10
Evaluating Model Performance

When only the wealthy could afford education, tests and exams did not evaluate students' potential. Instead, teachers were judged for parents who wanted to know whether their children had learned enough to justify the instructors' wages. Obviously, this has changed over the years. Now, such evaluations are used to distinguish between high- and low-achieving students, filtering them into careers and other opportunities.

Given the significance of this process, a great deal of effort is invested in developing accurate student assessments. Fair assessments have a large number of questions that cover a wide breadth of topics and reward true knowledge over lucky guesses. They also require students to think about problems they have never faced before. Correct responses therefore indicate that students can generalize their knowledge more broadly.

The process of evaluating machine learning algorithms is very similar to the process of evaluating students. Since algorithms have varying strengths and weaknesses, tests should distinguish among the learners. It is also important to forecast how a learner will perform on future data.

This chapter provides the information needed to assess machine learners, such as:

- The reasons why predictive accuracy is not sufficient to measure performance, and the performance measures you might use instead
- Methods to ensure that the performance measures reasonably reflect a model's ability to predict or forecast unseen cases
- How to use R to apply these more useful measures and methods to the predictive models covered in the previous chapters

Just as the best way to learn a topic is to attempt to teach it to someone else, the process of teaching and evaluating machine learners will provide you with greater insight into the methods you've learned so far.

Measuring performance for classification

In the previous chapters, we measured classifier accuracy by dividing the proportion of correct predictions by the total number of predictions. This indicates the percentage of cases in which the learner is right or wrong. For example, suppose that for 99,990 out of 100,000 newborn babies a classifier correctly predicted whether they were a carrier of a treatable but potentially fatal genetic defect. This would imply an accuracy of 99.99 percent and an error rate of only 0.01 percent.

At first glance, this appears to be an extremely accurate classifier. However, it would be wise to collect additional information before trusting your child's life to the test. What if the genetic defect is found in only 10 out of every 100,000 babies? A test that predicts *no defect* regardless of the circumstances will be correct for 99.99 percent of all cases, but incorrect for 100 percent of the cases that matter most. In other words, even though the predictions are extremely accurate, the classifier is not very useful to prevent treatable birth defects.

This is one consequence of the **class imbalance problem**, which refers to the trouble associated with data having a large majority of records belonging to a single class.

Though there are many ways to measure a classifier's performance, the best measure is always the one that captures whether the classifier is successful at its intended purpose. It is crucial to define the performance measures for utility rather than raw accuracy. To this end, we will begin exploring a variety of alternative performance measures derived from the confusion matrix. Before we get started, however, we need to consider how to prepare a classifier for evaluation.

Working with classification prediction data in R

The goal of evaluating a classification model is to have a better understanding of how its performance will extrapolate to future cases. Since it is usually unfeasible to test a still-unproven model in a live environment, we typically simulate future conditions by asking the model to classify a dataset made of cases that resemble what it will be asked to do in the future. By observing the learner's responses to this examination, we can learn about its strengths and weaknesses.

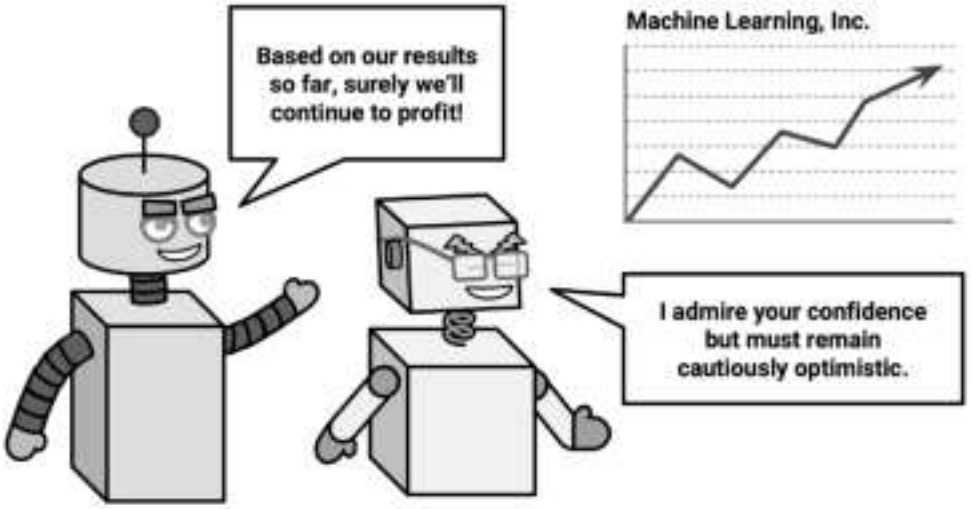
Though we've evaluated classifiers in the prior chapters, it's worth reflecting on the types of data at our disposal:

- Actual class values
- Predicted class values
- Estimated probability of the prediction

The actual and predicted class values may be self-evident, but they are the key to evaluation. Just like a teacher uses an answer key to assess the student's answers, we need to know the correct answer for a machine learner's predictions. The goal is to maintain two vectors of data: one holding the correct or actual class values, and the other holding the predicted class values. Both vectors must have the same number of values stored in the same order. The predicted and actual values may be stored as separate R vectors or columns in a single R data frame.

Obtaining this data is easy. The actual class values come directly from the target feature in the test dataset. Predicted class values are obtained from the classifier built upon the training data, and applied to the test data. For most machine learning packages, this involves applying the `predict()` function to a model object and a data frame of test data, such as: `predicted_outcome <- predict(model, test_data)`.

Until now, we have only examined classification predictions using these two vectors of data. Yet most models can supply another piece of useful information. Even though the classifier makes a single prediction about each example, it may be more confident about some decisions than others. For instance, a classifier may be 99 percent certain that an SMS with the words "free" and "ringtones" is spam, but is only 51 percent certain that an SMS with the word "tonight" is spam. In both cases, the classifier classifies the message as spam, but it is far more certain about one decision than the other.



Studying these internal prediction probabilities provides useful data to evaluate a model's performance. If two models make the same number of mistakes, but one is more capable of accurately assessing its uncertainty, then it is a smarter model. It's ideal to fid a learner that is extremely confident when making a correct prediction, but timid in the face of doubt. The balance between confidence and caution is a key part of model evaluation.

Unfortunately, obtaining internal prediction probabilities can be tricky because the method to do so varies across classifiers. In general, for most classifiers, the `predict()` function is used to specify the desired type of prediction. To obtain a single predicted class, such as spam or ham, you typically set the `type = "class"` parameter. To obtain the prediction probability, the `type` parameter should be set to one of `"prob"`, `"posterior"`, `"raw"`, or `"probability"` depending on the classifier used.

Nearly all of the classifiers presented in this book will provide prediction probabilities. The `type` parameter is included in the syntax box introducing each model.

For example, to output the predicted probabilities for the C5.0 classifier built in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, use the `predict()` function with `type = "prob"` as follows:

```
> predicted_prob <- predict(credit_model, credit_test, type = "prob")
```

To further illustrate the process of evaluating learning algorithms, let's look more closely at the performance of the SMS spam classification model developed in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*. To output the naive Bayes predicted probabilities, use `predict()` with `type = "raw"` as follows:

```
> sms_test_prob <- predict(sms_classifier, sms_test, type = "raw")
```

In most cases, the `predict()` function returns a probability for each category of the outcome. For example, in the case of a two-outcome model like the SMS classifier, the predicted probabilities might be a matrix or data frame as shown here:

```
> head(sms_test_prob)
              ham          spam
[1,] 9.999995e-01 4.565938e-07
[2,] 9.999995e-01 4.540489e-07
[3,] 9.998418e-01 1.582360e-04
[4,] 9.999578e-01 4.223125e-05
[5,] 4.816137e-10 1.000000e+00
[6,] 9.997970e-01 2.030033e-04
```

Each line in this output shows the classifier's predicted probability of `spam` and `ham`, which always sum up to 1 because these are the only two outcomes. While constructing an evaluation dataset, it is important to ensure that you are using the correct probability for the class level of interest. To avoid confusion, in the case of a binary outcome, you might even consider dropping the vector for one of the two alternatives.

For convenience during the evaluation process, it can be helpful to construct a data frame containing the predicted class values, actual class values, as well as the estimated probabilities of interest.

The steps required to construct the evaluation dataset have been omitted for brevity, but are included in this chapter's code on the Packt Publishing website. To follow along with the examples here, download the `sms_results.csv` file, and load to a data frame using the `sms_results <- read.csv("sms_results.csv")` command.

The `sms_results` data frame is simple. It contains four vectors of 1,390 values. One vector contains values indicating the actual type of SMS message (`spam` or `ham`), one vector indicates the naive Bayes model's predicted type, and the third and fourth vectors indicate the probability that the message was `spam` or `ham`, respectively:

```
> head(sms_results)
  actual_type predict_type prob_spam prob_ham
1         ham          ham   0.00000  1.00000
2         ham          ham   0.00000  1.00000
3         ham          ham   0.00016  0.99984
4         ham          ham   0.00004  0.99996
5        spam         spam   1.00000  0.00000
6         ham          ham   0.00020  0.99980
```

For these six test cases, the predicted and actual SMS message types agree; the model predicted their status correctly. Furthermore, the prediction probabilities suggest that model was extremely confident about these predictions, because they all fall close to zero or one.

What happens when the predicted and actual values are further from zero and one? Using the `subset()` function, we can identify a few of these records. The following output shows test cases where the model estimated the probability of `spam` somewhere between 40 and 60 percent:

```
> head(subset(sms_results, prob_spam > 0.40 & prob_spam < 0.60))
     actual_type predict_type prob_spam prob_ham
377         spam          ham   0.47536  0.52464
717          ham         spam   0.56188  0.43812
1311         ham         spam   0.57917  0.42083
```

By the model's own admission, these were cases in which a correct prediction was virtually a coin flip. Yet all three predictions were wrong—an unlucky result. Let's look at a few more cases where the model was wrong:

```
> head(subset(sms_results, actual_type != predict_type))
    actual_type predict_type prob_spam prob_ham
```

53	spam	ham	0.00071	0.99929
59	spam	ham	0.00156	0.99844
73	spam	ham	0.01708	0.98292
76	spam	ham	0.00851	0.99149
184	spam	ham	0.01243	0.98757
332	spam	ham	0.00003	0.99997

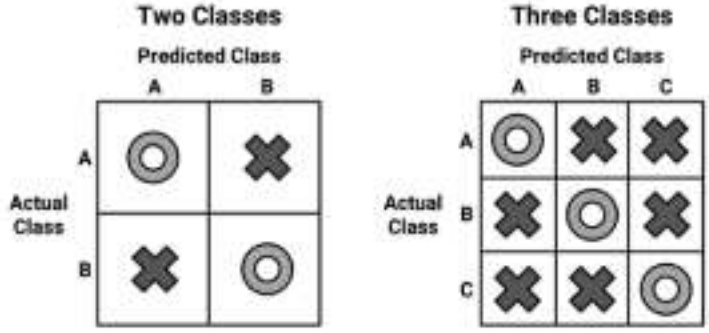
These cases illustrate the important fact that a model can be extremely confident and yet it can be extremely wrong. All six of these test cases were `spam` that the classifier believed to have no less than a 98 percent chance of being `ham`.

In spite of such mistakes, is the model still useful? We can answer this question by applying various error metrics to the evaluation data. In fact, many such metrics are based on a tool we've already used extensively in the previous chapters.

A closer look at confusion matrices

A **confusion matrix** is a table that categorizes predictions according to whether they match the actual value. One of the table's dimensions indicates the possible categories of predicted values, while the other dimension indicates the same for actual values. Although we have only seen 2 x 2 confusion matrices so far, a matrix can be created for models that predict any number of class values. The following figure depicts the familiar confusion matrix for a two-class binary model as well as the 3 x 3 confusion matrix for a three-class model.

When the predicted value is the same as the actual value, it is a correct classification. Correct predictions fall on the diagonal in the confusion matrix (denoted by O). The off-diagonal matrix cells (denoted by X) indicate the cases where the predicted value differs from the actual value. These are incorrect predictions. The performance measures for classification models are based on the counts of predictions falling on and off the diagonal in these tables:



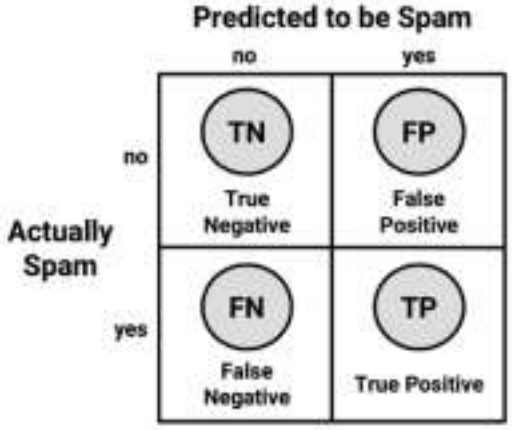
The most common performance measures consider the model's ability to discern one class versus all others. The class of interest is known as the **positive** class, while all others are known as **negative**.

The use of the terms positive and negative is not intended to imply any value judgment (that is, good versus bad), nor does it necessarily suggest that the outcome is present or absent (such as birth defect versus none). The choice of the positive outcome can even be arbitrary, as in cases where a model is predicting categories such as sunny versus rainy or dog versus cat.

The relationship between the positive class and negative class predictions can be depicted as a 2 x 2 confusion matrix that tabulates whether predictions fall into one of the four categories:

- **True Positive (TP)**: Correctly classified as the class of interest
- **True Negative (TN)**: Correctly classified as not the class of interest
- **False Positive (FP)**: Incorrectly classified as the class of interest
- **False Negative (FN)**: Incorrectly classified as not the class of interest

For the spam classifier, the positive class is `spam`, as this is the outcome we hope to detect. We can then imagine the confusion matrix as shown in the following diagram:



The confusion matrix, presented in this way, is the basis for many of the most important measures of model's performance. In the next section, we'll use this matrix to have a better understanding of what is meant by accuracy.

Using confusion matrices to measure performance

With the 2 x 2 confusion matrix, we can formalize our definition of prediction **accuracy** (sometimes called the **success rate**) as:

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

In this formula, the terms *TP*, *TN*, *FP*, and *FN* refer to the number of times the model's predictions fell into each of these categories. The accuracy is therefore a proportion that represents the number of true positives and true negatives, divided by the total number of predictions.

The **error rate** or the proportion of the incorrectly classified examples is specified as:

$$\text{error rate} = \frac{\text{FP} + \text{FN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = 1 - \text{accuracy}$$

Notice that the error rate can be calculated as one minus the accuracy. Intuitively, this makes sense; a model that is correct 95 percent of the time is incorrect 5 percent of the time.

An easy way to tabulate a classifier's predictions into a confusion matrix is to use R's `table()` function. The command to create a confusion matrix for the SMS data is shown as follows. The counts in this table could then be used to calculate accuracy and other statistics:

```
> table(sms_results$actual_type, sms_results$predict_type)

        ham spam
  ham  1203    4
  spam   31  152
```

If you like to create a confusion matrix with a more informative output, the `CrossTable()` function in the `gmodels` package offers a customizable solution. If you recall, we first used this function in *Chapter 2, Managing and Understanding Data*. If you didn't install the package at that time, you will need to do so using the `install.packages("gmodels")` command.

By default, the `CrossTable()` output includes proportions in each cell that indicate the cell count as a percentage of table's row, column, or overall total counts. The output also includes row and column totals. As shown in the following code, the syntax is similar to the `table()` function:

```
> library(gmodels)
> CrossTable(sms_results$actual_type, sms_results$predict_type)
```

The result is a confusion matrix with a wealth of additional detail:

```
   Cell Contents
|-------------------------|
|                       N |
| Chi-square contribution |
|           N / Row Total |
|           N / Col Total |
|         N / Table Total |
|-------------------------|


Total Observations in Table:  1390


                         | sms_results$predict_type
sms_results$actual_type |       ham |      spam | Row Total |
-------------------------|-----------|-----------|-----------|
                     ham |      1203 |         4 |      1207 |
                         |    16.128 |   127.580 |           |
                         |     0.997 |     0.003 |     0.868 |
                         |     0.975 |     0.026 |           |
                         |     0.865 |     0.003 |           |
-------------------------|-----------|-----------|-----------|
                    spam |        31 |       152 |       183 |
                         |   106.377 |   841.470 |           |
                         |     0.169 |     0.831 |     0.132 |
                         |     0.025 |     0.974 |           |
                         |     0.022 |     0.109 |           |
-------------------------|-----------|-----------|-----------|
            Column Total |      1234 |       156 |      1390 |
                         |     0.888 |     0.112 |           |
-------------------------|-----------|-----------|-----------|
```

We've used `CrossTable()` in several of the previous chapters, so by now you should be familiar with the output. If you ever forget how to interpret the output, simply refer to the key (labeled `Cell Contents`), which provides the definition of each number in the table cells.

We can use the confusion matrix to obtain the accuracy and error rate. Since the accuracy is *(TP + TN) / (TP + TN + FP + FN)*, we can calculate it using following command:

```
> (152 + 1203) / (152 + 1203 + 4 + 31)
[1] 0.9748201
```

We can also calculate the error rate *(FP + FN) / (TP + TN + FP + FN)* as:

```
> (4 + 31) / (152 + 1203 + 4 + 31)
[1] 0.02517986
```

This is the same as one minus accuracy:

```
> 1 - 0.9748201
[1] 0.0251799
```

Although these calculations may seem simple, it is important to practice thinking about how the components of the confusion matrix relate to one another. In the next section, you will see how these same pieces can be combined in different ways to create a variety of additional performance measures.

Beyond accuracy – other measures of performance

Countless performance measures have been developed and used for specific purposes in disciplines as diverse as medicine, information retrieval, marketing, and signal detection theory, among others. Covering all of them could fill hundreds of pages and makes a comprehensive description infeasible here. Instead, we'll consider only some of the most useful and commonly cited measures in the machine learning literature.

The Classification and Regression Training package `caret` by Max Kuhn includes functions to compute many such performance measures. This package provides a large number of tools to prepare, train, evaluate, and visualize machine learning models and data. In addition to its use here, we will also employ `caret` extensively in *Chapter 11, Improving Model Performance*. Before proceeding, you will need to install the package using the `install.packages("caret")` command.

For more information on `caret`, please refer to: Kuhn M. Building predictive models in R using the caret package. *Journal of Statistical* Software. 2008; 28.

The `caret` package adds yet another function to create a confusion matrix. As shown in the following command, the syntax is similar to `table()`, but with a minor difference. Because `caret` provides measures of model performance that consider the ability to classify the positive class, a `positive` parameter should be specified. In this case, since the SMS classifier is intended to detect `spam`, we will set `positive = "spam"` as follows:

```
> library(caret)
> confusionMatrix(sms_results$predict_type,
    sms_results$actual_type, positive = "spam")
```

This results in the following output:

```
Confusion Matrix and Statistics

          Reference
Prediction  ham spam
      ham  1203   31
      spam    4  152

               Accuracy : 0.9748
                 95% CI : (0.9652, 0.9824)
    No Information Rate : 0.8683
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.8825
 Mcnemar's Test P-Value : 1.109e-05

            Sensitivity : 0.8306
            Specificity : 0.9967
         Pos Pred Value : 0.9744
         Neg Pred Value : 0.9749
             Prevalence : 0.1317
         Detection Rate : 0.1094
   Detection Prevalence : 0.1122
      Balanced Accuracy : 0.9136

       'Positive' Class : spam
```

At the top of the output is a confusion matrix much like the one produced by the `table()` function, but transposed. The output also includes a set of performance measures. Some of these, like accuracy, are familiar, while many others are new. Let's take a look at few of the most important metrics.

The kappa statistic

The **kappa statistic** (labeled `Kappa` in the previous output) adjusts accuracy by accounting for the possibility of a correct prediction by chance alone. This is especially important for datasets with a severe class imbalance, because a classifier can obtain high accuracy simply by always guessing the most frequent class. The kappa statistic will only reward the classifier if it is correct more often than this simplistic strategy.

Kappa values range from 0 to a maximum of 1, which indicates perfect agreement between the model's predictions and the true values. Values less than one indicate imperfect agreement. Depending on how a model is to be used, the interpretation of the kappa statistic might vary. One common interpretation is shown as follows:

- Poor agreement = less than 0.20
- Fair agreement = 0.20 to 0.40
- Moderate agreement = 0.40 to 0.60
- Good agreement = 0.60 to 0.80
- Very good agreement = 0.80 to 1.00

It's important to note that these categories are subjective. While a "good agreement" may be more than adequate to predict someone's favorite ice cream flavor, "very good agreement" may not suffice if your goal is to identify birth defects.

For more information on the previous scale, refer to: Landis JR, Koch GG. The measurement of observer agreement for categorical data. *Biometrics*. 1997; 33:159-174.

The following is the formula to calculate the kappa statistic. In this formula, *Pr(a)* refers to the proportion of the actual agreement and *Pr(e)* refers to the expected agreement between the classifier and the true values, under the assumption that they were chosen at random:

$$\kappa = \frac{\Pr(a) - \Pr(e)}{1 - \Pr(e)}$$

There is more than one way to define the kappa statistic. The most common method described here uses **Cohen's kappa coefficient**, as described in the paper: Cohen J. A coefficient of agreement for nominal scales. *Education and Psychological Measurement*. 1960; 20:37-46.

These proportions are easy to obtain from a confusion matrix once you know where to look. Let's consider the confusion matrix for the SMS classification model created with the `CrossTable()` function, which is repeated here for convenience:

```
                     | sms_results$predict_type
sms_results$actual_type |       ham |      spam | Row Total |
-----------------------|-----------|-----------|-----------|
                   ham |      1203 |         4 |      1207 |
                       |    16.128 |   127.580 |           |
                       |     0.997 |     0.003 |     0.868 |
                       |     0.975 |     0.026 |           |
                       |     0.865 |     0.003 |           |
-----------------------|-----------|-----------|-----------|
                  spam |        31 |       152 |       183 |
                       |   106.377 |   841.470 |           |
                       |     0.169 |     0.831 |     0.132 |
                       |     0.025 |     0.974 |           |
                       |     0.022 |     0.109 |           |
-----------------------|-----------|-----------|-----------|
          Column Total |      1234 |       156 |      1390 |
                       |     0.888 |     0.112 |           |
-----------------------|-----------|-----------|-----------|
```

Remember that the bottom value in each cell indicates the proportion of all instances falling into that cell. Therefore, to calculate the observed agreement *Pr(a)*, we simply add the proportion of all instances where the predicted type and actual SMS type agree. Thus, we can calculate *Pr(a)* as:

```
> pr_a <- 0.865 + 0.109
> pr_a
[1] 0.974
```

For this classifier, the observed and actual values agree 97.4 percent of the time—you will note that this is the same as the accuracy. The kappa statistic adjusts the accuracy relative to the expected agreement *Pr(e)*, which is the probability that the chance alone would lead the predicted and actual values to match, under the assumption that both are selected randomly according to the observed proportions.

To find these observed proportions, we can use the probability rules we learned in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*. Assuming two events are independent (meaning that one does not affect the other), probability rules note that the probability of both occurring is equal to the product of the probabilities of each one occurring. For instance, we know that the probability of both choosing ham is:

*Pr(actual type is ham) * Pr(predicted type is ham)*

The probability of both choosing spam is:

*Pr(actual type is spam) * Pr(predicted type is spam)*

The probability that the predicted or actual type is spam or ham can be obtained from the row or column totals. For instance, *Pr(actual type is ham) = 0.868* and *Pr(predicted type is ham) = 0.888*.

Pr(e) is calculated as the sum of the probabilities that by chance the predicted and actual values agree that the message is either spam or ham. Recall that for mutually exclusive events (events that cannot happen simultaneously), the probability of either occurring is equal to the sum of their probabilities. Therefore, to obtain the final *Pr(e)*, we simply add both products, as shown in the following commands:

```
> pr_e <- 0.868 * 0.888 + 0.132 * 0.112
> pr_e
[1] 0.785568
```

Since *Pr(e)* is 0.786, by chance alone, we would expect the observed and actual values to agree about 78.6 percent of the time.

This means that we now have all the information needed to complete the kappa formula. Plugging the *Pr(a)* and *Pr(e)* values into the kappa formula, we find:

```
> k <- (pr_a - pr_e) / (1 - pr_e)
> k
[1] 0.8787494
```

The kappa is about 0.88, which agrees with the previous `confusionMatrix()` output from `caret` (the small difference is due to rounding). Using the suggested interpretation, we note that there is very good agreement between the classifier's predictions and the actual values.

There are a couple of R functions to calculate kappa automatically. The `Kappa()` function (be sure to note the capital 'K') in the Visualizing Categorical Data (`vcd`) package uses a confusion matrix of predicted and actual values. After installing the package by typing `install.packages("vcd")`, the following commands can be used to obtain kappa:

```
> library(vcd)
> Kappa(table(sms_results$actual_type, sms_results$predict_type))
             value        ASE
Unweighted 0.8825203 0.01949315
Weighted   0.8825203 0.01949315
```

We're interested in the unweighted kappa. The value 0.88 matches what we expected.

The weighted kappa is used when there are varying degrees of agreement. For example, using a scale of cold, cool, warm, and hot, a value of warm agrees more with hot than it does with the value of cold. In the case of a two-outcome event, such as spam and ham, the weighted and unweighted kappa statistics will be identical.

The `kappa2()` function in the Inter-Rater Reliability (`irr`) package can be used to calculate kappa from the vectors of predicted and actual values stored in a data frame. After installing the package using the `install.packages("irr")` command, the following commands can be used to obtain kappa:

```
> kappa2(sms_results[1:2])
 Cohen's Kappa for 2 Raters (Weights: unweighted)

 Subjects = 1390
   Raters = 2
    Kappa = 0.883

        z = 33
  p-value = 0
```

The `Kappa()` and `kappa2()` functions report the same kappa statistic, so use whichever option you are more comfortable with.

Be careful not to use the built-in `kappa()` function. It is completely unrelated to the kappa statistic reported previously!

Sensitivity and specificity

Finding a useful classifier often involves a balance between predictions that are overly conservative and overly aggressive. For example, an e-mail filter could guarantee to eliminate every spam message by aggressively eliminating nearly every ham message at the same time. On the other hand, guaranteeing that no ham message is inadvertently filtered might require us to allow an unacceptable amount of spam to pass through the filter. A pair of performance measures captures this tradeoff: sensitivity and specificity.

The **sensitivity** of a model (also called the **true positive rate**) measures the proportion of positive examples that were correctly classified. Therefore, as shown in the following formula, it is calculated as the number of true positives divided by the total number of positives, both correctly classified (the true positives) as well as incorrectly classified (the false negatives):

$$\text{sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

The **specificity** of a model (also called the **true negative rate**) measures the proportion of negative examples that were correctly classified. As with sensitivity, this is computed as the number of true negatives, divided by the total number of negatives—the true negatives plus the false positives:

$$\text{specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Given the confusion matrix for the SMS classifier, we can easily calculate these measures by hand. Assuming that spam is the positive class, we can confirm that the numbers in the `confusionMatrix()` output are correct. For example, the calculation for sensitivity is:

```
> sens <- 152 / (152 + 31)
> sens
[1] 0.8306011
```

Similarly, for specificity we can calculate:

```
> spec <- 1203 / (1203 + 4)
> spec
[1] 0.996686
```

The `caret` package provides functions to calculate sensitivity and specificity directly from the vectors of predicted and actual values. Be careful that you specify the `positive` or `negative` parameter appropriately, as shown in the following lines:

```
> library(caret)
> sensitivity(sms_results$predict_type, sms_results$actual_type,
              positive = "spam")
```

```
[1] 0.8306011

> specificity(sms_results$predict_type, sms_results$actual_type,
              negative = "ham")
[1] 0.996686
```

Sensitivity and specificity range from 0 to 1, with values close to 1 being more desirable. Of course, it is important to find an appropriate balance between the two—a task that is often quite context-specific.

For example, in this case, the sensitivity of 0.831 implies that 83.1 percent of the spam messages were correctly classified. Similarly, the specificity of 0.997 implies that 99.7 percent of the nonspam messages were correctly classified or, alternatively, 0.3 percent of the valid messages were rejected as spam. The idea of rejecting 0.3 percent of valid SMS messages may be unacceptable, or it may be a reasonable trade-off given the reduction in spam.

Sensitivity and specificity provide tools for thinking about such trade-offs. Typically, changes are made to the model and different models are tested until you find one that meets a desired sensitivity and specificity threshold. Visualizations, such as those discussed later in this chapter, can also assist with understanding the trade-off between sensitivity and specificity.

Precision and recall

Closely related to sensitivity and specificity are two other performance measures related to compromises made in classification: precision and recall. Used primarily in the context of information retrieval, these statistics are intended to provide an indication of how interesting and relevant a model's results are, or whether the predictions are diluted by meaningless noise.

The **precision** (also known as the **positive predictive value**) is defined as the proportion of positive examples that are truly positive; in other words, when a model predicts the positive class, how often is it correct? A precise model will only predict the positive class in cases that are very likely to be positive. It will be very trustworthy.

Consider what would happen if the model was very imprecise. Over time, the results would be less likely to be trusted. In the context of information retrieval, this would be similar to a search engine such as Google returning unrelated results. Eventually, users would switch to a competitor like Bing. In the case of the SMS spam filter, high precision means that the model is able to carefully target only the spam while ignoring the ham.

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

On the other hand, **recall** is a measure of how complete the results are. As shown in the following formula, this is defined as the number of true positives over the total number of positives. You may have already recognized this as the same as sensitivity. However, in this case, the interpretation differs slightly. A model with a high recall captures a large portion of the positive examples, meaning that it has wide breadth. For example, a search engine with a high recall returns a large number of documents pertinent to the search query. Similarly, the SMS spam filter has a high recall if the majority of spam messages are correctly identified.

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

We can calculate precision and recall from the confusion matrix. Again, assuming that `spam` is the positive class, the precision is:

```
> prec <- 152 / (152 + 4)
> prec
[1] 0.974359
```

The recall is:

```
> rec <- 152 / (152 + 31)
> rec
[1] 0.8306011
```

The `caret` package can be used to compute either of these measures from the vectors of predicted and actual classes. Precision uses the `posPredValue()` function:

```
> library(caret)
> posPredValue(sms_results$predict_type, sms_results$actual_type,
               positive = "spam")
[1] 0.974359
```

While recall uses the `sensitivity()` function that we used earlier:

```
> sensitivity(sms_results$predict_type, sms_results$actual_type,
              positive = "spam")
[1] 0.8306011
```

Similar to the inherent trade-off between sensitivity and specificity, for most of the real-world problems, it is difficult to build a model with both high precision and high recall. It is easy to be precise if you target only the low-hanging fruit—the easy to classify examples. Similarly, it is easy for a model to have high recall by casting a very wide net, meaning that the model is overly aggressive in identifying the positive cases. In contrast, having both high precision and recall at the same time is very challenging. It is therefore important to test a variety of models in order to find the combination of precision and recall that will meet the needs of your project.

The F-measure

A measure of model performance that combines precision and recall into a single number is known as the **F-measure** (also sometimes called the **F_1 score** or **F-score**). The F-measure combines precision and recall using the **harmonic mean**, a type of average that is used for rates of change. The harmonic mean is used rather than the common arithmetic mean since both precision and recall are expressed as proportions between zero and one, which can be interpreted as rates. The following is the formula for the F-measure:

$$\text{F-measure} = \frac{2 \times \text{precision} \times \text{recall}}{\text{recall} + \text{precision}} = \frac{2 \times \text{TP}}{2 \times \text{TP} + \text{FP} + \text{FN}}$$

To calculate the F-measure, use the precision and recall values computed previously:

```
> f <- (2 * prec * rec) / (prec + rec)
> f
[1] 0.8967552
```

This comes out exactly the same as using the counts from the confusion matrix:

```
> f <- (2 * 152) / (2 * 152 + 4 + 31)
> f
[1] 0.8967552
```

Since the F-measure describes the model performance in a single number, it provides a convenient way to compare several models side by side. However, this assumes that equal weight should be assigned to precision and recall, an assumption that is not always valid. It is possible to calculate F-scores using different weights for precision and recall, but choosing the weights could be tricky at the best and arbitrary at worst. A better practice is to use measures such as the F-score in combination with methods that consider a model's strengths and weaknesses more globally, such as those described in the next section.

Visualizing performance trade-offs

Visualizations are helpful to understand the performance of machine learning algorithms in greater detail. Where statistics such as sensitivity and specificity or precision and recall attempt to boil model performance down to a single number, visualizations depict how a learner performs across a wide range of conditions.

Because learning algorithms have different biases, it is possible that two models with similar accuracy could have drastic differences in how they achieve their accuracy. Some models may struggle with certain predictions that others make with ease, while breezing through the cases that others cannot get right. Visualizations provide a method to understand these trade-offs, by comparing learners side by side in a single chart.

The `ROCR` package provides an easy-to-use suite of functions for visualizing for visualizing the performance of classification models. It includes functions for computing large set of the most common performance measures and visualizations. The `ROCR` website at `http://rocr.bioinf.mpi-sb.mpg.de/` includes a list of the full set of features as well as several examples on visualization capabilities. Before continuing, install the package using the `install.packages("ROCR")` command.

For more information on the development of ROCR, see : Sing T, Sander O, Beerenwinkel N, Lengauer T. ROCR: visualizing classifier performance in R. *Bioinformatics*. 2005; 21:3940-3941.

To create visualizations with `ROCR`, two vectors of data are needed. The first must contain the predicted class values, and the second must contain the estimated probability of the positive class. These are used to create a prediction object that can be examined with the plotting functions of `ROCR`.

The prediction object for the SMS classifier requires the classifier's estimated spam probabilities and the actual class labels. These are combined using the `prediction()` function in the following lines:

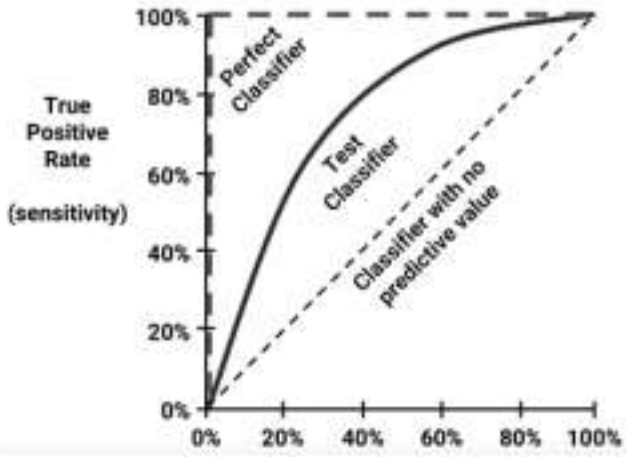
```
> library(ROCR)
> pred <- prediction(predictions = sms_results$prob_spam,
                     labels = sms_results$actual_type)
```

Next, the `performance()` function will allow us to compute measures of performance from the `prediction` object we just created, which can then be visualized using the R `plot()` function. Given these three steps, a large variety of useful visualizations can be created.

ROC curves

The **Receiver Operating Characteristic (ROC) curve** is commonly used to examine the trade-off between the detection of true positives, while avoiding the false positives. As you might suspect from the name, ROC curves were developed by engineers in the field of communications. Around the time of World War II, radar and radio operators used ROC curves to measure a receiver's ability to discriminate between true signals and false alarms. The same technique is useful today to visualize the efficacy of machine learning models.

The characteristics of a typical ROC diagram are depicted in the following plot. Curves are defined on a plot with the proportion of true positives on the vertical axis and the proportion of false positives on the horizontal axis. Because these values are equivalent to sensitivity and (1 – specificity), respectively, the diagram is also known as a sensitivity/specificity plot:



The points comprising ROC curves indicate the true positive rate at varying false positive thresholds. To create the curves, a classifier's predictions are sorted by the model's estimated probability of the positive class, with the largest values first. Beginning at the origin, each prediction's impact on the true positive rate and false positive rate will result in a curve tracing vertically (for a correct prediction) or horizontally (for an incorrect prediction).

To illustrate this concept, three hypothetical classifiers are contrasted in the previous plot. First, the diagonal line from the bottom-left to the top-right corner of the diagram represents a **classifier with no predictive value**. This type of classifier detects true positives and false positives at exactly the same rate, implying that the classifier cannot discriminate between the two. This is the baseline by which other classifiers may be judged. ROC curves falling close to this line indicate models that are not very useful. The **perfect classifier** has a curve that passes through the point at a 100 percent true positive rate and 0 percent false positive rate. It is able to correctly identify all of the positives before it incorrectly classifies any negative result. Most real-world classifiers are similar to the test classifier and they fall somewhere in the zone between perfect and useless.

The closer the curve is to the perfect classifier, the better it is at identifying positive values. This can be measured using a statistic known as the **area under the ROC curve** (abbreviated **AUC**). The AUC treats the ROC diagram as a two-dimensional square and measures the total area under the ROC curve. AUC ranges from 0.5 (for a classifier with no predictive value) to 1.0 (for a perfect classifier). A convention to interpret AUC scores uses a system similar to academic letter grades:

- **A**: Outstanding = 0.9 to 1.0
- **B**: Excellent/good = 0.8 to 0.9
- **C**: Acceptable/fair = 0.7 to 0.8
- **D**: Poor = 0.6 to 0.7
- **E**: No discrimination = 0.5 to 0.6

As with most scales similar to this, the levels may work better for some tasks than others; the categorization is somewhat subjective.

It's also worth noting that two ROC curves may be shaped very differently, yet have an identical AUC. For this reason, an AUC alone can be misleading. The best practice is to use AUC in combination with qualitative examination of the ROC curve.

Creating ROC curves with the `ROCR` package involves building a `performance` object from the `prediction` object we computed earlier. Since ROC curves plot true positive rates versus false positive rates, we simply call the `performance()` function while specifying the `tpr` and `fpr` measures, as shown in the following code:

```
> perf <- performance(pred, measure = "tpr", x.measure = "fpr")
```

Using the `perf` object, we can visualize the ROC curve with R's `plot()` function. As shown in the following code lines, many of the standard parameters to adjust the visualization can be used, such as `main` (to add a title), `col` (to change the line color), and `lwd` (to adjust the line width):

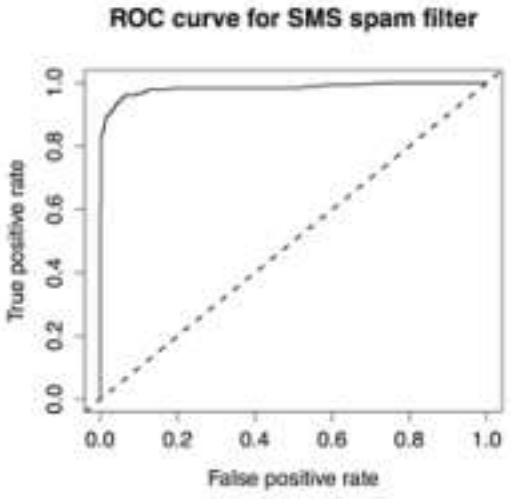
```
> plot(perf, main = "ROC curve for SMS spam filter",
       col = "blue", lwd = 3)
```

Although the `plot()` command is sufficient to create a valid ROC curve, it is helpful to add a reference line to indicate the performance of a classifier with no predictive value.

To plot such a line, we'll use the `abline()` function. This function can be used to specify a line in the slope-intercept form, where `a` is the intercept and `b` is the slope. Since we need an identity line that passes through the origin, we'll set the intercept to `a = 0` and the slope to `b = 1`, as shown in the following plot. The `lwd` parameter adjusts the line thickness, while the `lty` parameter adjusts the type of line. For example, `lty = 2` indicates a dashed line:

```
> abline(a = 0, b = 1, lwd = 2, lty = 2)
```

The end result is an ROC plot with a dashed reference line:



Qualitatively, we can see that this ROC curve appears to occupy the space at the top-left corner of the diagram, which suggests that it is closer to a perfect classifier than the dashed line representing a useless classifier. To confirm this quantitatively, we can use the ROCR package to calculate the AUC. To do so, we first need to create another `performance` object, this time specifying `measure = "auc"` as shown in the following code:

```
> perf.auc <- performance(pred, measure = "auc")
```

Since `perf.auc` is an R object (specifically known as an S4 object), we need to use a special type of notation to access the values stored within. S4 objects hold information in positions known as slots. The `str()` function can be used to see all of an object's slots:

```
> str(perf.auc)
Formal class 'performance' [package "ROCR"] with 6 slots
  ..@ x.name      : chr "None"
  ..@ y.name      : chr "Area under the ROC curve"
  ..@ alpha.name  : chr "none"
  ..@ x.values    : list()
  ..@ y.values    :List of 1
  .. ..$ : num 0.984
  ..@ alpha.values: list()
```

Notice that slots are prefixed with the @ symbol. To access the AUC value, which is stored as a list in the `y.values` slot, we can use the @ notation along with the `unlist()` function, which simplifies lists to a vector of numeric values:

```
> unlist(perf.auc@y.values)
[1] 0.9835862
```

The AUC for the SMS classifier is 0.98, which is extremely high. But how do we know whether the model is just as likely to perform well for another dataset? In order to answer such questions, we need to have a better understanding of how far we can extrapolate a model's predictions beyond the test data.

Estimating future performance

Some R machine learning packages present confusion matrices and performance measures during the model building process. The purpose of these statistics is to provide insight about the model's **resubstitution error**, which occurs when the training data is incorrectly predicted in spite of the model being built directly from this data. This information can be used as a rough diagnostic to identify obviously poor performers.

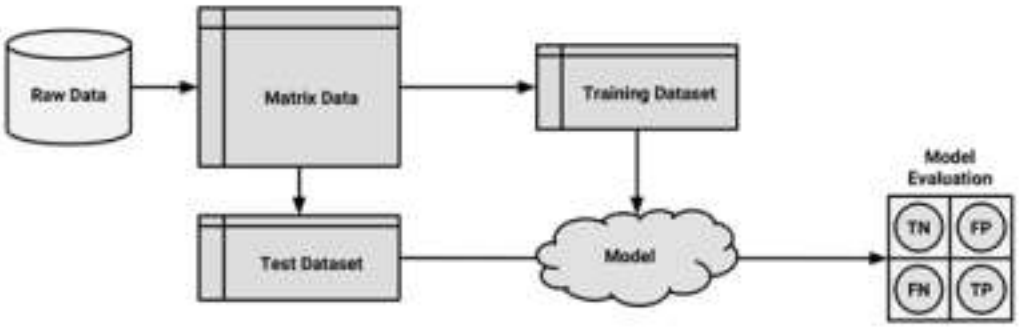
The resubstitution error is not a very useful marker of future performance. For example, a model that used rote memorization to perfectly classify every training instance with zero resubstitution error would be unable to generalize its predictions to data it has never seen before. For this reason, the error rate on the training data can be extremely optimistic about a model's future performance.

Instead of relying on resubstitution error, a better practice is to evaluate a model's performance on data it has not yet seen. We used this approach in previous chapters when we split the available data into a set for training and a set for testing. In some cases, however, it is not always ideal to create training and test datasets. For instance, in a situation where you have only a small pool of data, you might not want to reduce the sample any further.

Fortunately, there are other ways to estimate a model's performance on unseen data. The `caret` package we used to calculate performance measures also offers a number of functions to estimate future performance. If you are following the R code examples and haven't already installed the `caret` package, please do so. You will also need to load the package to the R session, using the `library(caret)` command.

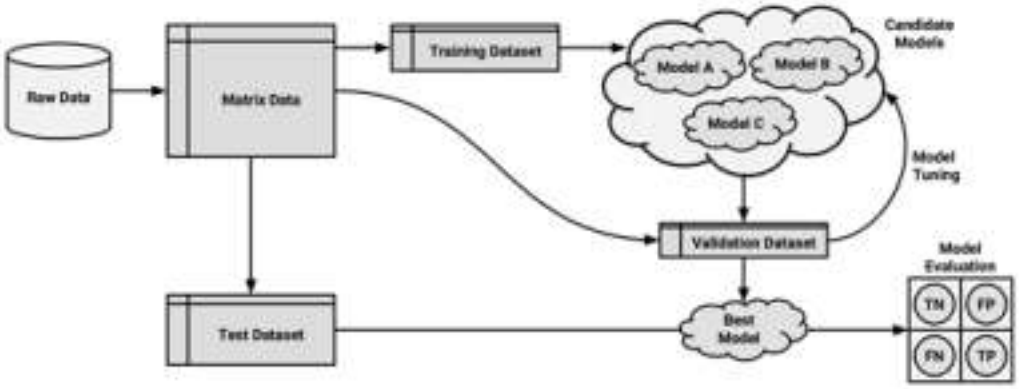
The holdout method

The procedure of partitioning data into training and test datasets that we used in previous chapters is known as the **holdout method**. As shown in the following diagram, the **training dataset** is used to generate the model, which is then applied to the **test dataset** to generate predictions for evaluation. Typically, about one-third of the data is held out for testing, and two-thirds is used for training, but this proportion can vary depending on the amount of available data. To ensure that the training and test data do not have systematic differences, their examples are randomly divided into the two groups.



For the holdout method to result in a truly accurate estimate of the future performance, at no time should the performance on the test dataset be allowed to influence the model. It is easy to unknowingly violate this rule by choosing the best model based upon the results of repeated testing. For example, suppose we built several models on the training data, and selected the one with the highest accuracy on the test data. Because we have cherry-picked the best result, the test performance is not an unbiased measure of the performance on unseen data.

To avoid this problem, it is better to divide the original data so that in addition to the training datasets and the test datasets, a **validation dataset** is available. The validation dataset would be used for iterating and refining the model or models chosen, leaving the test dataset to be used only once as a final step to report an estimated error rate for future predictions. A typical split between training, test, and validation would be 50 percent, 25 percent, and 25 percent, respectively.



A keen reader will note that holdout test data was used in the previous chapters to both evaluate models and improve model performance. This was done for illustrative purposes, but it would indeed violate the rule as stated previously. Consequently, the model performance statistics shown were not valid estimates of future performance on unseen data and the process could have been more accurately termed validation.

A simple method to create holdout samples uses random number generators to assign records to partitions. This technique was first used in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules* to create training and test datasets.

If you'd like to follow along with the following examples, download the `credit.csv` dataset from the Packt Publishing website, and load to a data frame using the `credit <- read.csv("credit.csv")` command.

Suppose we have a data frame named credit with 1000 rows of data. We can divide it into three partitions as follows. First, we create a vector of randomly ordered row IDs from 1 to 1000 using the `runif()` function, which by default generates a specified number of random values between 0 and 1. The `runif()` function gets its name from the random uniform distribution, which was discussed in *Chapter 2, Managing and Understanding Data*.

```
> random_ids <- order(runif(1000))
```

The `order()` used here returns a vector indicating the rank order of the 1,000 random numbers. For example, `order(c(0.5, 0.25, 0.75, 0.1))` returns the sequence `4 2 1 3` because the smallest number (0.1) appears fourth, the second smallest (0.25) appears second, and so on.

We can use the resulting random IDs to divide the `credit` data frame into 500, 250, and 250 records comprising the training, validation, and test datasets:

```
> credit_train <- credit[random_ids[1:500], ]
> credit_validate <- credit[random_ids[501:750], ]
> credit_test <- credit[random_ids[751:1000], ]
```

One problem with holdout sampling is that each partition may have a larger or smaller proportion of some classes. In certain cases, particularly those in which a class is a very small proportion of the dataset, this can lead a class to be omitted from the training dataset. This is a significant problem, because the model will not be able to learn this class.

In order to reduce the chance of this occurring, a technique called **stratified random sampling** can be used. Although in the long run a random sample should contain roughly the same proportion of each class value as the full dataset, stratified random sampling guarantees that the random partitions have nearly the same proportion of each class as the full dataset, even when some classes are small.

The `caret` package provides a `createDataPartition()` function that will create partitions based on stratified holdout sampling. The code to create a stratified sample of training and test data for the `credit` dataset is shown in the following commands. To use the function, a vector of the class values must be specified (here, `default` refers to whether a loan went into default) in addition to a parameter `p`, which specifies the proportion of instances to be included in the partition. The `list = FALSE` parameter prevents the result from being stored in the list format:

```
> in_train <- createDataPartition(credit$default, p = 0.75,
    list = FALSE)
> credit_train <- credit[in_train, ]
> credit_test <- credit[-in_train, ]
```

The `in_train` vector indicates row numbers included in the training sample. We can use these row numbers to select examples for the `credit_train` data frame. Similarly, by using a negative symbol, we can use the rows not found in the `in_train` vector for the `credit_test` dataset.

Although it distributes the classes evenly, stratified sampling does not guarantee other types of representativeness. Some samples may have too many or few difficult cases, easy-to-predict cases, or outliers. This is especially true for smaller datasets, which may not have a large enough portion of such cases to be divided among training and test sets.

In addition to potentially biased samples, another problem with the holdout method is that substantial portions of data must be reserved to test and validate the model. Since these data cannot be used to train the model until its performance has been measured, the performance estimates are likely to be overly conservative.

Since models trained on larger datasets generally perform better, a common practice is to retrain the model on the full set of data (that is, training plus test and validation) after a final model has been selected and evaluated.

A technique called **repeated holdout** is sometimes used to mitigate the problems of randomly composed training datasets. The repeated holdout method is a special case of the holdout method that uses the average result from several random holdout samples to evaluate a model's performance. As multiple holdout samples are used, it is less likely that the model is trained or tested on nonrepresentative data. We'll expand on this idea in the next section.

Cross-validation

The repeated holdout is the basis of a technique known as **k-fold cross-validation** (or **k-fold CV**), which has become the industry standard for estimating model performance. Rather than taking repeated random samples that could potentially use the same record more than once, k-fold CV randomly divides the data into *k* to completely separate random partitions called **folds**.

Although *k* can be set to any number, by far, the most common convention is to use **10-fold cross-validation** (10-fold CV). Why 10 folds? The reason is that the empirical evidence suggests that there is little added benefit in using a greater number. For each of the 10 folds (each comprising 10 percent of the total data), a machine learning model is built on the remaining 90 percent of data. The fold's matching 10 percent sample is then used for model evaluation. After the process of training and evaluating the model has occurred for 10 times (with 10 different training/testing combinations), the average performance across all the folds is reported.

An extreme case of k-fold CV is the **leave-one-out method**, which performs k-fold CV using a fold for each of the data's examples. This ensures that the greatest amount of data is used to train the model. Although this may seem useful, it is so computationally expensive that it is rarely used in practice.

Datasets for cross-validation can be created using the `createFolds()` function in the `caret` package. Similar to the stratified random holdout sampling, this function will attempt to maintain the same class balance in each of the folds as in the original dataset. The following is the command to create 10 folds:

```
> folds <- createFolds(credit$default, k = 10)
```

The result of the `createFolds()` function is a list of vectors storing the row numbers for each of the requested `k = 10` folds. We can peek at the contents, using `str()`:

```
> str(folds)
List of 10
 $ Fold01: int [1:100] 1 5 12 13 19 21 25 32 36 38 ...
 $ Fold02: int [1:100] 16 49 78 81 84 93 105 108 128 134 ...
```

```
$ Fold03: int [1:100] 15 48 60 67 76 91 102 109 117 123 ...
$ Fold04: int [1:100] 24 28 59 64 75 85 95 97 99 104 ...
$ Fold05: int [1:100] 9 10 23 27 29 34 37 39 53 61 ...
$ Fold06: int [1:100] 4 8 41 55 58 103 118 121 144 146 ...
$ Fold07: int [1:100] 2 3 7 11 14 33 40 45 51 57 ...
$ Fold08: int [1:100] 17 30 35 52 70 107 113 129 133 137 ...
$ Fold09: int [1:100] 6 20 26 31 42 44 46 63 79 101 ...
$ Fold10: int [1:100] 18 22 43 50 68 77 80 88 106 111 ...
```

Here, we see that the first fold is named `Fold01` and stores `100` integers, indicating the 100 rows in the credit data frame for the first fold. To create training and test datasets to build and evaluate a model, an additional step is needed. The following commands show how to create data for the first fold. We'll assign the selected 10 percent to the test dataset, and use the negative symbol to assign the remaining 90 percent to the training dataset:

```
> credit01_test <- credit[folds$Fold01, ]
> credit01_train <- credit[-folds$Fold01, ]
```

To perform the full 10-fold CV, this step would need to be repeated a total of 10 times; building a model and then calculating the model's performance each time. At the end, the performance measures would be averaged to obtain the overall performance. Thankfully, we can automate this task by applying several of the techniques we've learned earlier.

To demonstrate the process, we'll estimate the kappa statistic for a C5.0 decision tree model of the credit data using 10-fold CV. First, we need to load some R packages: `caret` (to create the folds), `C50` (for the decision tree), and `irr` (to calculate kappa). The latter two packages were chosen for illustrative purposes; if you desire, you can use a different model or a different performance measure along with the same series of steps.

```
> library(caret)
> library(C50)
> library(irr)
```

Next, we'll create a list of 10 folds as we have done previously. The `set.seed()` function is used here to ensure that the results are consistent if the same code is run again:

```
> set.seed(123)
> folds <- createFolds(credit$default, k = 10)
```

Finally, we will apply a series of identical steps to the list of folds using the `lapply()` function. As shown in the following code, because there is no existing function that does exactly what we need, we must define our own function to pass to `lapply()`. Our custom function divides the credit data frame into training and test data, builds a decision tree using the `C5.0()` function on the training data, generates a set of predictions from the test data, and compares the predicted and actual values using the `kappa2()` function:

```
> cv_results <- lapply(folds, function(x) {
    credit_train <- credit[-x, ]
    credit_test <- credit[x, ]
    credit_model <- C5.0(default ~ ., data = credit_train)
    credit_pred <- predict(credit_model, credit_test)
    credit_actual <- credit_test$default
    kappa <- kappa2(data.frame(credit_actual, credit_pred))$value
    return(kappa)
  })
```

The resulting kappa statistics are compiled into a list stored in the `cv_results` object, which we can examine using `str()`:

```
> str(cv_results)
List of 10
 $ Fold01: num 0.343
 $ Fold02: num 0.255
 $ Fold03: num 0.109
 $ Fold04: num 0.107
 $ Fold05: num 0.338
 $ Fold06: num 0.474
 $ Fold07: num 0.245
 $ Fold08: num 0.0365
 $ Fold09: num 0.425
 $ Fold10: num 0.505
```

There's just one more step remaining in the 10-fold CV process: we must calculate the average of these 10 values. Although you will be tempted to type `mean(cv_results)`, because `cv_results` is not a numeric vector, the result would be an error. Instead, use the `unlist()` function, which eliminates the list structure, and reduces `cv_results` to a numeric vector. From here, we can calculate the mean kappa as expected:

```
> mean(unlist(cv_results))
[1] 0.283796
```

This kappa statistic is fairly low, corresponding to "fair" on the interpretation scale, which suggests that the credit scoring model performs only marginally better than random chance. In the next chapter, we'll examine automated methods based on 10-fold CV that can assist us in improving the performance of this model.

Perhaps the current gold standard method to reliably estimate model performance is **repeated k-fold CV**. As you might guess from the name, this involves repeatedly applying k-fold CV and averaging the results. A common strategy is to perform 10-fold CV ten times. Although it is computationally intensive, it provides a very robust estimate.

Bootstrap sampling

A slightly less frequently used alternative to k-fold CV is known as **bootstrap sampling**, the **bootstrap** or **bootstrapping** for short. Generally speaking, these refer to the statistical methods of using random samples of data to estimate the properties of a larger set. When this principle is applied to machine learning model performance, it implies the creation of several randomly selected training and test datasets, which are then used to estimate performance statistics. The results from the various random datasets are then averaged to obtain a final estimate of future performance.

So, what makes this procedure different from k-fold CV? Whereas cross-validation divides the data into separate partitions in which each example can appear only once, the bootstrap allows examples to be selected multiple times through a process of **sampling with replacement**. This means that from the original dataset of *n* examples, the bootstrap procedure will create one or more new training datasets that will also contain *n* examples, some of which are repeated. The corresponding test datasets are then constructed from the set of examples that were not selected for the respective training datasets.

Using sampling with replacement as described previously, the probability that any given instance is included in the training dataset is 63.2 percent. Consequently, the probability of any instance being in the test dataset is 36.8 percent. In other words, the training data represents only 63.2 percent of available examples, some of which are repeated. In contrast to 10-fold CV, which uses 90 percent of the examples for training, the bootstrap sample is less representative of the full dataset.

Because a model trained on only 63.2 percent of the training data is likely to perform worse than a model trained on a larger training set, the bootstrap's performance estimates may be substantially lower than what would be obtained when the model is later trained on the full dataset. A special case of bootstrapping known as the **0.632 bootstrap** accounts for this by calculating the final performance measure as a function of performance on both the training data (which is overly optimistic) and the test data (which is overly pessimistic). The final error rate is then estimated as:

$$\text{error} = 0.632 \times \text{error}_{\text{test}} + 0.368 \times \text{error}_{\text{train}}$$

One advantage of bootstrap over cross-validation is that it tends to work better with very small datasets. Additionally, bootstrap sampling has applications beyond performance measurement. In particular, in the next chapter we'll learn how the principles of bootstrap sampling can be used to improve model performance.

Summary

This chapter presented a number of the most common measures and techniques for evaluating the performance of machine learning classification models. Although accuracy provides a simple method to examine how often a model is correct, this can be misleading in the case of rare events because the real-life cost of such events may be inversely proportional to how frequently they appear.

A number of measures based on confusion matrices better capture the balance among the costs of various types of errors. Closely examining the tradeoffs between sensitivity and specificity, or precision and recall can be a useful tool for thinking about the implications of errors in the real world. Visualizations such as the ROC curve are also helpful to this end.

It is also worth mentioning that sometimes the best measure of a model's performance is to consider how well it meets, or doesn't meet, other objectives. For instance, you may need to explain a model's logic in simple language, which would eliminate some models from consideration. Additionally, even if it performs very well, a model that is too slow or difficult to scale to a production environment is completely useless.

An obvious extension of measuring performance is to identify automated ways to find the best models for a particular task. In the next chapter, we will build upon our work so far to investigate ways to make smarter models by systematically iterating, refining, and combining learning algorithms.

11
Improving Model Performance

When a sports team falls short of meeting its goal—whether the goal is to obtain an Olympic gold medal, a league championship, or a world record time—it must search for possible improvements. Imagine that you're the team's coach. How would you spend your practice sessions? Perhaps you'd direct the athletes to train harder or train differently in order to maximize every bit of their potential. Or, you might emphasize better teamwork, utilizing the athletes' strengths and weaknesses more smartly.

Now imagine that you're training a world champion machine learning algorithm. Perhaps you hope to compete in data mining competitions such as those posted on Kaggle (`http://www.kaggle.com/competitions`). Maybe you simply need to improve business results. Where do you begin? Although the context differs, the strategies one uses to improve sports team performance can also be used to improve the performance of statistical learners.

As the coach, it is your job to find the combination of training techniques and teamwork skills that allow you to meet your performance goals. This chapter builds upon the material covered throughout this book to introduce a set of techniques for improving the predictive performance of machine learners. You will learn:

- How to automate model performance tuning by systematically searching for the optimal set of training conditions
- The methods for combining models into groups that use teamwork to tackle tough learning tasks
- How to apply a variant of decision trees, which has quickly become popular due to its impressive performance

None of these methods will be successful for every problem. Yet, looking at the winning entries to machine learning competitions, you'll likely find that at least one of them has been employed. To be competitive, you too will need to add these skills to your repertoire.

Tuning stock models for better performance

Some learning problems are well-suited to the stock models presented in the previous chapters. In such cases, it may not be necessary to spend much time iterating and refining the model; it may perform well enough as it is. On the other hand, some problems are inherently more difficult. The underlying concepts to be learned may be extremely complex, requiring an understanding of many subtle relationships, or it may be affected by random variation, making it difficult to define the signal within the noise.

Developing models that perform extremely well on difficult problems is every bit an art as it is a science. Sometimes a bit of intuition is helpful when trying to identify areas where performance can be improved. In other cases, finding improvements will require a brute-force, trial and error approach. Of course, the process of searching numerous possible improvements can be aided by the use of automated programs.

In *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules,* we attempted a difficult problem: identifying loans that were likely to enter into default. Although we were able to use performance tuning methods to obtain a respectable classification accuracy of about 82 percent, upon a more careful examination in *Chapter 10, Evaluating Model Performance,* we realized that the high accuracy was a bit misleading. In spite of the reasonable accuracy, the kappa statistic was only about 0.28, which suggested that the model was actually performing somewhat poorly. In this section, we'll revisit the credit scoring model to see whether we can improve the results.

To follow along with the examples, download the `credit.csv` file from the Packt Publishing website and save it to your R working directory. Load the file into R using the command `credit <- read.csv("credit.csv")`.

You will recall that we first used a stock C5.0 decision tree to build the classifier for the credit data. We then attempted to improve its performance by adjusting the `trials` parameter to increase the number of boosting iterations. By increasing the number of iterations from the default of 1 up to the value of 10, we were able to increase the model's accuracy. This process of adjusting the model options to identify the best fit is called **parameter tuning**.

Parameter tuning is not limited to decision trees. For instance, we tuned k-NN models when we searched for the best value of *k*. We also tuned neural networks and support vector machines as we adjusted the number of nodes or hidden layers, or chose different kernel functions. Most machine learning algorithms allow the adjustment of at least one parameter, and the most sophisticated models offer a large number of ways to tweak the model fit. Although this allows the model to be tailored closely to the learning task, the complexity of all the possible options can be daunting. A more systematic approach is warranted.

Using caret for automated parameter tuning

Rather than choosing arbitrary values for each of the model's parameters—a task that is not only tedious, but also somewhat unscientific—it is better to conduct a search through many possible parameter values to find the best combination.

The `caret` package, which we used extensively in *Chapter 10, Evaluating Model Performance*, provides tools to assist with automated parameter tuning. The core functionality is provided by a `train()` function that serves as a standardized interface for over 175 different machine learning models for both classification and regression tasks. By using this function, it is possible to automate the search for optimal models using a choice of evaluation methods and metrics.

Do not feel overwhelmed by the large number of models—we've already covered many of them in the earlier chapters. Others are simple variants or extensions of the base concepts. Given what you've learned so far, you should be confident that you have the ability to understand all of the available methods.

Automated parameter tuning requires you to consider three questions:

- What type of machine learning model (and specific implementation) should be trained on the data?
- Which model parameters can be adjusted, and how extensively should they be tuned to find the optimal settings?
- What criteria should be used to evaluate the models to find the best candidate?

Answering the first question involves finding a well-suited match between the machine learning task and one of the 175 models. Obviously, this requires an understanding of the breadth and depth of machine learning models. It can also help to work through a process of elimination. Nearly half of the models can be eliminated depending on whether the task is classification or numeric prediction; others can be excluded based on the format of the data or the need to avoid black box models, and so on. In any case, there's also no reason you can't try several approaches and compare the best results of each.

Addressing the second question is a matter largely dictated by the choice of model, since each algorithm utilizes a unique set of parameters. The available tuning parameters for the predictive models covered in this book are listed in the following table. Keep in mind that although some models have additional options not shown, only those listed in the table are supported by `caret` for automatic tuning.

Model	Learning Task	Method name	Parameters
k-Nearest Neighbors	Classification	`knn`	`k`
Naive Bayes	Classification	`nb`	`fL`, `usekernel`
Decision Trees	Classification	`C5.0`	`model`, `trials`, `winnow`
OneR Rule Learner	Classification	`OneR`	None
RIPPER Rule Learner	Classification	`JRip`	`NumOpt`
Linear Regression	Regression	`lm`	None
Regression Trees	Regression	`rpart`	`cp`
Model Trees	Regression	`M5`	`pruned`, `smoothed`, `rules`
Neural Networks	Dual use	`nnet`	`size`, `decay`
Support Vector Machines (Linear Kernel)	Dual use	`svmLinear`	`C`
Support Vector Machines (Radial Basis Kernel)	Dual use	`svmRadial`	`C`, `sigma`
Random Forests	Dual use	`rf`	`mtry`

For a complete list of the models and corresponding tuning parameters covered by `caret`, refer to the table provided by package author Max Kuhn at `http://topepo.github.io/caret/modelList.html`.

If you ever forget the tuning parameters for a particular model, the `modelLookup()` function can be used to find them. Simply supply the method name, as illustrated here for the C5.0 model:

```
> modelLookup("C5.0")
  model parameter                  label forReg forClass probModel
1  C5.0    trials # Boosting Iterations  FALSE     TRUE      TRUE
2  C5.0     model             Model Type  FALSE     TRUE      TRUE
3  C5.0   winnow                 Winnow  FALSE     TRUE      TRUE
```

The goal of automatic tuning is to search a set of candidate models comprising a matrix, or **grid**, of parameter combinations. Because it is impractical to search every conceivable combination, only a subset of possibilities is used to construct the grid. By default, `caret` searches at most three values for each of the *p* parameters. This means that at most *3^p* candidate models will be tested. For example, by default, the automatic tuning of k-Nearest Neighbors will compare *3^1 = 3* candidate models with `k=5`, `k=7`, and `k=9`. Similarly, tuning a decision tree will result in a comparison of up to 27 different candidate models, comprising the grid of *3^3 = 27* combinations of `model`, `trials`, and `winnow` settings. In practice, however, only 12 models are actually tested. This is because the `model` and `winnow` parameters can only take two values (`tree` versus `rules` and `TRUE` versus `FALSE`, respectively), which makes the grid size *3 * 2 * 2 = 12*.

Since the default search grid may not be ideal for your learning problem, `caret` allows you to provide a custom search grid defined by a simple command, which we will cover later.

The third and final step in automatic model tuning involves identifying the best model among the candidates. This uses the methods discussed in *Chapter 10, Evaluating Model Performance*, such as the choice of resampling strategy for creating training and test datasets and the use of model performance statistics to measure the predictive accuracy.

All of the resampling strategies and many of the performance statistics we've learned are supported by `caret`. These include statistics such as accuracy and kappa (for classifiers) and R-squared or RMSE (for numeric models). Cost-sensitive measures such as sensitivity, specificity, and area under the ROC curve (AUC) can also be used, if desired.

By default, `caret` will select the candidate model with the largest value of the desired performance measure. As this practice sometimes results in the selection of models that achieve marginal performance improvements via large increases in model complexity, alternative model selection functions are provided.

Given the wide variety of options, it is helpful that many of the defaults are reasonable. For instance, `caret` will use prediction accuracy on a bootstrap sample to choose the best performer for classification models. Beginning with these default values, we can then tweak the `train()` function to design a wide variety of experiments.

Creating a simple tuned model

To illustrate the process of tuning a model, let's begin by observing what happens when we attempt to tune the credit scoring model using the `caret` package's default settings. From there, we will adjust the options to our liking.

The simplest way to tune a learner requires you to only specify a model type via the `method` parameter. Since we used C5.0 decision trees previously with the credit model, we'll continue our work by optimizing this learner. The basic `train()` command for tuning a C5.0 decision tree using the default settings is as follows:

```
> library(caret)
> set.seed(300)
> m <- train(default ~ ., data = credit, method = "C5.0")
```

First, the `set.seed()` function is used to initialize R's random number generator to a set starting position. You may recall that we used this function in several prior chapters. By setting the `seed` parameter (in this case to the arbitrary number 300), the random numbers will follow a predefined sequence. This allows simulations that use random sampling to be repeated with identical results—a very helpful feature if you are sharing code or attempting to replicate a prior result.

Next, we define a tree as `default ~ .` using the R formula interface. This models loan default status (`yes` or `no`) using all of the other features in the `credit` data frame. The parameter `method = "C5.0"` tells `caret` to use the C5.0 decision tree algorithm.

After you've entered the preceding command, there may be a significant delay (dependent upon your computer's capabilities) as the tuning process occurs. Even though this is a fairly small dataset, a substantial amount of calculation must occur. R must repeatedly generate random samples of data, build decision trees, compute performance statistics, and evaluate the result.

The result of the experiment is saved in an object named `m`. If you would like to examine the object's contents, the `str(m)` command will list all the associated data, but this can be quite overwhelming. Instead, simply type the name of the object for a condensed summary of the results. For instance, typing `m` yields the following output (note that labels have been added for clarity):

```
1000 samples
  16 predictor
   2 classes: 'no', 'yes'
```

```
No pre-processing
Resampling: Bootstrapped (25 reps)

Summary of sample sizes: 1000, 1000, 1000, 1000, 1000, 1000, ...
```

```
Resampling results across tuning parameters:

  model  winnow  trials  Accuracy   Kappa      Accuracy SD  Kappa SD
  rules  FALSE    1      0.6847204  0.2578421  0.02558775   0.05622302
  rules  FALSE   10      0.7112829  0.3094601  0.02087257   0.04585890
  rules  FALSE   20      0.7221976  0.3260145  0.01977334   0.04512083
  rules   TRUE    1      0.6888432  0.2549192  0.02683844   0.05695277
  rules   TRUE   10      0.7113716  0.3038075  0.01947701   0.04484956
  rules   TRUE   20      0.7233222  0.3266866  0.01843672   0.03714053
  tree   FALSE    1      0.6769653  0.2285102  0.03027647   0.07001131
  tree   FALSE   10      0.7222552  0.2880662  0.02061900   0.05601918
  tree   FALSE   20      0.7297858  0.3067404  0.02087556   0.05616826
  tree    TRUE    1      0.6771020  0.2219533  0.02703456   0.05955907
  tree    TRUE   10      0.7173312  0.2777136  0.01700633   0.04358591
  tree    TRUE   20      0.7285714  0.3058474  0.01497973   0.04145128
```

```
Accuracy was used to select the optimal model using  the largest value.
The final values used for the model were trials = 20, model = tree
and winnow = FALSE.
```

The labels highlight four main components in the output:

1. **A brief description of the input dataset**: If you are familiar with your data and have applied the `train()` function correctly, this information should not be surprising.
2. **A report of the preprocessing and resampling methods applied**: Here, we see that 25 bootstrap samples, each including 1,000 examples, were used to train the models.
3. **A list of the candidate models evaluated**: In this section, we can confirm that 12 different models were tested, based on the combinations of three C5.0 tuning parameters—`model`, `trials`, and `winnow`. The average and standard deviation of the accuracy and kappa statistics for each candidate model are also shown.
4. **The choice of the best model**: As the footnote describes, the model with the largest accuracy was selected. This was the model that used a decision tree with 20 trials and the setting `winnow = FALSE`.

After identifying the best model, the `train()` function uses its tuning parameters to build a model on the full input dataset, which is stored in the `m` list object as `m$finalModel`. In most cases, you will not need to work directly with the `finalModel` sub-object. Instead, simply use the `predict()` function with the `m` object as follows:

```
> p <- predict(m, credit)
```

The resulting vector of predictions works as expected, allowing us to create a confusion matrix that compares the predicted and actual values:

```
> table(p, credit$default)

p       no yes
  no   700   2
  yes    0 298
```

Of the 1,000 examples used for training the final model, only two were misclassified. However, it is very important to note that since the model was built on both the training and test data, this accuracy is optimistic and thus, should not be viewed as indicative of performance on unseen data. The bootstrap estimate of 73 percent (shown in the summary output) is a more realistic estimate of future performance.

Using the `train()` and `predict()` functions also offers a couple of benefits in addition to the automatic parameter tuning.

First, any data preparation steps applied by the `train()` function will be similarly applied to the data used for generating predictions. This includes transformations such as centering and scaling as well as imputation of missing values. Allowing `caret` to handle the data preparation will ensure that the steps that contributed to the best model's performance will remain in place when the model is deployed.

Second, the `predict()` function provides a standardized interface for obtaining predicted class values and class probabilities, even for model types that ordinarily would require additional steps to obtain this information. The predicted classes are provided by default:

```
> head(predict(m, credit))
[1] no  yes no  no  yes no
Levels: no yes
```

To obtain the estimated probabilities for each class, use the `type = "prob"` parameter:

```
> head(predict(m, credit, type = "prob"))
         no        yes
1 0.9606970 0.03930299
2 0.1388444 0.86115561
3 1.0000000 0.00000000
4 0.7720279 0.22797208
5 0.2948062 0.70519385
6 0.8583715 0.14162851
```

Even in cases where the underlying model refers to the prediction probabilities using a different string (for example, `"raw"` for a `naiveBayes` model), the `predict()` function will translate `type = "prob"` to the appropriate string behind the scenes.

Customizing the tuning process

The decision tree we created previously demonstrates the `caret` package's ability to produce an optimized model with minimal intervention. The default settings allow optimized models to be created easily. However, it is also possible to change the default settings to something more specific to a learning task, which may assist with unlocking the upper echelon of performance.

Each step in the model selection process can be customized. To illustrate this flexibility, let's modify our work on the credit decision tree to mirror the process we had used in *Chapter 10, Evaluating Model Performance*. If you remember, we had estimated the kappa statistic using 10-fold cross-validation. We'll do the same here, using kappa to optimize the boosting parameter of the decision tree. Note that decision tree boosting was previously covered in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, and will also be covered in greater detail later this chapter.

The `trainControl()` function is used to create a set of configuration options known as a **control object**, which guides the `train()` function. These options allow for the management of model evaluation criteria such as the resampling strategy and the measure used for choosing the best model. Although this function can be used to modify nearly every aspect of a tuning experiment, we'll focus on the two important parameters: `method` and `selectionFunction`.

If you're eager for more details, you can use the `?trainControl` command for a list of all the parameters.

For the `trainControl()` function, the `method` parameter is used to set the resampling method, such as holdout sampling or k-fold cross-validation. The following table lists the possible method types as well as any additional parameters for adjusting the sample size and number of iterations. Although the default options for these resampling methods follow popular convention, you may choose to adjust these depending upon the size of your dataset and the complexity of your model.

Resampling method	Method name	Additional options and default values
Holdout sampling	`LGOCV`	`p = 0.75` (training data proportion)
k-fold cross-validation	`cv`	`number = 10` (number of folds)
Repeated k-fold cross-validation	`repeatedcv`	`number = 10` (number of folds) `repeats = 10` (number of iterations)
Bootstrap sampling	`boot`	`number = 25` (resampling iterations)
0.632 bootstrap	`boot632`	`number = 25` (resampling iterations)
Leave-one-out cross-validation	`LOOCV`	None

The `selectionFunction` parameter is used to specify the function that will choose the optimal model among the various candidates. Three such functions are included. The `best` function simply chooses the candidate with the best value on the specified performance measure. This is used by default. The other two functions are used to choose the most parsimonious, or simplest, model that is within a certain threshold of the best model's performance. The `oneSE` function chooses the simplest candidate within one standard error of the best performance, and `tolerance` uses the simplest candidate within a user-specified percentage.

Some subjectivity is involved with the `caret` package's ranking of models by simplicity. For information on how models are ranked, see the help page for the selection functions by typing `?best` at the R command prompt.

To create a control object named `ctrl` that uses 10-fold cross-validation and the `oneSE` selection function, use the following command (note that `number = 10` is included only for clarity; since this is the default value for `method = "cv"`, it could have been omitted):

```
> ctrl <- trainControl(method = "cv", number = 10,
                       selectionFunction = "oneSE")
```

We'll use the result of this function shortly.

In the meantime, the next step in defining our experiment is to create the grid of parameters to optimize. The grid must include a column named for each parameter in the desired model, prefixed by a period. It must also include a row for each desired combination of parameter values. Since we are using a C5.0 decision tree, this means we'll need columns named `.model`, `.trials`, and `.winnow`. For other machine learning models, refer to the table presented earlier in this chapter or use the `modelLookup()` function to lookup the parameters as described previously.

Rather than filling this data frame cell by cell—a tedious task if there are many possible combinations of parameter values—we can use the `expand.grid()` function, which creates data frames from the combinations of all the values supplied. For example, suppose we would like to hold constant `model = "tree"` and `winnow = "FALSE"` while searching eight different values of trials. This can be created as:

```
> grid <- expand.grid(.model = "tree",
                      .trials = c(1, 5, 10, 15, 20, 25, 30, 35),
                      .winnow = "FALSE")
```

The resulting grid data frame contains *1 * 8 * 1 = 8* rows:

```
> grid
  .model .trials .winnow
1   tree       1   FALSE
2   tree       5   FALSE
3   tree      10   FALSE
4   tree      15   FALSE
5   tree      20   FALSE
6   tree      25   FALSE
7   tree      30   FALSE
8   tree      35   FALSE
```

The `train()` function will build a candidate model for evaluation using each row's combination of model parameters.

Given this search grid and the control list created previously, we are ready to run a thoroughly customized `train()` experiment. As we did earlier, we'll set the random seed to the arbitrary number `300` in order to ensure repeatable results. But this time, we'll pass our control object and tuning grid while adding a parameter `metric = "Kappa"`, indicating the statistic to be used by the model evaluation function—in this case, `"oneSE"`. The full command is as follows:

```
> set.seed(300)
> m <- train(default ~ ., data = credit, method = "C5.0",
```

```
                metric = "Kappa",
                trControl = ctrl,
                tuneGrid = grid)
```

This results in an object that we can view by typing its name:

```
> m

1000 samples
  16 predictor
   2 classes: 'no', 'yes'

No pre-processing
Resampling: Cross-Validated (10 fold)

Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...

Resampling results across tuning parameters:

  trials  Accuracy  Kappa      Accuracy SD  Kappa SD
   1      0.724     0.3124461  0.02547330   0.05897140
   5      0.713     0.2921760  0.02110819   0.06018851
  10      0.719     0.2947271  0.03107339   0.06719720
  15      0.721     0.3009258  0.01969207   0.05105480
  20      0.717     0.2929875  0.02790858   0.07912362
  25      0.728     0.3150336  0.03224903   0.09367152
  30      0.729     0.3104144  0.02766867   0.08069045
  35      0.741     0.3389908  0.03142893   0.09352673

Tuning parameter 'model' was held constant at a value of tree
Tuning parameter 'winnow' was held constant at a value of FALSE
Kappa was used to select the optimal model using  the one SE rule.
The final values used for the model were trials = 1, model = tree
and winnow = FALSE.
```

Although much of the output is similar to the automatically tuned model, there are a few differences of note. As 10-fold cross-validation was used, the sample size to build each candidate model was reduced to 900 rather than the 1,000 used in the bootstrap. As we requested, eight candidate models were tested. Additionally, because `model` and `winnow` were held constant, their values are no longer shown in the results; instead, they are listed as a footnote.

The best model here differs quite significantly from the prior trial. Before, the best model used `trials = 20`, whereas here, it used `trials = 1`. This seemingly odd finding is due to the fact that we used the `oneSE` rule rather the `best` rule to select the optimal model. Even though the 35-trial model offers the best raw performance according to kappa, the 1-trial model offers nearly the same performance with a much simpler form. Not only are simple models more computationally efficient, but they also reduce the chance of overfitting the training data.

Improving model performance with meta-learning

As an alternative to increasing the performance of a single model, it is possible to combine several models to form a powerful team. Just as the best sports teams have players with complementary rather than overlapping skillsets, some of the best machine learning algorithms utilize teams of complementary models. Since a model brings a unique bias to a learning task, it may readily learn one subset of examples, but have trouble with another. Therefore, by intelligently using the talents of several diverse team members, it is possible to create a strong team of multiple weak learners.

This technique of combining and managing the predictions of multiple models falls into a wider set of **meta-learning** methods defining techniques that involve learning how to learn. This includes anything from simple algorithms that gradually improve performance by iterating over design decisions—for instance, the automated parameter tuning used earlier in this chapter—to highly complex algorithms that use concepts borrowed from evolutionary biology and genetics for self-modifying and adapting to learning tasks.

For the remainder of this chapter, we'll focus on meta-learning only as it pertains to modeling a relationship between the predictions of several models and the desired outcome. The teamwork-based techniques covered here are quite powerful, and are used quite often to build more effective classifiers.

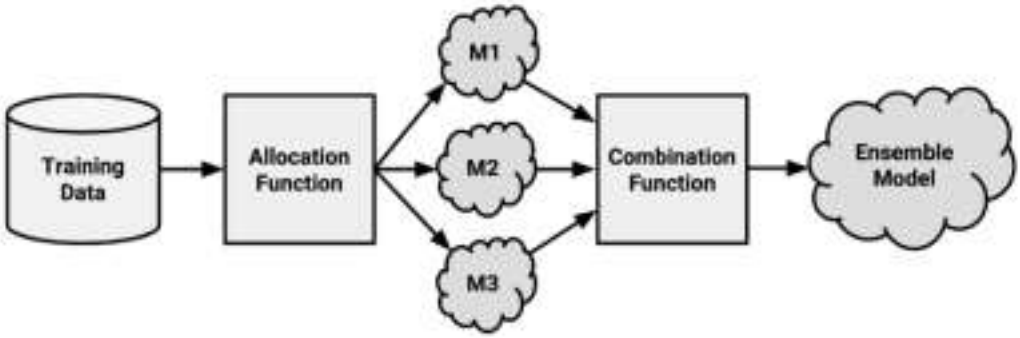
Understanding ensembles

Suppose you were a contestant on a television trivia show that allowed you to choose a panel of five friends to assist you with answering the final question for the million-dollar prize. Most people would try to stack the panel with a diverse set of subject matter experts. A panel containing professors of literature, science, history, and art, along with a current pop-culture expert would be a safely well-rounded group. Given their breadth of knowledge, it would be unlikely to find a question that stumps the group.

The meta-learning approach that utilizes a similar principle of creating a varied team of experts is known as an **ensemble**. All the ensemble methods are based on the idea that by combining multiple weaker learners, a stronger learner is created. The various ensemble methods can be distinguished, in large part, by the answers to these two questions:

- How are the weak learning models chosen and/or constructed?
- How are the weak learners' predictions combined to make a single final prediction?

When answering these questions, it can be helpful to imagine the ensemble in terms of the following process diagram; nearly all ensemble approaches follow this pattern:

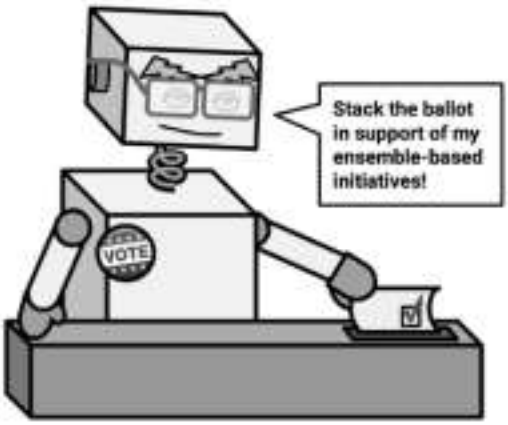


First, input training data is used to build a number of models. The **allocation function** dictates how much of the training data each model receives. Do they each receive the full training dataset or merely a sample? Do they each receive every feature or a subset?

Although the ideal ensemble includes a diverse set of models, the allocation function can increase diversity by artificially varying the input data to bias the resulting learners, even if they are the same type. For instance, it might use bootstrap sampling to construct unique training datasets or pass on a different subset of features or examples to each model. On the other hand, if the ensemble already includes a diverse set of algorithms—such as a neural network, a decision tree, and a k-NN classifier—the allocation function might pass the data on to each algorithm relatively unchanged.

After the models are constructed, they can be used to generate a set of predictions, which must be managed in some way. The **combination function** governs how disagreements among the predictions are reconciled. For example, the ensemble might use a majority vote to determine the final prediction, or it could use a more complex strategy such as weighting each model's votes based on its prior performance.

Some ensembles even utilize another model to learn a combination function from various combinations of predictions. For example, suppose that when *M1* and *M2* both vote yes, the actual class value is usually no. In this case, the ensemble could learn to ignore the vote of *M1* and *M2* when they agree. This process of using the predictions of several models to train a final arbiter model is known as **stacking**.



One of the benefits of using ensembles is that they may allow you to spend less time in pursuit of a single best model. Instead, you can train a number of reasonably strong candidates and combine them. Yet, convenience isn't the only reason why ensemble-based methods continue to rack up wins in machine learning competitions; ensembles also offer a number of performance advantages over single models:

- **Better generalizability to future problems**: As the opinions of several learners are incorporated into a single final prediction, no single bias is able to dominate. This reduces the chance of overfitting to a learning task.
- **Improved performance on massive or miniscule datasets**: Many models run into memory or complexity limits when an extremely large set of features or examples are used, making it more efficient to train several small models than a single full model. Conversely, ensembles also do well on the smallest datasets because resampling methods such as bootstrapping are inherently a part of many ensemble designs. Perhaps most importantly, it is often possible to train an ensemble in parallel using distributed computing methods.

- **The ability to synthesize data from distinct domains**: Since there is no one-size-fits-all learning algorithm, the ensemble's ability to incorporate evidence from multiple types of learners is increasingly important as complex phenomena rely on data drawn from diverse domains.
- **A more nuanced understanding of difficult learning tasks**: Real-world phenomena are often extremely complex with many interacting intricacies. Models that divide the task into smaller portions are likely to more accurately capture subtle patterns that a single global model might miss.

None of these benefits would be very helpful if you weren't able to easily apply ensemble methods in R, and there are many packages available to do just that. Let's take a look at several of the most popular ensemble methods and how they can be used to improve the performance of the credit model we've been working on.

Bagging

One of the first ensemble methods to gain widespread acceptance used a technique called **bootstrap aggregating** or **bagging** for short. As described by Leo Breiman in 1994, bagging generates a number of training datasets by bootstrap sampling the original training data. These datasets are then used to generate a set of models using a single learning algorithm. The models' predictions are combined using voting (for classification) or averaging (for numeric prediction).

For additional information on bagging, refer to Breiman L. *Bagging predictors*. Machine Learning. 1996; 24:123-140.

Although bagging is a relatively simple ensemble, it can perform quite well as long as it is used with relatively **unstable** learners, that is, those generating models that tend to change substantially when the input data changes only slightly. Unstable models are essential in order to ensure the ensemble's diversity in spite of only minor variations between the bootstrap training datasets. For this reason, bagging is often used with decision trees, which have the tendency to vary dramatically given minor changes in the input data.

The `ipred` package offers a classic implementation of bagged decision trees. To train the model, the `bagging()` function works similar to many of the models used previously. The `nbagg` parameter is used to control the number of decision trees voting in the ensemble (with a default value of `25`). Depending on the difficulty of the learning task and the amount of training data, increasing this number may improve the model's performance up to a limit. The downside is that this comes at the expense of additional computational expense because a large number of trees may take some time to train.

After installing the `ipred` package, we can create the ensemble as follows. We'll stick to the default value of 25 decision trees:

```
> library(ipred)
> set.seed(300)
> mybag <- bagging(default ~ ., data = credit, nbagg = 25)
```

The resulting model works as expected with the `predict()` function:

```
> credit_pred <- predict(mybag, credit)
> table(credit_pred, credit$default)

credit_pred  no yes
        no  699   2
        yes   1 298
```

Given the preceding results, the model seems to have fit the training data extremely well. To see how this translates into future performance, we can use the bagged trees with 10-fold CV using the `train()` function in the `caret` package. Note that the method name for the `ipred` bagged trees function is `treebag`:

```
> library(caret)
> set.seed(300)
> ctrl <- trainControl(method = "cv", number = 10)
> train(default ~ ., data = credit, method = "treebag",
        trControl = ctrl)

Bagged CART

1000 samples
  16 predictor
   2 classes: 'no', 'yes'

No pre-processing
Resampling: Cross-Validated (10 fold)

Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...

Resampling results

  Accuracy  Kappa      Accuracy SD  Kappa SD
  0.735     0.3297726  0.03439961   0.08590462
```

The kappa statistic of 0.33 for this model suggests that the bagged tree model performs at least as well as the best C5.0 decision tree we tuned earlier in this chapter. This illustrates the power of ensemble methods; a set of simple learners working together can outperform very sophisticated models.

To get beyond bags of decision trees, the `caret` package also provides a more general `bag()` function. It includes native support for a handful of models, though it can be adapted to other types with a bit of additional effort. The `bag()` function uses a control object to configure the bagging process. It requires the specification of three functions: one for fitting the model, one for making predictions, and one for aggregating the votes.

For example, suppose we wanted to create a bagged support vector machine model, using the `ksvm()` function in the `kernlab` package we used in *Chapter 7, Black Box Methods – Neural Networks and Support Vector Machines*. The `bag()` function requires us to provide functionality for training the SVMs, making predictions, and counting votes.

Rather than writing these ourselves, the `caret` package's built-in `svmBag` list object supplies three functions we can use for this purpose:

```
> str(svmBag)
List of 3
 $ fit      :function (x, y, ...)
 $ pred     :function (object, x)
 $ aggregate:function (x, type = "class")
```

By looking at the `svmBag$fit` function, we see that it simply calls the `ksvm()` function from the `kernlab` package and returns the result:

```
> svmBag$fit
function (x, y, ...)
{
    library(kernlab)
    out <- ksvm(as.matrix(x), y, prob.model = is.factor(y), ...)
    out
}
<environment: namespace:caret>
```

The `pred` and `aggregate` functions for `svmBag` are also similarly straightforward. By studying these functions and creating your own in the same format, it is possible to use bagging with any machine learning algorithm you would like.

The `caret` package also includes example objects for bags of naive Bayes models (`nbBag`), decision trees (`ctreeBag`), and neural networks (`nnetBag`).

Applying the three functions in the `svmBag` list, we can create a bagging control object:

```
> bagctrl <- bagControl(fit = svmBag$fit,
                        predict = svmBag$pred,
                        aggregate = svmBag$aggregate)
```

By using this with the `train()` function and the training control object (`ctrl`), defined earlier, we can evaluate the bagged SVM model as follows (note that the `kernlab` package is required for this to work; you will need to install it if you have not done so previously):

```
> set.seed(300)
> svmbag <- train(default ~ ., data = credit, "bag",
                  trControl = ctrl, bagControl = bagctrl)
> svmbag

Bagged Model
1000 samples
  16 predictors
   2 classes: 'no', 'yes'

No pre-processing
Resampling: Cross-Validation (10 fold)

Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...

Resampling results

  Accuracy  Kappa      Accuracy SD  Kappa SD
  0.728     0.2929505  0.04442222   0.1318101

Tuning parameter 'vars' was held constant at a value of 35
```

Given that the kappa statistic is below 0.30, it seems that the bagged SVM model performs worse than the bagged decision tree model. It's worth pointing out that the standard deviation of the kappa statistic is fairly large compared to the bagged decision tree model. This suggests that the performance varies substantially among the folds in the cross-validation. Such variation may imply that the performance might be improved further by upping the number of models in the ensemble.

Boosting

Another common ensemble-based method is called **boosting** because it boosts the performance of weak learners to attain the performance of stronger learners. This method is based largely on the work of Robert Schapire and Yoav Freund, who have published extensively on the topic.

For additional information on boosting, refer to Schapire RE, Freund Y. *Boosting: Foundations and Algorithms*. Cambridge, MA, The MIT Press; 2012.

Similar to bagging, boosting uses ensembles of models trained on resampled data and a vote to determine the final prediction. There are two key distinctions. First, the resampled datasets in boosting are constructed specifically to generate complementary learners. Second, rather than giving each learner an equal vote, boosting gives each learner's vote a weight based on its past performance. Models that perform better have greater influence over the ensemble's final prediction.

Boosting will result in performance that is often quite better and certainly no worse than the best of the models in the ensemble. Since the models in the ensemble are built to be complementary, it is possible to increase ensemble performance to an arbitrary threshold simply by adding additional classifiers to the group, assuming that each classifier performs better than random chance. Given the obvious utility of this finding, boosting is thought to be one of the most significant discoveries in machine learning.

Although boosting can create a model that meets an arbitrarily low error rate, this may not always be reasonable in practice. For one, the performance gains are incrementally smaller as additional learners are added, making some thresholds practically infeasible. Additionally, the pursuit of pure accuracy may result in the model being overfitted to the training data and not generalizable to unseen data.

A boosting algorithm called **AdaBoost** or **adaptive boosting** was proposed by Freund and Schapire in 1997. The algorithm is based on the idea of generating weak learners that iteratively learn a larger portion of the difficult-to-classify examples by paying more attention (that is, giving more weight) to frequently misclassified examples.

Beginning from an unweighted dataset, the first classifier attempts to model the outcome. Examples that the classifier predicted correctly will be less likely to appear in the training dataset for the following classifier, and conversely, the difficult-to-classify examples will appear more frequently. As additional rounds of weak learners are added, they are trained on data with successively more difficult examples. The process continues until the desired overall error rate is reached or performance no longer improves. At that point, each classifier's vote is weighted according to its accuracy on the training data on which it was built.

Though boosting principles can be applied to nearly any type of model, the principles are most commonly used with decision trees. We already used boosting in this way in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, as a method to improve the performance of a C5.0 decision tree.

The **AdaBoost.M1** algorithm provides another tree-based implementation of AdaBoost for classification. The AdaBoost.M1 algorithm can be found in the `adabag` package.

For more information about the `adabag` package, refer to Alfaro E, Gamez M, Garcia N. *adabag – an R package for classification with boosting and bagging*. Journal of Statistical Software. 2013; 54:1-35.

Let's create an `AdaBoost.M1` classifier for the credit data. The general syntax for this algorithm is similar to other modeling techniques:

```
> set.seed(300)
> m_adaboost <- boosting(default ~ ., data = credit)
```

As usual, the `predict()` function is applied to the resulting object to make predictions:

```
> p_adaboost <- predict(m_adaboost, credit)
```

Departing from convention, rather than returning a vector of predictions, this returns an object with information about the model. The predictions are stored in a sub-object called `class`:

```
> head(p_adaboost$class)
[1] "no"  "yes" "no"  "no"  "yes" "no"
```

A confusion matrix can be found in the `confusion` sub-object:

```
> p_adaboost$confusion
                Observed Class
Predicted Class  no yes
            no  700   0
            yes   0 300
```

Did you notice that the AdaBoost model made no mistakes? Before you get your hopes up, remember that the preceding confusion matrix is based on the model's performance on the training data. Since boosting allows the error rate to be reduced to an arbitrarily low level, the learner simply continued until it made no more errors. This likely resulted in overfitting on the training dataset.

For a more accurate assessment of performance on unseen data, we need to use another evaluation method. The `adabag` package provides a simple function to use 10-fold CV:

```
> set.seed(300)
> adaboost_cv <- boosting.cv(default ~ ., data = credit)
```

Depending on your computer's capabilities, this may take some time to run, during which it will log each iteration to screen. After it completes, we can view a more reasonable confusion matrix:

```
> adaboost_cv$confusion
                Observed Class
Predicted Class  no yes
            no  594 151
            yes 106 149
```

We can find the kappa statistic using the `vcd` package as described in *Chapter 10, Evaluating Model Performance*.

```
> library(vcd)
> Kappa(adaboost_cv$confusion)
                value       ASE
Unweighted 0.3606965 0.0323002
Weighted   0.3606965 0.0323002
```

With a kappa of about 0.36, this is our best-performing credit scoring model yet. Let's see how it compares to one last ensemble method.

The AdaBoost.M1 algorithm can be tuned in `caret` by specifying `method = "AdaBoost.M1"`.

Random forests

Another ensemble-based method called **random forests** (or **decision tree forests**) focuses only on ensembles of decision trees. This method was championed by Leo Breiman and Adele Cutler, and combines the base principles of bagging with random feature selection to add additional diversity to the decision tree models. After the ensemble of trees (the forest) is generated, the model uses a vote to combine the trees' predictions.

For more detail on how random forests are constructed, refer to Breiman L. *Random Forests*. Machine Learning. 2001; 45:5-32.

Random forests combine versatility and power into a single machine learning approach. As the ensemble uses only a small, random portion of the full feature set, random forests can handle extremely large datasets, where the so-called "curse of dimensionality" might cause other models to fail. At the same time, its error rates for most learning tasks are on par with nearly any other method.

Although the term "Random Forests" is trademarked by Breiman and Cutler, the term is sometimes used colloquially to refer to any type of decision tree ensemble. A pedant would use the more general term "decision tree forests" except when referring to the specific implementation by Breiman and Cutler.

It's worth noting that relative to other ensemble-based methods, random forests are quite competitive and offer key advantages relative to the competition. For instance, random forests tend to be easier to use and less prone to overfitting. The following table lists the general strengths and weaknesses of random forest models:

Strengths	Weaknesses
• An all-purpose model that performs well on most problems • Can handle noisy or missing data as well as categorical or continuous features • Selects only the most important features • Can be used on data with an extremely large number of features or examples	• Unlike a decision tree, the model is not easily interpretable • May require some work to tune the model to the data

Due to their power, versatility, and ease of use, random forests are quickly becoming one of the most popular machine learning methods. Later on in this chapter, we'll compare a random forest model head-to-head against the boosted C5.0 tree.

Training random forests

Though there are several packages to create random forests in R, the `randomForest` package is perhaps the implementation that is most faithful to the specification by Breiman and Cutler, and is also supported by `caret` for automated tuning. The syntax for training this model is as follows:

Random forest syntax

using the `randomForest()` function in the `randomForest` package

Building the classifier:

```
m <- randomForest(train, class, ntree = 500, mtry = sqrt(p))
```

- `train` is a data frame containing training data
- `class` is a factor vector with the class for each row in the training data
- `ntree` is an integer specifying the number of trees to grow
- `mtry` is an optional integer specifying the number of features to randomly select at each split (uses `sqrt(p)` by default, where `p` is the number of features in the data)

The function will return a random forest object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test, type = "response")
```

- `m` is a model trained by the `randomForest()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier
- `type` is either `"response"`, `"prob"`, or `"votes"` and is used to indicate whether the predictions vector should contain the predicted class, the predicted probabilities, or a matrix of vote counts, respectively.

The function will return predictions according to the value of the `type` parameter.

Example:

```
credit_model <- randomForest(credit_train, loan_default)
credit_prediction <- predict(credit_model, credit_test)
```

By default, the `randomForest()` function creates an ensemble of 500 trees that consider `sqrt(p)` random features at each split, where `p` is the number of features in the training dataset and `sqrt()` refers to R's square root function. Whether or not these default parameters are appropriate depends on the nature of the learning task and training data. Generally, more complex learning problems and larger datasets (either more features or more examples) work better with a larger number of trees, though this needs to be balanced with the computational expense of training more trees.

The goal of using a large number of trees is to train enough so that each feature has a chance to appear in several models. This is the basis of the `sqrt(p)` default value for the `mtry` parameter; using this value limits the features sufficiently so that substantial random variation occurs from tree-to-tree. For example, since the credit data has 16 features, each tree would be limited to splitting on four features at any time.

Let's see how the default `randomForest()` parameters work with the credit data. We'll train the model just as we did with other learners. Again, the `set.seed()` function ensures that the result can be replicated:

```
> library(randomForest)
> set.seed(300)
> rf <- randomForest(default ~ ., data = credit)
```

To look at a summary of the model's performance, we can simply type the resulting object's name:

```
> rf

Call:
 randomForest(formula = default ~ ., data = credit)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 4

        OOB estimate of error rate: 23.8%
Confusion matrix:
     no yes class.error
no  640  60  0.08571429
yes 178 122  0.59333333
```

The output notes that the random forest included 500 trees and tried four variables at each split, just as we expected. At first glance, you might be alarmed at the seemingly poor performance according to the confusion matrix—the error rate of 23.8 percent is far worse than the resubstitution error of any of the other ensemble methods so far. However, this confusion matrix does not show resubstitution error. Instead, it reflects the **out-of-bag error rate** (listed in the output as `OOB estimate of error rate`), which unlike resubstitution error, is an unbiased estimate of the test set error. This means that it should be a fairly reasonable estimate of future performance.

The out-of-bag estimate is computed during the construction of the random forest. Essentially, any example not selected for a single tree's bootstrap sample can be used to test the model's performance on unseen data. At the end of the forest construction, the predictions for each example each time it was held out are tallied, and a vote is taken to determine the final prediction for the example. The total error rate of such predictions becomes the out-of-bag error rate.

Evaluating random forest performance

As mentioned previously, the `randomForest()` function is supported by `caret`, which allows us to optimize the model while, at the same time, calculating performance measures beyond the out-of-bag error rate. To make things interesting, let's compare an auto-tuned random forest to the best auto-tuned boosted C5.0 model we've developed. We'll treat this experiment as if we were hoping to identify a candidate model for submission to a machine learning competition.

We must first load `caret` and set our training control options. For the most accurate comparison of model performance, we'll use repeated 10-fold cross-validation, or 10-fold CV repeated 10 times. This means that the models will take a much longer time to build and will be more computationally intensive to evaluate, but since this is our final comparison we should be *very* sure that we're making the right choice; the winner of this showdown will be our only entry into the machine learning competition.

```
> library(caret)
> ctrl <- trainControl(method = "repeatedcv",
                       number = 10, repeats = 10)
```

Next, we'll set up the tuning grid for the random forest. The only tuning parameter for this model is `mtry`, which defines how many features are randomly selected at each split. By default, we know that the random forest will use `sqrt(16)`, or four features per tree. To be thorough, we'll also test values half of that, twice that, as well as the full set of 16 features. Thus, we need to create a grid with values of `2`, `4`, `8`, and `16` as follows:

```
> grid_rf <- expand.grid(.mtry = c(2, 4, 8, 16))
```

A random forest that considers the full set of features at each split is essentially the same as a bagged decision tree model.

We can supply the resulting grid to the `train()` function with the `ctrl` object as follows. We'll use the kappa metric to select the best model:

```
> set.seed(300)
> m_rf <- train(default ~ ., data = credit, method = "rf",
                metric = "Kappa", trControl = ctrl,
                tuneGrid = grid_rf)
```

The preceding command may take some time to complete as it has quite a bit of work to do! When it finishes, we'll compare that to a boosted tree using `10`, `20`, `30`, and `40` iterations:

```
> grid_c50 <- expand.grid(.model = "tree",
                          .trials = c(10, 20, 30, 40),
                          .winnow = "FALSE")
> set.seed(300)
> m_c50 <- train(default ~ ., data = credit, method = "C5.0",
                 metric = "Kappa", trControl = ctrl,
                 tuneGrid = grid_c50)
```

When the C5.0 decision tree finally completes, we can compare the two approaches side-by-side. For the random forest model, the results are:

```
> m_rf

Resampling results across tuning parameters:

  mtry  Accuracy  Kappa      Accuracy SD  Kappa SD
   2    0.7247    0.1284142  0.01690466   0.06364740
   4    0.7499    0.2933332  0.02989865   0.08768815
   8    0.7539    0.3379986  0.03107160   0.08353988
  16    0.7556    0.3613151  0.03379439   0.08891300
```

For the boosted C5.0 model, the results are:

```
> m_c50

Resampling results across tuning parameters:

  trials  Accuracy  Kappa      Accuracy SD  Kappa SD
  10      0.7325    0.3215655  0.04021093   0.09519817
  20      0.7343    0.3268052  0.04033333   0.09711408
  30      0.7381    0.3343137  0.03672709   0.08942323
  40      0.7388    0.3335082  0.03934514   0.09746073
```

With a kappa of about 0.361, the random forest model with `mtry = 16` was the winner among these eight models. It was higher than the best C5.0 decision tree, which had a kappa of about 0.334, and slightly higher than the `AdaBoost.M1` model with a kappa of about 0.360. Based on these results, we would submit the random forest as our final model. Without actually evaluating the model on the competition data, we have no way of knowing for sure whether it will end up winning, but given our performance estimates, it's the safer bet. With a bit of luck, perhaps we'll come away with the prize.

Summary

After reading this chapter, you should now know the base techniques that are used to win data mining and machine learning competitions. Automated tuning methods can assist with squeezing every bit of performance out of a single model. On the other hand, performance gains are also possible by creating groups of machine learning models that work together.

Although this chapter was designed to help you prepare competition-ready models, note that your fellow competitors have access to the same techniques. You won't be able to get away with stagnancy; therefore, continue to add proprietary methods to your bag of tricks. Perhaps you can bring unique subject-matter expertise to the table, or perhaps your strengths include an eye for detail in data preparation. In any case, practice makes perfect, so take advantage of open competitions to test, evaluate, and improve your own machine learning skillset.

In the next chapter—the last in this book—we'll take a bird's eye look at ways to apply machine learning to some highly specialized and difficult domains using R. You'll gain the knowledge needed to apply machine learning to tasks at the cutting edge of the field.

12
Specialized Machine Learning Topics

Congratulations on reaching this point in your machine learning journey! If you have not already started work on your own projects, you will do so soon. And in doing so, you may find that the task of turning data into action is more difficult than it first appeared.

As you gathered data, you may have realized that the information was trapped in a proprietary format or spread across pages on the Web. Making matters worse, after spending hours reformatting the data, maybe your computer slowed to a crawl after running out of memory. Perhaps R even crashed or froze your machine. Hopefully, you were undeterred, as these issues can be remedied with a bit more effort.

This chapter covers techniques that may not apply to every project, but will prove useful for working around such specialized issues. You might find the information particularly useful if you tend to work with data that is:

- Stored in unstructured or proprietary formats such as web pages, web APIs, or spreadsheets
- From a specialized domain such as bioinformatics or social network analysis
- Too large to fit in memory or analyses take a very long time to complete

You're not alone if you suffer from any of these problems. Although there is no panacea—these issues are the bane of the data scientist as well as the reason data skills are in high demand—through the dedicated efforts of the R community, a number of R packages provide a head start toward solving the problem.

This chapter provides a cookbook of such solutions. Even if you are an experienced R veteran, you may discover a package that simplifies your workflow. Or, perhaps one day, you will author a package that makes work easier for everybody else!

Working with proprietary files and databases

Unlike the examples in this book, real-world data is rarely packaged in a simple CSV form that can be downloaded from a website. Instead, significant effort is needed to prepare data for analysis. Data must be collected, merged, sorted, filtered, or reformatted to meet the requirements of the learning algorithm. This process is informally known as **data munging** or **data wrangling**.

Data preparation has become even more important, as the size of typical datasets has grown from megabytes to gigabytes, and data is gathered from unrelated and messy sources, many of which are stored in massive databases. Several packages and resources for retrieving and working with proprietary data formats and databases are listed in the following sections.

Reading from and writing to Microsoft Excel, SAS, SPSS, and Stata files

A frustrating aspect of data analysis is the large amount of work required to pull and combine data from various proprietary formats. Vast troves of data exist in files and databases that simply need to be unlocked for use in R. Thankfully, packages exist for exactly this purpose.

What used to be a tedious and time-consuming process, requiring knowledge of specific tricks and tools across multiple R packages, has been made trivial by a relatively new R package called `rio` (an acronym for R input and output). This package, by Chung-hong Chan, Geoffrey CH Chan, Thomas J. Leeper, and Christopher Gandrud, is described as a "Swiss-army knife for data". It is capable of importing and exporting a large variety of file formats, including but not limited to: tab-separated (`.tsv`), comma-separated (`.csv`), JSON (`.json`), Stata (`.dta`), SPSS (`.sav` and `.por`), Microsoft Excel (`.xls` and `.xlsx`), Weka (`.arff`), and SAS (`.sas7bdat` and `.xpt`).

For the complete list of file types `rio` can import and export, as well as more detailed usage examples, see `http://cran.r-project.org/web/packages/rio/vignettes/rio.html`.

The `rio` package consists of three functions for working with proprietary data formats: `import()`, `export()`, and `convert()`. Each does exactly what you'd expect, given their name. Consistent with the package's philosophy of keeping things simple, each function uses the filename extension to guess the type of file to import, export, or convert.

For example, to import the credit data from previous chapters, which is stored in CSV format, simply type:

```
> library(rio)
> credit <- import("credit.csv")
```

This creates the `credit` data frame as expected; as a bonus, not only did we not have to specify the CSV file type, `rio` automatically set `stringsAsFactors = FALSE` as well as other reasonable defaults.

To export the `credit` data frame to Microsoft Excel (`.xlsx`) format, use the `export()` function while specifying the desired filename, as follows. For other formats, simply change the file extension to the desired output type:

```
> export(credit, "credit.xlsx")
```

It is also possible to convert the CSV file to another format directly, without an import step, using the `convert()` function. For example, this converts the `credit.csv` file to Stata (`.dta`) format:

```
> convert("credit.csv", "credit.dta")
```

Though the `rio` package covers many common proprietary data formats, it does not do everything. The next section covers other ways to get data into R via database queries.

Querying data in SQL databases

Large datasets are often stored in **Database Management Systems** (**DBMSs**) such as Oracle, MySQL, PostgreSQL, Microsoft SQL, or SQLite. These systems allow the datasets to be accessed using a **Structured Query Language** (**SQL**), a programming language designed to pull data from databases. If your DBMS is configured to allow **Open Database Connectivity** (**ODBC**), the `RODBC` package by Brian Ripley can be used to import this data directly into an R data frame.

If you have trouble using ODBC to connect to your database, you may try one of the DMBS-specific R packages. These include `ROracle`, `RMySQL`, `RPostgreSQL`, and `RSQLite`. Though they will function largely similar to the instructions here, refer to the package documentation on CRAN for instructions specific to each package.

ODBC is a standard protocol for connecting to databases regardless of operating system or DBMS. If you were previously connected to an ODBC database, you most likely would have referred to it via its **Data Source Name** (**DSN**). You will need the DSN, plus a username and password (if your database requires it) to use `RODBC`.

The instructions to configure an ODBC connection are highly specific to the combination of the OS and DBMS. If you are having trouble setting up an ODBC connection, check with your database administrator. Another way to obtain help is via the `RODBC` package `vignette`, which can be accessed in R with the `vignette("RODBC")` command after the `RODBC` package has been installed.

To open a connection called `my_db` for the database with the `my_dsn` DSN, use the `odbcConnect()` function:

```
> library(RODBC)
> my_db <- odbcConnect("my_dsn")
```

Alternatively, if your ODBC connection requires a username and password, they should be specified while calling the `odbcConnect()` function:

```
> my_db <- odbcConnect("my_dsn",
    uid = "my_username",
    pwd = "my_password")
```

With an open database connection, we can use the `sqlQuery()` function to create an R data frame from the database rows pulled by an SQL query. This function, like the many functions that create data frames, allows us to specify `stringsAsFactors = FALSE` to prevent R from automatically converting character data into factors.

The `sqlQuery()` function uses typical SQL queries, as shown in the following command:

```
> my_query <- "select * from my_table where my_value = 1"
> results_df <- sqlQuery(channel = my_db, query = sql_query,
    stringsAsFactors = FALSE)
```

The resulting `results_df` object is a data frame containing all of the rows selected using the SQL query stored in `sql_query`.

Once you are done using the database, the connection can be closed using the following command:

```
> odbcClose(my_db)
```

Although R will automatically close ODBC connections at the end of an R session, it is a better practice to do so explicitly.

Working with online data and services

With the growing amount of data available from web-based sources, it is increasingly important for machine learning projects to be able to access and interact with online services. R is able to read data from online sources natively, with some caveats. Firstly, by default, R cannot access secure websites (those using the `https://` rather than the `http://` protocol). Secondly, it is important to note that most web pages do not provide data in a form that R can understand. The data would need to be **parsed**, or broken apart and rebuilt into a structured form, before it can be useful. We'll discuss the workarounds shortly.

However, if neither of these caveats applies (that is, if data are already online on a nonsecure website and in a tabular form, like CSV, that R can understand natively), then R's `read.csv()` and `read.table()` functions will be able to access data from the Web just as if it were on your local machine. Simply supply the full URL for the dataset as follows:

```
> mydata <- read.csv("http://www.mysite.com/mydata.csv")
```

R also provides functionality to download other files from the Web, even if R cannot use them directly. For a text file, try the `readLines()` function as follows:

```
> mytext <- readLines("http://www.mysite.com/myfile.txt")
```

For other types of files, the `download.file()` function can be used. To download a file to R's current working directory, simply supply the URL and destination filename as follows:

```
> download.file("http://www.mysite.com/myfile.zip", "myfile.zip")
```

Beyond this base functionality, there are numerous packages that extend R's capabilities to work with online data. The most basic of them will be covered in the sections that follow. As the Web is massive and ever-changing, these sections are far from a comprehensive set of all the ways R can connect to online data. There are literally hundreds of packages for everything from niche projects to massive ones.

For the most complete and up-to-date list of packages, refer to the regularly updated CRAN Web Technologies and Services task view at `http://cran.r-project.org/web/views/WebTechnologies.html`.

Downloading the complete text of web pages

The `RCurl` package by Duncan Temple Lang supplies a more robust way of accessing web pages by providing an R interface to the **curl** (client for URLs) utility, a command-line tool to transfer data over networks. The curl program acts much like a programmable web browser; given a set of commands, it can access and download the content of nearly anything available on the Web. Unlike R, it can access secure websites as well as post data to online forms. It is an incredibly powerful utility.

Precisely because it is so powerful, a complete curl tutorial is outside the scope of this chapter. Instead, refer to the online `RCurl` documentation at `http://www.omegahat.org/RCurl/`.

After installing the `RCurl` package, downloading a page is as simple as typing:

```
> library(RCurl)
> packt_page <- ("https://www.packtpub.com/")
```

This will save the full text of the Packt Publishing homepage (including all the web markup) into the R character object named `packt_page`. As shown in the following lines, this is not very useful:

```
> str(packt_page, nchar.max=200)
 chr "<!DOCTYPE html>\n<html xmlns=\"http://www.w3.org/1999/xhtml\"
lang=\"en\" xml:lang=\"en\">\n\t<head>\n\t\t<title>Packt Publishing |
Technology Books, eBooks & Videos</title>\n\t\t<script>\n\t\t\tdata"|
__truncated__
```

The reason that the first 200 characters of the page look like nonsense is because the websites are written using **Hypertext Markup Language** (**HTML**), which combines the page text with special tags that tell web browsers how to display the text. The `<title>` and `</title>` tags here surround the page title, telling the browser that this is the Packt Publishing homepage. Similar tags are used to denote other portions of the page.

Though curl is the cross-platform standard to access online content, if you work with web data frequently in R, the `httr` package by Hadley Wickham builds upon the foundation of `RCurl` to make it more convenient and R-like. We can see some of the differences immediately by attempting to download the Packt Publishing homepage using the `httr` package's `GET()` function:

```
> library(httr)
> packt_page <- GET("https://www.packtpub.com")
> str(packt_page, max.level = 1)
```

```
List of 9
 $ url        : chr "https://www.packtpub.com/"
 $ status_code: int 200
 $ headers    : List of 11
 $ all_headers: List of 1
 $ cookies    : list()
 $ content    : raw [1:58560] 3c 21 44 4f ...
 $ date       : POSIXct[1:1], format: "2015-05-24 20:46:40"
 $ times      : Named num [1:6] 0 0.000071 0.000079 ...
 $ request    : List of 5
```

Where the `getURL()` function in `RCurl` downloaded only the HTML, the `GET()` function returns a list with site properties in addition to the HTML. To access the page content itself, we need to use the `content()` function:

```
> str(content(packt_page, type="text"), nchar.max=200)
 chr "<!DOCTYPE html>\n<html xmlns=\"http://www.w3.org/1999/xhtml\"
lang=\"en\" xml:lang=\"en\">\n\t<head>\n\t\t<title>Packt Publishing |
Technology Books, eBooks & Videos</title>\n\t\t<script>\n\t\t\tdata"|
__truncated__
```

In order to use this data in an R program, it is necessary to process the page to transform it into a structured format like a list or data frame. Functions to do so are discussed in the sections that follow.

For detailed `httr` documentation and tutorials, visit the project GitHub page at `https://github.com/hadley/httr`. The quickstart guide is particularly helpful to learn the base functionality.

Scraping data from web pages

Because there is a consistent structure of the HTML tags of many web pages, it is possible to write programs that look for desired sections of the page and extract them in order to compile them into a dataset. This process practice of harvesting data from websites and transforming it into a structured form is known as **web scraping**.

Though it is frequently used, scraping should be considered a last resort to get data from the Web. This is because any changes to the underlying HTML structure may break your code, requiring efforts to be fixed. Even worse, it may introduce unnoticed errors into your data. Additionally, many websites' terms of use agreements explicitly forbid automated data extraction, not to mention the fact that your program's traffic may overload their servers. Always check the site's terms before you begin your project; you may even find that the site offers its data freely via a developer agreement.

The `rvest` package (a pun on the term "harvest") by Hadley Wickham makes web scraping a largely effortless process, assuming the data you want can be found in a consistent place within HTML.

Let's start with a simple example using the Packt Publishing homepage. We begin by downloading the page as before, using the `html()` function in the `rvest` package. Note that this function, when supplied with a URL, simply calls the `GET()` function in Hadley Wickham's `httr` package:

```
> library(rvest)
> packt_page <- html("https://www.packtpub.com")
```

Suppose we'd like to scrape the page title. Looking at the previous HTML code, we know that there is only one title per page wrapped within `<title>` and `</title>` tags. To pull the title, we supply the tag name to the `html_node()` function, as follows:

```
> html_node(packt_page, "title")
<title>Packt Publishing | Technology Books, eBooks &amp;
Videos</title>
```

This keeps the HTML formatting in place, including the `<title>` tags and the `&amp;` code, which is the HTML designation for the ampersand symbol. To translate this into plain text, we simply run it through the `html_text()` function, as shown:

```
> html_node(packt_page, "title") %>% html_text()
[1] "Packt Publishing | Technology Books, eBooks & Videos"
```

Notice the use of the `%>%` operator. This is known as a pipe, because it essentially "pipes" data from one function to another. The use of pipes allows the creation of powerful chains of functions to process HTML data.

The pipe operator is a part of the `magrittr` package by Stefan Milton Bache and Hadley Wickham, installed by default with the `rvest` package. The name is a play on René Magritte's famous painting of a pipe (you may recall seeing it in *Chapter 1, Introducing Machine Learning*). For more information on the project, visit its GitHub page at `https://github.com/smbache/magrittr`.

Let's try a slightly more interesting example. Suppose we'd like to scrape a list of all the packages on the CRAN machine learning task view. We begin as in the same way we did it earlier, by downloading the HTML page using the `html()` function. Since we don't know how the page is structured, we'll also peek into HTML by typing `cran_ml`, the name of the R object we created:

```
> library(rvest)
> cran_ml <- html("http://cran.r-project.org/web/views/MachineLearning.
html")
> cran_ml
```

Looking over the output, we find that one section appears to have the data we're interested in. Note that only a subset of the output is shown here:

```
  <h3>CRAN packages:</h3>
  <ul>
    <li><a href="../packages/ahaz/index.html">ahaz</a></li>
    <li><a href="../packages/arules/index.html">arules</a></li>
    <li><a href="../packages/bigrf/index.html">bigrf</a></li>
    <li><a href="../packages/bigRR/index.html">bigRR</a></li>
    <li><a href="../packages/bmrm/index.html">bmrm</a></li>
    <li><a href="../packages/Boruta/index.html">Boruta</a></li>
    <li><a href="../packages/bst/index.html">bst</a></li>
    <li><a href="../packages/C50/index.html">C50</a></li>
    <li><a href="../packages/caret/index.html">caret</a></li>
```

The `<h3>` tags imply a header of size 3, while the `<ul>` and `<li>` tags refer to the creation of an unordered list and list items, respectively. The data elements we want are surrounded by `<a>` tags, which are hyperlink anchor tags that link to the CRAN page for each package.

Because the CRAN page is actively maintained and may be changed at any time, do not be surprised if your results differ from those shown here.

With this knowledge in hand, we can scrape the links much like we did previously. The one exception is that, because we expect to find more than one result, we need to use the `html_nodes()` function to return a vector of results rather than `html_node()`, which returns only a single item:

```
> ml_packages <- html_nodes(cran_ml, "a")
```

Let's peek at the result using the `head()` function:

```
> head(ml_packages, n = 7)
[[1]]
<a href="../packages/nnet/index.html">nnet</a>

[[2]]
<a href="../packages/RSNNS/index.html">RSNNS</a>

[[3]]
<a href="../packages/rpart/index.html">rpart</a>

[[4]]
<a href="../packages/tree/index.html">tree</a>

[[5]]
<a href="../packages/rpart/index.html">rpart</a>

[[6]]
<a href="http://www.cs.waikato.ac.nz/~ml/weka/">Weka</a>

[[7]]
<a href="../packages/RWeka/index.html">RWeka</a>
```

As we can see on line 6, it looks like the links to some other projects slipped in. This is because some packages are hyperlinked to additional websites; in this case, the `RWeka` package is linked to both CRAN and its homepage. To exclude these results, you might chain this output to another function that could look for the `/packages` string in the hyperlink.

In general, web scraping is always a process of iterate-and-refine as you identify more specific criteria to exclude or include specific cases. The most difficult cases may even require a human eye to achieve 100 percent accuracy.

These are simple examples that merely scratch the surface of what is possible with the `rvest` package. Using the pipe functionality, it is possible to look for tags nested within tags or specific classes of HTML tags. For these types of complex examples, refer to the package documentation.

Parsing XML documents

XML is a plaintext, human-readable, structured markup language upon which many document formats have been based. It employs a tagging structure in some ways similar to HTML, but is far stricter about formatting. For this reason, it is a popular online format to store structured datasets.

The `XML` package by Duncan Temple Lang provides a suite of R functionality based on the popular C-based `libxml2` parser to read and write XML documents. It is the grandfather of XML parsing packages in R and is still widely used.

Information on the `XML` package, including simple examples to get you started quickly, can be found on the project's website at `http://www.omegahat.org/RSXML/`.

Recently, the `xml2` package by Hadley Wickham has surfaced as an easier and more R-like interface to the `libxml2` library. The `rvest` package, which was covered earlier in this chapter, utilizes `xml2` behind the scenes to parse HTML. Moreover, `rvest` can be used to parse XML as well.

The `xml2` GitHub page is found at `https://github.com/hadley/xml2`.

Because parsing XML is so closely related to parsing HTML, the exact syntax is not covered here. Please refer to these packages' documentation for examples.

Parsing JSON from web APIs

Online applications communicate with one another using web-accessible functions known as **Application Programming Interfaces (APIs)**. These interfaces act much like a typical website; they receive a request from a client via a particular URL and return a response. The difference is that a normal website returns HTML meant for display in a web browser, while an API typically returns data in a structured form meant for processing by a machine.

Though it is not uncommon to find XML-based APIs, perhaps the most common API data structure today is **JavaScript Object Notation (JSON)**. Like XML, it is a standard, plaintext format, most often used for data structures and objects on the Web. The format has become popular recently due to its roots in browser-based JavaScript applications, but despite the pedigree, its utility is not limited to the Web. The ease in which JSON data structures can be understood by humans and parsed by machines makes it an appealing data structure for many types of projects.

JSON is based on a simple `{key: value}` format. The { } brackets denote a JSON object, and the `key` and `value` parameters denote a property of the object and the status of the property. An object can have any number of properties and the properties themselves may be objects. For example, a JSON object for this book might look something like this:

```
{
  "title": "Machine Learning with R",
  "author": "Brett Lantz",
  "publisher": {
     "name": "Packt Publishing",
     "url": "https://www.packtpub.com"
  },
  "topics": ["R", "machine learning", "data mining"],
  "MSRP": 54.99
}
```

This example illustrates the data types available to JSON: numeric, character, array (surrounded by `[` and `]` characters), and object. Not shown are the `null` and Boolean (`true` or `false`) values. The transmission of these types of objects from application to application and application to web browser, is what powers many of the most popular websites.

For details on the JSON format, go to `http://www.json.org/`.

Public-facing APIs allow programs like R to systematically query websites to retrieve results in the JSON format, using packages like `RCurl` and `httr`. Though a full tutorial on using web APIs is worthy of a separate book, the basic process relies on only a couple of steps—it's the details that are tricky.

Suppose we wanted to query the Google Maps API to locate the latitude and longitude of the Eiffel Tower in France. We first need to review the Google Maps API documentation to determine the URL and parameters needed to make this query. We then supply this information to the `httr` package's `GET()` function, adding a list of query parameters in order to apply the search address:

```
> library(httr)
> map_search <-
    GET("https://maps.googleapis.com/maps/api/geocode/json",
          query = list(address = "Eiffel Tower"))
```

By typing the name of the resulting object, we can see some details about the request:

```
> map_search
Response [https://maps.googleapis.com/maps/api/geocode/
json?address=Eiffel%20T
ower]
  Status: 200
  Content-Type: application/json; charset=UTF-8
  Size: 2.34 kB
{
   "results" : [
      {
         "address_components" : [
            {
               "long_name" : "Eiffel Tower",
               "short_name" : "Eiffel Tower",
               "types" : [ "point_of_interest", "establishment" ]
            },
            {
...
```

To access the resulting JSON, which the `httr` package parsed automatically, we use the `content()` function. For brevity, only a handful of lines are shown here:

```
> content(map_search)
$results[[1]]$formatted_address
[1] "Eiffel Tower, Champ de Mars, 5 Avenue Anatole France, 75007
Paris, France"

$results[[1]]$geometry
$results[[1]]$geometry$location
$results[[1]]$geometry$location$lat
[1] 48.85837

$results[[1]]$geometry$location$lng
[1] 2.294481
```

To access these contents individually, simply refer to them using list syntax. The names are based on the JSON objects returned by the Google API. For instance, the entire set of results is in an object appropriately named `results` and each result is numbered. In this case, we will access the formatted address property of the first result, as well as the latitude and longitude:

```
> content(map_search)$results[[1]]$formatted_address
[1] "Eiffel Tower, Champ de Mars, 5 Avenue Anatole France, 75007
Paris, France"

> content(map_search)$results[[1]]$geometry$location$lat
[1] 48.85837

> content(map_search)$results[[1]]$geometry$location$lng
[1] 2.294481
```

These data elements could then be used in an R program as desired.

Because the Google Maps API may be updated in the future, if you find that your results differ from those shown here, please check the Packt Publishing support page for updated code.

On the other hand, if you would like to do a conversion to and from the JSON format outside the `httr` package, there are a number of packages that add this functionality.

The `rjson` package by Alex Couture-Beil was one of the earliest packages to allow R data structures to be converted back and forth from the JSON format. The syntax is simple. After installing the `rjson` package, to convert from an R object to a JSON string, we use the `toJSON()` function. Notice that the quote characters have escaped using the `\*` notation:

```
> library(rjson)
> ml_book <- list(book_title = "Machine Learning with R",
                  author = "Brett Lantz")
> toJSON(ml_book)
[1] "{\"book_title\":\"Machine Learning with R\",
\"author\":\"Brett Lantz\"}"
```

To convert a JSON string into an R object, use the `fromJSON()` function. Quotation marks in the string need to be escaped, as shown:

```
> ml_book_json <- "{
  \"title\": \"Machine Learning with R\",
  \"author\": \"Brett Lantz\",
  \"publisher\": {
    \"name\": \"Packt Publishing\",
    \"url\": \"https://www.packtpub.com\"
  },
  \"topics\": [\"R\", \"machine learning\", \"data mining\"],
  \"MSRP\": 54.99
}"

> ml_book_r <- fromJSON(ml_book_json)
```

This results in a list structure in a form much like the original JSON:

```
> str(ml_book_r)
List of 5
 $ title    : chr "Machine Learning with R"
 $ author   : chr "Brett Lantz"
 $ publisher:List of 2
  ..$ name: chr "Packt Publishing"
```

```
 ..$ url : chr "https://www.packtpub.com"
$ topics  : chr [1:3] "R" "machine learning" "data mining"
$ MSRP    : num 55
```

Recently, two new JSON packages have arrived on the scene. The first, `RJSONIO`, by Duncan Temple Lang was intended to be a faster and more extensible version of the `rjson` package, though they are now virtually identical. A second package, `jsonlite`, by Jeroen Ooms has quickly gained prominence as it creates data structures that are more consistent and R-like, especially while using data from web APIs. Which of these packages you use is a matter of preference; all three are virtually identical in practice as they each implement a `fromJSON()` and `toJSON()` function.

For more information on the potential benefits of the `jsonlite` package, see: Ooms J. The jsonlite package: a practical and consistent mapping between JSON data and R objects. 2014. Available at: `http://arxiv.org/abs/1403.2805`

Working with domain-specific data

Machine learning has undoubtedly been applied to problems across every discipline. Although the basic techniques are similar across all domains, some are so specialized that communities are formed to develop solutions to the challenges unique to the field. This leads to the discovery of new techniques and new terminology that is relevant only to domain specific problems.

This section covers a pair of domains that use machine learning techniques extensively, but require specialized knowledge to unlock their full potential. Since entire books have been written on these topics, it will serve only as the briefest of introductions. For more details, seek out the help provided by the resources cited in each section.

Analyzing bioinformatics data

The field of **bioinformatics** is concerned with the application of computers and data analysis to the biological domain, particularly with regard to better understanding the genome. As genetic data is unique compared to many other types, data analysis in the field of bioinformatics offers a number of unique challenges. For example, because living creatures have a tremendous number of genes and genetic sequencing is still relatively expensive, typical datasets are much wider than they are long; that is, they have more features (genes) than examples (creatures that have been sequenced). This creates problems while attempting to apply conventional visualizations, statistical tests, and machine learning methods to such data. Additionally, the increasing use of proprietary **microarray** "lab-on-a-chip" techniques requires highly specialized knowledge simply to load the genetic data.

A CRAN task view, which lists some of R's specialized packages for statistical genetics and bioinformatics, is available at `http://cran.r-project.org/web/views/Genetics.html`.

The **Bioconductor** project of the Fred Hutchinson Cancer Research Center in Seattle, Washington, aims to solve some of these problems by providing a standardized set of methods for analyzing genomic data. Using R as its foundation, Bioconductor adds bioinformatics-specific packages and documentation on top of the base R software.

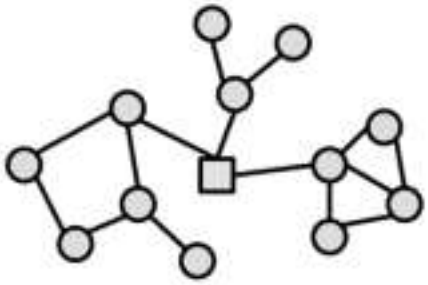
Bioconductor provides workflows to analyze DNA and protein microarray data from common microarray platforms such as Affymetrix, Illumina, Nimblegen, and Agilent. Additional functionality includes sequence annotation, multiple testing procedures, specialized visualizations, tutorials, documentation, and much more.

For more information on the Bioconductor project, visit the project website at `http://www.bioconductor.org`.

Analyzing and visualizing network data

Social network data and graph datasets consist of structures that describe connections, or **links** (sometimes also called **edges**), between people or objects known as **nodes**. With N nodes, a $N \times N = N^2$ matrix of potential links can be created. This creates tremendous computational complexity as the number of nodes grows.

The field of **network analysis** is concerned with statistical measures and visualizations that identify meaningful patterns of connections. For example, the following figure shows three clusters of circular nodes, all connected via a square node at the center. A network analysis may reveal the importance of the square node, among other key metrics.



The `network` package by Carter T. Butts, David Hunter, and Mark S. Handcock offers a specialized data structure to work with networks. This data structure is necessary due to the fact that the matrix needed to store N^2 potential links will quickly exceed available memory; the `network` data structure uses a sparse representation to store only existent links, saving a great deal of memory if most relationships are nonexistent. A closely related package, `sna` (social network analysis), allows the analysis and visualization of the `network` objects.

For more information on `network` and `sna`, including very detailed tutorials and documentation, refer to the project website hosted by the University of Washington at `http://www.statnet.org/`.

The `igraph` package by Gábor Csárdi provides another set of tools to visualize and analyze network data. It is capable of calculating metrics for very large networks. An additional benefit of `igraph` is the fact that it has analogous packages for the Python and C programming languages, allowing it to be used to perform analyses virtually anywhere. As we will demonstrate shortly, it is very easy to use.

For more information on the `igraph` package, including demos and tutorials, visit the homepage at `http://igraph.org/r/`.

Using network data in R requires the use of specialized formats, as network data are not typically stored in typical tabular data structures like CSV files and data frames. As mentioned previously, because there are N^2 potential connections between N network nodes, a tabular structure would quickly grow to be unwieldy for all but the smallest N values. Instead, graph data are stored in a form that lists only the connections that are truly present; absent connections are inferred from the absence of data.

Perhaps the simplest of such formats is **edgelist**, which is a text file with one line per network connection. Each node must be assigned a unique identifier and the links between the nodes are defined by placing the connected nodes' identifiers together on a single line separated by a space. For instance, the following edgelist defines three connections between node 0 and nodes 1, 2, and 3:

```
0 1
0 2
0 3
```

To load network data into R, the `igraph` package provides a `read.graph()` function that can read edgelist files as well as other more sophisticated formats like **Graph Modeling Language** (**GML**). To illustrate this functionality, we'll use a dataset describing friendship among the members of a small karate club. To follow along, download the `karate.txt` file from the Packt Publishing website and save it in your R working directory. After you've installed the `igraph` package, the karate network can be read into R as follows:

```
> library(igraph)
> karate <- read.graph("karate.txt", "edgelist", directed = FALSE)
```

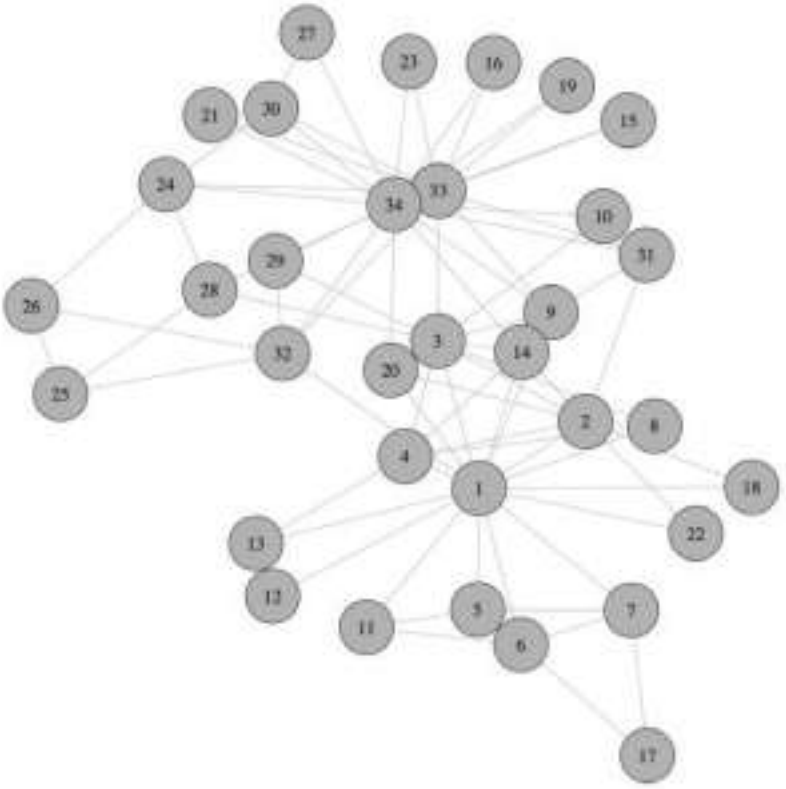
This will create a sparse matrix object that can be used for graphing and network analysis. Note that the `directed = FALSE` parameter forces the network to use undirected or bidirectional links between the nodes. Since the karate dataset describes friendship, it means that if person 1 is friends with person 2, then person 2 must be friends with person 1. On the other hand, if the dataset described fight outcomes, the fact that person 1 defeated person 2 would certainly not imply that person 2 defeated person 1. In this case, the `directed = TRUE` parameter should be set.

The karate network dataset used here was compiled by *M.E.J. Newman* of the University of Michigan. It was first presented in Zachary WW. *An information flow model for conflict and fission in small groups*. Journal of Anthropological Research. 1977; 33:452-473.

To examine the graph, use the `plot()` function:

```
> plot(karate)
```

This produces the following figure:



Examining the network visualization, it is apparent that there are a few highly connected members of the karate club. Nodes 1, 33, and 34 seem to be more central than the others, which remain at the club periphery.

Using `igraph` to calculate graph metrics, it is possible to demonstrate our hunch analytically. The **degree** of a node measures how many nodes it is linked to. The `degree()` function confirms our hunch that nodes 1, 33, and 34 are more connected than the others with `16`, `12`, and `17` connections, respectively:

```
> degree(karate)
 [1] 16  9 10  6  3  4  4  4  5  2  3  1  2  5  2  2  2  2
[19]  2  3  2  2  2  5  3  3  2  4  3  4  4  6 12 17
```

Because some connections are more important than others, a variety of network measures have been developed to measure node connectivity with this consideration. A network metric called **betweenness centrality** is intended to capture the number of shortest paths between nodes that pass through each node. Nodes that are truly more central to the entire graph will have a higher betweenness centrality value, because they act as a bridge between the other nodes. We obtain a vector of the centrality measures using the `betweenness()` function, as follows:

```
> betweenness(karate)
 [1] 231.0714286  28.4785714  75.8507937   6.2880952
 [5]   0.3333333  15.8333333  15.8333333   0.0000000
 [9]  29.5293651   0.4476190   0.3333333   0.0000000
[13]   0.0000000  24.2158730   0.0000000   0.0000000
[17]   0.0000000   0.0000000   0.0000000  17.1468254
[21]   0.0000000   0.0000000   0.0000000   9.3000000
[25]   1.1666667   2.0277778   0.0000000  11.7920635
[29]   0.9476190   1.5428571   7.6095238  73.0095238
[33]  76.6904762 160.5515873
```

As nodes 1 and 34 have much greater betweenness values than the others, they are more central to the karate club's friendship network. These two individuals, with extensive personal friendship networks, may be the "glue" that holds the network together.

Betweenness centrality is only one of many metrics intended to capture a node's importance, and it isn't even the only measure of centrality. Refer to the `igraph` documentation for definitions of other network properties.

The `sna` and `igraph` packages are capable of computing many such graph metrics, which may then be used as inputs to machine learning functions. For example, suppose we were attempting to build a model predicting who would win an election for the club's president. The fact that nodes 1 and 34 are well-connected suggests that they may have the social capital needed for such a leadership role. These might be the highly valuable predictors of the election's results.

By combining network analysis with machine learning, services like Facebook, Twitter, and LinkedIn provide vast stores of network data to make predictions about the users' future behavior. A high-profile example is the 2012 U.S. Presidential campaign in which chief data scientist Rayid Ghani utilized Facebook data to identify people who might be persuaded to vote for Barack Obama.

Improving the performance of R

R has a reputation for being slow and memory-inefficient, a reputation that is at least somewhat earned. These faults are largely unnoticed on a modern PC for datasets of many thousands of records, but datasets with a million records or more can exceed the limits of what is currently possible with consumer-grade hardware. The problem worsens if the dataset contains many features or if complex learning algorithms are being used.

CRAN has a high-performance computing task view that lists packages pushing the boundaries of what is possible in R. It can be viewed at `http://cran.r-project.org/web/views/HighPerformanceComputing.html`.

Packages that extend R past the capabilities of the base software are being developed rapidly. This work comes primarily on two fronts: some packages add the capability to manage extremely large datasets by making data operations faster or allowing the size of the data to exceed the amount of available system memory; others allow R to work faster, perhaps by spreading the work over additional computers or processors, utilizing specialized computer hardware, or providing machine learning algorithms optimized for big data problems.

Managing very large datasets

Extremely large datasets can cause R to grind to a halt when the system runs out of memory to store data. Even if the entire dataset can fit into the available memory, additional memory overhead will be needed for data processing. Furthermore, very large datasets can take a long amount of time to analyze for no reason other than the sheer volume of records; even a quick operation can cause delays when performed many millions of times.

Years ago, many would perform data preparation outside R in another programming language, or use R but perform analyses on a smaller subset of data. However, this is no longer necessary, as several packages have been contributed to R to address these problems.

Generalizing tabular data structures with dplyr

The `dplyr` package introduced in 2014 by Hadley Wickham and Romain Francois is perhaps the most straightforward way to begin working with large datasets in R. Though other packages may exceed its capabilities in terms of raw speed or the raw size of the data, `dplyr` is still quite capable. More importantly, it is virtually transparent after the initial learning curve has passed.

For more information on `dplyr`, including some very helpful tutorials, refer to the project's GitHub page at `https://github.com/hadley/dplyr`.

Put simply, the package provides an object called `tbl`, which is an abstraction of tabular data. It acts much like a data frame, with several important exceptions:

- Key functionality has been written in C++, which according to the authors results in a 20x to 1000x performance increase for many operations.
- R data frames are limited by available memory. The `dplyr` version of a data frame can be linked transparently to disk-based databases that can exceed what can be stored in memory.
- The `dplyr` package makes reasonable assumptions about data frames that optimize your effort as well as memory use. It doesn't automatically change data types. And, if possible, it avoids making copies of data by pointing to the original value instead.
- New operators are introduced that allow common data transformations to be performed with much less code while remaining highly readable.

Making the transition from data frames to `dplyr` is easy. To convert an existing data frame into a `tbl` object, use the `as.tbl()` function:

```
> library(dplyr)
> credit <- read.csv("credit.csv")
> credit_tbl <- as.tbl(credit)
```

Typing the name of the table provides information about the object. Even here, we see a distinction between `dplyr` and typical R behavior; where as a traditional data frame would have displayed many rows of data, `dplyr` objects are more considerate of real-world needs. For example, typing the name of the object provides output summarized in a form that fits a single screen:

```
> credit_tbl
```

```
Source: local data frame [1,000 x 17]

   checking_balance months_loan_duration credit_history             purpose amount
1            < 0 DM                    6       critical furniture/appliances   1169
2        1 - 200 DM                   48           good furniture/appliances   5951
3           unknown                   12       critical             education   2096
4            < 0 DM                   42           good furniture/appliances   7882
5            < 0 DM                   24           poor                   car   4870
6           unknown                   36           good             education   9055
7           unknown                   24           good furniture/appliances   2835
8        1 - 200 DM                   36           good                   car   6948
9           unknown                   12           good furniture/appliances   3059
10       1 - 200 DM                   30       critical                   car   5234
..              ...                  ...            ...                   ...    ...
Variables not shown: savings_balance (fctr), employment_duration (fctr),
  percent_of_income (int), years_at_residence (int), age (int), other_credit (fctr),
  housing (fctr), existing_loans_count (int), job (fctr), dependents (int), phone
  (fctr), default (fctr)
```

Connecting `dplyr` to an external database is straightforward as well. The `dplyr` package provides functions to connect to MySQL, PostgreSQL, and SQLite databases. These create a connection object that allows `tbl` objects to be pulled from the database.

Let's use the `src_sqlite()` function to create a SQLite database to store credit data. SQLite is a simple database that doesn't require a server. It simply connects to a database file, which we'll call `credit.sqlite3`. Since the file doesn't exist yet, we need to set the `create = TRUE` parameter to create the file. Note that for this step to work, you may require to install the `RSQLite` package if you have not already done so:

```
> credit_db_conn <- src_sqlite("credit.sqlite3", create = TRUE)
```

After creating the connection, we need to load the data into the database using the `copy_to()` function. This uses the `credit_tbl` object to create a database table within the database specified by `credit_db_conn`. The `temporary = FALSE` parameter forces the table to be created immediately. Since `dplyr` tries to avoid copying data unless it must, it will only create the table if it is explicitly asked to:

```
> copy_to(credit_db_conn, credit_tbl, temporary = FALSE)
```

Executing the `copy_to()` function will store the data in the `credit.sqlite3` file, which can be transported to other systems as needed. To access this file later, simply reopen the database connection and create a `tbl` object, as follows:

```
> credit_db_conn <- src_sqlite("credit.sqlite3")
> credit_tbl <- tbl(credit_db_conn, "credit_tbl")
```

In spite of the fact that `dplyr` is routed through a database, the `credit_tbl` object here will act exactly like any other `tbl` object and will gain all the other benefits of the `dplyr` package.

Making data frames faster with data.table

The `data.table` package by Matt Dowle, Tom Short, Steve Lianoglou, and Arun Srinivasan provides an enhanced version of a data frame called a **data table**. The `data.table` objects are typically much faster than data frames for subsetting, joining, and grouping operations. For the largest datasets—those with many millions of rows—these objects may be substantially faster than even `dplyr` objects. Yet, because it is essentially an improved data frame, the resulting objects can still be used by any R function that accepts a data frame.

The `data.table` project can be found on GitHub at `https://github.com/Rdatatable/data.table/wiki`.

After installing the `data.table` package, the `fread()` function will read tabular files like CSVs into data table objects. For instance, to load the credit data used previously, type:

```
> library(data.table)
> credit <- fread("credit.csv")
```

The credit data table can then be queried using syntax similar to R's `[row, col]` form, but optimized for speed and some additional useful conveniences. In particular, the data table structure allows the `row` portion to select rows using an abbreviated subsetting command, and the `col` portion to use a function that does something with the selected rows. For example, the following command computes the mean requested loan amount for people with a good credit history:

```
> credit[credit_history == "good", mean(amount)]
[1] 3040.958
```

By building larger queries with this simple syntax, very complex operations can be performed on data tables. Since the data structure is optimized for speed, it can be used with large datasets.

One limitation of the `data.table` structures is that like data frames they are limited by the available system memory. The next two sections discuss packages that overcome this shortcoming at the expense of breaking compatibility with many R functions.

The `dplyr` and `data.table` packages each have unique strengths. For an in-depth comparison, check out the following Stack Overflow discussion at `http://stackoverflow.com/questions/21435339/data-table-vs-dplyr-can-one-do-something-well-the-other-cant-or-does-poorly`. It is also possible to have the best of both worlds, as `data.table` structures can be loaded into `dplyr` using the `tbl_dt()` function.

Creating disk-based data frames with ff

The `ff` package by Daniel Adler, Christian Gläser, Oleg Nenadic, Jens Oehlschlägel, and Walter Zucchini provides an alternative to a data frame (`ffdf`) that allows datasets of over two billion rows to be created, even if this far exceeds the available system memory.

The `ffdf` structure has a physical component that stores the data on a disk in a highly efficient form, and a virtual component that acts like a typical R data frame, but transparently points to the data stored in the physical component. You can imagine the `ffdf` object as a map that points to a location of the data on a disk.

The `ff` project is on the Web at `http://ff.r-forge.r-project.org/`.

A downside of `ffdf` data structures is that they cannot be used natively by most R functions. Instead, the data must be processed in small chunks, and the results must be combined later on. The upside of chunking the data is that the task can be divided across several processors simultaneously using the parallel computing methods presented later in this chapter.

After installing the `ff` package, to read in a large CSV file, use the `read.csv.ffdf()` function, as follows:

```
> library(ff)
> credit <- read.csv.ffdf(file = "credit.csv", header = TRUE)
```

Unfortunately, we cannot work directly with the `ffdf` object, as attempting to treat it like a traditional data frame results in an error message:

```
> mean(credit$amount)
[1] NA
Warning message:
In mean.default(credit$amount) :
  argument is not numeric or logical: returning NA
```

The `ffbase` package by Edwin de Jonge, Jan Wijffels, and Jan van der Laan addresses this issue somewhat by adding capabilities for basic analyses using `ff` objects. This makes it possible to use `ff` objects directly for data exploration. For instance, after installing the `ffbase` package, the mean function works as expected:

```
> library(ffbase)
> mean(credit$amount)
[1] 3271.258
```

The package also provides other basic functionality such as mathematical operators, query functions, summary statistics, and wrappers to work with optimized machine learning algorithms like `biglm` (described later in this chapter). Though these do not completely eliminate the challenges of working with extremely large datasets, they make the process a bit more seamless.

For more information on advanced functionality, visit the `ffbase` project site at `http://github.com/edwindj/ffbase`.

Using massive matrices with bigmemory

The `bigmemory` package by Michael J. Kane, John W. Emerson, and Peter Haverty allows the use of extremely large matrices that exceed the amount of available system memory. The matrices can be stored on a disk or in shared memory, allowing them to be used by other processes on the same computer or across a network. This facilitates parallel computing methods, such as the ones covered later in this chapter.

Additional documentation on the `bigmemory` package can be found at `http://www.bigmemory.org/`.

Because `bigmemory` matrices are intentionally unlike data frames, they cannot be used directly with most of the machine learning methods covered in this book. They also can only be used with numeric data. That said, since they are similar to a typical R matrix, it is easy to create smaller samples or chunks that can be converted into standard R data structures.

The authors also provide the `bigalgebra`, `biganalytics`, and `bigtabulate` packages, which allow simple analyses to be performed on the matrices. Of particular note is the `bigkmeans()` function in the `biganalytics` package, which performs k-means clustering as described in *Chapter 9, Finding Groups of Data – Clustering with k-means*. Due to the highly specialized nature of these packages, use cases are outside the scope of this chapter.

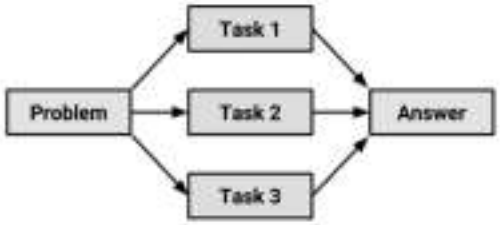
Learning faster with parallel computing

In the early days of computing, processors executed instructions in **serial** which meant that they were limited to performing a single task at a time. The next instruction could not be started until the previous instruction was complete. Although it was widely known that many tasks could be completed more efficiently by completing the steps simultaneously, the technology simply did not exist yet.

Serial computing:

This was addressed by the development of **parallel computing** methods, which use a set of two or more processors or computers to solve a larger problem. Many modern computers are designed for parallel computing. Even in the cases in which they have a single processor, they often have two or more **cores** that are capable of working in parallel. This allows tasks to be accomplished independently of one another.

Parallel computing:



Networks of multiple computers called **clusters** can also be used for parallel computing. A large cluster may include a variety of hardware and be separated over large distances. In this case, the cluster is known as a **grid**. Taken to an extreme, a cluster or grid of hundreds or thousands of computers running commodity hardware could be a very powerful system.

The catch, however, is that not every problem can be parallelized. Certain problems are more conducive to parallel execution than others. One might expect that adding 100 processors would result in accomplishing 100 times the work in the same amount of time (that is, the overall execution time would be 1/100), but this is typically not the case. The reason is that it takes effort to manage the workers. Work must be divided into equal, nonoverlapping tasks, and each of the workers' results must be combined into one final answer.

So-called **embarrassingly parallel** problems are ideal. It is easy to reduce these tasks into nonoverlapping blocks of work and recombine the results. An example of an embarrassingly parallel machine learning task would be 10-fold cross-validation; once the 10 samples are divided, each of the 10 blocks of work is independent, meaning that they do not affect the others. As you will soon see, this task can be sped up quite dramatically using parallel computing.

Measuring execution time

Efforts to speed up R will be wasted if it is not possible to systematically measure how much time is saved. Although a stopwatch is one option, an easier solution would be to wrap the code in a `system.time()` function.

For example, on my laptop, the `system.time()` function notes that it takes about `0.093` seconds to generate a million random numbers:

```
> system.time(rnorm(1000000))
   user  system elapsed
  0.092   0.000   0.093
```

The same function can be used to evaluate the improvement in performance obtained by using the methods that were just described or any R function.

For what it's worth, when the first edition was published, generating a million random numbers took 0.13 seconds. Although I'm now using a slightly more powerful computer, this reduction of about 30 percent of the processing time just two years later illustrates how quickly computer hardware and software are improving.

Working in parallel with multicore and snow

The `parallel` package, now included with R version 2.14.0 and higher, has lowered the entry barrier to deploy parallel algorithms by providing a standard framework to set up worker processes that can complete tasks simultaneously. It does this by including components of the `multicore` and `snow` packages, each taking a different approach towards multitasking.

If your computer is reasonably recent, you are likely to be able to use parallel processing. To determine the number of cores your machine has, use the `detectCores()` function as follows. Note that your output will differ depending on your hardware specifications:

```
> library(parallel)
> detectCores()
[1] 8
```

The `multicore` package was developed by Simon Urbanek and allows parallel processing on a single machine that has multiple processors or processor cores. It utilizes the multitasking capabilities of a computer's operating system to **fork** additional R sessions that share the same memory. It is perhaps the simplest way to get started with parallel processing in R. Unfortunately, because Windows does not support forking, this solution will not work everywhere.

An easy way to get started with the `multicore` functionality is to use the `mclapply()` function, which is a parallel version of `lapply()`. For instance, the following blocks of code illustrate how the task of generating a million random numbers can be divided across 1, 2, 4, and 8 cores. The `unlist()` function is used to combine the parallel results (a list) into a single vector after each core has completed its chunk of work:

```
> system.time(l1 <- rnorm(1000000))
   user  system elapsed
  0.094   0.003   0.097

> system.time(l2 <- unlist(mclapply(1:2, function(x) {
  rnorm(500000)}, mc.cores = 2)))
   user  system elapsed
  0.106   0.045   0.076

> system.time(l4 <- unlist(mclapply(1:4, function(x) {
  rnorm(250000) }, mc.cores = 4)))
   user  system elapsed
  0.135   0.055   0.063

> system.time(l8 <- unlist(mclapply(1:8, function(x) {
  rnorm(125000) }, mc.cores = 8)))
   user  system elapsed
  0.123   0.058   0.055
```

Notice how as the number of cores increases, the elapsed time decreases, and the benefit tapers off. Though this is a simple example, it can be adapted easily to many other tasks.

The `snow` package (simple networking of workstations) by Luke Tierney, A. J. Rossini, Na Li, and H. Sevcikova allows parallel computing on multicore or multiprocessor machines as well as on a network of multiple machines. It is slightly more difficult to use, but offers much more power and flexibility. After installing `snow`, to set up a cluster on a single machine, use the `makeCluster()` function with the number of cores to be used:

```
> library(snow)
> cl1 <- makeCluster(4)
```

Because `snow` communicates via network traffic, depending on your operating system, you may receive a message to approve access through your firewall.

To confirm whether the cluster is operational, we can ask each node to report back its hostname. The `clusterCall()` function executes a function on each machine in the cluster. In this case, we'll define a function that simply calls the `Sys.info()` function and returns the `nodename` parameter:

```
> clusterCall(cl1, function() { Sys.info()["nodename"] } )
[[1]]
                  nodename
"Bretts-Macbook-Pro.local"

[[2]]
                  nodename
"Bretts-Macbook-Pro.local"

[[3]]
                  nodename
"Bretts-Macbook-Pro.local"

[[4]]
                  nodename
"Bretts-Macbook-Pro.local"
```

Unsurprisingly, since all four nodes are running on a single machine, they report back the same hostname. To have the four nodes run a different command, supply them with a unique parameter via the `clusterApply()` function. Here, we'll supply each node with a different letter. Each node will then perform a simple function on its letter in parallel:

```
> clusterApply(cl1, c('A', 'B', 'C', 'D'),
                function(x) { paste("Cluster", x, "ready!") })
[[1]]
[1] "Cluster A ready!"

[[2]]
[1] "Cluster B ready!"

[[3]]
[1] "Cluster C ready!"

[[4]]
[1] "Cluster D ready!"
```

Once we're done with the cluster, it's important to terminate the processes it spawned. This will free up the resources each node is using:

```
> stopCluster(cl1)
```

Using these simple commands, it is possible to speed up many machine learning tasks. For larger big data problems, much more complex `snow` configurations are possible. For instance, you may attempt to configure a **Beowulf cluster**—a network of many consumer-grade machines. In academic and industry research settings with dedicated computing clusters, `snow` can use the `Rmpi` package to access these high-performance **message-passing interface** (**MPI**) servers. Working with such clusters requires the knowledge of network configurations and computing hardware, which is outside the scope of this book.

For a much more detailed introduction to `snow`, including some information on how to configure parallel computing on several computers over a network, see `http://homepage.stat.uiowa.edu/~luke/classes/295-hpc/notes/snow.pdf`.

Taking advantage of parallel with foreach and doParallel

The `foreach` package by Steve Weston of Revolution Analytics provides perhaps the easiest way to get started with parallel computing, particularly if you are running R on Windows, as some of the other packages are platform-specific.

The core of the package is a new `foreach` looping construct. If you have worked with other programming languages, you may be familiar with it. Essentially, it allows looping over a number of items in a set without explicitly counting the number of items; in other words, *for each* item in the set, *do* something.

In addition to the `foreach` package, Revolution Analytics (recently acquired by Microsoft) has developed high-performance, enterprise-ready R builds. Free versions are available for trial and academic use. For more information, see their website at `http://www.revolutionanalytics.com/`.

If you're thinking that R already provides a set of apply functions to loop over the sets of items (for example, `apply()`, `lapply()`, `sapply()`, and so on), you are correct. However, the `foreach` loop has an additional benefit: iterations of the loop can be completed in parallel using a very simple syntax. Let's see how this works.

Recall the command we've been using to generate a million random numbers:

```
> system.time(l1 <- rnorm(1000000))
   user  system elapsed
  0.096   0.000   0.096
```

After the `foreach` package has been installed, it can be expressed by a loop that generates four sets of 250,000 random numbers in parallel. The `.combine` parameter is an optional setting that tells `foreach` which function it should use to combine the final set of results from each loop iteration. In this case, since each iteration generates a set of random numbers, we simply use the `c()` concatenate function to create a single, combined vector:

```
> library(foreach)
> system.time(l4 <- foreach(i = 1:4, .combine = 'c')
                %do% rnorm(250000))
   user  system elapsed
  0.106   0.003   0.109
```

If you noticed that this function didn't result in a speed improvement, good catch! The reason is that by default, the `foreach` package runs each loop iteration in serial. The `doParallel` sister package provides a parallel backend for `foreach` that utilizes the `parallel` package included with R, which was described earlier in this chapter. After installing the `doParallel` package, simply register the number of cores and swap the `%do%` command with `%dopar%`, as follows:

```
> library(doParallel)
> registerDoParallel(cores = 4)
> system.time(l4p <- foreach(i = 1:4, .combine = 'c')
                        %dopar% rnorm(250000))
   user  system elapsed
  0.062   0.030   0.054
```

As shown in the output, this code results in the expected performance improvement, nearly cutting the execution time in half.

To close the `doParallel` cluster, simply type:

```
> stopImplicitCluster()
```

Though the cluster will be closed automatically at the conclusion of the R session, it is better form to do so explicitly.

Parallel cloud computing with MapReduce and Hadoop

The **MapReduce** programming model was developed at Google as a way to process their data on a large cluster of networked computers. MapReduce defined parallel programming as a two-step process:

- A **map** step in which a problem is divided into smaller tasks that are distributed across the computers in the cluster
- A **reduce** step in which the results of the small chunks of work are collected and synthesized into a final solution to the original problem

A popular open source alternative to the proprietary MapReduce framework is **Apache Hadoop**. The Hadoop software comprises of the MapReduce concept, plus a distributed filesystem capable of storing large amounts of data across a cluster of computers.

Packt Publishing has published a large number of books on Hadoop. To search current offerings, visit `https://www.packtpub.com/all/?search=hadoop`.

Several R projects that provide an R interface to Hadoop are in development. The RHadoop project by Revolution Analytics provides an R interface to Hadoop. The project provides a package, `rmr`, intended to be an easy way for R developers to write MapReduce programs. Another companion package, `plyrmr`, provides functionality similar to the `dplyr` package to process large datasets. Additional RHadoop packages provide R functions to access Hadoop's distributed data stores.

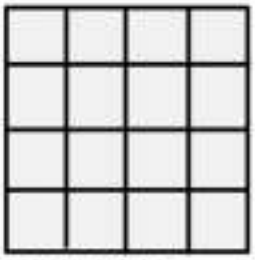
For more information on the RHadoop project, see `https://github.com/RevolutionAnalytics/RHadoop/wiki`.

Another similar project is RHIPE by Saptarshi Guha, which attempts to bring Hadoop's divide and recombine philosophy into R by managing the communication between R and Hadoop.

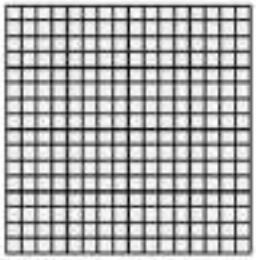
The `RHIPE` package is not yet available at CRAN, but it can be built from the source available on the Web at `http://www.datadr.org`.

GPU computing

An alternative to parallel processing uses a computer's **Graphics Processing Unit** (**GPU**) to increase the speed of mathematical calculations. A GPU is a specialized processor that is optimized to rapidly display images on a computer screen. Because a computer often needs to display complex 3D graphics (particularly for video games), many GPUs use hardware designed for parallel processing and extremely efficient matrix and vector calculations. A side benefit is that they can be used to efficiently solve certain types of mathematical problems. Where a computer processor may have 16 cores, a GPU may have thousands.



CPU with 16 cores



GPU with 1000+ cores

The downside of GPU computing is that it requires specific hardware that is not included in many computers. In most cases, a GPU from the manufacturer Nvidia is required, as they provide a proprietary framework called **Complete Unified Device Architecture** (**CUDA**) that makes the GPU programmable using common languages such as C++.

For more information on Nvidia's role in GPU computing, go to `http://www.nvidia.com/object/what-is-gpu-computing.html`.

The `gputools` package by Josh Buckner, Mark Seligman, and Justin Wilson implements several R functions, such as matrix operations, clustering, and regression modeling using the Nvidia CUDA toolkit. The package requires a CUDA 1.3 or higher GPU and the installation of the Nvidia CUDA toolkit.

Deploying optimized learning algorithms

Some of the machine learning algorithms covered in this book are able to work on extremely large datasets with relatively minor modifications. For instance, it would be fairly straightforward to implement Naive Bayes or the Apriori algorithm using one of the data structures for big datasets described in the previous sections. Some types of learners, such as ensembles, lend themselves well to parallelization, because the work of each model can be distributed across processors or computers in a cluster. On the other hand, some require larger changes to the data or algorithm, or need to be rethought altogether, before they can be used with massive datasets.

The following sections examine packages that provide optimized versions of the learning algorithms we've worked with so far.

Building bigger regression models with biglm

The `biglm` package by Thomas Lumley provides functions to train regression models on datasets that may be too large to fit into memory. It works by using an iterative process in which the model is updated little by little using small chunks of data. In spite of it being a different approach, the results will be nearly identical to what would be obtained by running the conventional `lm()` function on the entire dataset.

For convenience while working with the largest datasets, the `biglm()` function allows the use of a SQL database in place of a data frame. The model can also be trained with chunks obtained from data objects created by the `ff` package described previously.

Growing bigger and faster random forests with bigrf

The `bigrf` package by Aloysius Lim implements the training of random forests for classification and regression on datasets that are too large to fit into memory. It uses the `bigmemory` objects as described earlier in this chapter. For speedier forest growth, the package can be used with the `foreach` and `doParallel` packages described previously to grow trees in parallel.

For more information, including examples and Windows installation instructions, see the package's wiki, which is hosted on GitHub at `https://github.com/aloysius-lim/bigrf`.

Training and evaluating models in parallel with caret

The `caret` package by Max Kuhn (covered extensively in *Chapter 10, Evaluating Model Performance* and *Chapter 11, Improving Model Performance*) will transparently utilize a parallel backend if one has been registered with R using the `foreach` package described previously.

Let's take a look at a simple example in which we attempt to train a random forest model on the credit dataset. Without parallelization, the model takes about 109 seconds to be trained:

```
> library(caret)
> credit <- read.csv("credit.csv")
> system.time(train(default ~ ., data = credit, method = "rf"))
   user  system elapsed
107.862   0.990 108.873
```

On the other hand, if we use the `doParallel` package to register the four cores to be used in parallel, the model takes under 32 seconds to build—less than a third of the time—and we didn't need to change even a single line of the `caret` code:

```
> library(doParallel)
> registerDoParallel(cores = 4)
> system.time(train(default ~ ., data = credit, method = "rf"))
   user  system elapsed
114.578   2.037  31.362
```

Many of the tasks involved in training and evaluating models, such as creating random samples and repeatedly testing predictions for 10-fold cross-validation are embarrassingly parallel and ripe for performance improvements. With this in mind, it is wise to always register multiple cores before beginning a `caret` project.

Configuration instructions and a case study of the performance improvements needed to enable parallel processing in `caret` are available on the project's website at `http://topepo.github.io/caret/parallel.html`.

Summary

It is certainly an exciting time to be studying machine learning. Ongoing work on the relatively uncharted frontiers of parallel and distributed computing offers great potential for tapping the knowledge found in the deluge of big data. The burgeoning data science community is facilitated by the free and open source R programming language, which provides a very low barrier for entry—you simply need to be willing to learn.

The topics you have learned, both in this chapter and in the previous chapters, provide the foundation to understand more advanced machine learning methods. It is now your responsibility to keep learning and adding tools to your arsenal. Along the way, be sure to keep in mind the *No Free Lunch* theorem—no learning algorithm can rule them all, and they all have varying strengths and weaknesses. For this reason, there will always be a human element to machine learning, adding subject-specific knowledge and the ability to match the appropriate algorithm to the task at hand.

In the coming years, it will be interesting to see how the human side changes as the line between machine learning and human learning is blurred. Services such as Amazon's Mechanical Turk provide crowd-sourced intelligence, offering a cluster of human minds ready to perform simple tasks at a moment's notice. Perhaps one day, just as we have used computers to perform tasks that human beings cannot do easily, computers will employ human beings to do the reverse. What interesting food for thought!