CHAPTER

9

Supervised Learning—Classification Using K-Nearest Neighbors (KNN)

What Is K-Nearest Neighbors?

Up until this point, we have discussed three supervised learning algorithms: linear regression, logistics regression, and support vector machines. In this chapter, we will dive into another supervised machine learning algorithm known as *K-Nearest Neighbors (KNN)*.

KNN is a relatively simple algorithm compared to the other algorithms that we have discussed in previous chapters. It works by comparing the query instance's distance to the other training samples and selecting the K-nearest neighbors (hence its name). It then takes the majority of these K-neighbor classes to be the prediction of the query instance.

Figure 9.1 sums this up nicely. When k = 3, the closest three neighbors of the circle are the two squares and the one triangle. Based on the simple rule of majority, the circle is classified as a square. If k = 5, then the closest five neighbors are the two squares and the three triangles. Hence, the circle is classified as a triangle.

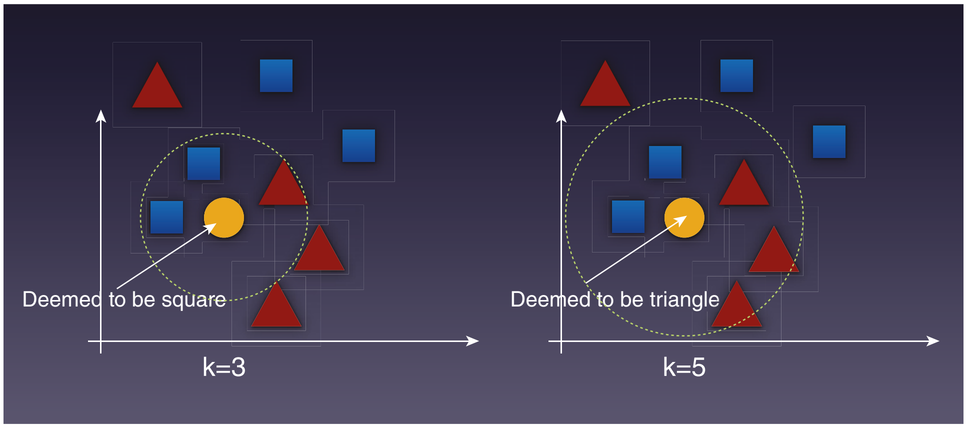

Figure 9.1: The classification of a point depends on the majority of its neighbors

TIP **KNN is also sometimes used for regression in addition to classification. For example, it can be used to calculate the average of the numerical target of the K-nearest neighbors. For this chapter, however, we are focusing solely on its more common use as a classification algorithm.**

Implementing KNN in Python

Now that you have seen how KNN works, let's try to implement KNN from scratch using Python. As usual, first let's import the modules that we'll need:

```
import pandas as pd
import numpy as np
import operator
import seaborn as sns
import matplotlib.pyplot as plt
```

Plotting the Points

For this example, you will use a file named `knn.csv` containing the following data:

```
x,y,c
1,1,A
2,2,A
4,3,B
3,3,A
3,5,B
5,6,B
5,4,B
```

As we have done in the previous chapters, a good way is to plot the points using Seaborn:

```
data = pd.read_csv("knn.csv")
sns.lmplot('x', 'y', data=data,
           hue='c', palette='Set1',
           fit_reg=False, scatter_kws={"s": 70})
plt.show()
```

Figure 9.2 shows the distribution of the various points. Points that belong to class A are displayed in red while those belonging to class B are displayed in blue.

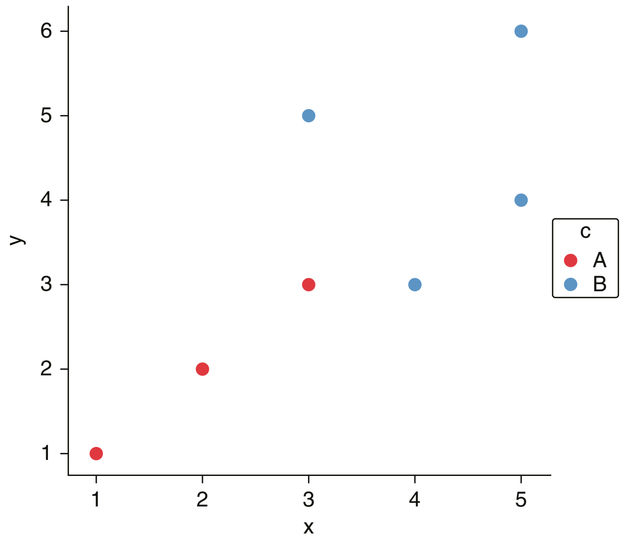

Figure 9.2: Plotting the points visually

Calculating the Distance Between the Points

In order to find the nearest neighbor of a given point, you need to calculate the Euclidean distance between two points.

TIP **In geometry, Euclidean space encompasses the two-dimensional Euclidean plane, the three-dimensional space of Euclidean geometry, and similar spaces of higher dimension.**

Given two points, $p = (p_1, p_2, \ldots, p_n)$ and $q = (q_1 q_2, \ldots, q_n)$, the distance between p and q is given by the following formula:

$$\sqrt{\left((q_1 - p_1)^2 + (q_2 - p_2)^2 + \ldots + (q_n - p_n)^2\right)}$$

Based on this formula, you can now define a function named `euclidean _ distance()` as follows:

```
#---to calculate the distance between two points---
def euclidean_distance(pt1, pt2, dimension):
    distance = 0
    for x in range(dimension):
        distance += np.square(pt1[x] - pt2[x])
    return np.sqrt(distance)
```

The `Euclidean _ distance()` function can find the distance between two points in any dimension. For this example, the points that we are dealing with are in 2D.

Implementing KNN

Next, define a function named `knn()`, which takes in the training points, the test point, and the value of k:

```
#---our own KNN model---
def knn(training_points, test_point, k):
    distances = {}

    #---the number of axes we are dealing with---
    dimension = test_point.shape[1]

    #--calculating euclidean distance between each
    # point in the training data and test data
    for x in range(len(training_points)):
        dist = euclidean_distance(test_point, training_points.iloc[x],
                                  dimension)
        #---record the distance for each training points---
        distances[x] = dist[0]

    #---sort the distances---
    sorted_d = sorted(distances.items(), key=operator.itemgetter(1))

    #---to store the neighbors---
    neighbors = []

    #---extract the top k neighbors---
    for x in range(k):
        neighbors.append(sorted_d[x][0])

    #---for each neighbor found, find out its class---
    class_counter = {}
    for x in range(len(neighbors)):
        #---find out the class for that particular point---
        cls = training_points.iloc[neighbors[x]][-1]
        if cls in class_counter:
            class_counter[cls] += 1
        else:
            class_counter[cls] = 1

    #---sort the class_counter in descending order---
    sorted_counter = sorted(class_counter.items(),
                            key=operator.itemgetter(1),
                            reverse=True)

    #---return the class with the most count, as well as the
    #neighbors found---
    return(sorted_counter[0][0], neighbors)
```

The function returns the class to which the test point belongs, as well as the indices of all the nearest k neighbors.

Making Predictions

With the `knn()` function defined, you can now make some predictions:

```
#---test point---
test_set = [[3,3.9]]
test = pd.DataFrame(test_set)
cls,neighbors = knn(data, test, 5)
print("Predicted Class: " + cls)
```

The preceding code snippet will print out the following output:

```
Predicted Class: B
```

Visualizing Different Values of K

It is useful to be able to visualize the effect of applying various values of k. The following code snippet draws a series of concentric circles around the test point based on the values of k, which range from 7 to 1, with intervals of –2:

```
#---generate the color map for the scatter plot---
#---if column 'c' is A, then use Red, else use Blue---
colors = ['r' if i == 'A' else 'b'  for i in data['c']]
ax = data.plot(kind='scatter', x='x', y='y', c = colors)
plt.xlim(0,7)
plt.ylim(0,7)

#---plot the test point---
plt.plot(test_set[0][0],test_set[0][1], "yo", markersize='9')

for k in range(7,0,-2):
    cls,neighbors = knn(data, test, k)
    print("============")
    print("k = ", k)
    print("Class", cls)
    print("Neighbors")
    print(data.iloc[neighbors])

    furthest_point = data.iloc[neighbors].tail(1)

    #---draw a circle connecting the test point
    #and the furthest point---
    radius = euclidean_distance(test, furthest_point.iloc[0], 2)

    #---display the circle in red if classification is A,
    # else display circle in blue---
    c = 'r' if cls=='A' else 'b'
    circle = plt.Circle((test_set[0][0], test_set[0][1]),
                        radius, color=c, alpha=0.3)
    ax.add_patch(circle)

plt.gca().set_aspect('equal', adjustable='box')
plt.show()
```

The preceding code snippet prints out the following output:

```
============
k =  7
Class B
Neighbors
   x  y  c
3  3  3  A
4  3  5  B
2  4  3  B
6  5  4  B
1  2  2  A
5  5  6  B
0  1  1  A
============
k =  5
Class B
Neighbors
   x  y  c
3  3  3  A
4  3  5  B
2  4  3  B
6  5  4  B
1  2  2  A
============
k =  3
Class B
Neighbors
```

```
   x  y  c
3  3  3  A
4  3  5  B
2  4  3  B
============
k =  1
Class A
Neighbors
   x  y  c
3  3  3  A
```

Figure 9.3 shows the series of circles centered around the test point, with varying values of k. The innermost circle is for k = 1, with the next outer ring for k = 3, and so on. As you can see, if k = 1, the circle is red, meaning that the yellow point has been classified as class A. If the circle is blue, this means that the yellow point has been classified as class B. This is evident in the outer three circles.

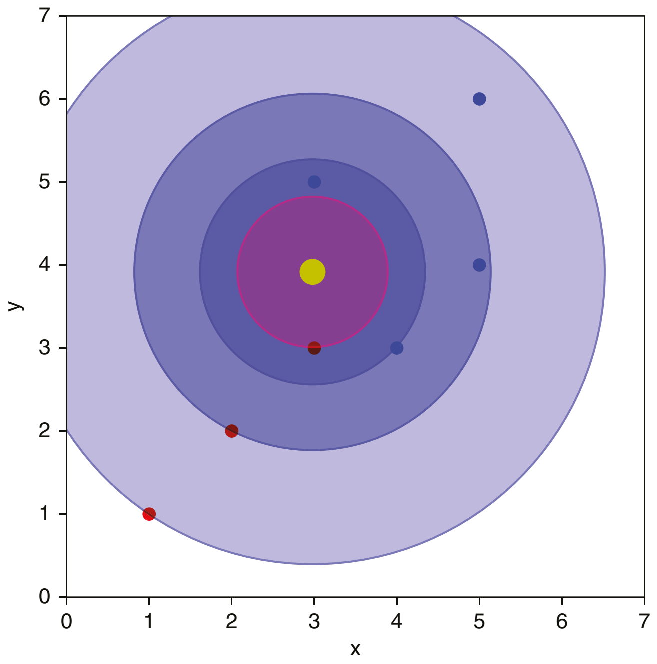

Figure 9.3: The classification of the yellow point based on the different values of k

Using Scikit-Learn's KNeighborsClassifier Class for KNN

Now that you have seen how KNN works and how it can be implemented manually in Python, let's use the implementation provided by Scikit-learn.

The following code snippet loads the Iris dataset and plots it out using a scatter plot:

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.patches as mpatches
from sklearn import svm, datasets
import matplotlib.pyplot as plt

iris = datasets.load_iris()

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

#---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()
```

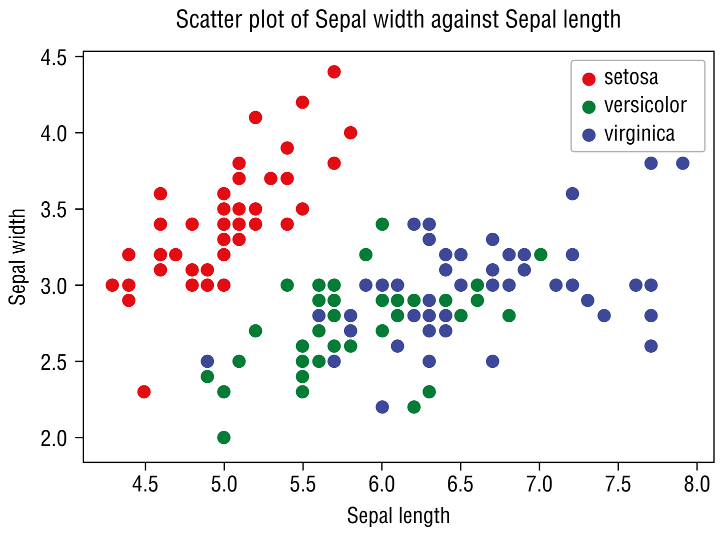

Figure 9.4: Plotting out the Sepal width against the Sepal length in a scatter plot

Figure 9.4 shows the scatter plot of the Sepal width against the Sepal length.

Exploring Different Values of K

We can now use Scikit-learn's `KNeighborsClassifier` class to help us train a model on the Iris dataset using KNN. For a start, let's use a k of 1:

```
from sklearn.neighbors import KNeighborsClassifier

k = 1
#---instantiate learning model---
knn = KNeighborsClassifier(n_neighbors=k)

#---fitting the model---
knn.fit(X, y)

#---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 = knn.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(f'KNN (k={k})')
plt.legend(loc='best', shadow=False, scatterpoints=1)

predictions = knn.predict(X)

#--classifications based on predictions---
print(np.unique(predictions, return_counts=True))
```

The preceding code snippet creates a *meshgrid* (a rectangular grid of values) of points scattered across the x- and y-axes. Each point is then used for prediction, and the result is drawn using a color plot.

Figure 9.5 shows the classification boundary using a k of 1. Notice that for k = 1, you perform your prediction based solely on a single sample—your nearest neighbor. This makes your prediction very sensitive to all sorts of distortions, such as outliers, mislabeling, and so on. In general, setting k = 1 usually leads to *overfitting*, and as a result your prediction is usually not very accurate.

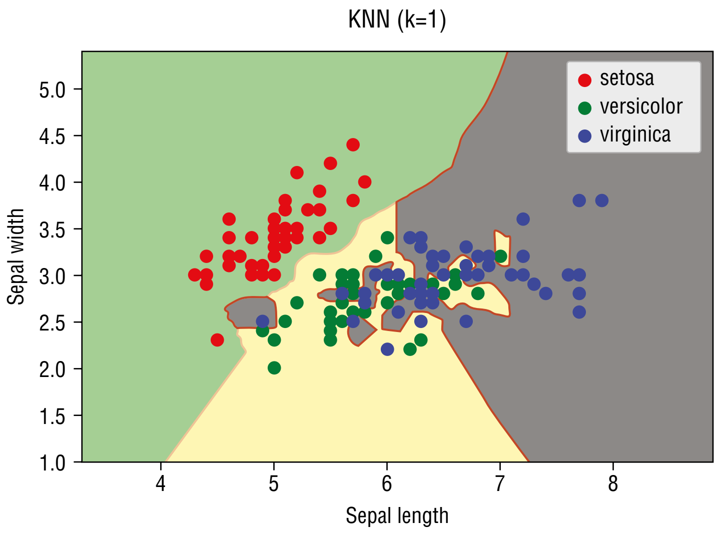

Figure 9.5: The classification boundary based on k = 1

TIP ***Overfitting* in machine learning means that the model you have trained fits the training data too well. This happens when all of the noises and fluctuations in your training data are picked up during the training process. In simple terms, this means that your model is trying very hard to fit all of your data perfectly. The key problem with an overfitted model is that it will not work well with new, unseen data.**

***Underfitting*, on the other hand, occurs when a machine learning model cannot accurately capture the underlying trend of the data. Specifically, the model does not fit the data well enough.**

Figure 9.6 shows an easy way to understand overfitting, underfitting, and a generally good fit.

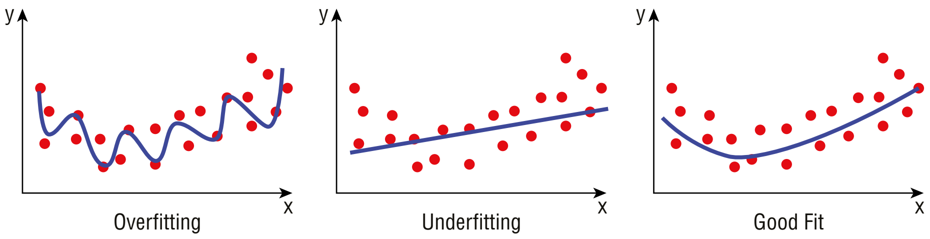

Figure 9.6: Understanding the concept of overfitting, underfitting, and a good fit

For KNN, setting k to a higher value tends to make your prediction more robust against noise in your data.

Using the same code snippet, let's vary the values of k. Figure 9.7 shows the classifications based on four different values of k.

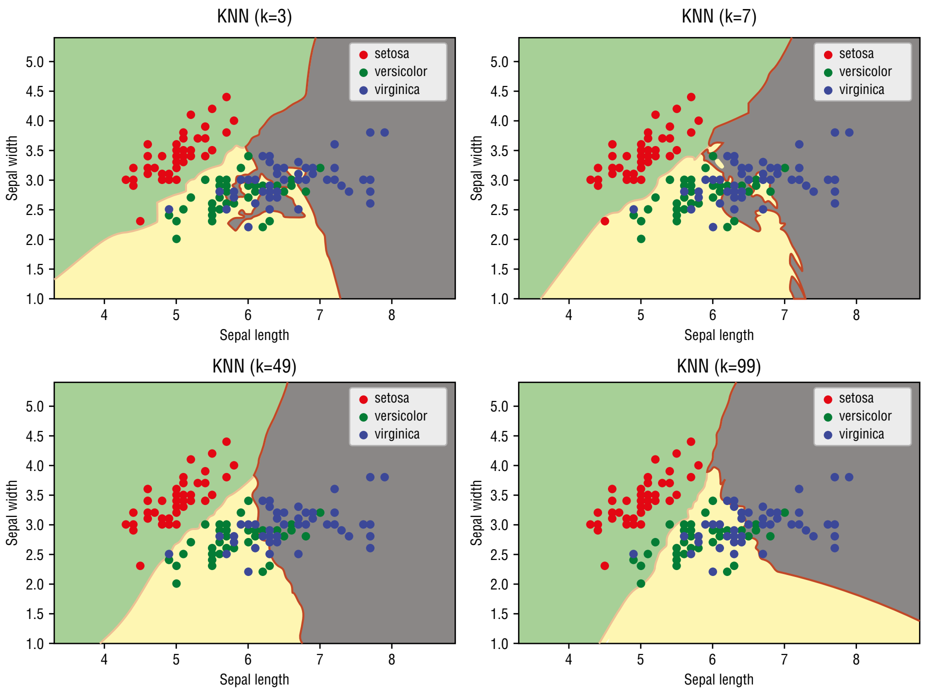

Figure 9.7: The effects of varying the values of k

Note that as k increases, the boundary becomes smoother. But it also means that more points will be classified incorrectly. When k increases to a large value, *underfitting* occurs.

The key issue with KNN is then how do you find out the ideal value of k to use?

Cross-Validation

In the previous few chapters, you have witnessed that we split our dataset into two individual sets—one for training and one for testing. However, the data in your dataset may not be distributed evenly, and as a result your test set may be too simple or too hard to predict, thereby making it very difficult to know if your model works well.

Instead of using part of the data for training and part for testing, you can split the data into *k-folds* and train the models *k* times, rotating the training and testing sets. By doing so, each data point is now being used for training and testing.

TIP **Do not confuse the *k* in k-folds with the *k* in KNN—they are not related.**

Figure 9.8 shows a dataset split into five folds (blocks). For the first run, blocks 1, 2, 3, and 4 will be used to train the model. Block 0 will be used to test the model. In the next run, blocks 0, 2, 3, and 4 will be used for training, and block 1 will be used for testing, and so on.

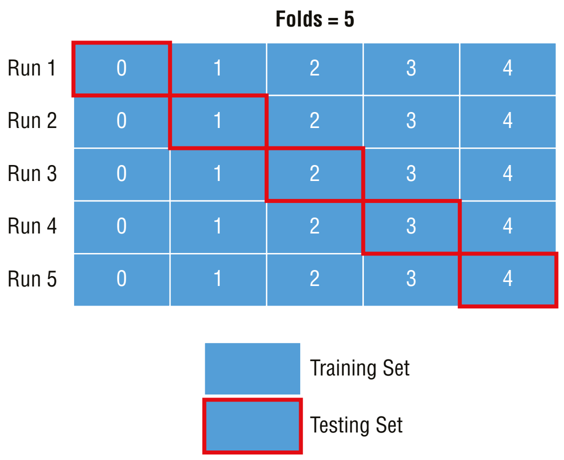

Figure 9.8: How cross-validation works

At the end of each run, the model is scored. At the end of the k-runs, the score is averaged. This averaged score will give you a good indication of how well your algorithm performs.

TIP **The purpose of *cross-validation* is not for training your model, but rather it is for model checking. Cross-validation is useful when you need to compare different machine learning algorithms to see how they perform with the given dataset. Once the algorithm is selected, you will use all of the data for training the model.**

Parameter-Tuning K

Now that you understand cross-validation, let's use it on our Iris dataset. We will train the model using all of the four features, and at the same time we shall use cross-validation on the dataset using 10 folds. We will do this for each value of k:

```
from sklearn.model_selection import cross_val_score

#---holds the cv (cross-validates) scores---
cv_scores = []

#---use all features---
X = iris.data[:, :4]
y = iris.target

#---number of folds---
folds = 10

#---creating odd list of K for KNN---
ks = list(range(1,int(len(X) * ((folds - 1)/folds))))

#---remove all multiples of 3---
ks = [k for k in ks if k % 3 != 0]

#---perform k-fold cross validation---
for k in ks:
    knn = KNeighborsClassifier(n_neighbors=k)

    #---performs cross-validation and returns the average accuracy---
    scores = cross_val_score(knn, X, y, cv=folds, scoring='accuracy')
    mean = scores.mean()
    cv_scores.append(mean)
    print(k, mean)
```

The Scikit-learn library provides the `cross _ val _ score()` function that performs cross-validation for you automatically, and it returns the metrics that you want (for example, accuracy).

When using cross-validation, be aware that at any one time, there will be *((folds-1)/folds) * total_rows* available for training. This is because *(1/folds) * total_rows* will be used for testing.

For KNN, there are three rules to which you must adhere:

- The value of k cannot exceed the number of rows for training.
- The value of k should be an odd number (so that you can avoid situations where there is a tie between the classes) for a two-class problem.
- The value of k must not be a multiple of the number of classes (to avoid ties, similar to the previous point).

Hence, the *ks* list in the preceding code snippet will contain the following values:

```
[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20, 22, 23, 25, 26, 28,
29, 31, 32, 34, 35, 37, 38, 40, 41, 43, 44, 46, 47, 49, 50, 52, 53, 55,
56, 58, 59, 61, 62, 64, 65, 67, 68, 70, 71, 73, 74, 76, 77, 79, 80, 82,
83, 85, 86, 88, 89, 91, 92, 94, 95, 97, 98, 100, 101, 103, 104, 106,
107, 109, 110, 112, 113, 115, 116, 118, 119, 121, 122, 124, 125, 127,
128, 130, 131, 133, 134]
```

After the training, the *cv _ scores* will contain a list of accuracies based on the different values of k:

```
1 0.96
2 0.9533333333333334
4 0.9666666666666666
5 0.9666666666666668
7 0.9666666666666668
8 0.9666666666666668
10 0.9666666666666668
11 0.9666666666666668
13 0.9800000000000001
14 0.9733333333333334
...
128 0.6199999999999999
130 0.6066666666666667
131 0.5933333333333332
133 0.5666666666666667
134 0.5533333333333333
```

Finding the Optimal K

To find the optimal k, you simply find the value of k that gives the highest accuracy. Or, in this case, you will want to find the lowest *misclassification error (MSE).*

The following code snippet finds the MSE for each k, and then finds the k with the lowest MSE. It then plots a line chart of MSE against k (see Figure 9.9):

```
#---calculate misclassification error for each k---
MSE = [1 - x for x in cv_scores]

#---determining best k (min. MSE)---
optimal_k = ks[MSE.index(min(MSE))]
print(f"The optimal number of neighbors is {optimal_k}")

#---plot misclassification error vs k---
plt.plot(ks, MSE)
plt.xlabel('Number of Neighbors K')
plt.ylabel('Misclassification Error')
plt.show()
```

The preceding code snippet prints out the following:

```
The optimal number of neighbors is 13
```

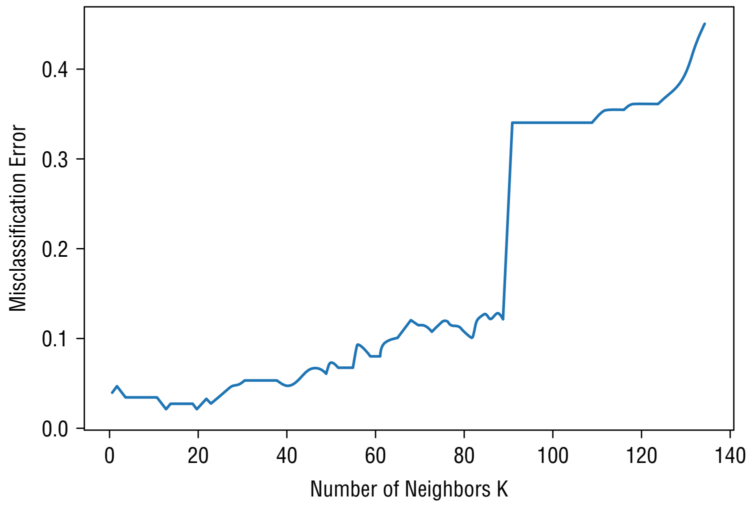

Figure 9.9: The chart of miscalculations for each k

Figure 9.10 shows the classification when k = 13.

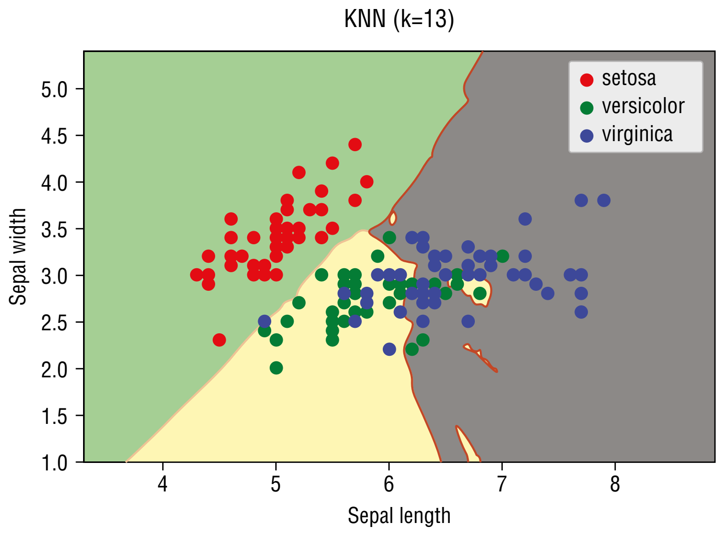

Figure 9.10: The optimal value of k at 13

Summary

Of the four algorithms that we have discussed in this book, KNN is considered one of the most straightforward. In this chapter, you learned how KNN works and how to derive the optimal k that minimizes the miscalculation of errors.

In the next chapter, you will learn a new type of algorithm—unsupervised learning. You will learn how to discover structures in your data by performing clustering using K-Means.