CHAPTER 4

Summary of scikit-learn methods and usage

In this chapter, we briefly recapitulate the main parts of the scikit-learn API that we have seen so far, as well as show some ways to simplify your code.

The Estimator Interface

All algorithms in scikit-learn, whether preprocessing, supervised learning or unsupervised learning algorithms are all implemented as classes. These classes are called *estimators* in scikit-learn. To apply an algorithm, you first have to instantiate an object of the particular class:

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()
```

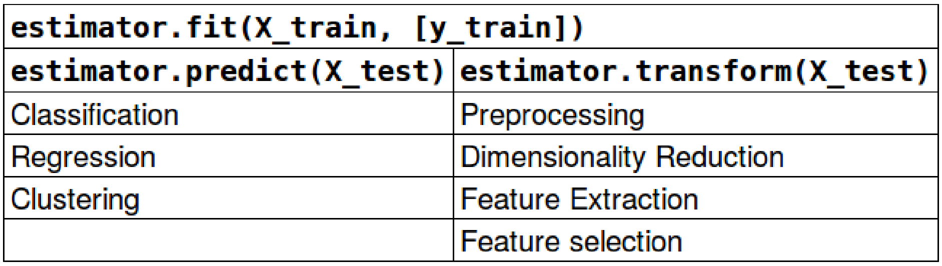
The estimator class contains the algorithm, and also stored the model that is learned from data using the algorithm.

When constructing the model object, this is also the time when you should set any parameters of the model. These parameters include regularization, complexity control, number of clusters to find, etc, as we discussed in detail in Chapter 2 and Chapter 3.

All estimators have a `fit` method, which is used to build the model. The `fit` method always requires as first argument the data `X`, represented as a numpy array or a scipy sparse matrix, where each row represents a single data point. The data `X` is always assumed to be a numpy array or scipy sparse matrix that has continuous (floating point) entries.

Supervised algorithms also require a `y` argument, which is a one-dimensional numpy array, containing target values for regression or classification, i.e. the known output labels or responses.

There are two main ways to apply a learned model in scikit-learn. To create a prediction in the form of a new output like `y`, you use the `predict` method. To create a new representation of the input data `X`, you use the `transform` method. Table api_summary summarizes the use-cases of the predict and transform methods.



estimator.fit(X_train, [y_train])	
estimator.predict(X_test)	**estimator.transform(X_test)**
Classification	Preprocessing
Regression	Dimensionality Reduction
Clustering	Feature Extraction
	Feature selection

Table api_summary

Additionally, all supervised models have a `score(X_test, y_test)` method, that allows an evaluation of the model.

Here `X_train` and `y_train` refere to the training data and training labels, while `X_test` and `y_test` refer to the test data and test labels (if applicable).

Fit resets a model

An important property of scikit-learn models is that calling `fit` will always reset everything a model previously learned. So if you build a model on one dataset, and then call `fit` again on a different dataset, the model will "forget" everything it learned from the first data. You can call `fit` as often as you like on a model, and the outcome will be the same as calling `fit` on a "new" model:

```
# get some data
from sklearn.datasets import make_blobs, load_iris
from sklearn.model_selection import train_test_split

# load iris
iris = load_iris()

# create some blobs
X, y = make_blobs(random_state=0, centers=4)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# build a model on the iris dataset
logreg = LogisticRegression()
```

```
logreg.fit(iris.data, iris.target)
# fit the model again on the blob dataset
logreg.fit(X_train, y_train)
# the outcome is the same as training a "fresh" model:
new_logreg = LogisticRegression()
new_logreg.fit(X_train, y_train)

# predictions made by the two models are the same
pred_new_logreg = new_logreg.predict(X_test)
pred_logreg = logreg.predict(X_test)

pred_logreg == pred_new_logreg
```

```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,

        True,  True,  True,  True,  True,  True,  True,  True,  True,

        True,  True,  True,  True,  True,  True,  True], dtype=bool)
```

As you can see, fitting the `logreg` model first on the `iris` dataset has no effect. The `iris` dataset has a different number of features and classes than the `blobs` dataset, but all about the first fit is erased when `fit` is called again.

Next, we will go into several shortcuts that allow you to write less code for common tasks, and speed up some computations. The first way to write more compact code is to make use *method chaining*.

Method chaining

The `fit` method of all scikit-learn models returns `self`. This allows you to write code like this:

```
# instantiate model and fit it in one line
logreg = LogisticRegression().fit(X_train, y_train)
```

Here, we used the return value of `fit` (which is `self`) to assign the trained model to the variable `logreg`. This concatenation of method calls (here `__init__` and then `fit`) is known as *method chaining*. Another common application of method chaining in scikit-learn is to `fit` and `predict` in one line:

```
logreg = LogisticRegression()
y_pred = logreg.fit(X_train, y_train).predict(X_test)
```

Finally, you can even do model instantiation, fitting and predicting in one line:

```
y_pred = LogisticRegression().fit(X_train, y_train).predict(X_test)
```

This very short variant is not ideal, though. A lot is happening in a single line, which might make the code hard to read. Additionally, the fitted logistic regression model isn't stored in any variable. So we can't inspect it, or use it to predict on any other data.

Shortcuts and efficient alternatives

Often, you want to fit a model on some dataset, and then immediately predict on the same data, or transform it. These are very common tasks, which can often be computed more efficiently than simply calling fit and then predict or fit and then transform. For this use-case, all models that have a predict method also have a fit_predict method, and all model that have a transform method also have a fit_transform method. Here is an example using PCA:

```
from sklearn.decomposition import PCA
pca = PCA()
# calling fit and transform in sequence (using method chaining)
X_pca = pca.fit(X).transform(X)
# same result, but more efficient computation
X_pca_2 = pca.fit_transform(X)
```

While fit_transform and fit_predict are not more efficient for all algorithms, it is still good practice to use them when trying to predict on, or transform the training set.

For some unsupervised methods that we saw in Chapter 3, like some clustering and manifold learning methods, using fit_transform and fit_predict are the only options. For example DBSCAN does not have a predict method, only fit_predict, and t-SNE does not have a transform method, only fit_transform. T-SNE and DBSCAN are algorithms that can not be applied to new data, they can only be applied to the training data.

Important Attributes

scikit-learn has some standard attributes that allow you to inspect what a model learned. All these attributes are available after the call to fit, and, as we mentioned before, all attributes learned from the data are marked with a trailing underscore.

We already discussed the following common attributes:

- For clustering algorithms, the labels_ attribute stores the cluster membership for the training data.
- For manifold learning algorithms, the embedding_ attribute stores the embedding (transformation) of the training data in the lower-dimensional space.
- For linear models, the coef_ attribute stores the weight or coefficient vector.

- For linear decomposition and dimensionality reduction methods, `components_` stores the array of components (the prototypes in the additive decomposition in Figure decomposition in Chapter 3).

Additionally, for classifiers, `classes_` contains the names of the classes the classifier was trained on, that is the unique entries of the training labels `y_train`:

```
import numpy as np
logreg = LogisticRegression()
# fit model using original data
logreg.fit(iris.data, iris.target)
print("unique entries of iris.target: %s" % np.unique(iris.target))
print("classes using iris.target: %s" % logreg.classes_)

# represent each target by its class name
named_target = iris.target_names[iris.target]
logreg.fit(iris.data, named_target)
print("unique entries of named_target: %s" % np.unique(named_target))
print("classes using named_target: %s" % logreg.classes_)
```

```
unique entries of iris.target: [0 1 2]

classes using iris.target: [0 1 2]

unique entries of named_target: ['setosa' 'versicolor' 'virginica']

classes using named_target: ['setosa' 'versicolor' 'virginica']
```

Summary and outlook

You should now be be intimately familiar with the interfaces of supervised and unsupervised models in scikit-learn, and how to use them. With a good grasp on how to use the different models, we will continue with more complex topics, such as evaluating and selecting models.