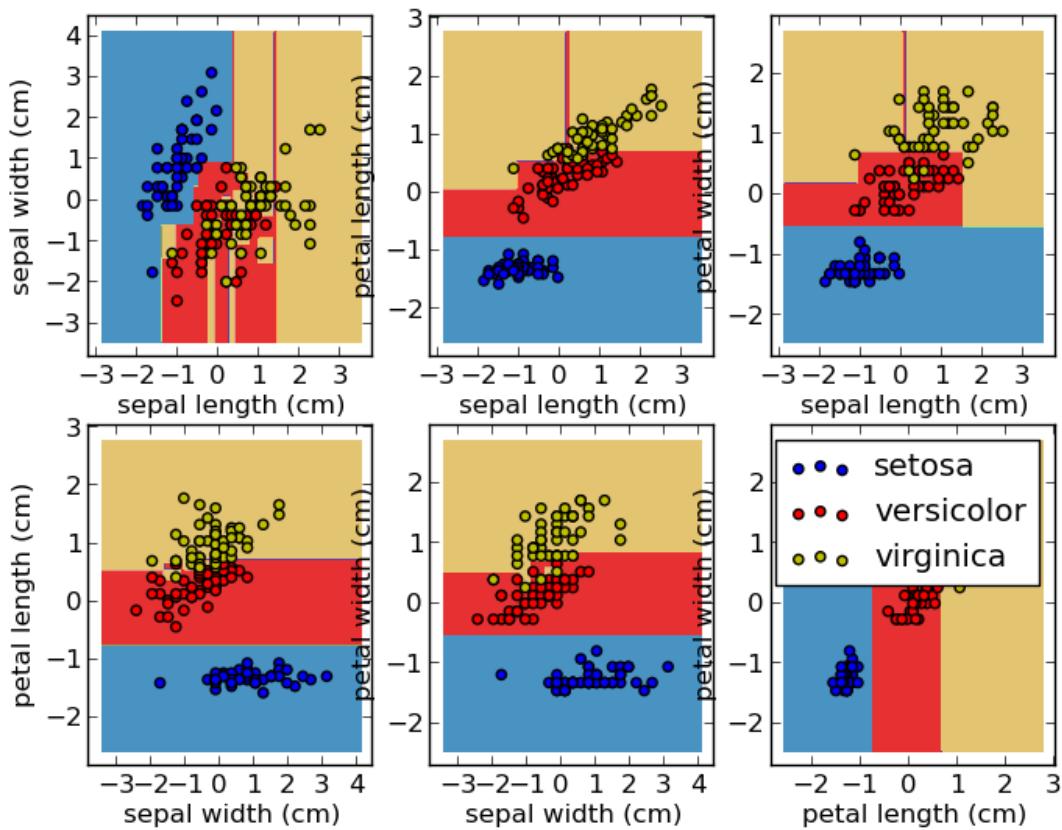
Figure 2.152: Plot the decision surface of a decision tree on the iris dataset

### 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.

For each pair of iris features, the decision tree learns decision boundaries made of combinations of simple thresholding rules inferred from the training samples.

Decision surface of a decision tree using paired features



**Python source code:** [plot\\_iris.py](#)

```
print __doc__

import numpy as np
import pylab as pl

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.shuffle(idx)
    X = X[idx]
    y = y[idx]
```

```

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
pl.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 = pl.contourf(xx, yy, Z, cmap=pl.cm.Paired)

pl.xlabel(iris.feature_names[pair[0]])
pl.ylabel(iris.feature_names[pair[1]])
pl.axis("tight")

# Plot the training points
for i, color in zip(xrange(n_classes), plot_colors):
    idx = np.where(y == i)
    pl.scatter(X[idx, 0], X[idx, 1], c=color, label=iris.target_names[i],
               cmap=pl.cm.Paired)

pl.axis("tight")

pl.suptitle("Decision surface of a decision tree using paired features")
pl.legend()
pl.show()

```

**Total running time of the example:** 0.52 seconds

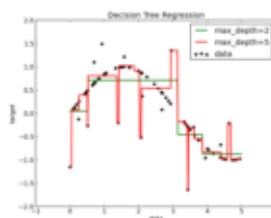
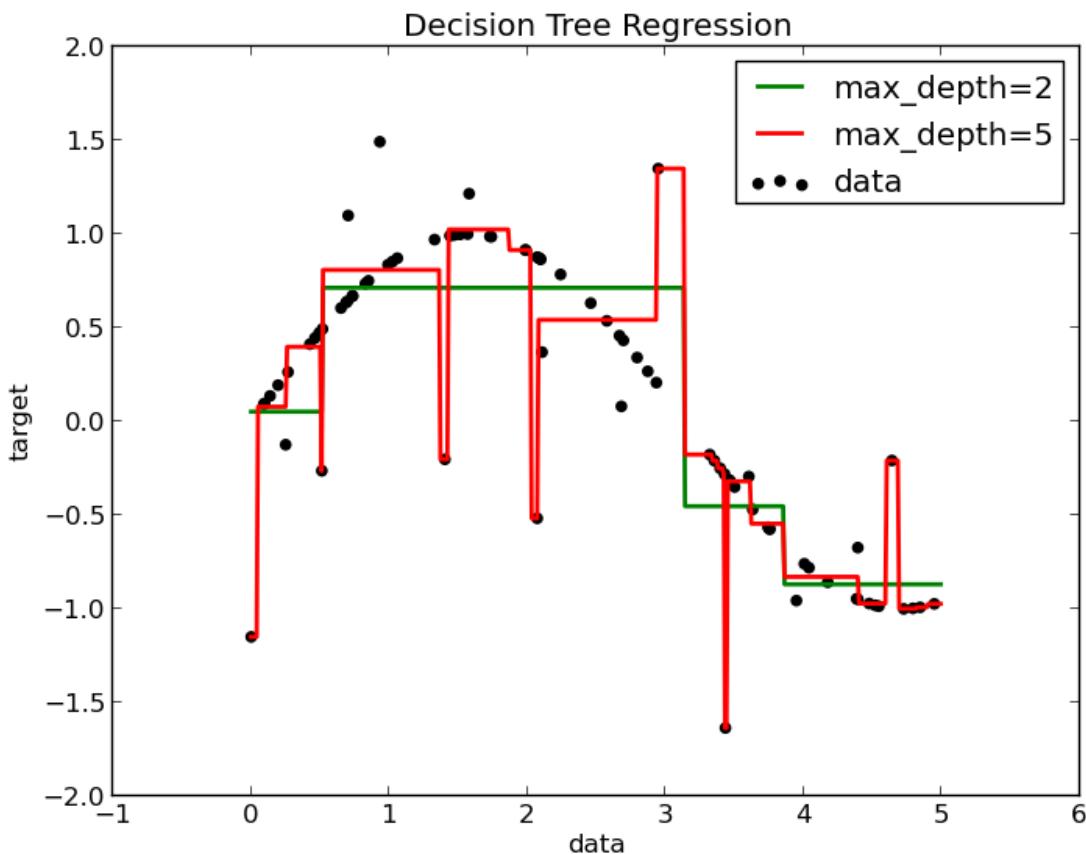


Figure 2.153: *Decision Tree Regression*

## Decision Tree Regression

1D regression with *decision trees*: the decision tree 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 (controled 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.



**Python source code:** [plot\\_tree\\_regression.py](#)

```
print __doc__

import numpy as np

# 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
from sklearn.tree import DecisionTreeRegressor

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
import pylab as pl

pl.figure()
pl.scatter(X, y, c="k", label="data")
pl.plot(X_test, y_1, c="g", label="max_depth=2", linewidth=2)
pl.plot(X_test, y_2, c="r", label="max_depth=5", linewidth=2)
pl.xlabel("data")
pl.ylabel("target")
pl.title("Decision Tree Regression")
pl.legend()
pl.show()

```

**Total running time of the example:** 0.14 seconds

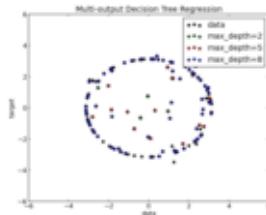
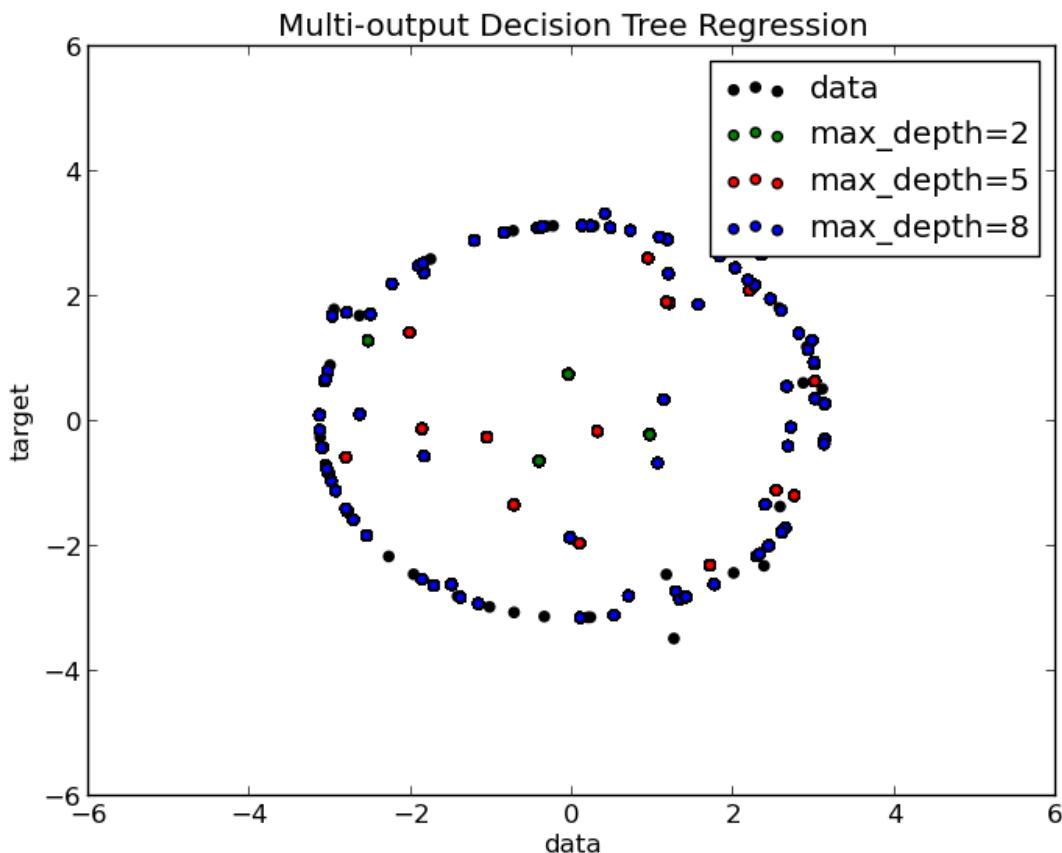


Figure 2.154: Multi-output Decision Tree Regression

## Multi-output Decision Tree Regression

Multi-output regression with *decision trees*: the decision tree 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.



**Python source code:** [plot\\_tree\\_regression\\_multioutput.py](#)

```
print __doc__

import numpy as np

# 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
from sklearn.tree import DecisionTreeRegressor

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
import pylab as pl

pl.figure()
pl.scatter(y[:, 0], y[:, 1], c="k", label="data")
pl.scatter(y_1[:, 0], y_1[:, 1], c="g", label="max_depth=2")
pl.scatter(y_2[:, 0], y_2[:, 1], c="r", label="max_depth=5")
pl.scatter(y_3[:, 0], y_3[:, 1], c="b", label="max_depth=8")
pl.xlim([-6, 6])
pl.ylim([-6, 6])
pl.xlabel("data")
pl.ylabel("target")
pl.title("Multi-output Decision Tree Regression")
pl.legend()
pl.show()
```

**Total running time of the example:** 0.61 seconds



# DEVELOPMENT

## 3.1 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>

### 3.1.1 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.

### 3.1.2 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
```

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.

### 3.1.3 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>

---

## How to contribute

The preferred way to contribute to scikit-learn is to fork the [main repository](#) on GitHub:

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. 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.

The documentation should also include expected time and space complexity of the algorithm and scalability, e.g. “this algorithm can scale to a large number of samples > 100000, but does not scale in dimensionality: n\_features is expected to be lower than 100”.

To build the documentation, see the documentation section below.

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.

---

## 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.

## 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` and `matplotlib`.

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. It is best to always start with a small paragraph with a hand-waving explanation of what the method does to the data and a figure (coming from an example) illustrating it.

**Warning: Sphinx version**

While we do our best to have the documentation build under as many versions of Sphinx as possible, the different versions tend to behave slightly differently. To get the best results, you should use version 1.0.

## 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 named, 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.
- 

## Developers web site

More information can be found on the [developer’s wiki](#).

## 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.

### 3.1.4 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":

### 3.1.5 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.
- **Please don't use ‘import \*’ in any case.** It is considered harmful by the [official Python recommendations](#). It makes the code harder to read as the origin of symbols is no longer explicitly referenced, but most important, it prevents using a static analysis tool like [pyflakes](#) to automatically find bugs in scikit-learn.
- Use the [numpy docstring standard](#) in all your docstrings.

A good example of code that we like can be found [here](#).

## Input validation

The module `sklearn.utils` contains various functions for doing input validation and conversion. Sometimes, `np.asarray` suffices for validation; do *not* use `np.asanyarray` or `np.atleast_2d`, since those let NumPy's `np.matrix` through, which has a different API (e.g., `*` means dot product on `np.matrix`, but Hadamard product on `np.ndarray`).

In other cases, be sure to call `safe_asarray`, `atleast2d_or_csr`, `as_float_array` or `array2d` on any array-like argument passed to a scikit-learn API function. The exact function to use depends mainly on whether `scipy.sparse` matrices must be accepted.

For more information, refer to the *Utilities for Developers* page.

## Random Numbers

If your code depends on a random number generator, do not use `numpy.random.random()` or similar routines. To ensure repeatability in error checking, the routine should accept a keyword `random_state` and use this to construct a `numpy.random.RandomState` object. See `sklearn.utils.check_random_state` in *Utilities for Developers*.

Here's a simple example of code using some of the above guidelines:

```
from sklearn.utils import 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]
```

## 3.1.6 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.

### Different objects

The main objects in scikit-learn are (one class can implement multiple interfaces):

**Estimator** The base object, implements:

```
estimator = obj.fit(data)
```

**Predictor** For supervised learning, or some unsupervised problems, implements:

```
prediction = obj.predict(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)
```

## 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__`). This method is not required for an object to be an estimator.

All estimators 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, a `__init__` should look like:

```
def __init__(self, param1=1, param2=2):
    self.param1 = param1
    self.param2 = param2
```

There should be no logic, and the parameters should not be changed. The corresponding logic should be put where the parameters are used. 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
```

Scikit-learn relies on this mechanism to introspect objects to set their parameters by cross-validation.

## 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	
<code>X</code>	array-like, with shape = [N, D], where N is the number of samples and D is the number of features.
<code>y</code>	array, with shape = [N], where N is the number of samples.
<code>kwargs</code>	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 are kindly asked to accept a `y=None` keyword argument in the second position that is just ignored by the estimator.

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`.

## Unresolved API issues

Some things are must still be decided:

- what should happen when `predict` is called before `fit()` ?
- which exception should be raised when the shape of arrays do not match in `fit()` ?

## Working notes

For unresolved issues, TODOs, and remarks on ongoing work, developers are advised to maintain notes on the GitHub wiki.

## Specific models

In linear models, coefficients are stored in an array called `coef_`, and the independent term is stored in `intercept_`.

## 3.2 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 late 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.

---

### 3.2.1 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.

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.

### 3.2.2 Profiling Python code

In order to profile Python code we recommend to write a script that loads and prepare you data and then use the IPython integrated profiler for interactively exploring the relevant part for the code.

Suppose we want to profile the Non Negative Matrix Factorization module of 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` columns 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 begining 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 them 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-profiler` 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
172
173
174      48       139      2.9      0.0
175      48    112141    2336.3      5.8
176      48    16144     336.3      0.8
177
178
179      48       144      3.0      0.0
180      48       113      2.4      0.0
181     638      1880      2.9      0.1
182     638     195133    305.9     10.2
183     638     495761    777.1     25.9
184     638      2449      3.8      0.1
185      48       130      2.7      0.0
186
187     1474      4474      3.0      0.2
188     1474     83833     56.9      4.4
189
190     1474     194239    131.8     10.1
191     1474     48858     33.1      2.5
192     1474     150407    102.0      7.8
193     1474     515390    349.7     26.9
...

```

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.

### 3.2.3 Memory usage profiling

You can analyze in detail the memory usage of any Python code with the help of `memory_profiler`. First, install the latest version:

```
$ pip install -U memory_profiler
```

Then, setup the magics in a manner similar to `line_profiler`.

- Under IPython <= 0.10, 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?`.

### 3.2.4 Performance tips for the Cython developer

If profiling of the Python code reveals that the Python interpreter overhead is larger by one order of magnitude or more than the cost of the actual numerical computation (e.g. `for` loops over vector components, nested evaluation of conditional expression, scalar arithmetics...), 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_1/)
- [http://conference.scipy.org/proceedings/SciPy2009/paper\\_2/](http://conference.scipy.org/proceedings/SciPy2009/paper_2/)

### 3.2.5 Profiling compiled extensions

When working with compiled extensions (written in C/C++ with a wrapper or directly as Cython extension), the default Python profiler is useless: we need a dedicated tool to introspect what's happening inside the compiled extension itself.

#### Using yep and 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](#).

---

#### Using gprof

In order to profile compiled Python extensions one could use `gprof` after having recompiled the project with `gcc -pg` and using the `python-dbg` variant of the interpreter on debian / ubuntu: however this approach requires to also have `numpy` and `scipy` recompiled with `-pg` which is rather complicated to get working.

Fortunately there exist two alternative profilers that don't require you to recompile everything.

## Using valgrind / callgrind / kcachegrind

TODO

### 3.2.6 Multi-core parallelism using `joblib.Parallel`

TODO: give a simple teaser example here.

Checkout the official joblib documentation:

- <http://packages.python.org/joblib/>

### 3.2.7 A sample algorithmic trick: warm restarts for cross validation

TODO: demonstrate the warm restart tricks for cross validation of linear regression with Coordinate Descent.

## 3.3 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.

### 3.3.1 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.
- `safe_asarray`: Convert input to array or sparse matrix. Equivalent to `np.asarray`, but sparse matrices are passed through.
- `as_float_array`: convert input to an array of floats. If a sparse matrix is passed, a sparse matrix will be returned.
- `array2d`: equivalent to `np.atleast_2d`, but the `order` and `dtype` of the input are maintained.
- `atleast2d_or_csr`: equivalent to `array2d`, but if a sparse matrix is passed, will convert to csr format. Also calls `assert_all_finite`.
- `check_arrays`: check that all input arrays have consistent first dimensions. This will work for an arbitrary number of arrays.
- `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])
```

### 3.3.2 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`.

### 3.3.3 Efficient Random Sampling

- `random.sample_without_replacement`: implements efficient algorithms for sampling `n_samples` integers from a population of size `n_population` without replacement.

### 3.3.4 Efficient Routines for Sparse Matrices

The `sklearn.utils.sparsefuncs` cython module hosts compiled extensions to efficiently process `scipy.sparse` data.

- `sparsefuncs.mean_variance_axis0`: compute the means and variances along axis 0 of a CSR matrix. Used for normalizing the tolerance stopping criterion in `sklearn.cluster.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`.

### 3.3.5 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`.

### 3.3.6 Backports

- `fixes.Counter` (partial backport of `collections.Counter` from Python 2.7) Used in `sklearn.feature_extraction.text`.
- `fixes.unique`: (backport of `np.unique` from numpy 1.4). Find the unique entries in an array. In numpy versions < 1.4, `np.unique` is less flexible. Used in `sklearn.cross_validation`.
- `fixes.copysign`: (backport of `np.copysign` from numpy 1.4). Change the sign of `x1` to that of `x2`, element-wise.
- `fixes.in1d`: (backport of `np.in1d` from numpy 1.4). Test whether each element of an array is in a second array. Used in `sklearn.datasets.twenty_newsgroups` and `sklearn.feature_extraction.image`.
- `fixes.savemat` (backport of `scipy.io.savemat` from scipy 0.7.2). Save an array in MATLAB-format. In earlier versions, the keyword `oned_as` is not available.
- `fixes.count_nonzero` (backport of `np.count_nonzero` from numpy 1.6). Count the nonzero elements of a matrix. Used in tests of `sklearn.linear_model`.
- `arrayfuncs.solve_triangular` (Back-ported from `scipy` v0.9) Used in `sklearn.linear_model.omp`, independent back-ports in `sklearn.mixture.gmm` and `sklearn.gaussian_process`.

- `sparsetools.cs_graph_components` (backported from `scipy.sparse.cs_graph_components` in `scipy 0.9`). Used in `sklearn.cluster.hierarchical`, as well as in tests for `sklearn.feature_extraction`.

## 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.

## Benchmarking

- `bench.total_seconds` (back-ported from `timedelta.total_seconds` in Python 2.7). Used in `benchmarks/bench_glm.py`.

### 3.3.7 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.
- `mock_urllib2`: Object which mocks the `urllib2` module to fake requests of `mldata`. Used in tests of `sklearn.datasets`.
- `testing.all_estimators`: returns a list of all estimators in `sklearn` to test for consistent behavior and interfaces.

### 3.3.8 Helper Functions

- `gen_even_slices`: generator to create n-packs of slices going up to n. Used in `sklearn.decomposition.dict_learning` and `sklearn.cluster.k_means`.
- `arraybuilder.ArrayBuilder`: Helper class to incrementally build a 1-d `numpy.ndarray`. Currently used in `sklearn.datasets._svmlight_format.pyx`.
- `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.

### 3.3.9 Hash Functions

- murmurhash3\_32 provides a python wrapper for the *MurmurHash3\_x86\_32* C++ non cryptographic hash function. This hash function is suitable for implementing lookup tables, Bloom filters, Count Min Sketch, feature hashing and implicitly defined sparse random projections:

```
>>> from sklearn.utils import murmurhash3_32
>>> murmurhash3_32("some feature", seed=0)
-384616559

>>> murmurhash3_32("some feature", seed=0, positive=True)
3910350737L
```

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.

### 3.3.10 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`.

## 3.4 Developers’ Tips for Debugging

### 3.4.1 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 is 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.sup`.
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.sup 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.

## 3.5 Maintainer / core-developer information

### 3.5.1 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 tarbal 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

## 3.6 About us

This is a community effort, and as such many people have contributed to it over the years.

### 3.6.1 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 striving international community has been leading the development.

### 3.6.2 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 Duchiennay
- 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

### 3.6.3 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}
}
```

### 3.6.4 Funding

INRIA actively supports this project. It has provided funding for Fabian Pedregosa to work on this project full time in the period 2010-2012. It also hosts coding sprints and other events.



Google sponsored David Cournapeau with a Summer of Code Scholarship in the summer of 2007 and Vlad Niculae in 2011. 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).

## 3.7 Support

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

### 3.7.1 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.

### 3.7.2 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 you trying to predict: binary, multiclass (1 out of n\_classes) or multilabel (k out of n\_classes) classification or continuous variable regression.

### 3.7.3 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.

## 3.7.4 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>

## 3.7.5 Documentation resources

This documentation is relative to 0.13-git. Documentation for other versions can be found here:

- [Development version](#)
- [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](#).

## 3.8 0.13

### 3.8.1 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: Combining feature extractors* 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 nonlinear dimensionality reduction by Wei Li. See *Spectral Embedding* in the user guide.

### 3.8.2 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.
- Implement `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` datastructure 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](#).

### 3.8.3 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__`.

### 3.8.4 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 Winternman
- 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

## 3.9 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

### 3.9.1 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](#)

### 3.9.2 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](#)

## 3.10 0.12

### 3.10.1 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*.

### 3.10.2 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`.

### 3.10.3 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 Brandy 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

## 3.11 0.11

### 3.11.1 Changelog

#### Highlights

- Gradient boosted regression trees (*Gradient Tree Boosting*) for classification and regression by Peter Prettenhofer and Scott White .
- Simple dict-based feature loader with support for categorical variables (`feature_extraction.DictVectorizer`) by Lars Buitinck.
- Added Matthews correlation coefficient (`metrics.matthews_corrcoef`) and added macro and micro average options to `metrics.precision_score`, `metrics.recall_score` and `metrics.f1_score` by Satrajit Ghosh.
- *Out of Bag Estimates* of generalization error for *Ensemble methods* by Andreas Müller.
- *Randomized sparse 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.

## 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: setting 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 simplication 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`.

### 3.11.2 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.

### 3.11.3 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 flyingmimidev
- 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](#)

## 3.12 0.10

### 3.12.1 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.

### 3.12.2 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.

### 3.12.3 People

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

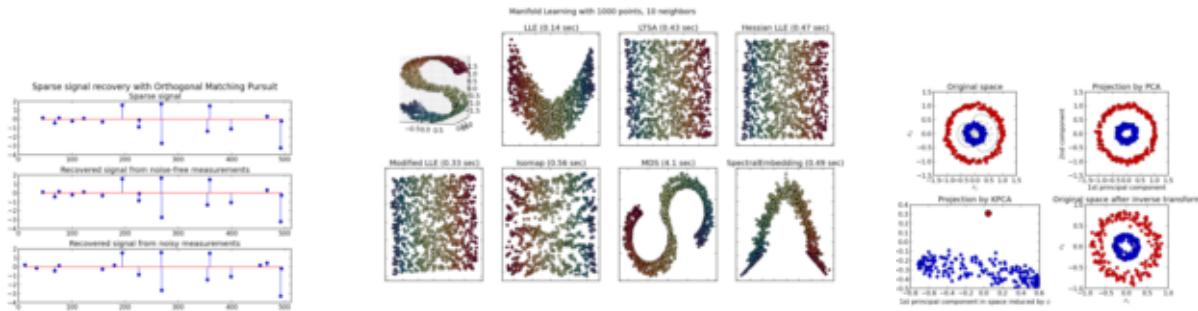
- 246 [Andreas Müller](#)
- 242 [Olivier Grisel](#)
- 220 [Gilles Louppe](#)
- 183 [Brian Holt](#)
- 166 [Gael Varoquaux](#)
- 144 [Lars Buitinck](#)
- 73 [Vlad Niculae](#)
- 65 [Peter Prettenhofer](#)
- 64 [Fabian Pedregosa](#)
- 60 [Robert Layton](#)
- 55 [Mathieu Blondel](#)
- 52 [Jake Vanderplas](#)
- 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

## 3.13 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.



### 3.13.1 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 *Bootstrapping cross-validation*, *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 utilites in the *Feature extraction* module by [Vlad Niculae](#)
- Implementation of `linear_model.LassoLarsCV` (cross-validated Lasso solver using the Lars algorithm) and `linear_model.LassoLarsIC` (BIC/AIC model selection in Lars) by [Gael Varoquaux](#) and [Alexandre Gramfort](#)
- Scalability improvements to `metrics.roc_curve` by [Olivier Hervieu](#)
- Distance helper functions `metrics.pairwise.pairwise_distances` and `metrics.pairwise.pairwise_kernels` by Robert Layton
- Mini-Batch K-Means by Nelle Varoquaux and Peter Prettenhofer.
- *mldata* utilities by Pietro Berkes.
- *olivetti\_faces* by [David Warde-Farley](#).

### 3.13.2 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\bsklearn/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\bcross_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.

### 3.13.3 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 Mitaireau
- 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

## 3.14 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*, *Partial Least Squares*, *Non-negative matrix factorization (NMF or NNMF)*, initial support for Python 3 and by important enhancements and bug fixes.

### 3.14.1 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
- *labeled\_faces\_in\_the\_wild* by Olivier Grisel.
- New *Partial Least Squares* module by Edouard Duchiensay.
- *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 `covered` to Gaussian Mixture Models by [Vincent Schut](#).
- Implement `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.ll_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](#).

### 3.14.2 People

People that made this release possible preceeded 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

## 3.15 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.

### 3.15.1 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]

### 3.15.2 People

People that made this release possible preceeded 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 Duchiensay](#)
- 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](#)

## 3.16 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.

### 3.16.1 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.

## 3.16.2 People

People that made this release possible preceeded 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

## 3.17 0.5

### 3.17.1 Changelog

### 3.17.2 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` 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`

### 3.17.3 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](#).

### 3.17.4 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.

### 3.17.5 Examples

- new examples using some of the mlcomp datasets: *Classification of text documents: using a MLComp dataset*, *Classification of text documents using sparse features*
- Many more examples. See [here](#) the full list of examples.

### 3.17.6 External dependencies

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

### 3.17.7 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.

### 3.17.8 Misc

- New sphinx theme for the web page.

### 3.17.9 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

## 3.18 0.4

### 3.18.1 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 content management 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 ...

## 3.18.2 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

## 3.19 Earlier versions

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

## 3.20 Presentations and tutorials on scikit-learn

For written tutorials, see the *Tutorial section* of the documentation.

### 3.20.1 Videos

- [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.

# BIBLIOGRAPHY

- [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 online learning and an application to boosting”, 1997.
- [ZZRH2009] J. Zhu, H. Zou, S. Rosset, T. Hastie. “Multi-class AdaBoost”, 2009.
- [D1997] 8. Drucker. “Improving Regressor using Boosting Techniques”, 1997.
- [HTF2009] 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] Identification 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)
- [R82] 12. Breiman, “Random Forests”, Machine Learning, 45(1), 5-32, 2001.
- [R83] 12. Breiman, “Random Forests”, Machine Learning, 45(1), 5-32, 2001.
- [R80] P. Geurts, D. Ernst., and L. Wehenkel, “Extremely randomized trees”, Machine Learning, 63(1), 3-42, 2006.
- [R81] 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
- [Rousseeuw1984] P. J. Rousseeuw. Least median of squares regression. *J. Am Stat Ass*, 79:871, 1984.
- [Rousseeuw1999] 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
- [R67] I. Guyon, “Design of experiments for the NIPS 2003 variable selection benchmark”, 2003.
- [R68] J. Friedman, “Multivariate adaptive regression splines”, *The Annals of Statistics* 19 (1), pages 1-67, 1991.
- [R69] L. Breiman, “Bagging predictors”, *Machine Learning* 24, pages 123-140, 1996.
- [R70] J. Friedman, “Multivariate adaptive regression splines”, *The Annals of Statistics* 19 (1), pages 1-67, 1991.
- [R71] L. Breiman, “Bagging predictors”, *Machine Learning* 24, pages 123-140, 1996.
- [R72] J. Friedman, “Multivariate adaptive regression splines”, *The Annals of Statistics* 19 (1), pages 1-67, 1991.
- [R73] L. Breiman, “Bagging predictors”, *Machine Learning* 24, pages 123-140, 1996.
- [R74] 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.
- [R75] S. Marsland, “Machine Learning: An Algorithmic Perpsective”, Chapter 10, 2009. <http://www-ist.massey.ac.nz/smarsland/Code/10/lle.py>
- [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
- [R82] 12. Breiman, “Random Forests”, *Machine Learning*, 45(1), 5-32, 2001.
- [R84] P. Geurts, D. Ernst., and L. Wehenkel, “Extremely randomized trees”, *Machine Learning*, 63(1), 3-42, 2006.
- [R85] Moosmann, F. and Triggs, B. and Jurie, F. “Fast discriminative visual codebooks using randomized clustering forests” NIPS 2007
- [R83] 12. Breiman, “Random Forests”, *Machine Learning*, 45(1), 5-32, 2001.
- [R80] P. Geurts, D. Ernst., and L. Wehenkel, “Extremely randomized trees”, *Machine Learning*, 63(1), 3-42, 2006.
- [R81] P. Geurts, D. Ernst., and L. Wehenkel, “Extremely randomized trees”, *Machine Learning*, 63(1), 3-42, 2006.
- [R76] Y. Freund, R. Schapire, “A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting”, 1995.
- [R77] 10. Zhu, H. Zou, S. Rosset, T. Hastie, “Multi-class AdaBoost”, 2009.
- [R78] Y. Freund, R. Schapire, “A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting”, 1995.
- [R79] 8. Drucker, “Improving Regressor using Boosting Techniques”, 1997.
- [Yates2011] R. Baeza-Yates and B. Ribeiro-Neto (2011). *Modern Information Retrieval*. Addison Wesley, pp. 68–74.

- [MSR2008] *C.D. Manning, H. Schütze and P. Raghavan (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 121–125.*
- [R86] 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.
- [R87] 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>
- [R88] Roweis, S. & Saul, L. Nonlinear dimensionality reduction by locally linear embedding. *Science* 290:2323 (2000).
- [R89] Donoho, D. & Grimes, C. Hessian eigenmaps: Locally linear embedding techniques for high-dimensional data. *Proc Natl Acad Sci U S A.* 100:5591 (2003).
- [R90] 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>
- [R91] Zhang, Z. & Zha, H. Principal manifolds and nonlinear dimensionality reduction via tangent space alignment. *Journal of Shanghai Univ.* 8:406 (2004)
- [R92] Roweis, S. & Saul, L. Nonlinear dimensionality reduction by locally linear embedding. *Science* 290:2323 (2000).
- [R93] Donoho, D. & Grimes, C. Hessian eigenmaps: Locally linear embedding techniques for high-dimensional data. *Proc Natl Acad Sci U S A.* 100:5591 (2003).
- [R94] 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>
- [R95] Zhang, Z. & Zha, H. Principal manifolds and nonlinear dimensionality reduction via tangent space alignment. *Journal of Shanghai Univ.* 8:406 (2004)
- [R100] Baldi, Brunak, Chauvin, Andersen and Nielsen, (2000). Assessing the accuracy of prediction algorithms for classification: an overview
- [R101] Wikipedia entry for the Matthews Correlation Coefficient
- [R96] Vinh, Epps, and Bailey, (2010). Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance, *JMLR*
- [R97] 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](http://en.wikipedia.org/wiki/Rand_index#Adjusted_Rand_index)
- [R98] Andrew Rosenberg and Julia Hirschberg, 2007. V-Measure: A conditional entropy-based external cluster evaluation measure
- [R99] Andrew Rosenberg and Julia Hirschberg, 2007. V-Measure: A conditional entropy-based external cluster evaluation measure
- [R104] Peter J. Rousseeuw (1987). “Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis”. *Computational and Applied Mathematics* 20: 53-65.
- [R105] Wikipedia entry on the Silhouette Coefficient

- [R102] Peter J. Rousseeuw (1987). “Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis”. Computational and Applied Mathematics 20: 53-65.
- [R103] Wikipedia entry on the Silhouette Coefficient
- [R106] Andrew Rosenberg and Julia Hirschberg, 2007. V-Measure: A conditional entropy-based external cluster evaluation measure
- [R107] “Solving multiclass learning problems via error-correcting output codes”, Dietterich T., Bakiri G., Journal of Artificial Intelligence Research 2, 1995.
- [R108] “The error coding method and PICTs”, James G., Hastie T., Journal of Computational and Graphical statistics 7, 1998.
- [R109] “The Elements of Statistical Learning”, Hastie T., Tibshirani R., Friedman J., page 606 (second-edition) 2008.
- [R110] Ping Li, T. Hastie and K. W. Church, 2006, “Very Sparse Random Projections”.  
[http://www.stanford.edu/~hastie/Papers/Ping/KDD06\\_rp.pdf](http://www.stanford.edu/~hastie/Papers/Ping/KDD06_rp.pdf)
- [R111] D. Achlioptas, 2001, “Database-friendly random projections”, <http://www.cs.ucsc.edu/~optas/papers/jl.pdf>
- [R112] [http://en.wikipedia.org/wiki/Johnson%E2%80%93Lindenstrauss\\_lemma](http://en.wikipedia.org/wiki/Johnson%E2%80%93Lindenstrauss_lemma)
- [R113] 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>
- [R114] [http://en.wikipedia.org/wiki/Decision\\_tree\\_learning](http://en.wikipedia.org/wiki/Decision_tree_learning)
- [R115] L. Breiman, J. Friedman, R. Olshen, and C. Stone, “Classification and Regression Trees”, Wadsworth, Belmont, CA, 1984.
- [R116] T. Hastie, R. Tibshirani and J. Friedman. “Elements of Statistical Learning”, Springer, 2009.
- [R117] L. Breiman, and A. Cutler, “Random Forests”, [http://www.stat.berkeley.edu/~breiman/RandomForests/cc\\_home.htm](http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm)
- [R118] [http://en.wikipedia.org/wiki/Decision\\_tree\\_learning](http://en.wikipedia.org/wiki/Decision_tree_learning)
- [R119] L. Breiman, J. Friedman, R. Olshen, and C. Stone, “Classification and Regression Trees”, Wadsworth, Belmont, CA, 1984.
- [R120] T. Hastie, R. Tibshirani and J. Friedman. “Elements of Statistical Learning”, Springer, 2009.
- [R121] L. Breiman, and A. Cutler, “Random Forests”, [http://www.stat.berkeley.edu/~breiman/RandomForests/cc\\_home.htm](http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm)
- [R122] P. Geurts, D. Ernst., and L. Wehenkel, “Extremely randomized trees”, Machine Learning, 63(1), 3-42, 2006.
- [R123] P. Geurts, D. Ernst., and L. Wehenkel, “Extremely randomized trees”, Machine Learning, 63(1), 3-42, 2006.
- [HTF2009] T. Hastie, R. Tibshirani and J. Friedman, “Elements of Statistical Learning Ed. 2”, Springer, 2009.

# PYTHON MODULE INDEX

## S

```
sklearn.cluster,??  
sklearn.covariance,??  
sklearn.cross_validation,??  
sklearn.datasets,??  
sklearn.decomposition,??  
sklearn.dummy,??  
sklearn.ensemble,??  
sklearn.ensemble.partial_dependence,??  
sklearn.feature_extraction,??  
sklearn.feature_extraction.image,??  
sklearn.feature_extraction.text,??  
sklearn.feature_selection,??  
sklearn.gaussian_process,??  
sklearn.grid_search,??  
sklearn.hmm,??  
sklearn.isotonic,??  
sklearn.kernel_approximation,??  
sklearn lda,??  
sklearn.linear_model,??  
sklearn.manifold,??  
sklearn.metrics,??  
sklearn.metrics.cluster,??  
sklearn.metrics.pairwise,??  
sklearn.mixture,??  
sklearn.multiclass,??  
sklearn.naive_bayes,??  
sklearn.neighbors,??  
sklearn.pipeline,??  
sklearn.pls,??  
sklearn.preprocessing,??  
sklearn.qda,??  
sklearn.random_projection,??  
sklearn.semi_supervised,??  
sklearn.svm,??  
sklearn.tree,??  
sklearn.utils,??
```



# PYTHON MODULE INDEX

## S

```
sklearn.cluster,??  
sklearn.covariance,??  
sklearn.cross_validation,??  
sklearn.datasets,??  
sklearn.decomposition,??  
sklearn.dummy,??  
sklearn.ensemble,??  
sklearn.ensemble.partial_dependence,??  
sklearn.feature_extraction,??  
sklearn.feature_extraction.image,??  
sklearn.feature_extraction.text,??  
sklearn.feature_selection,??  
sklearn.gaussian_process,??  
sklearn.grid_search,??  
sklearn.hmm,??  
sklearn.isotonic,??  
sklearn.kernel_approximation,??  
sklearn lda,??  
sklearn.linear_model,??  
sklearn.manifold,??  
sklearn.metrics,??  
sklearn.metrics.cluster,??  
sklearn.metrics.pairwise,??  
sklearn.mixture,??  
sklearn.multiclass,??  
sklearn.naive_bayes,??  
sklearn.neighbors,??  
sklearn.pipeline,??  
sklearn.pls,??  
sklearn.preprocessing,??  
sklearn.qda,??  
sklearn.random_projection,??  
sklearn.semi_supervised,??  
sklearn.svm,??  
sklearn.tree,??  
sklearn.utils,??
```