Machine Learning in Python®: Essential Techniques for Predictive Analysis
By Michael Bowles

CHAPTER

3

Predictive Model Building: Balancing Performance, Complexity, and Big Data

This chapter discusses the factors affecting the performance of machine learning models. The chapter provides technical definitions of *performance* for different types of machine learning problems. In an e-commerce application, for example, good performance might mean returning correct search results or presenting ads that site visitors frequently click. In a genetic problem, it might mean isolating a few genes responsible for a heritable condition. The chapter describes relevant performance measures for these different problems.

The goal of selecting and fitting a predictive algorithm is to achieve the best possible performance. Achieving performance goals involves three factors: complexity of the problem, complexity of the algorithmic model employed, and the amount and richness of the data available. The chapter includes some visual examples that demonstrate the relationship between problem and model complexity and then provides technical guidelines for use in design and development.

The Basic Problem: Understanding Function Approximation

The algorithms covered in this book address a specific class of predictive problem. The problem statement for these problems has two types of variables:

- The variable that you are attempting to predict (for example, whether a visitor to a website will click an ad)
- Other variables (for example, the visitor's demographics or past behavior on the site) that you can use to make the prediction

Problems of this type are referred to as *function approximation problems* because the goal is to construct a model generating predictions of the first of these as a function of the second.

In a function approximation problem, the designer starts with a collection of historical examples for which the correct answer is known. For example, historical web log files will indicate whether a visitor clicked an ad when shown the ad. The data scientist next has to find other data that can be used to build a predictive model. For example, to predict whether a site visitor will click an ad, the data scientist might try using other pages that the visitor viewed before seeing the ad. If the user is registered with the site, data on past purchases or pages viewed might be available for making a prediction.

The variable being predicted is referred to by a number of different names, such as *target*, *label*, and *outcome*. The variables being used to make the predictions are variously called *predictors*, *regressors*, *features*, and *attributes*. These terms are used interchangeably in this text, as they are in general practice. Determining what attributes to use for making predictions is called *feature engineering*. Data cleaning and feature engineering take 80 percent to 90 percent of a data scientist's time.

Feature engineering usually requires a manual, iterative process for selecting features, determining optimal potential, and experimenting with different combinations of features. The algorithms covered in this book assign numeric importance values to each attribute. These values indicate the relative importance of attributes in making predictions. That information helps speed up the feature engineering process.

Working with Training Data

The data scientist starts algorithm development with a training set. The training set consists of outcome examples and the assemblage of features chosen by the data scientist. The training set comprises two types of data:

- The outcomes you want to predict
- The features available for making the prediction

Table 3-1 provides an example of a training set. The leftmost column contains outcomes (whether a site visitor clicked a link) and features to be used to make predictions about whether visitors will click the link in the future.

Table 3-1: *Example Training Set*

OUTCOMES: CLICKED ON LINK	FEATURE1: GENDER	FEATURE2: MONEY SPENT ON SITE	FEATURE3: GENDER
Yes	M	0	25
No	F	250	32
Yes	F	12	17

The predictor values (a.k.a., features, attributes, and so on) can be arranged in the form of a matrix (see Equation 3-1). The notational convention used in this book is as follows. The table of predictors will be called *X*, and it has the following form:

$$X = \begin{matrix} x_{11} & x_{12} & x_{1n} \\ x_{21} & x_{22} & x_{2n} \\ \vdots & \vdots & \ddots \\ x_{m1} & x_{m2} & x_{mn} \end{matrix}$$

Equation 3-1: Notation for set of predictors

Referring to the data set in Table 3-1, x_{11} would be M (gender), x_{12} would be 0.00 (money spent on site), x_{21} would be F (gender), and so on.

Sometimes it will be convenient to refer to all the attribute values for a particular example. For that purpose, x_i (with a single index) will refer to the *i*th row of X. For the data set in Table 3-1, x_2 would be a row vector containing the values F, 250, 32.

Strictly speaking, the X is not a matrix because the predictors may not all be the same type of variable. (A proper matrix contains variables that are all the same type. Predictors, however come in different types.) Using the example of predicting ad clicks, the predictors might include demographic data about the site visitor. Those data could include marital status and yearly income, among other things. Yearly income is a real number, and marital status is a categorical variable. That means that marital status does not admit arithmetic operations such as adding or multiplication and that no order relation exists between *single*, *married*, and *divorced*. The entries in a column from X all have the same type, but the type may vary from one column to the next.

Attributes such as marital status, gender, or the state of residence go by several different designations. They may be called *factor* or *categorical*. Attributes like age or income that are represented by numbers are called *numeric* or *real-valued*.

The distinction between these two types of attributes is important because some algorithms may not handle one type or the other. For example, linear methods, including the ones covered in this book, require numeric attributes. (Chapter 4, "Penalized Linear Regression," which covers linear methods, shows methods for converting [or coding] categorical variables to numeric in order to apply linear methods to problems with categorical variables.)

The targets corresponding to each row in X are arranged in a column vector Y (see Equation 3-2), as follows:

$$Y = \begin{matrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{matrix}$$

Equation 3-2: Notation for vector of targets

The target y_i corresponds to x_i—the predictors in the *i*th row of X. Referring to the data in Table 3-1, y_1 is Yes, and y_2 is No.

Targets may be of several different forms. For example, they may be real numbers, like if the objective were to predict how much a customer will spend. When the targets are real numbers, the problem is called a *regression problem*. Linear regression implies using a linear method to solve a regression problem. (This book covers both linear and nonlinear regression methods.)

If the targets are two-valued, as in Table 3-1, the problem is called a *binary classification problem*. Predicting whether a customer will click an advertisement is a binary classification problem. If the targets contain several discrete values, the problem is a *multiclass classification problem*. Predicting which of several ads a customer will click would be a multiclass classification problem.

The basic problem is to find a prediction function, `pred()`, that uses the attributes to predict outcomes (see Equation 3-3):

$$y_t \sim pred(x_t)$$

Equation 3-3: Basic equation for making predictions

The function `pred()` uses the attribute x_i to predict y_i. This book describes some of the very best current methods for producing the function `pred()`.

Assessing Performance of Predictive Models

Good performance means using the attributes x_i to generate a prediction that is close to y_i, but *close* has different meanings for different problems. For a regression problem where y_i is a real number, performance is measured in terms like the mean squared error (MSE) or the mean absolute error (MAE) (see Equation 3-4).

$$Mean\,squared\,error = \left(\frac{1}{m}\right)\sum_{i=1}^{m}\left(y_i - pred\left(x_i\right)\right)^2$$

Equation 3-4: Performance measure for a regression problem

In a regression problem, the target (y_i) and the prediction, pred(x_i), are both real numbers, so it makes sense to describe the error as being the numeric difference between them. Equation 3-4 for MSE squares the errors and averages over the data set to produce a measure of the overall level of errors. MAE averages the absolute values of the errors (see Equation 3-5) instead of averaging the squares of the errors.

$$Mean\,absolute\,error = \left(\frac{1}{m}\right)\sum_{i=1}^{m}\left|y_i - pred\left(x_i\right)\right|$$

Equation 3-5: Another performance measure for regression

If the problem is a classification problem, you must use some other measure of performance. One of the most used is the misclassification error—that is, the fraction of examples that the function `pred()` predicts incorrectly. The section "Performance Measures for Different Types of Problems" shows how to calculate misclassification.

For our function `pred()` to be useful for making predictions, there must be some way to predict what level of errors it will generate on new examples as they arrive. What is the performance on new data—data that were not involved in developing the function `pred()`? This chapter covers the best methods for estimating performance on new data.

This section introduced the basic type of prediction problem that will be addressed in this book and described how constructing these prediction models amounts to constructing a function that maps attributes (or features) into predicted outcomes. It also gave an overview of how the errors in these predictions can be assessed. Performing these steps leads to several complications. The remaining sections of this chapter describe these complications, how to deal with them, and how to arrive at the best possible model given the constraints of the problem and the data available.

Factors Driving Algorithm Choices and Performance—Complexity and Data

Several factors affect the overall performance of a predictive algorithm. Among these factors are the complexity of the problem, the complexity of the model used, and the amount of training data available. The following sections describe how these factors interrelate to determine performance.

Contrast Between a Simple Problem and a Complex Problem

The preceding section of this chapter described several ways to quantify performance and highlighted the importance of performance on new data. The goal of designing a predictive model is to make accurate predictions on new examples (such as new visitors to your site). As a practicing data scientist, you will want an estimate of an algorithm's performance so that you can set expectations with your customer and compare algorithms with one another. Best practice in predictive modeling requires that you hold out some data from the training set. These held-out examples have labels associated with them and can be compared to predictions produced by models training on the remaining data. Statisticians refer to this technique as *out-of-sample error* because it is an error on data not used in training. (The section "Measuring Performance of Predictive Models" later in this chapter goes into more detail about the mechanics of this process.) The important thing is that the only performance that counts is the performance of the model when it is run against new examples.

One of the factors affecting performance is the complexity of the problem being solved. Figure 3-1 shows a relatively simple classification problem in two dimensions. There are two groups of points: dark and light points. The dark points are randomly drawn from a 2D Gaussian distribution centered at (1,0) with unit variance in both dimensions. The light points are also drawn from a Gaussian distribution having the same variance but centered at (0,1). The

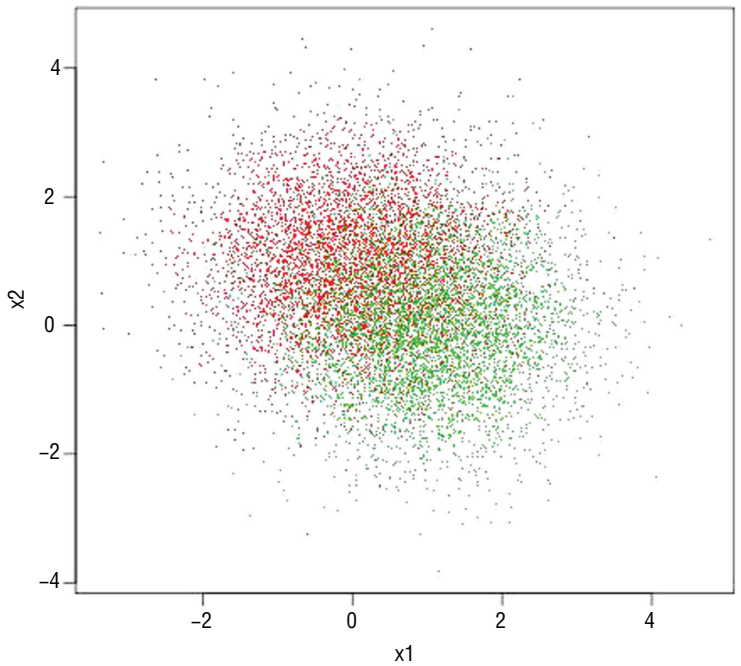

Figure 3-1: A simple classification problem

attributes for the problem are the two axes in the plot: x_1 and x_2. The classification task is to draw some boundaries in the x_1, x_2 plane to separate the light points from the dark points. About the best that can be done in this circumstance is to draw a 45-degree line in the plot—that is the line where x_1 equals x_2. In a precise probabilistic sense, that is the best possible classifier for this problem. Because a straight line separates the lights and darks as well as possible, a linear classifier will do as well as nonlinear classifier. The linear methods covered in this book will do a splendid job on this problem.

Figure 3-2 depicts a more complicated problem. The points shown in Figure 3-2 are generated by drawing points at random. The main difference from the random draw that generated Figure 3-1 is that the points in Figure 3-2 are drawn from several distributions for the light points and several different ones for dark. This is called a *mixture model*. The general goal is basically the same: draw boundaries in the x_1, x_2 plane to separate the light points from the dark points. In Figure 3-2, however, it is clear that a linear boundary will not separate the points as well as a curve. The ensemble methods covered in Chapter 6, "Ensemble Methods," will work well on this problem.

However, complexity of the decision boundaries is not the only factor influencing whether linear or nonlinear methods will deliver better performance. Another important factor is the size of the data set. Figure 3-3 illustrates this element of performance. The points plotted in Figure 3-3 are a 1 percent subsample of data plotted in Figure 3-2.

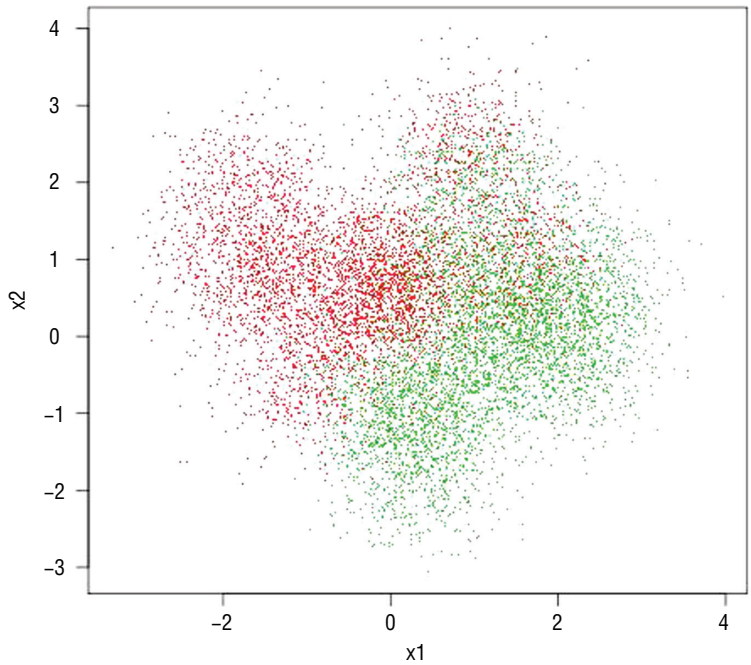

Figure 3-2: A complicated classification problem

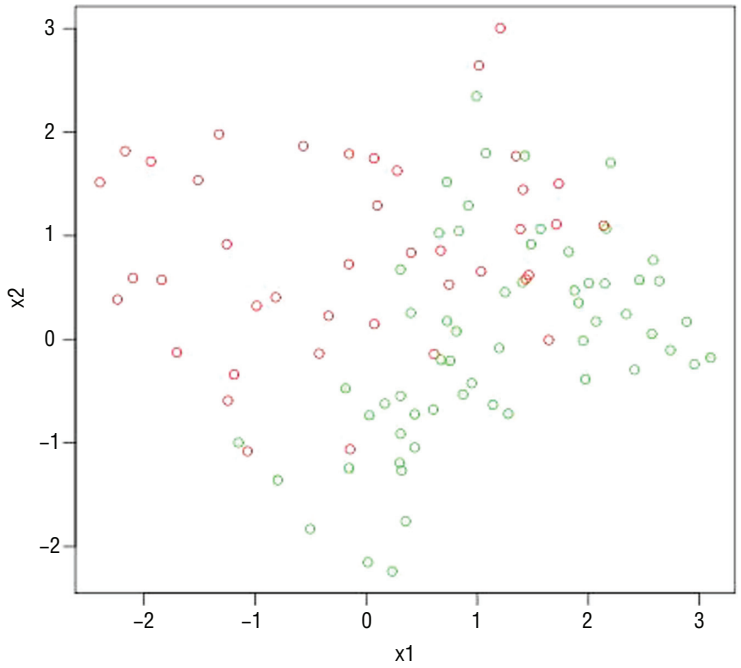

Figure 3-3: A complicated classification problem without much data

In Figure 3-2, there was enough data to visualize the curved boundaries delineating the sets of light and dark points. Without as much data, the sets are not so easily discerned visually, and in this circumstance, a linear model may give equal or better performance than a nonlinear model. With less data, the boundaries are harder to visualize, and they are more difficult to compute. This gives a graphic demonstration of the value of having a large volume of data. If the underlying problem is complicated (for example, personalizing responses for individual shoppers), a complicated model with a lot of data can produce accurate results. However, if the model is not complicated, as in Figure 3-1, or there is not sufficient data, as in Figure 3-3, a linear model may produce the best answer.

Contrast Between a Simple Model and a Complex Model

The previous section showed visual comparisons between simple and complex problems. This section describes how the various models available to solve these problems differ from one another. Intuitively, it seems that a complex model should be fit to a complex problem, but the visual example from the last section demonstrates that data set size may dictate that a simple model fits a complex problem better than a complex model.

Another important concept is that modern machine learning algorithms generate families of models, not just single models. The algorithms covered

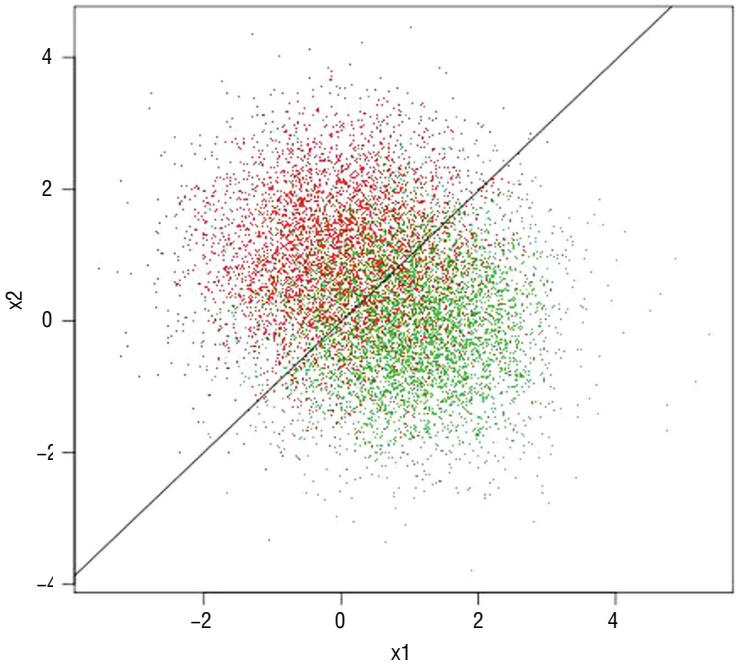

Figure 3-4: Linear model fit to simple data

in this book each generate hundreds or even thousands of different models. Generally, the ensemble methods covered in Chapter 6 yield more complex models than linear methods covered in Chapter 4, but both of these methods generate multiple models of varying complexity. (This will become clearer in Chapters 4 and 6, which cover linear and ensemble techniques in detail.)

Figure 3-4 shows a linear model fit to the simple problem introduced in the previous section. The linear model shown in Figure 3-4 was generated using the glmnet algorithm (covered in Chapter 4). The linear model fit to these data divides the data roughly in half. The line in the figure is given by Equation 3-6.

$$x_2 = -0.01 + 0.99x_1$$

Equation 3-6: Linear model fit to simple problem

This is very close to the line where x_2 equals x_1, which is the best possible boundary in a probabilistic sense. The boundary appears sensible from a visual intuitive standpoint. Fitting a more complicated model to this simple problem is not going to improve performance.

A more complicated problem with more complicated decision boundaries gives a complicated model an opportunity to outperform a simple linear model.

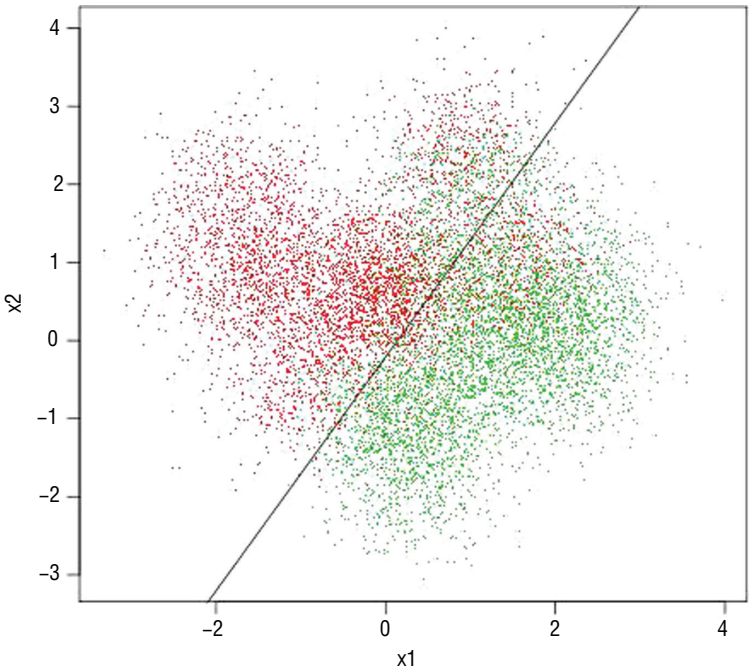

Figure 3-5: Linear model fit to complex data

Figure 3-5 shows a linear model fit to data indicating a nonlinear decision boundary. In this circumstance, the linear model misclassifies regions as dark when they should be light and vice versa.

Figure 3-6 shows how much better a complicated model can do with complicated data. The model used to generate this decision boundary is an ensemble (collection) of 1,000 binary decision trees constructed using the gradient boosting algorithm. (Gradient boosted decision trees are covered in detail in Chapter 6 on ensemble methods.) The nonlinear decision boundary curves are used to better delineate regions where the dark points are denser and regions where the light points are denser.

It is tempting to draw the conclusion that the best approach is to use complicated models for complicated problems and simple models for simple problems. But, you must consider one more dimension to the problem. As mentioned in the previous section, you must consider data set size. Figures 3-7 and 3-8 show 1 percent of the data from a complicated problem. Figure 3-7 shows a linear model fit to the data, and Figure 3-8 shows an ensemble model fit to the data. Count the number of points that are misclassified. There are 100 points in the data set. The linear model in Figure 3-7 misclassifies 11 points, for a misclassification error rate of 11 percent. The complex model misclassifies 8, for an 8 percent error rate. Their performance is roughly equal.

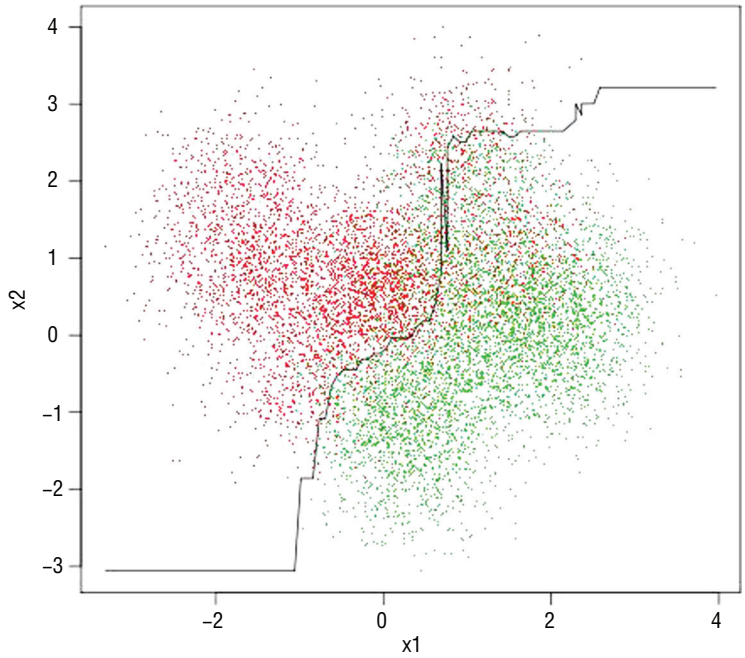

Figure 3-6: Ensemble model fit to complex data

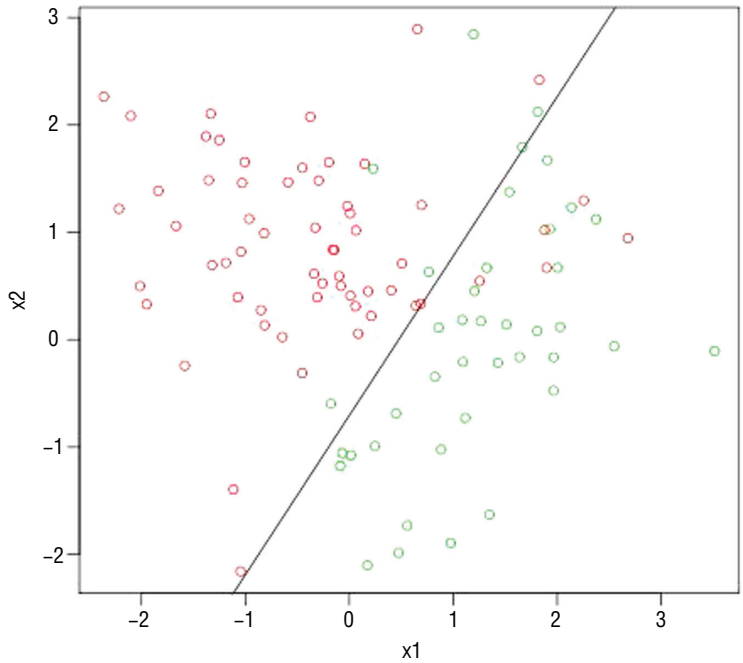

Figure 3-7: Linear model fit to small sample of complex data

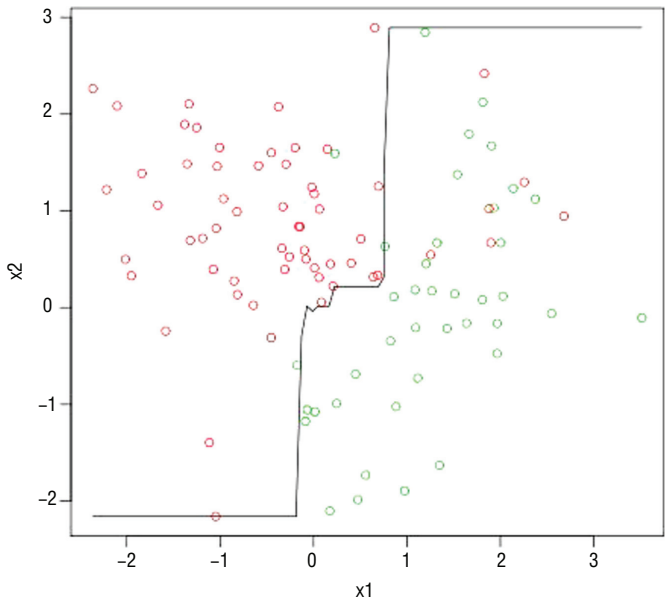

Figure 3-8: Ensemble model fit to small sample of complex data

Factors Driving Predictive Algorithm Performance

These results explain the excitement over large volumes of data. Accurate predictions for complicated problems require large volumes of data. But the size isn't quite a precise enough measure. The shape of the data also matters.

Equation 3-1 portrayed predictor data as a matrix having a number of rows (height) and a number of columns (width). The number of entries in the matrix is the product of the number of rows and the number of columns. An important difference exists between the number of rows and the number of columns when the data are being used for predictive modeling. Adding a column means adding a new attribute. Adding a new row means getting an additional historical example of the existing attributes. To understand how the effects of a new row differ from the effects of a new column, consider a linear model relating the attributes from Equation 3-1 to the labels of Equation 3-2.

Assume a model of the following form (see Equation 3-7):

$$y_i \sim x_i * \beta$$
$$= x_{i1} * \beta 1 + x_{i2} * \beta 2 + \ldots + x_i m * \beta m$$

Equation 3-7: Linear relation between attributes and outcomes

Here, x_i is a row of attributes, and β is a column vector of coefficients to be determined. Adding a column to the matrix of attributes adds another β coefficient that needs to be determined. This added coefficient is also called *degree*

of freedom. Adding another degree of freedom is making the model more complicated. The preceding examples demonstrated that making the model more complicated required more data. For this reason, it is common to think in terms of the ratio of rows to columns—the aspect ratio.

Biological data sets and natural language processing data sets are examples that are quite large because they have a lot of columns, but they are sometimes not large enough to get good performance out of a complex modeling approach. In biology, genomic data sets can easily contain 10,000 to 50,000 attributes. Even with tens of thousands of individual experiments (rows of data), a genomic data set may not be enough to train a complex ensemble model. A linear model may give equivalent or better performance.

Genomic data are expensive. One of the experiments (rows) can cost upward of $5,000, making the full data set cost upward of $50 million. Text can be relatively inexpensive to collect and store, but can also be even wider than genomic data. In some natural language processing problems, the attributes are words, and rows are documents. Entries in the matrix of attributes are the number of times a word appears in a document. The number of columns is the vocabulary size for a document collection. Depending on preprocessing (for example, removing common words like *a*, *and*, and *of*), the vocabulary can be from a few thousand to a few tens of thousands. The attribute matrix for text becomes very wide when n-grams are counted alongside words. N-grams are groups of two, three, or four words that appear next to one another (or close enough to be a phrase). When groups of two, three, or four words are also counted, the attribute space for natural language processing can grow to more than a million attributes. Once again, a linear model may give equivalent or better performance than a more complicated ensemble model.

Choosing an Algorithm: Linear or Nonlinear?

The visual examples you have just seen give some idea of the performance tradeoffs between linear and nonlinear predictive models. Linear models are preferable when the data set has more columns than rows or when the underlying problem is simple. Nonlinear models are preferable for complex problems with many more rows than columns of data. An additional factor is training time. Fast linear techniques train much faster than nonlinear techniques. (You will have more of a basis for making this decision after you've covered the techniques described in Chapter 4 and Chapter 6 and have worked through some examples.)

Choosing a nonlinear model (say an ensemble method) entails training a number of different models of differing complexity. For example, the ensemble model that generated the decision boundary in Figure 3-6 was one of roughly a thousand different models generated during the training process. These models had a variety of different complexities. Some of them would have given a much cruder approximation to the boundaries that are visually apparent in Figure 3-6. The model that generated the decision boundary in Figure 3-6 was chosen because it performed the best on out-of-sample data. This process holds for many modern machine learning

algorithms. Examples will be covered in covered in the section "Choosing a Model to Balance Problem Complexity, Model Complexity, and Data Set Size."

This section has used data sets and classifier solutions that can be visualized in order to give you an intuitive grasp of the factors affecting the performance of the predictive models you build. Generally, you'll use numeric measures of performance instead of relying on pictures. The next section describes the methods and considerations for producing numeric performance measures for predictive models and how to use these to estimate the performance your models will achieve when deployed.

Measuring the Performance of Predictive Models

This section covers two broad areas relating to performance measures for predictive models. The first one is the different metrics that you can use for different types of problems (for example, using MSE for a regression problem and misclassification error for a classification problem). In the literature (and in machine learning competitions), you will also see measures like receiver operating curves (ROC curves) and area under the curve (AUC). Besides that, these ideas are useful for optimizing performance.

The second broad area consists of techniques for gathering out-of-sample error estimates. Recall that out-of-sample errors are meant to simulate errors on new data. It's an important part of design practice to use these techniques to compare different algorithms and to select the best model complexity for a given problem complexity and data set size. That process is discussed in detail later in this chapter and is then used in examples throughout the rest of the book.

Performance Measures for Different Types of Problems

Performance measures for regression problems are relatively straightforward. In a regression problem, both the target and the prediction are real numbers. Error is naturally defined as the difference between the target and the prediction. It is useful to generate statistical summaries of the errors for comparisons and for diagnostics. The most frequently used summaries are the mean squared error (MSE) and the mean absolute error (MAE). Listing 3-1 compares the calculation of the MSE, MAE, and root MSE (RMSE, which is the square root of MSE).

Listing 3-1: Comparison of MSE, MAE and RMSE—regressionErrorMeasures.py

```
__author__ = 'mike-bowles'


#here are some made-up numbers to start with
target = [1.5, 2.1, 3.3, -4.7, -2.3, 0.75]
prediction = [0.5, 1.5, 2.1, -2.2, 0.1, -0.5]
```

```
error = []
for i in range(len(target)):
    error.append(target[i] - prediction[i])

#print the errors
print("Errors ",)
print(error)
#ans:  [1.0, 0.60000000000000009, 1.1999999999999997, -2.5,
#-2.3999999999999999, 1.25]



#calculate the squared errors and absolute value of errors
squaredError = []
absError = []
for val in error:
    squaredError.append(val*val)
    absError.append(abs(val))


#print squared errors and absolute value of errors
print("Squared Error")
print(squaredError)
#ans: [1.0, 0.3600000000000001, 1.4399999999999993, 6.25,
#5.7599999999999998, 1.5625]
print("Absolute Value of Error")
print(absError)
#ans: [1.0, 0.60000000000000009, 1.1999999999999997, 2.5,
#2.3999999999999999, 1.25]


#calculate and print mean squared error MSE
print("MSE = ", sum(squaredError)/len(squaredError))
#ans: 2.72875


from math import sqrt
#calculate and print square root of MSE (RMSE)
print("RMSE = ", sqrt(sum(squaredError)/len(squaredError)))
#ans: 1.65189285367


#calculate and print mean absolute error MAE
print("MAE = ", sum(absError)/len(absError))
#ans: 1.49166666667


#compare MSE to target variance
targetDeviation = []
targetMean = sum(target)/len(target)
for val in target:
    targetDeviation.append((val - targetMean)*(val - targetMean))
```

continues

continued

```
#print the target variance
print("Target Variance = ", sum(targetDeviation)/len(targetDeviation))
#ans: 7.5703472222222219

#print the the target standard deviation (square root of variance)
print("Target Standard Deviation = ", sqrt(sum(targetDeviation)
   /len(targetDeviation)))
#ans: 2.7514263977475797
```

The example starts with some made-up numbers for the targets and the predictions. First, it calculates the errors by simple subtraction; then it shows the calculation of MSE, MAE, and RMSE. Notice that MSE comes out markedly different in magnitude than MAE and RMSE. That's because MSE is in squared units. For that reason, the RMSE is usually a more usable number to calculate. At the bottom of the listing is a calculation of the variance (mean squared deviation from the mean) and the standard deviation (square root of variance) of the targets. These quantities are useful to compare (respectively) to the MSE and RMSE of the prediction errors. For example, if the MSE of the prediction error is roughly the same as the target variance (or the RMSE is roughly the same as target standard deviation), the prediction algorithm is not performing well. You could replace the prediction algorithm with a simple calculation of the mean of the targets and perform as well. The errors in Listing 3-1 have RMSE that's about half the standard deviation of the targets. That is fairly good performance.

Besides calculating summary statistics for the error, it may sometimes be useful for analyzing sources and magnitudes of error to look at things like histogram of the error or tail behavior (quantile or decile boundaries), degree of normality, and so forth. Sometimes those investigations will yield insights into error sources and potential performance improvements.

Classification problems require different treatment. The approaches to classification problems generally revolve around misclassification error rates—the fraction of examples that are incorrectly classified. Suppose, for instance, that the classification problem is to predict click or not-click on a link being considered for presentation to a site visitor. Generally, algorithms for doing classification can present predictions in the form of a probability instead of a hard click versus not-click decision. The algorithms considered in this book all output probabilities.

Here's why that's useful. If the prediction of click or not-click is given as a probability—say 80 percent chance of click (and correspondingly 20 percent chance of not-click)—the data scientist has the option to use 50 percent as a threshold for presenting the link or not presenting the link. In some cases, however, a higher or lower threshold value will give a better end result.

Suppose, for example, that the problem is fraud detection (for credit cards, automatic clearinghouses [checking], insurance claims, and so on). The actions that proceed from making a fraud-or-not decision are to have a call center representative intervene in the transaction or to let it go. There are costs involved

with either decision. If the call is made, there's the call center cost and the cost of the customer's reaction. If the call isn't made, there's the cost of the potential fraud. If the costs of taking the action are very low relative to the costs of not taking the action, the minimum total comes at a relatively low threshold. More transactions get flagged for intervention.

But where do you draw the line for interrupting your customer's checkout and requiring the customer to call card services to proceed? Do you interrupt the transactions where your predictive algorithm indicates a 20 percent, 50 percent, or 80 percent probability that the transaction is fraudulent? If you place the threshold for interruption at 20 percent, you'll be intervening more frequently—preventing more fraudulent transactions—but also irritating more customers and keeping many call center reps busy. Maybe it is better to place the threshold higher (say 80 percent) and to accept more fraud.

A useful way to think about this is to arrange the possible outcomes into what is called a *confusion matrix* or *contingency table* (`http://en.wikipedia.org/wiki/Confusion_matrix`). Figure 3-9 shows a toy example of a contingency matrix. The numbers in the contingency table represent the performance based on a choice for the threshold value discussed in the last paragraph. The contingency matrix in Figure 3-9 summarizes the results of making predictions for 135 test examples for a particular choice of the threshold probability. The matrix has two columns representing the possible predictions. It also has two rows representing the truth (label) for each example. So, each example in the test set can be assigned to one of the four cells in the table. The two classifications portrayed in Figure 3-9 are "click" and "not click," appropriate for selecting an ad. These could also correspond to "fraud" and "not fraud"—or other pairs—depending on the specific problem being addressed.

The upper-left cell contains examples that are predicted as click and where that matches the label (truth). These are called *true positive* and are generally abbreviated as TP. The entries in the lower-left box correspond to examples where the prediction was positive (click) but the truth was negative (not-click).

	Predicted Class		
Actual Class		Positive (Click)	Negative (Not Click)
	Positive (Click)	True Positive 10	False Negative 7
	Negative (Not-click)	False Positive 22	True Negative 96

Figure 3-9: Confusion matrix example

These are called *false positive* and abbreviated as FP. The right column of the matrix contains the examples that were predicted not-click. The examples in the upper right were click in truth and are called *false negatives* or FN. The lower-right examples were predicted not-click and agree with the real outcome. They are called *false negative* or FN.

What happens when the probability threshold is changed? Consider the extreme values. If the probability threshold is set to 0.0, no matter what probability your model predicts, it will get designated as a click. All the examples wind up in the left column. There are only 0s in the right column. The number of TPs would go up to 17. The number of FPs would go up to 118. If there were no cost for an FP and no reward for a true negative (TN), that might be a good choice, but no predictive algorithm is required to assume click for every input example. Similarly, if there is no cost for an FN and no benefit for a TP, the threshold can be set at 1.0 so that all examples are classified as not-click. These extremes aid understanding, but they're not useful in a deployed system. The following example shows how the process would work to build a classifier for the rocks-versus-mines data set.

The rocks-versus-mines data set presents the problem of building a classifier that uses sonar data to determine whether seabed objects are rocks or mines. (For a more thorough discussion and exploration of the data set, see Chapter 2, "Understand the Problem by Understanding the Data.") Listing 3-2 shows the Python code for training a simple classifier on the rocks-versus-mines data set and then predicts performance for the classifier.

Listing 3-2: Measuring Performance for Classifier Trained on Rocks-Versus-Mines—classifierPerformance_RocksVMines.py

```
__author__ = 'mike-bowles'
#use scikit learn package to build classified on rocks-versus-mines data
#assess classifier performance

import urllib2
import numpy
import random
from sklearn import datasets, linear_model
from sklearn.metrics import roc_curve, auc
import pylab as pl


def confusionMatrix(predicted, actual, threshold):
    if len(predicted) != len(actual): return -1
    tp = 0.0
    fp = 0.0
    tn = 0.0
    fn = 0.0
    for i in range(len(actual)):
        if actual[i] > 0.5: #labels that are 1.0  (positive examples)
```

```
            if predicted[i] > threshold:
                tp += 1.0 #correctly predicted positive
            else:
                fn += 1.0 #incorrectly predicted negative
        else:              #labels that are 0.0 (negative examples)
            if predicted[i] < threshold:
                tn += 1.0 #correctly predicted negative
            else:
                fp += 1.0 #incorrectly predicted positive
    rtn = [tp, fn, fp, tn]
    return rtn


#read in the rocks versus mines data set from uci.edu data repository
target_url = ("https://archive.ics.uci.edu/ml/machine-learning-"
"databases/undocumented/connectionist-bench/sonar/sonar.all-data")
data = urllib2.urlopen(target_url)

#arrange data into list for labels and list of lists for attributes
xList = []
labels = []
for line in data:
    #split on comma
    row = line.strip().split(",")
    #assign label 1.0 for "M" and 0.0 for "R"
    if(row[-1] == 'M'):
        labels.append(1.0)
    else:
        labels.append(0.0)
    #remove lable from row
    row.pop()
    #convert row to floats
    floatRow = [float(num) for num in row]
    xList.append(floatRow)

#divide attribute matrix and label vector into training(2/3 of data)
#and test sets (1/3 of data)
indices = range(len(xList))
xListTest = [xList[i] for i in indices if i%3 == 0 ]
xListTrain = [xList[i] for i in indices if i%3 != 0 ]
labelsTest = [labels[i] for i in indices if i%3 == 0]
labelsTrain = [labels[i] for i in indices if i%3 != 0]

#form list of list input into numpy arrays to match input class
#for scikit-learn linear model
xTrain = numpy.array(xListTrain); yTrain = numpy.array(labelsTrain)
xTest = numpy.array(xListTest); yTest = numpy.array(labelsTest)

#check shapes to see what they look like
print("Shape of xTrain array", xTrain.shape)
print("Shape of yTrain array", yTrain.shape)
```

continues

continued

```
print("Shape of xTest array", xTest.shape)
print("Shape of yTest array", yTest.shape)

#train linear regression model
rocksVMinesModel = linear_model.LinearRegression()
rocksVMinesModel.fit(xTrain,yTrain)

#generate predictions on in-sample error
trainingPredictions = rocksVMinesModel.predict(xTrain)
print("Some values predicted by model", trainingPredictions[0:5],
 trainingPredictions[-6:-1])

#generate confusion matrix for predictions on training set (in-sample
confusionMatTrain = confusionMatrix(trainingPredictions, yTrain, 0.5)
#pick threshold value and generate confusion matrix entries
tp = confusionMatTrain[0]; fn = confusionMatTrain[1]
fp = confusionMatTrain[2]; tn = confusionMatTrain[3]

print("tp = " + str(tp) + "\tfn = " + str(fn) + "\n" + "fp = " +
str(fp) + "\ttn = " + str(tn) + '\n')

#generate predictions on out-of-sample data
testPredictions = rocksVMinesModel.predict(xTest)

#generate confusion matrix from predictions on out-of-sample data
conMatTest = confusionMatrix(testPredictions, yTest, 0.5)
#pick threshold value and generate confusion matrix entries
tp = conMatTest[0]; fn = conMatTest[1]
fp = conMatTest[2]; tn = conMatTest[3]
print("tp = " + str(tp) + "\tfn = " + str(fn) + "\n" + "fp = " +
str(fp) + "\ttn = " + str(tn) + '\n')

#generate ROC curve for in-sample

fpr, tpr, thresholds = roc_curve(yTrain,trainingPredictions)
roc_auc = auc(fpr, tpr)
print( 'AUC for in-sample ROC curve: %f' % roc_auc)

# Plot ROC curve
pl.clf()
pl.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
pl.plot([0, 1], [0, 1], 'k-')
pl.xlim([0.0, 1.0])
pl.ylim([0.0, 1.0])
pl.xlabel('False Positive Rate')
pl.ylabel('True Positive Rate')
pl.title('In sample ROC rocks versus mines')
pl.legend(loc="lower right")
pl.show()

#generate ROC curve for out-of-sample
fpr, tpr, thresholds = roc_curve(yTest,testPredictions)
```

```
roc_auc = auc(fpr, tpr)
print( 'AUC for out-of-sample ROC curve: %f' % roc_auc)

# Plot ROC curve
pl.clf()
pl.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
pl.plot([0, 1], [0, 1], 'k-')
pl.xlim([0.0, 1.0])
pl.ylim([0.0, 1.0])
pl.xlabel('False Positive Rate')
pl.ylabel('True Positive Rate')
pl.title('Out-of-sample ROC rocks versus mines')
pl.legend(loc="lower right")
pl.show()
```

The first section of the code reads the input data from the University of California Irvine data repository and then formats it as a list for the labels and a list of lists for the attributes. The next step is to break the data (labels and attributes) into two subsets: a test set that contains one third of the data, and a training set that contains the other two thirds. The data labeled *test* will not be used in training the classifier, but will be reserved for assessing performance after the classifier is trained. This step simulates the behavior of the classifier on new data examples after it has been deployed. Later, this chapter discusses a variety of different methods for holding out data and making estimates of performance on new data.

The classifier is trained by converting the labels M (for mine) and R (for rock) in the original data set into numeric values—1.0 corresponding to mine, and 0.0 corresponding to rock—and then using the ordinary least squares regression to fit a linear model. This is a fairly simple method to understand and to implement and will often generate very similar performance to the more sophisticated algorithms discussed later. The program in Listing 3-2 employs the linear regression class from scikit-learn to train the ordinary least squares model. Then the trained model is used to generate predictions on the training set and on the test set.

The code prints out some representative values for the predictions. The linear regression model generates numbers that are mostly in the interval between 0.0 and 1.0, but not entirely. The predictions aren't quite probabilities. They can still be used to generate predicted classifications by comparing to a threshold value. The function `confusionMatrix()` produces the values for a confusion matrix, similar to Figure 3-9. It takes the predictions, the corresponding actual values (labels), and a threshold value as input. It compares the predictions to the threshold to determine whether to assign each example to the "predicted positive" or "predicted negative" column in the confusion matrix. It uses the actual value to make the assignment to the appropriate row of the confusion matrix.

The error rates for each threshold value can be read out of the confusion matrix. The total number of errors is the sum of FPs and FNs. The example code produces confusion matrices for the in-sample data and the out-of-sample data and prints them both out. The misclassification error rate on the in-sample data is about 8 percent, and about 26 percent on the out-of-sample data. Generally, the out-of-sample performance will be worse than performance on in-sample data. It will also be more representative of the expected error on new examples.

The misclassification error changes when the thresholds are changed. Table 3-2 shows how misclassification error rate changes as the threshold value changes. The numbers in the table are based on out-of-sample results. That will be generally true of numbers characterizing performance throughout the book. Any in-sample errors will have warning labels attached: "Warning: These are in-sample errors." If the goal is to minimize the misclassification error, the best threshold value is 0.25.

Table 3-2: *Dependence of Misclassification Error on Decision Threshold*

DECISION THRESHOLD	MISCLASSIFICATION ERROR RATE
0.0	28.6 percent
0.25	24.3 percent
0.5	25.7 percent
0.75	30.0 percent
1.0	38.6 percent

The best value for the threshold may be the one that minimizes the misclassification error. Sometimes, however, there's more cost associated with one type of error than with another. Suppose, for instance, that for the rocks-versus-mines problem it costs $100 to send a diver to do a visual inspection and that unexploded mines cost $1,000 in expected injuries and property damage if not removed. An FP costs $100, and an FN costs $1,000. Given these assumptions, Table 3-3 summarizes the dollar cost of mistakes for different threshold values. The higher cost of mistaking a mine for a rock (and leaving it in place to threaten health and safety) has pushed the decision threshold down to zero. That means more FNs, but they aren't as expensive. A more thorough analysis could include the costs associated with TP and TN. For example, the TP might have costs associated with removing the mine and a benefit of +$1,000 associated with its removal. If these figures are available (or can be reasonably approximated) in your problem, it behooves you to use them to derive better threshold values.

Table 3-3: *Cost of Mistakes for Different Decision Thresholds*

DECISION THRESHOLD	FALSE NEGATIVE COST	FALSE POSITIVE COST	TOTAL COST
0.0	1,000	1,900	2,900
0.25	3,000	1,400	4,400
0.5	9,000	900	9,900
0.75	18,000	300	18,300
1.00	26,000	100	26,100

Note that the relative cost of total FPs versus FNs depends on the proportion of positive and negative examples in the data set. The rocks-versus-mines data set has an equal number of positives and negatives (mines and rocks). That was presumably determined by an experimental protocol. The proportion of positives and negatives encountered in actual practice may differ. If the numbers are likely to be different when the system is deployed, you need to make some adjustments to account for the proportions in actual use.

The data scientist may not have the costs available but may still want a method to characterize the overall performance of the classifier instead of using the misclassification error rate for a particular decision threshold. A common technique for doing that is called the *receiver operating characteristic* or ROC curve (`http://en.wikipedia.org/wiki/Receiver_operating_characteristic`).

ROC inherits its name from its original application—processing returns from a radar receiver to determine the presence or absence of hostile aircraft. The ROC curve yields a single plot that summarizes all of these different contingency tables. The ROC curve plots the true positive rate (abbreviated TPR) versus the false positive rate (FPR). TPR is the proportion of positive examples that are correctly classified as positive (see Equation 3-8). FPR is the number of FPs relative to the total number of actual negatives (see Equation 3-9). In terms of the elements of the contingency table, these are given by the following formulas:

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

Equation 3-8: True positive rate

$$FPR = \frac{FP}{TN + FP}$$

Equation 3-9: False positive rate

As a simple thought experiment, consider using an extremely low value for the decision threshold. For a low value, every example is predicted as positive.

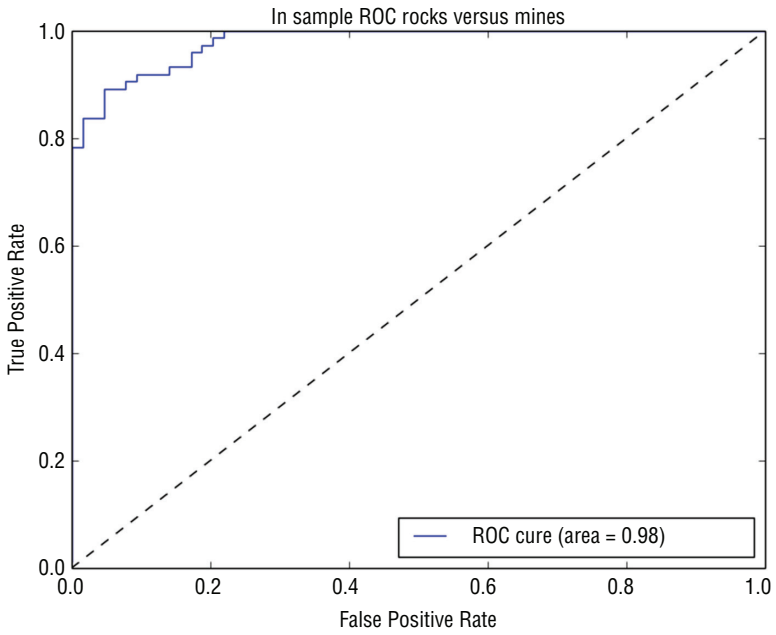

Figure 3-10: In-sample ROC for rocks-versus-mines classifier

That gives 1.0 for TPR. Because everything is classified as positive, there are no FNs (FN is 0.0). It also gives 1.0 for FPR because nothing gets classified as negative (TN is 0.0). However, when the decision threshold is set very high, TP is equal to zero, and so TPR is also zero and FP is also zero because nothing gets classified as positive. Therefore, FPR is also zero. The following two figures were drawn using the `pylab roc_curve()` and `auc()` functions. Figure 3-10 shows the ROC curve-based performance on in-sample data. Figure 3-11 shows the ROC curve based on out-of-sample data.

The ROC curve for the classifier that operates by randomly deciding rock or mine forms a diagonal line from the lower-left corner to the upper-right corner of the plot. That line is often drawn onto ROC curves as a reference point. For a perfect classifier, the ROC curve steps straight up from (0, 0) to (0, 1) and then goes straight across to (1, 1). Not surprisingly, Figure 3-10 (on in-sample data) comes closer to perfection than Figure 3-11 (on out-of-sample data). The closer that a classifier can come to hitting the upper-left corner, the better it is. If the ROC curve drops significantly below the diagonal line, it usually means that the data scientist has gotten a sign switched somewhere and should examine his code carefully.

Figures 3-10 and 3-11 also show the area under the curve (AUC) numbers. AUC, as the name suggests, is the area under the ROC curve. A perfect classifier has an AUC of 1.0, and random guessing has an AUC of 0.5. AUCs for Figures 3-10 and 3-11 provide another demonstration that performance estimates based

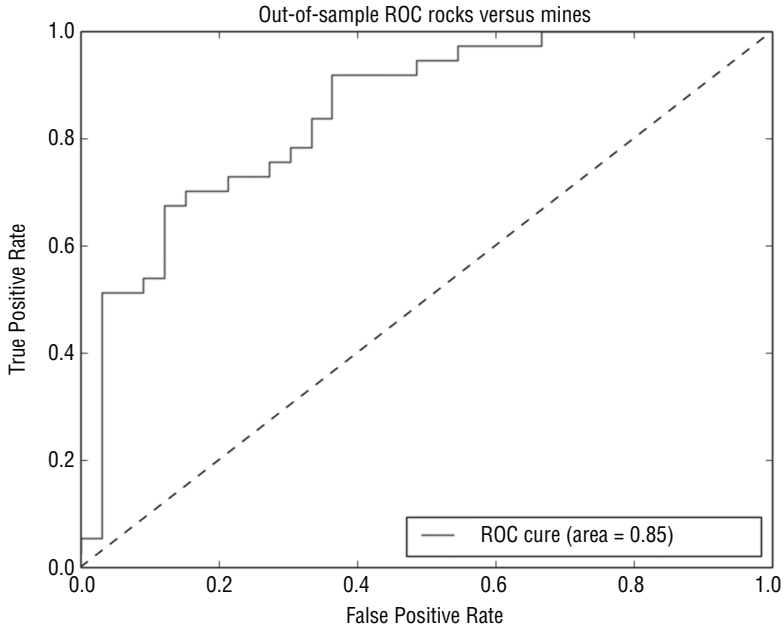

Figure 3-11: Out-of-sample ROC for rocks-versus-mines classifier

on the error on the training set (in-sample data) overestimate performance. The AUC on in-sample data is 0.98. The AUC on out-of-sample data is 0.85.

Some of the methods used for measuring binary classifier performance will also work for multiclass classifiers. Misclassification error still makes sense, and the confusion matrix also works. There are also multiclass generalizations of the ROC curve and AUC.[1]

Simulating Performance of Deployed Models

The examples from the preceding section demonstrated the need for testing performance on data not included in the training set to get a useful estimate of expected performance once a predictive model is deployed. The example broke the available labeled data into two subsets. One subset, called the *training set*, contained approximately two-thirds of the available data and was used to fitting an ordinary least squares model. The second subset, which contained the remaining third of the available data, was called the *test set* and was used only for determining performance (not used during training of the model). This is a standard procedure in machine learning.

Test set sizes range from 25 percent to 35 percent of the data, although there aren't any hard-and-fast rules about the sizes. One thing to keep in mind is that the performance of the trained model deteriorates as the size of the training data set shrinks. Taking out too much data from the training set can prove detrimental to end performance.

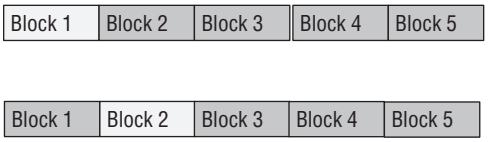

Figure 3-12: N-fold cross-validation

Another approach to holding out data is called *n-fold cross-validation*. Figure 3-12 shows schematically how a data set is divided up for training and testing with n-fold cross-validation. The set is divided into *n* disjointed sets of roughly equal sizes. In the figure, *n* is 5. Several training and testing passes are made through the data. In the first pass, the first block of data is held out for testing, and the remaining *n*-1 are used for training. In the second pass, the second block is held out for testing, and the other *n*-1 are used for training. This process is continued until all the data have been held out (five times for the five-fold example depicted in Figure 3-12).

The n-fold cross-validation process yields an estimate of the prediction error and has several samples of the error so that it can estimate error bounds on the error. It can keep more of the data in the training set, which generally gives lower generalization errors and better final performance. For example, if the 10-fold cross-validation is chosen, then only 10 percent of the data is held out for each training pass. These features of n-fold cross-validation come at the expense of taking more training time. The approach of taking a fixed holdout set has the advantage of faster training, because it employs only one pass through the training data. Taking a fixed holdout set is probably a better choice when the training times are unbearable with n-fold cross-validation and when there's so much training data available that some extra in the holdout set won't adversely affect performance.

Another thing to keep in mind is that the sample should be representative of the whole data set. The sampling plan used in the example in the last section was not a random sample. It was a sample of every third data point. Spreading the samples uniformly through the data usually works fine. However, you do need to avoid sampling in a way that introduces a bias in training and test sets. For example, if you were given data that was sampled once per day and arranged in order of sampling date, then coding seven-fold cross-validation and sampling every seventh point should be avoided.

Sampling may need to be carefully controlled if the phenomenon being studied has unusual statistics. Care may have to be taken to preserve the statistical peculiarities in the test sample. Examples of this include predicting rare events like fraud or ad clicks. The events being modeled are so infrequent that random sampling may over- or under-represent them in the test set and lead to erroneous estimates of performance. Stratified sampling (`http://en.wikipedia.org/wiki/Stratified_sampling`) divides the data into separate subsets that are

separately sampled and then recombined. When the labels are rare events, you might need to separately sample the fraudulent examples and the legitimate examples and then combine them for the test set to match the training set and, more importantly, the new data upon which the model will be used.

After a model has been trained and tested, it is good practice to recombine the training and test data into a single set and retrain the model on the larger data set. The out-of-sample testing procedure will have already given good estimates of the expected prediction errors. That was the purpose of holding out some of the data. The model will perform better and generalize better if trained on more data. The deployed model should be trained on all the data.

This section supplied you with tools to quantify the performance of your predictive model. The next section shows you how to replace the intuitive graphical comparisons of model and problem complexity that you saw in the section "Factors Driving Algorithm Choices and Performance" with numerical comparison. This replacement makes it possible to mechanize some of the selection process.

Achieving Harmony Between Model and Data

This section uses ordinary least squares (OLS) regression to illustrate several things. First, it illustrates how OLS can sometimes *overfit* a problem. Overfitting means that there's a significant discrepancy between errors on the training data and errors on the test data, such as you saw in the previous section where OLS was used to solve the rocks-versus-mines classification problem. Second, it introduces two methods for overcoming the overfit problem with OLS. These methods will cultivate your intuition and set the stage for the penalized linear regression methods that are covered in more depth in Chapter 4. In addition, the methods for overcoming overfitting have a property that is common to most modern machine learning algorithms. Modern algorithms generate a number of models of varying complexity and then use out-of-sample performance to balance model complexity, problem complexity, and data set richness and thus determine which model to deploy. This process will be used repeatedly throughout the rest of the book.

Ordinary least squares regression serves as a good prototype for machine learning algorithms in general. It's a supervised algorithm that has a training procedure and a deployment procedure. It can be overfit in some circumstances. It shares these features with other more modern function approximation algorithms. OLS is missing an important feature of modern algorithms, however. In its original formulation (the most familiar formulation), there's no means to throttle it back when it overfits. It's like having a car that only runs at full throttle (great when there's plenty of road, but tough to use in tight circumstances). Fortunately, there's been a lot of work on ordinary least squares regression

since its invention more than 200 years ago by Gauss and Legendre. This section introduces two of the methods for adjusting the throttle on ordinary least squares regression. One is called *forward stepwise regression*; the other is called *ridge regression*.

Choosing a Model to Balance Problem Complexity, Model Complexity, and Data Set Size

A couple of examples will illustrate how modern machine learning techniques can be tuned to best fit a given problem and data set. The first example is a modification to ordinary least squares regression called forward stepwise regression. Here's how it works. Recall Equations 3-1 and 3-2, which define the problem being solved (see Equations 3-10 and 3-11 here, which repeat those equations). The vector Y contains the labels. And the matrix X contains the attributes available to predict the labels.

$$Y = \begin{matrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{matrix}$$

Equation 3-10: Vector of numeric labels

$$X = \begin{matrix} x_{11} & x_{12} & & x_{1n} \\ x_{21} & x_{22} & & x_{2n} \\ \vdots & \vdots & \ddots & \\ x_{m1} & x_{m2} & & x_{mn} \end{matrix}$$

Equation 3-11: Matrix of numeric attributes

If this is a regression problem, then Y is a column vector of real numbers, and the linear problem is to find a column vector of weights β and a scalar $\beta 0$ (see Equation 3-12).

$$\beta = \begin{matrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{matrix}$$

Equation 3-12: Vector of coefficients for linear model

The values for β are selected so that Y is well approximated (see Equation 3-13).

$$Y \sim X\beta + \begin{matrix} \beta_0 \\ \beta_0 \\ \vdots \\ \beta_0 \end{matrix}$$

Equation 3-13: Approximating labels as linear function of attributes

If the number of columns of X is the same as the number of rows of X and the columns of X are independent (not linear multiples of one another), then X can be inverted and the ~ can be replaced with =. A coefficient vector β will make the linear fit the labels exactly. That's too good to be true. The problem is one of overfitting (that is, getting terrific performance on the training data that cannot be replicated on new data). In real problems, this is not a good outcome. The source of overfitting is having too many columns of data in X. The answer might be to get rid of some of the columns of X. However, getting rid of some involves deciding how many to eliminate and which ones should be eliminated. The brute-force method is called *best subset selection*.

Using Forward Stepwise Regression to Control Overfitting

The following code provides an outline of the algorithm for best subset selection. The basic idea is to impose a constraint (say nCol) on the number of columns and then take all subsets of the columns of X that have that number of columns, perform ordinary least squares regression, identify the nCol subset that has the least out-of-sample error, increment nCol, and repeat. The process results in a list of the best choice of one-column subsets: two-column subsets up to the full matrix X (the all-column subset). It also yields the performance of each of these. Then the next step is to determine whether to deploy the one-column version, the two-column version, and so on. But that's relatively easy; just pick the one with the least errors.

```
Initialize: Out_of_sample_error = NULL
   Break X and Y into test and training sets
for i in range(number of columns in X):
   for each subset of X having i+1 columns:
      fit ordinary least squares model
   Out_of_sample_error.append(least error amoung subsets containing
   i+1 columns)
Pick the subset corresponding to least overall error
```

The problem with best subset selection is that it requires too much calculation for even modest numbers of attributes (columns of X). For example, 10 attributes leads to $2^{10} = 1,000$ subsets. There are several techniques that avoid this. The following code shows the procedure for forward stepwise regression. The idea with forward stepwise regression is to start with one-column subsets and then,

given the best single column, to find the best second column to append instead of evaluating all possible two-column subsets. Pseudo-code for forward stepwise regression is given here.

```
Initialize: ColumnList = NULL
   Out-of-sample-error = NULL
   Break X and Y into test and training sets
For number of column in X:
   For each trialColumn (column not in ColumnList):
      Build submatrix of X using ColumnList + trialColumn
      Train OLS on submatrix and store RSS Error on test data
   ColumnList.append(trialColumn that minimizes RSS Error)
   Out-of-sample-error.append(minimum RSS Error)
```

Best subset selection and forward stepwise regression have similar processes. They train a series of models (several for one column, several for two columns, and so on). They result in a parameterized family of models (all linear regression parameterized on number of columns). The models vary in complexity, and the final model is selected from the family on the basis of performance on out-of-sample error.

Listing 3-3 shows Python code implementing forward stepwise regression on the wine data set.

Listing 3-3: Forward Stepwise Regression: Wine Quality Data—fwdStepwiseWine.py

```
import numpy
from sklearn import datasets, linear_model
from math import sqrt
import matplotlib.pyplot as plt

def xattrSelect(x, idxSet):
    #takes X matrix as list of list and returns subset containing
    #columns in idxSet
    xOut = []
    for row in x:
        xOut.append([row[i] for i in idxSet])
    return(xOut)

#read data into iterable
target_url = ("http://archive.ics.uci.edu/ml/machine-learning-
databases/"
"wine-quality/winequality-red.csv")
data = urllib2.urlopen(target_url)
xList = []
labels = []
names = []
firstLine = True
for line in data:
```

```
    if firstLine:
        names = line.strip().split(";")
        firstLine = False
    else:
        #split on semi-colon
        row = line.strip().split(";")
        #put labels in separate array
        labels.append(float(row[-1]))
        #remove label from row
        row.pop()
        #convert row to floats
        floatRow = [float(num) for num in row]
        xList.append(floatRow)

#divide attributes and labels into training and test sets
indices = range(len(xList))
xListTest = [xList[i] for i in indices if i%3 == 0 ]
xListTrain = [xList[i] for i in indices if i%3 != 0 ]
labelsTest = [labels[i] for i in indices if i%3 == 0]
labelsTrain = [labels[i] for i in indices if i%3 != 0]

#build list of attributes one-at-a-time - starting with empty
attributeList = []
index = range(len(xList[1]))
indexSet = set(index)
indexSeq = []
oosError = []

for i in index:
    attSet = set(attributeList)
    #attributes not in list already
    attTrySet = indexSet - attSet
    #form into list
    attTry = [ii for ii in attTrySet]
    errorList = []
    attTemp = []
    #try each attribute not in set to see which
    #one gives least oos error
    for iTry in attTry:
        attTemp = [] + attributeList
        attTemp.append(iTry)
        #use attTemp to form training and testing sub matrices
        #as list of lists
        xTrainTemp = xattrSelect(xListTrain, attTemp)
        xTestTemp = xattrSelect(xListTest, attTemp)
        #form into numpy arrays
        xTrain = numpy.array(xTrainTemp)
        yTrain = numpy.array(labelsTrain)
        xTest = numpy.array(xTestTemp)
        yTest = numpy.array(labelsTest)
```

continues

continued

```
        #use sci-kit learn linear regression
        wineQModel = linear_model.LinearRegression()
        wineQModel.fit(xTrain,yTrain)
        #use trained model to generate prediction and calculate rmsError
        rmsError = numpy.linalg.norm((yTest-wineQModel.predict(xTest)),
            2)/sqrt(len(yTest))
        errorList.append(rmsError)
        attTemp = []

    iBest = numpy.argmin(errorList)
    attributeList.append(attTry[iBest])
    oosError.append(errorList[iBest])

print("Out of sample error versus attribute set size" )
print(oosError)
print("\n" + "Best attribute indices")
print(attributeList)
namesList = [names[i] for i in attributeList]
print("\n" + "Best attribute names")
print(namesList)

#Plot error versus number of attributes
x = range(len(oosError))
plt.plot(x, oosError, 'k')
plt.xlabel('Number of Attributes')
plt.ylabel('Error (RMS)')
plt.show()

#Plot histogram of out of sample errors for best number of attributes
#Identify index corresponding to min value,
#retrain with the corresponding attributes
#Use resulting model to predict against out of sample data.
#Plot errors (aka residuals)
indexBest = oosError.index(min(oosError))
attributesBest = attributeList[1:(indexBest+1)]

#Define column-wise subsets of xListTrain and xListTest
#and convert to numpy
xTrainTemp = xattrSelect(xListTrain, attributesBest)
xTestTemp = xattrSelect(xListTest, attributesBest)
xTrain = numpy.array(xTrainTemp); xTest = numpy.array(xTestTemp)

#train and plot error histogram
wineQModel = linear_model.LinearRegression()
wineQModel.fit(xTrain,yTrain)
errorVector = yTest-wineQModel.predict(xTest)
plt.hist(errorVector)
plt.xlabel("Bin Boundaries")
plt.ylabel("Counts")
plt.show()
```

```
#scatter plot of actual versus predicted
plt.scatter(wineQModel.predict(xTest), yTest, s=100, alpha=0.10)
plt.xlabel('Predicted Taste Score')
plt.ylabel('Actual Taste Score')
plt.show()
```

The preceding listing includes a small function to extract selected columns from the X matrix (in the form of a list of lists). Then it breaks the X matrix and the vector of labels into training and test sets. After that, the code follows the preceding algorithm description. A pass through the algorithm begins with a subset of attributes that are included in the solution. For the first pass, this subset is empty. For subsequent passes, the subset includes the attributes selected one at a time during earlier passes. Each pass selects a single new attribute to add to the subset of attributes. The attribute to be added is chosen by testing each non-included attribute to see which one results in the best performance when added to the subset. In turn, each attribute is added to the attribute subset and ordinary least squares is used to fit a linear model with the resulting attribute subset. For each attribute tested, the out-of-sample performance is measured. The tested attribute which yields the best root sum of squares (RSS) error is added to the attribute set, and the associated RSS error is captured.

Figure 3-13 plots the RMSEs as a function of the number of attributes included in the regression. The error decreases until nine attributes are included and then increases somewhat.

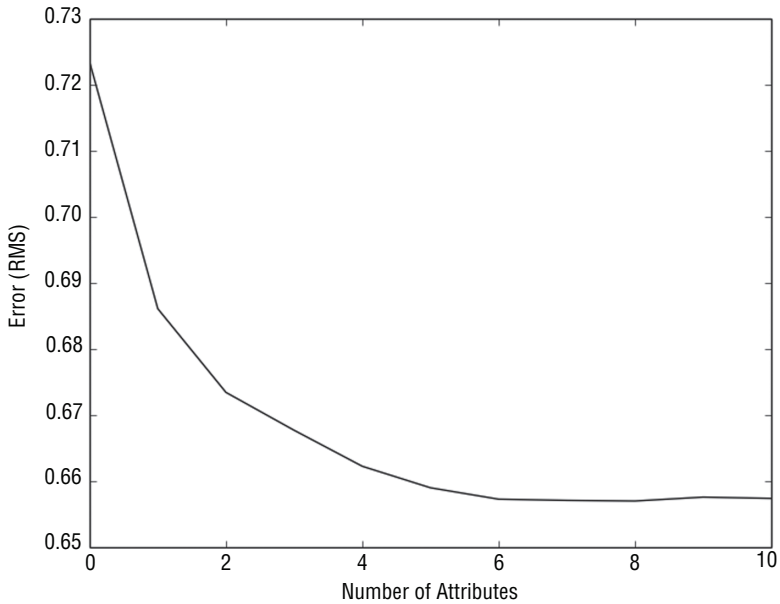

Figure 3-13: Wine quality prediction error using forward stepwise regression

Listing 3-4 shows numeric output for forward stepwise regression applied to wine quality data.

Listing 3-4: Forward Stepwise Regression Output—fwdStepwiseWineOutput.txt

```
Out of sample error versus attribute set size
[0.7234259255116281, 0.68609931528371915, 0.67343650334202809,
0.66770332138977984, 0.66225585685222743, 0.65900047541546247,
0.65727172061430772, 0.65709058062076986, 0.65699930964461406,
0.65758189400434675, 0.65739098690113373]

Best attribute indices
[10, 1, 9, 4, 6, 8, 5, 3, 2, 7, 0]

Best attribute names
['"alcohol"', '"volatile acidity"', '"sulphates"', '"chlorides"',
 '"total sulfur dioxide"', '"pH"', '"free sulfur dioxide"',
 '"residual sugar"', '"citric acid"', '"density"', '"fixed acidity"']
```

The first list shows the RSS error. The error decreases until the 10th element in the list, and then gets larger again. The associated column indices are shown in the next list. The last list gives the names (column headers) of the associated attributes.

Evaluating and Understanding Your Predictive Model

Several other plots are helpful in understanding the performance of a trained algorithm and can point the way to making improvements in its performance. Figure 3-14 shows a scatter plot of the true labels plotted versus the predicted labels for points in the test set. Ideally, all of the points in Figure 3-1 would lie on a 45-degree line—the line where the true labels and the predicted labels are equal. Because the real scores are integers, the scatter plot shows horizontal rows of points. When the true values take on a small number of values, it is useful to make the data points partially transparent so that the darkness can indicate the accumulation of many points in one area of the graph. Actual taste scores of 5 and 6 are reproduced fairly well. The more extreme values are not as well predicted by the system. Generally speaking, machine learning algorithms do worse at the edges of a data set.

Figure 3-15 shows a histogram of the prediction error for forward stepwise prediction predicting wine taste scores. Sometimes the error histogram will have two or more discrete peaks. Perhaps it will have a small peak on the far right or far left of the graph. In that case, it may be possible to find an explanation for the different peaks in the error and to reduce the prediction error by adding a new attribute that explains the membership in one or the other of the groups of points.

You want to note several things about this output. First, let's reiterate the process. The process is to train a family of models (in this case, ordinary linear

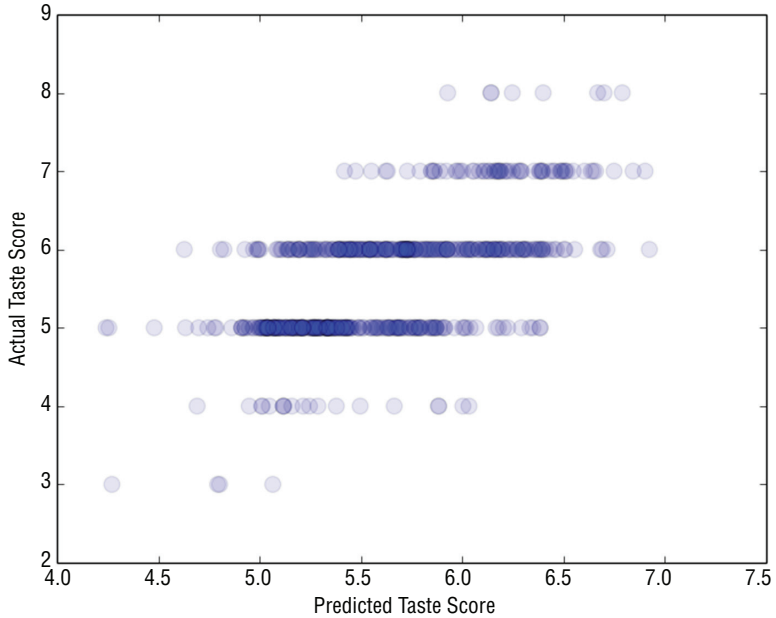

Figure 3-14: Actual taste scores versus predictions generated with forward stepwise regression

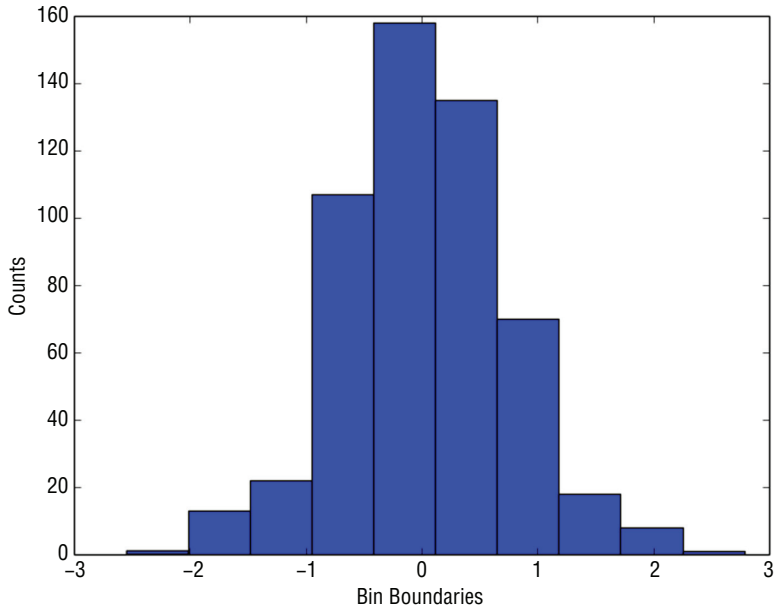

Figure 3-15: Histogram of wine taste prediction error with forward stepwise regression

regression trained on column-wise subsets of X). The series of models is parameterized (in this case, by the number of attributes that are used in the linear model). The model to deploy is chosen to minimize the out-of-sample error. The number of attributes to be incorporated in the solution can be called a *complexity parameter*. Models with larger complexity parameters have more free parameters and are more likely to overfit the data than less-complex models.

Also note that the attributes have become ordered by their importance in predicting quality. In the list of column numbers and the associated list of attribute names, the first in the list is the first attribute chosen, the second was next, and so on. The attributes used come out in a nice ordered list. This is an important and desirable feature of a machine learning technique. Early stages of a machine learning task mostly involve hunting for (or constructing) the best set of attributes for making predictions. Having techniques to rank attributes in order of importance helps in that process. The other algorithms developed in this book will also have this property.

The last observation regards picking a model from the family that machine learning techniques generate. The more complicated the model, the less well it will generalize. It is better to err on the side of a less-complicated model. The earlier example indicates that there's very little degradation in performance between the 9th (best) model and the 10th model (a change in the 4th significant digit). Best practice would be to remove those attributes even if they were better in the 4th significant digit in order to be conservative.

Control Overfitting by Penalizing Regression Coefficients—Ridge Regression

This section describes another method for modifying ordinary least squares regression to control model complexity and to avoid overfitting. This method serves as a first introduction to penalized linear regression. You'll see more coverage of this in Chapter 4.

Ordinary least squares regression seeks to find scalar β_0 and vector β that satisfy (see Equation 3-14).

$$\beta_0^*, \beta^* = argmin_{\beta_0, \beta}\left(\frac{1}{m}\sum_{i=1}^{m}\left(y_i - (\beta_0 + x_i\beta)\right)^2\right)$$

Equation 3-14: OLS minimization problem

The expression *argmin* means the "values of β_0 and β that minimize the expression." The resulting coefficients β_0^*, β^* are the ordinary least squares solution. Best subset regression and forward stepwise regression throttle back ordinary regression by limiting the number of attributes used. That's equivalent to

imposing a constraint that some of the entries in the vector β be equal to zero. Another approach is called *coefficient penalized regression*. Coefficient penalized regression accomplishes the same thing by making all the coefficients smaller instead of making some of them zero. One version of coefficient penalized linear regression is called *ridge regression*. Equation 3-15 shows the problem formulation for ridge regression.

$$\beta_0^*, \beta^* = argmin_{\beta_0, \beta} \left(\frac{1}{m} \sum_{i=1}^{m} \left(y_i - (\beta_0 + x_i \beta) \right)^2 + \alpha \beta^T \beta \right)$$

Equation 3-15: Ridge regression minimization problem

The difference between Equation 3-15 and ordinary least squares (Equation 3-14) is the addition of the $\alpha\beta^T\beta$ term. The $\beta^T\beta$ term is the square of the Euclidean norm of β (the vector of coefficients). The variable β is a complexity parameter for this formulation of the problem. If $\alpha = 0$, the problem becomes ordinary least squares regression. When α becomes large, β (the vector of coefficients) approaches zero, and only the constant term β_0 is available to predict the labels y_i. Ridge regression is available in scikit-learn. Listing 3-5 shows the code for solving the wine taste regression problem using ridge regression.

Listing 3-5: Predicting Wine Taste with Ridge Regression—ridgeWine.py

```
__author__ = 'mike-bowles'

import urllib2
import numpy
from sklearn import datasets, linear_model
from math import sqrt
import matplotlib.pyplot as plt

#read data into iterable
target_url = ("http://archive.ics.uci.edu/ml/machine-learning-
databases/"
"wine-quality/winequality-red.csv")
data = urllib2.urlopen(target_url)

xList = []
labels = []
names = []
firstLine = True
for line in data:
    if firstLine:
        names = line.strip().split(";")
        firstLine = False
    else:
        #split on semi-colon
        row = line.strip().split(";")
```

continues

continued

```
        #put labels in separate array
        labels.append(float(row[-1]))
        #remove label from row
        row.pop()
        #convert row to floats
        floatRow = [float(num) for num in row]
        xList.append(floatRow)

#divide attributes and labels into training and test sets
indices = range(len(xList))
xListTest = [xList[i] for i in indices if i%3 == 0 ]
xListTrain = [xList[i] for i in indices if i%3 != 0 ]
labelsTest = [labels[i] for i in indices if i%3 == 0]
labelsTrain = [labels[i] for i in indices if i%3 != 0]

xTrain = numpy.array(xListTrain); yTrain = numpy.array(labelsTrain)
xTest = numpy.array(xListTest); yTest = numpy.array(labelsTest)

alphaList = [0.1**i for i in [0,1, 2, 3, 4, 5, 6]]

rmsError = []
for alph in alphaList:
    wineRidgeModel = linear_model.Ridge(alpha=alph)
    wineRidgeModel.fit(xTrain, yTrain)
    rmsError.append(numpy.linalg.norm((yTest-wineRidgeModel.predict(
       xTest)), 2)/sqrt(len(yTest)))

print("RMS Error             alpha")
for i in range(len(rmsError)):
    print(rmsError[i], alphaList[i])

#plot curve of out-of-sample error versus alpha
x = range(len(rmsError))
plt.plot(x, rmsError, 'k')
plt.xlabel('-log(alpha)')
plt.ylabel('Error (RMS)')
plt.show()

#Plot histogram of out of sample errors for best alpha value and
#scatter plot of actual versus predicted

#Identify index corresponding to min value, retrain with
#the corresponding value of alpha

#Use resulting model to predict against out of sample data.
#Plot errors (aka residuals)
indexBest = rmsError.index(min(rmsError))
alph = alphaList[indexBest]
wineRidgeModel = linear_model.Ridge(alpha=alph)
wineRidgeModel.fit(xTrain, yTrain)
errorVector = yTest-wineRidgeModel.predict(xTest)
```

```
plt.hist(errorVector)
plt.xlabel("Bin Boundaries")
plt.ylabel("Counts")
plt.show()

plt.scatter(wineRidgeModel.predict(xTest), yTest, s=100, alpha=0.10)
plt.xlabel('Predicted Taste Score')
plt.ylabel('Actual Taste Score')
plt.show()
```

Recall that the forward stepwise regression the algorithm produced a sequence of different models—the first with one attribute, the next with two attributes, and so on until the final model included all the attributes. The code for ridge regression also has a sequence of models. Instead of different numbers of attributes, the sequence of ridge regression models have different values of α—the parameter that determines the severity of the penalty on the β's. The construction of sequence of α's decreases them by powers of 10. Generally speaking, you'll want to make them decrease exponentially, not by a fixed increment. The range needs to be fairly wide and may take some experimentation to establish.

Figure 3-16 plots the RMSE as a function of the ridge complexity parameter α. The parameter is arranged from largest value on the left to smallest value on the right. It is conventional to show the least complex model on the left side of the plot and the most complex on the right side. The plot shows much the same character as with forward stepwise regression. The errors are roughly the same, but favor forward stepwise regression slightly.

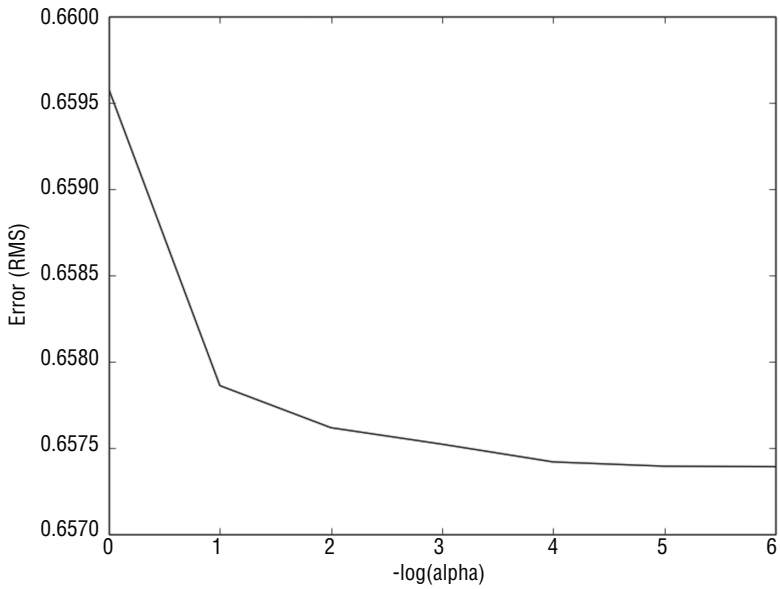

Figure 3-16: Wine quality prediction error using ridge regression

Listing 3-6 shows the output from the ridge regression. The numbers show that ridge regression has roughly the same character as forward stepwise regression. The numbers slightly favor forward stepwise regression.

Listing 3-6: Ridge Regression Output—ridgeWineOutput.txt

```
RMS Error             alpha
(0.65957881763424564, 1.0)
(0.65786109188085928, 0.1)
(0.65761721446402455, 0.010000000000000002)
(0.65752164826417536, 0.0010000000000000002)
(0.65741906801092931, 0.00010000000000000002)
(0.65739416288512531, 1.0000000000000003e-05)
(0.65739130871558593, 1.0000000000000004e-06)
```

Figure 3-17 shows the scatter plot of actual taste score versus predicted taste score for the ridge regression predictor trained on wine taste data. Figure 3-18 shows the histogram of prediction error.

You can apply the same general method to classification problems. The section "Measuring the Performance of Predictive Models" discussed several methods for quantifying classifier performance. The methods outlined included using misclassification error, associating economic costs to the various prediction outcomes, and using the area under the ROC curve (AUC) to quantify performance.

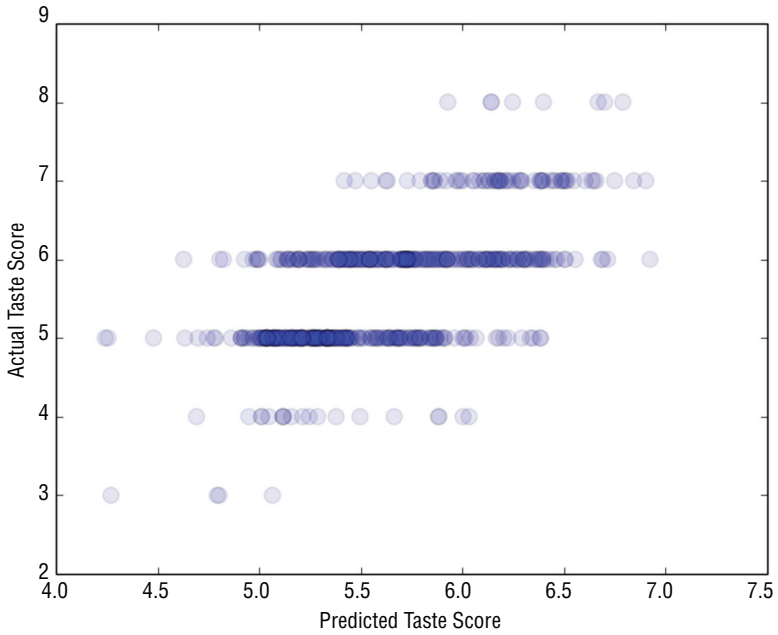

Figure 3-17: Actual taste scores versus predictions generated with ridge regression

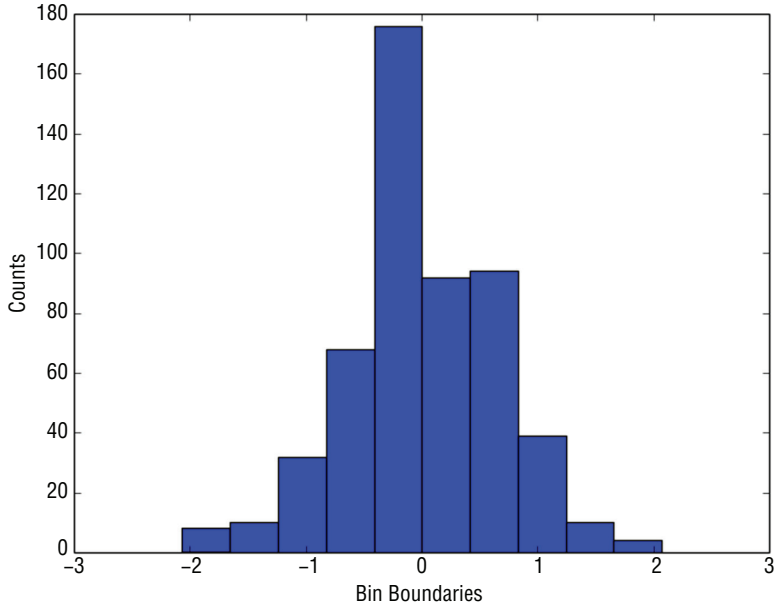

Figure 3-18: Histogram of wine taste prediction error with ridge regression

That section built a classifier using ordinary least squares regression. Listing 3-7 shows Python code that follows that same general plan. Instead of OLS, it uses ridge regression as a regression method (with a complexity tuning parameter) for building the rocks-versus-mines classifier and uses AUC as the performance measure for the classifier. The program in Listing 3-7 is similar to the wine taste prediction with ridge regression. The big difference is that the program uses the predictions on the test data and the test labels as input to the roc_curve program from the scikit-learn package. That makes it easy to calculate the AUC for each pass through the training. These are accumulated, and the printed values are shown in Listing 3-8.

Listing 3-7: Rocks Versus Mines Using Ridge Regression—classifierRidgeRocksVMines.py

```
__author__ = 'mike-bowles'
import urllib2
import numpy
from sklearn import datasets, linear_model
from sklearn.metrics import roc_curve, auc
import pylab as plt

#read data from uci data repository
target_url = ("https://archive.ics.uci.edu/ml/machine-earning-"
"databases/undocumented/connectionist-bench/sonar/sonar.all-data")
data = urllib2.urlopen(target_url)
```

continues

continued

```
#arrange data into list for labels and list of lists for attributes
xList = []
labels = []
for line in data:
    #split on comma
    row = line.strip().split(",")
    #assign label 1.0 for "M" and 0.0 for "R"
    if(row[-1] == 'M'):
        labels.append(1.0)
    else:
        labels.append(0.0)
        #remove lable from row
    row.pop()
    #convert row to floats
    floatRow = [float(num) for num in row]
    xList.append(floatRow)

#divide attribute matrix and label vector into training(2/3 of data)
#and test sets (1/3 of data)
indices = range(len(xList))
xListTest = [xList[i] for i in indices if i%3 == 0 ]
xListTrain = [xList[i] for i in indices if i%3 != 0 ]
labelsTest = [labels[i] for i in indices if i%3 == 0]
labelsTrain = [labels[i] for i in indices if i%3 != 0]

#form list of list input into numpy arrays to match input class for
#scikit-learn linear model
xTrain = numpy.array(xListTrain); yTrain = numpy.array(labelsTrain)
xTest = numpy.array(xListTest); yTest = numpy.array(labelsTest)

alphaList = [0.1**i for i in [-3, -2, -1, 0,1, 2, 3, 4, 5]]

aucList = []
for alph in alphaList:
    rocksVMinesRidgeModel = linear_model.Ridge(alpha=alph)
    rocksVMinesRidgeModel.fit(xTrain, yTrain)
    fpr, tpr, thresholds = roc_curve(yTest,rocksVMinesRidgeModel.
      predict(xTest))
    roc_auc = auc(fpr, tpr)
    aucList.append(roc_auc)


print("AUC             alpha")
for i in range(len(aucList)):
    print(aucList[i], alphaList[i])

#plot auc values versus alpha values
x = [-3, -2, -1, 0,1, 2, 3, 4, 5]
plt.plot(x, aucList)
plt.xlabel('-log(alpha)')
```

```
plt.ylabel('AUC')
plt.show()

#visualize the performance of the best classifier
indexBest = aucList.index(max(aucList))
alph = alphaList[indexBest]
rocksVMinesRidgeModel = linear_model.Ridge(alpha=alph)
rocksVMinesRidgeModel.fit(xTrain, yTrain)

#scatter plot of actual vs predicted
plt.scatter(rocksVMinesRidgeModel.predict(xTest),
   yTest, s=100, alpha=0.25)
plt.xlabel("Predicted Value")
plt.ylabel("Actual Value")
plt.show()
```

Listing 3-8 shows the AUC and associated alpha (multiplier on the coefficient penalty).

Listing 3-8: Output from Classification Model for Rocks Versus Mines Using Ridge Regression—classifierRidgeRocksVMinesOutput.txt

```
AUC             alpha
(0.84111384111384113, 999.9999999999999)
(0.86404586404586403, 99.99999999999999)
(0.9074529074529073, 10.0)
(0.91809991809991809, 1.0)
(0.88288288288288286, 0.1)
(0.8615888615888615, 0.010000000000000002)
(0.85176085176085159, 0.0010000000000000002)
(0.85094185094185093, 0.00010000000000000002)
(0.84930384930384917, 1.0000000000000003e-05)
```

A value of AUC close to 1 means great performance. A value near 0.5 is not good. So the goal with AUC is to maximize it instead of minimizing it, as was done with MSE in the earlier examples. AUC shows a fairly sharp peak at $\alpha = 1.0$. The numbers and the plot show a fairly significant drop off in performance relative to $\alpha = 1.0$. Recall that as alpha gets smaller, the solution approaches the solution to the unconstrained linear regression problem. The drop-off in performance for values of alpha smaller than 1.0 indicates that the unconstrained solution won't perform as well as ridge regression does. In the earlier section "Measuring Performance of Predictive Models," you saw the results for unconstrained ordinary least squares. The AUC on in-sample data was 0.98, and on out-of-sample data it was 0.85—very close to the AUC using ridge regression with a relatively small alpha (1E-5). Ridge regression results in a significant improvement in performance.

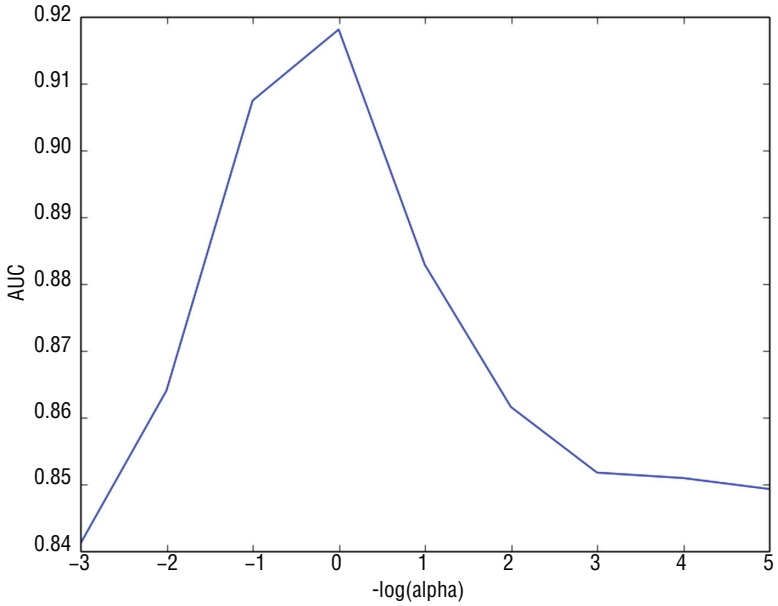

Figure 3-19: AUC for the rocks-versus-mines classifier using ridge regression

The issue here is that the attribute space for the rocks-versus-mines problem is 60 attributes wide while the full data set contains 208 rows of data. After the removal of 70 examples to be used as holdout data, 138 rows of data are available for training. That's more than twice the number of attributes, but the unconstrained (ordinary least squares) solution still overfits the data. This situation might be a good candidate for trying 10-fold cross-validation. That would result in only 20 examples (10 percent of the data set) being held out on each of the folds and might show some consequent improvement in performance. That approach comes up in Chapter 5, "Building Predictive Models Using Penalized Linear Methods."

Figure 3-19 plots the AUC as a function of the alpha parameter. That gives a visual demonstration of the value of reducing the complexity of the ordinary least squares solution by imposing a constraint on the Euclidean length of the coefficient vector.

Figure 3-20 shows the scatter plot of actual classification versus prediction for this classifier. This plot has a similar character to the scatter plot for wine prediction. Because there are a discrete number of actual outcomes, the scatter plot is composed of two horizontal rows of points.

This section introduced and explored two extensions to ordinary least squares regression. These served as illustrations of the process of training and balancing a modern predictive model. In addition, these extensions help introduce the more general penalized regression methods that will be explained in Chapter 4 and used to solve a variety of problem in Chapter 5.

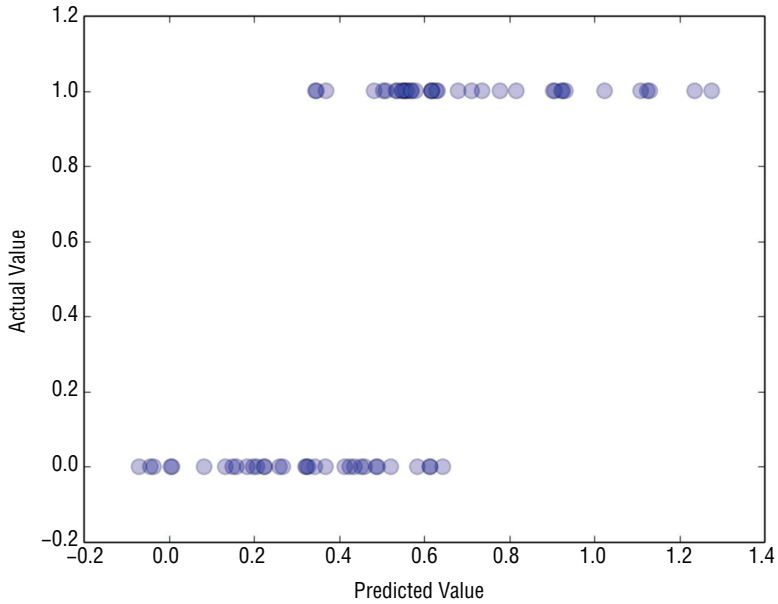

Figure 3-20: Plot of actual versus prediction for the rocks-versus-mines classifier using ridge regression

Summary

This chapter covered several topics that serve as a foundation for what comes later. First, the chapter provided visual demonstrations of problem complexity and model complexity and discussed how those factors and data set sizes conspire to determine classifier performance on a given problem. The discussion then turned to a number of different metrics for prediction performance associated with the different problem types (regression, classification, and multiclass classification) that arise as part of the function approximation problem. The chapter described two methods (holdout and n-fold cross-validation) for estimating performance on new data. The chapter introduced the conceptual framework that a machine learning technique produces a parameterized family of models and that one of these is selected for deployment on the basis of out-of-sample performance. Several examples based on modifications of ordinary least squares regression (forward stepwise regression and ridge regression) then instantiated that conceptual framework.

References

1. David J. Hand and Robert J. Till (2001). A Simple Generalization of the Area Under the ROC Curve for Multiple Class Classification Problems. *Machine Learning*, 45(2), 171–186.