CHAPTER 8

Machine Learning with scikit-learn

In the chain of processes that make up the data analysis, the construction phase of predictive models and their validation are done by a powerful library called **scikit-learn**. In this chapter you will see some examples that will illustrate the basic construction of predictive models with some different methods.

The scikit-learn Library

scikit-learn is a Python module that integrates many of machine learning algorithms. This library was developed initially by Cournapeu in 2007, but the first real release was in 2010.

This library is part of the SciPy (Scientific Python) group, a set of libraries created for scientific computing and especially for data analysis, many of which are discussed in this book. Generally these libraries are defined as **SciKits**, hence the first part of the name of this library. The second part of the library's name is derived from **Machine Learning**, the discipline pertaining to this library.

Machine Learning

Machine Learning is a discipline that deals with the study of methods for pattern recognition in data sets undergoing data analysis. In particular, it deals with the development of algorithms that learn from data and make predictions. Each methodology is based on building a specific model.

There are very many methods that belong to the learning machine, each with its unique characteristics, which are specific to the nature of the data and the predictive model that you want to build. The choice of which method is to be applied is called **learning problem**.

The data to be subjected to a pattern in the learning phase can be arrays composed by a single value per element, or by a multivariate value. These values are often referred to as **features** or **attributes**.

Supervised and Unsupervised Learning

Depending on the type of the data and the model to be built, you can separate the learning problems into two broad categories:

Supervised learning. They are the methods in which the training set contains additional attributes that you want to predict (**target**). Thanks to these values, you can instruct the model to provide similar values when you have to submit new values (**test set**).

- **Classification**: the data in the training set belong to two or more classes or categories; then, the data, already being labeled, allow us to teach the system to recognize the characteristics that distinguish each class. When you will need to consider a new value unknown to the system, the system will evaluate its class according to its characteristics.

- **Regression**: when the value to be predicted is a continuous variable. The simplest case to understand is when you want to find the line which describes the trend from a series of points represented in a scatterplot.

Unsupervised learning. These are the methods in which the training set consists of a series of input values x without any corresponding target value.

- **Clustering**: the goal of these methods is to discover groups of similar examples in a dataset.
- **Dimensionality reduction**: reduction of a high-dimensional dataset to one with only two or three dimensions is useful not just for data visualization, but for converting data of very high dimensionality into data of much lower dimensionality such that each of the lower dimensions conveys much more information.

In addition to these two main categories, there is a further group of methods which have the purpose of validation and evaluation of the models.

Training Set and Testing Set

Machine learning enables learning some properties by a model from a data set and applying them to new data. This is because a common practice in machine learning is to evaluate an algorithm. This valuation consists of splitting the data into two parts, one called the **training set**, with which we will learn the properties of the data, and the other called the **testing set**, on which to test these properties.

Supervised Learning with scikit-learn

In this chapter you will see a number of examples of **supervised learning**.

- Classification, using the Iris Dataset
  - K-Nearest Neighbors Classifier
  - Support Vector Machines (SVC)
- Regression, using the Diabetes Dataset
  - Linear Regression
  - Support Vector Machines (SVR)

Supervised learning consists of learning possible patterns between two or more features reading values from a training set; the learning is possible because the training set contains known results (**target** or **labels**). All models in scikit-learn are referred to as **supervised estimators**, using the **fit(x, y)** function that makes their training. *x* comprises the features observed, while *y* indicates the target. Once the estimator has carried out the training, it will be able to predict the value of *y* for any new observation *x* not labeled. This operation will make it through the **predict(x)** function.

The Iris Flower Dataset

The **Iris Flower Dataset** is a particular dataset used for the first time by Sir Ronald Fisher in 1936. It is often also called Anderson Iris Dataset, after the person who collected the data directly measuring the size of the various parts of the iris flowers. In this dataset, data from three different species of iris (Iris silky, virginica Iris, and Iris versicolor) are collected and exactly these data correspond to the length and width of the sepals and the length and width of the petals (see Figure 8-1).

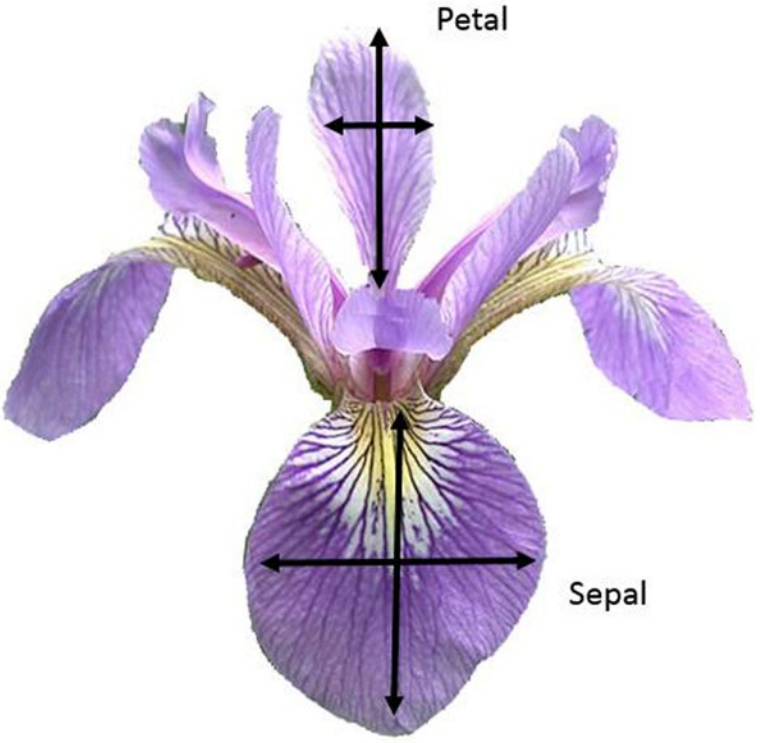

Figure 8-1. *Iris versicolor and the petal and sepal width and length*

This dataset is currently being used as a good example to utilize for many types of analysis, in particular as regards the problems of **classification** that can be approached by means of machine learning methodologies. It is no coincidence then that this dataset is provided along with the scikit-learn library as a 150x4 NumPy array.

Now you will study this dataset in detail importing it in the IPython QtConsole or in a normal Python session.

```
In [ ]: from sklearn import datasets
   ...: iris = datasets.load_iris()
```

In this way you loaded all the data and metadata concerning the Iris Dataset in the *iris* variable. In order to see the values of the data collected in it, it is sufficient to call the attribute **data** of the variable *iris*.

```
In [ ]: iris.data
Out[ ]:
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ 4.6,  3.1,  1.5,  0.2],
       ...
```

As you can see you will get an array of 150 elements, each containing 4 numeric values: the size of sepals and petals respectively.

To know instead what kind of flower belongs each item you will refer to the **target** attribute.

```
In [ ]: iris.target
Out[ ]:
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

You obtain 150 items with three possible integer values (0, 1 and 2) which correspond to the three species of iris. To know the correspondence between the species and number you have to call the **target_names** attribute.

```
In [ ]: iris.target_names
Out[ ]:
array(['setosa', 'versicolor', 'virginica'],
      dtype='|S10')
```

To better understand this data set you can use the matplotlib library, using the techniques you learned in Chapter 7. Therefore create a scatterplot that displays the three different species in three different colors. X-axis will represent the length of the sepal while the y-axis will represent the width of the sepal.

```
 In [ ]: import matplotlib.pyplot as plt
   ...: import matplotlib.patches as mpatches
   ...: from sklearn import datasets
   ...:
   ...: iris = datasets.load_iris()
   ...: x = iris.data[:,0]  #X-Axis - sepal length
   ...: y = iris.data[:,1]  #Y-Axis - sepal length
   ...: species = iris.target     #Species
   ...:
   ...: x_min, x_max = x.min() - .5,x.max() + .5
   ...: y_min, y_max = y.min() - .5,y.max() + .5
   ...:
   ...: #SCATTERPLOT
   ...: plt.figure()
   ...: plt.title('Iris Dataset - Classification By Sepal Sizes')
   ...: plt.scatter(x,y, c=species)
   ...: plt.xlabel('Sepal length')
   ...: plt.ylabel('Sepal width')
   ...: plt.xlim(x_min, x_max)
   ...: plt.ylim(y_min, y_max)
   ...: plt.xticks(())
   ...: plt.yticks(())
```

As a result you get the scatterplot as shown in Figure 8-2. In blue are represented the Iris setosa, flowers, green ones are the Iris versicolor, and red ones are the Iris virginica.

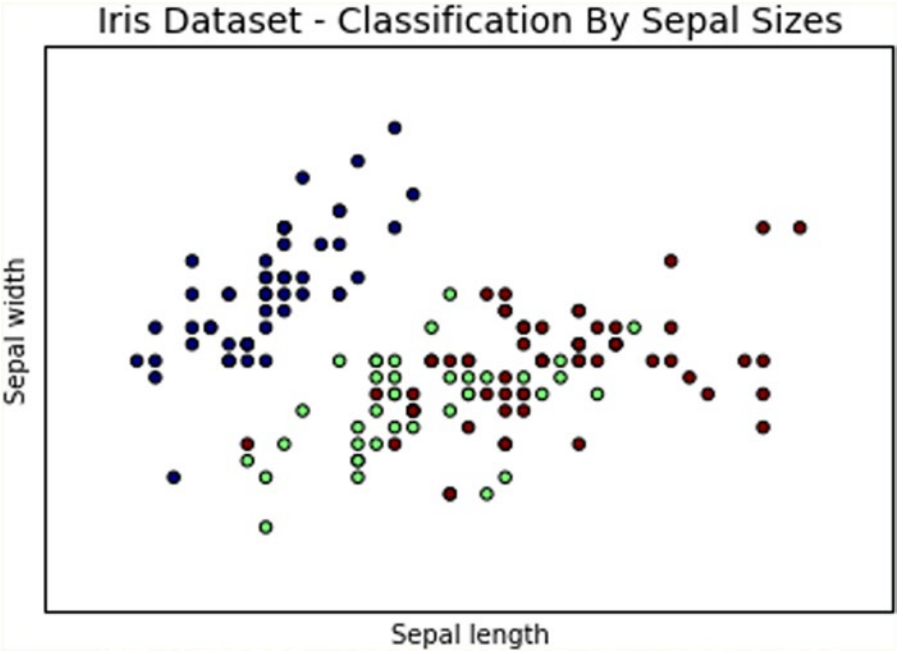

Figure 8-2. *The different species of irises are shown with different colors*

From Figure 8-2 you can see how the Iris setosa features differ from the other two, forming a cluster of blue dots separate from the others.

Try to follow the same procedure, but this time using the other two variables, that is the measure of the length and width of the petal. You can use the same code above only by changing some values.

```
In [ ]: import matplotlib.pyplot as plt
   ...: import matplotlib.patches as mpatches
   ...: from sklearn import datasets
   ...:
   ...: iris = datasets.load_iris()
   ...: x = iris.data[:,2]  #X-Axis - petal length
   ...: y = iris.data[:,3]  #Y-Axis - petal length
   ...: species = iris.target     #Species
   ...:
   ...: x_min, x_max = x.min() - .5,x.max() + .5
   ...: y_min, y_max = y.min() - .5,y.max() + .5
   ...: #SCATTERPLOT
   ...: plt.figure()
   ...: plt.title('Iris Dataset - Classification By Petal Sizes', size=14)
   ...: plt.scatter(x,y, c=species)
   ...: plt.xlabel('Petal length')
   ...: plt.ylabel('Petal width')
   ...: plt.xlim(x_min, x_max)
   ...: plt.ylim(y_min, y_max)
   ...: plt.xticks(())
   ...: plt.yticks(())
```

The result is the scatterplot shown in Figure 8-3. In this case the division between the three species is much more evident. As you can see you have three different clusters.

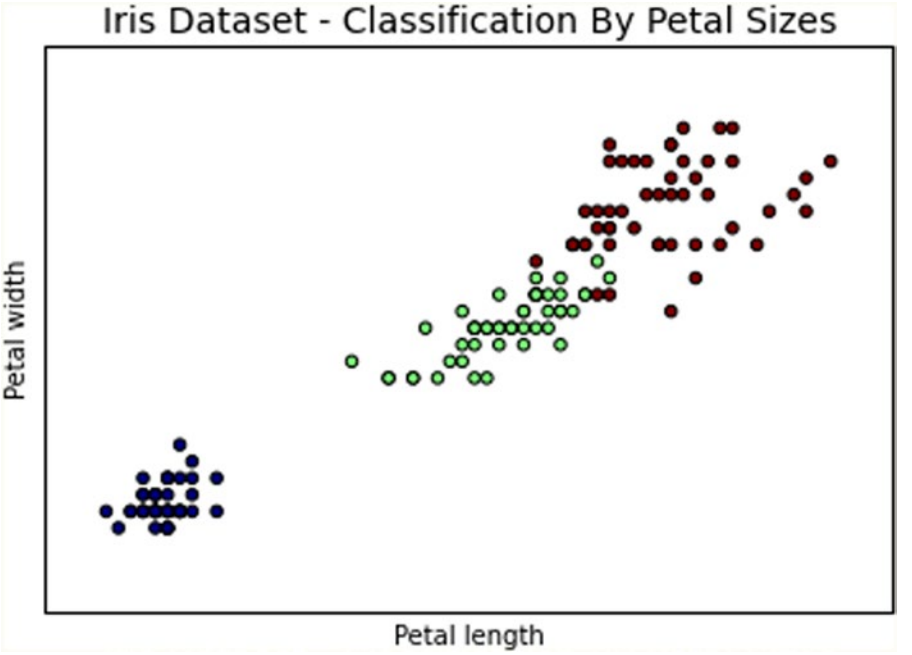

Figure 8-3. *The different species of irises are shown with different colors*

The PCA Decomposition

You have seen how the three species could be characterized taking into account four measurements of the petals and sepals size. We represented two scatterplots, one for the petals and one for sepals, but how can you can unify the whole thing? Four dimensions are a problem that even a Scatterplot 3D is not able to solve.

In this regard a special technique called **Principal Component Analysis (PCA)** has been developed. This technique allows you to reduce the number of dimensions of a system keeping all the information for the characterization of the various points, the new dimensions generated are called **principal components**. In our case, so you can reduce the system from 4 to 3 dimensions and then plot the results within a 3D scatterplot. In this way you can use measures both of sepals and of petals for characterizing the various species of iris of the test elements in the dataset.

The Scikit-learn function which allows you to do the dimensional reduction, is the **fit_transform()** function which belongs to the **PCA** object. In order to use it, first you need to import the PCA module **sklearn.decomposition**. Then you have to define the object constructor using PCA() defining the number of new dimensions (principal components) as value of the **n_components** option. In your case it is 3. Finally you have to call the **fit_transform()** function passing the four-dimensional Iris Dataset as argument.

```
from sklearn.decomposition import PCA
x_reduced = PCA(n_components=3).fit_transform(iris.data)
```

In addition, in order to visualize the new values you will use a scatterplot 3D using the **mpl_toolkits.mplot3d** module of matplotlib. If you don't remember how to do it, see the Scatterplot 3D section in Chapter 7.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets
from sklearn.decomposition import PCA

iris = datasets.load_iris()
x = iris.data[:,1]  #X-Axis - petal length
y = iris.data[:,2]  #Y-Axis - petal length
species = iris.target     #Species
x_reduced = PCA(n_components=3).fit_transform(iris.data)

#SCATTERPLOT 3D
fig = plt.figure()
ax = Axes3D(fig)
ax.set_title('Iris Dataset by PCA', size=14)
ax.scatter(x_reduced[:,0],x_reduced[:,1],x_reduced[:,2], c=species)
ax.set_xlabel('First eigenvector')
ax.set_ylabel('Second eigenvector')
ax.set_zlabel('Third eigenvector')
ax.w_xaxis.set_ticklabels(())
ax.w_yaxis.set_ticklabels(())
ax.w_zaxis.set_ticklabels(())
```

The result will be a scatterplot as shown in Figure 8-4. The three species of iris are well characterized with respect to each other to form a cluster.

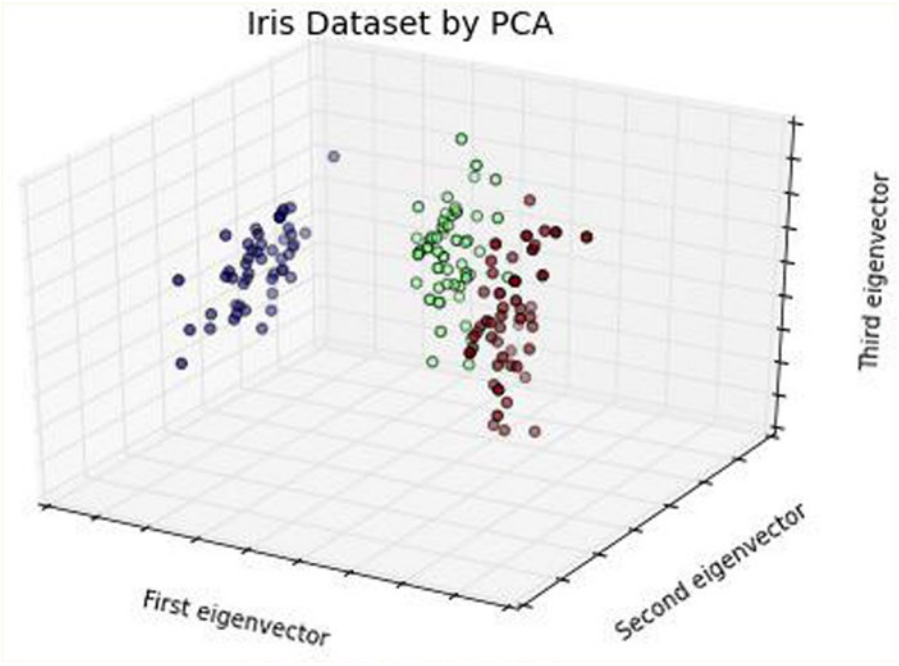

Figure 8-4. *3D scatterplot with three clusters representative of each species of iris*

K-Nearest Neighbors Classifier

Now, you will perform a **classification,** and to do this operation with the scikit-learn library you need a **classifier**.

Given a new measurement of an iris flower, the task of the classifier is to figure out to which of the three species it belongs. The simplest possible classifier is the **nearest neighbor**. This algorithm will search within the training set the observation that most closely approaches the new test sample.

A very important thing to consider at this point are the concepts of **training set** and **testing set** (already seen in Chapter 1). Indeed, if you have only a single dataset of data, it is important not to use the same data both for the test and for the training. In this regard, the elements of the dataset are divided into two parts, one dedicated to train the algorithm and the other to perform its validation.

Thus, before proceeding further you have to divide your *Iris* Dataset into two parts. However, it is wise to randomly mix the array elements and then make the division. In fact, often the data may have been collected in a particular order, and in your case the Iris Dataset contains items sorted by species. So to make a blending of elements of the data set you will use a NumPy function called **random.permutation()**. The mixed dataset consists of 150 different observations; the first 140 will be used as training set, the remaining 10 as test set.

```
import numpy as np
from sklearn import datasets
np.random.seed(0)
iris = datasets.load_iris()
x = iris.data
y = iris.target
i = np.random.permutation(len(iris.data))
x_train = x[i[:-10]]
y_train = y[i[:-10]]
x_test = x[i[-10:]]
y_test = y[i[-10:]]
```

Now you can apply the K-Nearest Neighbor algorithm. Import the KneighborsClassifier, call the constructor of the classifier, and then train it with the fit() function.

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
knn.fit(x_train,y_train)
Out[86]:
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_neighbors=5, p=2, weights='uniform')
```

Now that you have a predictive model which consists of the *knn* classifier, trained by 140 observations, you will find out how it is valid. The classifier should correctly predict the species of iris of the 10 observations of the test set. In order to obtain the prediction you have to use the **predict()** function, which will be applied directly to the predictive model *knn*. Finally, you will compare the results predicted with the actual observed contained in *y_test*.

```
knn.predict(x_test)
Out[100]: array([1, 2, 1, 0, 0, 0, 2, 1, 2, 0])
y_test
Out[101]: array([1, 1, 1, 0, 0, 0, 2, 1, 2, 0])
```

You can see that you obtained a 10% error. Now you can visualize all this using decision boundaries in a space represented by the 2D scatterplot of sepals.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
iris = datasets.load_iris()
x = iris.data[:,:2]      #X-Axis - sepal length-width
y = iris.target          #Y-Axis - species

x_min, x_max = x[:,0].min() - .5,x[:,0].max() + .5
y_min, y_max = x[:,1].min() - .5,x[:,1].max() + .5

#MESH
cmap_light = ListedColormap(['#AAAAFF','#AAFFAA','#FFAAAA'])
h = .02
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
knn = KNeighborsClassifier()
knn.fit(x,y)
Z = knn.predict(np.c_[xx.ravel(),yy.ravel()])
Z = Z.reshape(xx.shape)
plt.figure()
plt.pcolormesh(xx,yy,Z,cmap=cmap_light)

#Plot the training points
plt.scatter(x[:,0],x[:,1],c=y)
plt.xlim(xx.min(),xx.max())
plt.ylim(yy.min(),yy.max())

Out[120]: (1.5, 4.900000000000003)
```

You get a subdivision of the scatterplot in decision boundaries, as shown in Figure 8-5.

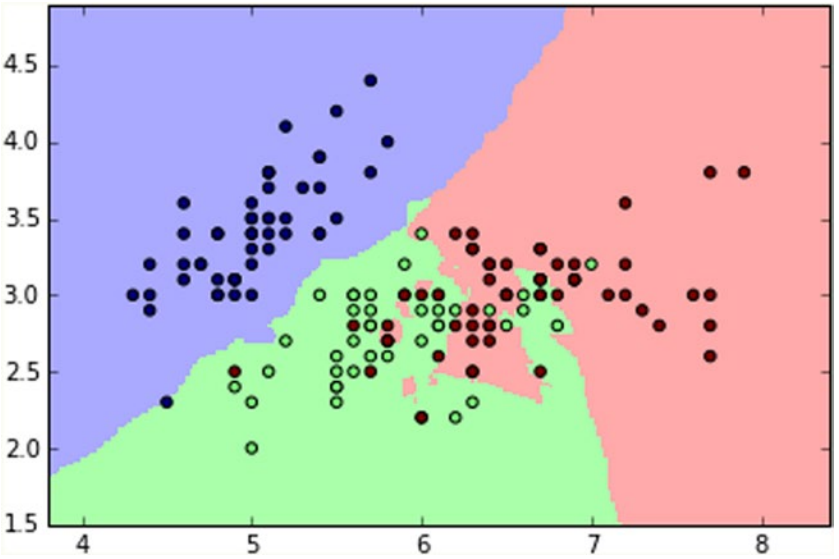

Figure 8-5. *The three decision boundaries are represented by three different colors*

You can do the same thing considering the size of the petals.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
iris = datasets.load_iris()
x = iris.data[:,2:4]      #X-Axis - petals length-width
y = iris.target           #Y-Axis - species

x_min, x_max = x[:,0].min() - .5,x[:,0].max() + .5
y_min, y_max = x[:,1].min() - .5,x[:,1].max() + .5

#MESH
cmap_light = ListedColormap(['#AAAAFF','#AAFFAA','#FFAAAA'])
h = .02
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
knn = KNeighborsClassifier()
knn.fit(x,y)
Z = knn.predict(np.c_[xx.ravel(),yy.ravel()])
Z = Z.reshape(xx.shape)
plt.figure()
plt.pcolormesh(xx,yy,Z,cmap=cmap_light)

#Plot the training points
plt.scatter(x[:,0],x[:,1],c=y)
plt.xlim(xx.min(),xx.max())
plt.ylim(yy.min(),yy.max())

Out[126]: (-0.40000000000000002, 2.9800000000000031)
```

As shown in Figure 8-6, you will have the corresponding decision boundaries regarding the characterization of iris flowers taking into account the size of the petals.

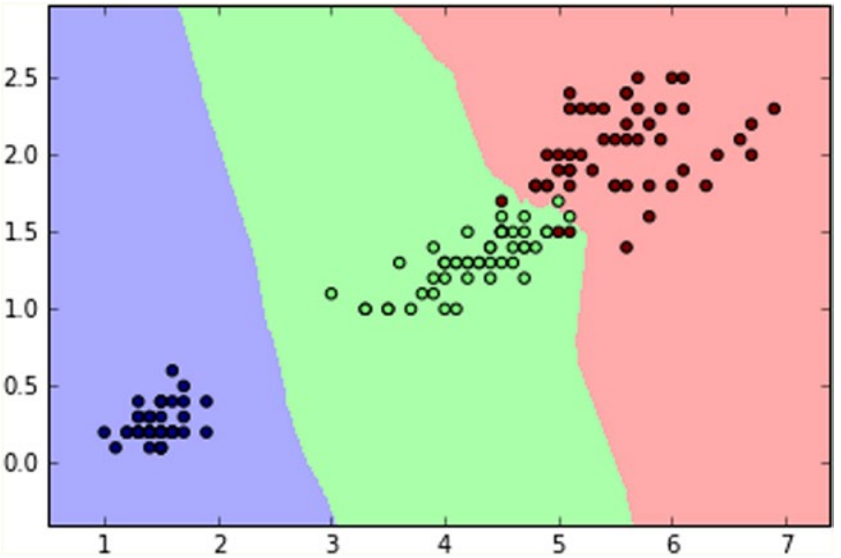

Figure 8-6. *The decision boundaries on a 2D scatterplot describing the petal sizes*

Diabetes Dataset

Among the various datasets available within the scikit-learn library, there is the diabetes dataset. This dataset was used for the first time in 2004 (*Annals of Statistics*, by Efron, Hastie, Johnston, and Tibshirani). Since then it has become an example widely used to study various predictive models and their effectiveness.

To upload the data contained in this dataset, before you have to import the **datasets** module of the scikit-learn library and then you call the **load_diabetes()** function to load the data set into a variable that will be called just *diabetes*.

```
In [ ]: from sklearn import datasets
   ...: diabetes = datasets.load_diabetes()
```

This dataset contains physiological data collected on 442 patients and as corresponding target an indicator of the disease progression after a year. The physiological data occupy the first 10 columns with values that indicate respectively:

- Age
- Sex
- Body Mass Index
- Blood Pressure
- S1, S2, S3, S4, S5, S6 (six blood serum measurements)

These measurements can be obtained by calling the **data** attribute. But going to check the values in the dataset you will find values very different from what you would have expected. For example, we look at the 10 values for the first patient.

```
diabetes.data[0]
Out[ ]:
array([ 0.03807591,  0.05068012,  0.06169621,  0.02187235, -0.0442235 ,
       -0.03482076, -0.04340085, -0.00259226,  0.01990842, -0.01764613])
```

These values are in fact the result of a processing. Each of the ten values was mean centered and subsequently scaled by the standard deviation times the number of the samples. Checking will reveal that the sum of squares of each column is equal to 1. Try doing this calculation with the age measurements; you will obtain a value very close to 1.

```
np.sum(diabetes.data[:,0]**2)
Out[143]: 1.0000000000000746
```

Even though these values are normalized and therefore difficult to read, they continue to express the 10 physiological characteristics and therefore have not lost their value or statistical information.

As for the indicators of the progress of the disease, that is, the values that must correspond to the results of your predictions, these are obtainable by means of the **target** attribute.

```
diabetes.target
Out[146]:
array([ 151.,   75.,  141.,  206.,  135.,   97.,  138.,   63.,  110.,
        310.,  101.,   69.,  179.,  185.,  118.,  171.,  166.,  144.,
         97.,  168.,   68.,   49.,   68.,  245.,  184.,  202.,  137
        . . .
```

You obtain a series of 442 integer values between 25 and 346.

Linear Regression: The Least Square Regression

Linear regression is a procedure that uses data contained in the training set to build a linear model. The most simple is based on the equation of a rect with the two parameters *a* and *b* to characterize it. These parameters will be calculated so as to make the sum of squared residuals as small as possible.

$$y = a^*x + c$$

In this expression, *x* is the training set, *y* is the target, *b* is the slope, and *c* is the intercept of the rect represented by the model. In scikit-learn, to use the predictive model for the linear regression you must import **linear_model** module and then use the manufacturer **LinearRegression()** constructor for creating the predictive model, which you call **linreg**.

```
from sklearn import linear_model
linreg = linear_model.LinearRegression()
```

For practicing with an example of linear regression you can use the diabetes dataset described earlier. First you will need to break the 442 patients into a training set (composed of the first 422 patients) and a test set (the last 20 patients).

```
from sklearn import datasets
diabetes = datasets.load_diabetes()
x_train = diabetes.data[:-20]
y_train = diabetes.target[:-20]
x_test = diabetes.data[-20:]
y_test = diabetes.target[-20:]
```

Now, apply the training set to the predictive model through the use of **fit()** function.

```
linreg.fit(x_train,y_train)
Out[ ]: LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
```

Once the model is trained you can get the ten *b* coefficients calculated for each physiological variable, using the **coef_** attribute of the predictive model.

```
linreg.coef_
Out[164]:
array([  3.03499549e-01,  -2.37639315e+02,   5.10530605e+02,
         3.27736980e+02,  -8.14131709e+02,   4.92814588e+02,
         1.02848452e+02,   1.84606489e+02,   7.43519617e+02,
         7.60951722e+01])
```

If you apply the test set to the **linreg** prediction model you will get a series of targets to be compared with the values actually observed.

```
linreg.predict(x_test)
Out[ ]:
array([ 197.61846908,  155.43979328,  172.88665147,  111.53537279,
        164.80054784,  131.06954875,  259.12237761,  100.47935157,
        117.0601052 ,  124.30503555,  218.36632793,   61.19831284,
        132.25046751,  120.3332925 ,   52.54458691,  194.03798088,
        102.57139702,  123.56604987,  211.0346317 ,   52.60335674])

y_test
Out[ ]:
array([ 233.,   91.,  111.,  152.,  120.,   67.,  310.,   94.,  183.,
         66.,  173.,   72.,   49.,   64.,   48.,  178.,  104.,  132.,
        220.,   57.])
```

However, a good indicator of what prediction should be perfect is the **variance**. The more the variance is close to 1 the more the prediction is perfect.

```
linreg.score(x_test, y_test)
Out[ ]: 0.5850753022690571 3
```

Now you will start with the linear regression taking into account a single physiological factor, for example, you can start from the age.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn import datasets
```

```
diabetes = datasets.load_diabetes()
x_train = diabetes.data[:-20]
y_train = diabetes.target[:-20]
x_test = diabetes.data[-20:]
y_test = diabetes.target[-20:]

x0_test = x_test[:,0]
x0_train = x_train[:,0]
x0_test = x0_test[:,np.newaxis]
x0_train = x0_train[:,np.newaxis]
linreg = linear_model.LinearRegression()
linreg.fit(x0_train,y_train)
y = linreg.predict(x0_test)
plt.scatter(x0_test,y_test,color='k')
plt.plot(x0_test,y,color='b',linewidth=3)

Out[230]: [<matplotlib.lines.Line2D at 0x380b1908>]
```

Figure 8-7 shows the blue line representing the linear correlation between the ages of patients and the disease progression.

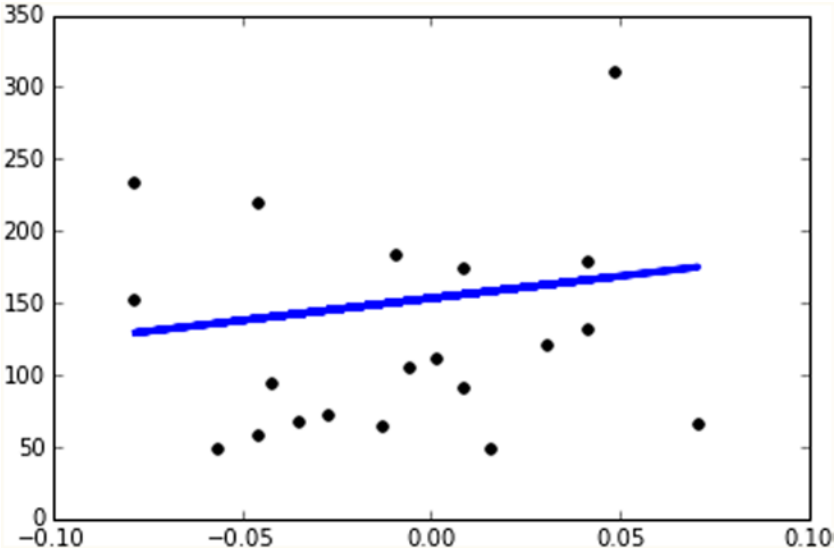

Figure 8-7. *A linear regression represents a linear correlation between a feature and the targets*

Actually, you have 10 physiological factors within the diabetes dataset. Therefore, to have a more complete picture of all the training set, you can make a linear regression for every physiological feature, creating 10 models and seeing the result for each of them through a linear chart.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn import datasets
diabetes = datasets.load_diabetes()
x_train = diabetes.data[:-20]
y_train = diabetes.target[:-20]
x_test = diabetes.data[-20:]
y_test = diabetes.target[-20:]
plt.figure(figsize=(8,12))
for f in range(0,10):
   xi_test = x_test[:,f]
   xi_train = x_train[:,f]
   xi_test = xi_test[:,np.newaxis]
   xi_train = xi_train[:,np.newaxis]
   linreg.fit(xi_train,y_train)
   y = linreg.predict(xi_test)
   plt.subplot(5,2,f+1)
   plt.scatter(xi_test,y_test,color='k')
   plt.plot(xi_test,y,color='b',linewidth=3)
```

Figure 8-8 shows ten linear charts, each of which represents the correlation between a physiological factor and the progression of diabetes.

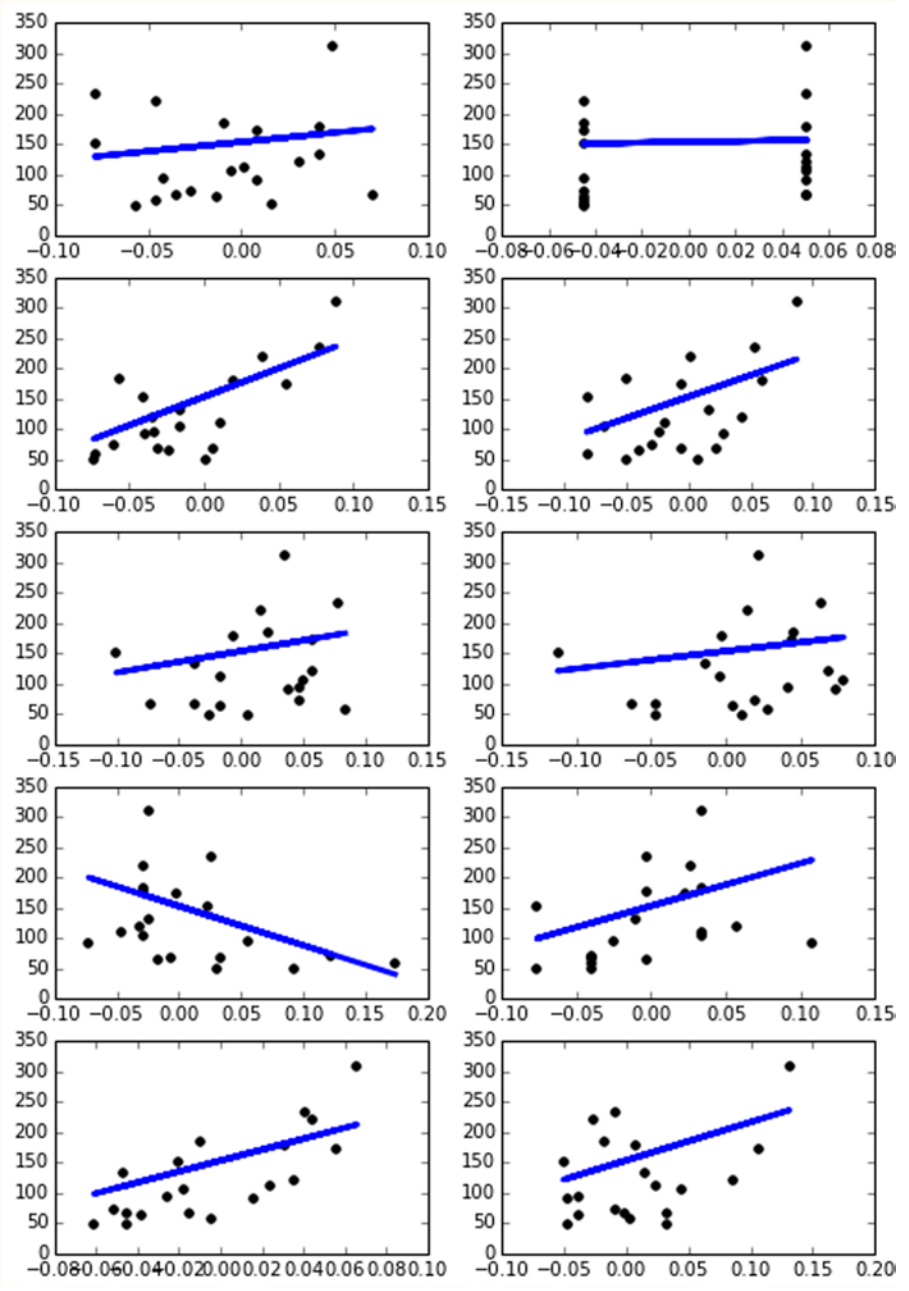

Figure 8-8. *Ten linear charts showing the correlations between physiological factors and the progression of diabetes*

Support Vector Machines (SVMs)

Support Vector Machines are a number of machine learning techniques that were first developed in the AT&T laboratories by Vapnik and colleagues in the early 90s. The basis of this class of procedures is in fact an algorithm called **Support Vector**, which is a generalization of a previous algorithm called Generalized Portrait, developed in Russia in 1963 by Vapnik as well.

In simple words the SVM classifiers are binary or discriminating models, working on two classes of differentiation. Their main task is basically to discriminate against new observations between two classes. During the learning phase, these classifiers project the observations in a multidimensional space called **decisional space** and build a separation surface called **decision boundary** that divides this space into two areas of belonging. In the simplest case, that is, the linear case, the decision boundary will be represented by a plane (in 3D) or by a straight line (in 2D). In more complex cases the separation surfaces are curved shapes with increasingly articulated shapes.

The SVM can be used both in regression with the **SVR (Support Vector Regression)** and in classification with the **SVC (Support Vector Classification)**.

Support Vector Classification (SVC)

If you wish to better understand how this algorithm works, you can start by referring to the simplest case, that is the linear 2D case, where the decision boundary will be a straight line separating into two parts the decisional area. Take for example a simple training set where some points are assigned to two different classes. The training set will consist of 11 points (observations) with two different attributes that will have values between 0 and 4. These values will be contained within a NumPy array called *x*. Their belonging to one of two classes will be defined by 0 or 1 values contained in another array called *y*.

Visualize distribution of these points in space with a scatterplot which will then be defined as decision space (see Figure 8-9).

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
     [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
Out[360]: <matplotlib.collections.PathCollection at 0x545634a8>
```

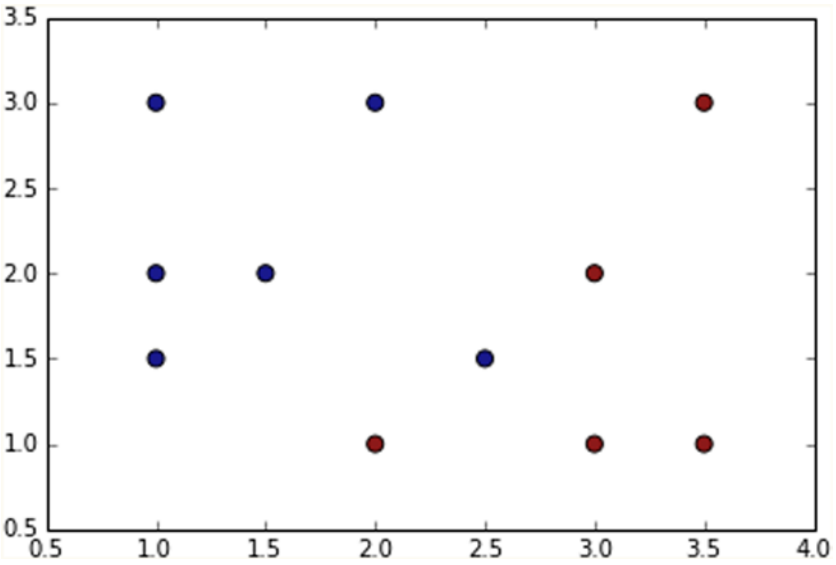

Figure 8-9. *The scatterplot of the training set displays the decision space*

Now that you have defined the training set, you can apply the SVC (Support Vector Classification) algorithm. This algorithm will create a line (decision boundary) in order to divide the decision area into two parts (see Figure 8-10), and this straight line will be placed so as to maximize its distance of closest observations contained in the training set. This condition should produce two different portions in which all points of a same class should be contained.

Then you apply the SVC algorithm to the training set and to do so first you define the model with the **SVC()** constructor defining the kernel as linear. (A kernel is a class of algorithms for pattern analysis.) Then you will use the **fit()** function with the training set as argument. Once the model is trained you can plot the decision boundary with the **decision_function()** function. Then you draw the scatterplot giving a different color to the two portions of the decision space.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
     [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='linear').fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k'], linestyles=['-'],levels=[0])
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)

Out[363]: <matplotlib.collections.PathCollection at 0x54acae10>
```

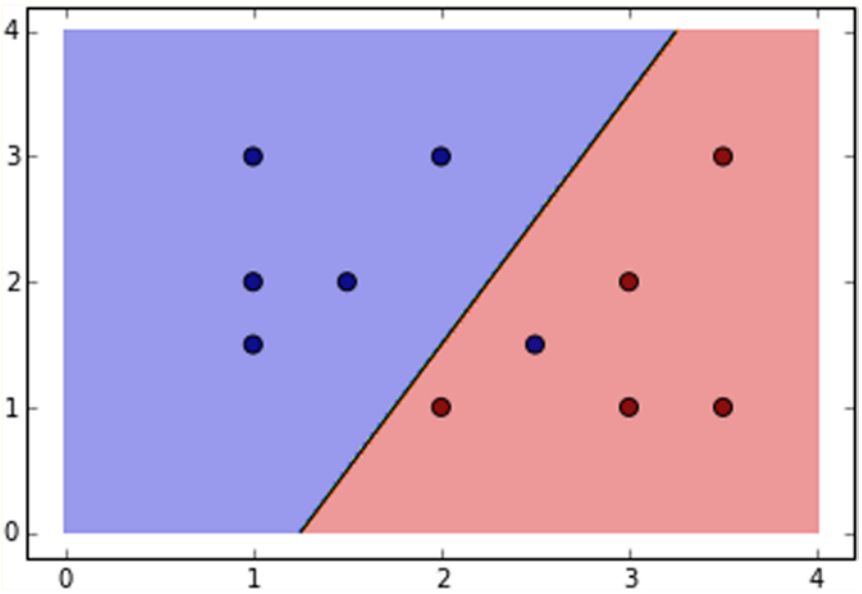

Figure 8-10. *The decision area is split into two portions*

In Figure 8-10 you can see the two portions containing the two classes. It can be said that the division is successful except for a blue dot in the red portion.

So once the model has been trained, it is simple to understand how the predictions operate. Graphically, depending on the position occupied by the new observation, you will know its corresponding membership in one of the two classes.

Instead, from a more programmatic point of view, the **predict()** function will return the number of the corresponding class of belonging (0 for class in blue, 1 for the class in red).

```
svc.predict([1.5,2.5])
Out[56]: array([0])

svc.predict([2.5,1])
Out[57]: array([1])
```

A related concept with the SVC algorithm is **regularization**. It is set by the parameter **C**: a small value of C means that the margin is calculated using many or all of the observations around the line of separation (greater regularization), while a large value of C means that the margin is calculated on the observations near to the line separation (lower regularization). Unless otherwise specified, the default value of C is equal to 1. You can highlight points that participated in the margin calculation, identifying them through the **support_vectors** array.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
     [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='linear',C=1).fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
```

```
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k','k','k'], linestyles=['--','-','--'],levels=[-1,0,1])
plt.scatter(svc.support_vectors_[:,0],svc.support_vectors_[:,1],s=120,facecolors='none')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
```

```
Out[23]: <matplotlib.collections.PathCollection at 0x177066a0>
```

These points are represented by rimmed circles in the scatterplot. Furthermore, they will be within an evaluation area in the vicinity of the separation line (see the dashed lines in Figure 8-11).

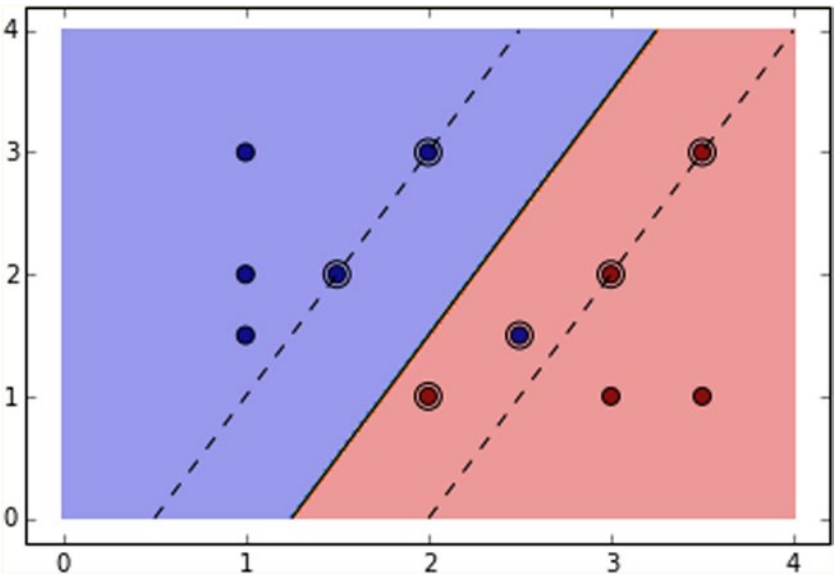

Figure 8-11. *The number of points involved in the calculation depends on the C parameter*

To see the effect on the decision boundary, you can restrict the value to C = 0.1. Let's see how many points will be taken into consideration.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
     [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='linear',C=0.1).fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k','k','k'], linestyles=['--','-','--'],levels=[-1,0,1])
plt.scatter(svc.support_vectors_[:,0],svc.support_vectors_[:,1],s=120,facecolors='none')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
```

```
Out[24]: <matplotlib.collections.PathCollection at 0x1a01ecc0>
```

The points taken into consideration are increased and consequently the separation line (decision boundary) has changed orientation. But now there are two points that are in the wrong decision areas (see Figure 8-12).

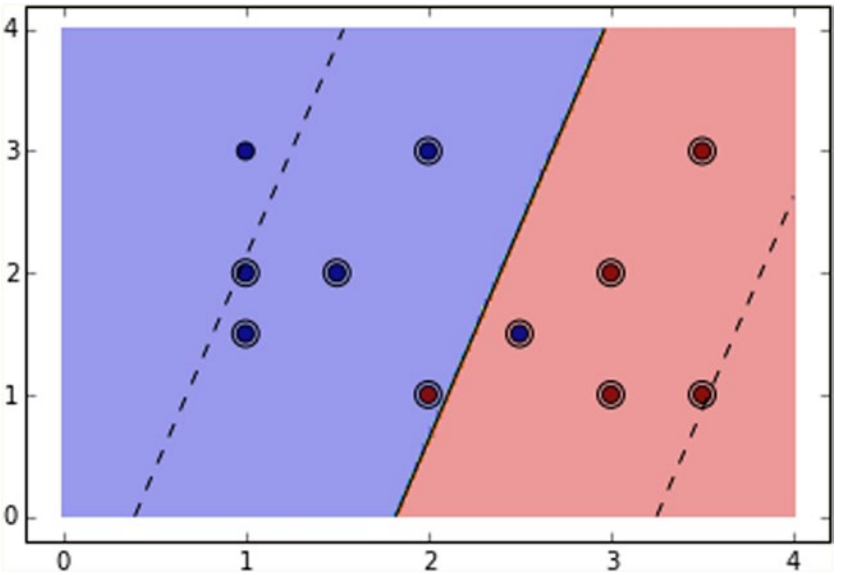

Figure 8-12. *The number of points involved in the calculation grows with decreasing of C*

Nonlinear SVC

So far you have seen the SVC linear algorithm defining a line of separation that was intended to split the two classes. There are also more complex SVC algorithms able to establish curves (2D) or curved surfaces (3D) based on the same principles of maximizing the distances between the points closest to the surface. Let's see the system using a polynomial kernel.

As the name implies, you can define a polynomial curve that separates the area decision in two portions. The degree of the polynomial can be defined by the **degree** option. Even in this case C is the coefficient of regularization. So try to apply an SVC algorithm with a polynomial kernel of third degree and with a C coefficient equal to 1.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
     [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='poly',C=1, degree=3).fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k','k','k'], linestyles=['--','-','--'],levels=[-1,0,1])
```

```
plt.scatter(svc.support_vectors_[:,0],svc.support_vectors_[:,1],s=120,facecolors='none')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)

Out[34]: <matplotlib.collections.PathCollection at 0x1b6a9198>
```

As you can see, you get the situation shown in Figure 8-13.

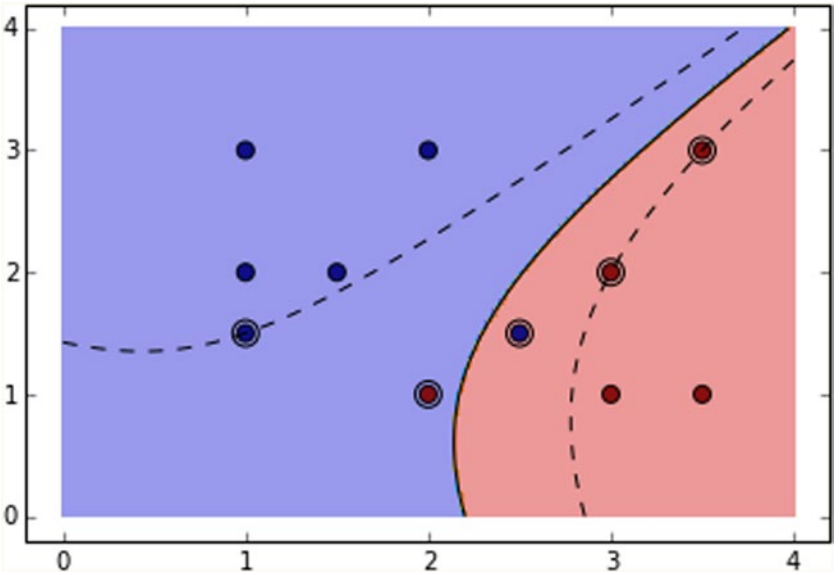

Figure 8-13. *The decision space using a SCV with a polynomial kernel*

Another type of nonlinear kernel is the **Radial Basis Function (RBF)**. In this case the separation curves tend to define the zones radially with respect to the observation points of the training set.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
     [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='rbf', C=1, gamma=3).fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k','k','k'], linestyles=['--','-','--'],levels=[-1,0,1])
plt.scatter(svc.support_vectors_[:,0],svc.support_vectors_[:,1],s=120,facecolors='none')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)

Out[43]: <matplotlib.collections.PathCollection at 0x1cb8d550>
```

In Figure 8-14 we can see the two portions of the decision with all points of the training set correctly positioned.

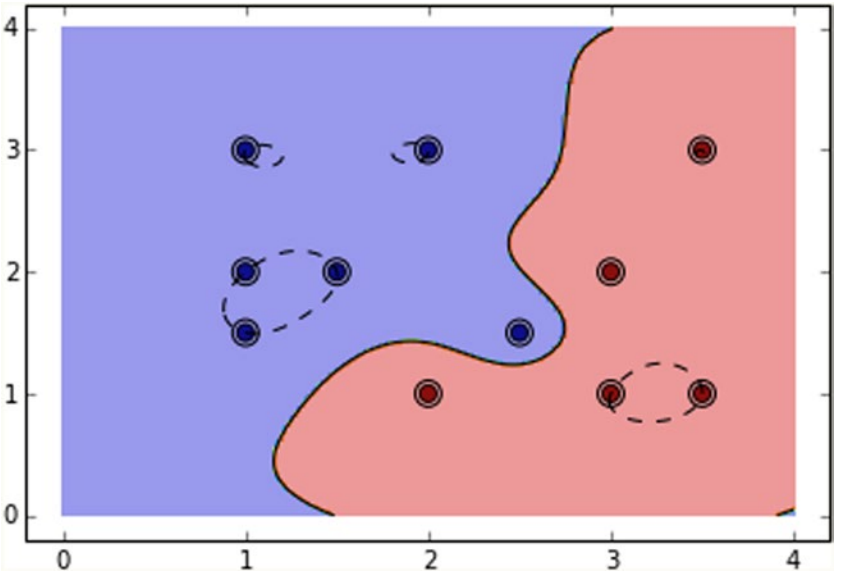

Figure 8-14. *The decisional area using a SVC with rgb kernel*

Plotting Different SVM Classifiers Using the Iris Dataset

The SVM example that you just saw is based on a very simple dataset. Use a more complex datasets for a classification problem with SVC. Use the previously used dataset: the Iris Dataset.

The SVC algorithm used before learned from a training set containing only two classes. In this case you will extend the case to three classifications, as three are the classes of the Iris Dataset is split, corresponding to the three different species of flowers.

In this case the decision boundaries intersect each other subdividing the decision area (in the case 2D) or the decision volume (3D) in several portions.

Both linear models have linear decision boundaries (intersecting hyperplanes) while models with nonlinear kernels (polynomial or Gaussian RBF) have nonlinear decision boundaries more flexible with figures that are dependent on the type of kernel and its parameters.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

iris = datasets.load_iris()
x = iris.data[:,:2]
y = iris.target
h = .05
svc = svm.SVC(kernel='linear',C=1.0).fit(x,y)
x_min,x_max = x[:,0].min() - .5, x[:,0].max() + .5
y_min,y_max = x[:,1].min() - .5, x[:,1].max() + .5
```

```
h = .02
X, Y = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min,y_max,h))

Z = svc.predict(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z,alpha=0.4)
plt.contour(X,Y,Z,colors='k')
plt.scatter(x[:,0],x[:,1],c=y)

Out[49]: <matplotlib.collections.PathCollection at 0x1f2bd828>
```

In Figure 8-15, the decision space is divided into three portions separated by decisional boundaries.

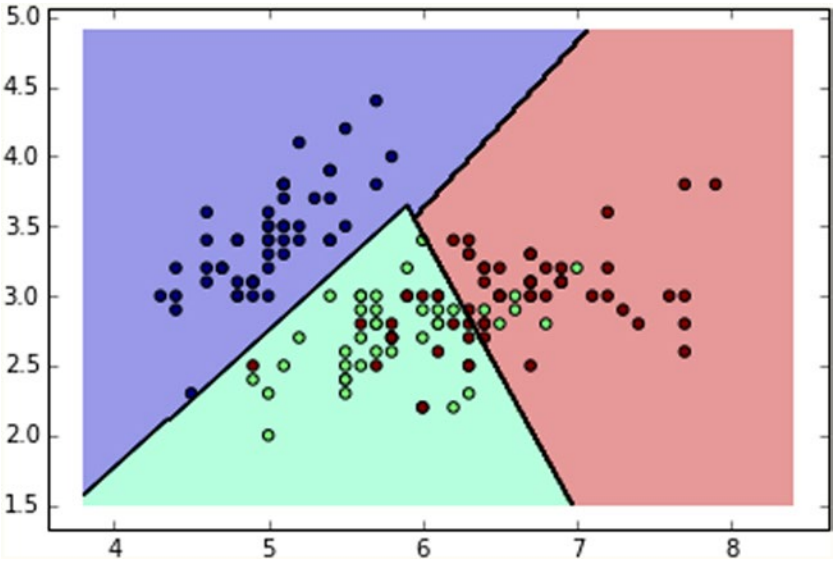

Figure 8-15. *The decisional boundaries split the decisional area into three different portions*

Now it's time to apply a nonlinear kernel for generating nonlinear decision boundaries, such as the polynomial kernel.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

iris = datasets.load_iris()
x = iris.data[:,:2]
y = iris.target
h = .05
svc = svm.SVC(kernel='poly',C=1.0,degree=3).fit(x,y)
x_min,x_max = x[:,0].min() - .5, x[:,0].max() + .5
y_min,y_max = x[:,1].min() - .5, x[:,1].max() + .5

h = .02
X, Y = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min,y_max,h))
```

```
Z = svc.predict(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z,alpha=0.4)
plt.contour(X,Y,Z,colors='k')
plt.scatter(x[:,0],x[:,1],c=y)

Out[50]: <matplotlib.collections.PathCollection at 0x1f4cc4e0>
```

Figure 8-16 shows how the polynomial decision boundaries split the area in a very different way compared to the linear case.

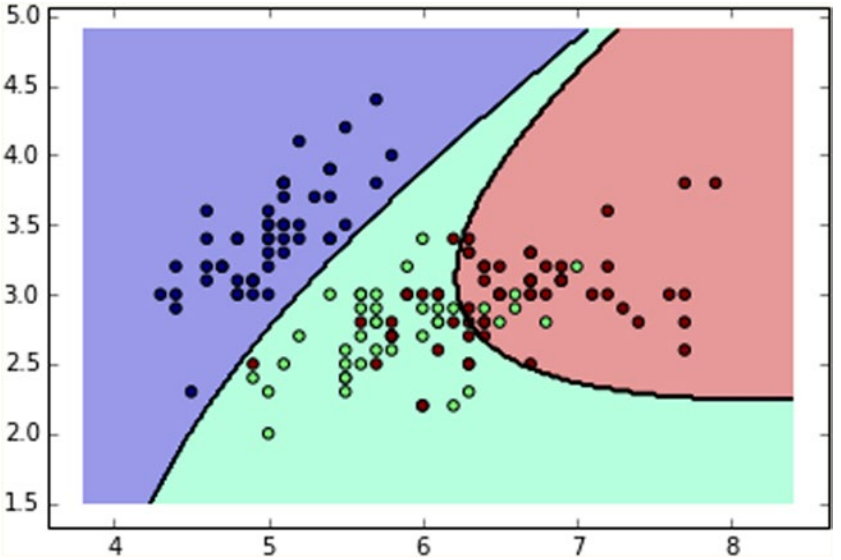

Figure 8-16. *In the polynomial case the blue portion is not directly connected with the red portion*

Now you can apply the RBF kernel to see the difference in the distribution of areas.

```
svc = svm.SVC(kernel='rbf', gamma=3, C=1.0).fit(x,y)
```

Figure 8-17 shows how the RBF kernel generates radial areas.

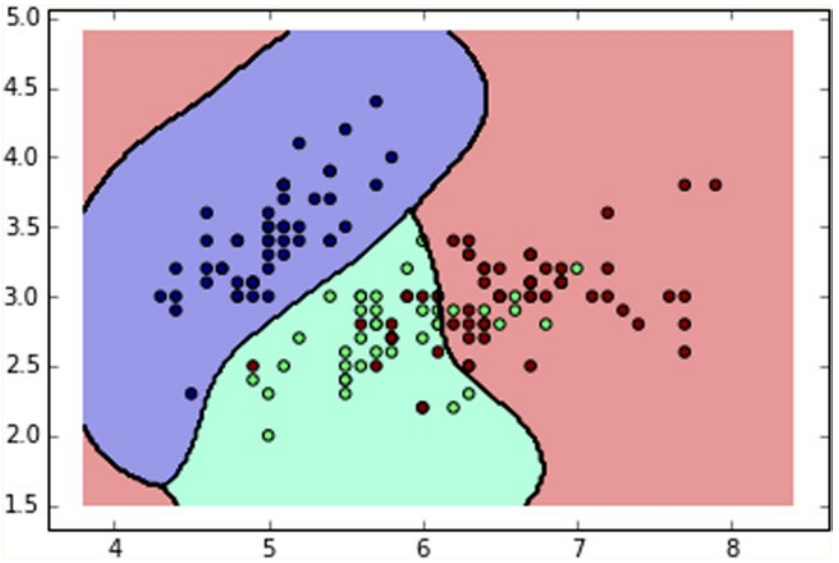

Figure 8-17. *The kernel RBF defines radial decisional areas*

Support Vector Regression (SVR)

The SVC method can be extended to solve even regression problems. This method is called **Support Vector Regression**.

The model produced by SVC actually does not depend on the complete training set, but uses only a subset of elements, i.e., those closest to the decisional boundary. In a similar way, the model produced by SVR also depends only on a subset of the training set.

We will demonstrate how the SVR algorithm will use the diabetes dataset that you have already seen in this chapter. By way of example you will refer only to the third physiological data. We will perform three different regressions, a linear and two nonlinear (polynomial). The linear case will produce a straight line as linear predictive model very similar to the linear regression seen previously, whereas polynomial regressions will be built of the second and third degrees. The SVR() function is almost identical to the SVC()function seen previously.

The only aspect to consider is that the test set of data must be sorted in ascending order.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn import datasets
diabetes = datasets.load_diabetes()
x_train = diabetes.data[:-20]
y_train = diabetes.target[:-20]
x_test = diabetes.data[-20:]
y_test = diabetes.target[-20:]

x0_test = x_test[:,2]
x0_train = x_train[:,2]
x0_test = x0_test[:,np.newaxis]
x0_train = x0_train[:,np.newaxis]
```

```
x0_test.sort(axis=0)
x0_test = x0_test*100
x0_train = x0_train*100

svr = svm.SVR(kernel='linear',C=1000)
svr2 = svm.SVR(kernel='poly',C=1000,degree=2)
svr3 = svm.SVR(kernel='poly',C=1000,degree=3)
svr.fit(x0_train,y_train)
svr2.fit(x0_train,y_train)
svr3.fit(x0_train,y_train)
y = svr.predict(x0_test)
y2 = svr2.predict(x0_test)
y3 = svr3.predict(x0_test)
plt.scatter(x0_test,y_test,color='k')
plt.plot(x0_test,y,color='b')
plt.plot(x0_test,y2,c='r')
plt.plot(x0_test,y3,c='g')

Out[155]: [<matplotlib.lines.Line2D at 0x262e10b8>]
```

The three regression curves will be represented with three different colors. The linear regression will be blue; the polynomial of second degree that is, a parabola, will be red; and the polynomial of third degree will be green (see Figure 8-18).

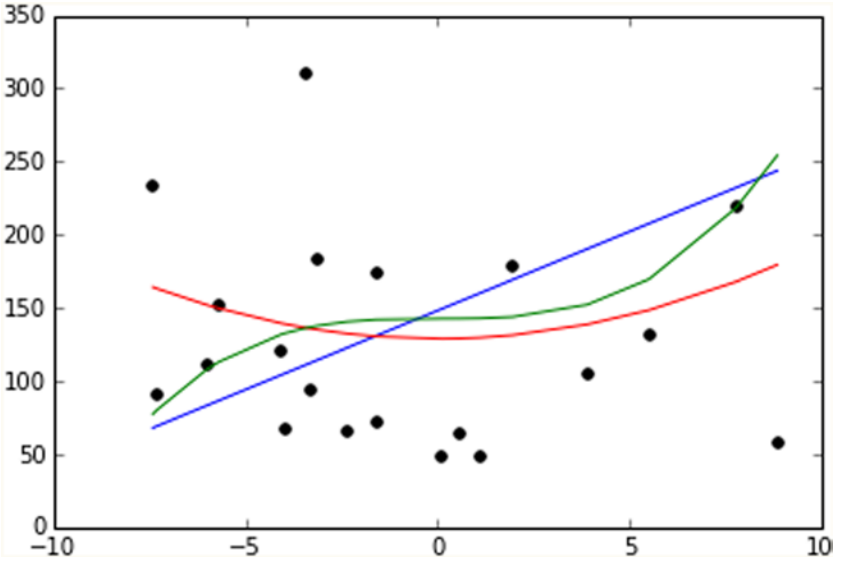

Figure 8-18. *The three regression curves produce very different trends starting from the training set*

Conclusions

In this chapter you saw the simplest cases of regression and classification problems solved using the scikit-learn library. Many concepts of the validation phase for a predictive model were presented in a practical way through some practical examples.

In the next chapter you will see a complete case in which all steps of data analysis are discussed through a single practical example. Everything will be implemented on IPython Notebook, an interactive and innovative environment well suited for sharing every step of the data analysis with the form of an interactive documentation useful as a report or as a web presentation.