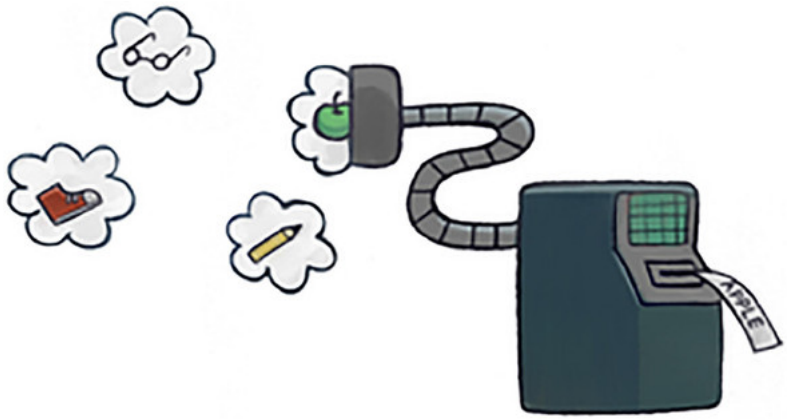
Chapter 4. A gentle introduction to classification



This chapter covers

- Writing formal notation
- Using logistic regression
- Working with a confusion matrix
- Understanding multiclass classification

Imagine an advertisement agency collecting information about user interactions to decide what type of ad to show. That's not uncommon. Google, Twitter, Facebook, and other big tech giants that rely on ads have creepy-good personal profiles of their users to help deliver personalized ads. A user who's recently searched for gaming keyboards or graphics cards is probably more likely to click ads about the latest and greatest video games.

Delivering a specially crafted advertisement to each individual may be difficult, so grouping users into categories is a common technique. For example, a user may be categorized as a "gamer" to receive relevant video game–related ads.

Machine learning is the go-to tool to accomplish such a task. At the most fundamental level, machine-learning practitioners want to build a tool to help them understand data. Labeling data items as belonging in separate categories is an excellent way to characterize data for specific needs.

The previous chapter dealt with regression, which was about fitting a curve to data. As you recall, the best-fit curve is a function that takes as input a data item and assigns it a number. Creating a machine-learning model that instead assigns discrete labels to its inputs is called *classification*. It's a supervised-learning algorithm for dealing with discrete output. (Each discrete value is called a *class*.) The input is typically a feature vector, and the output is a class. If there are only two class labels (for example, True/False, On/Off, Yes/No), we call this learning algorithm a *binary classifier*. Otherwise, it's called a *multiclass classifier*.

There are many types of classifiers, but this chapter focuses on the ones outlined in table 4.1. Each has its advantages and disadvantages, which we'll delve into deeper after we start implementing each one in TensorFlow.

Linear regression is the easiest to implement because we already did most of the hard work in chapter 3, but as you'll see, it's a terrible classifier. A much better classifier is the logistic regression algorithm. As the name suggests, it uses logarithmic properties to define a better cost function. And lastly, softmax regression is a direct approach to solving multiclass classification. It's a natural generalization of logistic regression. It's called softmax regression because a function called `softmax` is applied as the last step.

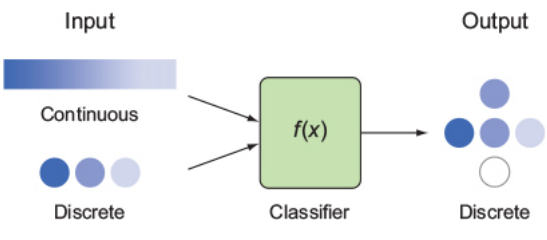
Table 4.1. Classifiers

Type	Pros	Cons
Linear regression	Simple to implement	Not guaranteed to work Supports only binary labels
Logistic regression	Highly accurate Flexible ways to regularize model for custom adjustment Model responses are measures of probability Easy-to-update model with new data	Supports only binary labels
Softmax regression	Supports multiclass classification Model responses are measures of probability	More complicated to implement

4.1. FORMAL NOTATION

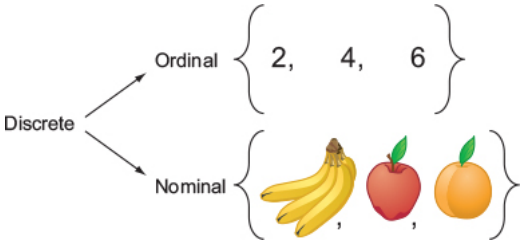
In mathematical notation, a classifier is a function $y = f(x)$, where x is the input data item and y is the output category (figure 4.1). Adopting from traditional scientific literature, we often refer to the input vector x as the *independent variable*, and the output y as the *dependent variable*.

Figure 4.1. A classifier produces discrete outputs but may take either continuous or discrete inputs.



Formally, a category label is restricted to a range of possible values. You can think of two-valued labels as being like Boolean variables in Python. When the input features have only a fixed set of possible values, you need to ensure that your model can understand how to handle them. Because the set of functions in a model typically deal with continuous real numbers, you need to preprocess the dataset to account for discrete variables, which fall into one of two types: ordinal or nominal (figure 4.2).

Figure 4.2. There are two types of discrete sets: those with values that can be ordered (ordinal) and those with values that can't (nominal).

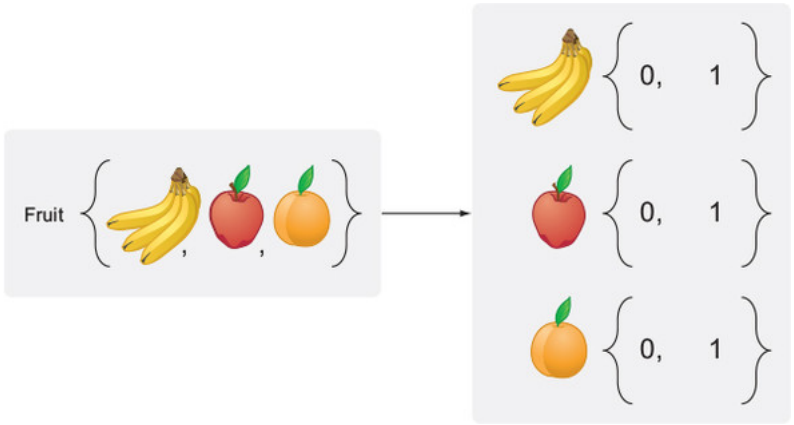


Values of an ordinal type, as the name suggests, can be ordered. For example, the values in a set of even numbers from 1 to 10 are ordinal because integers can be compared with each other. On the other hand, an element from a set of fruits `{banana, apple, orange}` might not come with a natural ordering. We call values from such a set nominal, because they can be described by only their names.

A simple approach to representing nominal variables in a dataset is to assign a number to each label. Our set `{banana, apple, orange}` could instead be processed as `{0, 1, 2}`. But some classification models may have a strong bias about how the data behaves. For example, linear regression would interpret our apple as midway between a banana and an orange, which makes no natural sense.

A simple workaround to represent nominal categories of a dependent variable is by adding *dummy variables* for each value of the nominal variable. In this example, the `fruit` variable would be removed, and replaced by three separate variables: `banana`, `apple`, and `orange`. Each variable holds a value of `0` or `1` (figure 4.3), depending on whether the category for that fruit holds true. This process is often referred to as *one-hot encoding*.

Figure 4.3. If the values of a variable are nominal, they might need to be preprocessed. One solution is to treat each nominal value as a Boolean variable, as shown on the right: `banana`, `apple`, and `orange` are three newly added variables, each having a value of `0` or `1`. The original `fruit` variable is removed.



Just as in linear regression from chapter 3, the learning algorithm must traverse the possible functions supported by the underlying model, called M. In linear regression, the model was parameterized by w. The function $y = M(w)$ can then be tried out to measure its cost. In the end, we choose a value of w with the least cost. The only difference between regression and classification is that the output is no longer a continuous spectrum, but instead a discrete set of class labels.

Exercise 4.1

Is it a better idea to treat each of the following as a regression or classification task? (a) Predicting stock prices; (b) Deciding which stocks you should buy, sell, or hold; (c) Rating the quality of a computer on a 1–10 scale

ANSWER

(a) Regression, (b) Classification, (c) Either

Because the input/output types for regression are even more general than those of classification, nothing prevents you from running a linear regression algorithm on a classification task. In fact, that's exactly what you'll do in section 4.3. Before you begin implementing TensorFlow code, it's important to gauge the strength of a classifier. The next section covers state-of-the-art approaches to measuring a classifier's success.

4.2. MEASURING PERFORMANCE

Before you begin writing classification algorithms, you should be able to check the success of your results. This section covers essential techniques to measure performance in classification problems.

4.2.1. Accuracy

Do you remember those multiple-choice exams in high school or college? Classification problems in machine learning are similar. Given a statement, your job is to classify it as one of the given multiple-choice "answers." If you have only two choices, as in a true-or-false exam, we call it a *binary classifier*. If this were a graded exam in school, the typical way to measure your score would be to count the number of correct answers and divide that by the total number of questions.

Machine learning adopts this same scoring strategy and calls it *accuracy*. Accuracy is measured by the following formula:

$$accuracy = \frac{\#correct}{\#total}$$

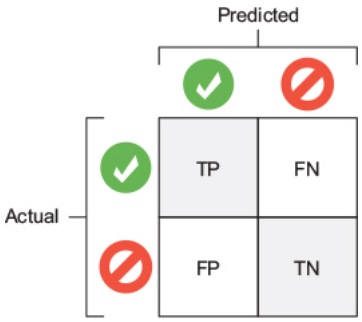
This formula gives a crude summary of the performance, which may be sufficient if you're worried only about the overall correctness of the algorithm. But the accuracy measure doesn't reveal a breakdown of correct and incorrect results for each label.

To account for this limitation, a *confusion matrix* is a more detailed report of a classifier's success. A useful way to describe how well a classifier performs is by

inspecting the way it performs on each of the classes.

For instance, consider a binary classifier with "positive" and "negative" labels. As shown in figure 4.4, a confusion matrix is a table that compares how the predicted responses compare with actual ones. Data items that are correctly predicted as positive are called *true positives* (TP). Those that are incorrectly predicted as positive are called *false positives* (FP). If the algorithm accidentally predicts an element to be negative when in reality it is positive, we call this situation a *false negative* (FN). Lastly, when the prediction and reality both agree that a data item is a negative label, it's called a *true negative* (TN). As you can see, it's called a *confusion matrix* because it enables you to easily see how often a model confuses two classes that it's trying to differentiate.

Figure 4.4. You can compare predicted results to actual results by using a matrix of positive (green check mark) and negative (red forbidden) labels.



Note to Print Book Readers

Many graphics in this book include color, which can be viewed in the eBook versions. To get your free eBook in PDF, ePub, or Kindle format, go to www.manning.com/books/machine-learning-with-tensorflow (http://www.manning.com/books/machine-learning-with-tensorflow) to register your print book.

4.2.2. Precision and recall

Although the definitions of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) are all useful individually, the true power comes in the interplay between them.

The ratio of true positives to total positive examples is called *precision*. It's a score of how likely a positive prediction is to be correct. The left column in figure 4.4 is the total number of positive predictions (TP + FP), so the equation for precision is the following:

$$precision = \frac{TP}{TP + FP}$$

The ratio of true positives to all possible positives is called *recall*. It measures the ratio of true positives found. It's is a score of how many true positives were successfully predicted (that is, recalled). The top row in figure 4.4 is the total number of all positives (TP + FN), so the equation for recall is the following:

$$recall = \frac{TP}{TP + FN}$$

Simply put, precision is a measure of the predictions the algorithm got right, and recall is a measure of the right things the algorithm identified in the final set. If the precision is higher than the recall, the model is better at successfully identifying correct items than not identifying some wrong items, and vice versa.

Let's do a quick example. Let's say you're trying to identify cats in a set of 100 pictures; 40 of the pictures are cats, and 60 are dogs. When you run your classifier, 10 of the cats are identified as dogs, and 20 of the dogs are identified as cats. Your confusion matrix looks like figure 4.5.

Figure 4.5. An example of a confusion matrix for evaluating the performance of a classification algorithm

Confusion matrix		Predicted	
		Cat	Dog
Actual	Cat	30 True positives	20 False positives
	Dog	10 False negatives	40 True negatives

You can see the total number of cats on the left side of the prediction column: 30 identified correctly, and 10 not, totaling 40.

Exercise 4.2

What are the precision and recall for cats? What's the accuracy of the system?

ANSWER

For cats, the precision is 30 / (30 + 20) or 3/5. The recall is 30 / (30 + 10), or 3/4. The accuracy is (30 + 40) / 100, or 70%.

4.2.3. Receiver operating characteristic curve

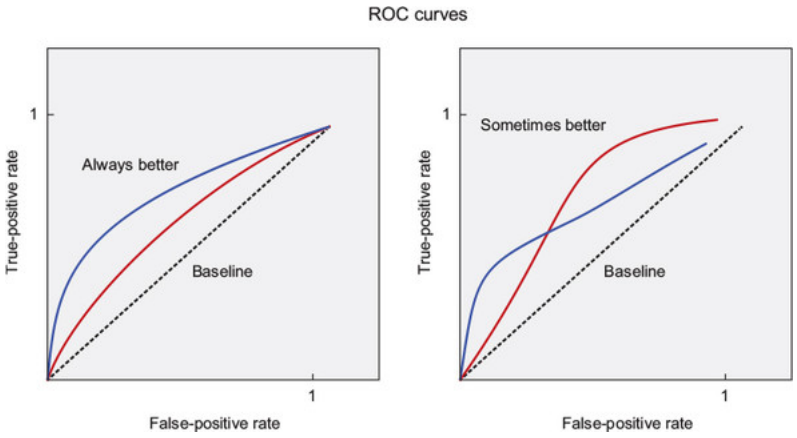
Because binary classifiers are among the most popular tools, many mature techniques exist for measuring their performance, such as the receiver operating characteristic (ROC) curve. The ROC curve is a plot that lets you compare the trade-offs between false positives and true positives. The x-axis is the measure of false-positive values, and the y-axis is the measure of true-positive values.

A binary classifier reduces its input feature vector into a number and then decides the class based on whether the number is greater than or less than a specified threshold. As you adjust a threshold of the machine-learning classifier, you plot the various values of false-positive and true-positive rates.

A robust way to compare various classifiers is by comparing their ROC curves. When two curves don't intersect, one method is certainly better than the other. Good

algorithms are far above the baseline. A quantitative way to compare classifiers is by measuring the area under the ROC curve. If a model has an area-under-curve (AUC) value higher than 0.9, it's an excellent classifier. A model that randomly guesses the output will have an AUC value of about 0.5. See figure 4.6 for an example.

Figure 4.6. The principled way to compare algorithms is by examining their ROC curves. When the true-positive rate is greater than the false-positive rate in every situation, it's straightforward to declare that one algorithm is dominant in terms of its performance. If the true-positive rate is less than the false-positive rate, the plot dips below the baseline shown by the dotted line.



Exercise 4.3

How would a 100% correct rate (all true positives, no false positives) look as a point on an ROC curve?

Answer

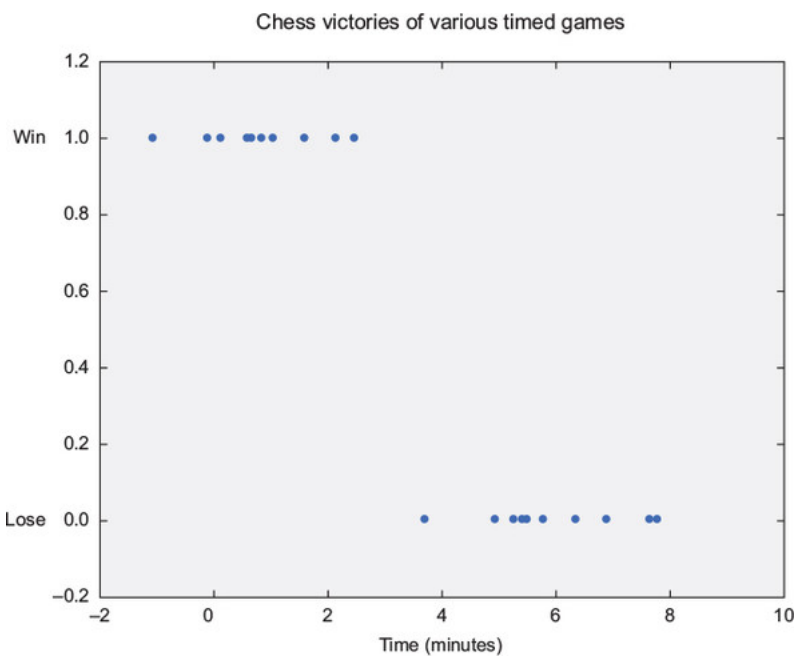
The point for a 100% correct rate would be located on the positive y-axis of the ROC curve.

4.3. USING LINEAR REGRESSION FOR CLASSIFICATION

One of the simplest ways to implement a classifier is to tweak a linear regression algorithm, like the ones in chapter 3. As a reminder, the linear regression model is a set of functions that look linear, $f(x) = wx$. The function $f(x)$ takes continuous real numbers as input and produces continuous real numbers as output. Remember, classification is all about discrete outputs. So, one way to force the regression model to produce a two-valued (binary) output is by setting values above a certain threshold to a number (such as 1) and values below that threshold to a different number (such as 0).

We'll proceed with the following motivating example. Imagine that Alice is an avid chess player, and you have records of her win/loss history. Moreover, each game has a time limit ranging from 1 to 10 minutes. You can plot the outcome of each game as shown in figure 4.7. The x-axis represents the time limit of the game, and the y-axis signifies whether she won ($y = 1$) or lost ($y = 0$).

Figure 4.7. A visualization of a binary classification training dataset. The values are divided into two classes: all points where $y = 1$, and all points where $y = 0$.

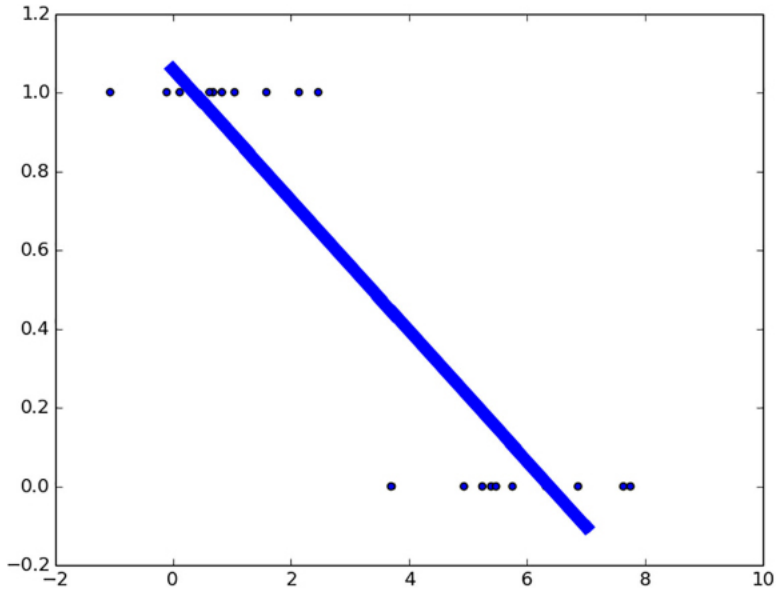


As you see from the data, Alice is a quick thinker: she always wins short games. But she usually loses games that have longer time limits. From the plot, you'd like to predict the critical game time-limit that decides whether she'll win.

You want to challenge her to a game that you're sure of winning. If you choose an obviously long game, such as one that takes 10 minutes, she'll refuse to play. So, let's set up the game time to be as short as possible so she'll be willing to play against you, while tilting the balance to your advantage.

A linear fit on the data gives you something to work with. Figure 4.8 shows the best-fit line computed using linear regression from listing 4.1 (appearing shortly). The value of the line is closer to 1 than it is to 0 for games that Alice will likely win. It appears that if you pick a time corresponding to when the value of the line is less than 0.5 (that is, when Alice is more likely to lose than to win), then you have a good chance of winning.

Figure 4.8. The diagonal line is the best-fit line on a classification dataset. Clearly, the line doesn't fit the data well, but it provides an imprecise approach for classifying new data.



The line is trying to fit the data as best possible. Due to the nature of the training data, the model will respond with values near 1 for positive examples and values near 0 for negative examples. Because you're modeling this data with a line, some input may produce values between 0 and 1. As you may imagine, values too far into one category will result in values greater than 1 or less than 0. You need a way to decide when an item belongs to one category more than another. Typically, you choose the midpoint, 0.5, as a deciding boundary (also called the *threshold*). Are you've seen, this procedure uses linear regression to perform classification.

Exercise 4.4

What are the disadvantages of using linear regression as a tool for classification? (See listing 4.4 for a hint.)

ANSWER

Linear regression is sensitive to outliers in your data, so it isn't an accurate classifier.

Let's write your first classifier! Open a new Python source file, and call it linear.py. Use the following listing to write the code. In the TensorFlow code, you'll need to first define placeholder nodes and then inject values into them from the `session.run()` statement.

Listing 4.1. Using linear regression for classification

```
import tensorflow as tf                                                    1
import numpy as np                                                         1
import matplotlib.pyplot as plt                                            1

x_label0 = np.random.normal(5, 1, 10)                                      2
x_label1 = np.random.normal(2, 1, 10)                                      2
xs = np.append(x_label0, x_label1)                                         2
labels = [0.] * len(x_label0) + [1.] * len(x_label1)                       3

plt.scatter(xs, labels)                                                    4

learning_rate = 0.001                                                      5
training_epochs = 1000                                                     5

X = tf.placeholder("float")                                                6
Y = tf.placeholder("float")                                                6

def model(X, w):                                                           7
    return tf.add(tf.multiply(w[1], tf.pow(X, 1)),                         7
                  tf.multiply(w[0], tf.pow(X, 0)))                         7

w = tf.Variable([0., 0.], name="parameters")                               8
y_model = model(X, w)                                                      9
cost = tf.reduce_sum(tf.square(Y-y_model))                                 10

train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost) 11
```

- ***1*** **Imports TensorFlow for the core learning algorithm, NumPy for manipulating data, and matplotlib for visualizing**

- ***2*** **Initializes fake data, 10 instances of each label**
- ***3*** **Initializes the corresponding labels**
- ***4*** **Plots the data**
- ***5*** **Declares the hyperparameters**
- ***6*** **Sets up the placeholder nodes for the input/output pairs**
- ***7*** **Defines a linear y = w1 * x + w0 model**
- ***8*** **Sets up the parameter variables**
- ***9*** **Defines a helper variable, because you'll refer to this multiple times**
- ***10*** **Defines the cost function**
- ***11*** **Defines the rule to learn the parameters**

After designing the TensorFlow graph, you'll see in the following listing how to open a new session and execute the graph. `train_op` updates the model's parameters to better and better guesses. You run `train_op` multiple times in a loop because each step iteratively improves the parameter estimate. The following listing generates a plot similar to figure 4.8.

Listing 4.2. Executing the graph

```
sess = tf.Session()                                              1
init = tf.global_variables_initializer()                         1
sess.run(init)                                                   1

for epoch in range(training_epochs):                             2
    sess.run(train_op, feed_dict={X: xs, Y: labels})             2
    current_cost = sess.run(cost, feed_dict={X: xs, Y: labels})  3
    if epoch % 100 == 0:
        print(epoch, current_cost)                               4

w_val = sess.run(w)                                              5
print('learned parameters', w_val)                               5

sess.close()                                                     6

all_xs = np.linspace(0, 10, 100)                                 7
plt.plot(all_xs, all_xs*w_val[1] + w_val[0])                     7
plt.show()                                                       7
```

- ***1*** **Opens a new session, and initializes the variables**
- ***2*** **Runs the learning operation multiple times**
- ***3*** **Records the cost computed with the current parameters**
- ***4*** **Prints out log info while the code runs**
- ***5*** **Prints the learned parameters**
- ***6*** **Closes the session when no longer in use**
- ***7*** **Shows the best-fit line**

To measure success, you can count the number of correct predictions and compute a success rate. In the next listing, you'll add two more nodes to the previous code in

linear.py, called `correct_prediction` and `accuracy`. You can then print the value of accuracy to see the success rate. The code can be executed right before closing the session.

Listing 4.3. Measuring accuracy

```
correct_prediction = tf.equal(Y, tf.to_float(tf.greater(y_model, 0.5)))   1
accuracy = tf.reduce_mean(tf.to_float(correct_prediction))                2

print('accuracy', sess.run(accuracy, feed_dict={X: xs, Y: labels}))       3
```

- ***1* When the model's response is greater than 0.5, it should be a positive label, and vice versa.**
- ***2* Computes the percent of success**
- ***3* Prints the success measure from provided input**

The preceding code produces the following output:

```
('learned parameters', array([ 1.2816, -0.2171], dtype=float32))
('accuracy', 0.95)
```

If classification were that easy, this chapter would be over by now. Unfortunately, the linear regression approach fails miserably if you train on more-extreme data, also called *outliers*.

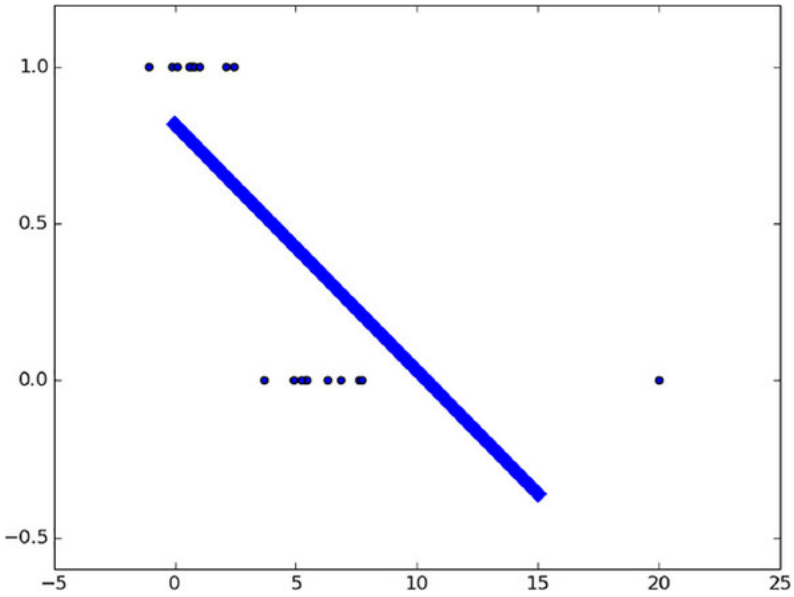
For example, let's say Alice lost a game that took 20 minutes. You train the classifier on a dataset that includes this new outlier data point. The following listing replaces one of the game times with the value of 20. Let's see how introducing an outlier affects the classifier's performance.

Listing 4.4. Linear regression failing miserably for classification

```
x_label0 = np.append(np.random.normal(5, 1, 9), 20)
```

When you rerun the code with these changes, you'll see a result similar to figure 4.9.

Figure 4.9. A new training element of value 20 greatly influences the best-fit line. The line is too sensitive to outlying data, and therefore linear regression is a sloppy classifier.



The original classifier suggested that you could beat Alice in a three-minute game. She'd probably agree to play such a short game. But the revised classifier, if you stick with the same 0.5 threshold, is now suggesting that the shortest game she'll lose is a five-minute game. She'll likely refuse to play such a long game!

4.4. USING LOGISTIC REGRESSION

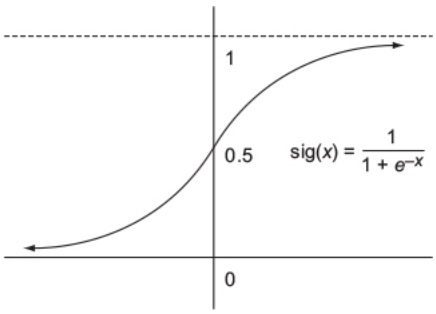
Logistic regression provides you with an analytic function with theoretical guarantees on accuracy and performance. It's just like linear regression, except you use a different cost function and slightly transform the model response function.

Let's revisit the linear function shown here:

$y(x) = wx$

In linear regression, a line with a nonzero slope may range from negative infinity to infinity. If the only sensible results for classification are 0 or 1, it would be intuitive to instead fit a function with that property. Fortunately, the sigmoid function depicted in figure 4.10 works well because it converges to 0 or 1 quickly.

Figure 4.10. A visualization of the sigmoid function



When x is 0, the sigmoid function results in 0.5. As x increases, the function converges to 1. And as x decreases to negative infinity, the function converges to 0.

In logistic regression, our model is sig(linear(x)). As it turns out, the best-fit parameters of this function imply a linear separation between the two classes. This separating line is also called a *linear decision boundary*.

4.4.1. Solving one-dimensional logistic regression

The cost function used in logistic regression is a bit different from the one you used in linear regression. Although you could use the same cost function as before, it won't be as fast or guarantee an optimal solution. The sigmoid function is the culprit here, because it causes the cost function to have many "bumps." TensorFlow and most other machine-learning libraries work best with simple cost functions. Scholars have found a neat way to modify the cost function to use sigmoids for logistic regression.

The new cost function between the actual value y and model response h will be a two-part equation as follows:

$$Cost(y, h) = \begin{cases} -\log(h), & \text{if } y = 1 \\ -\log(1 - h), & \text{if } y = 0 \end{cases}$$

You can condense the two equations into one long equation:

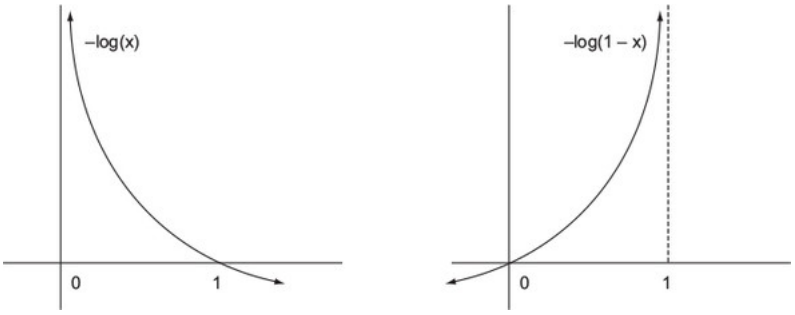
$$Cost(y, h) = -y \log(h) - (1 - y)\log(1 - h)$$

This function has exactly the qualities needed for efficient and optimal learning. Specifically, it's convex, but don't worry too much about what that means. You're trying to minimize the cost: think of cost as an altitude and the cost function as a terrain. You're trying to find the lowest point in the terrain. It's a lot easier to find the lowest point in the terrain if there's no place you can ever go uphill. Such a place is called *convex*. There are no hills.

You can think of it as a ball rolling down a hill. Eventually, the ball will settle to the bottom, which is the *optimal point*. A nonconvex function might have a rugged terrain, making it difficult to predict where a ball will roll. It might not even end up at the lowest point. Your function is convex, so the algorithm will easily figure out how to minimize this cost and "roll the ball downhill."

Convexity is nice, but correctness is also an important criterion when picking a cost function. How do you know this cost function does exactly what you intended it to do? To answer that question most intuitively, take a look at figure 4.11. You use $-\log(x)$ to compute the cost when you want your desired value to be 1 (notice: $-\log(1) = 0$). The algorithm strays away from setting the value to 0, because the cost approaches infinity. Adding these functions together gives a curve that approaches infinity at both 0 and 1, with the negative parts cancelling out.

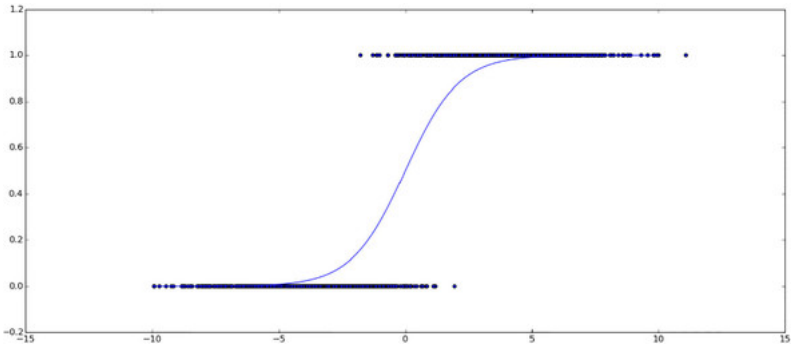
Figure 4.11. Here's a visualization of how the two cost functions penalize values at 0 and 1. Notice that the left function heavily penalizes 0 but has no cost at 1. The right cost function displays the opposite phenomena.



Sure, figures are an informal way to convince you, but the technical discussion about why the cost function is optimal is beyond the scope of this book. If you're interested in the mathematics behind it, you'll be interested to learn that the cost function is derived from the principle of maximum entropy, which you can look up anywhere online.

See figure 4.12 for a best-fit result from logistic regression on a one-dimensional dataset. The sigmoid curve that you'll generate will provide a better linear decision boundary than that from linear regression.

Figure 4.12. Here's a best-fit sigmoid curve for a binary classification dataset. Notice that the curve resides within $y = 0$ and $y = 1$. That way, this curve isn't that sensitive to outliers.



You'll start to notice a pattern in the code listings. In a simple/typical usage of TensorFlow, you generate a fake dataset, define placeholders, define variables, define a model, define a cost function on that model (which is often mean squared error or mean squared log error), create a `train_op` by using gradient descent, iteratively feed it example data (possibly with a label or output), and, finally, collect the optimized values. Create a new source file called logistic_1d.py and copy into it listing 4.5, which will generate figure 4.12.

Listing 4.5. Using one-dimensional logistic regression

```
import numpy as np                                                          1
import tensorflow as tf                                                     1
import matplotlib.pyplot as plt                                             1
learning_rate = 0.01                                                        2
training_epochs = 1000                                                      2

def sigmoid(x):                                                             3
    return 1. / (1. + np.exp(-x))                                           3

x1 = np.random.normal(-4, 2, 1000)                                          4
x2 = np.random.normal(4, 2, 1000)                                           4
xs = np.append(x1, x2)                                                      4
ys = np.asarray([0.] * len(x1) + [1.] * len(x2))                            4

plt.scatter(xs, ys)                                                         5

X = tf.placeholder(tf.float32, shape=(None,), name="x")                     6
Y = tf.placeholder(tf.float32, shape=(None,), name="y")                     6
w = tf.Variable([0., 0.], name="parameter", trainable=True)                 7
y_model = tf.sigmoid(w[1] * X + w[0])                                       8
cost = tf.reduce_mean(-Y * tf.log(y_model) - (1 - Y) * tf.log(1 – y_model))9

train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost) 10

with tf.Session() as sess:                                                  11
    sess.run(tf.global_variables_initializer())                             11
```

```
    prev_err = 0                                                    12
    for epoch in range(training_epochs):                            13
        err, _ = sess.run([cost, train_op], {X: xs, Y: ys})         14
        print(epoch, err)
        if abs(prev_err - err) < 0.0001:                            15
            break
        prev_err = err                                              16
    w_val = sess.run(w, {X: xs, Y: ys})                             17

all_xs = np.linspace(-10, 10, 100)                                  18
plt.plot(all_xs, sigmoid((all_xs * w_val[1] + w_val[0])))           18
plt.show()                                                          18
```

- ***1*** **Imports relevant libraries**
- ***2*** **Sets the hyperparameters**
- ***3*** **Defines a helper function to calculate the sigmoid function**
- ***4*** **Initializes fake data**
- ***5*** **Visualizes the data**
- ***6*** **Defines the input/output placeholders**
- ***7*** **Defines the parameter node**
- ***8*** **Defines the model using TensorFlow's sigmoid function**
- ***9*** **Defines the cross-entropy loss function**
- ***10*** **Defines the minimizer to use**
- ***11*** **Opens a session, and defines all variables**
- ***12*** **Defines a variable to keep track of the previous error**
- ***13*** **Iterates until convergence or until the maximum number of epochs is reached**
- ***14*** **Computes the cost, and updates the learning parameters**
- ***15*** **Checks for convergence—if you're changing by < .01% per iteration, you're done**
- ***16*** **Updates the previous error value**
- ***17*** **Obtains the learned parameter value**
- ***18*** **Plots the learned sigmoid function**

And there you have it! If you were playing chess against Alice, you'd now have a binary classifier to decide the threshold indicating when a chess match might result in a win or loss.

Cross-entropy loss in TensorFlow

As shown in listing 4.5, the cross-entropy loss is averaged over each input/output pair by using the `tf.reduce_mean` op. Another handy and more general function is provided by the TensorFlow library, called

`tf.nn.softmax_cross_entropy_with_logits`. You can find more about it in the official documentation: http://mng.bz/8mEk (http://mng.bz/8mEk).

4.4.2. Solving two-dimensional logistic regression

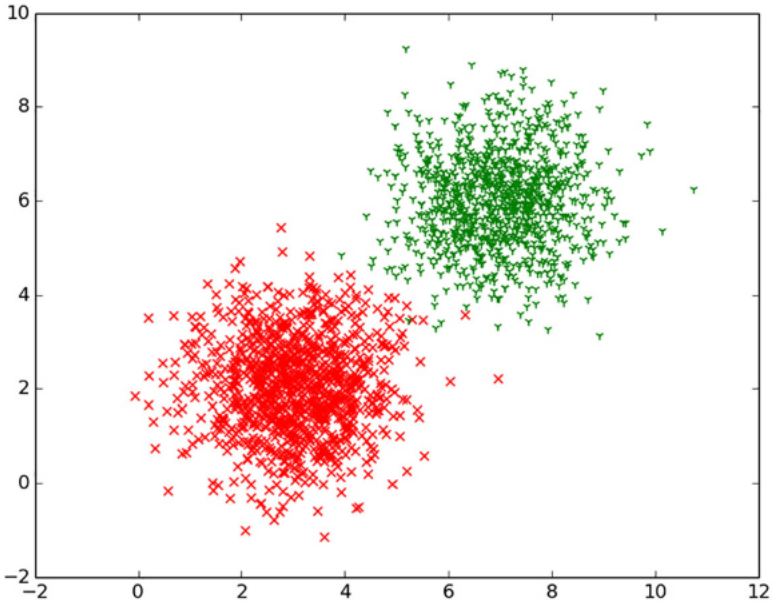
Now we'll explore how to use logistic regression with multiple independent variables. The number of independent variables corresponds to the number of dimensions. In our case, a two-dimensional logistic regression problem will try to label a pair of independent variables. The concepts you learn in this section extrapolate to arbitrary dimensions.

Note

Let's say you're thinking about buying a new phone. The only attributes you care about are (1) operating system, (2) size, and (3) cost. The goal is to decide whether a phone is a worthwhile purchase. In this case, there are three independent variables (the attributes of the phone) and one dependent variable (whether it's worth buying). So we regard this as a classification problem in which the input vector is three-dimensional.

Consider the dataset shown in figure 4.13. It represents crime activity of two gangs in a city. The first dimension is the x-axis, which can be thought of as the latitude, and the second dimension is the y-axis, representing longitude. There's one cluster around (3, 2) and another around (7, 6). Your job is to decide which gang is most likely responsible for a new crime that occurred at location (6, 4).

Figure 4.13. The *x*-axis and *y*-axis represent the two independent variables. The dependent variable holds two possible labels, represented by the shape and color of the plotted points.



Create a new source file called logistic_2d.py, and follow along with listing 4.6.

Listing 4.6. Setting up data for two-dimensional logistic regression

```
import numpy as np                                                    1
import tensorflow as tf                                               1
import matplotlib.pyplot as plt                                       1

learning_rate = 0.1                                                   2
training_epochs = 2000                                                2

def sigmoid(x):                                                       3
    return 1. / (1. + np.exp(-x))                                     3

x1_label1 = np.random.normal(3, 1, 1000)                              4
x2_label1 = np.random.normal(2, 1, 1000)                              4
x1_label2 = np.random.normal(7, 1, 1000)                              4
x2_label2 = np.random.normal(6, 1, 1000)                              4
x1s = np.append(x1_label1, x1_label2)                                 4
x2s = np.append(x2_label1, x2_label2)                                 4
ys = np.asarray([0.] * len(x1_label1) + [1.] * len(x1_label2))        4
```

- ***1*** **Imports relevant libraries**
- ***2*** **Sets the hyperparameters**
- ***3*** **Defines a helper sigmoid function**
- ***4*** **Initializes fake data**

You have two independent variables ($x1$ and $x2$). A simple way to model the mapping between the input x's and output $M(x)$ is the following equation, where w is the parameter to be found using TensorFlow:

$$M(x; w) = sig(w_2 x_2 + w_1 x_1 + w_0)$$

In the following listing, you'll implement the equation and its corresponding cost function to learn the parameters.

Listing 4.7. Using TensorFlow for multidimensional logistic regression

```
X1 = tf.placeholder(tf.float32, shape=(None,), name="x1")                  1
X2 = tf.placeholder(tf.float32, shape=(None,), name="x2")                  1
Y = tf.placeholder(tf.float32, shape=(None,), name="y")                    1
w = tf.Variable([0., 0., 0.], name="w", trainable=True)                    2

y_model = tf.sigmoid(w[2] * X2 + w[1] * X1 + w[0])                         3
cost = tf.reduce_mean(-tf.log(y_model * Y + (1 - y_model) * (1 - Y)))      4
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost) 4
with tf.Session() as sess:                                                 5
    sess.run(tf.global_variables_initializer())                            5
    prev_err = 0                                                           5
    for epoch in range(training_epochs):                                   5
        err, _ = sess.run([cost, train_op], {X1: x1s, X2: x2s, Y: ys})     5
        print(epoch, err)                                                  5
        if abs(prev_err - err) < 0.0001:                                   5
            break                                                          5
        prev_err = err                                                     5
    w_val = sess.run(w, {X1: x1s, X2: x2s, Y: ys})                         6

x1_boundary, x2_boundary = [], []                                          7
for x1_test in np.linspace(0, 10, 100):                                    8
    for x2_test in np.linspace(0, 10, 100):                                8
        z = sigmoid(-x2_test*w_val[2] - x1_test*w_val[1] - w_val[0])       9
        if abs(z - 0.5) < 0.01:                                            9
            x1_boundary.append(x1_test)                                    9
            x2_boundary.append(x2_test)                                    9
```

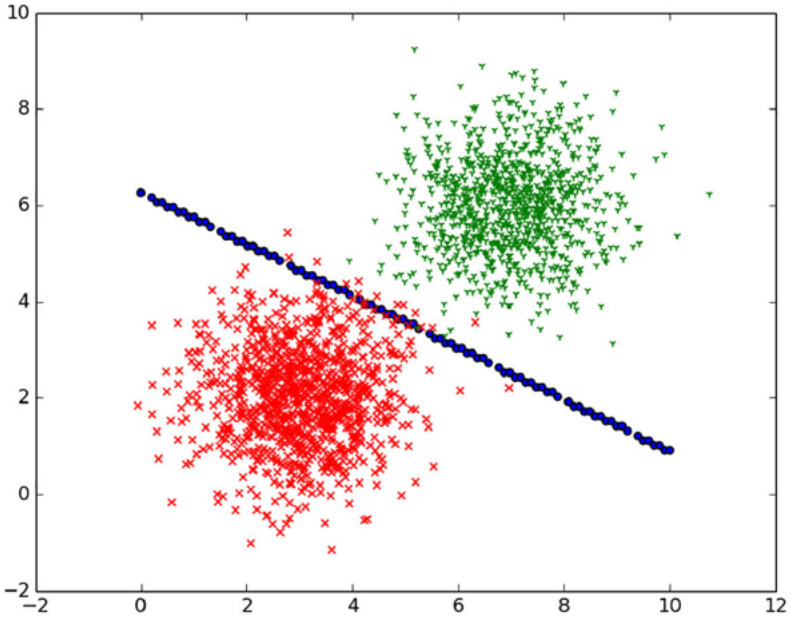
```
plt.scatter(x1_boundary, x2_boundary, c='b', marker='o', s=20)
plt.scatter(x1_label1, x2_label1, c='r', marker='x', s=20)
plt.scatter(x1_label2, x2_label2, c='g', marker='1', s=20)

plt.show()
```

- ***1*** **Defines the input/output placeholder nodes**
- ***2*** **Defines the parameter node**
- ***3*** **Defines the sigmoid model using both input variables**
- ***4*** **Defines the learning step**
- ***5*** **Creates a new session, initializes variables, and learns parameters until convergence**
- ***6*** **Obtains the learned parameter value before closing the session**
- ***7*** **Defines arrays to hold boundary points**
- ***8*** **Loops through a window of points**
- ***9*** **If the model response is close the 0.5, updates the boundary points**
- ***10*** **Shows the boundary line along with the data**

Figure 4.14 depicts the linear boundary line learned from the training data. A crime that occurs on this line has an equal chance of being committed by either gang.

Figure 4.14. The diagonal dotted line represents when the probability between the two decisions is split equally. The confidence of making a decision increases as data lies farther away from the line.



4.5. MULTICLASS CLASSIFIER

So far, you've dealt with multidimensional input, but not multivariate output, as shown in figure 4.15. For example, instead of binary labels on the data, what if you have 3, or 4, or 100 classes? Logistic regression requires two labels, no more.

Figure 4.15. The independent variable is two-dimensional, indicated by the *x*-axis and *y*-axis. The dependent variable can be one of three labels, shown by the color and shape of the data points.

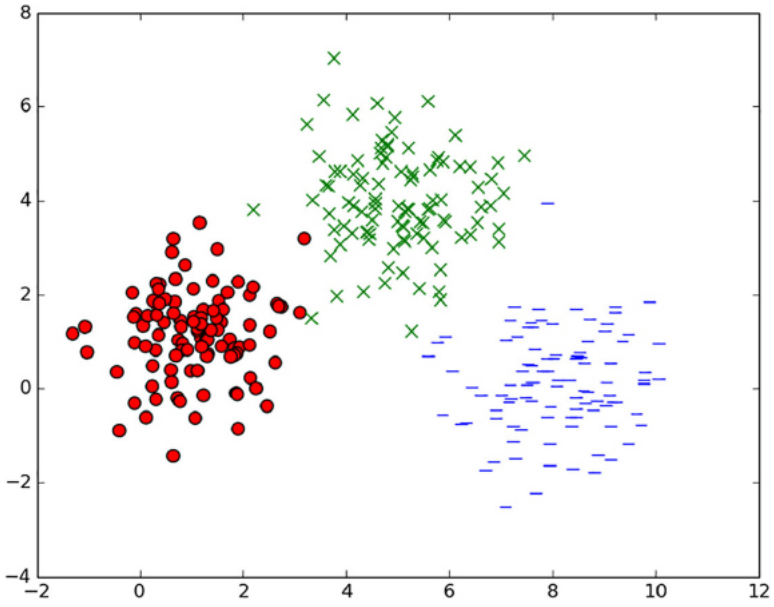


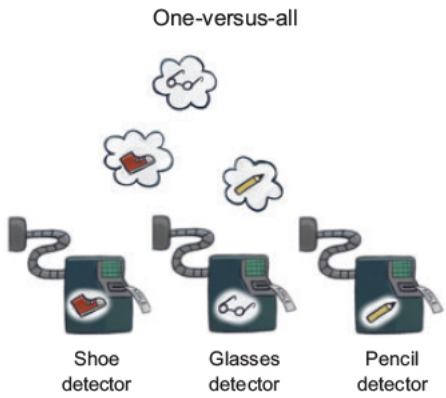
Image classification, for example, is a popular multivariate classification problem because the goal is to decide the class of an image from a collection of candidates. A photograph may be bucketed into one of hundreds of categories.

To handle more than two labels, you may reuse logistic regression in a clever way (using a one-versus-all or one-versus-one approach) or develop a new approach (softmax regression). Let's look at each of the approaches in the next sections. The logistic regression approaches require a decent amount of ad hoc engineering, so let's focus on softmax regression.

4.5.1. One-versus-all

First, you train a classifier for each of the labels, as shown in figure 4.16. If there are three labels, you have three classifiers available to use: `f1`, `f2`, and `f3`. To test on new data, you run each of the classifiers to see which one produced the most confident response. Intuitively, you label the new point by the label of the classifier that responded most confidently.

Figure 4.16. One-versus-all is a multiclass classifier approach that requires a detector for each class.

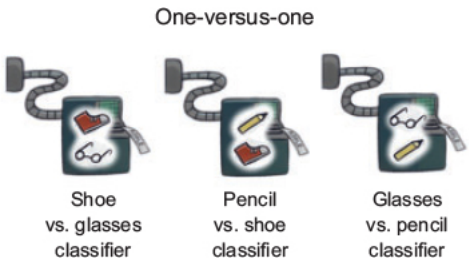


4.5.2. One-versus-one

Then you train a classifier for each pair of labels (see figure 4.17). If there are three labels, that's just three unique pairs. But for k number of labels, that's $k(k-1)/2$ pairs

of labels. On new data, you run all the classifiers and choose the class with the most wins.

Figure 4.17. In one-versus-one multiclass classification, there's a detector for each pair of classes.



4.5.3. Softmax regression

Softmax regression is named after the traditional `max` function, which takes a vector and returns the max value; but softmax isn't exactly the `max` function, because it has the added benefit of being continuous and differentiable. As a result, it has the helpful properties for stochastic gradient descent to work efficiently.

In this type of multiclass classification setup, each class has a confidence (or probability) score for each input vector. The softmax step picks the highest-scoring output.

Open a new file called softmax.py, and follow along with the next listing. First, you'll visualize fake data to reproduce figure 4.15 (also reproduced in figure 4.18).

Listing 4.8. Visualizing multiclass data

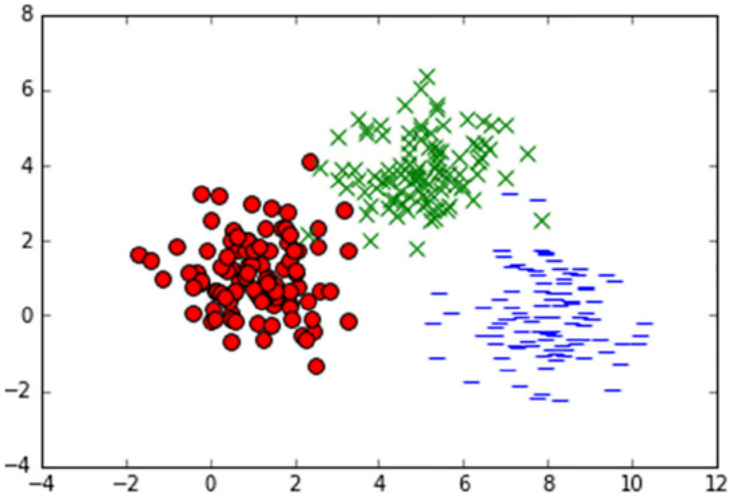
```
import numpy as np                                                    1
import matplotlib.pyplot as plt                                       1

x1_label0 = np.random.normal(1, 1, (100, 1))                          2
x2_label0 = np.random.normal(1, 1, (100, 1))                          2
x1_label1 = np.random.normal(5, 1, (100, 1))                          3
x2_label1 = np.random.normal(4, 1, (100, 1))                          3
x1_label2 = np.random.normal(8, 1, (100, 1))                          4
x2_label2 = np.random.normal(0, 1, (100, 1))                          4

plt.scatter(x1_label0, x2_label0, c='r', marker='o', s=60)            5
plt.scatter(x1_label1, x2_label1, c='g', marker='x', s=60)            5
plt.scatter(x1_label2, x2_label2, c='b', marker='_', s=60)            5
plt.show()                                                            5
```

- ***1*** **Imports NumPy and matplotlib**
- ***2*** **Generates points near (1, 1)**
- ***3*** **Generates points near (5, 4)**
- ***4*** **Generates points near (8, 0)**
- ***5*** **Visualizes the three labels on a scatter plot**

Figure 4.18. 2D training data for multi-output classification



Next, in listing 4.9, you'll set up the training and test data to prepare for the softmax regression step. The labels must be represented as a vector in which only one element is 1 and the rest are 0s. This representation is called *one-hot encoding*. For instance, if there are three labels, they'd be represented as the following vectors: [1, 0, 0], [0, 1, 0], and [0, 0, 1].

Exercise 4.5

One-hot encoding might appear to be an unnecessary step. Why not just have a one-dimensional output with values of 1, 2, and 3 representing the three classes?

ANSWER

Regression may induce a semantic structure in the output. If outputs are similar, regression implies that their inputs were also similar. If you use just one dimension, you're implying that labels 2 and 3 are more similar to each other than 1 and 3. You must be careful about making unnecessary or incorrect assumptions, so it's a safe bet to use one-hot encoding.

Listing 4.9. Setting up training and test data for multiclass classification

```
xs_label0 = np.hstack((x1_label0, x2_label0))                                  1
xs_label1 = np.hstack((x1_label1, x2_label1))                                  1
xs_label2 = np.hstack((x1_label2, x2_label2))                                  1
xs = np.vstack((xs_label0, xs_label1, xs_label2))                              1

labels = np.matrix([[1., 0., 0.]] * len(x1_label0) + [[0., 1., 0.]] *
     len(x1_label1) + [[0., 0., 1.]] * len(x1_label2))                         2

arr = np.arange(xs.shape[0])                                                   3
np.random.shuffle(arr)                                                         3
xs = xs[arr, :]                                                                3
labels = labels[arr, :]                                                        3

test_x1_label0 = np.random.normal(1, 1, (10, 1))                               4
test_x2_label0 = np.random.normal(1, 1, (10, 1))                               4
test_x1_label1 = np.random.normal(5, 1, (10, 1))                               4
test_x2_label1 = np.random.normal(4, 1, (10, 1))                               4
test_x1_label2 = np.random.normal(8, 1, (10, 1))                               4
test_x2_label2 = np.random.normal(0, 1, (10, 1))                               4
test_xs_label0 = np.hstack((test_x1_label0, test_x2_label0))                   4
```

```
test_xs_label1 = np.hstack((test_x1_label1, test_x2_label1))                4
test_xs_label2 = np.hstack((test_x1_label2, test_x2_label2))                4

test_xs = np.vstack((test_xs_label0, test_xs_label1, test_xs_label2))       4
test_labels = np.matrix([[1., 0., 0.]] * 10 + [[0., 1., 0.]] * 10 + [[0., 0.,
     1.]] * 10)                                                             4

train_size, num_features = xs.shape                                         5
```

- ***1*** **Combines all input data into one big matrix**
- ***2*** **Creates the corresponding one-hot labels**
- ***3*** **Shuffles the dataset**
- ***4*** **Constructs the test dataset and labels**
- ***5*** **The shape of the dataset tells you the number of examples and features per example.**

Finally, in listing 4.10, you'll use softmax regression. Unlike the sigmoid function in logistic regression, here you'll use the `softmax` function provided by the TensorFlow library. The `softmax` function is similar to the `max` function, which outputs the maximum value from a list of numbers. It's called softmax because it's a "soft" or "smooth" approximation of the `max` function, which is not smooth or continuous (and that's bad). Continuous and smooth functions facilitate learning the correct weights of a neural network by back-propagation.

Exercise 4.6

Which of the following functions is continuous?

```
f(x) = x2
f(x) = min(x, 0)
f(x) = tan(x)
```

Answer

The first two are continuous. The last one, `tan(x)`, has periodic asymptotes, so there are some values for which there are no valid results.

Listing 4.10. Using softmax regression

```
import tensorflow as tf

learning_rate = 0.01                                                        1
training_epochs = 1000                                                      1
num_labels = 3                                                              1
batch_size = 100                                                            1

X = tf.placeholder("float", shape=[None, num_features])                     2
Y = tf.placeholder("float", shape=[None, num_labels])                       2

W = tf.Variable(tf.zeros([num_features, num_labels]))                       3
b = tf.Variable(tf.zeros([num_labels]))                                     3
```

```
y_model = tf.nn.softmax(tf.matmul(X, W) + b)                                    4

cost = -tf.reduce_sum(Y * tf.log(y_model))                                      5
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)      5

correct_prediction = tf.equal(tf.argmax(y_model, 1), tf.argmax(Y, 1))           6
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))                 6
```

- *1* **Defines hyperparameters**
- *2* **Defines the input/output placeholder nodes**
- *3* **Defines the model parameters**
- *4* **Designs the softmax model**
- *5* **Sets up the learning algorithm**
- *6* **Defines an op to measure success rate**

Now that you've defined the TensorFlow computation graph, execute it from a session. You'll try a new form of iteratively updating the parameters this time, called *batch learning*. Instead of passing in the data one piece at a time, you'll run the optimizer on batches of data. This speeds things up but introduces a risk of converging to a local optimum solution instead of the global best. Use the following listing for running the optimizer in batches.

Listing 4.11. Executing the graph

```
with tf.Session() as sess:                                                      1
    tf.global_variables_initializer().run()                                     1

    for step in range(training_epochs * train_size // batch_size):              2
        offset = (step * batch_size) % train_size                               3
        batch_xs = xs[offset:(offset + batch_size), :]                          3
        batch_labels = labels[offset:(offset + batch_size)]
        err, _ = sess.run([cost, train_op], feed_dict={X: batch_xs, Y:
     batch_labels})                                                             4
        print (step, err)                                                       5

    W_val = sess.run(W)                                                         6
    print('w', W_val)                                                           6
    b_val = sess.run(b)                                                         6
    print('b', b_val)                                                           6
    print("accuracy", accuracy.eval(feed_dict={X: test_xs, Y: test_labels}))7
```

- *1* **Opens a new session and initializes all variables**
- *2* **Loops only enough times to complete a single pass through the dataset**
- *3* **Retrieves a subset of the dataset corresponding to the current batch**
- *4* **Runs the optimizer on this batch**
- *5* **Prints ongoing results**
- *6* **Prints the final learned parameters**
- *7* **Prints the success rate**

The final output of running the softmax regression algorithm on the dataset is the following:

```
('w', array([[-2.101, -0.021,  2.122],
             [-0.371,  2.229, -1.858]], dtype=float32))
('b', array([10.305, -2.612, -7.693], dtype=float32))
Accuracy 1.0
```

You've learned the weights and biases of the model. You can reuse these learned parameters to infer on test data. A simple way to do so is by saving and loading the variables using TensorFlow's `Saver` object (see www.tensorflow.org/programmers_guide/saved_model (http://www.tensorflow.org/programmers_guide/saved_model)). You can run the model (called `y_model` in our code) to obtain the model responses on your test input data.

4.6. APPLICATION OF CLASSIFICATION

Emotion is a difficult concept to operationalize. Happiness, sadness, anger, excitement, and fear are examples of emotions that are subjective. What comes across as exciting to someone might appear sarcastic to another. Text that appears to convey anger to some might convey fear to others. If humans have so much trouble, what luck can computers have?

At the very least, machine-learning researchers have figured out ways to classify positive and negative sentiments within text. For example, let's say you're building an Amazon-like website in which each item has user reviews. You want your intelligent search engine to prefer items with positive reviews. Perhaps the best metric you have available is the average star rating or number of thumbs-ups. But what if you have a lot of heavy-text reviews without explicit ratings?

Sentiment analysis can be considered a binary classification problem. The input is natural language text, and the output is a binary decision that infers positive or negative sentiment. The following are datasets you can find online to solve this exact problem:

- Large Movie Review Dataset: http://mng.bz/60nj (http://mng.bz/60nj)
- Sentiment Labelled Sentences Data Set: http://mng.bz/CzSM (http://mng.bz/CzSM)
- Twitter Sentiment Analysis Dataset: http://mng.bz/2M4d (http://mng.bz/2M4d)

The biggest hurdle is to figure out how to represent raw text as an input to a classification algorithm. Throughout this chapter, the input to classification has always been a feature vector. One of the oldest methods of converting raw text into a feature vector is called *bag-of-words*. You can find a nice tutorial and code implementation for it here: http://mng.bz/K8yz (http://mng.bz/K8yz).

4.7. SUMMARY

- There are many ways to solve classification problems, but logistic regression and softmax regression are two of the most robust in terms of accuracy and performance.

- It's important to preprocess data before running classification. For example, discrete independent variables can be readjusted into binary variables.
- So far, you've approached classification from the point of view of regression. In later chapters, you'll revisit classification using neural networks.
- There are various ways to approach multiclass classification. There's no clear answer to which one you should try first among one-versus-one, one-versus-all, and softmax regression. But the softmax approach is a little more hands-free and allows you to fiddle with more hyperparameters.

PREV
Chapter 3. Linear regression and beyond

NEXT
Chapter 5. Automatically clustering data