CHAPTER 8

Machine Learning with scikit-learn

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

The scikit-learn Library

`scikit-learn` is a Python module that integrates many 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, which is 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 datasets 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 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 to apply is called a *learning problem*.

The data to be subjected to a pattern in the learning phase can be arrays composed of a single value per element, or of 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 learning problems into two broad categories—supervised and unsupervised.

F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-9532-8_8

Supervised Learning

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

- *Classification*—The data in the training set belong to two or more classes or categories; then, the data, already being labeled, allow you to teach the system to recognize the characteristics that distinguish each class. When you need to consider a new value unknown to the system, the system can 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 that 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 another group of methods that serves to validate and evaluate the models.

Training Set and Testing Set

Machine learning enables the system to learn properties of a model from a dataset and apply these properties 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*, which is used to 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 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 does the training. `x` comprises the features observed, while `y` indicates the target. Once the estimator has carried out the training, it can predict the value of `y` for any new observation `x` not labeled. This operation is completed using 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 the Anderson Iris Dataset, after the person who collected the data by measuring the size of the various parts of the iris. In this dataset, data from three different species of iris (Iris silky, virginica Iris, and Iris versicolor) are collected. These data correspond to the length and width of the sepals and the length and width of the petals (see Figure 8-1).

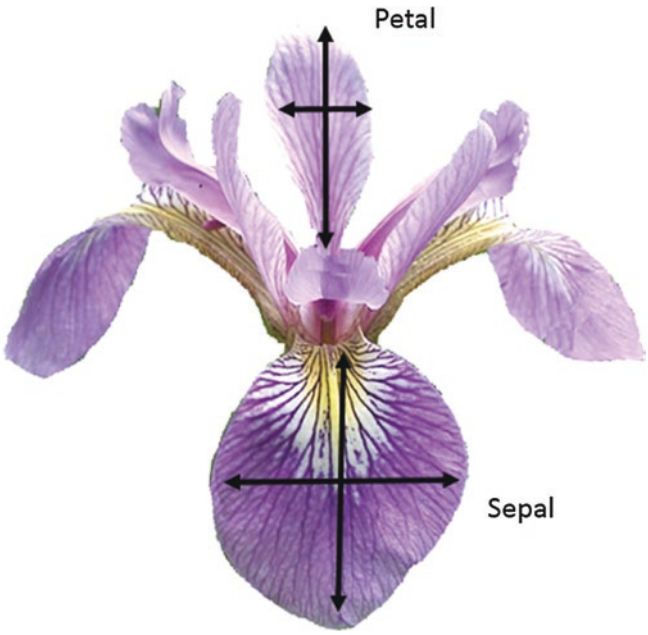

Figure 8-1. *Iris versicolor and the petal and sepal width and length*

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

Now you will study this dataset in detail, importing it into a Jupyter Notebook or into a normal Python session.

```
from sklearn import datasets
iris = datasets.load_iris()
```

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

```
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 get an array of 150 elements, each containing four numeric values: the size of sepals and petals respectively.

To determine instead what kind of flower belongs to each item, refer to the `target` attribute.

```
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 determine the correspondence between the species and number, you have to call the `target_names` attribute.

```
iris.target_names
Out[ ]:
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

To better understand this dataset, 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. The x-axis will represent the length of the sepal, while the y-axis will represent the width of the sepal.

```
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 WITH MATPLOTLIB
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(())
plt.show()
```

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

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.

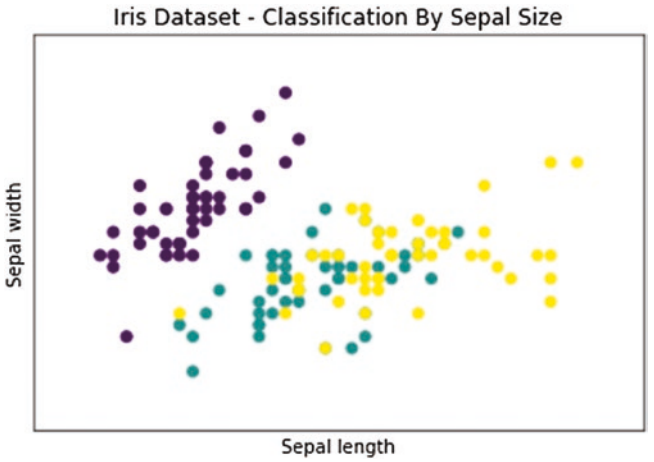

Figure 8-2. *The different species of irises are shown with different colors*

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 and change just a few values.

```
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.

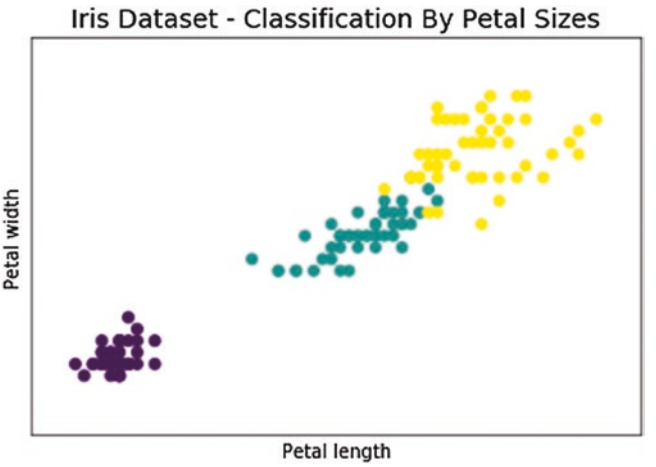

Figure 8-3. *The different species of irises are shown with different colors*

The PCA Decomposition

You have seen how the three species can be characterized, taking into account four measurements of the petal and sepal sizes. It represented two scatterplots, one for the petals and one for sepals, but how can you unify the whole thing? Four dimensions are a problem that even a 3D scatterplot cannot 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 are called *principal components*.

Hence, PCA is employed before applying the machine learning algorithm, as it minimizes the number of variables used. It does this by analyzing the contribution of each of them to the maximum amount of variance in the dataset. It determines which of these features do not contribute significantly (they provide the same information as other features) and discards them, eliminating them from the calculation of the machine learning model. This greatly lightens the model and the calculations related to it.

This technique compresses the number of features involved in the machine learning calculation, creating new ones (or rather transforming the existing ones) in a smaller number, but with the same information as the starting dataset (approximation).

From a strictly mathematical point of view, from the input data matrix `X` (`num_observations` x `num_features`), the covariance matrix `Cov(X)` is obtained, which is then used to calculate the eigenvectors (principal components) and their corresponding eigenvalues. The k eigenvectors obtained are the principal components (see Figure 8-4). The first k principal components are the eigenvectors corresponding to the k largest eigenvalues.

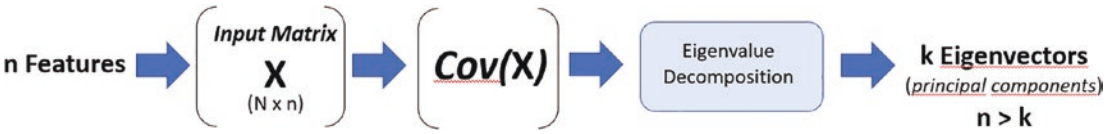

Figure 8-4. *The processes involved in the PCA technique*

Let's put this into practice using the Iris Dataset. You saw in the previous section that this dataset is made up of four features. You can apply the PCA method to analyze this technique in detail. To see if it is possible to reduce this number but still keep as much of the information as possible for the machine learning model, you write the following code:

```
from sklearn.decomposition import PCA
import numpy as np

covar_matrix = PCA(n_components=4)
covar_matrix.fit(iris.data)
variance = covar_matrix.explained_variance_ratio_
var = np.cumsum(np.round(variance, decimals=3)*100)
var
Out[ ]:
array([ 92.5, 97.8, 99.5, 100. ])
```

This code first imports the `PCA()` function from the `sklearn.decomposition` module and uses it to derive the covariance matrix. In this case, the technique considers all four features in the calculation (therefore, n = k) by imposing the `n_components` parameter equal to 4. Then it calculates the eigenvalue decomposition on this matrix using the `fit()` method. At this point, you can evaluate the contribution of each of the four features to the maximum value of the dataset variance. You extract the variance and then, with the NumPy function `cumsum()`, you obtain the cumulative contribution as a percentage made by each added feature. From the result, you can see that even if you reduced the size of the input dataset to just one feature, you would still retain 92 percent of the information. It can therefore be said that the information of a single feature could already be almost sufficient to obtain a good machine learning model. You can see it all graphically through `matplotlib`.

```
plt.ylabel('% Variance explained')
plt.xlabel('# of Features')
plt.title('PCA Analysis')
plt.ylim(90, 100.5)
plt.xticks([0, 1, 2, 3], [1,2,3,4])
plt.axvline(2, linestyle='--', c='#bbbbbb' )
plt.plot(var)
```

Running this code produces a plot like the one shown in Figure 8-5.

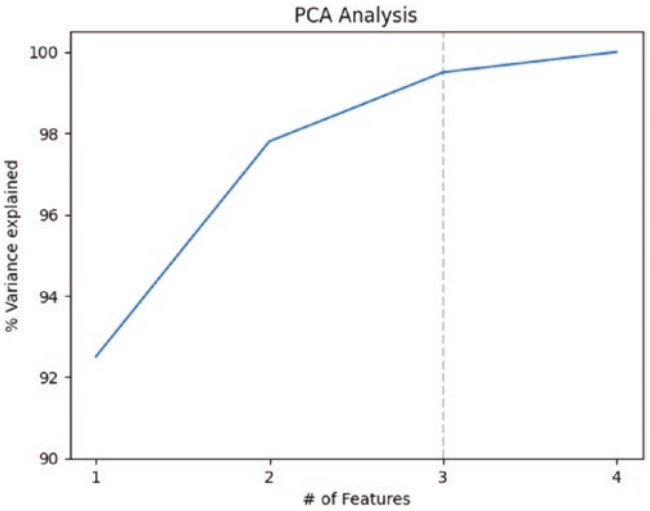

Figure 8-5. *The contribution of each feature to the total variance*

As you can clearly see from Figure 8-5, the use of three features is more than enough to build a good model, with a value of 99.5 percent. The use of a fourth feature is almost useless for this purpose, and it can be eliminated, thus simplifying the calculation.

In this case, you can reduce the system from four to three dimensions and plot the results in a 3D scatterplot.

The `scikit-learn` function that allows you to do the dimensional reduction is the `fit_transform()` function. It belongs to the `PCA` object. Apply the functions and methods you just used again.

```
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 3D scatterplot using the `mpl_toolkits.mplot3d` module of `matplotlib`. If you don't remember how to do this, see the section called "Scatter Plots in 3D" 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()
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 the scatterplot shown in Figure 8-6. The three species of iris are well characterized with respect to each other to form a cluster.

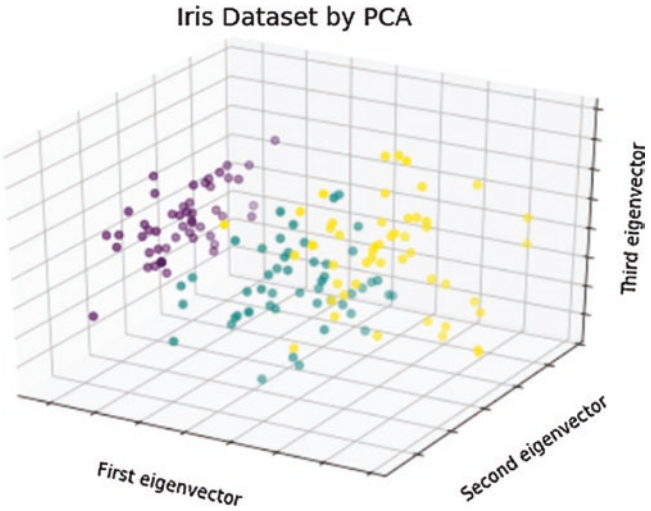

Figure 8-6. *3D scatterplot with three clusters representative of each species of iris*

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 determine to which of the three species it belongs. The simplest possible classifier is the *nearest neighbor.* This algorithm searches within the training set for the observation that most closely approaches the new test sample.

In fact, the mechanism behind the KNN algorithm is one of the simplest and most direct in machine learning. Despite this, KNN remains one of the most used techniques in this discipline and not only for classification, but also for other problems such as regression and outlier detection.

The algorithm first calculates the distance between the new data point to be classified in the (multidimensional) feature space and all the other data points whose class it belongs to (or has been previously evaluated). To calculate the distance between two points you can use different techniques such as Euclidean (the one we all study at school), Minkowski, Manhattan, and so on. The Euclidean method would appear to be the best, but in reality for a large number of dimensions, this is no longer so valid.

Once the distance has been calculated, the KNN algorithm selects, among all the points, the K points closest to the one to be classified. This K number is what gives the "K-Nearest Neighbors" algorithm its name. In a classification problem, the new point will be awarded to the class that has the most points among the K-nearest neighbors. See Figure 8-7.

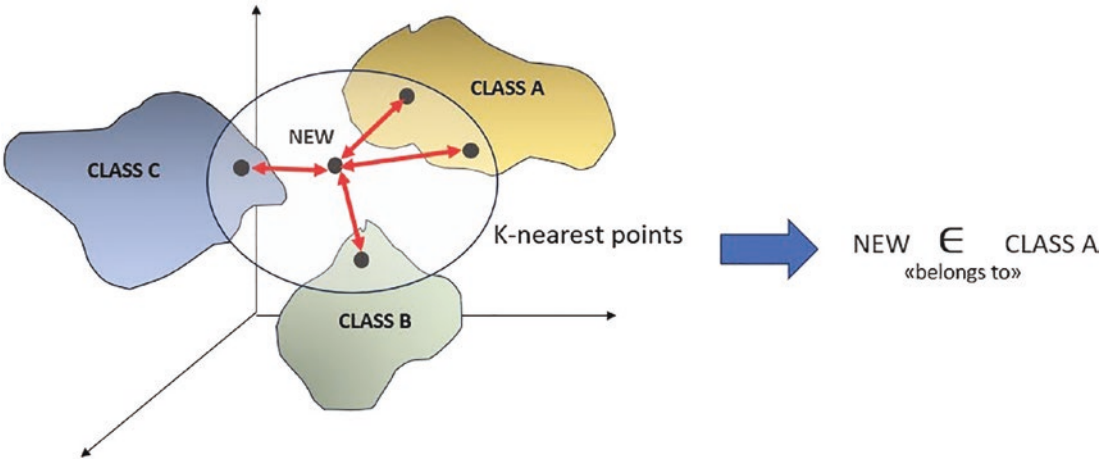

Figure 8-7. *The KNN mechanism of classification*

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, 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. To blend elements of the dataset, you use a NumPy function called `random.permutation()`. The mixed dataset consists of 150 different observations; the first 140 are used as the training set, the remaining 10 as the 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)
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 that 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 ten 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 compare the results predicted with the actual observed results contained in `y_test`.

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

You can see that you obtained a 10 percent error. 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, shading='auto')
#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[ ]: (1.5, 4.900000000000003)
```

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

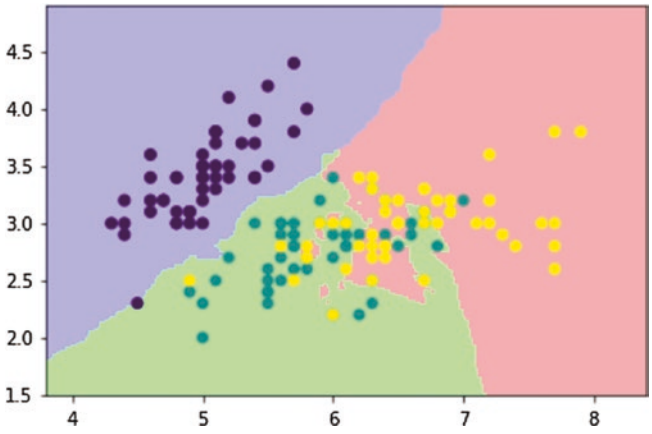

Figure 8-8. *The three decision boundaries are represented by three different colors*

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, shading='auto')
#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[ ]: (-0.40000000000000002, 2.9800000000000031)
```

As shown in Figure 8-9, you have the corresponding decision boundaries regarding the characterization of iris flowers taking into account the size of the petals.

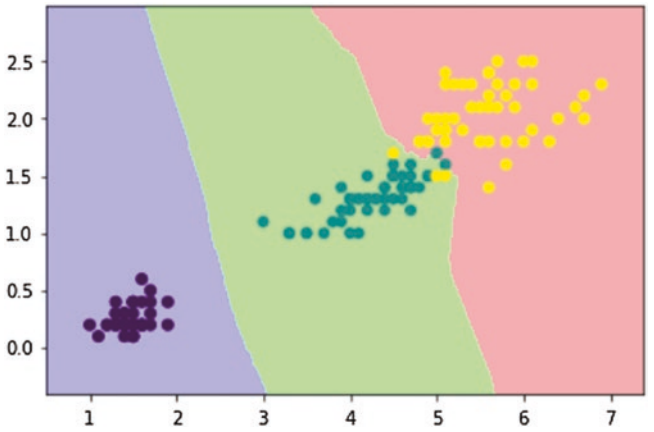

Figure 8-9. *The decision boundaries on a 2D scatterplot describing the petal sizes*

Diabetes Dataset

Among the various datasets available in 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, Johnstone, 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, you must first import the `datasets` module of the `scikit-learn` library and then call the `load_diabetes()` function to load the dataset into a variable called `diabetes`.

```
from sklearn import datasets
 diabetes = datasets.load_diabetes()
```

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

- Age
- Sex
- Body mass index
- Blood pressure
- S1, S2, S3, S4, S5, and S6 (six blood serum measurements)

These measurements can be obtained by calling the `data` attribute. If you check the values in the dataset, you will find values very different from what you would have expected. For example, look at the ten 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 processing. Each of the ten values was mean-centered and subsequently scaled by the standard deviation times the number of samples. Checking will reveal that the sum of the 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[ ]: 1.0000000000000746
```

Even though these values are normalized and therefore difficult to read, they continue to express the ten 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[ ]:
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 simplest 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 the 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 the `linear_model` module and then use the manufacturer `LinearRegression()` constructor to create the predictive model, which you call `linreg`.

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

To practice with an example of linear regression, you can use the diabetes dataset described earlier. First you 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 the `fit()` function.

```
linreg.fit(x_train,y_train)
Out[ ]: LinearRegression()
```

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[ ]:
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 which prediction should be perfect is the *variance*. The closer the variance is to 1, the more perfect the prediction.

```
linreg.score(x_test, y_test)
Out[ ]: 0.58507530226905713
```

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

Figure 8-10 shows the line representing the linear correlation between the ages of the patients and the disease progression.

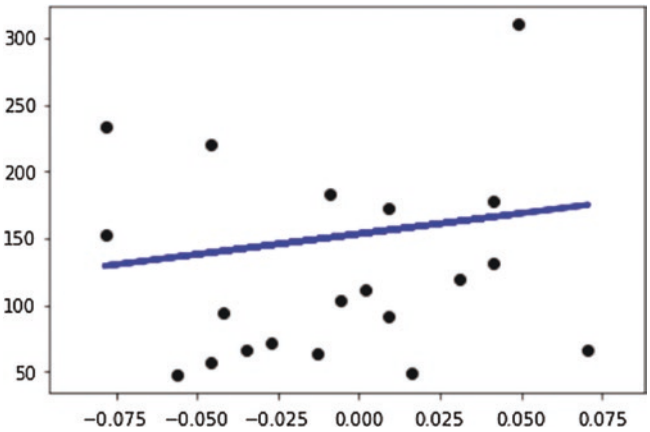

Figure 8-10. *A linear regression represents a linear correlation between a feature and the targets*

Actually, you have ten physiological factors in the diabetes dataset. Therefore, to have a more complete picture of all the training set, you can create a linear regression for every physiological feature, creating ten 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 8-11 shows ten linear charts, each of which represents the correlation between a physiological factor and the progression of diabetes.

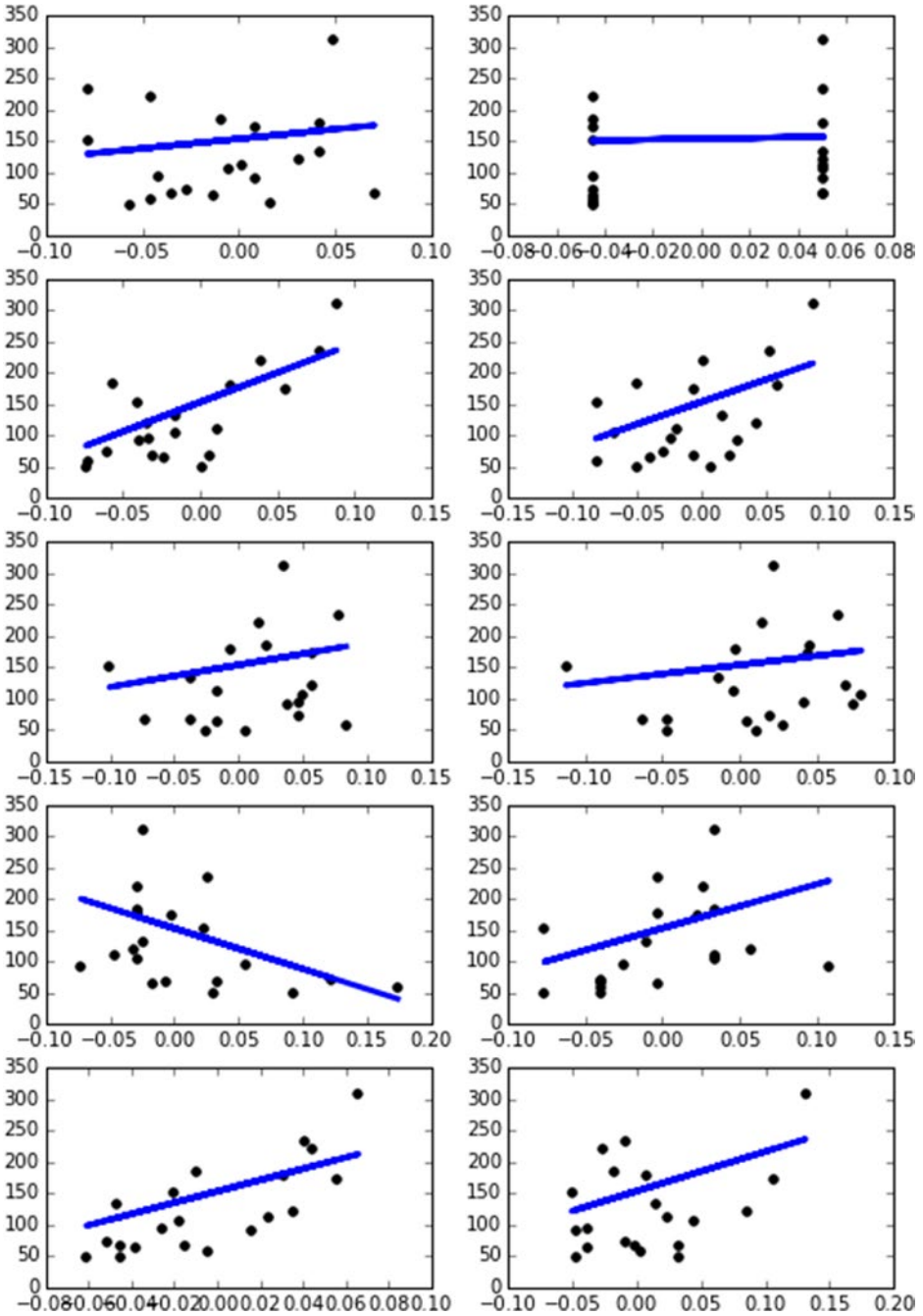

Figure 8-11. *Ten linear charts showing the correlations between physiological factors and the progression of diabetes*

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 *decision space* and build a separation surface called the *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.

Also for SVM, the data points are distributed in a multidimensional space, where the values of a specific feature are distributed on each axis. The purpose of SVM is to find a hyperplane separating the space between the two classes (see Figure 8-12).

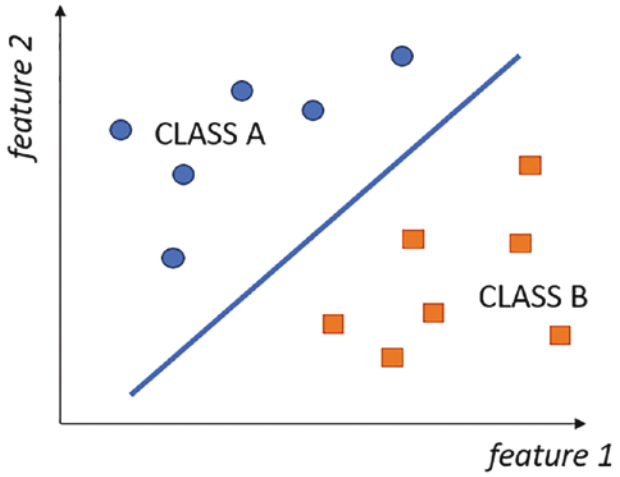

Figure 8-12. *SVM divides the decision space into two areas, each for each class, defining a straight line (decision boundary)*

The straight line that must be defined must not limit itself to separating the points of the two classes from each other, but must respond to particular requirements. SVM finds the points that are closest to the dividing line. These are called "support vectors" and they give the algorithm its name (see Figure 8-13).

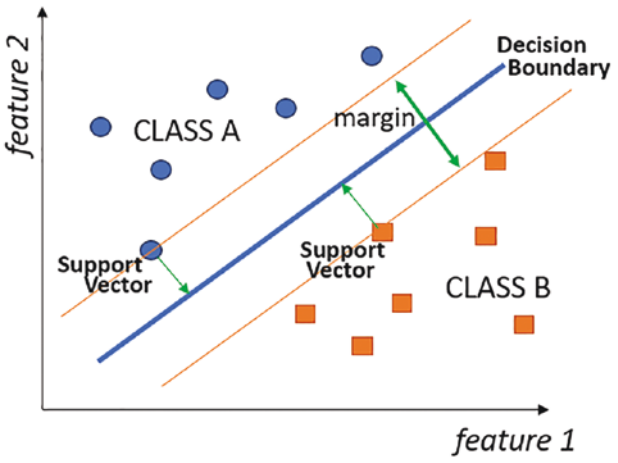

Figure 8-13. *SVM optimizes the decision boundary, maximizing the margin between support vector points*

The distance between the support vectors and the decision boundary (the straight line) is called the margin. SVM's task is to maximize this margin. When the margin reaches its maximum value, then the decision boundary will be the optimal one.

SVM is a very flexible technique that lends itself well to many applications. It 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 want 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 is a straight line separating the decision area into two parts. 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 have values between 0 and 4. These values will be contained within a NumPy array called `x`. Their belonging to one of two classes is 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 a decision space (see Figure 8-14).

```
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)
```

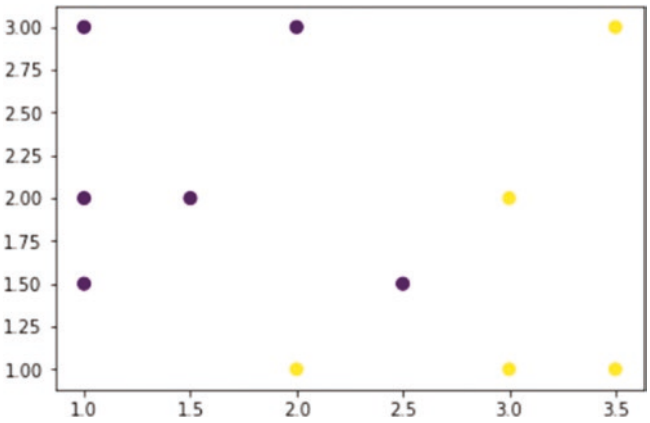

Figure 8-14. *The scatterplot of the training set displays the decision space*

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 8-15), and this straight line will be placed to maximize the 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 use the `fit()` function with the training set as an argument. Once the model is trained, you can plot the decision boundary with the `decision_function()` function. Then you draw the scatterplot and provide 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=0.1)
plt.contour(X,Y,Z,colors=['k'], linestyles=['-'],levels=[0])
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
```

In Figure 8-15, you can see the two portions containing the two classes. It can be said that the division is successful except for a dark dot in the lighter portion.

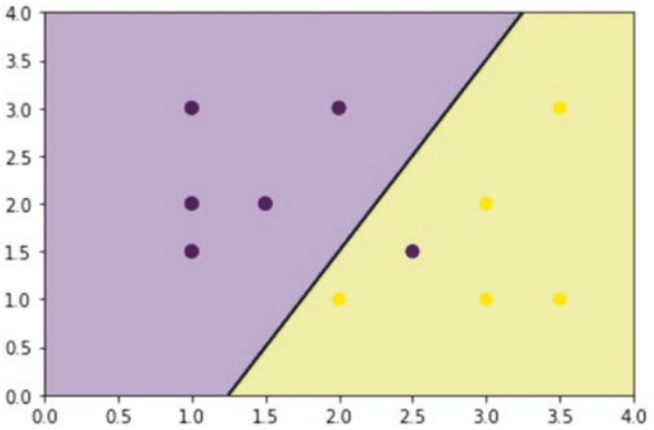

Figure 8-15. *The decision area is split into two portions*

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[ ]: array([0])
svc.predict([[2.5,1]])
Out[ ]: array([1])
```

A related concept in the SVC algorithm is *regularization*. It is set by the `C` parameter and 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=0.1)
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='r')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
```

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 8-16).

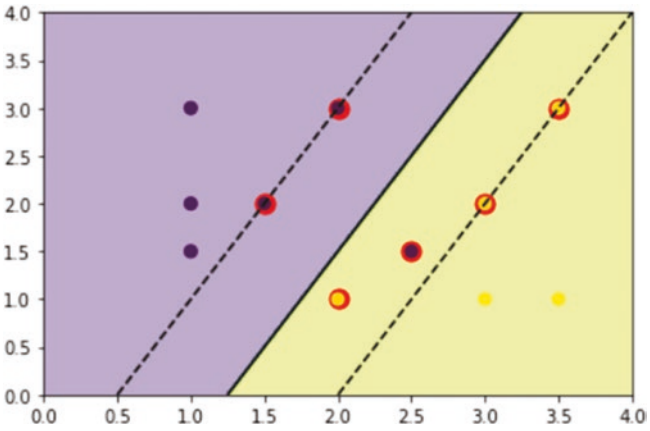

Figure 8-16. *The number of points involved in the calculation depends on the C parameter*

To see the effect on the decision boundary, you can restrict the value to `C = 0.1`. Take a look at 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=0.1)
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='w')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
```

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 8-17).

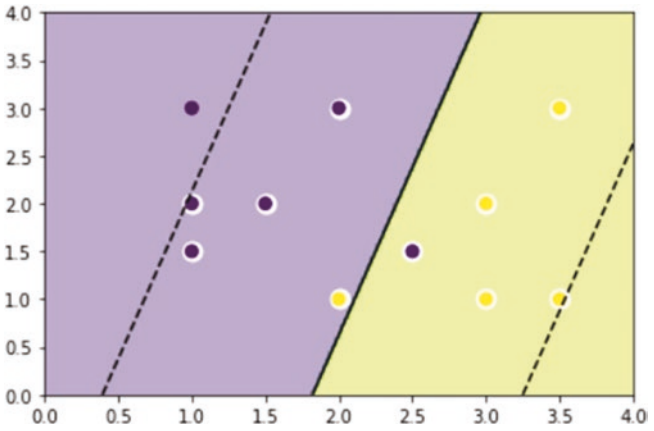

Figure 8-17. *The number of points involved in the calculation grows when C decreases*

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 that can establish curves (2D) or curved surfaces (3D) based on the same principles of maximizing the distances between the points closest to the surface. This section looks at the system using a polynomial kernel.

As the name implies, you can define a polynomial curve that separates the area decision into two portions. The degree of the polynomial can be defined by the `degree` option. Even in this case, `C` is the coefficient of regularization. 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=0.1)
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='w')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
```

You get the situation shown in Figure 8-18.

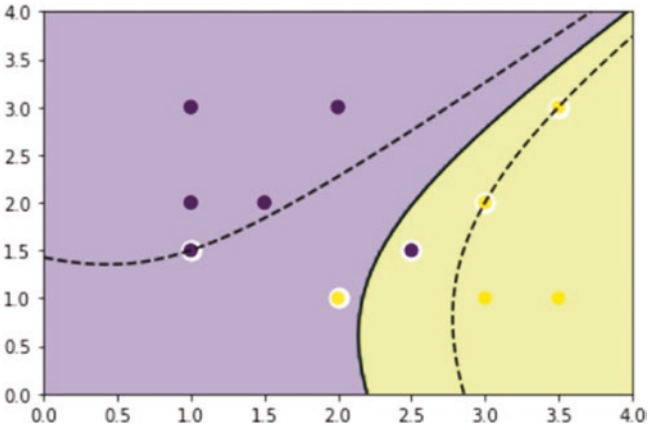

Figure 8-18. *The decision space using an SVC with a polynomial kernel*

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=0.1)
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='w')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
```

In Figure 8-19, you can see the two portions of the decision with all points of the training set correctly positioned.

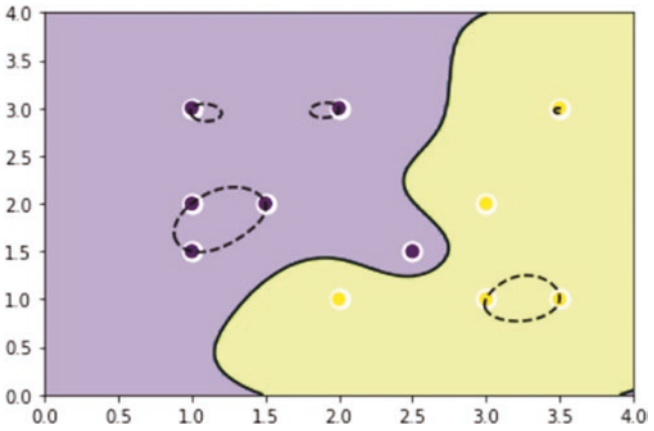

Figure 8-19. *The decision area using an SVC with the RBF kernel*

Plotting Different SVM Classifiers Using the Iris Dataset

The SVM example that you just saw is based on a very simple dataset. This section uses more complex datasets for a classification problem with SVC. In fact, it uses the previously used dataset: the Iris Dataset.

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

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

Both linear models have linear decision boundaries (intersecting hyperplanes), while models with nonlinear kernels (polynomial or Gaussian RBF) have nonlinear decision boundaries. These boundaries are 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=0.1)
plt.contour(X,Y,Z,colors='k')
plt.scatter(x[:,0],x[:,1],c=y)
```

In Figure 8-20, the decision space is divided into three portions separated by decision boundaries.

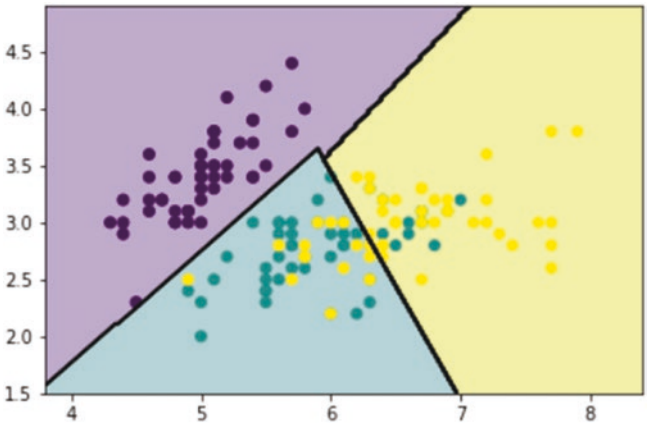

Figure 8-20. *The decision boundaries split the decision area into three different portions*

Now it's time to apply a nonlinear kernel to generate 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=0.1)
plt.contour(X,Y,Z,colors='k')
plt.scatter(x[:,0],x[:,1],c=y)
```

Figure 8-21 shows how the polynomial decision boundaries split the area in a very different way compared to the linear case.

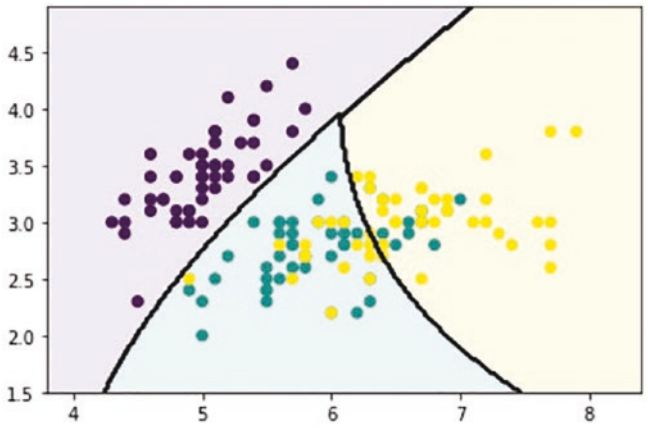

Figure 8-21. *Even in the polynomial case, the three portions remain almost unchanged*

Now you can apply the RBF kernel to see the difference in the distribution of areas. It is sufficient to update the kernel value with `rbf`, inside the `svm.SVC()` function.

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

Figure 8-22 shows how the RBF kernel generates radial areas.

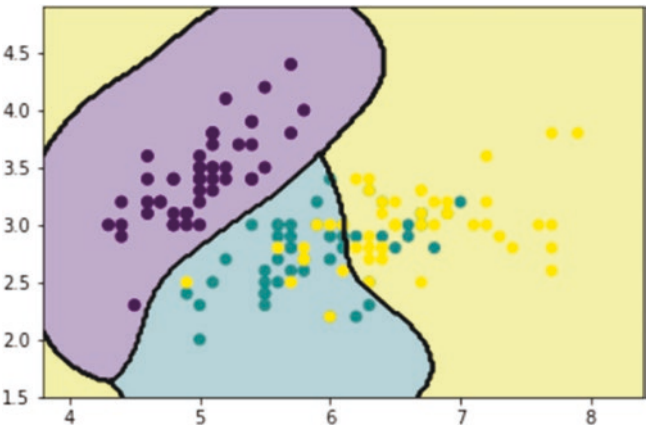

Figure 8-22. *The kernel RBF defines radial decision areas*

Support Vector Regression (SVR)

The SVC method can be extended to solve 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, that is, those closest to the decision boundary. In a similar way, the model produced by SVR also depends only on a subset of the training set.

This section demonstrates how the SVR algorithm uses the diabetes dataset, which you have seen in this chapter. By way of example, it refers only to the third physiological data. It performs three different regressions, one linear and two nonlinear (polynomial). The linear case produces a straight line, as the linear predictive model is very similar to the linear regression seen previously, whereas polynomial regressions are built from 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')
```

The three regression curves are represented with three colors. The linear regression will be blue (the straight line); the polynomial of the second degree (the concave line upwards, i.e. the parabolic line) will be red; and the polynomial of the third degree will be green (see Figure 8-23).

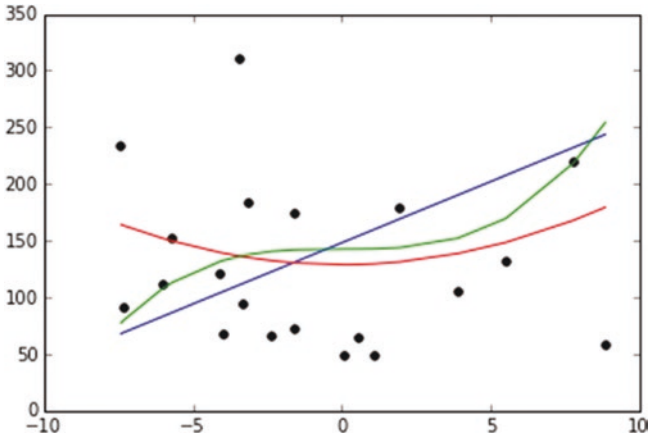

Figure 8-23. *The three regression curves produce very different trends starting from the training set*

Conclusions

In this chapter you saw simple 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 see a complete case in which all steps of data analysis are discussed by way of a single practical example. Everything is implemented in IPython Notebook, an interactive and innovative environment well suited for sharing every step of the data analysis. It includes interactive documentation that's useful as a report or as a web presentation.