CHAPTER 15

Machine Learning

In this chapter we explore machine learning. This topic is closely related to statistical modeling, which we considered in Chapter 14, in the sense that both deal with using data to describe and predict outcomes of uncertain or unknown processes. However, while statistical modeling emphasizes the model used in the analysis, machine learning side steps the model part and focuses on algorithms that can be trained to predict the outcome of new observations. In other words, the approach taken in statistical modeling emphasizes understanding how the data is generated, by devising models and tuning their parameters by fitting to the data. If the model is found to fit the data well and if it satisfies the relevant model assumptions, then the model gives an overall description of the process, and it can be used to compute statistics with known distributions and for evaluating statistical tests. However, if the actual data is too complex to be explained using available statistical models, this approach has reached its limits. In machine learning, on the other hand, the actual process that generates the data, and potential models thereof, is not central. Instead, the observed data and the explanatory variables are the fundamental starting points of a machine-learning application. Given data, machine-learning methods can be used to find patterns and structure in the data, which can be used to predict the outcome for new observations. Machine learning therefore does not provide understanding of how data was generated, and because fewer assumptions are made regarding the distribution and statistical properties of the data, we typically cannot compute statistics and perform statistical tests regarding the significance of certain observations. Instead, machine learning puts strong emphasis on the accuracy with which new observations are predicted.

Although there are significant differences in the fundamental approach taken in statistical modeling and machine learning, many of the mathematical methods that are used are closely related or sometimes even the same. In the course of this chapter, we are going to recognize several methods that we used in Chapter 14 on statistical modeling, but they will be employed with a different mindset and with slightly different goals.

In this chapter we give a brief introduction to basic machine-learning methods and we survey how such methods can be used in Python. The focus is on machine-learning methods that have broad application in many fields of scientific and technical computing. The most prominent and comprehensive machine learning library for Python is scikit-learn, although there are several alternative and complementary libraries as well: For example mlpy,[1] PyBrain,[2] and pylearn2,[3] to mention a few. Here we exclusively use the scikit-learn library, which provides implementations of the most common machine learning algorithm. However, readers that are particularly interested in machine learning are encouraged to also explore the other libraries mentioned above as well.

[1] http://mlpy.sourceforge.net.
[2] http://pybrain.org.
[3] http://deeplearning.net/software/pylearn2.

R. Johansson, *Numerical Python*, DOI 10.1007/978-1-4842-0553-2_15

■ **scikit-learn** The scikit-learn library contains a comprehensive collection of machine-learning related algorithms, including regression, classification, dimensionality reduction, and clustering. For more information about the project and its documentation, see the projects web page at http://scikit-learn.org. At the time of writing the latest version of scikit-learn is 0.16.1.

Importing Modules

In this chapter we work with the scikit-learn library, which provides the `sklearn` Python module. With the `sklearn` module, here we use the same import strategy as we use with the SciPy library: that is, we explicitly import modules from the library that we need for our work. In this chapter we use the following modules from the `sklearn` library:

```
In [1]: from sklearn import datasets
In [2]: from sklearn import cross_validation
In [3]: from sklearn import linear_model
In [4]: from sklearn import metrics
In [5]: from sklearn import tree
In [6]: from sklearn import neighbors
In [7]: from sklearn import svm
In [8]: from sklearn import ensemble
In [9]: from sklearn import cluster
```

For plotting and basic numerics we also require the Matplotlib and NumPy libraries, which we import in the usual manner:

```
In [10]: import matplotlib.pyplot as plt
In [11]: import numpy as np
```

We also use the Seaborn library for graphics and figure styling:

```
In [12]: import seaborn as sns
```

Brief Review of Machine Learning

Machine learning is a topic in the artificial-intelligence field of computer science. Machine learning can be viewed as including all applications where feeding training data into a computer program makes it able perform a given task. This is a very broad definition, but in practice machine learning is often associated with a much more specific set of techniques and methods. Here we take a practical approach and explore, by example, several basic methods and key concepts in machine learning. Before we get started with specific examples, we begin with a brief introduction of the terminology and core concepts.

In machine learning, the process of fitting a model or an algorithm to observed data is known as *training*. Machine-learning applications can often be classified into either of two types: *supervised* and *unsupervised* learning, which differ in the type of data the application is trained with. In *supervised learning*, the data includes feature variables and known response variables. Both feature and response variables can be continuous or discrete. Preparing such data typically requires manual effort, and sometimes even expert domain knowledge. The application is thus trained with handcrafted data, and the training can therefore

be viewed as supervised machine learning. Examples of applications include regression (prediction of a continuous response variable) and classification (prediction of a discrete response variable), where the value of the response variable is known for the training dataset, but not for new samples.

In contrast, *unsupervised learning* corresponds to situations where machine-learning applications are trained with raw data that is not labeled or otherwise manually prepared. An example of unsupervised learning is clustering of data into groups, or in other words, grouping of data into suitable categories. In contrast to supervised classification, it is typical for unsupervised learning that the final categories are not known in advance, and the training data therefore cannot be labeled accordingly. It may also be the case that the manual labeling of the data is difficult or costly, for example, because the number of samples is too large. It goes without saying that unsupervised machine learning is more difficult and limited in what it can be used for than supervised machine learning, and supervised machine learning therefore should be preferred whenever possible. However, unsupervised machine learning can be a powerful tool when creating labeled training datasets is not possible.

There is naturally much more complexity to machine learning than suggested by the basic types of problems outlined above, but these concepts are recurring themes in many machine-learning applications. In this chapter we look at a few examples of basic machine-learning techniques that demonstrates several central concepts of machine learning. Before we do so we briefly introduce common machine-learning terminology that we will refer to in the following sections:

- *Cross-validation* is the practice of dividing the available data into *training data* and *testing data* (also known as *validation data*), where only the training data is used to train the machine learning application, and where the test data allows the trained application to be tested on previously unseen data. The purpose of this is to measure how well the model predicts new observations, and to limit problems with overfitting. There are several approaches to dividing the data into training and testing datasets. For example, one extreme approach is to test all possible ways to divide the data (*exhaustive cross-validation*) and use an aggregate of the result (for example, average, or the minimum value, depending on the situation). However, for large datasets the number of possible combinations of train and test data becomes extremely large, making exhaustive cross-validation impractical. Another extreme is to use all but one sample in the training set, and the remaining sample in the training set (*leave-one-out cross-validation*), and to repeat the training-test cycle for all combinations in which one sample is chosen from the available data. A variant of this method is to divide the available data into k groups and perform a leave-one-out cross-validation with the k groups of datasets. This method is known as k-fold cross-validation, and is a popular technique that often is used in practice. In the scikit-learn library, the module `sklearn.cross_validation` contains functions for working with cross-validation.

- *Feature extraction* is an important step in the preprocessing stage of a machine-learning problem. It involves creating suitable feature variables and the corresponding feature matrices that can be passed to one of many machine-learning algorithms implemented in the scikit-learn library. The scikit-learn module `sklearn.feature_extraction` plays a similar role in many machine-learning applications as the Patsy formula library does in statistical models, especially for text- and image-based machine learning problems. Using methods from the `sklearn.feature_extraction` module, we can automatically assemble feature matrices (design matrices) from various data sources.

- *Dimensionality reduction* and *feature selection* are techniques that are frequently used in machine-learning applications where it is common to have a large number of explanatory variables (features), many of which may not significantly contribute to the predictive power of the application. To reduce the complexity of the model it is then often desirable to eliminate less useful features and thereby reduce the dimensionality of the problem. This is particularly important when the number of features is comparable to or larger than the number of observations. The scikit-learn modules `sklearn.decomposition` and `sklearn.feature_selection` contains function for reducing the dimensionality of a machine-learning problem: For example, principle component analysis (PCA) is a popular technique for dimensionality reduction that works by performing a singular-value decomposition of the feature matrix and keeping only most significant singular vectors.

In the following sections we look how scikit-learn can be used to solve examples of machine-learning problems using the techniques discussed above. Here we work with generated data and built-in datasets. Like the statsmodels library, scikit-learn comes with a number of built-in datasets that can be used for exploring machine-learning methods. The `datasets` module in `sklearn` provides three groups of functions for loading built-in datasets (with prefix `load_`, for example `load_boston`), for fetching external datasets (with prefix `fetch_`, for example `fetch_californa_housing`), and finally for generating datasets from random numbers (with prefix `make_`, for example `make_regression`).

Regression

Regression is a central part of machine learning and statistical modeling, as we already saw in Chapter 14. In machine learning we are not so concerned with how well the regression model fits to the data, but rather care about how well it predicts new observations. For example, if we have a large number of features and less number of observations, we can typically fit the regression perfectly to the data without it being very useful for predicting new values. This is an example of overfitting: a small residual between the data and regression model is not a guarantee that the model is able to accurately predict future observations. In machine learning, a common method to deal with this problem is to partition the available data into a training dataset and a testing dataset that is used for validating the regression results against previously unseen data.

To see how fitting a training data set and validating the result against a testing data set can work out, let's consider a regression problem with 50 samples and 50 features out of which only 10 features are informative (linearly correlated with the response variable). This simulates a scenario when we have a 50 known features, but it turns out that only 10 of those features contribute to the predictive power of the regression model. The `make_regression` function in the `sklearn.datasets` module generates data of kind:

```
In [13]: X_all, y_all = datasets.make_regression(n_samples=50, n_features=50, n_informative=10)
```

The result is two arrays, `X_all` and `y_all`, of shapes `(50, 50)` and `(50,)`, corresponding to the design matrices for a regression problem with 50 samples and 50 features. Instead of performing a regression on the entire dataset (and obtaining a perfect fit because of the small number of observations), here we split the dataset into two equal size datasets, using the `train_test_split` function from `sklearn.cross_validation` module. The result is a training dataset `X_train`, `y_train`, and a testing dataset `X_test`, `y_test`:

```
In [14]: X_train, X_test, y_train, y_test = \
    ...:     cross_validation.train_test_split(X_all, y_all, train_size=0.5)
```

In scikit-learn, ordinary linear regression can be carried out using the `LinearRegression` class from the `sklearn.linear_model` module, which is comparable with the `statsmodels.api.OLS` from the statsmodels library. To perform a regression we first create a `LinearRegression` instance:

```
In [15]: model = linear_model.LinearRegression()
```

To actually fit the model to the data, we need to invoke the `fit` method, which takes the feature matrix and the response variable vector as first and second argument:

```
In [16]: model.fit(X_train, y_train)
Out[16]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

Note that compared to the `OLS` class in statsmodels, the order of the feature matrix and response variable vector is reversed, and in statsmodels the data is specified when the class instance is created instead of when calling the `fit` method. Also, in scikit-learn calling the `fit` method does not return new result objects, but the result is instead stored directly in the model instance. These minor differences are small inconveniences when working interchangeably with the statsmodels and scikit-learn modules but worth taking note of.[4]

Since the regression problem has 50 features and we only trained the model with 25 samples, we can expect complete overfitting that perfectly fits the data. This can be quantified by computing the sum of squared errors (SSE) between the model and the data. To evaluate the model for a given set of features we can use the `predict` method, from which we can compute the residuals and the SSE:

```
In [17]: def sse(resid):
    ...:     return np.sum(resid**2)
In [18]: sse_train = sse(y_train - model.predict(X_train))
    ...: sse_train
Out[18]: 8.1172209425431673e-25
```

As expected, for the training dataset the residuals are all essentially zero, due to the overfitting allowed by having twice as many features as data points. This overfitted model is, however, not at all suitable for predicting unseen data. This can be verified by computing the SSE for our test dataset:

```
In [19]: sse_test = sse(y_test - model.predict(X_test))
    ...: sse_test
Out[19]: 213555.61203039082
```

The result is a very large SSE value, which indicates that the model does not do a good job at predicting new observations. An alternative measure of the fit of a model to a dataset is the *r*-squared score (see Chapter 14), which we can compute using the `score` method. It takes a feature matrix and response variable vector as arguments and computes the score. For the training dataset we obtain, as expected, an r-square score of 1.0, but for the testing dataset we obtain a low score:

```
In [20]: model.score(X_train, y_train)
Out[20]: 1.0
In [21]: model.score(X_test, y_test)
Out[21]: 0.31407400675201746
```

[4]In practice it is common to work with both statsmodels and scikit-learn, as they, in many respects, complement each other. However, in this chapter we focus solely on scikit-learn.

The big difference between the scores for the training and testing datasets once again indicates that the model is overfitted.

Finally, we can also take a graphical approach and plot the residuals of the training and testing datasets, and visually inspect the values of the coefficients computed by the residual. From a `LinearRegression` object, we can extract the fitted parameters using the `coef_` attribute. To simplify repeated plotting of the training and testing residuals and the model parameters, here we first create a function `plot_residuals_and_coeff` for plotting these quantities. We then call the function with the result from the ordinary linear regression model trained and tested on the training and testing datasets, respectively. The result is shown in Figure 15-1, and it is clear that there is a large difference in the magnitude of the residuals for the test and the training datasets, for every sample.

```
In [22]: def plot_residuals_and_coeff(resid_train, resid_test, coeff):
    ...:     fig, axes = plt.subplots(1, 3, figsize=(12, 3))
    ...:     axes[0].bar(np.arange(len(resid_train)), resid_train)
    ...:     axes[0].set_xlabel("sample number")
    ...:     axes[0].set_ylabel("residual")
    ...:     axes[0].set_title("training data")
    ...:     axes[1].bar(np.arange(len(resid_test)), resid_test)
    ...:     axes[1].set_xlabel("sample number")
    ...:     axes[1].set_ylabel("residual")
    ...:     axes[1].set_title("testing data")
    ...:     axes[2].bar(np.arange(len(coeff)), coeff)
    ...:     axes[2].set_xlabel("coefficient number")
    ...:     axes[2].set_ylabel("coefficient")
    ...:     fig.tight_layout()
    ...:     return fig, axes
In [23]: fig, ax = plot_residuals_and_coeff(resid_train, resid_test, model.coef_)
```

Figure 15-1. *The residual between the ordinary linear regression model and the training data (left), the model and the test data (middle), and the values of the coefficients for the 50 features (right)*

The overfitting in this example happens because we have too few samples, and one solution could be to collect more samples until overfitting is no longer a problem. However, this may not always be practical, as collecting observations may be expensive, and because in some applications we might have a very large number of features. For such situations it is desirable to be able to fit a regression problem in a way that avoids overfitting as much as possible (at the expanse of not fitting the training data perfectly), so that the model can give meaningful predictions for new observations.

Regularized regression is one possible solution to this problem. In the following we look at a few different variations of regularized regression. In ordinary linear regression the model parameters are chosen such that the sum of squared residuals are minimized. Viewed as an optimization problem, the objective function is therefore $\min_\beta \|X\beta - y\|_2^2$, where X is the feature matrix, y is the response variables, and β is

the vector of model parameters, and where $\|\cdot\|_2$ denotes the L2 norm. In *regularized* regression, we add a *penalty term* in the objective function of the minimization problem. Different types of penalty terms impose different types of regularization of the original regression problem. Two popular types of regularization known as LASSO and Ridge regression are obtained by adding the L1 or L2 norms of the parameter vector to the minimization objective function, $\min_\beta \left\{ \|X\beta - y\|_2^2 + \alpha \|\beta\|_1 \right\}$ and $\min_\beta \left\{ \|X\beta - y\|_2^2 + \alpha \|\beta\|_2^2 \right\}$, respectively. Here α is a free parameter that determines the strength of the regularization. Adding the L2 norm $\|\beta\|_2^2$ favors model parameter vectors with smaller coefficients, and adding the L1 norm $\|\beta\|_1$ favors a model parameter vectors with as few nonzero elements as possible. Which type of regularization is more suitable depends on the problem at hand: When we wish to elminate as many features as possible we can use L1 regularization with LASSO regression, and when we wish to limit the magnitude of the model coefficients we can use L2 regularization with Ridge regression.

With scikit-learn, we can perform Ridge regression using the `Ridge` class from the `sklearn.linear_model` module. The usage of this class is almost the same as the `LinearRegression` class that we used above, but we can also give the value of the α parameter that determines the strength of the regularization as argument when we initialize the class. Here we chose the value $\alpha = 2.5$. A more systematic approach to choosing α is introduced later in this chapter.

```
In [24]: model = linear_model.Ridge(alpha=2.5)
```

To fit the regression model to the data we again use the fit method, passing the training feature matrix and response variable as arguments:

```
In [25]: model.fit(X_train, y_train)
Out[25]: Ridge(alpha=2.5, copy_X=True, fit_intercept=True, max_iter=None,
               normalize=False, solver='auto', tol=0.001)
```

Once the model has been fitted to the training data, we can compute the model predictions for the training and testing datasets, and compute the corresponding SSE values:

```
In [26]: sse_train = sse(y_train - model.predict(X_train))
    ...: sse_train
Out[26]: 178.50695164950841
In [27]: sse_test = sse(y_test - model.predict(X_test))
    ...: sse_test
Out[27]: 212737.00160105844
```

We note that the SSE of the training data is no longer close to zero, since the minimization object function no longer coincides with the SSE, but there is a slight decrease in the SSE for the testing data. For comparison with ordinary regression, we also plot the training and testing residuals and the model parameters using the function `plot_residuals_and_coeff` that we defined above. The result is shown in Figure 15-2.

```
In [28]: fig, ax = plot_residuals_and_coeff(resid_train, resid_test, model.coef_)
```

Figure 15-2. *The residual between the Ridge regulalized regression model and the training data (left), the model and the test data (middle), and the values of the coefficients for the 50 features (right)*

Similarly, we can perform the L1 regularized LASSO regression using the `Lasso` class from the `sklearn.linear_model` module. It also accepts the value of the α parameter as argument when the class instance is initialized. Here we choose $\alpha = 1.0$ and perform the fitting to the training data and the computation of the SSE for the training and testing data in the same way as described previously:

```
In [29]: model = linear_model.Lasso(alpha=1.0)
In [30]: model.fit(X_train, y_train)
Out[30]: Lasso(alpha=1.0, 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)
In [31]: sse_train = sse(y_train - model.predict(X_train))
    ...: sse_train
Out[31]: 309.74971389531891
In [32]: sse_test = sse(y_test - model.predict(X_test))
    ...: sse_test
Out[32]: 1489.1176065002333
```

Here we note that the while the SSE of the training data increased compared to that of the ordinary regression, the SSE for the testing data decreased significantly. Thus, by paying a price for how well the regression model fit the training data, we have obtained a model with significantly improved ability to predict the testing dataset. For comparison with the earlier methods we graph the residuals and the model parameters once again with the `plot_residuals_and_coeff` function. The result is shown in Figure 15-3. In the rightmost panel of this figure we see that the coefficient profile is significantly different from those shown in Figure 15-1 and Figure 15-2, and the coefficient vector produced with the Lasso regression contains mostly zeros. This is a suitable method to the current data because in the beginning, when we generated the dataset, we choose 50 features out of which only 10 are informative. If we suspect that we might have a large number of features that might not contribute much in the regression model, using the L1 regularization of the LASSO regression can thus be a good approach to try.

```
In [33]: fig, ax = plot_residuals_and_coeff(resid_train, resid_test, model.coef_)
```

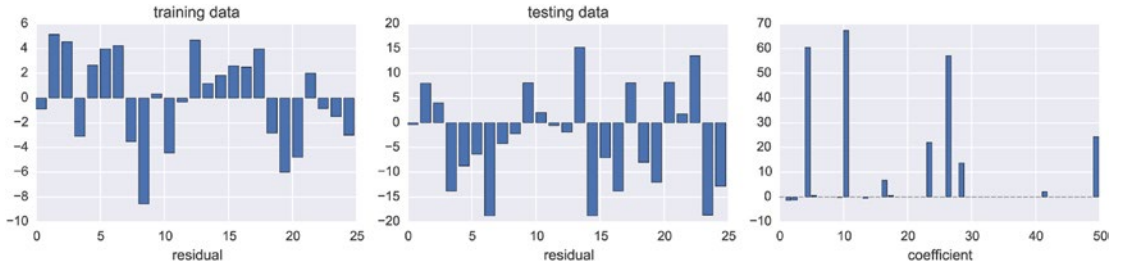

Figure 15-3. *The residual between the LASSO regulalized regression model and the training data (left), the model and the test data (middle), and the values of the coefficients for the 50 features (right)*

The values of α that we used in the two previous examples using Ridge and LASSO regression were chosen arbitrarily. The most suitable value of α is problem dependent, and for every new problem we need to find a suitable value using trial and error. The scikit-learn library provides methods for assisting this process, as we will see below, but before we explore those methods it is instructive to explore how the regression model parameters and the SSE for the training and testing datasets depend on the value of α for a specific problem. Here we focus on LASSO regression, since it was seen to work well for the current problem, and we repeatedly solve the same problem using different values for the regularization strength parameter α, while storing the values of the coefficients and SSE values in NumPy arrays.

We begin with creating the required NumPy arrays. We use `np.logspace` to create a range of α values that spans several orders of magniture:

```
In [34]: alphas = np.logspace(-4, 2, 100)
In [35]: coeffs = np.zeros((len(alphas), X_train.shape[1]))
In [36]: sse_train = np.zeros_like(alphas)
In [37]: sse_test = np.zeros_like(alphas)
```

Next we loop through the α values and perform the LASSO regression for each value:

```
In [38]: for n, alpha in enumerate(alphas):
    ...:     model = linear_model.Lasso(alpha=alpha)
    ...:     model.fit(X_train, y_train)
    ...:     coeffs[n, :] = model.coef_
    ...:     sse_train[n] = sse(y_train - model.predict(X_train))
    ...:     sse_test[n] = sse(y_test - model.predict(X_test))
```

Finally we plot the coefficients and the SSE for the training and testing datasets using Matplotlib. The result is shown in Figure 15-4. We can see in the left panel of this figure that a large number coefficients are nonzero for very small values of α which corresponds to the overfitting regime, and also that when α is increased above a certain threshold, many of the coefficients collapse to zero and only a few coefficients remain nonzero. This is the sought-after effect in LASSO regression, and in the right panel of the figure we see that while the SSE for the training set is steadily increasing with increasing α, there is also a sharp drop in the SSE for the testing dataset. For too large values of α all coefficients converges to zero and the SSE for both the training and testing datasets becomes large. There is therefore an optimal region of α that prevents overfitting and improves the models ability to predict unseen data. While these observations are not universally true, a similar pattern can be seen for many problems.

```
In [39]: fig, axes = plt.subplots(1, 2, figsize=(12, 4), sharex=True)
    ...: for n in range(coeffs.shape[1]):
    ...:     axes[0].plot(np.log10(alphas), coeffs[:, n], color='k', lw=0.5)
    ...:
    ...: axes[1].semilogy(np.log10(alphas), sse_train, label="train")
    ...: axes[1].semilogy(np.log10(alphas), sse_test, label="test")
    ...: axes[1].legend(loc=0)
    ...:
    ...: axes[0].set_xlabel(r"${\log_{10}}\alpha$", fontsize=18)
    ...: axes[0].set_ylabel(r"coefficients", fontsize=18)
    ...: axes[1].set_xlabel(r"${\log_{10}}\alpha$", fontsize=18)
    ...: axes[1].set_ylabel(r"sse", fontsize=18)
```

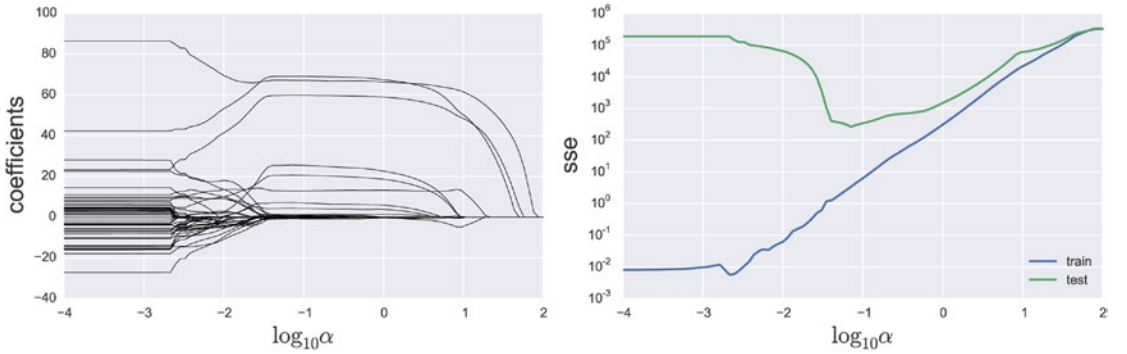

Figure 15-4. *The coefficients (left) and sum of squared errors (SSE) for the training and testing datasets (right), for LASSO regression as a function of the logartihm of the regularization strength parameter* α

The process of testing a regularized regression with several values of α can be carried out automatically using, for example, the `RidgeCV` and `LassoCV` classes. These variants of the Ridge and LASSO regression internally perform a search for the optimal α using a cross-validation approach. By default a k-fold cross-validation with $k = 3$ is used, although this can be changed using the `cv` argument to the classes. Because of the built-in cross-validation we do not need to explicitly divide the dataset into training and testing datasets, as we have done previously.

To use the LASSO method with an automatically chosen α, we simply create and instance of `LassoCV` and invoke its `fit` method:

```
In [40]: model = linear_model.LassoCV()
In [41]: model.fit(X_all, y_all)
Out[41]: LassoCV(alphas=None, copy_X=True, cv=None, eps=0.001, fit_intercept=True,
                 max_iter=1000, n_alphas=100, n_jobs=1, normalize=False, positive=False,
                 precompute='auto', random_state=None, selection='cyclic', tol=0.0001,
                 verbose=False)
```

The value of regularization strength parameter α selected through the cross-validation search is accessible through the `alpha_` attribute:

```
In [42]: model.alpha_
Out[42]: 0.13118477495069433
```

We note that the suggested value of α agrees reasonable well with what we might have guessed from Figure 15-4. For comparison with the previous method we also compute the SSE for the training and testing datasets (although both were used for training in the call to `LassoCV.fit`), and graph the SSE values together with the model parameters, as shown in Figure 15-5. By using the cross-validated LASSO method we obtain a model that predicts both the training and testing datasets with relatively high accuracy, and we are no longer as likely to suffer from the problem of overfitting, in spite of having few samples compared to the number of features.[5]

```
In [43]: sse_train = sse(y_train - model.predict(X_train))
    ...: sse_train
Out[43]: 66.900068715063625
In [44]: sse_test = sse(y_test - model.predict(X_test))
    ...: sse_test
Out[44]: 966.39293785448456
In [45]: fig, ax = plot_residuals_and_coeff(resid_train, resid_test, model.coef_)
```

Figure 15-5. *The residuals of the LASSO regularized regression model with cross-validation for the training data (left) and the testing data (middle). The values of the coefficients for the 50 features are also shown (right)*

Finally, yet another type of popular regularized regression, which combines the L1 and L2 regularization of the LASSO and Ridge methods, is known as elastic net. The minimization objective function for this method is $\min_\beta \left\{ \|X\beta - y\|_2^2 + \alpha\rho\|\beta\|_1 + \alpha(1-\rho)\|\beta\|_2^2 \right\}$, where the parameter ρ (`l1_ratio` in scikit-learn) determines the relative weight of the L1 and L2 penalties, and thus how much the method behaves like the LASSO and Ridge methods. In scikit-learn, we can perform an elastic net regression using the `ElasticNet` class, to which we can give explicit values of the α (`alpha`) and ρ (`l1_ratio`) parameters, or the cross-validated version `ElasticNetCV`, which automatically finds suitable values of the α and ρ parameters:

```
In [46]: model = linear_model.ElasticNetCV()
In [47]: model.fit(X_train, y_train)
Out[47]: ElasticNetCV(alphas=None, copy_X=True, cv=None, eps=0.001, fit_intercept=True,
                      l1_ratio=0.5, max_iter=1000, n_alphas=100, n_jobs=1,
                      normalize=False, positive=False, precompute='auto',
                      random_state=None, selection='cyclic', tol=0.0001, verbose=0)
```

[5]However, note that we can never be sure that a machine learning application does not suffer from overfitting before we see how the application performs on new observations, and a repeated reevaluation of the application on a regular basis is a good practice.

The value of regularization parameters α and ρ suggested by the cross-validation search are available throught the `alpha_` and `l1_ratio` attributes:

```
In [48]: model.alpha_
Out[48]: 0.13118477495069433
In [49]: model.l1_ratio
Out[49]: 0.5
```

For comparison with the previous method we once again compute the SSE and plot the model coefficients, as shown in Figure 15-6. As expected with $\rho = 0.5$, the result has characteristics of both LASSO regression (favoring a sparse solution vector with only a few dominating elements) and Ridge regression (suppressing the magnitude of the coefficients).

```
In [50]: sse_train = sse(y_train - model.predict(X_train))
    ...: sse_train
Out[50]: 2183.8391729391255
In [51]: sse_test = sse(y_test - model.predict(X_test))
    ...: sse_test
Out[51]: 2650.0504463382508
In [52]: fig, ax = plot_residuals_and_coeff(resid_train, resid_test, model.coef_)
```

Figure 15-6. *The residuals of the elastic-net regularized regression model with cross-validation for the training data (left) and the testing data (middle). The values of the coefficients for the 50 features are also shown (right)*

Classification

Like regression, classification is a central topic in machine learning. In Chapter 14, about statistical modeling, we already saw examples of classification, where we used a logistic regression model to classify observations into discrete categories. Logistic regression is also used in machine learning for the same task, but there are also a wide variety of alternative algorithms for classification, such as decision trees, nearest neighbor methods, support-vector machines (SVM), and Random Forest methods. The scikit-learn library provides a convenient unified API that allows all these different methods to be used interchangeably for any given classification problems.

To see how we can train a classification model with a training dataset and tests its performance on a testing dataset, let's once again look at the Iris datasets, which provides features for Iris flower samples (sepal and petal width and height), together with the species of each sample (Setosa, Versicolor, and Virginica). The Iris dataset that is included in the scikit-learn library (as it is in the statsmodels library) is a classic dataset that is commonly used for testing and demonstrating machine-learning algorithms and statistical models. We therefore here once again revisit the classification problem in which we wish to correctly classify

the species of a flower sample given its sepal and petal width and height (see also Chapter 14). First, to load the dataset we call the `load_iris` function in the `datasets` module. The result is a container object (called a `Bunch` object in scikit-learn jargon) that contains the data as well as metadata.

```
In [53]: iris = datasets.load_iris()
In [54]: type(iris)
Out[54]: sklearn.datasets.base.Bunch
```

For example, descriptive names of the features and target classes are available through the `feature_names` and `target_names` attributes:

```
In [55]: iris.target_names
Out[55]: array(['setosa', 'versicolor', 'virginica'], dtype='|S10')
In [56]: iris.feature_names
Out[56]: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

and the actual dataset is available through the `data` and `target` attributes:

```
In [57]: iris.data.shape
Out[57]: (150, 4)
In [58]: iris.target.shape
Out[58]: (150,)
```

We begin by splitting the dataset into a training and testing part, using the `train_test_split` function. There we chose to include 70% of the samples in the training set, leaving the remaining 30% for testing and validation:

```
In [59]: X_train, X_test, y_train, y_test = \
    ...:      cross_validation.train_test_split(iris.data, iris.target, train_size=0.7)
```

The first step in training a classifier and performing classification tasks using scikit-learn is to create a classifier instance. There are, as mentioned above and demonstrated in the following, numerous available classifiers. We begin with a logistic regression classifier, which is provided by the `LogisticRegression` class in the `linear_model` module:

```
In [60]: classifier = linear_model.LogisticRegression()
```

The training of the classifier is accomplished by calling the `fit` method of the classifier instance. The arguments are the design matrices for the feature and target variables. Here we use the training part of the Iris dataset arrays that was created for us when loading the dataset using the `load_iris` function. If the design matrices are not already available we can use the same techniques that we used in Chapter 14: that is, constructing the matrices by hand using NumPy functions or use the Patsy library to automatically construct the appropriate arrays. We can also use the feature extraction utilities in `feature_extraction` module in the scikit-learn library.

```
In [61]: classifier.fit(X_train, y_train)
Out[61]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr',
                   penalty='l2', random_state=None, solver='liblinear', tol=0.0001, verbose=0)
```

Once the classifier has been trained with data, we can immediately start using it for predicting the class for new observations using the `predict` method. Here we apply this method to predict the class for the samples assigned to the testing datasets, so that we can compare the predictions with the actual values.

```
In [62]: y_test_pred = classifier.predict(X_test)
```

The `sklearn.metrics` module contains helper functions for assisting in the analysis of the performance and accuracy of classifiers. For example, the `classification_report` function, which takes arrays of actual values and the predicted values, returns a tabular summary of the informative classification metrics related to the rate of false negatives and false positives:

```
In [63]: print(metrics.classification_report(y_test, y_test_pred))
             precision    recall  f1-score   support
          0       1.00      1.00      1.00        13
          1       1.00      0.92      0.96        13
          2       0.95      1.00      0.97        19
avg / total       0.98      0.98      0.98        45
```

The so-called confusion matrix, which can be computed using the `confusion_matrix` function, also presents useful classification metrics in a compact form: the diagonals correspond to the number of samples that are correctly classified for each level of the category variable, and the off-diagonal elements are the number of incorrectly classified samples. More specifically, the element C_{ij} of the confusion matrix C is the number of samples of category i that were categorized as j. For the current data we obtain the confusion matrix:

```
In [64]: metrics.confusion_matrix(y_test, y_test_pred)
Out[64]: array([[13  0  0]
                [ 0 12  1]
                [ 0  0 19]])
```

This confusion matrix shows that all elements in the first and third class were classified correctly, but one element of the second class was mistakenly classified as class 3. Note that the elements in each row of the confusion matrix sum up to the total number of samples for the corresponding category. In this testing sample we therefore have 13 elements each in the first and second class, and 19 elements of the third class, as also can be seen by counting unique value in the `y_test` array:

```
In [65]: np.bincount(y_test)
Out[65]: array([13, 13, 19])
```

To perform a classification using a different classifier algorithm, all we need to do is to create an instance of the corresponding classifier class. For example, to use a decision tree instead of logistic regression, we can use the `DesicisionTreeClassifier` class from the `sklearn.tree` module. Training the classifier and predicting new observations is done in exactly the same way for all classifiers:

```
In [66]: classifier = tree.DecisionTreeClassifier()
    ...: classifier.fit(X_train, y_train)
    ...: y_test_pred = classifier.predict(X_test)
    ...: metrics.confusion_matrix(y_test, y_test_pred)
Out[66]: array([[13,  0,  0],
                [ 0, 12,  1],
                [ 0,  1, 18]])
```

With the decision tree classifier the resulting confusion matrix is somewhat different, corresponding to one additional misclassification in the testing dataset.

Other popular classifiers that are available in scikit-learn include, for example, the nearest neighbor classifier `KNeighborsClassifier` form the `sklearn.neighbors` module, support-vector classifier `SVC` from the `sklearn.svm` module, and the Random Forest classifier `RandomForestClassifier` from the `sklearn.ensemble` module. Since they all have the same usage pattern, we can programmatically apply a series of classifiers on the same problem and compare their performance (on this particular problem), for example, as a function of the training and testing sample sizes. To this end, we create a NumPy array with training size ratios, ranging from 10% to 90%:

```
In [67]: train_size_vec = np.linspace(0.1, 0.9, 30)
```

Next we create a list of classifier classes that we wish to apply:

```
In [68]: classifiers = [tree.DecisionTreeClassifier,
    ...:                neighbors.KNeighborsClassifier,
    ...:                svm.SVC,
    ...:                ensemble.RandomForestClassifier]
```

and an array in which we can store the diagonals of the confusion matrix as a function of training size ratio and classifier:

```
In [69]: cm_diags = np.zeros((3, len(train_size_vec), len(classifiers)), dtype=float)
```

Finally, we loop over each training size ratio and classifier, and for each combination we train the classifier, predict the values of the testing data, compute the confusion matrix and store its diagonal divided by the ideal values in the `cm_diags` array:

```
In [70]: for n, train_size in enumerate(train_size_vec):
    ...:     X_train, X_test, y_train, y_test = \
    ...:         cross_validation.train_test_split(iris.data, iris.target,
    ...:                                           train_size=train_size)
    ...:     for m, Classifier in enumerate(classifiers):
    ...:         classifier = Classifier()
    ...:         classifier.fit(X_train, y_train)
    ...:         y_test_p = classifier.predict(X_test)
    ...:         cm_diags[:, n, m] = metrics.confusion_matrix(y_test, y_test_p).diagonal()
    ...:         cm_diags[:, n, m] /= np.bincount(y_test)
```

The resulting classification accuracy for each classifier, as a function of training size ratio, is plotted below and shown in Figure 15-7.

```
In [71]: fig, axes = plt.subplots(1, len(classifiers), figsize=(12, 3))
    ...: for m, Classifier in enumerate(classifiers):
    ...:     axes[m].plot(train_size_vec, cm_diags[2, :, m], label=iris.target_names[2])
    ...:     axes[m].plot(train_size_vec, cm_diags[1, :, m], label=iris.target_names[1])
    ...:     axes[m].plot(train_size_vec, cm_diags[0, :, m], label=iris.target_names[0])
    ...:     axes[m].set_title(type(Classifier()).__name__)
    ...:     axes[m].set_ylim(0, 1.1)
    ...:     axes[m].set_ylabel("classification accuracy")
    ...:     axes[m].set_xlabel("training size ratio")
    ...:     axes[m].legend(loc=4)
```

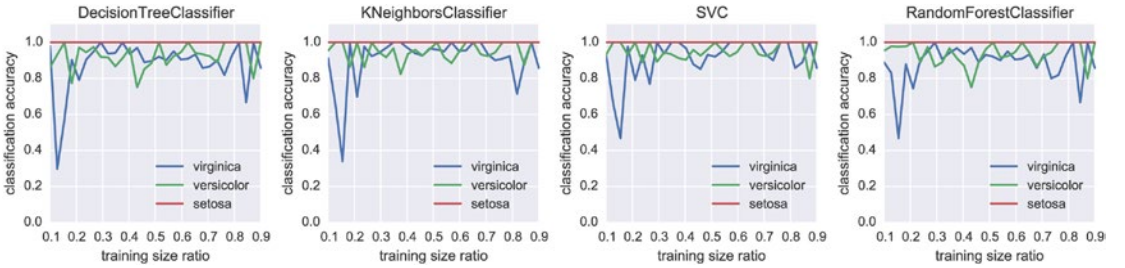

Figure 15-7. *Comparison of classification accuracy of four different classifiers*

In Figure 15-7, we see that classification error is different for each model, but for this particular example they have comparable performance. Which classifier is best depends on the problem at hand, and it is difficult to give any definite answer to which one is more suitable in general. Fortunately, it is easy to switch between different classifiers in scikit-learn, and therefore effortless to try a few different classifier for a given classification problem. In addition to the classification accuracy, another important aspect is the computational performance and scaling to larger problems. For large classification problems, with many features, decision tree methods such as Random Forest is often a good starting point.

Clustering

In the two previous sections we explored regression and classification, which are both examples of supervised learning, since the response variables are given in the dataset. Clustering is a different type of problem that is also an important topic of machine learning. It can be thought of as a classification problem where the classes are unknown, which makes clustering an example of unsupervised learning. The training dataset for a clustering algorithm therefore contains only the feature variables, and the output of the algorithm is an array of integers that assigns each sample to a cluster (or class). This output array corresponds to the response variable in a supervised classification problem.

The scikit-learn library implements a large number of clustering algorithms that are suitable for different types of clustering problems and for different types of datasets. Popular general-purpose clustering methods include the *K-means algorithm*, which groups the samples into clusters such that the within-group sum of square deviation from the group center is minimized, and the *mean-shift algorithm*, which clusters the samples by fitting the data to density functions (for example Gaussian functions).

In scikit-learn, the `sklearn.cluster` module contains several clustering algorithms, including the K-means algorithm `KMeans`, and the Mean-shift algorithm `MeanShift`, just to mention a few. To perform a clustering task with one of these methods we first initialize an instance of the corresponding class, train it with a feature-only dataset using the `fit` method, and we finally obtain the result of the clustering by calling the `predict` method. Many clustering algorithm require the number of clusters as an input parameters, which we can specify using the `n_clusters` parameter when the class instance is created.

As an example of clustering, consider again the Iris dataset that we used in the previous section, but now we will not use the response variable, which was used in supervised classification, but instead we attempt to automatically discovering a suitable clustering of the samples using the *K*-means method. We begin by loading the Iris data as before, and store the feature and target data in the variables `X` and `y`, respectively:

```
In [72]: X, y = iris.data, iris.target
```

With the K-means clustering method we need to specify how many clusters we want in the output. The most suitable number of clusters is not always obvious in advance, and trying clustering with a few different numbers of clusters is often necessary. However, here we know that the data corresponds to three different species of Iris flowers, so we use three clusters. To perform the clustering we create an instance of `KMeans` class, using the `n_clusters` argument to set the number of clusters.

```
In [73]: n_clusters = 3
In [74]: clustering = cluster.KMeans(n_clusters=n_clusters)
```

To actually perform the computation we call the `fit` method with the Iris feature matrix as argument:

```
In [75]: clustering.fit(X)
Out[75]: KMeans(copy_x=True, init='k-means++', max_iter=300, n_clusters=3, n_init=10,
                n_jobs=1, precompute_distances='auto', random_state=None, tol=0.0001,
                verbose=0)
```

The clustering result is available through the `predict` method, to which we also pass a feature dataset that optionally can contain features of new samples. However, not all the clustering methods implemented in scikit-learn support predicting clusters for new sample. In this case the `predict` method is not available, and we need to use the `fit_predict` method instead. Here, we use the `predict` method with the training feature dataset to obtain the clustering result:

```
In [76]: y_pred = clustering.predict(X)
```

The result is an integer array of the same length and the number of samples in the training dataset. The elements in the array indicate which group (from `0` up to `n_samples-1`) each sample is assigned to. Since the resulting array `y_pred` is long, we only display every 8th element in the array using the NumPy stride indexing `::8`.

```
In [77]: y_pred[::8]
Out[77]: array([1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0], dtype=int32)
```

We can compare the obtained clustering with the supervised classification of the Iris samples:

```
In [78]: y[::8]
Out[78]: array([0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2])
```

There seems to be a good correlation between the two, but the output of the clustering has assigned different integer values to the groups than what was used in the target vector in the supervised classification. To be able to compare the two arrays with metrics such as the `confusion_matrix` function, we first need to rename the elements so that the same integer values are used for the same group. We can do this operation with NumPy array manipulations:

```
In [79]: idx_0, idx_1, idx_2 = (np.where(y_pred == n) for n in range(3))
In [80]: y_pred[idx_0], y_pred[idx_1], y_pred[idx_2] = 2, 0, 1
In [81]: y_pred[::8]
Out[81]: array([0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2], dtype=int32)
```

Now that the corresponding groups are represented with the same integers, we can summarize the overlaps between the supervised and unsupervised classification of the Iris samples using the `confusion_matrix` function:

```
In [82]: metrics.confusion_matrix(y, y_pred)
Out[82]: array([[50,  0,  0],
                [ 0, 48,  2],
                [ 0, 14, 36]])
```

This confusion matrix indicates that the clustering algorithm was able to correctly identity all samples corresponding to the first species as a group of its own, but due to the overlapping samples in the second and third group those could not be completely resolved as different groups, as 2 elements from group 1 was assigned to group 2, and 14 elements from group 2 was assigned to group 1.

The result of the clustering can also be visualized by plotting scatter plots for each pair of features, as we do in the following. We loop over each pair of features and each cluster and plot a scatter graph for each cluster using different colors (orange, blue, and green, displayed as different shades of gray in Figure 15-8), and we also draw a red square around each sample for which the clustering does not agree with the supervised classification. The result is shown in Figure 15-8.

```
In [83]: N = X.shape[1]
    ...: fig, axes = plt.subplots(N, N, figsize=(12, 12), sharex=True, sharey=True)
    ...: colors = ["coral", "blue", "green"]
    ...: markers = ["^", "v", "o"]
    ...: for m in range(N):
    ...:     for n in range(N):
    ...:         for p in range(n_clusters):
    ...:             mask = y_pred == p
    ...:             axes[m, n].scatter(X[:, m][mask], X[:, n][mask], s=30,
    ...:                                marker=markers[p], color=colors[p], alpha=0.25)
    ...:         for idx in np.where(y != y_pred):
    ...:             axes[m, n].scatter(X[idx, m], X[idx, n], s=30,
    ...:                                marker="s", edgecolor="red", facecolor=(1,1,1,0))
    ...:     axes[N-1, m].set_xlabel(iris.feature_names[m], fontsize=16)
    ...:     axes[m, 0].set_ylabel(iris.feature_names[m], fontsize=16)
```

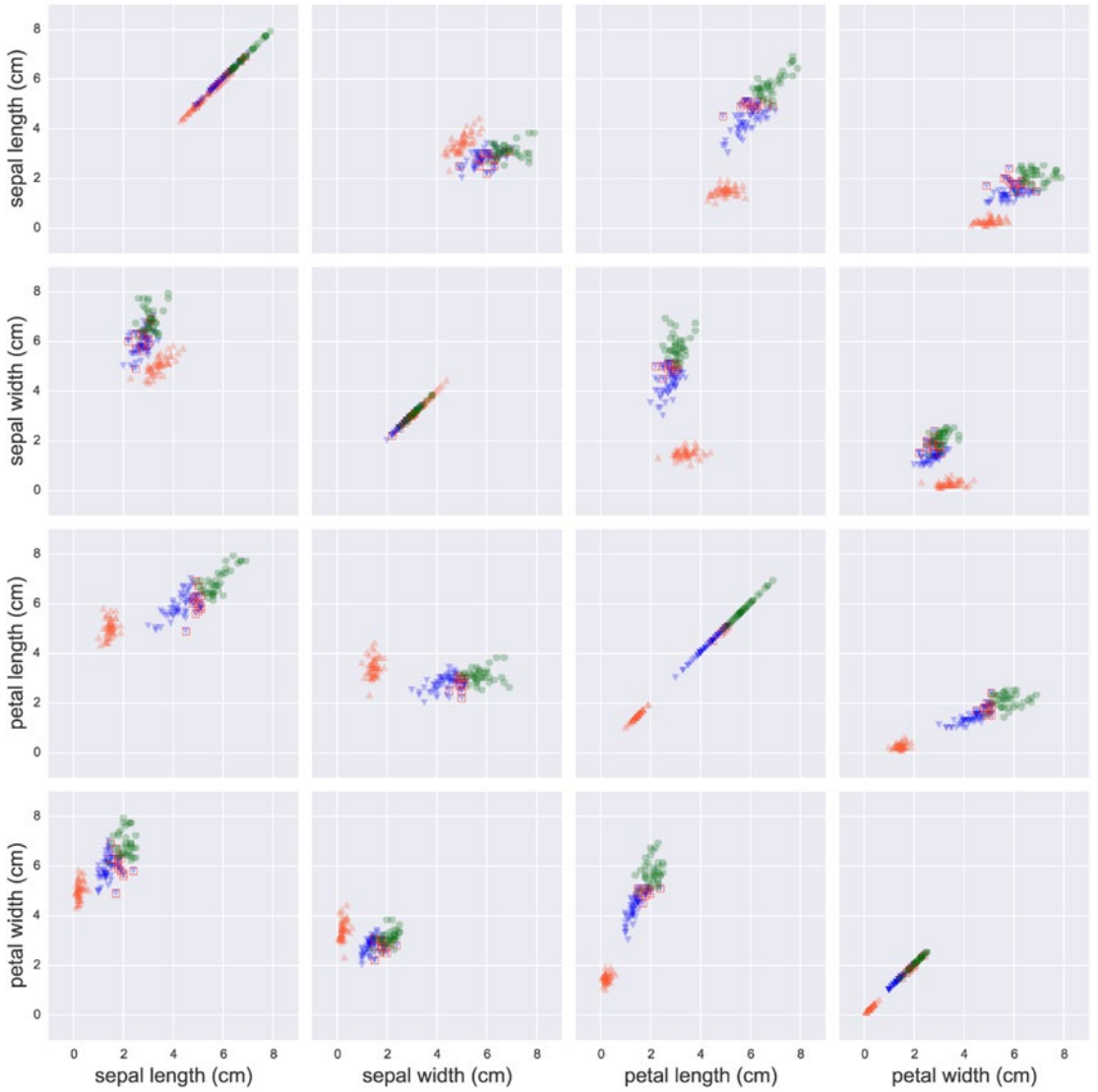

Figure 15-8. *The result of clustering, using the K-means algorithm, of the Iris dataset features*

The result of the clustering of the Iris samples in Figure 15-8 shows that the clustering does a remarkably good job at recognizing which samples belongs to distinct groups. Of course, because of the overlap in the features for classes shown in blue (dark gray) and green (medium gray) in the graph, we cannot expect that any unsupervised clustering algorithm can fully resolve the various groups in the dataset, and some deviation from the supervised response variable is therefore expected.

Summary

In this chapter we have given an introduction to machine learning using Python. We began with a brief review and summary of the subject and its terminology, and continued with introducing the Python library scikit-learn, which we applied in three different types of problems that are fundamental topics in machine learning: First we revisited regression, from the point of view of machine learning, followed by classification, and finally we considered examples of clustering. The first two of these topics are examples of supervised machine learning, while the clustering method is an example of unsupervised machine learning. Beyond of what we have been able to cover here, there are many more methods and problem domains covered by the broad subject of machine learning. For example, an important part of machine learning that we have not touched upon in this brief introduction is text-based problems. The scikit-learn contains an extensive module (`sklearn.text`) with tools and method for processing text-based problems, and the Natural Language Toolkit (`http://www.nltk.org`) is a powerful platform for working with and processing data in the form of human language text. Image processing and computer vision is another prominent problem domain in machine learning, which for example can be treated with OpenCV (`http://opencv.org`) and its Python bindings. Other examples of big topics in machine learning are neural networks and deep learning, which are have received much attention in recent years. The readers who are interested in such methods are recommended to explore the Python libraries Theano (`http://www.deeplearning.net/software/theano`), Lasange (`http://lasagne.readthedocs.org/en/latest`), pylearn2 (`http://deeplearning.net/software/pylearn2`), and PyBrain (`http://pybrain.org`).

Further Reading

Machine learning is a part of the computer science field artificial intelligence, which is a broad field with numerous techniques, methods, and applications. In this chapter we have only been able to show examples of a few basic machine-learning methods, which nonetheless can be useful in many practical applications. For a more thorough introduction to machine learning see Hastie's book and for introductions to machine learning specific to the Python environment, see, for example, books by Garreta, Hackeling, or Coelho.

References

Pedro Coelho, W. R. (2015). *Building Machine Learning Systems with Python.* Mumbai: Packt.

Garreta, G. M. (2013). *Learning Scikit-Learn: Machine Learning in Python.* Mumbai: Packt.

Hackeling, G. (2014). *Mastering Machine Learning with scikit-learn.* Mumbai: Packt.

Hastie, R. T. (2013). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* New York: Springer.