CHAPTER

7

Supervised Learning—Classification Using Logistic Regression

What Is Logistic Regression?

In the previous chapter, you learned about linear regression and how you can use it to predict future values. In this chapter, you will learn another supervised machine learning algorithm—*logistic regression*. Unlike linear regression, logistic regression does not try to predict the value of a numeric variable given a set of inputs. Instead, the output of logistic regression is the probability of a given input point belonging to a specific class. The output of logistic regression always lies in [0,1].

To understand the use of logistic regression, consider the example shown in Figure 7.1. Suppose that you have a dataset containing information about voter income and voting preferences. For this dataset, you can see that low-income voters tend to vote for candidate B, while high-income voters tend to favor candidate A.

With this dataset, you would be very interested in trying to predict which candidate future voters will vote for based on their income level. At first glance, you might be tempted to apply what you have just learned to this problem; that is, using linear regression. Figure 7.2 shows what it looks like when you apply linear regression to this problem.

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

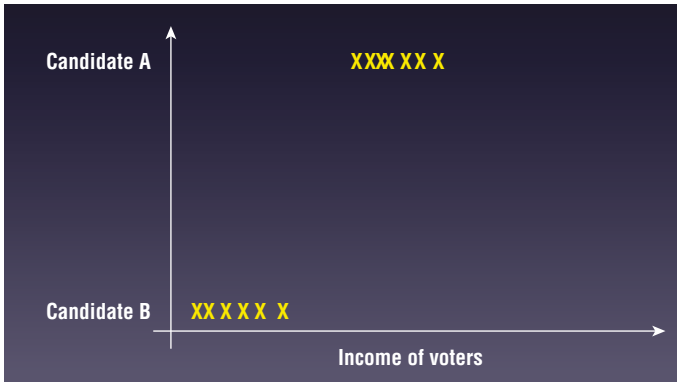

Figure 7.1: Some problems have binary outcomes

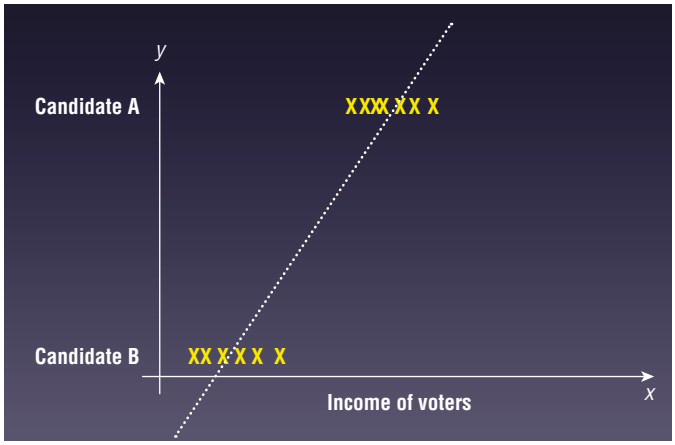

Figure 7.2: Using linear regression to solve the voting preferences problem leads to strange values

The main problem with linear regression is that the predicted value does not always fall within the expected range. Consider the case of a very low-income voter (near to 0), and you can see from the chart that the predicted result is a negative value. What you really want is a way to return the prediction as a value from 0 to 1, where this value represents the probability of an event happening.

Figure 7.3 shows how logistic regression solves this problem. Instead of drawing a straight line cutting through the points, you now use a curved line to try to fit all of the points on the chart.

Using logistic regression, the output will be a value from 0 to 1, where anything less than (or equal to) 0.5 (known as the *threshold*) will be considered as voting for candidate B, and anything greater than 0.5 will be considered as voting for candidate A.

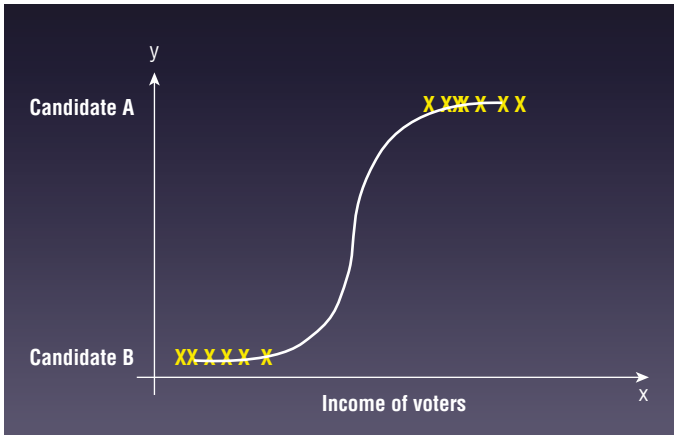

Figure 7.3: Logistic regression predicts the probability of an outcome, rather than a specific value

Understanding Odds

Before we discuss the details of the logistic regression algorithm, we first need to discuss one important term—*odds*. Odds are defined as the ratio of the probability of success to the probability of failure (see Figure 7.4).

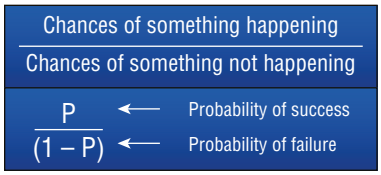

Figure 7.4: How to calculate the odds of an event happening

For example, the odds of landing a head when you flip a coin are 1. This is because you have a 0.5 probability of landing a head and a 0.5 probability of landing a tail. When you say that the odds of landing a head are 1, this means you have a 50 percent chance of landing a head.

But if the coin is rigged in such a way that the probability of landing a head is 0.8 and the probability of landing a tail is 0.2, then the odds of landing a head is 0.8/0.2 = 4. That is, you are 4 times more likely to land a head than a tail. Likewise, the odds of getting a tail are 0.2/0.8 = 0.25.

Logit Function

When you apply the natural logarithm function to the odds, you get the *logit function*. The logit function is the logarithm of the odds (see Figure 7.5).

$$L = \ln\left(\frac{P}{1-P}\right)$$

Figure 7.5: The formula for the logit function

The logit function transfers a variable on (0, 1) into a new variable on (–∞, ∞). To see this relationship, you can use the following code snippet:

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

def logit(x):
    return np.log( x / (1 - x) )

x = np.arange(0.001,0.999, 0.0001)
y = [logit(n) for n in x]
plt.plot(x,y)
plt.xlabel("Probability")
plt.ylabel("Logit - L")
```

Figure 7.6 shows the logit curve plotted using the preceding code snippet.

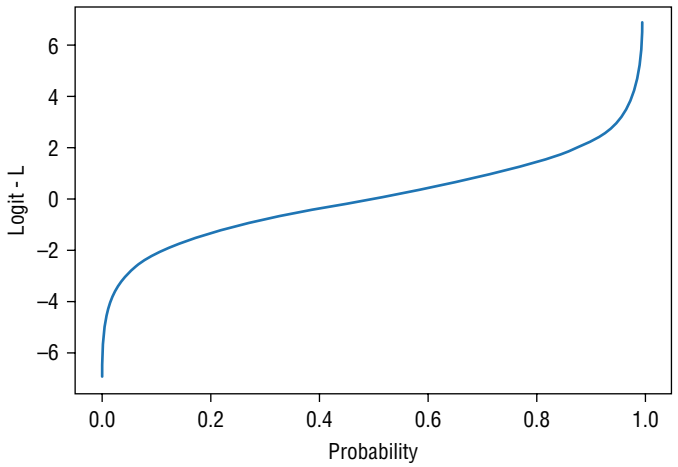

Figure 7.6: The logit curve

Sigmoid Curve

For the logit curve, observe that the x-axis is the probability and the y-axis is the real-number range. For logistic regression, what you really want is a function that maps numbers on the real-number system to the probabilities, which is exactly what you get when you flip the axes of the logit curve (see Figure 7.7).

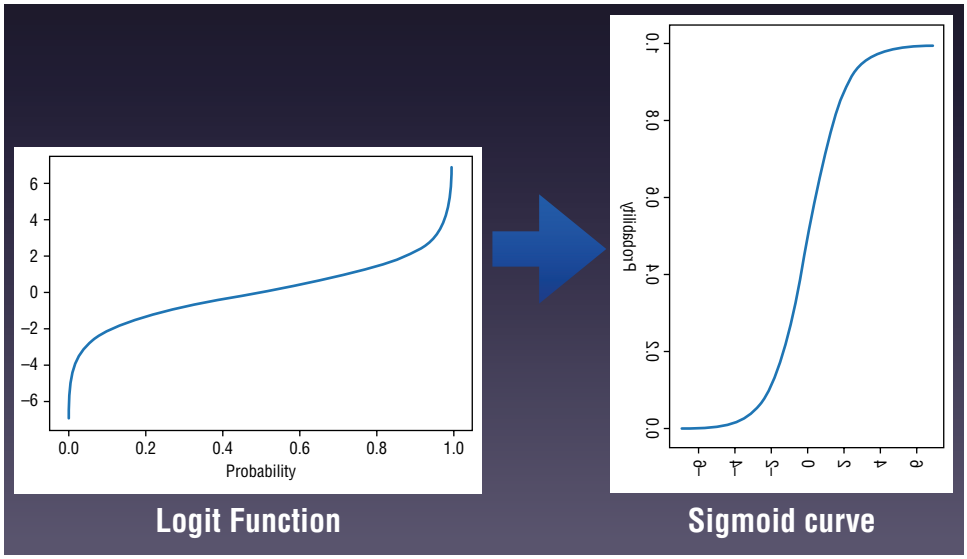

Figure 7.7: Flipping the logit curve into a Sigmoid curve

When you flip the axes, the curve that you get is called the *sigmoid curve*. The sigmoid curve is obtained using the *Sigmoid function*, which is the inverse of the logit function. The Sigmoid function is used to transform values on (–∞, ∞) into numbers on (0, 1). The Sigmoid function is shown in Figure 7.8.

$$P = \frac{1}{(1 + e^{-(L)})}$$

Figure 7.8: The formula for the Sigmoid function

The following code snippet shows how the sigmoid curve is obtained:

```
def sigmoid(x):
    return (1 / (1 + np.exp(-x)))

x = np.arange(-10, 10, 0.0001)
y = [sigmoid(n) for n in x]
plt.plot(x,y)
plt.xlabel("Logit - L")
plt.ylabel("Probability")
```

Figure 7.9 shows the sigmoid curve.

Just like you try to plot a straight line that fits through all of the points in linear regression (as explain in Chapter 5), in logistics regression, we would also like to plot a sigmoid curve that fits through all of the points. Mathematically, this can be expressed by the formula shown in Figure 7.10.

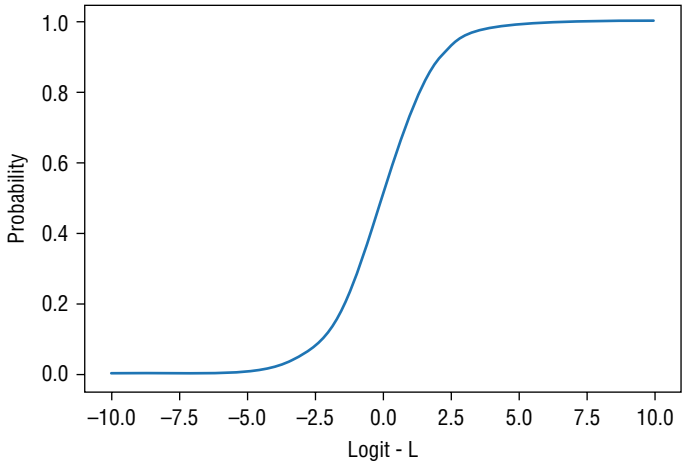

Figure 7.9: The sigmoid curve plotted using matplotlib

$$P = \frac{1}{(1 + e^{-(\beta_0 + x\beta)})}$$

Figure 7.10: Expressing the sigmoid function using the intercept and coefficient

Notice that the key difference between the formula shown in Figure 7.8 and 7.10 is that now L has been replaced by β_0 and $x\beta$. The coefficients β_0 and β are unknown, and they must be estimated based on the available training data using a technique known as *Maximum Likelihood Estimation (MLE)*. In logistics regression, β_0 is known as the intercept and $x\beta$ is known as the coefficient.

Using the Breast Cancer Wisconsin (Diagnostic) Data Set

Scikit-learn ships with the Breast Cancer Wisconsin (Diagnostic) Data Set. It is a classic dataset that is often used to illustrate binary classifications. This dataset contains 30 features, and they are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. The label of the dataset is a binary classification—M for malignant or B for benign. Interested readers can check out more information at `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)`.

Examining the Relationship Between Features

You can load the Breast Cancer dataset by first importing the `datasets` module from `sklearn`. Then use the `load _ breast _ cancer()` function as follows:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
```

Now that the Breast Cancer dataset has been loaded, it is useful to examine the relationships between some of its features.

Plotting the Features in 2D

For a start, let's plot the first two features of the dataset in 2D and examine their relationships. The following code snippet:

- Loads the Breast Cancer dataset
- Copies the first two features of the dataset into a two-dimensional list
- Plots a scatter plot showing the distribution of points for the two features
- Displays malignant growths in red and benign growths in blue

```
%matplotlib inline

import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()

#---copy from dataset into a 2-d list---
X = []
for target in range(2):
    X.append([[], []])
    for i in range(len(cancer.data)):              # target is 0 or 1
        if cancer.target[i] == target:
            X[target][0].append(cancer.data[i][0]) # first feature -
mean radius
            X[target][1].append(cancer.data[i][1]) # second feature —
mean texture

colours = ("r", "b")   # r: malignant, b: benign
fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(111)
for target in range(2):
    ax.scatter(X[target][0],
               X[target][1],
               c=colours[target])

ax.set_xlabel("mean radius")
ax.set_ylabel("mean texture")
plt.show()
```

Figure 7.11 shows the scatter plot of the points.

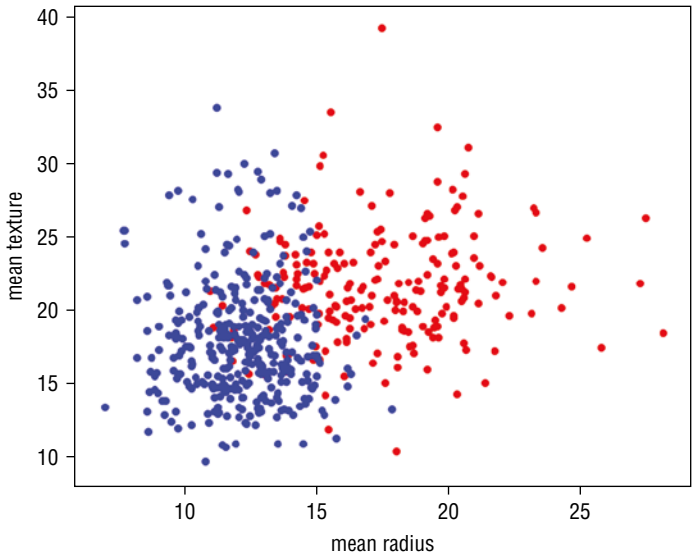

Figure 7.11: The scatter plot showing the relationships between the mean radius and mean texture of the tumor

From this scatter plot, you can gather that as the tumor grows in radius and increases in texture, the more likely that it would be diagnosed as malignant.

Plotting in 3D

In the previous section, you plotted the points based on two features using a scatter plot. It would be interesting to be able to visualize more than two features. In this case, let's try to visualize the relationships between three features. You can use matplotlib to plot a 3D plot. The following code snippet shows how this is done. It is very similar to the code snippet in the previous section, with the additional statements in bold:

```
%matplotlib inline

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()

#---copy from dataset into a 2-d array---
X = []
for target in range(2):
    X.append([[], [], []])
    for i in range(len(cancer.data)):    # target is 0,1
        if cancer.target[i] == target:
```

```
            X[target][0].append(cancer.data[i][0])
            X[target][1].append(cancer.data[i][1])
            X[target][2].append(cancer.data[i][2])

colours = ("r", "b")   # r: malignant, b: benign
fig = plt.figure(figsize=(18,15))
ax = fig.add_subplot(111, projection='3d')
for target in range(2):
    ax.scatter(X[target][0],
               X[target][1],
               X[target][2],
               c=colours[target])

ax.set_xlabel("mean radius")
ax.set_ylabel("mean texture")
ax.set_zlabel("mean perimeter")
plt.show()
```

Instead of plotting using two features, you now have a third feature: mean perimeter. Figure 7.12 shows the 3D plot.

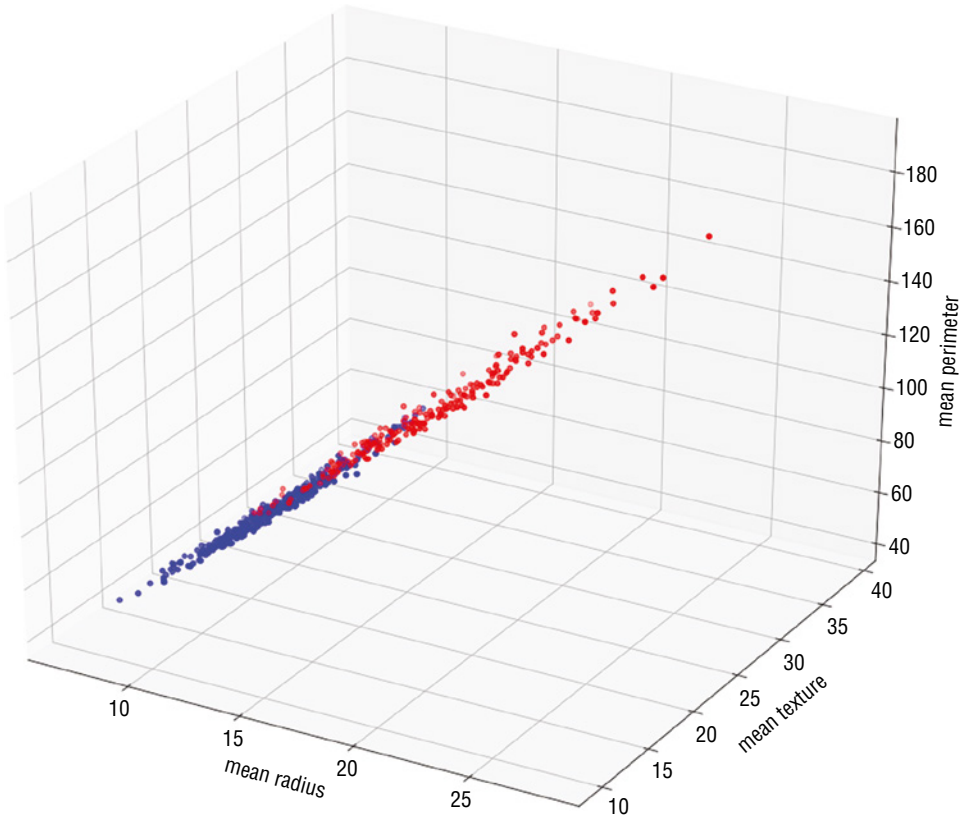

Figure 7.12: Plotting three features using a 3D map

Jupyter Notebook displays the 3D plot statically. As you can see from Figure 7.12, you can't really have a good look at the relationships between the three features. A much better way to display the 3D plot would be to run the preceding code snippet outside of Jupyter Notebook. To do so, save the code snippet (minus the first line containing the statement "`%matplotlib inline`") to a file named, say, `3dplot.py`. Then run the file in Terminal using the `python` command, as follows:

```
$ python 3dplot.py
```

Once you do that, matplotlib will open a separate window to display the 3D plot. Best of all, you will be able to interact with it. Use your mouse to drag the plot, and you are able to visualize the relationships better between the three features. Figure 7.13 gives you a better perspective: as the mean perimeter of the tumor growth increases, the chance of the growth being malignant also increases.

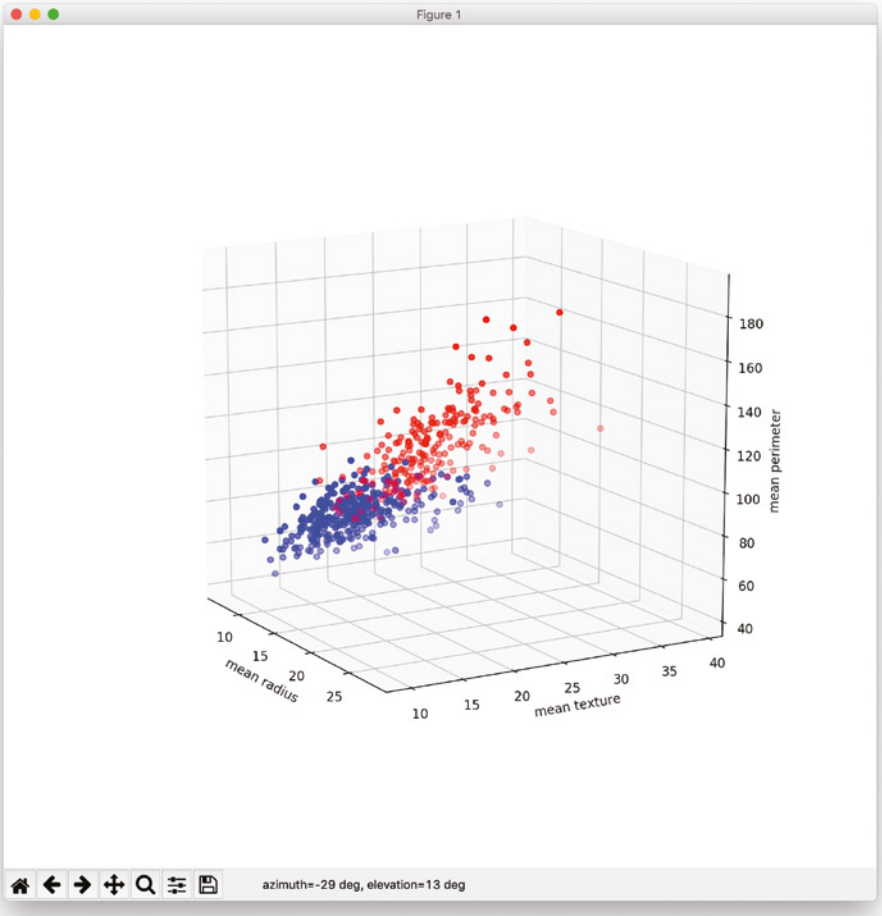

Figure 7.13: You can interact with the 3D plot when you run the application outside of Jupyter Notebook

Training Using One Feature

Let's now use logistic regression to try to predict if a tumor is cancerous. To get started, let's use only the first feature of the dataset: mean radius. The following code snippet plots a scatter plot showing if a tumor is malignant or benign based on the mean radius:

```
%matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()    # Load dataset
x = cancer.data[:,0]             # mean radius
y = cancer.target                # 0: malignant, 1: benign
colors = {0:'red', 1:'blue'}     # 0: malignant, 1: benign

plt.scatter(x,y,
            facecolors='none',
            edgecolors=pd.DataFrame(cancer.target)[0].apply(lambda x:
colors[x]),
            cmap=colors)

plt.xlabel("mean radius")
plt.ylabel("Result")

red   = mpatches.Patch(color='red',   label='malignant')
blue  = mpatches.Patch(color='blue',  label='benign')

plt.legend(handles=[red, blue], loc=1)
```

Figure 7.14 shows the scatter plot.

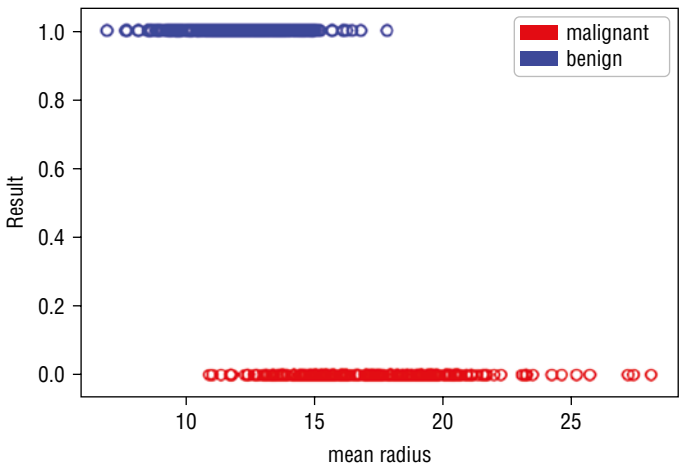

Figure 7.14: Plotting a scatter plot based on one feature

As you can see, this is a good opportunity to use logistic regression to predict if a tumor is cancerous. You could try to plot an "s" curve (albeit flipped horizontally).

Finding the Intercept and Coefficient

Scikit-learn comes with the `LogisticRegression` class that allows you to apply logistic regression to train a model. Thus, in this example, you are going to train a model using the first feature of the dataset:

```
from sklearn import linear_model
import numpy as np

log_regress = linear_model.LogisticRegression()

#---train the model---
log_regress.fit(X = np.array(x).reshape(len(x),1),
                y = y)

#---print trained model intercept---
print(log_regress.intercept_)     # [ 8.19393897]

#---print trained model coefficients---
print(log_regress.coef_)          # [[-0.54291739]]
```

Once the model is trained, what we are most interested in at this point is the intercept and coefficient. If you recall from the formula in Figure 7.10, the intercept is β_0 and the coefficient is $x\beta$. Knowing these two values allows us to plot the sigmoid curve that tries to fit the points on the chart.

Plotting the Sigmoid Curve

With the values of β_0 and $x\beta$ obtained, you can now plot the sigmoid curve using the following code snippet:

```
def sigmoid(x):
    return (1 / (1 +
        np.exp(-(log_regress.intercept_[0] +
        (log_regress.coef_[0][0] * x)))))

x1 = np.arange(0, 30, 0.01)
y1 = [sigmoid(n) for n in x1]

plt.scatter(x,y,
    facecolors='none',
```

```
    edgecolors=pd.DataFrame(cancer.target)[0].apply(lambda x:
colors[x]),
    cmap=colors)

plt.plot(x1,y1)
plt.xlabel("mean radius")
plt.ylabel("Probability")
```

Figure 7.15 shows the sigmoid curve.

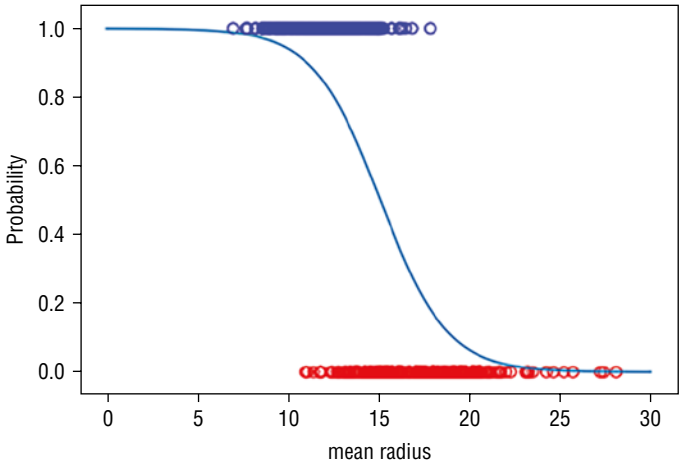

Figure 7.15: The sigmoid curve fitting to the two sets of points

Making Predictions

Using the trained model, let's try to make some predictions. Let's try to predict the result if the mean radius is 20:

```
print(log_regress.predict_proba(20)) # [[0.93489354 0.06510646]]
print(log_regress.predict(20)[0])    # 0
```

As you can see from the output, the `predict _ proba()` function in the first statement returns a two-dimensional array. The result of 0.93489354 indicates the probability that the prediction is 0 (malignant) while the result of 0.06510646 indicates the probability that the prediction is 1. Based on the default *threshold* of 0.5, the prediction is that the tumor is malignant (value of 0), since its predicted probability (0.93489354) of 0 is more than 0.5.

The `predict()` function in the second statement returns the class that the result lies in (which in this case can be a 0 or 1). The result of 0 indicates that

the prediction is that the tumor is malignant. Try another example with the mean radius of 8 this time:

```
print(log_regress.predict_proba(8))  # [[0.02082411 0.97917589]]
print(log_regress.predict(8)[0])     # 1
```

As you can see from the result, the prediction is that the tumor is benign.

Training the Model Using All Features

In the previous section, you specifically trained the model using one feature. Let's now try to train the model using all of the features and then see how well it can accurately perform the prediction.

First, load the dataset:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()   # Load dataset
```

Instead of training the model using all of the rows in the dataset, you are going to split it into two sets, one for training and one for testing. To do so, you use the `train _ test _ split()` function. This function allows you to split your data into random train and test subsets. The following code snippet splits the dataset into a 75 percent training and 25 percent testing set:

```
from sklearn.model_selection import train_test_split
train_set, test_set, train_labels, test_labels = train_test_split(
                              cancer.data,               # features
                              cancer.target,             # labels
                              test_size = 0.25,          # split ratio
                              random_state = 1,          # set random
seed
                              stratify = cancer.target)  # randomize
based on labels
```

Figure 7.16 shows how the dataset is split. The `random _ state` parameter of the `train _ test _ split()` function specifies the seed used by the random number generator. If this is not specified, every time you run this function you will get a different training and testing set. The `stratify` parameter allows you to specify which column (feature/label) to use so that the split is proportionate. For example, if the column specified is a categorical variable with 80 percent 0s and 20 percent 1s, then the training and test sets would each have 80 percent of 0s and 20 percent of 1s.

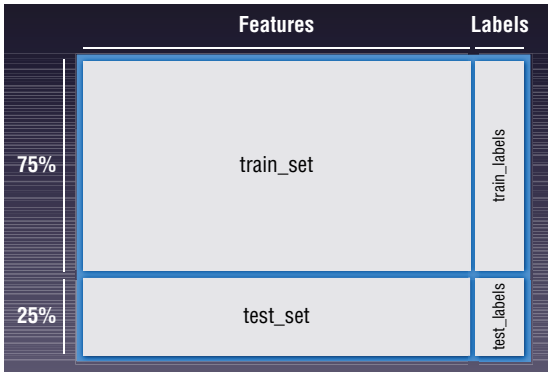

Figure 7.16: Splitting the dataset into training and test sets

Once the dataset is split, it is now time to train the model. The following code snippet trains the model using logistic regression:

```
from sklearn import linear_model
x = train_set[:,0:30]          # mean radius
y = train_labels               # 0: malignant, 1: benign
log_regress = linear_model.LogisticRegression()
log_regress.fit(X = x,
                y = y)
```

In this example, we are training it with all of the 30 features in the dataset. When the training is done, let's print out the intercept and model coefficients:

```
print(log_regress.intercept_)      #
print(log_regress.coef_)           #
```

The following output shows the intercept and coefficients:

```
[0.34525124]
[[ 1.80079054e+00  2.55566824e-01 -3.75898452e-02 -5.88407941e-03
  -9.57624689e-02 -3.16671611e-01 -5.06608094e-01 -2.53148889e-01
  -2.26083101e-01 -1.03685977e-02  4.10103139e-03  9.75976632e-01
   2.02769521e-01 -1.22268760e-01 -8.25384020e-03 -1.41322029e-02
  -5.49980366e-02 -3.32935262e-02 -3.05606774e-02  1.09660157e-04
   1.62895414e+00 -4.34854352e-01 -1.50305237e-01 -2.32871932e-02
  -1.94311394e-01 -9.91201314e-01 -1.42852648e+00 -5.40594994e-01
  -6.28475690e-01 -9.04653541e-02]]
```

Because we have trained the model using 30 features, there are 30 coefficients.

Testing the Model

It's time to make a prediction. The following code snippet uses the test set and feeds it into the model to obtain the predictions:

```
import pandas as pd

#---get the predicted probablities and convert into a dataframe---
preds_prob = pd.DataFrame(log_regress.predict_proba(X=test_set))

#---assign column names to prediction---
preds_prob.columns = ["Malignant", "Benign"]

#---get the predicted class labels---
preds = log_regress.predict(X=test_set)
preds_class = pd.DataFrame(preds)
preds_class.columns = ["Prediction"]

#---actual diagnosis---
original_result = pd.DataFrame(test_labels)
original_result.columns = ["Original Result"]

#---merge the three dataframes into one---
result = pd.concat([preds_prob, preds_class, original_result], axis=1)
print(result.head())
```

The results of the predictions are then printed out. The predictions and original diagnosis are displayed side-by-side for easy comparison:

```
Malignant        Benign  Prediction  Original Result
0   0.999812  1.883317e-04           0                0
1   0.998356  1.643777e-03           0                0
2   0.057992  9.420079e-01           1                1
3   1.000000  9.695339e-08           0                0
4   0.207227  7.927725e-01           1                0
```

Getting the Confusion Matrix

While it is useful to print out the predictions together with the original diagnosis from the test set, it does not give you a clear picture of how good the model is in predicting if a tumor is cancerous. A more scientific way would be to use the *confusion matrix*. The confusion matrix shows the number of actual and predicted labels and how many of them are classified correctly. You can use Pandas's `crosstab()` function to print out the confusion matrix:

```
#---generate table of predictions vs actual---
print("---Confusion Matrix---")
print(pd.crosstab(preds, test_labels))
```

The `crosstab()` function computes a simple cross-tabulation of two factors. The preceding code snippet prints out the following:

```
---Confusion Matrix---
col_0    0   1
row_0
0       48   3
1        5  87
```

The output is interpreted as shown in Figure 7.17.

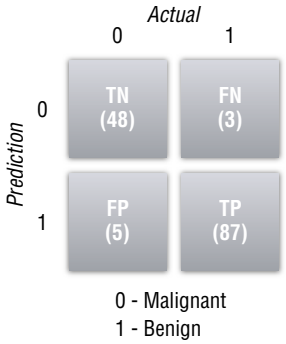

Figure 7.17: The confusion matrix for the prediction

The columns represent the actual diagnosis (0 for malignant and 1 for benign). The rows represent the prediction. Each individual box represents one of the following:

- *True Positive (TP)*: The model correctly predicts the outcome as positive. In this example, the number of TP (87) indicates the number of correct predictions that a tumor is benign.
- *True Negative (TN)*: The model correctly predicts the outcome as negative. In this example, tumors were correctly predicted to be malignant.
- *False Positive (FP)*: The model incorrectly predicted the outcome as positive, but the actual result is negative. In this example, it means that the tumor is actually malignant, but the model predicted the tumor to be benign.
- *False Negative (FN)*: The model incorrectly predicted the outcome as negative, but the actual result is positive. In this example, it means that the tumor is actually benign, but the model predicted the tumor to be malignant.

This set of numbers is known as the *confusion matrix*.

Besides using the `crosstab()` function, you can also use the `confusion_matrix()` function to print out the confusion matrix:

```
from sklearn import metrics
#---view the confusion matrix---
print(metrics.confusion_matrix(y_true = test_labels,  # True labels
                               y_pred = preds))       # Predicted labels
```

Note that the output is switched for the rows and columns.

```
[[48  5]
 [ 3 87]]
```

Computing Accuracy, Recall, Precision, and Other Metrics

Based on the confusion matrix, you can calculate the following metrics:

- ***Accuracy*:** This is defined as the sum of all correct predictions divided by the total number of predictions, or mathematically:

  $$(TP / TN) / (TP + TN + FP + FN)$$

- This metric is easy to understand. After all, if the model correctly predicts 99 out of 100 samples, the accuracy is 0.99, which would be very impressive in the real world. But consider the following situation: Imagine that you're trying to predict the failure of equipment based on the sample data. Out of 1,000 samples, only three are defective. If you use a dumb algorithm that always returns negative (meaning no failure) for all results, then the accuracy is 997/1000, which is 0.997. This is very impressive, but does this mean it's a good algorithm? No. If there are 500 defective items in the dataset of 1,000 items, then the accuracy metric immediately indicates the flaw of the algorithm. In short, accuracy works best with evenly distributed data points, but it works really badly for a skewed dataset. Figure 7.18 summarizes the formula for accuracy.
- ***Precision*:** This metric is defined to be TP / (TP + FP). This metric is concerned with number of correct positive predictions. You can think of precision as "of those predicted to be positive, how many were actually predicted correctly?" Figure 7.19 summarizes the formula for precision.
- ***Recall* (also known as *True Positive Rate (TPR)*):** This metric is defined to be TP / (TP + FN). This metric is concerned with the number of correctly predicted positive events. You can think of recall as "of those positive events, how many were predicted correctly?" Figure 7.20 summarizes the formula for recall.

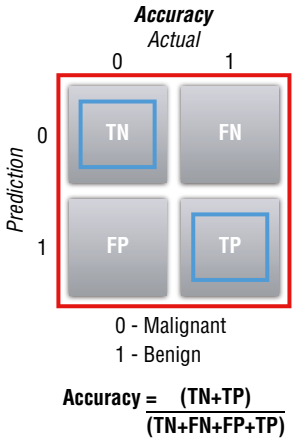

Figure 7.18: Formula for calculating accuracy

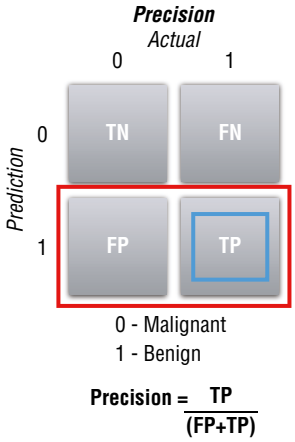

Figure 7.19: Formula for calculating precision

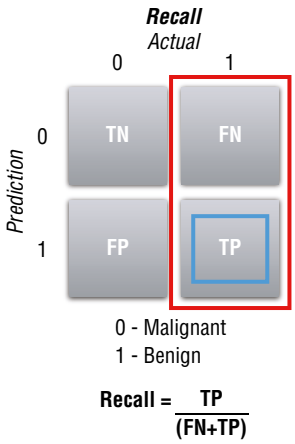

Figure 7.20: Formula for calculating recall

- ***F1 Score*:** This metric is defined to be 2 * (precision * recall) / (precision + recall). This is known as the *harmonic mean of precision and recall*, and it is a good way to summarize the evaluation of the algorithm in a single number.
- ***False Positive Rate (FPR)*:** This metric is defined to be FP / (FP+TN). FPR corresponds to the proportion of negative data points that are mistakenly considered as positive, with respect to all negative data points. In other words, the higher FPR, the more negative data points you'll misclassify.

The concept of precision and recall may not be apparent immediately, but if you consider the following scenario, it will be much clearer. Consider the case of breast cancer diagnosis. If a malignant tumor is represented as negative and a benign tumor is represented as positive, then:

- If the precision or recall is high, it means that more patients with benign tumors are diagnosed correctly, which indicates that the algorithm is good.
- If the precision is low, it means that more patients with malignant tumors are diagnosed as benign.
- If the recall is low, it means that more patients with benign tumors are diagnosed as malignant.

For the last two points, having a low precision is more serious than a low recall (although wrongfully diagnosed as having breast cancer when you do not have it will likely result in unnecessary treatment and mental anguish) because it causes the patient to miss treatment and potentially causes death. Hence, for cases like diagnosing breast cancer, it's important to consider both the precision and recall metrics when evaluating the effectiveness of an ML algorithm.

To get the accuracy of the model, you can use the `score()` function of the model:

```
#---get the accuracy of the prediction---
print("---Accuracy---")
print(log_regress.score(X = test_set ,
                        y = test_labels))
```

You should see the following result:

```
---Accuracy---
0.9440559440559441
```

To get the precision, recall, and F1-score of the model, use the `classification_report()` function of the `metrics` module:

```
# View summary of common classification metrics
print("---Metrices---")
```

```
print(metrics.classification_report(
      y_true = test_labels,
      y_pred = preds))
```

You will see the following results:

```
---Metrices---
             precision    recall  f1-score   support

          0       0.94      0.91      0.92        53
          1       0.95      0.97      0.96        90

avg / total       0.94      0.94      0.94       143
```

Receiver Operating Characteristic (ROC) Curve

With so many metrics available, what is an easy way to examine the effectiveness of an algorithm? One way would be to plot a curve known as the *Receiver Operating Characteristic (ROC) curve*. The ROC curve is created by plotting the TPR against the FPR at various threshold settings.

So how does it work? Let's run through a simple example. Using the existing project that you have been working on, you have derived the confusion matrix based on the default threshold of 0.5 (meaning that all of those predicted probabilities less than or equal to 0.5 belong to one class, while those greater than 0.5 belong to another class). Using this confusion matrix, you then find the recall, precision, and subsequently FPR and TPR. Once the FPR and TPR are found, you can plot the point on the chart, as shown in Figure 7.21.

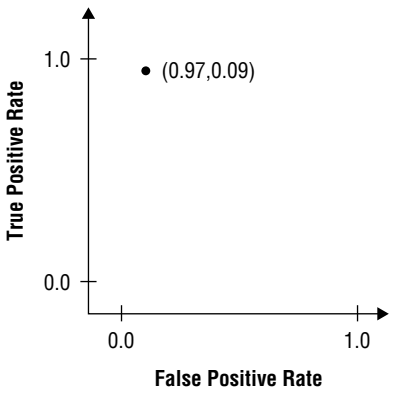

Figure 7.21: The point at threshold 0.5

Then you regenerate the confusion matrix for a threshold of 0, and recalculate the recall, precision, FPR, and TPR. Using the new FPR and TPR, you plot

another point on the chart. You then repeat this process for thresholds of 0.1, 0.2, 0.3, and so on, all the way to 1.0.

At threshold 0, in order for a tumor to be classified as benign (1), the predicted probability must be greater than 0. Hence, all of the predictions would be classified as benign (1). Figure 7.22 shows how to calculate the TPR and FPR. For a threshold of 0, both the TPR and FPR are 1.

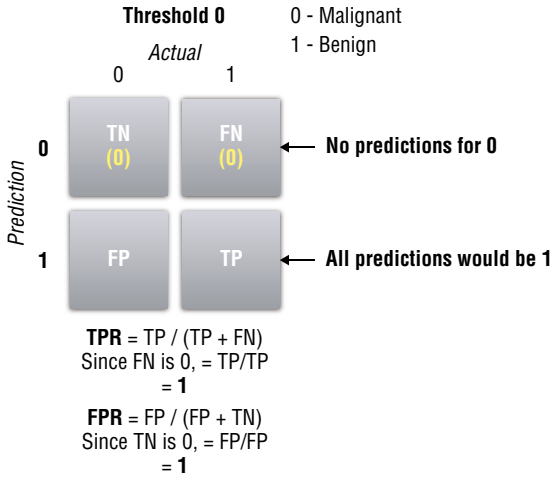

Figure 7.22: The value of TPR and FPR for threshold 0

At threshold 1.0, in order for a tumor to be classified as benign (1), the predicted probability must be equal to exactly 1. Hence, all of the predictions would be classified as malignant (0). Figure 7.23 shows how to calculate the TPR and FPR when the threshold is 1.0. For a threshold of 1.0, both the TPR and FPR are 0.

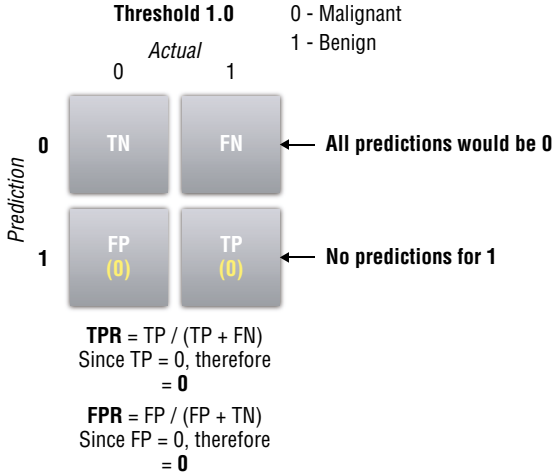

Figure 7.23: The value of TPR and FPR for threshold 1

We can now plot two more points on our chart (see Figure 7.24).

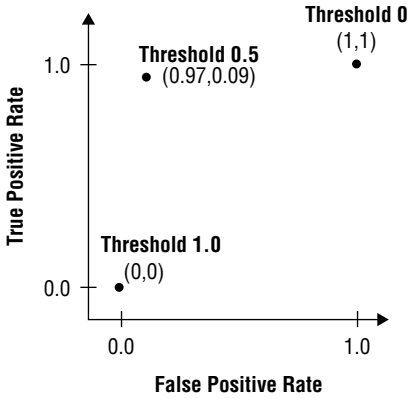

Figure 7.24: Plotting the points for threshold 0, 0.5, and 1.0.

You then calculate the metrics for the other threshold values. Calculating all of the metrics based on different threshold values is a very tedious process. Fortunately, Scikit-learn has the `roc _ curve()` function, which will calculate the FPR and TPR automatically for you based on the supplied test labels and predicted probabilities:

```
from sklearn.metrics import roc_curve, auc

#---find the predicted probabilities using the test set
probs = log_regress.predict_proba(test_set)
preds = probs[:,1]

#---find the FPR, TPR, and threshold---
fpr, tpr, threshold = roc_curve(test_labels, preds)
```

The `roc _ curve()` function returns a tuple containing the FPR, TPR, and threshold. You can print them out to see the values:

```
print(fpr)
print(tpr)
print(threshold)
```

You should see the following:

```
[ 0.          0.          0.01886792  0.01886792  0.03773585  0.03773585
  0.09433962  0.09433962  0.11320755  0.11320755  0.18867925  0.18867925
  1.        ]

[ 0.01111111  0.88888889  0.88888889  0.91111111  0.91111111  0.94444444
  0.94444444  0.96666667  0.96666667  0.98888889  0.98888889  1.
  1.        ]

[  9.99991063e-01   9.36998422e-01   9.17998921e-01   9.03158173e-01
   8.58481867e-01   8.48217940e-01   5.43424515e-01   5.26248925e-01
```

```
3.72174142e-01   2.71134211e-01   1.21486104e-01   1.18614069e-01
1.31142589e-21]
```

As you can see from the output, the threshold starts at 0.99999 (9.99e-01) and goes down to 1.311e-21.

Plotting the ROC and Finding the Area Under the Curve (AUC)

To plot the ROC, you can use matplotlib to plot a line chart using the values stored in the *fpr* and *tpr* variables. You can use the `auc()` function to find the area under the ROC:

```
#---find the area under the curve---
roc_auc = auc(fpr, tpr)

import matplotlib.pyplot as plt
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc)
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate (TPR)')
plt.xlabel('False Positive Rate (FPR)')
plt.title('Receiver Operating Characteristic (ROC)')
plt.legend(loc = 'lower right')
plt.show()
```

The area under an ROC curve is a measure of the usefulness of a test in general, where a greater area means a more useful test and the areas under ROC curves are used to compare the usefulness of tests. Generally, aim for the algorithm with the highest AUC.

Figure 7.25 shows the ROC curve as well as the AUC.

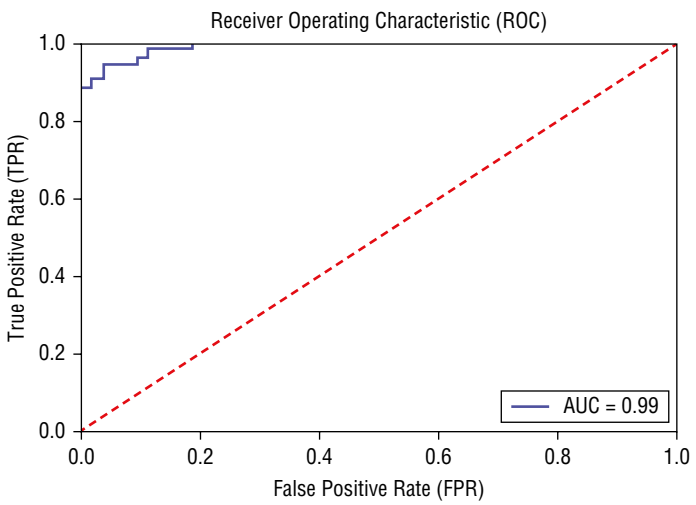

Figure 7.25: Plotting the ROC curve and calculating the AUC

Summary

In this chapter, you learned about another supervised machine learning algorithm—logistics regression. You first learned about the logit function and how to transform it into a sigmoid function. You then applied the logistic regression to the breast cancer dataset and used it to predict if a tumor is malignant or benign. More importantly, this chapter discussed some of the metrics that are useful in determining the effectiveness of a machine learning algorithm. In addition, you learned about what an ROC curve is, how to plot it, and how to calculate the area under the curve.