```
recall    = TP/[TP + FN]
```

Accuracy can be an unreliable metric because it yields misleading results in unbalanced datasets. When the number of observations in different classes are significantly different, it gives equal importance to both false positive and false negative classifications. For example, declaring cancer as benign is worse than incorrectly informing patients that they are suffering from cancer. Unfortunately, accuracy won't differentiate between these two cases.

Keep in mind that the confusion matrix can be an nxn matrix and not just a 2x2 matrix. For example, if a class has 5 possible values, then the confusion matrix is a 5x5 matrix, and the numbers on the main diagonal are the *true positive* results.

5.8.4 The ROC Curve

The receiver operating characteristic (ROC) curve is a curve that plots the the true positive rate (TPR; i.e., the recall) against the false positive rate (FPR). Note that the the true negative rate (TNR) is also called the specificity.

The following link contains a Python code sample using SKLearn and the Iris dataset, and also code for plotting the ROC:

https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html

The following link contains an assortment of Python code samples for plotting the ROC:

https://stackoverflow.com/questions/25009284/how-to-plot-roc-curve-in-python

5.9 Other Useful Statistical Terms

Machine learning relies on a number of statistical quantities in order to assess the validity of a model, some of which are listed here:

- RSS
- TSS
- R^2
- F1 score
- p-value

The definitions of `RSS`, `TSS`, and `R^2` are shown in the following, where `y^` is the y-coordinate of a point on a best-fitting line and `y_` is the mean of the y-values of the points in the dataset:

```
➡ RSS = sum of squares of residuals (y - y^)**2
➡ TSS = toal sum of squares         (y - y_)**2
R^2 = 1 - RSS/TSS
```

5.9.1 What Is an F1 score?

The F1 score is a measure of the accuracy of a test, and it's defined as the harmonic mean of precision and recall. Here are the relevant formulas, where `p` is the precision and `r` is the recall:

```
p = (# of correct positive results)/(# of all positive
results)
r = (# of correct positive results)/(# of all relevant
samples)

F1-score  = 1/[((1/r) + (1/p))/2]
          = 2*[p*r]/[p+r]
```

The best value of an F1 score is 0 and the worse value is 0. Keep in mind that an F1 score tends to be used for categorical classification problems, whereas the R^2 value is typically for regression tasks (such as linear regression).

5.9.2 What Is a p-value?

The p-value is used to reject the null hypothesis if the p-value is small enough (< 0.005) which indicates a higher significance. Recall that the null hypothesis states that there is no correlation between a dependent variable (such as y) and an independent variable (such as x). The threshold value for p is typically 1% or 5%.

There is no straightforward formula for calculating p-values, which are values that are always between 0 and 1. In fact, p-values are statistical quantities to evaluate the so-called *null hypothesis*, and they are calculated by means of p-value tables or via spreadsheet/statistical software.

5.10 What is Linear Regression?

The goal of linear regression is to find the best-fitting line that represents a dataset. Keep in mind two key points. First, the best-fitting line does

not necessarily pass through all (or even most of) the points in the dataset. The purpose of a best-fitting line is to minimize the vertical distance of that line from the points in dataset. Second, linear regression does not determine the best-fitting polynomial: the latter involves finding a higher-degree polynomial that passes through many of the points in a dataset.

Moreover, a dataset in the plane can contain two or more points that lie on the same *vertical* line, which is to say that those points have the same x value. However, a function *cannot* pass through such a pair of points: if two points `(x1,y1)` and `(x2,y2)` have the same x value then they must have the same y value (i.e., `y1=y2`). On the other hand, a function can have two or more points that lie on the same *horizontal* line.

Now consider a scatter plot with many points in the plane that are sort of *clustered* in an elongated cloud-like shape: a best-fitting line will probably intersect only limited number of points (in fact, a best-fitting line might not intersect *any* of the points).

One other scenario to keep in mind: suppose a dataset contains a set of points that lie on the same line. For instance, let's say the x values are in the set `{1,2,3,...,10}` and the y values are in the set `{2,4,6,...,20}`. Then the equation of the best-fitting line is `y=2*x+0`. In this scenario, all the points are *collinear*, which is to say that they lie on the same line.

5.10.1 Linear Regression versus Curve-Fitting

Suppose a dataset consists of n data points of the form (x, y), and no two of those data points have the same `x` value. Then according to a well-known result in mathematics, there is a polynomial of degree less than or equal to n-1 that passes through those n points (if you are really interested, you can find a mathematical proof of this statement in online articles). For example, a line is a polynomial of degree one and it can intersect any pair of nonvertical points in the plane. For any triple of points (that are not all on the same line) in the plane, there is a quadratic equation that passes through those points.

In addition, sometimes a lower degree polynomial is available. For instance, consider the set of 100 points in which the `x` value equals the `y` value: in this case, the line `y = x` (which is a polynomial of degree one) passes through all 100 points.

However, keep in mind that the extent to which a line represents a set of points in the plane depends on how closely those points can be approximated by a line, which is measured by the *variance* of the points (the variance

is a statistical quantity). The more collinear the points, the smaller the variance; conversely, the more spread out the points are, the larger the variance.

5.10.2 When Are Solutions Exact Values?

Although statistics-based solutions provide closed-form solutions for linear regression, neural networks provide *approximate* solutions. This is due to the fact that machine learning algorithms for linear regression involve a sequence of approximations that converges to optimal values, which means that machine learning algorithms produce estimates of the exact values. For example, the slope `m` and y-intercept `b` of a best-fitting line for a set of points a 2D plane have a closed-form solution in statistics, but they can only be approximated via machine learning algorithms (exceptions do exist, but they are rare situations).

Keep in mind that even though a closed-form solution for traditional linear regression provides an exact value for both `m` and `b`, sometimes you can only use an approximation of the exact value. For instance, suppose that the slope m of a best-fitting line equals the square root of 3 and the y-intercept b is the square root of 2. If you plan to use these values in source code, you can only work with an approximation of these two numbers. In the same scenario, a neural network computes approximations for `m` and `b`, regardless of whether or not the exact values for `m` and `b` are irrational, rational, or integer values. However, machine learning algorithms are better suited for complex, nonlinear, multi-dimensional datasets, which is beyond the capacity of linear regression.

As a simple example, suppose that the closed form solution for a linear regression problem produces integer or rational values for both `m` and `b`. Specifically, let's suppose that a closed form solution yields the values 2.0 and 1.0 for the slope and y-intercept, respectively, of a best-fitting line. The equation of the line looks like this:

```
y = 2.0 * x + 1.0
```

However, the corresponding solution from training a neural network might produce the values 2.0001 and 0.9997 for the slope m and the y-intercept `b`, respectively, as the values of `m` and `b` for a best-fitting line. Always keep this point in mind, especially when you are training a neural network.

5.10.3 What is Multivariate Analysis?

Multivariate analysis generalizes the equation of a line in the Euclidean plane to higher dimensions, and it's called a *hyperplane* instead of a line. The generalized equation has the following form:

```
y = w1*x1 + w2*x2 + . . . + wn*xn + b
```

In the case of 2D linear regression, you only need to find the value of the slope (`m`) and the y-intercept (`b`), whereas in multivariate analysis you need to find the values for `w1, w2, . . ., wn`. Note that multivariate analysis is a term from statistics, and in machine learning it's often referred to as *generalized linear regression*.

Keep in mind that most of the code samples in this book that pertain to linear regression involve 2D points in the Euclidean plane.

5.11 Other Types of Regression

Linear regression finds the best-fitting line that represents a dataset, but what happens if a line in the plane is not a good fit for the dataset? This is a relevant question when you work with datasets.

Some alternatives to linear regression include quadratic equations, cubic equations, or higher-degree polynomials. However, these alternatives involve trade-offs, as we'll discuss later.

Another possibility is a sort of hybrid approach that involves piece-wise linear functions, which comprises a set of line segments. If contiguous line segments are connected then it's a piece-wise linear continuous function; otherwise it's a piece-wise linear discontinuous function.

Thus, given a set of points in the plane, regression involves addressing the following questions:

1. What type of curve fits the data well? How do we know?

2. Does another type of curve fit the data better?

3. What does "best fit" mean?

One way to check if a line fits the data involves a visual check, but this approach does not work for data points that are higher than two dimensions. Moreover, this is a subjective decision, and some sample datasets are displayed later in this chapter. By visual inspection of a dataset, you might decide that a quadratic or cubic (or even higher degree) polynomial has the potential of being a better fit for the data. However, visual inspection is probably limited to points in a 2D plane or in three dimensions.

Let's defer the nonlinear scenario and let's make the assumption that a line would be a good fit for the data. There is a well-known technique for finding the "best-fitting" line for such a dataset that involves minimizing the mean squared error (MSE) that we'll discuss later in this chapter.

The next section provides a quick review of linear equations in the plane, along with some images that illustrate examples of linear equations.

5.12 Working with Lines in the Plane (optional)

This section contains a short review of lines in the Euclidean plane, so you can skip this section if you are comfortable with this topic. A minor point that's often overlooked is that lines in the Euclidean plane have infinite length. If you select two distinct points of a line, then all the points between those two selected points is a *line segment*. A *ray* is a line that is infinite in one direction: when you select one point as an endpoint, then all the points on one side of the line constitutes a ray.

For example, the points in the plane whose y-coordinate is 0 is a line and also the x-axis, whereas the points between (0,0) and (1,0) on the x-axis form a line segment. In addition, the points on the x-axis that are to the right of (0,0) form a ray, and the points on the x-axis that are to the left of (0,0) also form a ray.

For simplicity and convenience, in this book we'll use the terms "line" and "line segment" interchangeably, and now let's delve into the details of lines in the Euclidean plane. Just in case you're a bit fuzzy on the details, here is the equation of a (nonvertical) line in the Euclidean plane:

```
y = m*x + b
```

The value of `m` is the slope of the line and the value of `b` is the y-intercept (i.e., the place where the line intersects the y-axis).

If need be, you can use a more general equation that can also represent vertical lines, as shown here:

```
a*x + b*y + c = 0
```

However, we won't be working with vertical lines, so we'll stick with the first formula.

Figure 5.1 displays three horizontal lines whose equations (from top to bottom) are `y = 3`, `y = 0`, and `y = -3`, respectively.

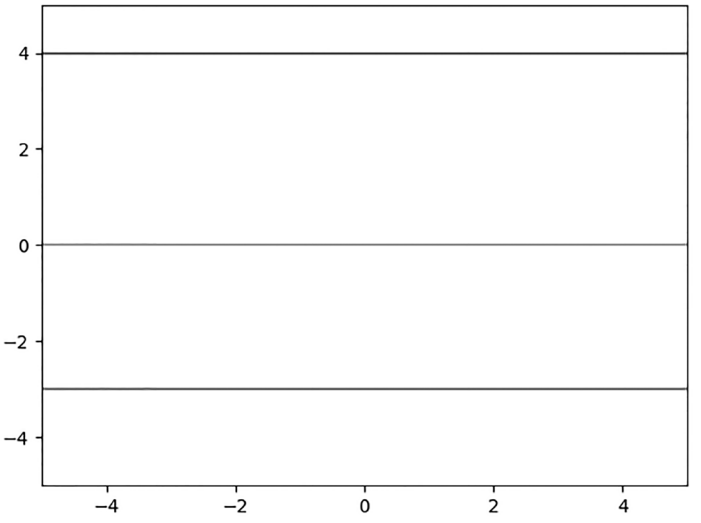

FIGURE 5.1: A graph of three horizontal line segments.

Figure 5.2 displays two slanted lines whose equations are `y = x` and `y = -x`, respectively.

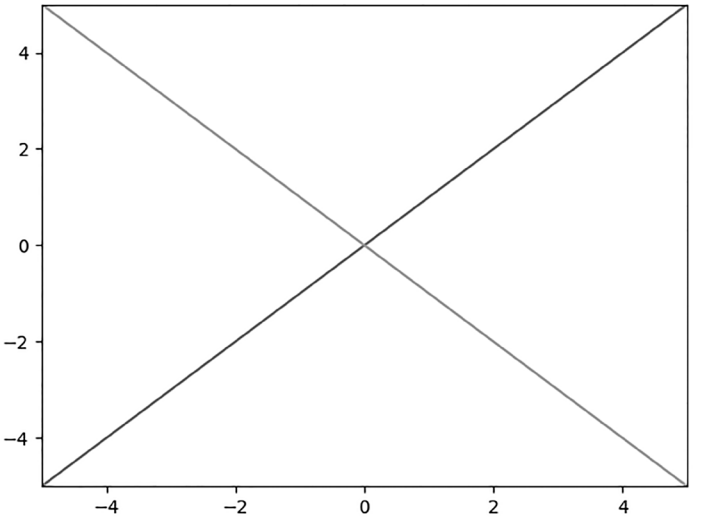

FIGURE 5.2: A graph of two diagonal line segments.

Figure 5.3 displays two slanted parallel lines whose equations are `y = 2*x` and `y = 2*x + 3`, respectively.

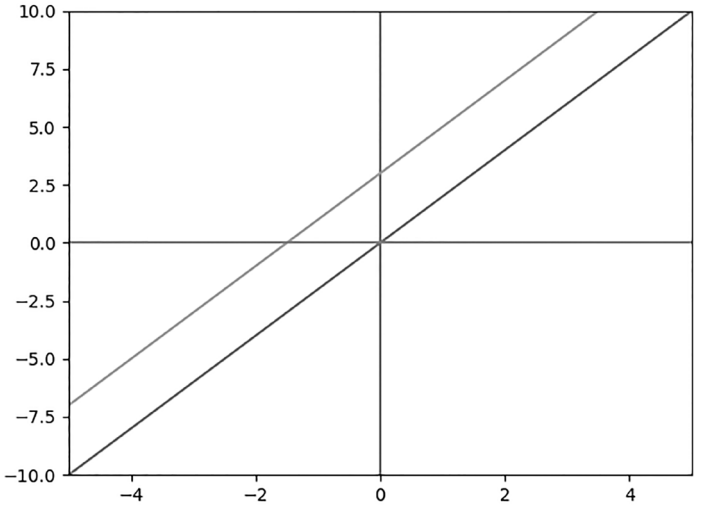

FIGURE 5.3: A graph of two slanted parallel line segments.

Figure 5.4 displays a piece-wise linear graph consisting of connected line segments.

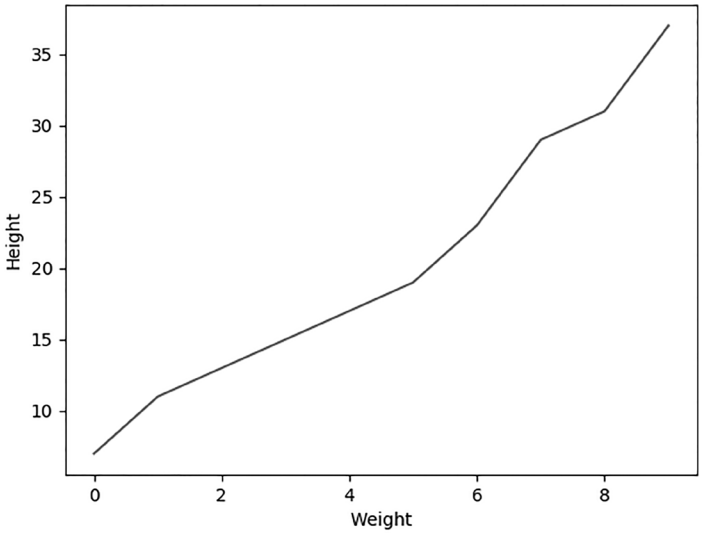

FIGURE 5.4: A piecewise linear graph of line segments.

Now let's turn our attention to generating quasi-random data using a NumPy API, and then we'll plot the data using Matplotlib.

5.13 Scatter Plots with NumPy and Matplotlib (1)

Listing 5.1 displays the contents of `np_plot1.py` that illustrates how to use the `NumPy randn()` API to generate a dataset and then the `scatter()` API in Matplotlib to plot the points in the dataset.

One detail to note is that all the adjacent horizontal values are equally spaced, whereas the vertical values are based on a linear equation plus a "perturbation" value. This *perturbation technique* (which is not a standard term) is used in other code samples in this chapter in order to add a slightly randomized effect when the points are plotted. The advantage of this technique is that the best-fitting values for `m` and `b` are known in advance, and therefore we do not need to guess their values.

Listing 5.1: np_plot1.py

```
import numpy as np
import matplotlib.pyplot as plt

x = np.random.randn(15,1)
y = 2.5*x + 5 + 0.2*np.random.randn(15,1)

print("x:",x)
print("y:",y)

plt.scatter(x,y)
plt.show()
```

Listing 5.1 contains two `import` statements and then initializes the array variable `x` with 15 random numbers between 0 and 1.

Next, the array variable `y` is defined in two parts: the first part is a linear equation `2.5*x + 5` and the second part is a "perturbation" value that is based on a random number. Thus, the array variable `y` simulates a set of values that closely approximate a line segment.

This technique is used in code samples that simulate a line segment, and then the training portion approximates the values of `m` and `b` for the best-fitting line. Obviously we already *know* the equation of the best-fitting line: the purpose of this technique is to compare the trained values for the slope `m` and y-intercept `b` with the known values (which in this case are 2.5 and 5).

A partial output from Listing 5.1 is here:

```
x: [[-1.42736308]
 [ 0.09482338]
 [-0.45071331]
 [ 0.19536304]
 [-0.22295205]
 // values omitted for brevity
y: [[1.12530514]
 [5.05168677]
 [3.93320782]
 [5.49760999]
 [4.46994978]
 // values omitted for brevity
```

Figure 5.5 displays a scatter plot of points based on the values of `x` and `y`.

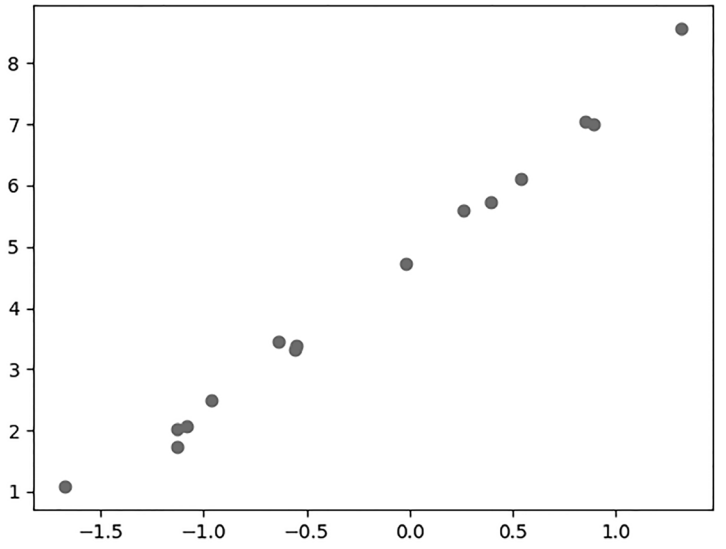

FIGURE 5.5: A scatter plot of points for a line segment.

5.13.1 Why the "Perturbation Technique" is Useful

You already saw how to use the "perturbation technique" and by way of comparison, consider a dataset with the following points that are defined in the Python array variables `X` and `Y`:

```
X = [0,0.12,0.25,0.27,0.38,0.42,0.44,0.55,0.92,1.0]
Y = [0,0.15,0.54,0.51, 0.34,0.1,0.19,0.53,1.0,0.58]
```

If you need to find the best-fitting line for the preceding dataset, how would you guess the values for the slope `m` and the y-intercept `b`? In most cases, you probably cannot guess their values. On the other hand, the "per-

turbation technique" enables you to randomize the points on a line whose value for the slope m (and optionally the value for the y-intercept b) is specified in advance.

Keep in mind that the "perturbation technique" only works when you introduce small random values that do not result in different values for m and b.

5.14 Scatter Plots with NumPy and Matplotlib (2)

The code in Listing 5.1 assigned random values to the variable x, whereas a hard-coded value is assigned to the slope m. The y values are a hard-coded multiple of the x values, plus a random value that is calculated via the "perturbation technique." Hence we do not know the value of the y-intercept b.

In this section the values for trainX are based on the np.linspace() API, and the values for trainY involve the "perturbation technique" that is described in the previous section.

The code in this example simply prints the values for trainX and trainY, which correspond to data points in the Euclidean plane. Listing 5.2 displays the contents of np_plot2.py that illustrates how to simulate a linear dataset in NumpPy.

Listing 5.2: np_plot2.py

```
import numpy as np

  trainX = np.linspace(-1, 1, 11)
trainY = 4*trainX + np.random.randn(*trainX.shape)*0.5

print("trainX: ",trainX)
print("trainY: ",trainY)
```

Listing 5.6 initializes the NumPy array variable trainX via the NumPy linspace() API, followed by the array variable trainY that is defined in two parts. The first part is the linear term 4*trainX and the second part involves the "perturbation technique" that is a randomly generated number. The output from Listing 5.6 is here:

```
trainX:  [-1.  -0.8 -0.6 -0.4 -0.2  0.   0.2  0.4  0.6
0.8  1. ]
trainY:  [-3.60147459 -2.66593108 -2.26491189
```

```
-1.65121314 -0.56454605  0.22746004 0.86830728
1.60673482  2.51151543  3.59573877  3.05506056]
```

The next section contains an example that is similar to Listing 5.2, using the same "perturbation technique" to generate a set of points that approximate a quadratic equation instead of a line segment.

5.15 A Quadratic Scatterplot with NumPy and matplotlib

Listing 5.3 displays the contents of np_plot_quadratic.py that illustrates how to plot a quadratic function in the plane.

Listing 5.3: np_plot_quadratic.py

```
import numpy as np
import matplotlib.pyplot as plt

#see what happens with this set of values:
#x = np.linspace(-5,5,num=100)

x = np.linspace(-5,5,num=100)[:,None]
y = -0.5 + 2.2*x +0.3*x**2 + 2*np.random.randn(100,1)
print("x:",x)

plt.plot(x,y)
plt.show()
```

Listing 5.3 initializes the array variable x with the values that are generated via the np.linspace() API, which in this case is a set of 100 equally spaced decimal numbers between -5 and 5. Notice the snippet [:,None] in the initialization of x, which results in an array of elements, each of which is an array consisting of a single number.

The array variable y is defined in two parts: the first part is a quadratic equation -0.5 + 2.2*x +0.3*x**2 and the second part is a "perturbation" value that is based on a random number (similar to the code in Listing 5.1). Thus, the array variable y simulates a set of values that approximates a quadratic equation. The output from Listing 5.3 is here:

```
x:
[[-5.        ]
 [-4.8989899 ]
 [-4.7979798 ]
 [-4.6969697 ]
 [-4.5959596 ]
```

```
[-4.49494949]
// values omitted for brevity
[ 4.8989899 ]
[ 5.        ]]
```

Figure 5.6 displays a scatter plot of points based on the values of `x` and `y`, which have an approximate shape of a quadratic equation.

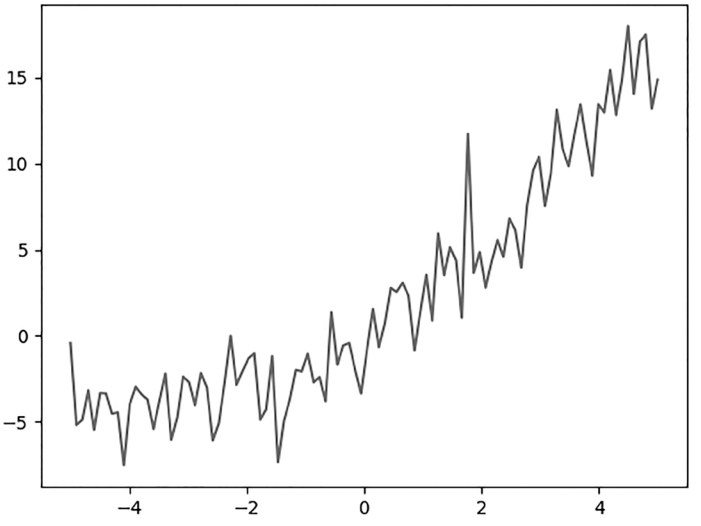

FIGURE 5.6: A scatter plot of points for a quadratic equation.

5.16 The MSE Formula

In plain English, the MSE is the sum of the squares of the difference between an actual y value and the predicted y value, divided by the number of points. Notice that the predicted y value is the y value that each point would have if that point were actually on the best-fitting line.

Although the MSE is popular for linear regression, there are other error types available, some of which are discussed briefly in the next section.

5.16.1 A List of Error Types

Although we will only discuss MSE for linear regression in this book, there are other types of formulas that you can use for linear regression, some of which are listed here:

- MSE
- RMSE

- RMSPROP
- MAE

The MSE is the basis for the preceding error types. For example, RMSE is *root mean squared error*, which is the square root of MSE.

On the other hand, MAE is *mean absolute error*, which is the sum of *the absolute value of the differences of the y terms* (*not* the square of the differences of the y terms), which is then divided by the number of terms.

The RMSProp optimizer utilizes the magnitude of recent gradients to normalize the gradients. Specifically, RMSProp maintain a moving average over the root mean squared (RMS) gradients, and then divides that term by the current gradient.

Although it's easier to compute the derivative of MSE, it's also true that MSE is more susceptible to outliers, whereas MAE is less susceptible to outliers. The reason is simple: a squared term can be significantly larger than the absolute value of a term. For example, if a difference term is 10, then a squared term of 100 is added to MSE, whereas only 10 is added to MAE. Similarly, if a difference term is -20, then a squared term 400 is added to MSE, whereas only 20 (which is the absolute value of -20) is added to MAE.

5.16.2 Nonlinear Least Squares

When predicting housing prices, where the dataset contains a wide range of values, techniques such as linear regression or random forests can cause the model to overfit the samples with the highest values in order to reduce quantities such as mean absolute error.

In this scenario, you probably want an error metric, such as relative error that reduces the importance of fitting the samples with the largest values. This technique is called *nonlinear least squares*, which may use a log-based transformation of labels and predicted values.

The next section contains several code samples, the first of which involves calculating the MSE manually, followed by an example that uses `NumPy` formulas to perform the calculations. Finally, we'll look at a TensorFlow example for calculating the MSE.

5.17 Calculating the MSE Manually

This section contains two line graphs, both of which contain a line that approximates a set of points in a scatter plot.

Figure 5.7 displays a line segment that approximates a scatter plot of points (some of which intersect the line segment). The MSE for the line in Figure 5.7 is computed as follows:

```
MSE = [1*1 + (-1)*(-1) + (-1)*(-1) + 1*1]/7 = 4/7
```

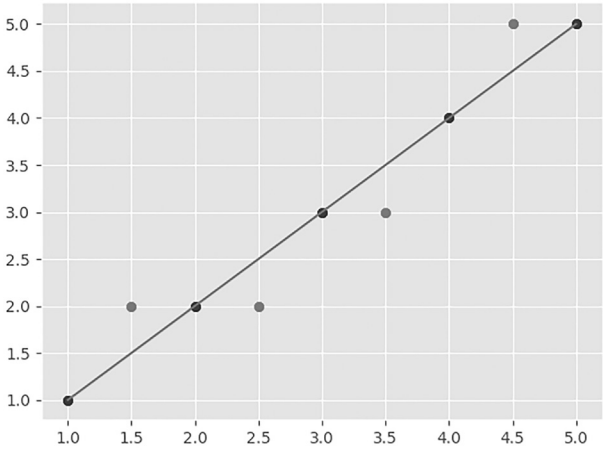

FIGURE 5.7: A line graph that approximates points of a scatter plot.

Figure 5.8 displays a set of points and a line that is a potential candidate for best-fitting line for the data. The MSE for the line in Figure 5.8 is computed as follows:

```
MSE = [(-2)*(-2) + 2*2]/7 = 8/7
```

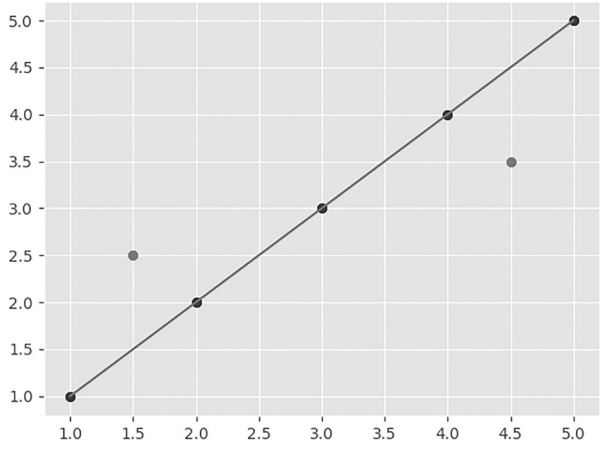

FIGURE 5.8: A line graph that approximates points of a scatter plot.

Thus, the line in Figure 5.7 has a smaller MSE than the line in Figure 5.8, which might have surprised you (or did you guess correctly?)

In these two figures we calculated the MSE easily and quickly, but in general it's significantly more difficult. For instance, if we plot 10 points in the Euclidean plane that do not closely fit a line, with individual terms that involve noninteger values, we would probably need a calculator.

A better solution involves NumPy functions, such as the `np.linspace()` API, as discussed in the next section.

5.18 Approximating Linear Data with `np.linspace()`

Listing 5.4 displays the contents of `np_linspace1.py` that illustrates how to generate some data with the `np.linspace()` API in conjunction with the "perturbation technique."

Listing 5.4: np_linspace1.py

```
import numpy as np

trainX = np.linspace(-1, 1, 6)
trainY = 3*trainX+ np.random.randn(*trainX.shape)*0.5

print("trainX: ", trainX)
print("trainY: ", trainY)
```

The purpose of this code sample is merely to generate and display a set of randomly generated numbers. Later in this chapter we will use this code as a starting point for an actual linear regression task.

Listing 5.4 starts with the definition of the array variable `trainX` that is initialized via the `np.linspace()` API. Next, the array variable `trainY` is defined via the "perturbation technique" that you have seen in previous code samples. The output from Listing 5.4 is here:

```
trainX:  [-1.  -0.6 -0.2  0.2  0.6  1. ]
trainY:  [-2.9008553  -2.26684745 -0.59516253
0.66452207  1.82669051  2.30549295]
trainX:  [-1.  -0.6 -0.2  0.2  0.6  1. ]
trainY:  [-2.9008553  -2.26684745 -0.59516253
0.66452207  1.82669051  2.30549295]
```

Now that we know how to generate `(x,y)` values for a linear equation, let's learn how to calculate the MSE, which is discussed in the next section.

The next example generates a set of data values using the `np.linspace()` method and the `np.random.randn()` method in order to introduces some randomness in the data points.

5.19 Calculating MSE with np.linspace() API

The code sample in this section differs from many of the earlier code samples in this chapter: it uses a hard-coded array of values for `X` and also for `Y` instead of the "perturbation" technique. Hence, you will *not* know the correct value for the slope and y-intercept (and you probably will not be able to guess their correct values). Listing 5.5 displays the contents of `plain_linreg1.py` that illustrates how to compute the MSE with simulated data.

Listing 5.5: plain_linreg1.py

```
import numpy as np
import matplotlib.pyplot as plt

X = [0,0.12,0.25,0.27,0.38,0.42,0.44,0.55,0.92,1.0]
Y = [0,0.15,0.54,0.51, 0.34,0.1,0.19,0.53,1.0,0.58]

costs = []
#Step 1: Parameter initialization
W = 0.45
b = 0.75

for i in range(1, 100):
  #Step 2: Calculate Cost
  Y_pred = np.multiply(W, X) + b
  Loss_error = 0.5 * (Y_pred - Y)**2
  cost = np.sum(Loss_error)/10

  #Step 3: Calculate dW and db
  db = np.sum((Y_pred - Y))
  dw = np.dot((Y_pred - Y), X)
  costs.append(cost)

  #Step 4: Update parameters:
  W = W - 0.01*dw
  b = b - 0.01*db

  if i%10 == 0:
    print("Cost at", i,"iteration = ", cost)
```

```
#Step 5: Repeat via a for loop with 1000 iterations

#Plot cost versus # of iterations
print("W = ", W,"& b = ",  b)
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per tens)')
plt.show()
```

Listing 5.5 initializes the array variables `X` and `Y` with hard-coded values, and then initializes the scalar variables `W` and `b`. The next portion of Listing 5.5 contains a `for` loop that iterates 100 times. After each iteration of the loop, the variables `Y_pred`, `Loss_error`, and `cost` are calculated. Next, the values for `dw` and `db` are calculated, based on the sum of the terms in the array `Y_pred-Y`, and the inner product of `Y_pred-y` and `X`, respectively.

Notice how `W` and b are updated: their values are decremented by the term `0.01*dw` and `0.01*db`, respectively. This calculation ought to look somewhat familiar: the code is programmatically calculating an approximate value of the gradient for `W` and `b`, both of which are multiplied by the learning rate (the hard-coded value 0.01), and the resulting term is decremented from the current values of `W` and `b` in order to produce a new approximation for `W` and b. Although this technique is very simple, it does calculate reasonable values for `W` and `b`.

The final block of code in Listing 5.5 displays the intermediate approximations for `W` and `b`, along with a plot of the cost (vertical axis) versus the number of iterations (horizontal axis). The output from Listing 5.5 is here:

```
Cost at 10 iteration =  0.04114630674619492
Cost at 20 iteration =  0.026706242729839392
Cost at 30 iteration =  0.024738889446900423
Cost at 40 iteration =  0.023850565034634254
Cost at 50 iteration =  0.0231499048706651
Cost at 60 iteration =  0.02255361434242207
Cost at 70 iteration =  0.0220425055291673
Cost at 80 iteration =  0.021604128492245713
Cost at 90 iteration =  0.021228111750568435
W =  0.47256473531193927 & b =  0.19578262688662174
```

Figure 5.9 displays a scatter plot of points generated by the code in Listing 5.5.

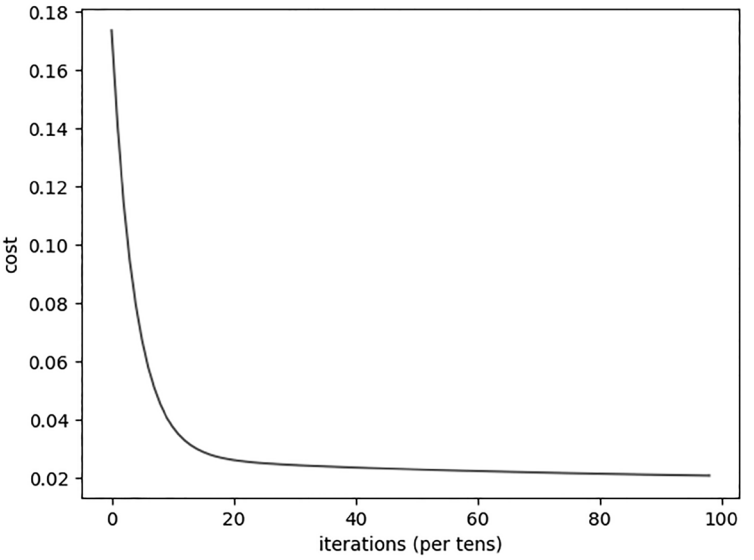

FIGURE 5.9: MSE values with linear regression.

The code sample `plain-linreg2.py` is similar to the code in Listing 5.5: the difference is that instead of a single loop with 100 iterations, there is an outer loop that execute 100 times, and during each iteration of the outer loop, the inner loop also execute 100 times.

5.20 Linear Regression with `Keras`

The code sample in this section contains primarily Keras code in order to perform linear regression. If you have read the previous examples in this chapter, this section will be easier for you to understand because the steps for linear regression are the same.

Listing 5.6 displays the contents of `keras_linear_regression.py` that illustrates how to perform linear regression in `Keras`.

Listing 5.6: keras_linear_regression.py

```
##########################################################
#######
#Keep in mind the following important points:
```

```
#1) Always standardize both input features and target
variable:
#doing so only on input feature produces incorrect
predictions
#2) Data might not be normally distributed: check the
data and
#based on the distribution apply StandardScaler,
MinMaxScaler,
#Normalizer or RobustScaler
#########################################################
#######

import tensorflow as tf
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

df = pd.read_csv('housing.csv')
X  = df.iloc[:,0:13]
y  = df.iloc[:,13].values

mmsc = MinMaxScaler()
X  = mmsc.fit_transform(X)
y  = y.reshape(-1,1)
y  = mmsc.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(X,
y, test_size=0.3)

# this Python method creates a Keras model
def build_keras_model():
  model = tf.keras.models.Sequential()
  model.add(tf.keras.layers.Dense(units=13, input_
dim=13))
  model.add(tf.keras.layers.Dense(units=1))
  model.compile(optimizer='adam',loss='mean_squared_erro
r',metrics=['mae','accuracy'])
  return model

batch_size=32
epochs = 40

# specify the Python method 'build_keras_model' to
create a Keras model
```

```
# using the implementation of the scikit-learn regressor
API for Keras
model = tf.keras.wrappers.scikit_learn.
KerasRegressor(build_fn=build_keras_model, batch_
size=batch_size,epochs=epochs)

# train ('fit') the model and then make predictions:
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
#print("y_test:",y_test)
#print("y_pred:",y_pred)

# scatter plot of test values-vs-predictions
fig, ax = plt.subplots()
ax.scatter(y_test, y_pred)
ax.plot([y_test.min(), y_test.max()], [y_test.min(), y_
test.max()], 'r*--')
ax.set_xlabel('Calculated')
ax.set_ylabel('Predictions')
plt.show()
```

Listing 5.6 starts with multiple `import` statements and then initializes the dataframe `df` with the contents of the `CSV` file `housing.csv` (a portion of which is shown in Listing 5.7). Notice that the training set `X` is initialized with the contents of the first 13 columns of the dataset `housing.csv`, and the variable `y` contains the rightmost column of the dataset `housing.csv`.

The next section in Listing 5.6 uses the `MinMaxScaler` class to calculate the mean and standard deviation, and then invokes the `fit_transform()` method in order to update the `X` values and the `y` values so that they have a mean of 0 and a standard deviation of 1.

Next, the `build_keras_mode()` Python method creates a Keras-based model with two dense layers. Notice that the input layer has size 13, which is the number of columns in the dataframe `X`. The next code snippet compiles the model with an `adam` optimizer, the MSE loss function, and also specifies the MAE and accuracy for the metrics. The compiled model is then returned to the caller.

The next portion of Listing 5.6 initializes the `batch_size` variable to 32 and the epochs variable to 40, and specifies them in the code snippet that creates the model, as shown here:

```
model = tf.keras.wrappers.scikit_learn.
KerasRegressor(build_fn=build_keras_model, batch_
size=batch_size,epochs=epochs)
```

The short comment block that appears in Listing 5.6 explains the purpose of the preceding code snippet, which constructs our `Keras` model.

The next portion of Listing 5.6 invokes the `fit()` method to train the model and then invokes the `predict()` method on the `X_test` data to calculate a set of predictions and initialize the variable `y_pred` with those predictions.

The final portion of Listing 5.6 displays a scatter plot in which the horizontal axis is the values in `y_test` (the actual values from the CSV file `housing.csv`) and the vertical axis is the set of predicted values.

Figure 5.10 displays a scatter plot of points based on the test values and the predictions for those test values.

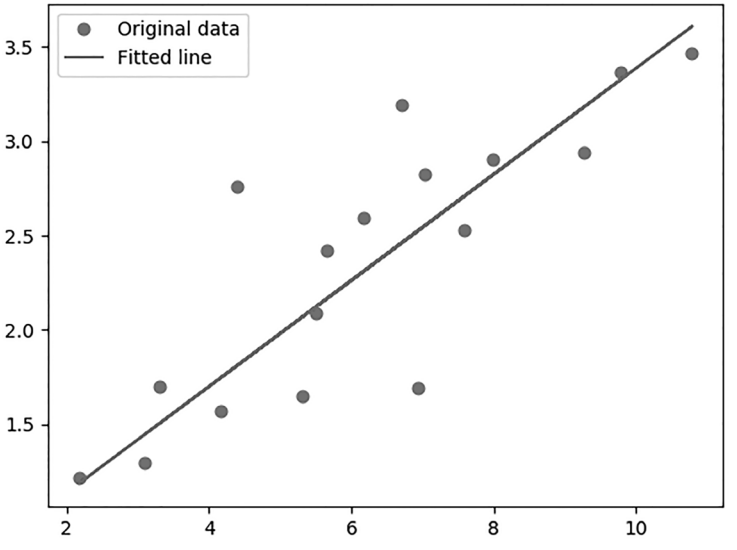

FIGURE 5.10: A scatter plot and a best-fitting line.

Listing 5.7 displays the first four rows of the CSV file `housing.csv` that is used in the Python code in Listing 5.6.

Listing 5.7: housing.csv

```
0.00632,18,2.31,0,0.538,6.575,65.2,4.09,1,296,15.3,396.9
,4.98,24
```

```
0.02731,0,7.07,0,0.469,6.421,78.9,4.9671,2,242,17.8,396.
9,9.14,21.6
0.02729,0,7.07,0,0.469,7.185,61.1,4.9671,2,242,17.8,392.
83,4.03,34.7
0.03237,0,2.18,0,0.458,6.998,45.8,6.0622,3,222,18.7,394.
63,2.94,33.4
```

5.21 Summary

This chapter introduced you to machine learning and concepts such as feature selection, feature engineering, data cleaning, training sets, and test sets. Next you learned about supervised, unsupervised, and semisupervised learning. Then you learned regression tasks, classification tasks, and clustering, as well as the steps that are typically required in order to prepare a dataset. These steps include *feature selection* or *feature extraction* that can be performed using various algorithms. Then you learned about issue that can arise with the data in datasets, and how to rectify them.

In addition, you also learned about linear regression, along with a brief description of how to calculate a best-fitting line for a dataset of values in the Euclidean plane. You saw how to perform linear regression using `NumPy` in order to initialize arrays with data values, along with a "perturbation" technique that introduces some randomness for the y values. This technique is useful because you will know the correct values for the slope and y-intercept of the best-fitting line, which you can then compare with the trained values.

You then learned how to perform linear regression in code samples that involve `Keras`. In addition, you saw how to use Matplotlib in order to display line graphs for best-fitting lines and graphs that display the cost versus the number of iterations during the training-related code blocks.

CHAPTER 6

CLASSIFIERS IN MACHINE LEARNING

- What is Classification?
- What Are Linear Classifiers?
- What is kNN?
- What Are Decision Trees?
- What Are Random Forests?
- What Are SVMs?
- What is Bayesian Inference?
- What is a Bayesian Classifier?
- Training Classifiers
- Evaluating Classifiers
- What Are Activation Functions?
- Common Activation Functions
- The ReLU and ELU Activation Functions
- Sigmoid, Softmax, and Hardmax Similarities
- Sigmoid, Softmax, and HardMax Differences
- What is Logistic Regression?
- Keras and Logistic Regression
- Keras, Logistic Regression, and Iris Dataset
- Summary

This chapter presents numerous classification algorithms in machine learning. This includes algorithms such as the k nearest neighbor (kNN) algorithm, logistic regression (despite its name it *is* a classifier), decision trees, random forests, SVMs, and Bayesian classifiers. The emphasis on algorithms is intended to introduce you to machine learning, which includes a tree-based code sample that relies on `scikit-learn`. The latter portion of this chapter contains `Keras`-based code samples for standard datasets.

Due to space constraints, this chapter does not cover other well-known algorithms, such as linear discriminant analysis and the kMeans algorithm (which is for unsupervised learning and clustering). However, there are many online tutorials available that discuss these and other algorithms in machine learning.

With the preceding points in mind, the first section of this chapter briefly discusses the classifiers that are mentioned in the introductory paragraph. The second section of this chapter provides an overview of activation functions, which will be very useful if you decide to learn about deep neural networks. In this section you will learn how and why they are used in neural networks. This section also contains a list of the TensorFlow APIs for activation functions, followed by a description of some of their merits.

The third section introduces logistic regression, which relies on the sigmoid function, which is also used in recurrent neural networks (RNNs) and long short term memory (LSTMs). The fourth part of this chapter contains a code sample involving logistic regression and the `MNIST` dataset.

In order to give you some context, classifiers are one of three major types of algorithms: regression algorithms (such as linear regression in Chapter 4), classification algorithms (discussed in this chapter), and clustering algorithms (such as kMeans, which is not discussed in this book).

Another point: the section pertaining to activation functions does involve a basic understanding of hidden layers in a neural network. Depending on your comfort level, you might benefit from reading some preparatory material before diving into this section (there are many articles available online).

6.1 What is Classification?

Given a dataset that contains observations whose class membership is known, classification is the task of determining the class to which a new