(while we may actually use a different optimizer, we will refer to updates as gradient descent for convenience). How can we iteratively perform gradient descent to learn on this dataset?

The simple answer is that we use a Python `for`-loop. In each iteration, we use `sess.run()` to fetch the `train_op` along with the merged summary op `merged` and the loss `l` from the graph. We feed all datapoints and labels into `sess.run()` using a feed dictionary.

The code snippet in Example 3-10 demonstrates this simple learning method. Note that we don't make use of minibatches for pedagogical simplicity. Code in following chapters will use minibatches when training on larger datasets.

Example 3-10. A simple example of training a model

```
n_steps = 1000
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  # Train model
  for i in range(n_steps):
    feed_dict = {x: x_np, y: y_np}
    _, summary, loss = sess.run([train_op, merged, l], feed_dict=feed_dict)
    print("step %d, loss: %f" % (i, loss))
    train_writer.add_summary(summary, i)
```

Training Linear and Logistic Models in TensorFlow

This section ties together all the TensorFlow concepts introduced in the previous section to train linear and logistic regression models upon the toy datasets we introduced previously in the chapter.

Linear Regression in TensorFlow

In this section, we will provide code to define a linear regression model in TensorFlow and learn its weights. This task is straightforward and you can do it without TensorFlow easily. Nevertheless, it's a good exercise to do in TensorFlow since it will bring together the new concepts that we have introduced throughout the chapter.

Defining and training linear regression in TensorFlow

The model for a linear regression is straightforward:

$$y = wx + b$$

Here w and b are the weights we wish to learn. We transform these weights into `tf.Variable` objects. We then use tensorial operations to construct the L^2 loss:

$$\mathcal{L}(x, y) = (y - wx - b)^2$$

The code in Example 3-11 implements these mathematical operations in TensorFlow. It also uses `tf.name_scope` to group various operations, and adds a `tf.train.AdamOptimizer` for learning and `tf.summary` operations for TensorBoard usage.

Example 3-11. Defining a linear regression model

```
# Generate tensorflow graph
with tf.name_scope("placeholders"):
  x = tf.placeholder(tf.float32, (N, 1))
  y = tf.placeholder(tf.float32, (N,))
with tf.name_scope("weights"):
  # Note that x is a scalar, so W is a single learnable weight.
  W = tf.Variable(tf.random_normal((1, 1)))
  b = tf.Variable(tf.random_normal((1,)))
with tf.name_scope("prediction"):
  y_pred = tf.matmul(x, W) + b
with tf.name_scope("loss"):
  l = tf.reduce_sum((y - y_pred)**2)
# Add training op
with tf.name_scope("optim"):
  # Set learning rate to .001 as recommended above.
  train_op = tf.train.AdamOptimizer(.001).minimize(l)
with tf.name_scope("summaries"):
  tf.summary.scalar("loss", l)
  merged = tf.summary.merge_all()

train_writer = tf.summary.FileWriter('/tmp/lr-train', tf.get_default_graph())
```

Example 3-12 then trains this model as discussed previously (without using mini-batches).

Example 3-12. Training the linear regression model

```
n_steps = 1000
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  # Train model
  for i in range(n_steps):
    feed_dict = {x: x_np, y: y_np}
    _, summary, loss = sess.run([train_op, merged, l], feed_dict=feed_dict)
    print("step %d, loss: %f" % (i, loss))
    train_writer.add_summary(summary, i)
```

All code for this example is provided in the GitHub repository associated with this book. We encourage all readers to run the full script for the linear regression example to gain a firsthand sense for how the learning algorithm functions. The example is

small enough that readers will not need access to any special-purpose computing hardware to run.

Taking Gradients for Linear Regression

The equation for the linear system we're modeling is $y = wx + b$ where w, b are the learnable weights. As we mentioned previously, the loss for this system is $\mathscr{L} = (y - wx - b)^2$. Some matrix calculus can be used to compute the gradients of the learnable parameters directly for w:

$$\nabla w = \frac{\partial \mathscr{L}}{\partial w} = -2(y - wx - b)x^T$$

and for b

$$\nabla b = \frac{\partial \mathscr{L}}{\partial b} = -2(y - wx - b)$$

We place these equations here only for reference for curious readers. We will not attempt to systematically teach how to take the derivatives of the loss functions we encounter in this book. However, we will note that for complicated systems, taking the derivative of the loss function by hand helps build up an intuition for how the deep network learns. This intuition can serve as a powerful guide for the designer, so we encourage advanced readers to pursue this topic on their own.

Visualizing linear regression models with TensorBoard

The model defined in the previous section uses `tf.summary.FileWriter` to write logs to a logging directory */tmp/lr-train*. We can invoke TensorBoard on this logging directory with the command in Example 3-13 (TensorBoard is installed by default with TensorFlow).

Example 3-13. Invoking TensorBoard

```
tensorboard --logdir=/tmp/lr-train
```

This command will start TensorBoard on a port attached to localhost. Use your browser to open this port. The TensorBoard screen will look something like Figure 3-8. (The precise appearance may vary depending on your version of TensorBoard.)

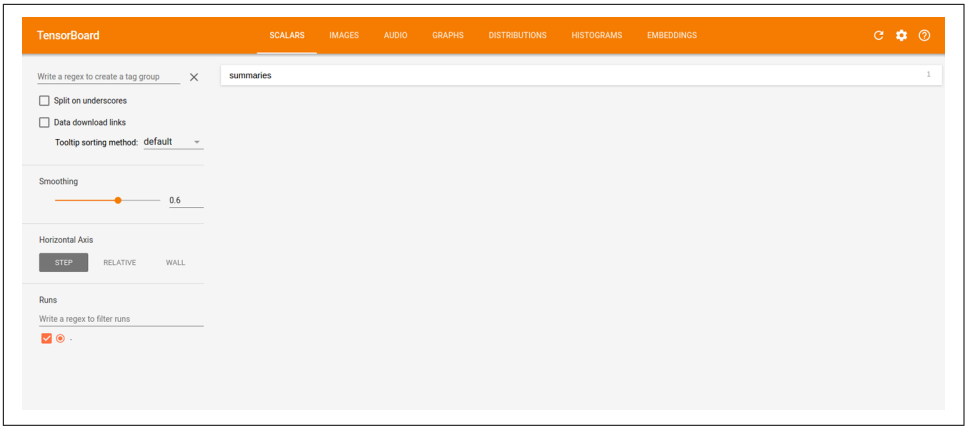

Figure 3-8. Screenshot of TensorBoard panel.

Navigate to the Graphs tab, and you will see a visualization of the TensorFlow architecture we have defined as illustrated in Figure 3-9.

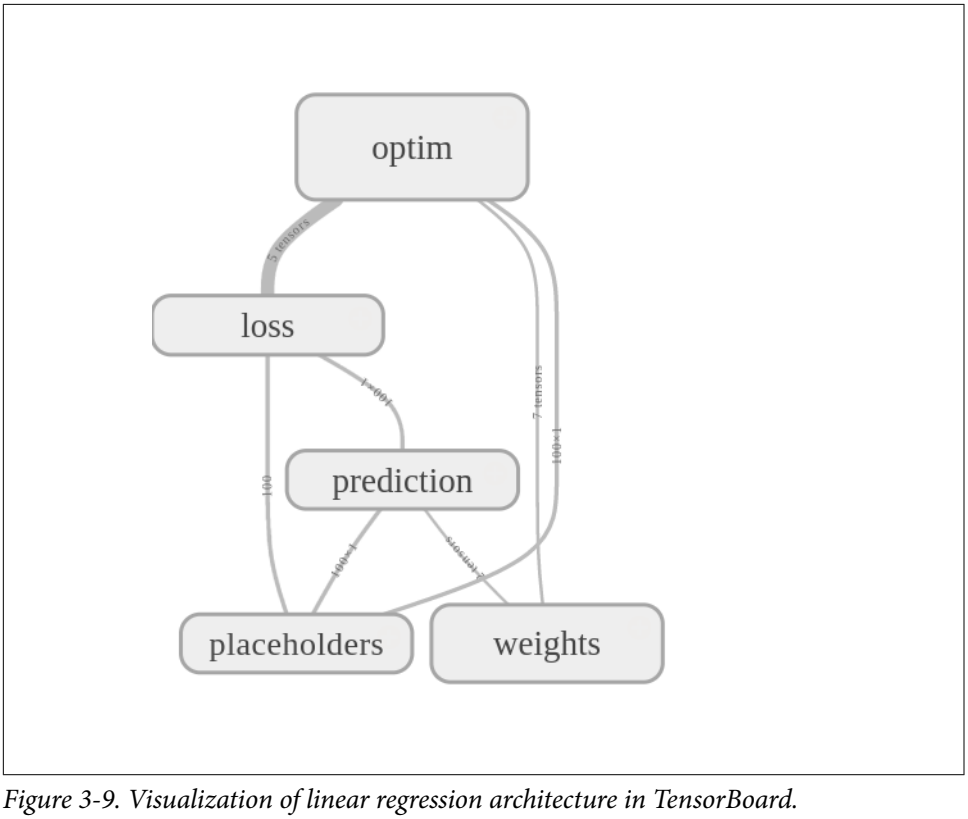

Figure 3-9. Visualization of linear regression architecture in TensorBoard.

Note that this visualization has grouped all computational graph elements belonging to various `tf.name_scopes`. Different groups are connected according to their dependencies in the computational graph. You can expand all of the grouped elements to view their contents. Figure 3-10 illustrates the expanded architecture.

As you can see, there are many hidden nodes that suddenly become visible! TensorFlow functions like `tf.train.AdamOptimizer` often hide many internal variables under a `tf.name_scope` of their own. Expanding in TensorBoard provides an easy way to peer underneath the hood to see what the system is actually creating. Although the visualization looks quite complex, most of these details are under the hood and not anything you need to worry about just yet.

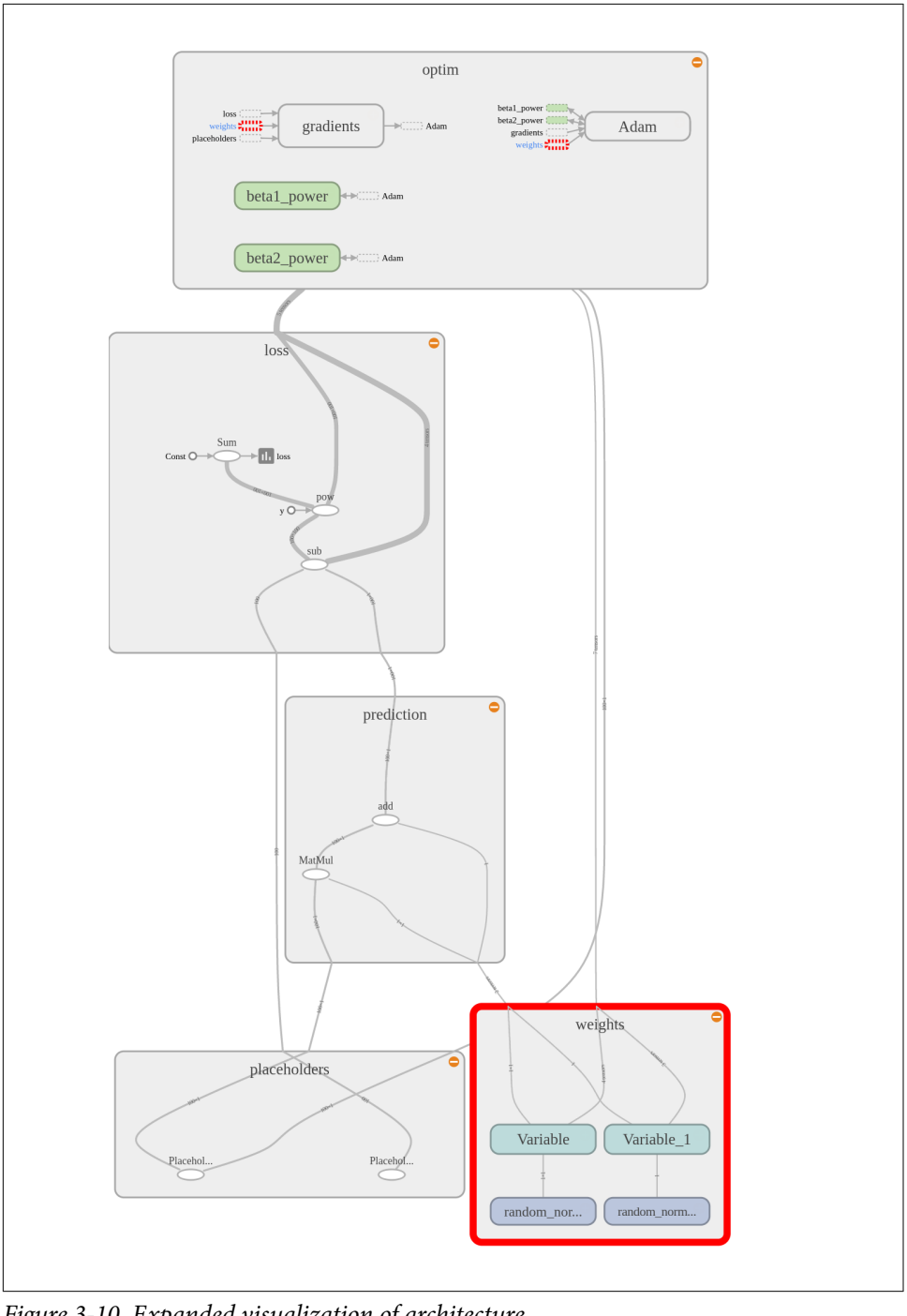

Figure 3-10. Expanded visualization of architecture.

Navigate back to the Home tab and open the Summaries section. You should now see a loss curve that looks something like Figure 3-11. Note the smooth falling shape. The loss falls rapidly at the beginning as the prior is learned, then tapers off and settles.

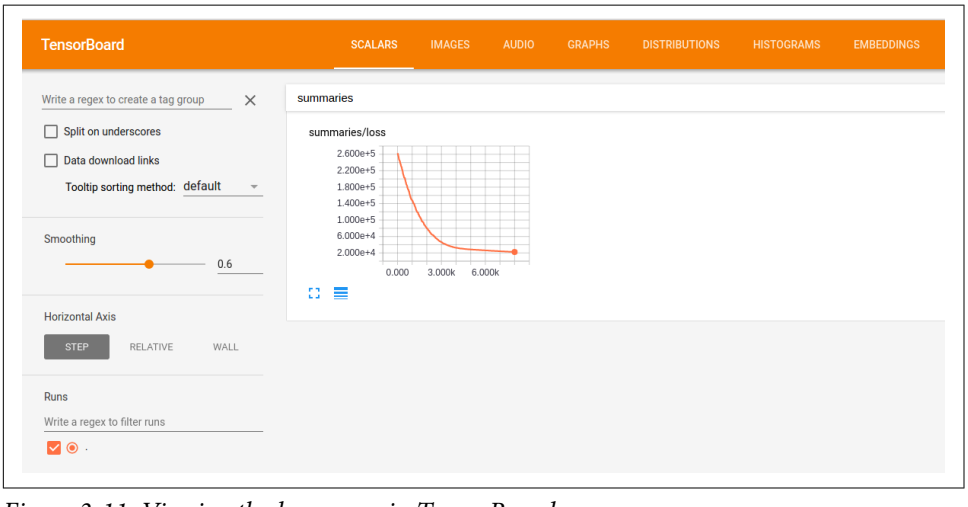

Figure 3-11. Viewing the loss curve in TensorBoard.

Visual and Nonvisual Debugging Styles

Is using a tool like TensorBoard necessary to get good use out of a system like TensorFlow? It depends. Is using a GUI or an interactive debugger necessary to be a professional programmer?

Different programmers have different styles. Some will find that the visualization capabilities of TensorBoard come to form a critical part of their tensorial programming workflows. Others will find that TensorBoard isn't terribly useful and will make greater use of print-statement debugging. Both styles of tensorial programming and debugging are valid, just as there are great programmers who swear by debuggers and others who loathe them.

In general, TensorBoard is quite useful for debugging and for building basic intuition about the dataset at hand. We recommend that you follow the style that works best for you.

Metrics for evaluating regression models

So far, we haven't discussed how to evaluate whether a trained model has actually learned anything. The first tool for evaluating whether a model has trained is by looking at the loss curve to ensure it has a reasonable shape. You learned how to do this in the previous section. What's the next thing to try?

We now want you to look at *metrics* associated with the model. A metric is a tool for comparing predicted labels to true labels. For regression problems, there are two common metrics: R^2 and RMSE (root-mean-squared error). The R^2 is a measure of the correlation between two variables that takes values between +1 and 0. +1 indicates perfect correlation, while 0 indicates no correlation. Mathematically, the R^2 for two datasets X and Y is defined as

$$R^2 = \frac{\text{cov}(X, Y)^2}{\sigma_X^2 \sigma_Y^2}$$

Where cov(X, Y) is the covariance of X and Y, a measure of how the two datasets jointly vary, while σ_X and σ_Y are standard deviations, measures of how much each set individually varies. Intuitively, the R^2 measures how much of the independent variation in each set can be explained by their joint variation.

Multiple Types of R^2!

Note that there are two common definitions of R^2 used in practice. A common beginner (and expert) mistake is to confuse the two definitions. In this book, we will always use the squared Pearson correlation coefficient (Figure 3-12). The other definition is called the coefficient of determination. This other R^2 is often much more confusing to deal with since it doesn't have a lower bound of 0 like the squared Pearson correlation does.

In Figure 3-12, predicted and true values are highly correlated with an R^2 of nearly 1. It looks like learning has done a wonderful job on this system and succeeded in learning the true rule. *Not so fast*. You will note that the scale on the two axes in the figure isn't the same! It turns out that R^2 doesn't penalize for differences in scale. In order to understand what's happened on this system, we need to consider an alternate metric in Figure 3-13.

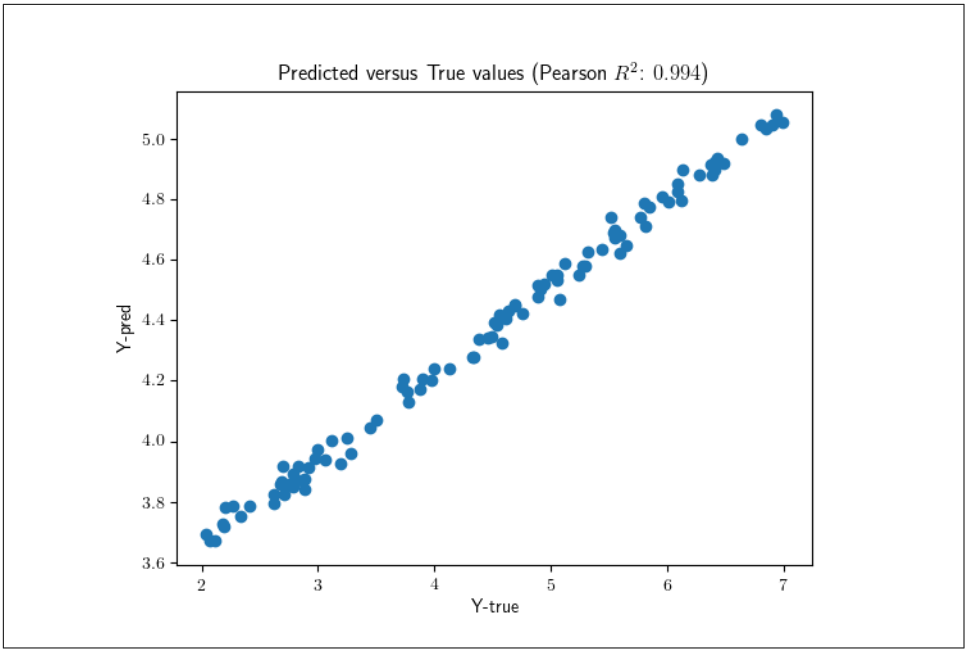

Figure 3-12. Plotting the Pearson correlation coefficient.

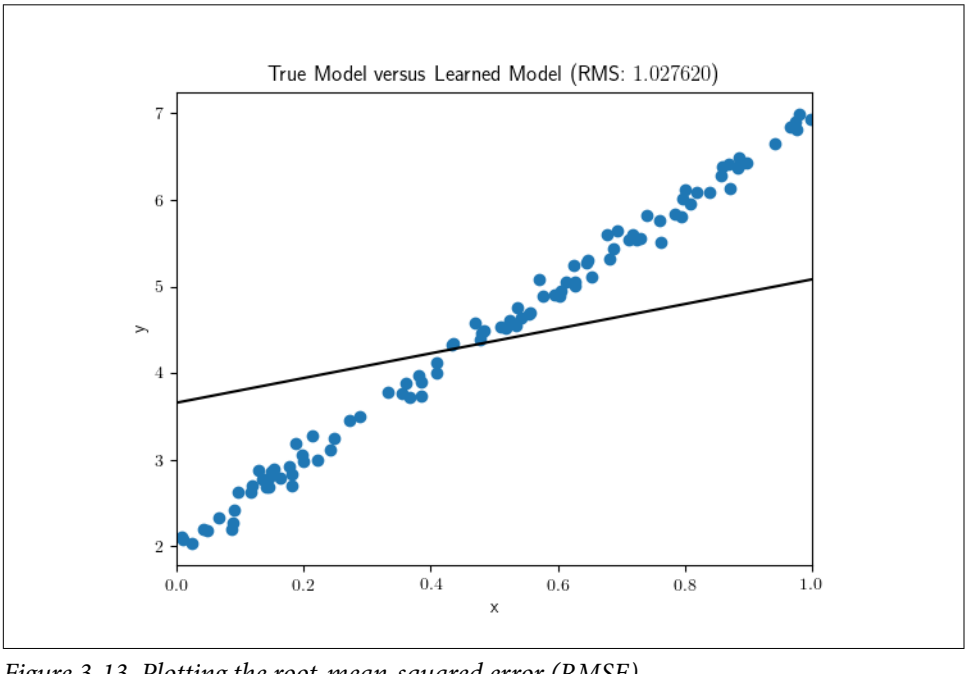

Figure 3-13. Plotting the root-mean-squared error (RMSE).

The RMSE is a measure of the average difference between predicted values and true values. In Figure 3-13 we plot predicted values and true labels as two separate functions using datapoints x as our x-axis. Note that the line learned isn't the true function! The RMSE is relatively high and diagnoses the error, unlike the R^2, which didn't pick up on this error.

What happened on this system? Why didn't TensorFlow learn the correct function despite being trained to convergence? This example provides a good illustration of one of the weaknesses of gradient descent algorithms. There is no guarantee of finding the true solution! The gradient descent algorithm can get trapped in *local minima*. That is, it can find solutions that look good, but are not in fact the lowest minima of the loss function $\mathscr{L}$.

Why use gradient descent at all then? For simple systems, it is indeed often better to avoid gradient descent and use other algorithms that have stronger performance guarantees. However, on complicated systems, such as those we will show you in later chapters, there do not yet exist alternative algorithms that perform better than gradient descent. We encourage you to remember this fact as we proceed further into deep learning.

Logistic Regression in TensorFlow

In this section, we will define a simple classifier using TensorFlow. It's worth first considering what the equation is for a classifier. The mathematical trick that is commonly used is exploiting the sigmoid function. The sigmoid, plotted in Figure 3-14, commonly denoted by σ, is a function from the real numbers $\mathbb{R}$ to (0, 1). This property is convenient since we can interpret the output of a sigmoid as probability of an event happening. (The trick of converting discrete events into continuous values is a recurring theme in machine learning.)

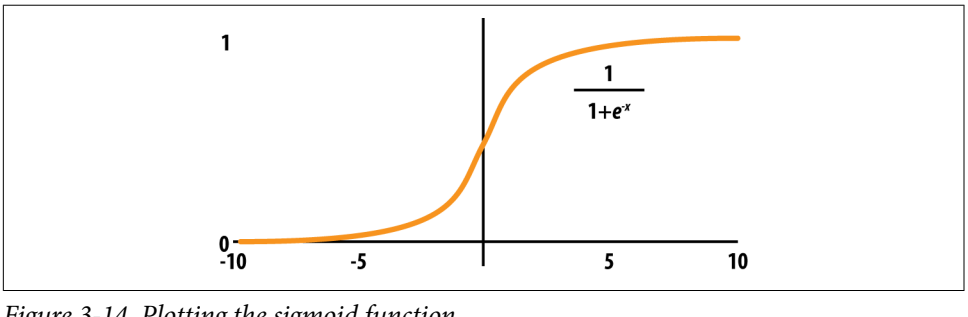

Figure 3-14. Plotting the sigmoid function.

The equations for predicting the probabilities of a discrete 0/1 variable follow. These equations define a simple logistic regression model: