

SCIKIT-LEARN TUTORIALS

2.1 An introduction to machine learning with scikit-learn

Section contents

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

2.1.1 Machine learning: the problem setting

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

Learning problems fall into a few categories:

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

Training set and testing set

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

2.1.2 Loading an example dataset

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

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

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

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

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

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

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

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

Shape of the data arrays

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

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

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

Loading from external datasets

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

2.1.3 Learning and predicting

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

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

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

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

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

Choosing the parameters of the model

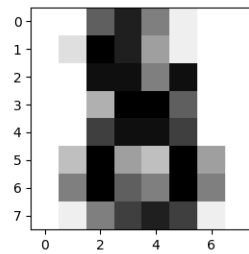
In this example, we set the value of `gamma` manually. To find good values for these parameters, we can use tools such as *grid search* and *cross validation*.

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

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

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

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



The corresponding image is:
are of poor resolution. Do you agree with the classifier?

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

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

2.1.4 Model persistence

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

```
>>> from sklearn import svm
>>> from sklearn import datasets
>>> clf = svm.SVC(gamma='scale')
>>> 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,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)

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

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

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

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

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

Note: *joblib.dump* and *joblib.load* functions also accept file-like object instead of filenames. More information on data persistence with *Joblib* is available [here](#).

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

2.1.5 Conventions

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

Type casting

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

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

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

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

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

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

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

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

>>> clf.fit(iris.data, iris.target_names[iris.target])
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)

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

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

Refitting and updating parameters

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

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

>>> clf = SVC()
>>> clf.set_params(kernel='linear').fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
    kernel='linear', max_iter=-1, probability=False, random_state=None,
    shrinking=True, tol=0.001, verbose=False)
>>> clf.predict(X[:5])
array([0, 0, 0, 0, 0])

>>> clf.set_params(kernel='rbf', gamma='scale').fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
>>> clf.predict(X[:5])
array([0, 0, 0, 0, 0])
```

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

Multiclass vs. multilabel fitting

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

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

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

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

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

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

Here, the classifier is `fit()` on a 2d binary label representation of `y`, using the `LabelBinarizer`. In this case `predict()` returns a 2d array representing the corresponding multilabel predictions.

Note that the fourth and fifth instances returned all zeroes, indicating that they matched none of the three labels `fit` upon. With multilabel outputs, it is similarly possible for an instance to be assigned multiple labels:

```
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> y = [[0, 1], [0, 2], [1, 3], [0, 2, 3], [2, 4]]
>>> y = MultiLabelBinarizer().fit_transform(y)
>>> classif.fit(X, y).predict(X)
array([[1, 1, 0, 0, 0],
       [1, 0, 1, 0, 0],
       [0, 1, 0, 1, 0],
       [1, 0, 1, 0, 0],
       [1, 0, 1, 0, 0]])
```

In this case, the classifier is fit upon instances each assigned multiple labels. The `MultiLabelBinarizer` is used to binarize the 2d array of multilabels to fit upon. As a result, `predict()` returns a 2d array with multiple predicted labels for each instance.

2.2 A tutorial on statistical-learning for scientific data processing

Statistical learning

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

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

Scikit-learn is a Python module integrating classic machine learning algorithms in the tightly-knit world of scientific Python packages (**NumPy**, **SciPy**, **matplotlib**).

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

Datasets

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

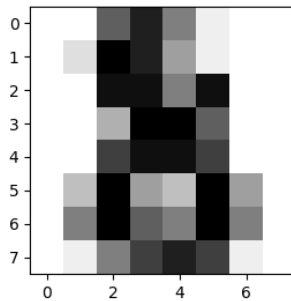
A simple example shipped with scikit-learn: iris dataset

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> data = iris.data
>>> data.shape
(150, 4)
```

It is made of 150 observations of irises, each described by 4 features: their sepal and petal length and width, as detailed in `iris.DESCR`.

When the data is not initially in the `(n_samples, n_features)` shape, it needs to be preprocessed in order to be used by scikit-learn.

An example of reshaping data would be the digits dataset



The digits dataset is made of 1797 8x8 images of hand-written digits

```
>>> digits = datasets.load_digits()
>>> digits.images.shape
(1797, 8, 8)
>>> import matplotlib.pyplot as plt
>>> plt.imshow(digits.images[-1], cmap=plt.cm.gray_r)
<matplotlib.image.AxesImage object at ...>
```

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

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

Estimators objects

Fitting data: the main API implemented by scikit-learn is that of the *estimator*. An estimator is any object that learns from data; it may be a classification, regression or clustering algorithm or a *transformer* that extracts/filters useful features from raw data.

All estimator objects expose a `fit` method that takes a dataset (usually a 2-d array):

```
>>> estimator.fit(data)
```

Estimator parameters: All the parameters of an estimator can be set when it is instantiated or by modifying the corresponding attribute:

```
>>> estimator = Estimator(param1=1, param2=2)
>>> estimator.param1
1
```

Estimated parameters: When data is fitted with an estimator, parameters are estimated from the data at hand. All the estimated parameters are attributes of the estimator object ending by an underscore:

```
>>> estimator.estimated_param_
```


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

The problem solved in supervised learning

Supervised learning consists in learning the link between two datasets: the observed data X and an external variable y that we are trying to predict, usually called “target” or “labels”. Most often, y is a 1D array of length `n_samples`.

All supervised *estimators* in scikit-learn implement a `fit(X, y)` method to fit the model and a `predict(X)` method that, given unlabeled observations X , returns the predicted labels y .

Vocabulary: classification and regression

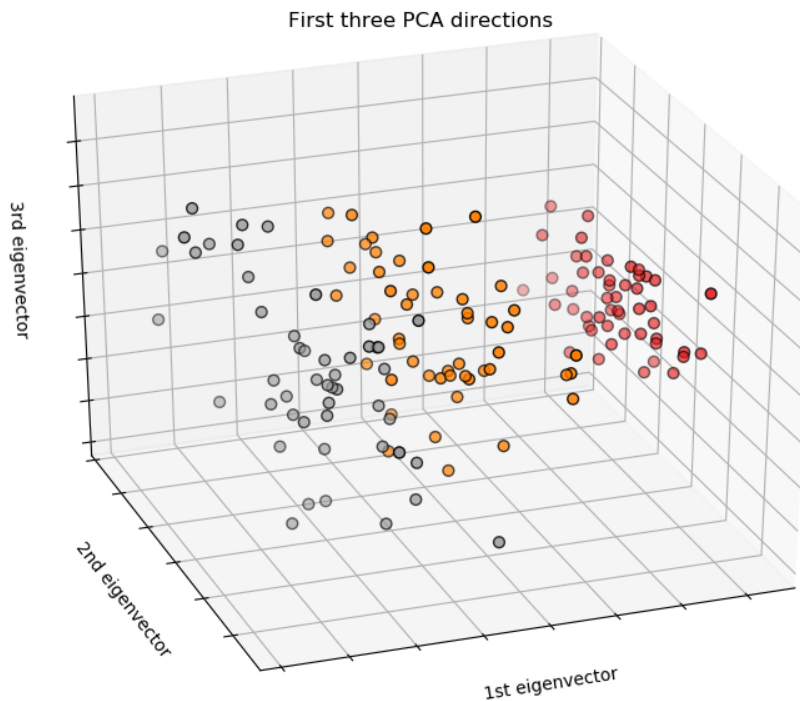
If the prediction task is to classify the observations in a set of finite labels, in other words to “name” the objects observed, the task is said to be a **classification** task. On the other hand, if the goal is to predict a continuous target variable, it is said to be a **regression** task.

When doing classification in scikit-learn, y is a vector of integers or strings.

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

Nearest neighbor and the curse of dimensionality

Classifying irises:



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

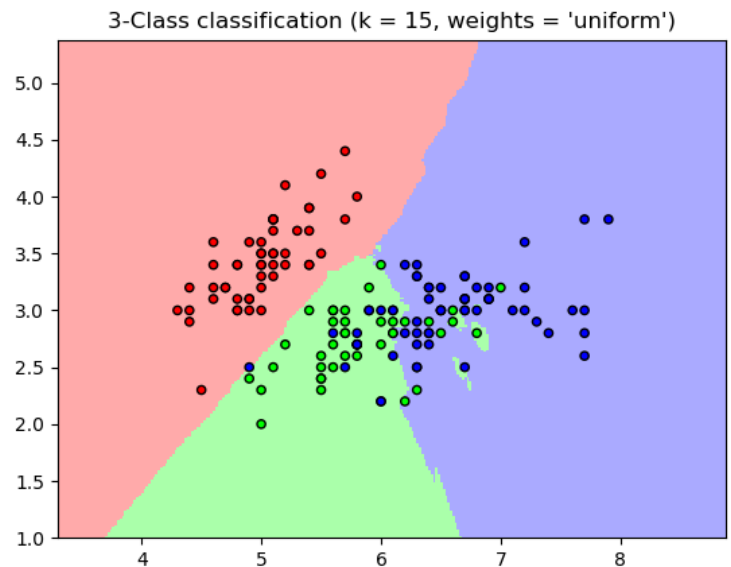
```
>>> import numpy as np
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> iris_X = iris.data
>>> iris_y = iris.target
>>> np.unique(iris_y)
array([0, 1, 2])
```

k-Nearest neighbors classifier

The simplest possible classifier is the [nearest neighbor](#): given a new observation X_{test} , find in the training set (i.e. the data used to train the estimator) the observation with the closest feature vector. (Please see the [Nearest Neighbors section](#) of the online Scikit-learn documentation for more information about this type of classifier.)

Training set and testing set

While experimenting with any learning algorithm, it is important not to test the prediction of an estimator on the data used to fit the estimator as this would not be evaluating the performance of the estimator on **new data**. This is why datasets are often split into *train* and *test* data.



KNN (k nearest neighbors) classification example:

```
>>> # Split iris data in train and test data
>>> # A random permutation, to split the data randomly
>>> np.random.seed(0)
>>> indices = np.random.permutation(len(iris_X))
>>> iris_X_train = iris_X[indices[:-10]]
>>> iris_y_train = iris_y[indices[:-10]]
>>> iris_X_test = iris_X[indices[-10:]]
>>> iris_y_test = iris_y[indices[-10:]]
>>> # Create and fit a nearest-neighbor classifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier()
>>> knn.fit(iris_X_train, iris_y_train)
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                    weights='uniform')
>>> knn.predict(iris_X_test)
array([1, 2, 1, 0, 0, 0, 2, 1, 2, 0])
>>> iris_y_test
array([1, 1, 1, 0, 0, 0, 2, 1, 2, 0])
```

The curse of dimensionality

For an estimator to be effective, you need the distance between neighboring points to be less than some value d , which depends on the problem. In one dimension, this requires on average $n \sim 1/d$ points. In the context of the above k -NN example, if the data is described by just one feature with values ranging from 0 to 1 and with n training observations, then new data will be no further away than $1/n$. Therefore, the nearest neighbor decision rule will be efficient as soon as $1/n$ is small compared to the scale of between-class feature variations.

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

For example, if each point is just a single number (8 bytes), then an effective k -NN estimator in a paltry $p \sim 20$ dimensions would require more training data than the current estimated size of the entire internet (± 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

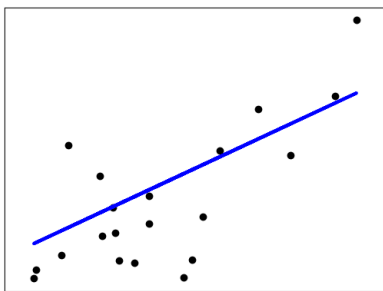
The diabetes dataset consists of 10 physiological variables (age, sex, weight, blood pressure) measure on 442 patients, and an indication of disease progression after one year:

```
>>> 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 its simplest form, fits a linear model to the data set by adjusting a set of parameters in order to make the sum of the squared residuals of the model as small as possible.



Linear models: $y = X\beta + \epsilon$

- X : data
- y : target variable
- β : Coefficients
- ϵ : Observation noise

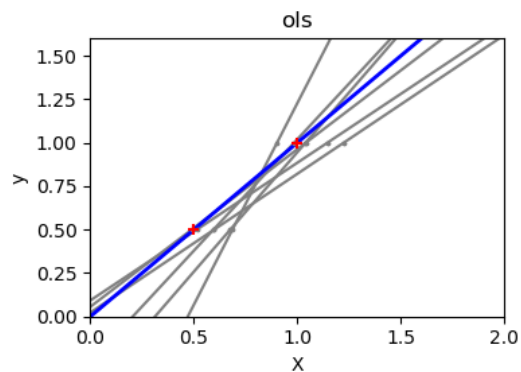
```
>>> from sklearn import linear_model
>>> regr = linear_model.LinearRegression()
>>> regr.fit(diabetes_X_train, diabetes_y_train)
...
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                  normalize=False)
>>> print(regr.coef_)
[  0.30349955 -237.63931533  510.53060544  327.73698041 -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:

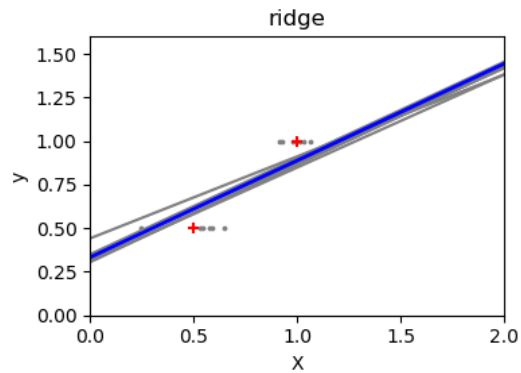


```
>>> X = np.c_[.5, 1].T
>>> y = [.5, 1]
>>> test = np.c_[0, 2].T
>>> regr = linear_model.LinearRegression()

>>> import matplotlib.pyplot as plt
>>> plt.figure()

>>> np.random.seed(0)
>>> for _ in range(6):
...     this_X = .1 * np.random.normal(size=(2, 1)) + X
...     regr.fit(this_X, y)
...     plt.plot(test, regr.predict(test))
...     plt.scatter(this_X, y, s=3)
```

A solution in high-dimensional statistical learning is to *shrink* the regression coefficients to zero: any two randomly chosen set of observations are likely to be uncorrelated. This is called *Ridge* regression:



```
>>> regr = linear_model.Ridge(alpha=.1)

>>> plt.figure()

>>> np.random.seed(0)
>>> for _ in range(6):
...     this_X = .1 * np.random.normal(size=(2, 1)) + X
...     regr.fit(this_X, y)
...     plt.plot(test, regr.predict(test))
...     plt.scatter(this_X, y, s=3)
```

This is an example of **bias/variance tradeoff**: the larger the ridge alpha parameter, the higher the bias and the lower the variance.

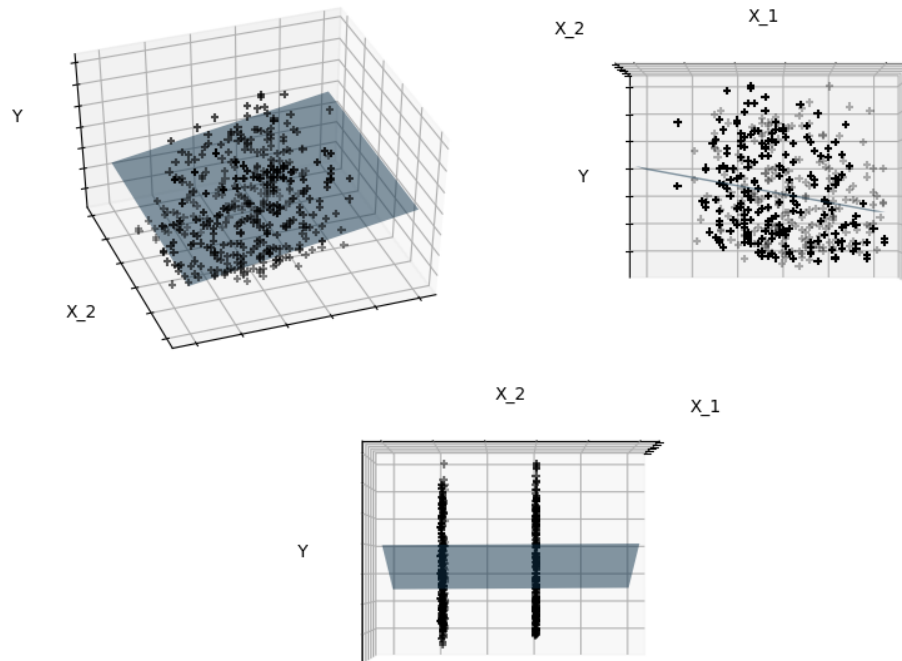
We can choose alpha to minimize left out error, this time using the diabetes dataset rather than our synthetic data:

```
>>> alphas = np.logspace(-4, -1, 6)
>>> print([regr.set_params(alpha=alpha)
...       .fit(diabetes_X_train, diabetes_y_train)
...       .score(diabetes_X_test, diabetes_y_test)
...       for alpha in alphas])
...
[0.5851110683883..., 0.5852073015444..., 0.5854677540698...,
 0.5855512036503..., 0.5830717085554..., 0.57058999437...]
```

Note: Capturing in the fitted parameters noise that prevents the model to generalize to new data is called [overfitting](#). The bias introduced by the ridge regression is called a [regularization](#).

Sparsity

Fitting only features 1 and 2



Note: A representation of the full diabetes dataset would involve 11 dimensions (10 feature dimensions and one of the target variable). It is hard to develop an intuition on such representation, but it may be useful to keep in mind that it would be a fairly *empty* space.

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

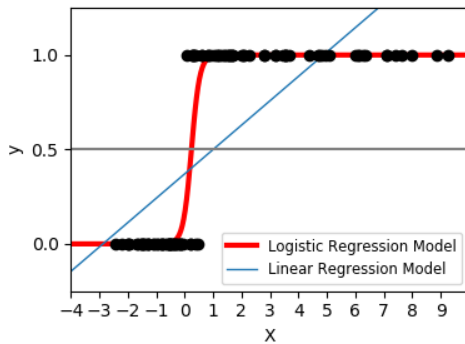
To improve the conditioning of the problem (i.e. mitigating the *The curse of dimensionality*), it would be interesting to select only the informative features and set non-informative ones, like feature 2 to 0. Ridge regression will decrease their contribution, but not set them to zero. Another penalization approach, called *Lasso* (least absolute shrinkage and selection operator), can set some coefficients to zero. Such methods are called **sparse method** and sparsity can be seen as an application of Occam's razor: *prefer simpler models*.

```
>>> regr = linear_model.Lasso()
>>> scores = [regr.set_params(alpha=alpha)
...           .fit(diabetes_X_train, diabetes_y_train)
...           .score(diabetes_X_test, diabetes_y_test)
...           for alpha in alphas]
>>> best_alpha = alphas[scores.index(max(scores))]
>>> regr.alpha = best_alpha
>>> regr.fit(diabetes_X_train, diabetes_y_train)
...
Lasso(alpha=0.025118864315095794, copy_X=True, fit_intercept=True,
      max_iter=1000, normalize=False, positive=False, precompute=False,
      random_state=None, selection='cyclic', tol=0.0001, warm_start=False)
>>> print(regr.coef_)
[  0.         -212.43764548  517.19478111  313.77959962 -160.8303982   -0.
 -187.19554705   69.38229038  508.66011217   71.84239008]
```

Different algorithms for the same problem

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

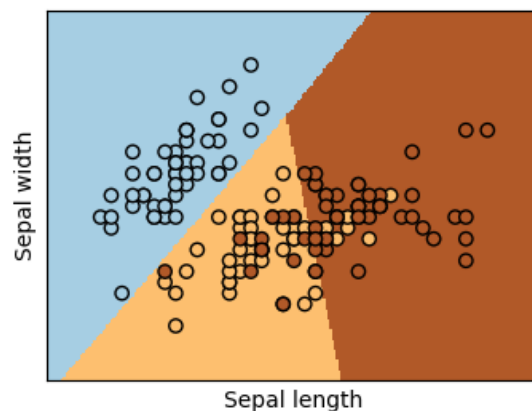
Classification



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

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

```
>>> log = linear_model.LogisticRegression(solver='lbfgs', C=1e5,
...                                     multi_class='multinomial')
>>> log.fit(iris_X_train, iris_y_train)
LogisticRegression(C=100000.0, class_weight=None, dual=False,
fit_intercept=True, intercept_scaling=1, l1_ratio=None, max_iter=100,
multi_class='multinomial', n_jobs=None, penalty='l2', random_state=None,
solver='lbfgs', tol=0.0001, verbose=0, warm_start=False)
```



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

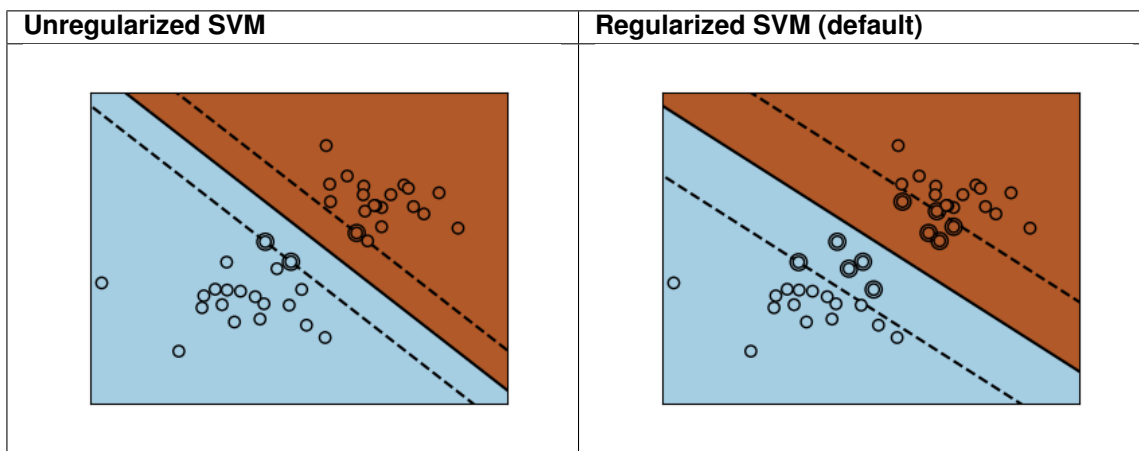
digits = datasets.load_digits()
X_digits = digits.data / digits.data.max()
y_digits = digits.target
```

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).



Example:

- *Plot different SVM classifiers in the iris dataset*

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

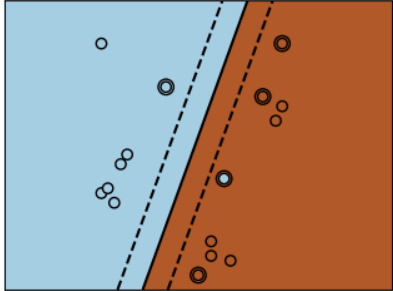
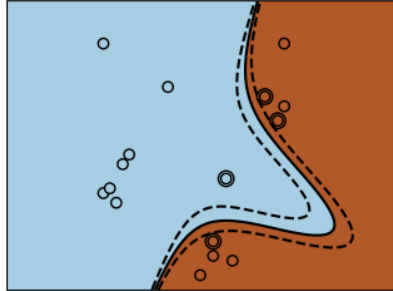
```
>>> from sklearn import svm
>>> svc = svm.SVC(kernel='linear')
>>> svc.fit(iris_X_train, iris_y_train)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
    kernel='linear', max_iter=-1, probability=False, random_state=None,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

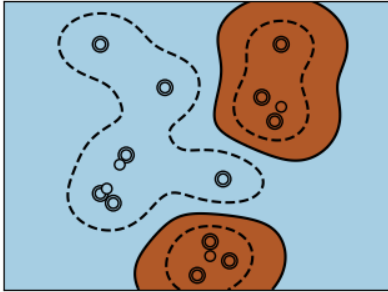
Warning: Normalizing data

For many estimators, including the SVMs, having datasets with unit standard deviation for each feature is important to get good prediction.

Using kernels

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

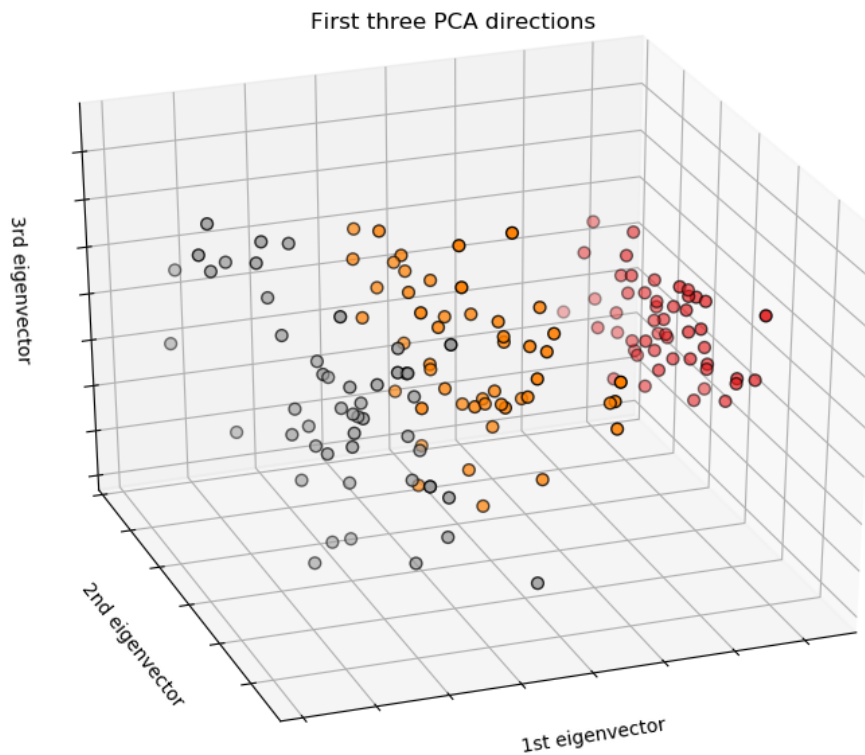
| Linear kernel | Polynomial kernel |
|---|---|
|  |  |
| <pre>>>> svc = svm.SVC(kernel='linear')</pre> | <pre>>>> svc = svm.SVC(kernel='poly', ... degree=3) >>> # degree: polynomial degree</pre> |

RBF kernel (Radial Basis Function)

```
>>> svc = svm.SVC(kernel='rbf')
>>> # gamma: inverse of size of
>>> # radial kernel
```

Interactive example

See the [SVM GUI](#) to download `svm_gui.py`; add data points of both classes with right and left button, fit the model and change parameters and data.



Exercise

Try classifying classes 1 and 2 from the iris dataset with SVMs, with the 2 first features. Leave out 10% of each class and test prediction performance on these observations.

Warning: the classes are ordered, do not leave out the last 10%, you would be testing on only one class.

Hint: You can use the `decision_function` method on a grid to get intuitions.

```
iris = datasets.load_iris()
X = iris.data
y = iris.target
```

```
X = X[y != 0, :2]
y = y[y != 0]
```

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

2.2.3 Model selection: choosing estimators and their parameters

Score, and cross-validated scores

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

```
>>> from sklearn import datasets, svm
>>> 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:])
0.98
```

To get a better measure of prediction accuracy (which we can use as a proxy for goodness of fit of the model), we can successively split the data in *folds* that we use for training and testing:

```
>>> import numpy as np
>>> X_folds = np.array_split(X_digits, 3)
>>> y_folds = np.array_split(y_digits, 3)
>>> scores = list()
>>> for k in range(3):
...     # We use 'list' to copy, in order to 'pop' later on
...     X_train = list(X_folds)
...     X_test = X_train.pop(k)
...     X_train = np.concatenate(X_train)
...     y_train = list(y_folds)
...     y_test = y_train.pop(k)
...     y_train = np.concatenate(y_train)
...     scores.append(svc.fit(X_train, y_train).score(X_test, y_test))
>>> print(scores)
[0.934..., 0.956..., 0.939...]
```

This is called a *KFold* cross-validation.

Cross-validation generators

Scikit-learn has a collection of classes which can be used to generate lists of train/test indices for popular cross-validation strategies.

They expose a `split` method which accepts the input dataset to be split and yields the train/test set indices for each iteration of the chosen cross-validation strategy.

This example shows an example usage of the `split` method.

```
>>> from sklearn.model_selection import KFold, cross_val_score
>>> X = ["a", "a", "a", "b", "b", "c", "c", "c", "c", "c"]
>>> k_fold = KFold(n_splits=5)
>>> for train_indices, test_indices in k_fold.split(X):
...     print('Train: %s | test: %s' % (train_indices, test_indices))
Train: [2 3 4 5 6 7 8 9] | test: [0 1]
Train: [0 1 4 5 6 7 8 9] | test: [2 3]
Train: [0 1 2 3 6 7 8 9] | test: [4 5]
Train: [0 1 2 3 4 5 8 9] | test: [6 7]
Train: [0 1 2 3 4 5 6 7] | test: [8 9]
```

The cross-validation can then be performed easily:

```
>>> [svc.fit(X_digits[train], y_digits[train]).score(X_digits[test], y_digits[test])
...   for train, test in k_fold.split(X_digits)]
[0.963..., 0.922..., 0.963..., 0.963..., 0.930...]
```

The cross-validation score can be directly calculated using the `cross_val_score` helper. Given an estimator, the cross-validation object and the input dataset, the `cross_val_score` splits the data repeatedly into a training and a testing set, trains the estimator using the training set and computes the scores based on the testing set for each iteration of cross-validation.

By default the estimator's `score` method is used to compute the individual scores.

Refer the [metrics module](#) to learn more on the available scoring methods.

```
>>> cross_val_score(svc, X_digits, y_digits, cv=k_fold, n_jobs=-1)
array([0.96388889, 0.92222222, 0.9637883 , 0.9637883 , 0.93036212])
```

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

Alternatively, the `scoring` argument can be provided to specify an alternative scoring method.

```
>>> cross_val_score(svc, X_digits, y_digits, cv=k_fold,
...                 scoring='precision_macro')
array([0.96578289, 0.92708922, 0.96681476, 0.96362897, 0.93192644])
```

Cross-validation generators

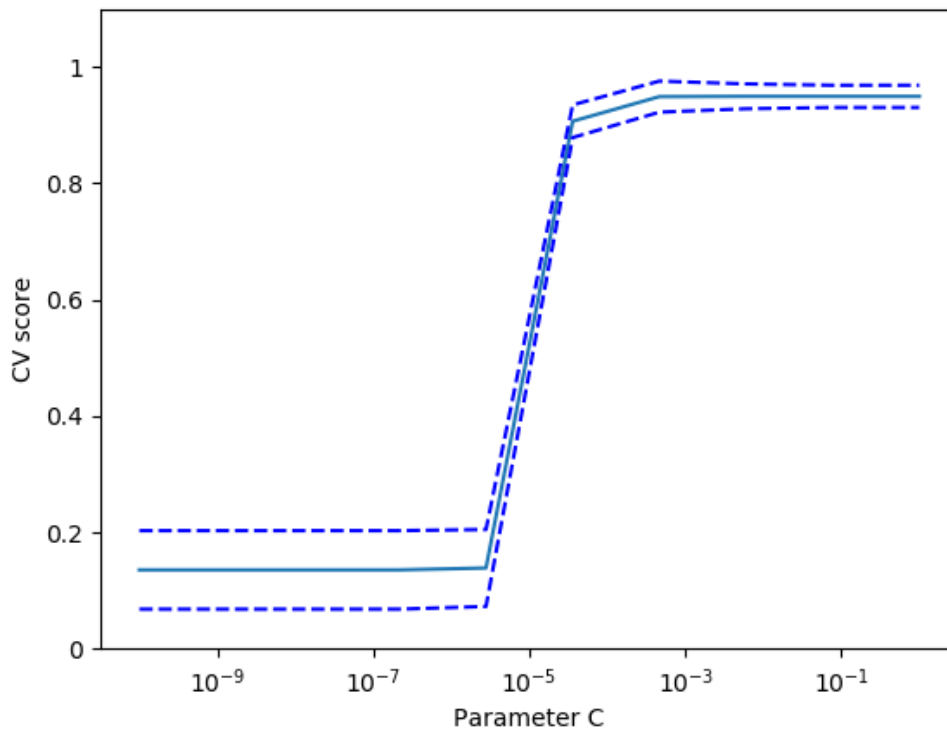
| <i>KFold</i> (<i>n_splits</i> , <i>shuffle</i> , <i>random_state</i>) | <i>StratifiedKFold</i> (<i>n_splits</i> , <i>shuffle</i> , <i>random_state</i>) | <i>GroupKFold</i> (<i>n_splits</i>) |
|---|---|---|
| Splits it into K folds, trains on K-1 and then tests on the left-out. | Same as K-Fold but preserves the class distribution within each fold. | Ensures that the same group is not in both testing and training sets. |

| | | |
|--|---|---|
| <i>ShuffleSplit</i> (n_splits , test_size , train_size , random_state) | <i>StratifiedShuffleSplit</i> | <i>GroupShuffleSplit</i> |
| Generates train/test indices based on random permutation. | Same as shuffle split but preserves the class distribution within each iteration. | Ensures that the same group is not in both testing and training sets. |

| | | |
|--|--|----------------------------|
| <i>LeaveOneGroupOut</i> () | <i>LeavePGroupsOut</i> (n_groups) | <i>LeaveOneOut</i> () |
| Takes a group array to group observations. | Leave P groups out. | Leave one observation out. |

| | |
|-------------------------------|--|
| <i>LeavePOut</i> (p) | <i>PredefinedSplit</i> |
| Leave P observations out. | Generates train/test indices based on predefined splits. |

Exercise



On the digits dataset, plot the cross-validation score of a *SVC* estimator with an linear kernel as a function of parameter C (use a logarithmic grid of points, from 1 to 10).

```
import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn import datasets, svm

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

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

Solution: *Cross-validation on Digits Dataset Exercise*

Grid-search and cross-validated estimators

Grid-search

scikit-learn provides an object that, given data, computes the score during the fit of an estimator on a parameter grid and chooses the parameters to maximize the cross-validation score. This object takes an estimator during the construction and exposes an estimator API:

```
>>> from sklearn.model_selection import GridSearchCV, cross_val_score
>>> Cs = np.logspace(-6, -1, 10)
>>> clf = GridSearchCV(estimator=svc, param_grid=dict(C=Cs),
...                   n_jobs=-1)
>>> clf.fit(X_digits[:1000], y_digits[:1000])
GridSearchCV(cv=None,...
>>> clf.best_score_
0.925...
>>> clf.best_estimator_.C
0.0077...

>>> # Prediction performance on test set is not as good as on train set
>>> clf.score(X_digits[1000:], y_digits[1000:])
0.943...
```

By default, the *GridSearchCV* uses a 3-fold cross-validation. However, if it detects that a classifier is passed, rather than a regressor, it uses a stratified 3-fold. The default will change to a 5-fold cross-validation in version 0.22.

Nested cross-validation

```
>>> cross_val_score(clf, X_digits, y_digits)
array([0.938..., 0.963..., 0.944...])
```

Two cross-validation loops are performed in parallel: one by the *GridSearchCV* estimator to set gamma and the other one by *cross_val_score* to measure the prediction performance of the estimator. The resulting scores are unbiased estimates of the prediction score on new data.

Warning: You cannot nest objects with parallel computing (*n_jobs* different than 1).

Cross-validated estimators

Cross-validation to set a parameter can be done more efficiently on an algorithm-by-algorithm basis. This is why, for certain estimators, scikit-learn exposes *Cross-validation: evaluating estimator performance* estimators that set their parameter automatically by cross-validation:

```
>>> from sklearn import linear_model, datasets
>>> lasso = linear_model.LassoCV(cv=3)
>>> 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=3, eps=0.001, fit_intercept=True,
        max_iter=1000, n_alphas=100, n_jobs=None, normalize=False,
        positive=False, precompute='auto', random_state=None,
        selection='cyclic', tol=0.0001, verbose=False)
>>> # The estimator chose automatically its lambda:
>>> lasso.alpha_
0.01229...
```

These estimators are called similarly to their counterparts, with ‘CV’ appended to their name.

Exercise

On the diabetes dataset, find the optimal regularization parameter alpha.

Bonus: How much can you trust the selection of alpha?

```
from sklearn import datasets
from sklearn.linear_model import LassoCV
from sklearn.linear_model import Lasso
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV

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

Solution: *Cross-validation on diabetes Dataset Exercise*

2.2.4 Unsupervised learning: seeking representations of the data

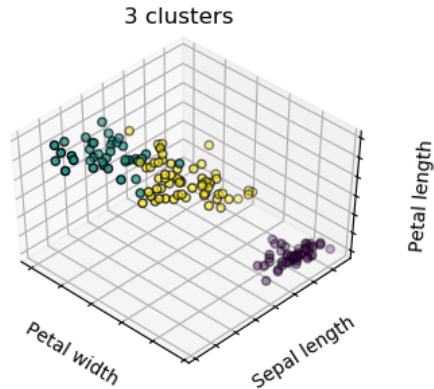
Clustering: grouping observations together

The problem solved in clustering

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

K-means clustering

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

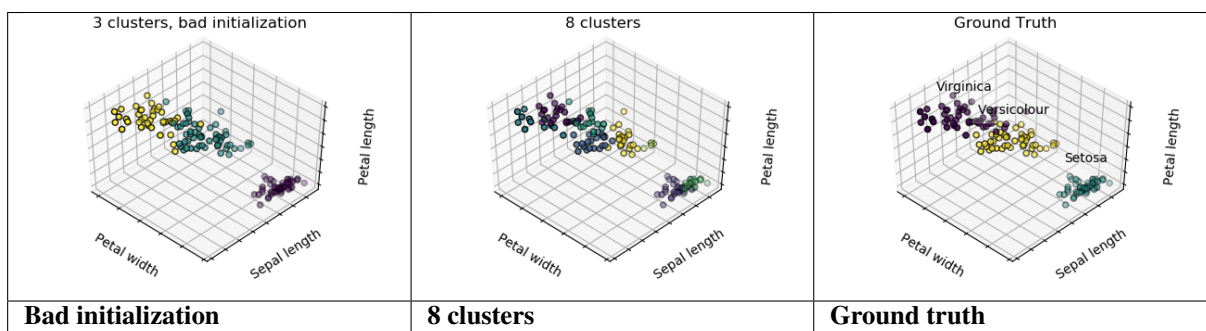


is *K-means*.

```
>>> from sklearn import cluster, datasets
>>> 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(algorithm='auto', copy_x=True, init='k-means++', ...)
>>> print(k_means.labels_[:10])
[1 1 1 1 0 0 0 0 2 2]
>>> print(y_iris[:10])
[0 0 0 0 1 1 1 1 2 2]
```

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



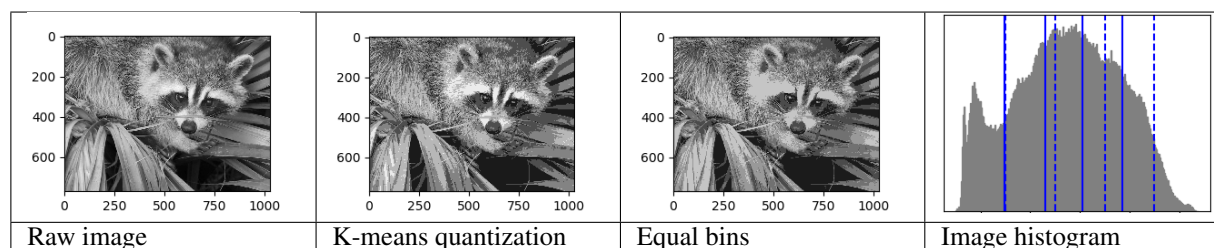
Don't over-interpret clustering results

Application example: vector quantization

Clustering in general and KMeans, in particular, can be seen as a way of choosing a small number of exemplars to compress the information. The problem is sometimes known as **vector quantization**. For instance, this can be used

to posterize an image:

```
>>> import scipy as sp
>>> try:
...     face = sp.face(gray=True)
... except AttributeError:
...     from scipy import misc
...     face = misc.face(gray=True)
>>> X = face.reshape((-1, 1)) # We need an (n_sample, n_feature) array
>>> k_means = cluster.KMeans(n_clusters=5, n_init=1)
>>> k_means.fit(X)
KMeans(algorithm='auto', copy_x=True, init='k-means++', ...
>>> values = k_means.cluster_centers_.squeeze()
>>> labels = k_means.labels_
>>> face_compressed = np.choose(labels, values)
>>> face_compressed.shape = face.shape
```



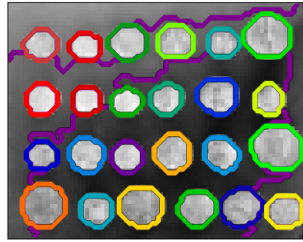
Hierarchical agglomerative clustering: Ward

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: each observation starts in its own cluster, and clusters are iteratively merged in such a way to minimize a *linkage* criterion. This approach is particularly interesting when the clusters of interest are made of only a few observations. When the number of clusters is large, it is much more computationally efficient than k-means.
- **Divisive** - top-down approaches: all observations start in one cluster, which is iteratively split as one moves down the hierarchy. For estimating large numbers of clusters, this approach is both slow (due to all observations starting as one cluster, which it splits recursively) and statistically ill-posed.

Connectivity-constrained clustering

With agglomerative clustering, it is possible to specify which samples can be clustered together by giving a connectivity graph. Graphs in scikit-learn are represented by their adjacency matrix. Often, a sparse matrix is used. This can be useful, for instance, to retrieve connected regions (sometimes also referred to as connected components) when



clustering an image:

```
from scipy.ndimage.filters import gaussian_filter

import matplotlib.pyplot as plt

import skimage
from skimage.data import coins
from skimage.transform import rescale

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

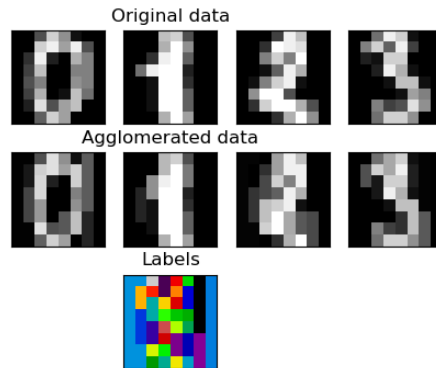
# these were introduced in skimage-0.14
if LooseVersion(skimage.__version__) >= '0.14':
    rescale_params = {'anti_aliasing': False, 'multichannel': False}
else:
    rescale_params = {}

# #####
# Generate data
orig_coins = coins()

# Resize it to 20% of the original size to speed up the processing
# Applying a Gaussian filter for smoothing prior to down-scaling
# reduces aliasing artifacts.
smoothened_coins = gaussian_filter(orig_coins, sigma=2)
```

Feature agglomeration

We have seen that sparsity could be used to mitigate the curse of dimensionality, *i.e.* an insufficient amount of observations compared to the number of features. Another approach is to merge together similar features: **feature agglomeration**. This approach can be implemented by clustering in the feature direction, in other words clustering



the transposed data.

```
>>> digits = datasets.load_digits()
>>> images = digits.images
>>> X = np.reshape(images, (len(images), -1))
>>> connectivity = grid_to_graph(*images[0].shape)

>>> agglo = cluster.FeatureAgglomeration(connectivity=connectivity,
...                                     n_clusters=32)
>>> agglo.fit(X)
FeatureAgglomeration(affinity='euclidean', compute_full_tree='auto',...
>>> X_reduced = agglo.transform(X)

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

transform and inverse_transform methods

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

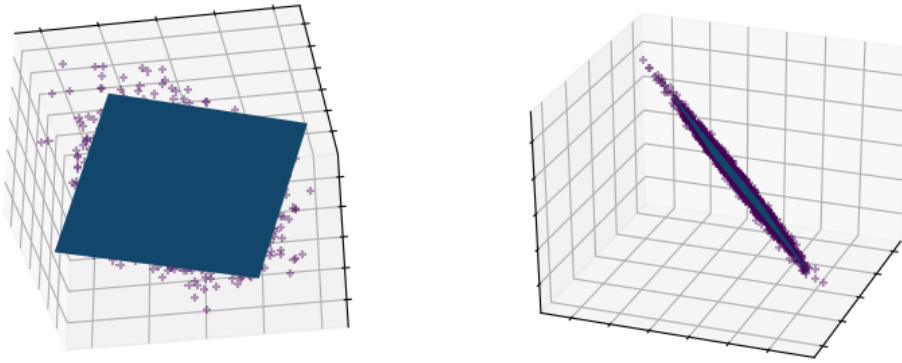
Decompositions: from a signal to components and loadings

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.

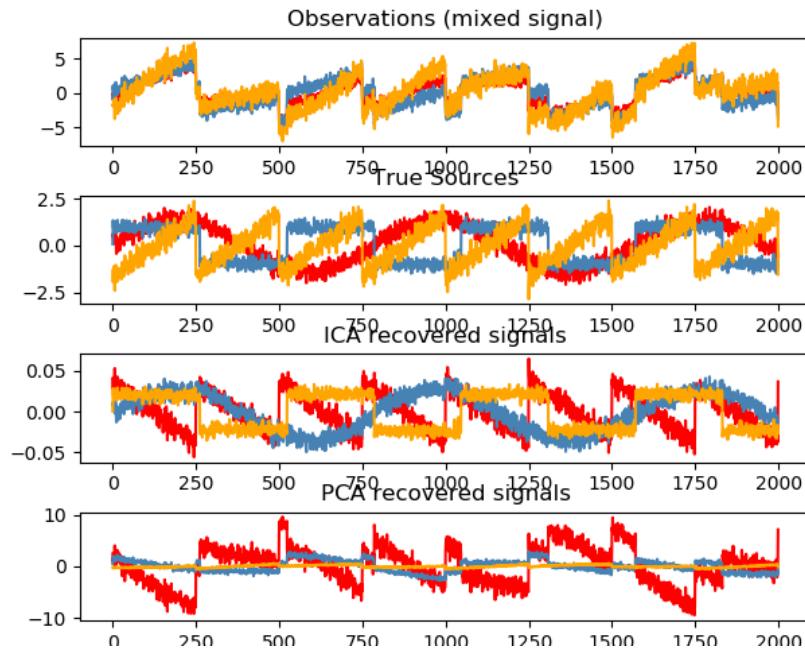
```
>>> # Create a signal with only 2 useful dimensions
>>> x1 = np.random.normal(size=100)
>>> x2 = np.random.normal(size=100)
>>> x3 = x1 + x2
>>> X = np.c_[x1, x2, x3]

>>> from sklearn import decomposition
>>> pca = decomposition.PCA()
>>> pca.fit(X)
PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)
>>> print(pca.explained_variance_)
[ 2.18565811e+00  1.19346747e+00  8.43026679e-32]

>>> # As we can see, only the 2 first components are useful
>>> pca.n_components = 2
>>> X_reduced = pca.fit_transform(X)
>>> X_reduced.shape
(100, 2)
```

Independent Component Analysis: ICA

Independent component analysis (ICA) selects components so that the distribution of their loadings carries a maximum amount of independent information. It is able to recover **non-Gaussian** independent signals:



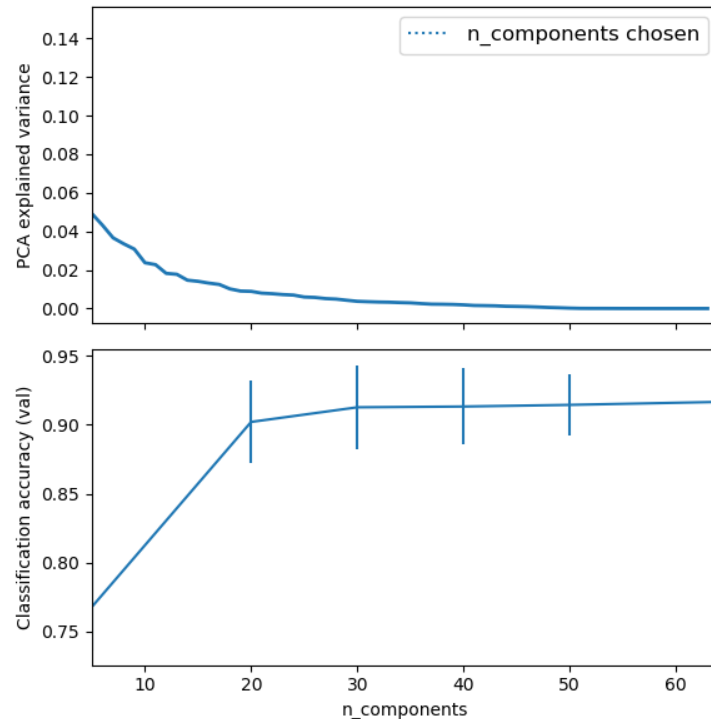
```
>>> # Generate sample data
>>> import numpy as np
>>> from scipy import signal
>>> time = np.linspace(0, 10, 2000)
>>> s1 = np.sin(2 * time) # Signal 1 : sinusoidal signal
>>> s2 = np.sign(np.sin(3 * time)) # Signal 2 : square signal
>>> s3 = signal.sawtooth(2 * np.pi * time) # Signal 3: saw tooth signal
>>> S = np.c_[s1, s2, s3]
>>> S += 0.2 * np.random.normal(size=S.shape) # Add noise
>>> S /= S.std(axis=0) # Standardize data
>>> # Mix data
>>> A = np.array([[1, 1, 1], [0.5, 2, 1], [1.5, 1, 2]]) # Mixing matrix
>>> X = np.dot(S, A.T) # Generate observations

>>> # Compute ICA
>>> ica = decomposition.FastICA()
>>> S_ = ica.fit_transform(X) # Get the estimated sources
>>> A_ = ica.mixing_.T
>>> np.allclose(X, np.dot(S_, A_) + ica.mean_)
True
```

2.2.5 Putting it all together

Pipelining

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



create combined estimators:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.linear_model import SGDClassifier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV

# Define a pipeline to search for the best combination of PCA truncation
# and classifier regularization.
logistic = SGDClassifier(loss='log', penalty='l2', early_stopping=True,
                        max_iter=10000, tol=1e-5, random_state=0)
pca = PCA()
pipe = Pipeline(steps=[('pca', pca), ('logistic', logistic)])

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

# Parameters of pipelines can be set using '__' separated parameter names:
param_grid = {
    'pca__n_components': [5, 20, 30, 40, 50, 64],
    'logistic__alpha': np.logspace(-4, 4, 5),
}
search = GridSearchCV(pipe, param_grid, iid=False, cv=5)
search.fit(X_digits, y_digits)
```

```

print("Best parameter (CV score=%0.3f):" % search.best_score_)
print(search.best_params_)

# Plot the PCA spectrum
pca.fit(X_digits)

fig, (ax0, ax1) = plt.subplots(nrows=2, sharex=True, figsize=(6, 6))
ax0.plot(pca.explained_variance_ratio_, linewidth=2)
ax0.set_ylabel('PCA explained variance')

ax0.axvline(search.best_estimator_.named_steps['pca'].n_components,
            linestyle=':', label='n_components chosen')

```

Face recognition with eigenfaces

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

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

```

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

The dataset used in this example is a preprocessed excerpt of the
"Labeled Faces in the Wild", aka LFW:

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

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

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

=====
precision    recall  f1-score   support

Ariel Sharon      0.67      0.92      0.77        13
Colin Powell      0.75      0.78      0.76        60
Donald Rumsfeld   0.78      0.67      0.72        27
George W Bush     0.86      0.86      0.86       146
Gerhard Schroeder 0.76      0.76      0.76        25
Hugo Chavez       0.67      0.67      0.67        15
Tony Blair        0.81      0.69      0.75        36

avg / total       0.80      0.80      0.80       322
=====

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

from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.datasets import fetch_lfw_people
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

```



```

from sklearn.decomposition import PCA
from sklearn.svm import SVC

print(__doc__)

# Display progress logs on stdout
logging.basicConfig(level=logging.INFO, format='%(asctime)s %(message)s')

#####
# Download the data, if not already on disk and load it as numpy arrays

lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)

# introspect the images arrays to find the shapes (for plotting)
n_samples, h, w = lfw_people.images.shape

# for machine learning we use the 2 data directly (as relative pixel
# positions info is ignored by this model)
X = lfw_people.data
n_features = X.shape[1]

# the label to predict is the id of the person
y = lfw_people.target
target_names = lfw_people.target_names
n_classes = target_names.shape[0]

print("Total dataset size:")
print("n_samples: %d" % n_samples)
print("n_features: %d" % n_features)
print("n_classes: %d" % n_classes)

#####
# Split into a training set and a test set using a stratified k fold

# split into a training and testing set
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42)

#####
# Compute a PCA (eigenfaces) on the face dataset (treated as unlabeled
# dataset): unsupervised feature extraction / dimensionality reduction
n_components = 150

print("Extracting the top %d eigenfaces from %d faces"
      % (n_components, X_train.shape[0]))
t0 = time()
pca = PCA(n_components=n_components, svd_solver='randomized',
          whiten=True).fit(X_train)
print("done in %0.3fs" % (time() - t0))

eigenfaces = pca.components_.reshape((n_components, h, w))

print("Projecting the input data on the eigenfaces orthonormal basis")
t0 = time()

```

```

X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
print("done in %0.3fs" % (time() - t0))

# #####
# Train a SVM classification model

print("Fitting the classifier to the training set")
t0 = time()
param_grid = {'C': [1e3, 5e3, 1e4, 5e4, 1e5],
              'gamma': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1], }
clf = GridSearchCV(SVC(kernel='rbf', class_weight='balanced'),
                  param_grid, cv=5, iid=False)
clf = clf.fit(X_train_pca, y_train)
print("done in %0.3fs" % (time() - t0))
print("Best estimator found by grid search:")
print(clf.best_estimator_)

# #####
# Quantitative evaluation of the model quality on the test set

print("Predicting people's names on the test set")
t0 = time()
y_pred = clf.predict(X_test_pca)
print("done in %0.3fs" % (time() - t0))

print(classification_report(y_test, y_pred, target_names=target_names))
print(confusion_matrix(y_test, y_pred, labels=range(n_classes)))

# #####
# Qualitative evaluation of the predictions using matplotlib

def plot_gallery(images, titles, h, w, n_row=3, n_col=4):
    """Helper function to plot a gallery of portraits"""
    plt.figure(figsize=(1.8 * n_col, 2.4 * n_row))
    plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray)
        plt.title(titles[i], size=12)
        plt.xticks(())
        plt.yticks(())

# plot the result of the prediction on a portion of the test set

def title(y_pred, y_test, target_names, i):
    pred_name = target_names[y_pred[i]].rsplit(' ', 1)[-1]
    true_name = target_names[y_test[i]].rsplit(' ', 1)[-1]
    return 'predicted: %s\ntrue:      %s' % (pred_name, true_name)

prediction_titles = [title(y_pred, y_test, target_names, i)
                    for i in range(y_pred.shape[0])]

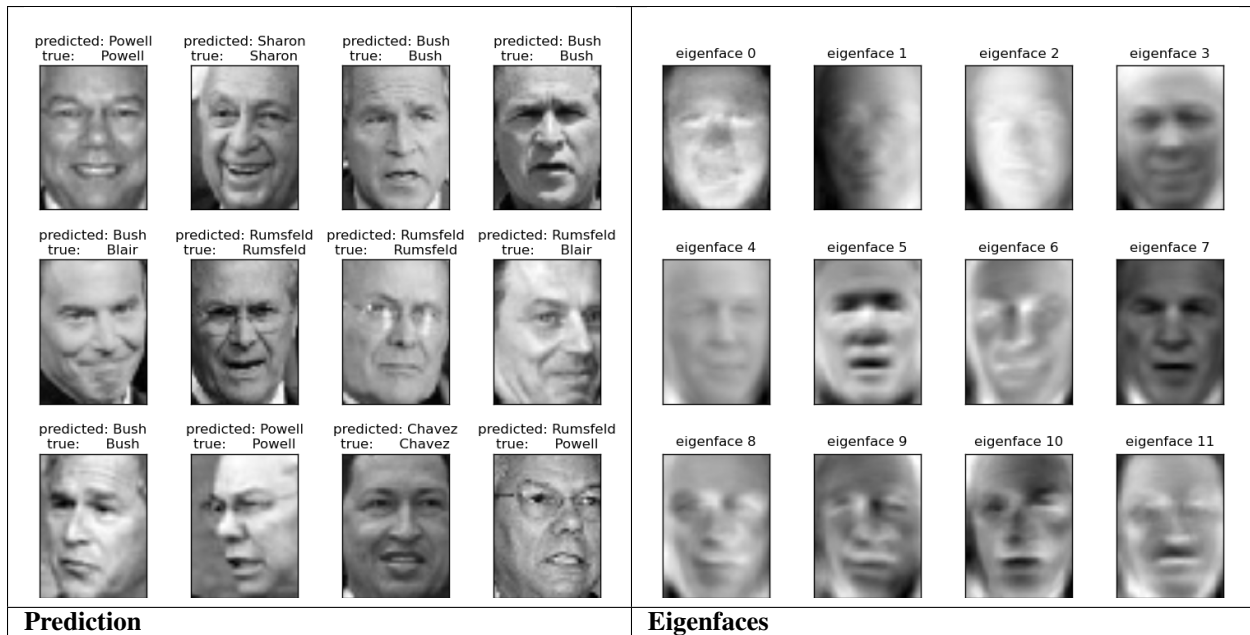
plot_gallery(X_test, prediction_titles, h, w)

```

```
# plot the gallery of the most significant eigenfaces

eigenface_titles = ["eigenface %d" % i for i in range(eigenfaces.shape[0])]
plot_gallery(eigenfaces, eigenface_titles, h, w)

plt.show()
```



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

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

Open problem: Stock Market Structure

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

Learning a graph structure

2.2.6 Finding help

The project mailing list

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

Q&A communities with Machine Learning practitioners

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

Stack Exchange The Stack Exchange family of sites hosts [multiple subdomains](#) for Machine Learning questions.

– ‘An excellent free online course for Machine Learning taught by Professor Andrew Ng of Stanford’: <https://www.coursera.org/learn/machine-learning>

– ‘Another excellent free online course that takes a more general approach to Artificial Intelligence’: <https://www.udacity.com/course/intro-to-artificial-intelligence-cs271>

2.3 Working With Text Data

The goal of this guide is to explore some of the main `scikit-learn` tools on a single practical task: analyzing a collection of text documents (newsgroups posts) on twenty different topics.

In this section we will see how to:

- load the file contents and the categories
- extract feature vectors suitable for machine learning
- train a linear model to perform categorization
- use a grid search strategy to find a good configuration of both the feature extraction components and the classifier

2.3.1 Tutorial setup

To get started with this tutorial, you must first install *scikit-learn* and all of its required dependencies.

Please refer to the [installation instructions](#) page for more information and for system-specific instructions.

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

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

The source can also be found [on Github](#).

The tutorial folder should contain the following sub-folders:

- `*.rst` files - the source of the tutorial document written with sphinx
- `data` - folder to put the datasets used during the tutorial
- `skeletons` - sample incomplete scripts for the exercises
- `solutions` - solutions of the exercises

You can already copy the `skeletons` into a new folder somewhere on your hard-drive named `sklearn_tut_workspace` where you will edit your own files for the exercises while keeping the original `skeletons` intact:

```
% cp -r skeletons work_directory/sklearn_tut_workspace
```

Machine learning algorithms need data. Go to each `$TUTORIAL_HOME/data` sub-folder and run the `fetch_data.py` script from there (after having read them first).

For instance:

```
% cd $TUTORIAL_HOME/data/languages
% less fetch_data.py
% python fetch_data.py
```

2.3.2 Loading the 20 newsgroups dataset

The dataset is called “Twenty Newsgroups”. Here is the official description, quoted from the [website](#):

The 20 Newsgroups data set is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different newsgroups. To the best of our knowledge, it was originally collected by Ken Lang, probably for his paper “Newsweeder: Learning to filter netnews,” though he does not explicitly mention this collection. The 20 newsgroups collection has become a popular data set for experiments in text applications of machine learning techniques, such as text classification and text clustering.

In the following we will use the built-in dataset loader for 20 newsgroups from scikit-learn. Alternatively, it is possible to download the dataset manually from the website and use the `sklearn.datasets.load_files` function by pointing it to the `20news-bydate-train` sub-folder of the uncompressed archive folder.

In order to get faster execution times for this first example we will work on a partial dataset with only 4 categories out of the 20 available in the dataset:

```
>>> categories = ['alt.atheism', 'soc.religion.christian',
...               'comp.graphics', 'sci.med']
```

We can now load the list of files matching those categories as follows:

```
>>> from sklearn.datasets import fetch_20newsgroups
>>> twenty_train = fetch_20newsgroups(subset='train',
...                                   categories=categories, shuffle=True, random_state=42)
```

The returned dataset is a scikit-learn “bunch”: a simple holder object with fields that can be both accessed as python dict keys or object attributes for convenience, for instance the `target_names` holds the list of the requested category names:

```
>>> twenty_train.target_names
['alt.atheism', 'comp.graphics', 'sci.med', 'soc.religion.christian']
```

The files themselves are loaded in memory in the `data` attribute. For reference the filenames are also available:

```
>>> len(twenty_train.data)
2257
>>> len(twenty_train.filenames)
2257
```

Let’s print the first lines of the first loaded file:

```
>>> print("\n".join(twenty_train.data[0].split("\n")[:3]))
From: sd345@city.ac.uk (Michael Collier)
Subject: Converting images to HP LaserJet III?
Nntp-Posting-Host: hampton

>>> print(twenty_train.target_names[twenty_train.target[0]])
comp.graphics
```

Supervised learning algorithms will require a category label for each document in the training set. In this case the category is the name of the newsgroup which also happens to be the name of the folder holding the individual documents.

For speed and space efficiency reasons `scikit-learn` loads the target attribute as an array of integers that corresponds to the index of the category name in the `target_names` list. The category integer id of each sample is stored in the `target` attribute:

```
>>> twenty_train.target[:10]
array([1, 1, 3, 3, 3, 3, 3, 2, 2, 2])
```

It is possible to get back the category names as follows:

```
>>> for t in twenty_train.target[:10]:
...     print(twenty_train.target_names[t])
...
comp.graphics
comp.graphics
soc.religion.christian
soc.religion.christian
soc.religion.christian
soc.religion.christian
soc.religion.christian
sci.med
sci.med
sci.med
```

You might have noticed that the samples were shuffled randomly when we called `fetch_20newsgroups(..., shuffle=True, random_state=42)`: this is useful if you wish to select only a subset of samples to quickly train a model and get a first idea of the results before re-training on the complete dataset later.

2.3.3 Extracting features from text files

In order to perform machine learning on text documents, we first need to turn the text content into numerical feature vectors.

Bags of words

The most intuitive way to do so is to use a bags of words representation:

1. Assign a fixed integer id to each word occurring in any document of the training set (for instance by building a dictionary from words to integer indices).
2. For each document #*i*, count the number of occurrences of each word *w* and store it in `X[i, j]` as the value of feature #*j* where *j* is the index of word *w* in the dictionary.

The bags of words representation implies that `n_features` is the number of distinct words in the corpus: this number is typically larger than 100,000.

If `n_samples == 10000`, storing `X` as a NumPy array of type float32 would require 10000 x 100000 x 4 bytes = **4GB in RAM** which is barely manageable on today's computers.

Fortunately, **most values in X will be zeros** since for a given document less than a few thousand distinct words will be used. For this reason we say that bags of words are typically **high-dimensional sparse datasets**. We can save a lot of memory by only storing the non-zero parts of the feature vectors in memory.

`scipy.sparse` matrices are data structures that do exactly this, and `scikit-learn` has built-in support for these structures.

Tokenizing text with `scikit-learn`

Text preprocessing, tokenizing and filtering of stopwords are all included in `CountVectorizer`, which builds a dictionary of features and transforms documents to feature vectors:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count_vect = CountVectorizer()
>>> X_train_counts = count_vect.fit_transform(twenty_train.data)
>>> X_train_counts.shape
(2257, 35788)
```

`CountVectorizer` supports counts of N-grams of words or consecutive characters. Once fitted, the vectorizer has built a dictionary of feature indices:

```
>>> count_vect.vocabulary_.get(u'algorithm')
4690
```

The index value of a word in the vocabulary is linked to its frequency in the whole training corpus.

From occurrences to frequencies

Occurrence count is a good start but there is an issue: longer documents will have higher average count values than shorter documents, even though they might talk about the same topics.

To avoid these potential discrepancies it suffices to divide the number of occurrences of each word in a document by the total number of words in the document: these new features are called `tf` for Term Frequencies.

Another refinement on top of `tf` is to downscale weights for words that occur in many documents in the corpus and are therefore less informative than those that occur only in a smaller portion of the corpus.

This downscaling is called `tf-idf` for “Term Frequency times Inverse Document Frequency”.

Both `tf` and `tf-idf` can be computed as follows using `TfidfTransformer`:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> tf_transformer = TfidfTransformer(use_idf=False).fit(X_train_counts)
>>> X_train_tf = tf_transformer.transform(X_train_counts)
>>> X_train_tf.shape
(2257, 35788)
```

In the above example-code, we firstly use the `fit(...)` method to fit our estimator to the data and secondly the `transform(...)` method to transform our count-matrix to a `tf-idf` representation. These two steps can be combined to achieve the same end result faster by skipping redundant processing. This is done through using the `fit_transform(...)` method as shown below, and as mentioned in the note in the previous section:

```
>>> tfidf_transformer = TfidfTransformer()
>>> X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
>>> X_train_tfidf.shape
(2257, 35788)
```

2.3.4 Training a classifier

Now that we have our features, we can train a classifier to try to predict the category of a post. Let's start with a *naïve Bayes* classifier, which provides a nice baseline for this task. `scikit-learn` includes several variants of this classifier; the one most suitable for word counts is the multinomial variant:

```
>>> from sklearn.naive_bayes import MultinomialNB
>>> clf = MultinomialNB().fit(X_train_tfidf, twenty_train.target)
```

To try to predict the outcome on a new document we need to extract the features using almost the same feature extracting chain as before. The difference is that we call `transform` instead of `fit_transform` on the transformers, since they have already been fit to the training set:

```
>>> docs_new = ['God is love', 'OpenGL on the GPU is fast']
>>> X_new_counts = count_vect.transform(docs_new)
>>> X_new_tfidf = tfidf_transformer.transform(X_new_counts)

>>> predicted = clf.predict(X_new_tfidf)

>>> for doc, category in zip(docs_new, predicted):
...     print('%r => %s' % (doc, twenty_train.target_names[category]))
...
'God is love' => soc.religion.christian
'OpenGL on the GPU is fast' => comp.graphics
```

2.3.5 Building a pipeline

In order to make the vectorizer => transformer => classifier easier to work with, `scikit-learn` provides a *Pipeline* class that behaves like a compound classifier:

```
>>> from sklearn.pipeline import Pipeline
>>> text_clf = Pipeline([
...     ('vect', CountVectorizer()),
...     ('tfidf', TfidfTransformer()),
...     ('clf', MultinomialNB()),
... ])
```

The names `vect`, `tfidf` and `clf` (classifier) are arbitrary. We will use them to perform grid search for suitable hyperparameters below. We can now train the model with a single command:

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

2.3.6 Evaluation of the performance on the test set

Evaluating the predictive accuracy of the model is equally easy:

```
>>> import numpy as np
>>> twenty_test = fetch_20newsgroups(subset='test',
...     categories=categories, shuffle=True, random_state=42)
>>> docs_test = twenty_test.data
>>> predicted = text_clf.predict(docs_test)
>>> np.mean(predicted == twenty_test.target)
0.8348...
```


We achieved 83.5% accuracy. Let's see if we can do better with a linear *support vector machine (SVM)*, which is widely regarded as one of the best text classification algorithms (although it's also a bit slower than naïve Bayes). We can change the learner by simply plugging a different classifier object into our pipeline:

```
>>> from sklearn.linear_model import SGDClassifier
>>> text_clf = Pipeline([
...     ('vect', CountVectorizer()),
...     ('tfidf', TfidfTransformer()),
...     ('clf', SGDClassifier(loss='hinge', penalty='l2',
...                           alpha=1e-3, random_state=42,
...                           max_iter=5, tol=None)),
... ])

>>> text_clf.fit(twenty_train.data, twenty_train.target)
Pipeline(...)
>>> predicted = text_clf.predict(docs_test)
>>> np.mean(predicted == twenty_test.target)
0.9101...
```

We achieved 91.3% accuracy using the SVM. `scikit-learn` provides further utilities for more detailed performance analysis of the results:

```
>>> from sklearn import metrics
>>> print(metrics.classification_report(twenty_test.target, predicted,
...     target_names=twenty_test.target_names))
...
              precision    recall  f1-score   support

alt.atheism          0.95      0.80      0.87         319
comp.graphics        0.87      0.98      0.92         389
sci.med              0.94      0.89      0.91         396
soc.religion.christian 0.90      0.95      0.93         398

accuracy              0.91
macro avg             0.91      0.91      0.91         1502
weighted avg          0.91      0.91      0.91         1502

>>> metrics.confusion_matrix(twenty_test.target, predicted)
array([[256,  11,  16,  36],
       [  4, 380,   3,   2],
       [  5,  35, 353,   3],
       [  5,  11,   4, 378]])
```

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

2.3.7 Parameter tuning using grid search

We've already encountered some parameters such as `use_idf` in the `TfidfTransformer`. Classifiers tend to have many parameters as well; e.g., `MultinomialNB` includes a smoothing parameter `alpha` and `SGDClassifier` has a penalty parameter `alpha` and configurable loss and penalty terms in the objective function (see the module documentation, or use the Python `help` function to get a description of these).

Instead of tweaking the parameters of the various components of the chain, it is possible to run an exhaustive search of the best parameters on a grid of possible values. We try out all classifiers on either words or bigrams, with or without `idf`, and with a penalty parameter of either 0.01 or 0.001 for the linear SVM:

```
>>> from sklearn.model_selection import GridSearchCV
>>> parameters = {
...     'vect__ngram_range': [(1, 1), (1, 2)],
...     'tfidf__use_idf': (True, False),
...     'clf__alpha': (1e-2, 1e-3),
... }
```

Obviously, such an exhaustive search can be expensive. If we have multiple CPU cores at our disposal, we can tell the grid searcher to try these eight parameter combinations in parallel with the `n_jobs` parameter. If we give this parameter a value of `-1`, grid search will detect how many cores are installed and use them all:

```
>>> gs_clf = GridSearchCV(text_clf, parameters, cv=5, iid=False, n_jobs=-1)
```

The grid search instance behaves like a normal `scikit-learn` model. Let's perform the search on a smaller subset of the training data to speed up the computation:

```
>>> gs_clf = gs_clf.fit(twenty_train.data[:400], twenty_train.target[:400])
```

The result of calling `fit` on a `GridSearchCV` object is a classifier that we can use to predict:

```
>>> twenty_train.target_names[gs_clf.predict(['God is love'])[0]]
'soc.religion.christian'
```

The object's `best_score_` and `best_params_` attributes store the best mean score and the parameters setting corresponding to that score:

```
>>> gs_clf.best_score_
0.9...
>>> for param_name in sorted(parameters.keys()):
...     print("%s: %r" % (param_name, gs_clf.best_params_[param_name]))
...
clf__alpha: 0.001
tfidf__use_idf: True
vect__ngram_range: (1, 1)
```

A more detailed summary of the search is available at `gs_clf.cv_results_`.

The `cv_results_` parameter can be easily imported into `pandas` as a `DataFrame` for further inspection.

Exercises

To do the exercises, copy the content of the 'skeletons' folder as a new folder named 'workspace':

```
% cp -r skeletons workspace
```

You can then edit the content of the workspace without fear of losing the original exercise instructions.

Then fire an `ipython` shell and run the work-in-progress script with:

```
[1] %run workspace/exercise_XX_script.py arg1 arg2 arg3
```

If an exception is triggered, use `%debug` to fire-up a post mortem `ipdb` session.

Refine the implementation and iterate until the exercise is solved.

For each exercise, the skeleton file provides all the necessary import statements, boilerplate code to load the data and sample code to evaluate the predictive accuracy of the model.

2.3.8 Exercise 1: Language identification

- Write a text classification pipeline using a custom preprocessor and `CharNGramAnalyzer` using data from Wikipedia articles as training set.
- Evaluate the performance on some held out test set.

ipython command line:

```
%run workspace/exercise_01_language_train_model.py data/languages/paragraphs/
```

2.3.9 Exercise 2: Sentiment Analysis on movie reviews

- Write a text classification pipeline to classify movie reviews as either positive or negative.
- Find a good set of parameters using grid search.
- Evaluate the performance on a held out test set.

ipython command line:

```
%run workspace/exercise_02_sentiment.py data/movie_reviews/txt_sentoken/
```

2.3.10 Exercise 3: CLI text classification utility

Using the results of the previous exercises and the `cPickle` module of the standard library, write a command line utility that detects the language of some text provided on `stdin` and estimate the polarity (positive or negative) if the text is written in English.

Bonus point if the utility is able to give a confidence level for its predictions.

2.3.11 Where to from here

Here are a few suggestions to help further your scikit-learn intuition upon the completion of this tutorial:

- Try playing around with the analyzer and token normalisation under [CountVectorizer](#).
- If you don't have labels, try using [Clustering](#) on your problem.
- If you have multiple labels per document, e.g categories, have a look at the [Multiclass and multilabel section](#).
- Try using [Truncated SVD](#) for latent semantic analysis.
- Have a look at using [Out-of-core Classification](#) to learn from data that would not fit into the computer main memory.
- Have a look at the [Hashing Vectorizer](#) as a memory efficient alternative to [CountVectorizer](#).

2.4 Choosing the right estimator

Often the hardest part of solving a machine learning problem can be finding the right estimator for the job.

Different estimators are better suited for different types of data and different problems.

The flowchart below is designed to give users a bit of a rough guide on how to approach problems with regard to which estimators to try on your data.

Click on any estimator in the chart below to see its documentation.

2.5 External Resources, Videos and Talks

For written tutorials, see the *[Tutorial section](#)* of the documentation.

2.5.1 New to Scientific Python?

For those that are still new to the scientific Python ecosystem, we highly recommend the [Python Scientific Lecture Notes](#). This will help you find your footing a bit and will definitely improve your scikit-learn experience. A basic understanding of NumPy arrays is recommended to make the most of scikit-learn.

2.5.2 External Tutorials

There are several online tutorials available which are geared toward specific subject areas:

- [Machine Learning for NeuroImaging in Python](#)
- [Machine Learning for Astronomical Data Analysis](#)

2.5.3 Videos

- An introduction to scikit-learn [Part I](#) and [Part II](#) at Scipy 2013 by [Gael Varoquaux](#), [Jake Vanderplas](#) and [Olivier Grisel](#). Notebooks on [github](#).
- [Introduction to scikit-learn](#) by [Gael Varoquaux](#) at ICML 2010

A three minute video from a very early stage of scikit-learn, explaining the basic idea and approach we are following.
- [Introduction to statistical learning with scikit-learn](#) by [Gael Varoquaux](#) at SciPy 2011

An extensive tutorial, consisting of four sessions of one hour. The tutorial covers the basics of machine learning, many algorithms and how to apply them using scikit-learn. The material corresponding is now in the scikit-learn documentation section *[A tutorial on statistical-learning for scientific data processing](#)*.
- [Statistical Learning for Text Classification with scikit-learn and NLTK](#) (and slides) by [Olivier Grisel](#) at PyCon 2011

Thirty minute introduction to text classification. Explains how to use NLTK and scikit-learn to solve real-world text classification tasks and compares against cloud-based solutions.
- [Introduction to Interactive Predictive Analytics in Python with scikit-learn](#) by [Olivier Grisel](#) at PyCon 2012

3-hours long introduction to prediction tasks using scikit-learn.
- [scikit-learn - Machine Learning in Python](#) by [Jake Vanderplas](#) at the 2012 PyData workshop at Google

Interactive demonstration of some scikit-learn features. 75 minutes.
- [scikit-learn tutorial](#) by [Jake Vanderplas](#) at PyData NYC 2012

Presentation using the online tutorial, 45 minutes.

Note: Doctest Mode

The code-examples in the above tutorials are written in a *python-console* format. If you wish to easily execute these examples in **IPython**, use:

```
%doctest_mode
```

in the IPython-console. You can then simply copy and paste the examples directly into IPython without having to worry about removing the `>>>` manually.
