CHAPTER

8

Supervised Learning—Classification Using Support Vector Machines

What Is a Support Vector Machine?

In the previous chapter, you saw how to perform classification using logistics regression. In this chapter, you will learn another supervised machine learning algorithm that is also very popular among data scientists—*Support Vector Machines (SVM)*. Like logistics regression, SVM is also a classification algorithm.

The main idea behind SVM is to draw a line between two or more classes in the best possible manner (see Figure 8.1).

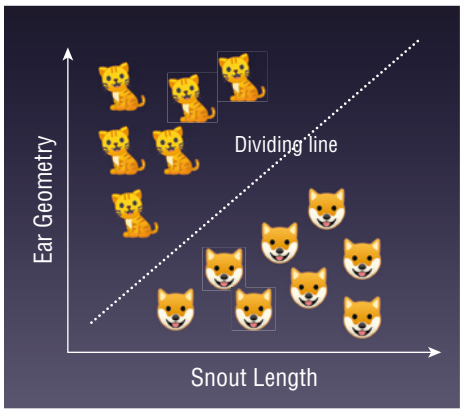

Figure 8.1: Using SVM to separate two classes of animals

Python® Machine Learning, First Edition. Wei-Meng Lee.

Once the line is drawn to separate the classes, you can then use it to predict future data. For example, given the snout length and ear geometry of a new unknown animal, you can now use the dividing line as a classifier to predict if the animal is a dog or a cat.

In this chapter, you will learn how SVM works and the various techniques you can use to adapt SVM for solving nonlinearly-separable datasets.

Maximum Separability

How does SVM separate two or more classes? Consider the set of points in Figure 8.2. Before you look at the next figure, visually think of a straight line dividing the points into two groups.

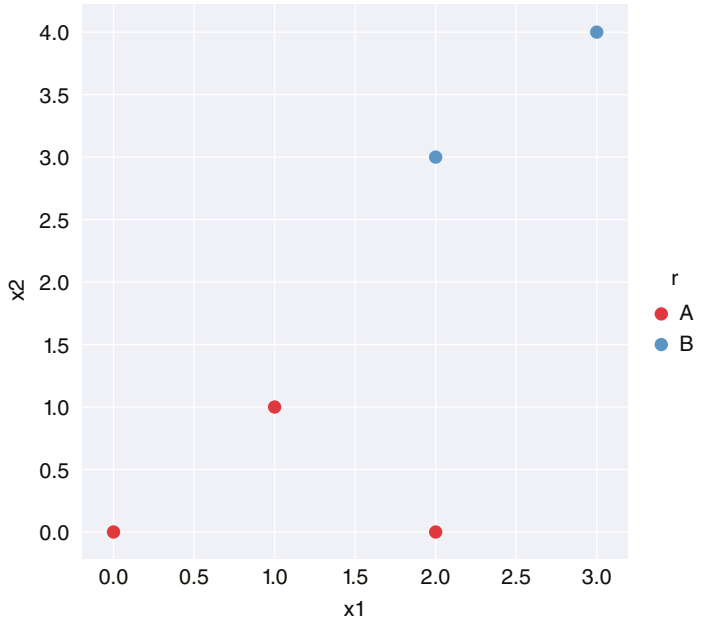

Figure 8.2: A set of points that can be separated using SVM

Now look at Figure 8.3, which shows two possible lines separating the two groups of points. Is this what you had in mind?

Though both lines separate the points into two distinct groups, which one is the right one? For SVM, the right line is the one that has the widest margins (with each margin touching at least a point in each class), as shown in Figure 8.4. In that figure, d1 and d2 are the width of the margins. The goal is to find the largest possible width for the margin that can separate the two groups. Hence, in this case d2 is the largest. Thus the line chosen is the one on the right.

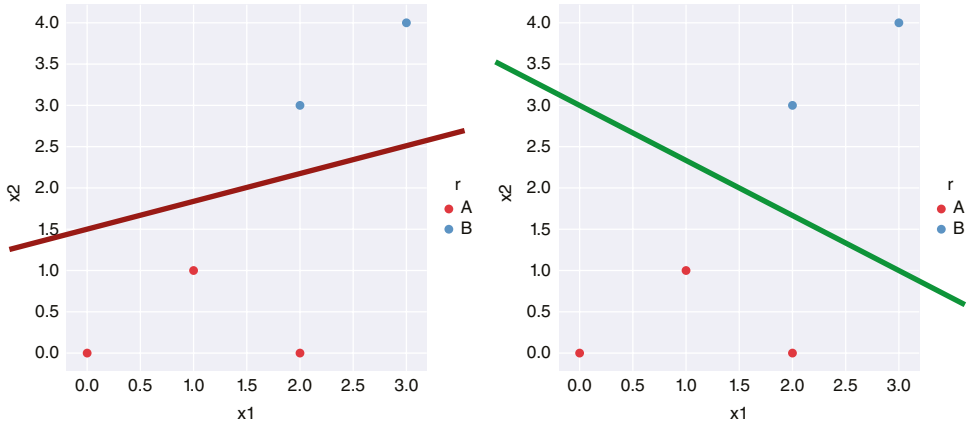

Figure 8.3: Two possible ways to split the points into two classes

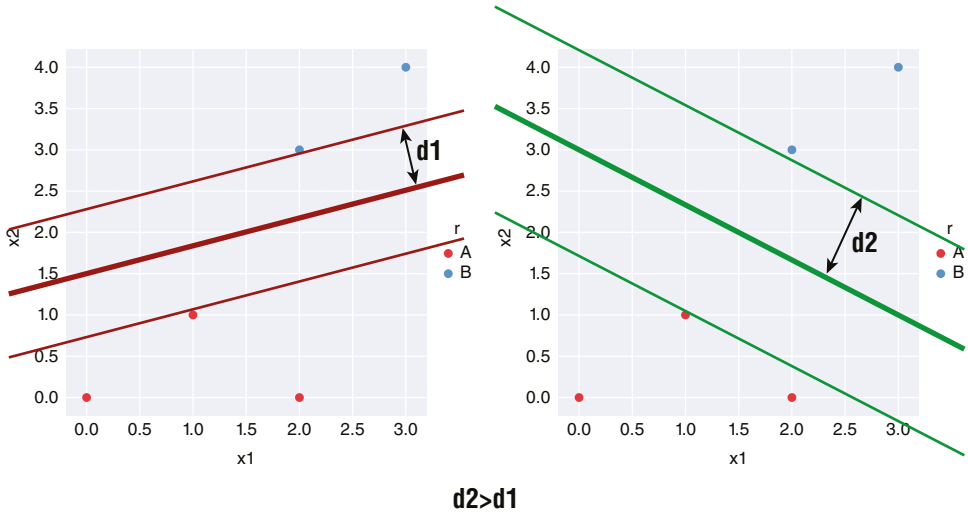

Figure 8.4: SVM seeks to split the two classes with the widest margin

Each of the two margins touches the closest point(s) to each group of points, and the center of the two margins is known as the *hyperplane*. The hyperplane is the line separating the two groups of points. We use the term "hyperplane" instead of "line" because in SVM we typically deal with more than two dimensions, and using the word "hyperplane" more accurately conveys the idea of a plane in a multidimensional space.

Support Vectors

A key term in SVM is *support vectors*. Support vectors are the points that lie on the two margins. Using the example from the previous section, Figure 8.5 shows the two support vectors lying on the two margins.

In this case, we say that there are two support vectors—one for each class.

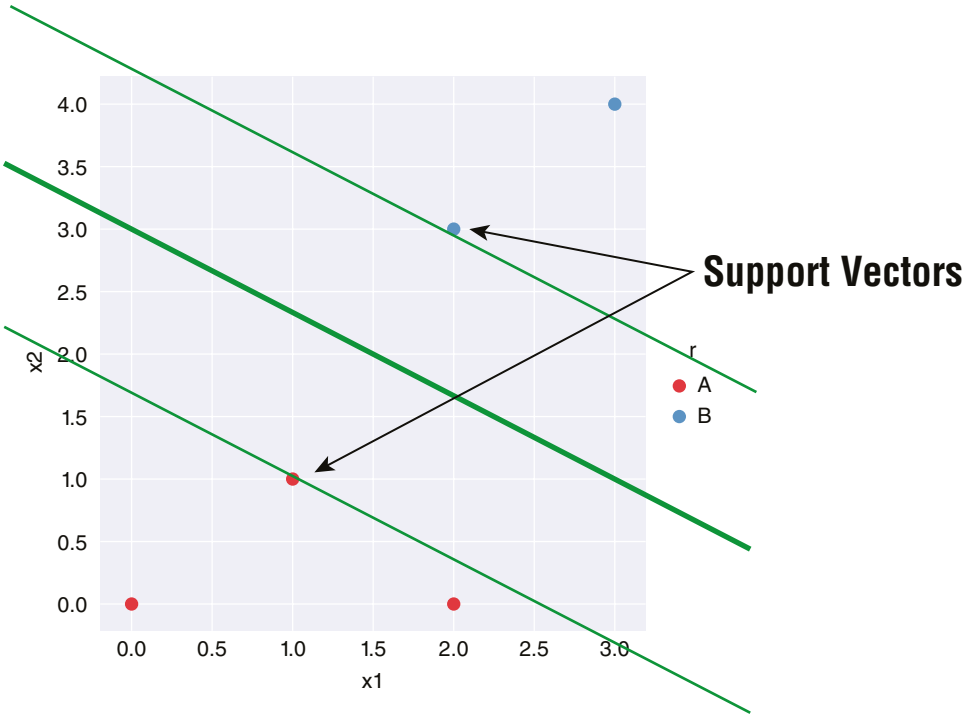

Figure 8.5: Support vectors are points that lie on the margins

Formula for the Hyperplane

With the series of points, the next question would be to find the formula for the hyperplane, together with the two margins. Without delving too much into the math behind this, Figure 8.6 shows the formula for getting the hyperplane.

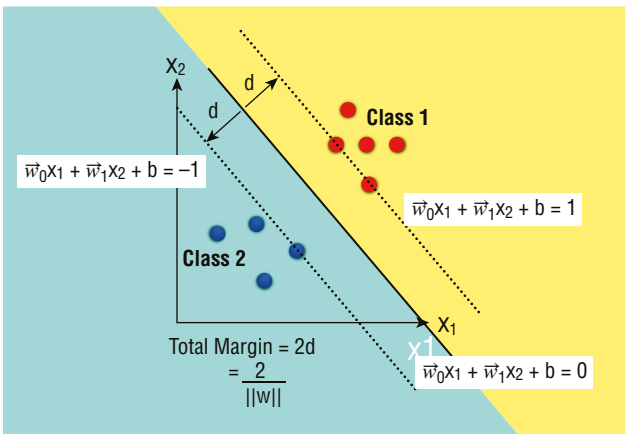

Figure 8.6: The formula for the hyperplane and its accompanying two margins

As you can see from Figure 8.6, the formula for hyperplane (g) is given as:

$$g(x) = \vec{W}_0 x_1 + \vec{W}_1 x_2 + b$$

where x_1 and x_2 are the inputs, $\vec{W}_0$ and $\vec{W}_1$ are the weight vectors, and b is the bias.

If the value of g is >= 1, then the point specified is in Class 1, and if the value of g is <= −1, then the point specified is in Class 2. As mentioned, the goal of SVM is to find the widest margins that divide the classes, and the total margin (2d) is defined by:

$$2 / \|w\|$$

where $\|w\|$ is the normalized weight vectors ($\vec{W}_0$ and $\vec{W}_1$). Using the training set, the goal is to minimize the value of $\|w\|$ so that you can get the maximum separability between the classes. Once this is done, you will be able to get the values of $\vec{W}_0$, $\vec{W}_1$, and b.

Finding the margin is a *Constrained Optimization* problem, which can be solved using the *Larange Multipliers* technique. It is beyond the scope of this book to discuss how to find the margin based on the dataset, but suffice it to say that we will make use of the Scikit-learn library to find them.

Using Scikit-learn for SVM

Now let's work on an example to see how SVM works and how to implement it using Scikit-learn. For this example, we have a file named `svm.csv` containing the following data:

```
x1,x2,r
0,0,A
1,1,A
2,3,B
2,0,A
3,4,B
```

The first thing that we will do is to plot the points using Seaborn:

```
%matplotlib inline
import pandas as pd
import numpy as np
import seaborn as sns; sns.set(font_scale=1.2)
import matplotlib.pyplot as plt

data = pd.read_csv('svm.csv')
sns.lmplot('x1', 'x2',
           data=data,
           hue='r',
           palette='Set1',
           fit_reg=False,
           scatter_kws={"s": 50});
```

Figure 8.7 shows the points plotted using Seaborn.

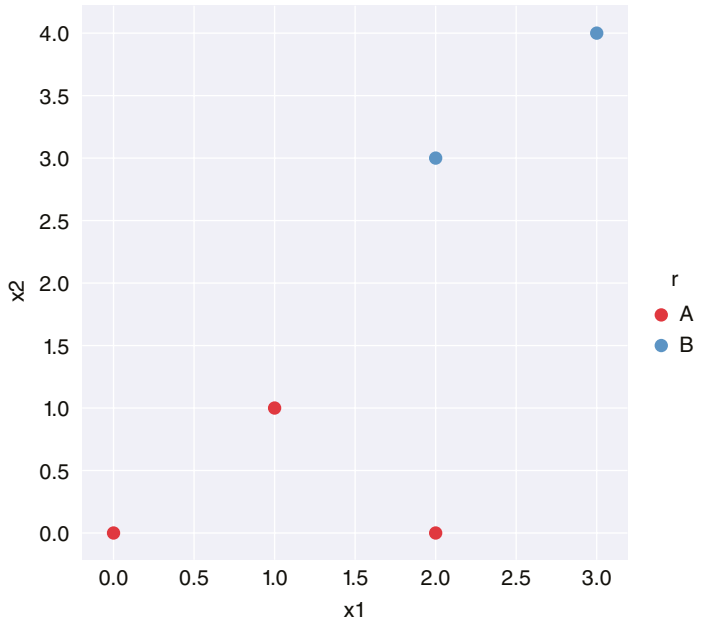

Figure 8.7: Plotting the points using Seaborn

Using the data points that we have previously loaded, now let's use Scikit-learn's `svm` module's `SVC` class to help us derive the value for the various variables that we need to compute otherwise. The following code snippet uses the *linear kernel* to solve the problem. The linear kernel assumes that the dataset can be separated linearly.

```
from sklearn import svm
#---Converting the Columns as Matrices---
points = data[['x1','x2']].values
result = data['r']

clf = svm.SVC(kernel = 'linear')
clf.fit(points, result)

print('Vector of weights (w) = ',clf.coef_[0])
print('b = ',clf.intercept_[0])
print('Indices of support vectors = ', clf.support_)
print('Support vectors = ', clf.support_vectors_)
print('Number of support vectors for each class = ', clf.n_support_)
print('Coefficients of the support vector in the decision function = ',
       np.abs(clf.dual_coef_))
```

The `SVC` stands for *Support Vector Classification*. The `svm` module contains a series of classes that implement SVM for different purposes:

`svm.LinearSVC`: Linear Support Vector Classification

`svm.LinearSVR`: Linear Support Vector Regression

`svm.NuSVC`: Nu-Support Vector Classification

`svm.NuSVR`: Nu-Support Vector Regression

`svm.OneClassSVM`: Unsupervised Outlier Detection

`svm.SVC`: C-Support Vector Classification

`svm.SVR`: Epsilon-Support Vector Regression

TIP **For this chapter, our focus is on using SVM for classification, even though SVM can also be used for regression.**

The preceding code snippet yields the following output:

```
Vector of weights (w) =  [0.4 0.8]
b =  -2.2
Indices of support vectors =  [1 2]
Support vectors =  [[1. 1.]
 [2. 3.]]
Number of support vectors for each class =  [1 1]
Coefficients of the support vector in the decision function =  [[0.4 0.4]]
```

As you can see, the vector of weights has been found to be [0.4 0.8], meaning that $\vec{W}_0$ is now 0.4 and $\vec{W}_1$ is now 0.8. The value of b is –2.2, and there are two support vectors. The index of the support vectors is 1 and 2, meaning that the points are the ones in bold:

x1	x2	r	
0	0	0	A
1	**1**	**1**	**A**
2	**2**	**3**	**B**
3	2	0	A
4	3	4	B

Figure 8.8 shows the relationship between the various variables in the formula and the variables in the code snippet.

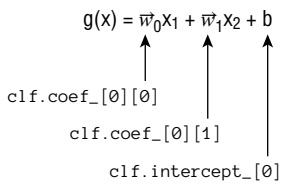

Figure 8.8: Relationships between the variables in the formula and the variables in the code snippet

Plotting the Hyperplane and the Margins

With the values of the variables all obtained, it is now time to plot the hyperplane and its two accompanying margins. Do you remember the formula for the hyperplane? It is as follows:

$$g(x) = \vec{W}_{0X_1} + \vec{W}_{1X_2} + b,$$

To plot the hyperplane, set $\vec{W}_{0X_1} + \vec{W}_{1X_2} + b$ to 0, like this:

$$\vec{W}_{0X_1} + \vec{W}_{1X_2} + b = 0$$

In order to plot the hyperplane (which is a straight line in this case), we need two points: one on the x-axis and one on the y-axis.

Using the preceding formula, when $x_1 = 0$, we can solve for x_2 as follows:

$$\vec{W}_0(0) + \vec{W}_{1x2} + b = 0$$
$$\vec{W}_{1x2} = -b$$
$$x_2 = -b/\vec{W}_1$$

When $x_2 = 0$, we can solve for x_1 as follows:

$$\vec{W}_{0x1} + \vec{W}_1(0) + b = 0$$
$$\vec{W}_{0x1} = -b$$
$$x_1 = -b/\vec{W}_0$$

The point $(0, -b/\vec{W}_1)$ is the *y-intercept* of the straight line. Figure 8.9 shows the two points on the two axes.

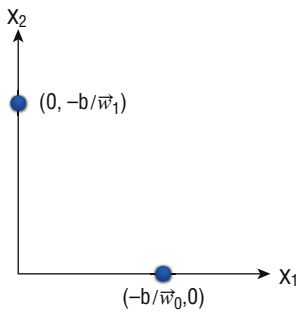

Figure 8.9: The two intercepts for the hyperplane

Once the points on each axis are found, you can now calculate the *slope* as follows:

$$\text{Slope} = (-b/\vec{W}_1) \;/\; (b/\vec{W}_0)$$
$$\text{Slope} = -(\vec{W}_0/\vec{W}_1)$$

With the slope and y-intercept of the line found, you can now go ahead and plot the hyperplane. The following code snippet does just that:

```
#---w is the vector of weights---
w = clf.coef_[0]

#---find the slope of the hyperplane---
slope = -w[0] / w[1]

b = clf.intercept_[0]

#---find the coordinates for the hyperplane---
xx = np.linspace(0, 4)
yy = slope * xx - (b / w[1])

#---plot the margins---
s = clf.support_vectors_[0]    #---first support vector---
yy_down = slope * xx + (s[1] - slope * s[0])

s = clf.support_vectors_[-1]   #---last support vector---
yy_up   = slope * xx + (s[1] - slope * s[0])

#---plot the points---
sns.lmplot('x1', 'x2', data=data, hue='r', palette='Set1',
fit_reg=False, scatter_kws={"s": 70})

#---plot the hyperplane---
plt.plot(xx, yy, linewidth=2, color='green');

#---plot the 2 margins---
plt.plot(xx, yy_down, 'k--')
plt.plot(xx, yy_up, 'k--')
```

Figure 8.10 shows the hyperplane and the two margins.

Making Predictions

Remember, the goal of SVM is to separate the points into two or more classes, so that you can use it to predict the classes of future points. Having trained your model using SVM, you can now perform some predictions using the model.

The following code snippet uses the model that you have trained to perform some predictions:

```
print(clf.predict([[3,3]])[0])  # 'B'
print(clf.predict([[4,0]])[0])  # 'A'
print(clf.predict([[2,2]])[0])  # 'B'
print(clf.predict([[1,2]])[0])  # 'A'
```

Check the points against the chart shown in Figure 8.10 and see if it makes sense to you.

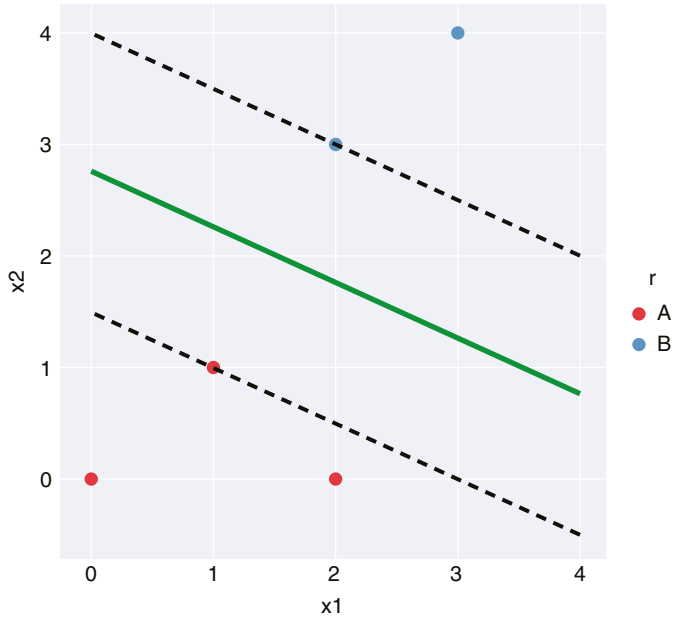

Figure 8.10: The hyperplane and the two margins

Kernel Trick

Sometimes, the points in a dataset are not always linearly separable. Consider the points shown in Figure 8.11.

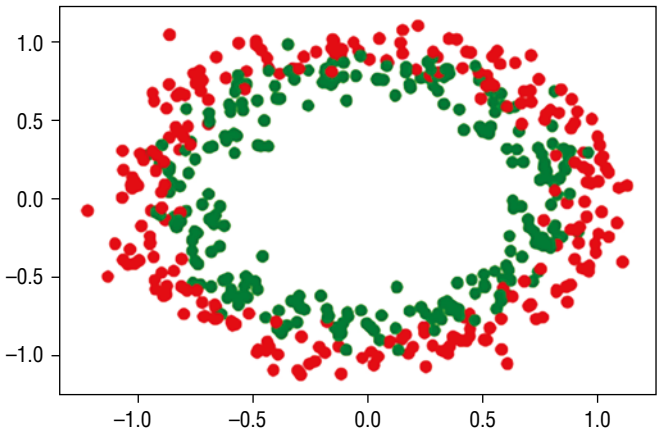

Figure 8.11: A scatter plot of two groups of points distributed in circular fashion

You can see that it is not possible to draw a straight line to separate the two sets of points. With some manipulation, however, you can make this set of points linearly separable. This technique is known as the *kernel trick*. The kernel trick is a technique in machine learning that transforms data into a higher dimension space so that, after the transformation, it has a clear dividing margin between classes of data.

Adding a Third Dimension

To do so, we can add a third dimension, say the z-axis, and define z to be:

$$z = x^2 + y^2$$

Once we plot the points using a 3D chart, the points are now linearly separable. It is difficult to visualize this unless you plot the points out. The following code snippet does just that:

```
%matplotlib inline

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import make_circles

#---X is features and c is the class labels---
X, c = make_circles(n_samples=500, noise=0.09)

rgb = np.array(['r', 'g'])
plt.scatter(X[:, 0], X[:, 1], color=rgb[c])
plt.show()

fig = plt.figure(figsize=(18,15))
ax = fig.add_subplot(111, projection='3d')
z = X[:,0]**2 + X[:,1]**2
ax.scatter(X[:, 0], X[:, 1], z, color=rgb[c])
plt.xlabel("x-axis")
plt.ylabel("y-axis")
plt.show()
```

We first create two sets of random points (a total of 500 points) distributed in circular fashion using the `make_circles()` function. We then plot them out on a 2D chart (as what was shown in Figure 8.11). We then add the third axis, the z-axis, and plot the chart in 3D (see Figure 8.12).

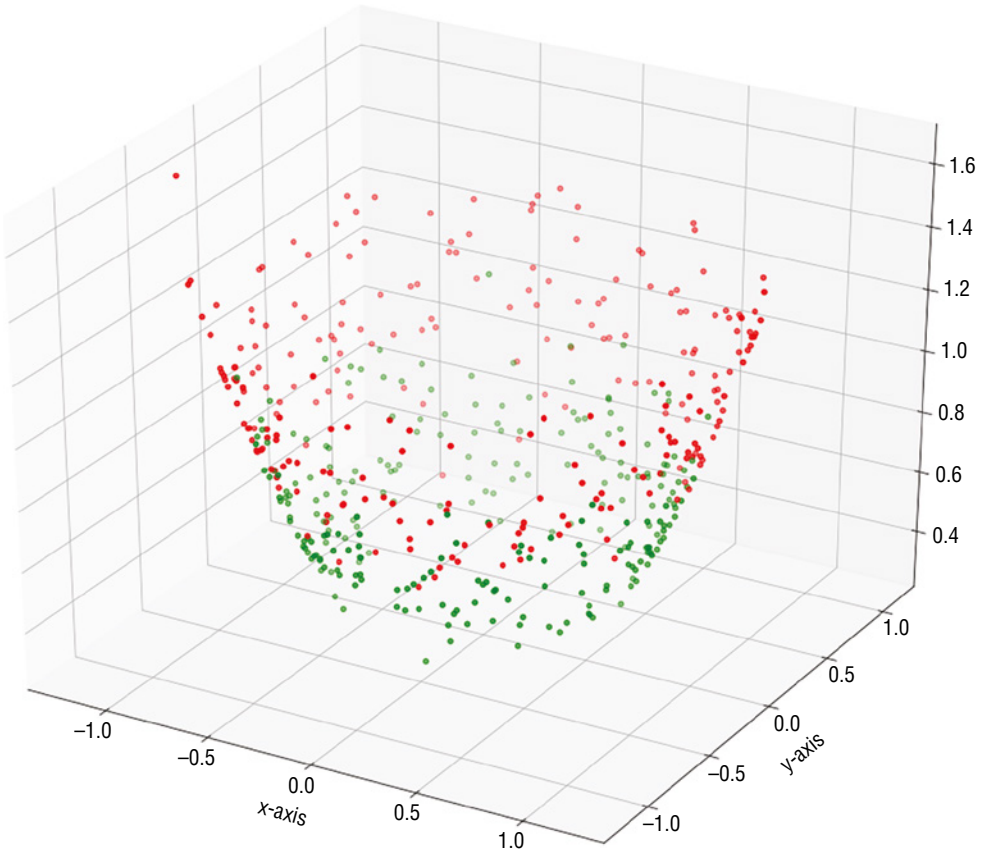

Figure 8.12: Plotting the points in the three dimensions

TIP **If you run the preceding code in Terminal (just remove the `%matplotlib inline` statement at the top of the code snippet) using the `python` command, you will be able to rotate and interact with the chart. Figure 8.13 shows the different perspectives of the 3D chart.**

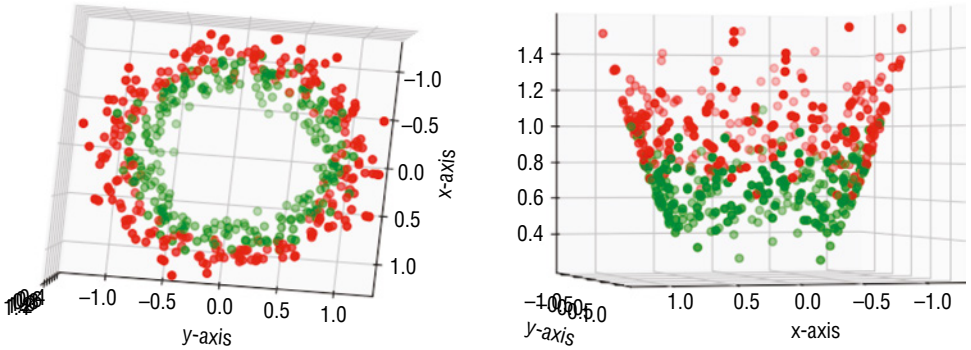

Figure 8.13: The various perspectives on the same dataset in 3D

Plotting the 3D Hyperplane

With the points plotted in a 3D chart, let's now train the model using the third dimension:

```
#---combine X (x-axis,y-axis) and z into single ndarray---
features = np.concatenate((X,z.reshape(-1,1)), axis=1)

#---use SVM for training---
from sklearn import svm

clf = svm.SVC(kernel = 'linear')
clf.fit(features, c)
```

First, we combined the three axes into a single `ndarray` using the `np.concatenate()` function. We then trained the model as usual. For a linearly-separable set of points in two dimensions, the formula for the hyperplane is as follows:

$$g(x) = \vec{W}_{0x1} + \vec{W}_{1x2} + b$$

For the set of points now in three dimensions, the formula now becomes the following:

$$g(x) = \vec{W}_{0x1} + \vec{W}_{1x2} + \vec{W}_{2x3} + b$$

In particular, $\vec{W}_2$ is now represented by `clf.coef_[0][2]`, as shown in Figure 8.14.

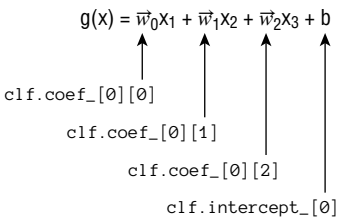

Figure 8.14: The formula for the hyperplane in 3D and its corresponding variables in the code snippet

The next step is to draw the hyperplane in 3D. In order to do that, you need to find the value of `x3`, which can be derived, as shown in Figure 8.15.

$$\vec{w}_0x_1 + \vec{w}_1x_2 + \vec{w}_2x_3 + b = 0$$
$$\vec{w}_2x_3 = -\vec{w}_0x_1 - \vec{w}_1x_2 - b$$
$$x_3 = \frac{-\vec{w}_0x_1 - \vec{w}_1x_2 - b}{\vec{w}_2}$$

Figure 8.15: Formula for finding the hyperplane in 3D

This can be expressed in code as follows:

```
x3 = lambda x,y: (-clf.intercept_[0] - clf.coef_[0][0] * x-clf.coef_[0][1] * y) /
                  clf.coef_[0][2]
```

To plot the hyperplane in 3D, use the `plot_surface()` function:

```
tmp = np.linspace(-1.5,1.5,100)
x,y = np.meshgrid(tmp,tmp)

ax.plot_surface(x, y, x3(x,y))
plt.show()
```

The entire code snippet is as follows:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import make_circles

#---X is features and c is the class labels---
X, c = make_circles(n_samples=500, noise=0.09)
z = X[:,0]**2 + X[:,1]**2

rgb = np.array(['r', 'g'])

fig = plt.figure(figsize=(18,15))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X[:, 0], X[:, 1], z, color=rgb[c])
plt.xlabel("x-axis")
plt.ylabel("y-axis")
# plt.show()

#---combine X (x-axis,y-axis) and z into single ndarray---
features = np.concatenate((X,z.reshape(-1,1)), axis=1)

#---use SVM for training---
from sklearn import svm

clf = svm.SVC(kernel = 'linear')
clf.fit(features, c)
x3 = lambda x,y: (-clf.intercept_[0] - clf.coef_[0][0] * x-clf.coef_[0][1]
                  * y) / clf.coef_[0][2]
```

```
tmp = np.linspace(-1.5,1.5,100)
x,y = np.meshgrid(tmp,tmp)

ax.plot_surface(x, y, x3(x,y))
plt.show()
```

Figure 8.16 shows the hyperplane, as well as the points, plotted in 3D.

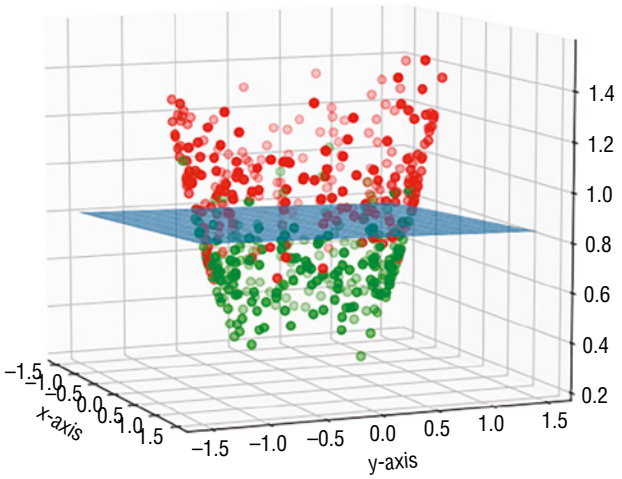

Figure 8.16: The hyperplane in 3D cutting through the two sets of points

Types of Kernels

Up to this point, we only discussed one type of SVM—linear SVM. As the name implies, linear SVM uses a straight line to separate the points. In the previous section, you also learned about the use of kernel tricks to separate two sets of data that are distributed in a circular fashion and then used linear SVM to separate them.

Sometimes, not all points can be separated linearly, nor can they be separated using the kernel tricks that you observed in the previous section. For this type of data, you need to "bend" the lines to separate them. In machine learning, *kernels* are functions that transform your data from nonlinear spaces to linear ones (see Figure 8.17).

To understand how kernels work, let's use the Iris dataset as an example. The following code snippet loads the Iris dataset and prints out the features, target, and target names:

```
%matplotlib inline
import pandas as pd
import numpy as np
```

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

iris = datasets.load_iris()
print(iris.data[0:5])      # print first 5 rows
print(iris.feature_names)  # ['sepal length (cm)', 'sepal width (cm)',
                           #  'petal length (cm)', 'petal width (cm)']
print(iris.target[0:5])    # print first 5 rows
print(iris.target_names)   # ['setosa' 'versicolor' 'virginica']
```

To illustrate, we will only use the first two features of the Iris dataset:

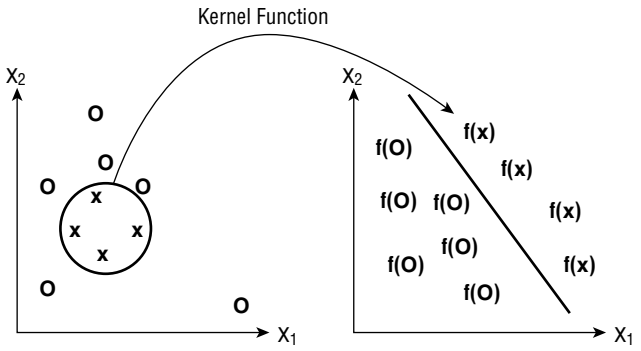

Figure 8.17: A kernel function transforms your data from nonlinear spaces to linear ones

```
X = iris.data[:, :2]       #  take the first two features
y = iris.target
```

We will plot the points using a scatter plot (see Figure 8.18):

```
#---plot the points---
colors = ['red', 'green', 'blue']
for color, i, target in zip(colors, [0, 1, 2], iris.target_names):
    plt.scatter(X[y==i, 0], X[y==i, 1], color=color, label=target)

plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.legend(loc='best', shadow=False, scatterpoints=1)

plt.title('Scatter plot of Sepal width against Sepal length')
plt.show()
```

Next, we will use the SVC class with the linear kernel:

```
C = 1  # SVM regularization parameter
clf = svm.SVC(kernel='linear', C=C).fit(X, y)
title = 'SVC with linear kernel'
```

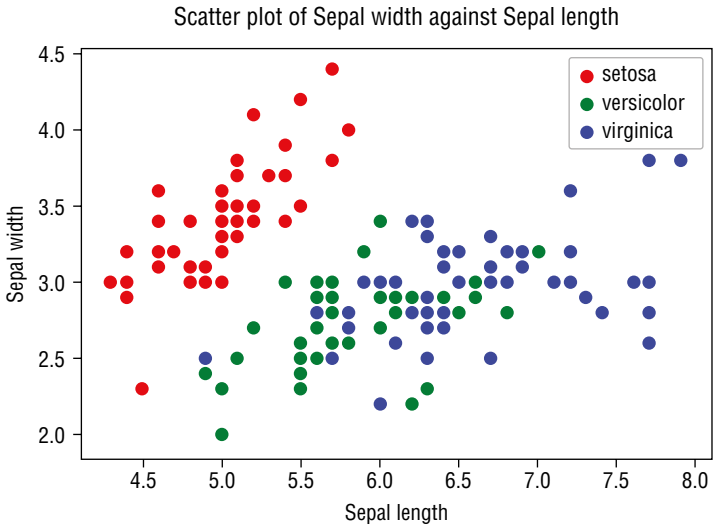

Figure 8.18: Scatter plot of the Iris dataset's first two features

TIP **Notice that this time around, we have a new parameter C. We will discuss this in a moment.**

Instead of drawing lines to separate the three groups of Iris flowers, this time we will paint the groups in colors using the `contourf()` function:

```
#---min and max for the first feature---
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1

#---min and max for the second feature---
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

#---step size in the mesh---
h = (x_max / x_min)/100

#---make predictions for each of the points in xx,yy---
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

#---draw the result using a color plot---
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=plt.cm.Accent, alpha=0.8)

#---plot the training points---
colors = ['red', 'green', 'blue']
for color, i, target in zip(colors, [0, 1, 2], iris.target_names):
    plt.scatter(X[y==i, 0], X[y==i, 1], color=color, label=target)
```

```
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.title(title)
plt.legend(loc='best', shadow=False, scatterpoints=1)
```

Figure 8.19 shows the scatter plots as well as the groups determined by the SVM linear kernel.

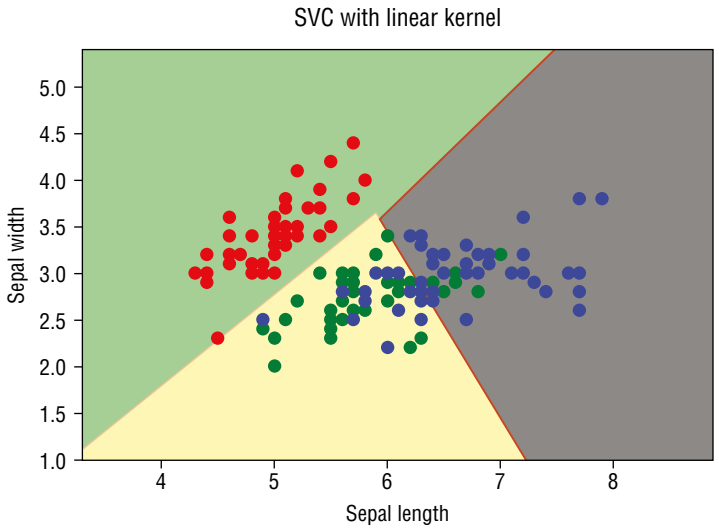

Figure 8.19: Using the SVM linear kernel

Once the training is done, we will perform some predictions:

```
predictions = clf.predict(X)
print(np.unique(predictions, return_counts=True))
```

The preceding code snippet returns the following:

```
(array([0, 1, 2]), array([50, 53, 47]))
```

This means that after the feeding the model with the Iris dataset, 50 are classified as "setosa," 53 are classified as "versicolor," and 47 are classified as "virginica."

C

In the previous section, you saw the use of the C parameter:

```
C = 1
clf = svm.SVC(kernel='linear', C=C).fit(X, y)
```

C is known as the *penalty parameter of the error term*. It controls the tradeoff between the smooth decision boundary and classifying the training points correctly. For example, if the value of C is high, then the SVM algorithm will seek to ensure that all points are classified correctly. The downside to this is that it may result in a narrower margin, as shown in Figure 8.20.

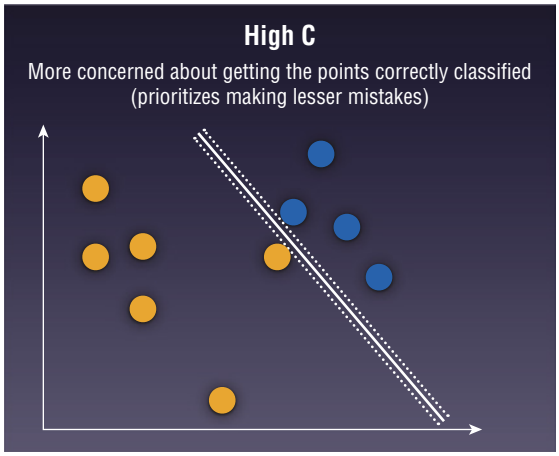

Figure 8.20: A high C focuses more on getting the points correctly classified

In contrast, a lower C will aim for the widest margin possible, but it will result in some points being classified incorrectly (see Figure 8.21).

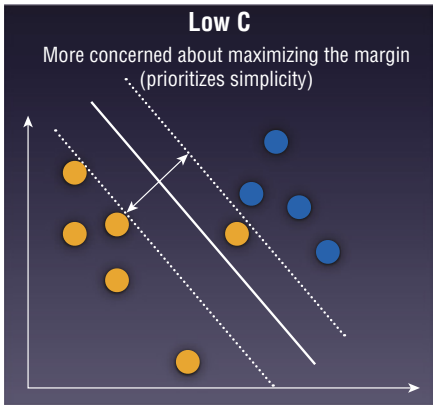

Figure 8.21: A low C aims for the widest margin, but may classify some points incorrectly

Figure 8.22 shows the effects of varying the value of C when applying the SVM linear kernel algorithm. The result of the classification appears at the bottom of each chart.

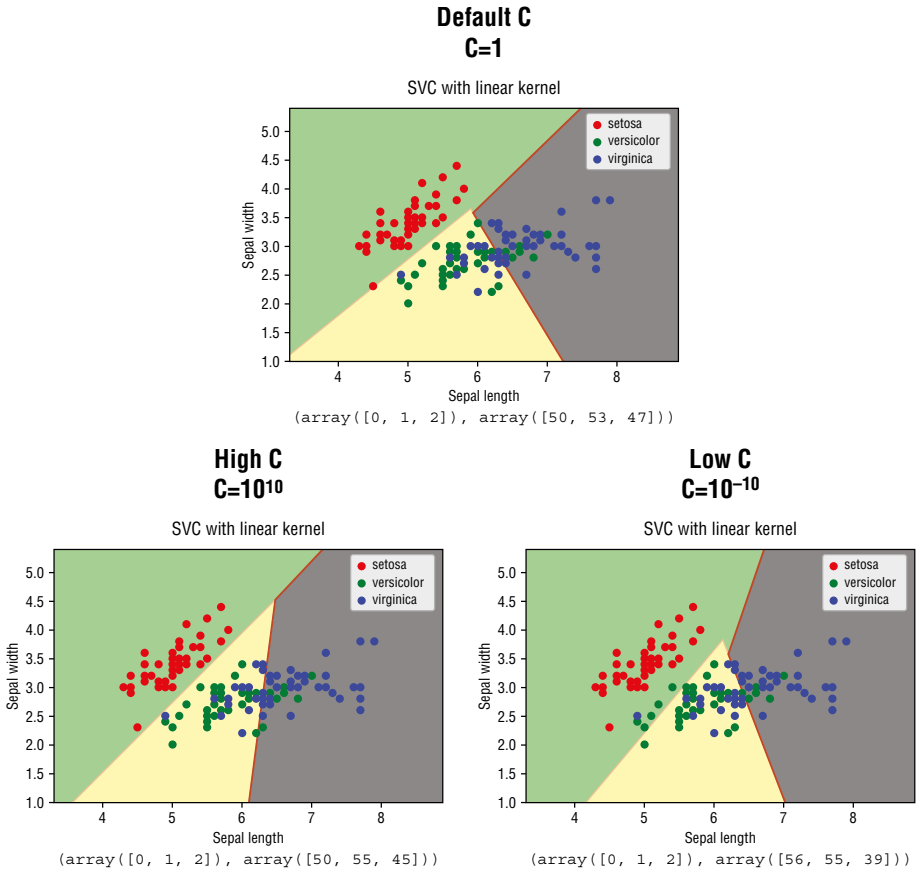

Figure 8.22: Using SVM with varying values of C

Note that when C is 1 or 10^{10}, there isn't too much difference among the classification results. However, when C is small (10^{-10}), you can see that a number of points (belonging to "versicolor" and "virginica") are now misclassified as "setosa."

TIP **In short, a low C makes the decision surface smooth while trying to classify *most* points, while a high C tries to classify *all* of the points correctly.**

Radial Basis Function (RBF) Kernel

Besides the linear kernel that we have seen so far, there are some commonly used nonlinear kernels:

- *Radial Basis function (RBF)*, also known as *Gaussian Kernel*
- Polynomial

The first, RBF, gives value to each point based on its distance from the origin or a *fixed* center, commonly on a Euclidean space. Let's use the same example that we used in the previous section, but this time modify the kernel to use `rbf`:

```
C = 1
clf = svm.SVC(kernel='rbf', gamma='auto', C=C).fit(X, y)
title = 'SVC with RBF kernel'
```

Figure 8.23 shows the same sample trained using the RBF kernel.

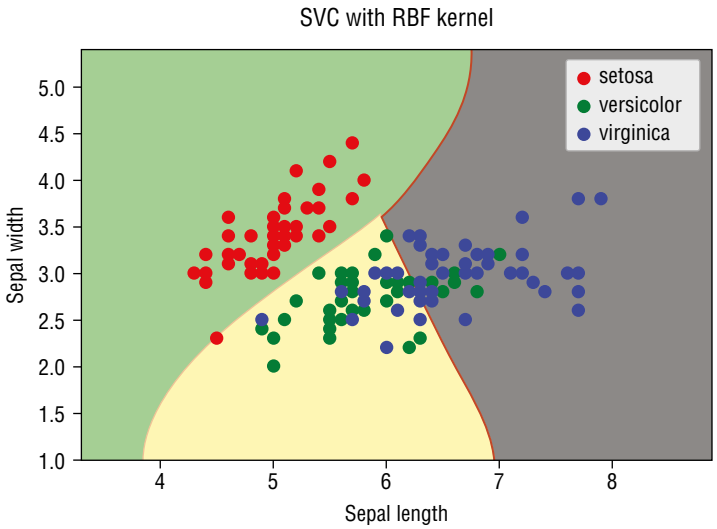

Figure 8.23: The Iris dataset trained using the RBF kernel

Gamma

If you look at the code snippet carefully, you will discover a new parameter—`gamma`. *Gamma* defines how far the influence of a single training example reaches. Consider the set of points shown in Figure 8.24. There are two classes of points—x's and o's.

A low Gamma value indicates that every point has a far reach (see Figure 8.25).

On the other hand, a high Gamma means that the points closest to the decision boundary have a close reach. The higher the value of Gamma, the more it will try to fit the training dataset exactly, resulting in overfitting (see Figure 8.26).

Figure 8.27 shows the classification of the points using RBF, with varying values of C and Gamma.

Note that if Gamma is high (10), overfitting occurs. You can also see from this figure that the value of C controls the smoothness of the curve.

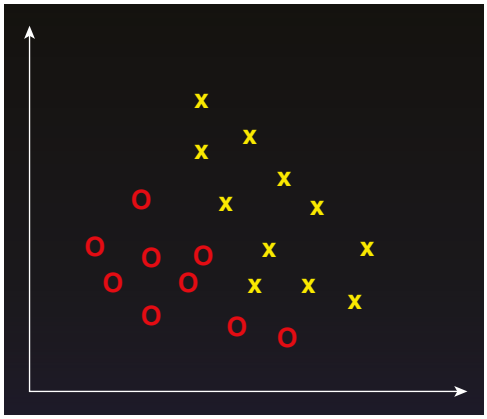

Figure 8.24: A set of points belonging to two classes

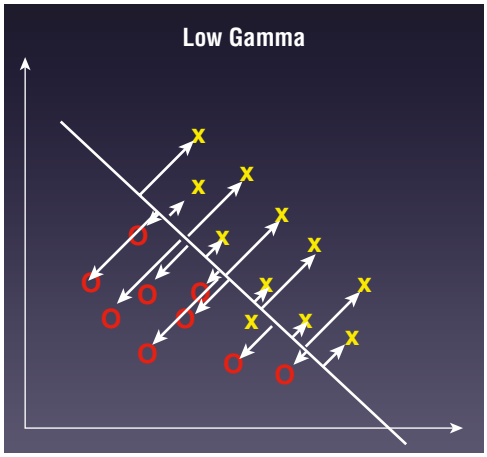

Figure 8.25: A low Gamma value allows every point to have equal reach

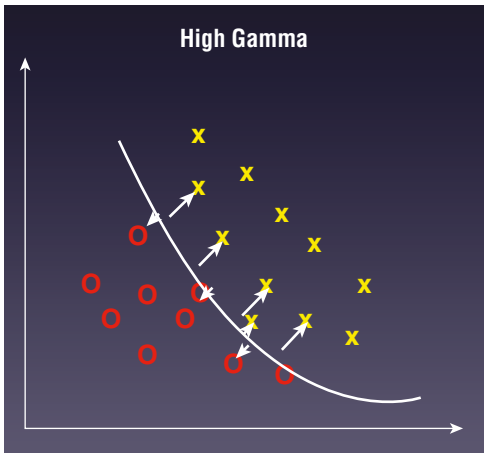

Figure 8.26: A high Gamma value focuses more on points close to the boundary

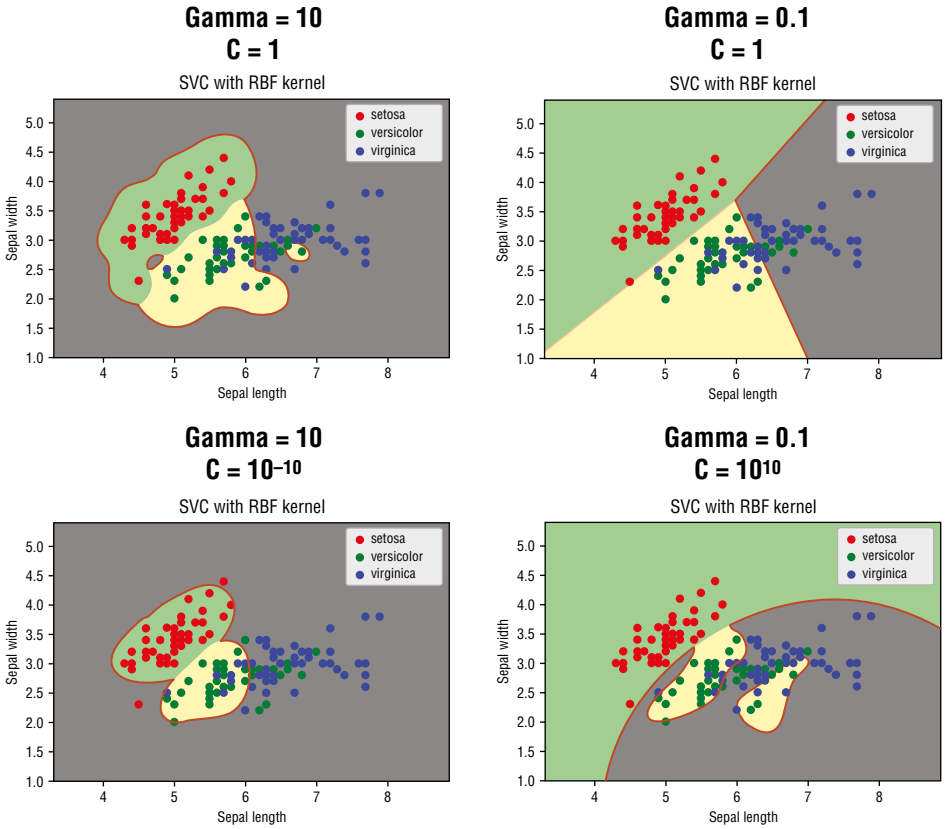

Figure 8.27: The effects of classifying the points using varying values of C and Gamma

TIP **To summarize, C controls the smoothness of the boundary and Gamma determines if the points are overfitted.**

Polynomial Kernel

Another type of kernel is called the *polynomial kernel*. A polynomial kernel of degree 1 is similar to that of the linear kernel. Higher-degree polynomial kernels afford a more flexible decision boundary. The following code snippet shows the Iris dataset trained using the polynomial kernel with degree 4:

```
C = 1  # SVM regularization parameter
clf = svm.SVC(kernel='poly', degree=4, C=C, gamma='auto').fit(X, y)
title = 'SVC with polynomial (degree 4) kernel'
```

Figure 8.28 shows the dataset separated with polynomial kernels of degree 1 to 4.

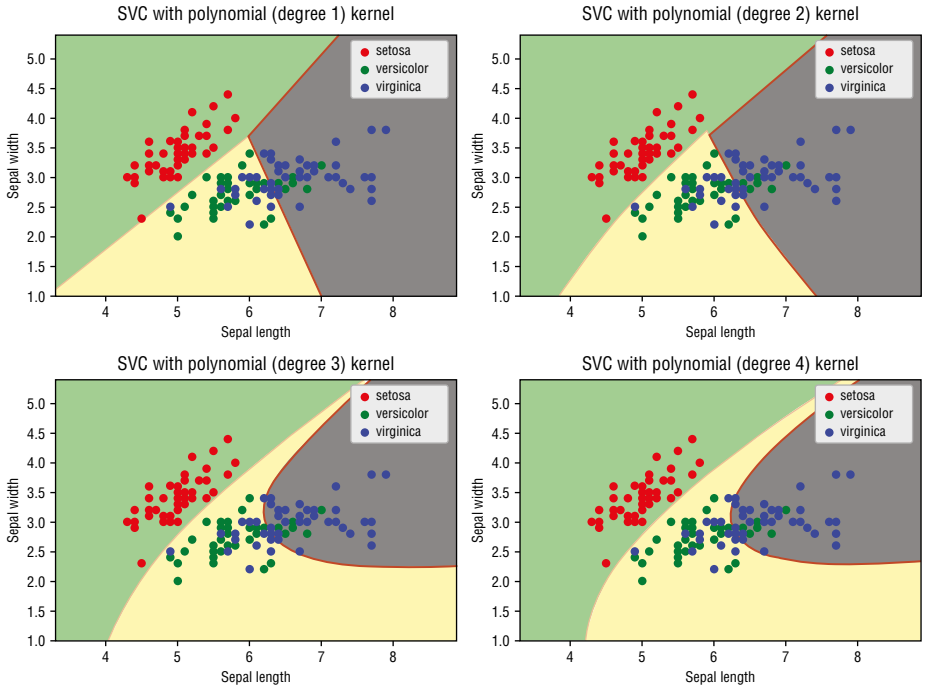

Figure 8.28: The classification of the Iris dataset using polynomial kernel of varying degrees

Using SVM for Real-Life Problems

We will end this chapter by applying SVM to a common problem in our daily lives. Consider the following dataset (saved in a file named `house_sizes_prices_svm.csv`) containing the size of houses and their asking prices (in thousands) for a particular area:

```
size,price,sold
550,50,y
1000,100,y
1200,123,y
1500,350,n
3000,200,y
2500,300,y
750, 45,y
1500,280,n
780,400,n
1200, 450,n
2750, 500,n
```

The third column indicates if the house was sold. Using this dataset, you want to know if a house with a specific asking price would be able to sell.

First, let's plot out the points:

```
%matplotlib inline

import pandas as pd
import numpy as np
from sklearn import svm
import matplotlib.pyplot as plt
import seaborn as sns; sns.set(font_scale=1.2)

data = pd.read_csv('house_sizes_prices_svm.csv')

sns.lmplot('size', 'price',
           data=data,
           hue='sold',
           palette='Set2',
           fit_reg=False,
           scatter_kws={"s": 50});
```

Figure 8.29 shows the points plotted as a scatter plot.

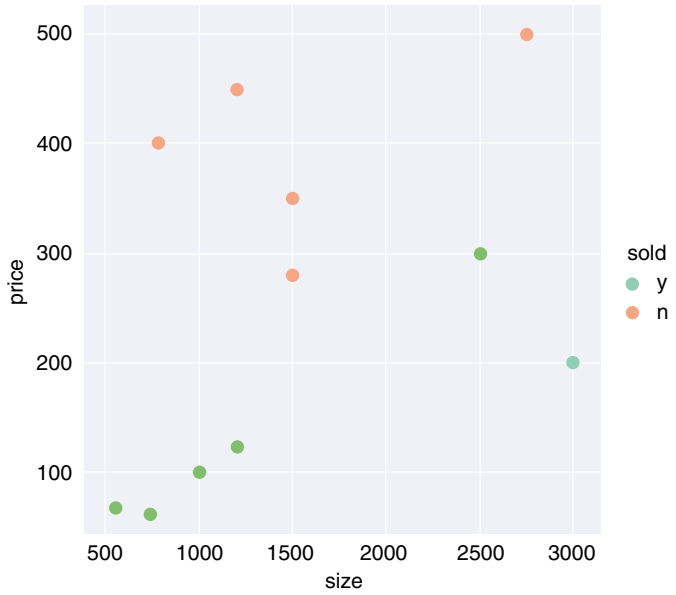

Figure 8.29: Plotting the points on a scatter plot

Visually, you can see that this is a problem that can be solved with SVM's linear kernel:

```
X = data[['size','price']].values
y = np.where(data['sold']=='y', 1, 0) #--1 for Y and 0 for N---
model = svm.SVC(kernel='linear').fit(X, y)
```

With the trained model, you can now perform predictions and paint the two classes:

```
#---min and max for the first feature---
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1

#---min and max for the second feature---
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

#---step size in the mesh---
h = (x_max / x_min) / 20

#---make predictions for each of the points in xx,yy---
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

Z = model.predict(np.c_[xx.ravel(), yy.ravel()])

#---draw the result using a color plot---
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=plt.cm.Blues, alpha=0.3)

plt.xlabel('Size of house')
plt.ylabel('Asking price (1000s)')
plt.title("Size of Houses and Their Asking Prices")
```

Figure 8.30 shows the points and the classes to which they belong.

Figure 8.30: Separating the points into two classes

You can now try to predict if a house of a certain size with a specific selling price will be able to sell:

```
def will_it_sell(size, price):
    if(model.predict([[size, price]]))==0:
        print('Will not sell!')
    else:
        print('Will sell!')

#---do some prediction---
will_it_sell(2500, 400)  # Will not sell!
will_it_sell(2500, 200)  # Will sell!
```

Summary

In this chapter, you learned about how Support Vector Machines help in classification problems. You learned about the formula for finding the hyperplane, as well as the two accompanying margins. Fortunately, Scikit-learn provides the classes needed for training models using SVM, and with the parameters returned, you can plot the hyperplane and margins visually so that you can understand how SVM works. You also learned about the various kernels that you can apply to your SVM algorithms so that the dataset can be separated linearly.