4

Probabilistic Learning – Classification Using Naive Bayes

When a meteorologist provides a weather forecast, precipitation is typically described with terms such as "70 percent chance of rain." Such forecasts are known as probability of precipitation reports. Have you ever considered how they are calculated? It is a puzzling question, because in reality, either it will rain or not.

Weather estimates are based on probabilistic methods or those concerned with describing uncertainty. They use data on past events to extrapolate future events. In the case of weather, the chance of rain describes the proportion of prior days to similar measurable atmospheric conditions in which precipitation occurred. A 70 percent chance of rain implies that in 7 out of the 10 past cases with similar conditions, precipitation occurred somewhere in the area.

This chapter covers the Naive Bayes algorithm, which uses probabilities in much the same way as a weather forecast. While studying this method, you will learn:

- Basic principles of probability
- The specialized methods and data structures needed to analyze text data with R
- How to employ Naive Bayes to build an SMS junk message filter

If you've taken a statistics class before, some of the material in this chapter may be a review. Even so, it may be helpful to refresh your knowledge on probability, as these principles are the basis of how Naive Bayes got such a strange name.

Understanding Naive Bayes

The basic statistical ideas necessary to understand the Naive Bayes algorithm have existed for centuries. The technique descended from the work of the 18th century mathematician Thomas Bayes, who developed foundational principles to describe the probability of events, and how probabilities should be revised in the light of additional information. These principles formed the foundation for what are now known as **Bayesian methods**.

We will cover these methods in greater detail later on. But, for now, it suffices to say that a probability is a number between 0 and 1 (that is, between 0 percent and 100 percent), which captures the chance that an event will occur in the light of the available evidence. The lower the probability, the less likely the event is to occur. A probability of 0 indicates that the event will definitely not occur, while a probability of 1 indicates that the event will occur with 100 percent certainty.

Classifiers based on Bayesian methods utilize training data to calculate an observed probability of each outcome based on the evidence provided by feature values. When the classifier is later applied to unlabeled data, it uses the observed probabilities to predict the most likely class for the new features. It's a simple idea, but it results in a method that often has results on par with more sophisticated algorithms. In fact, Bayesian classifiers have been used for:

- Text classification, such as junk e-mail (spam) filtering
- Intrusion or anomaly detection in computer networks
- Diagnosing medical conditions given a set of observed symptoms

Typically, Bayesian classifiers are best applied to problems in which the information from numerous attributes should be considered simultaneously in order to estimate the overall probability of an outcome. While many machine learning algorithms ignore features that have weak effects, Bayesian methods utilize all the available evidence to subtly change the predictions. If large number of features have relatively minor effects, taken together, their combined impact could be quite large.

Basic concepts of Bayesian methods

Before jumping into the Naive Bayes algorithm, it's worth spending some time defining the concepts that are used across Bayesian methods. Summarized in a single sentence, Bayesian probability theory is rooted in the idea that the estimated likelihood of an **event**, or a potential outcome, should be based on the evidence at hand across multiple **trials**, or opportunities for the event to occur.

The following table illustrates events and trials for several real-world outcomes:

Event	Trial
Heads result	Coin flip
Rainy weather	A single day
Message is spam	Incoming e-mail message
Candidate becomes president	Presidential election
Win the lottery	Lottery ticket

Bayesian methods provide insights into how the probability of these events can be estimated from the observed data. To see how, we'll need to formalize our understanding of probability.

Understanding probability

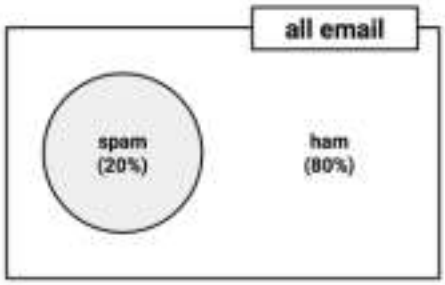
The probability of an event is estimated from the observed data by dividing the number of trials in which the event occurred by the total number of trials. For instance, if it rained 3 out of 10 days with similar conditions as today, the probability of rain today can be estimated as $3 / 10 = 0.30$ or 30 percent. Similarly, if 10 out of 50 prior email messages were spam, then the probability of any incoming message being spam can be estimated as $10 / 50 = 0.20$ or 20 percent.

To denote these probabilities, we use notation in the form $P(A)$, which signifies the probability of event A. For example, $P(rain) = 0.30$ and $P(spam) = 0.20$.

The probability of all the possible outcomes of a trial must always sum to 1, because a trial always results in some outcome happening. Thus, if the trial has two outcomes that cannot occur simultaneously, such as rainy versus sunny or spam versus ham (nonspam), then knowing the probability of either outcome reveals the probability of the other. For example, given the value $P(spam) = 0.20$, we can calculate $P(ham) = 1 - 0.20 = 0.80$. This concludes that spam and ham are **mutually exclusive and exhaustive** events, which implies that they cannot occur at the same time and are the only possible outcomes.

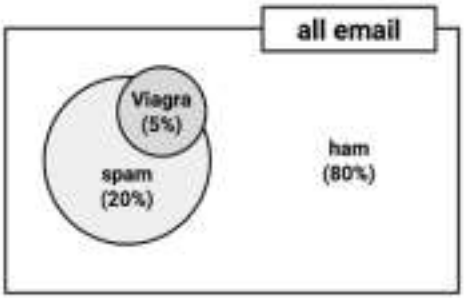
Because an event cannot simultaneously happen and not happen, an event is always mutually exclusive and exhaustive with its **complement**, or the event comprising of the outcomes in which the event of interest does not happen. The complement of event A is typically denoted A^c or A'. Additionally, the shorthand notation $P(\neg A)$ can used to denote the probability of event A not occurring, as in $P(\neg spam) = 0.80$. This notation is equivalent to $P(A^c)$.

To illustrate events and their complements, it is often helpful to imagine a two-dimensional space that is partitioned into probabilities for each event. In the following diagram, the rectangle represents the possible outcomes for an e-mail message. The circle represents the 20 percent probability that the message is spam. The remaining 80 percent represents the complement *P(¬spam)* or the messages that are not spam:



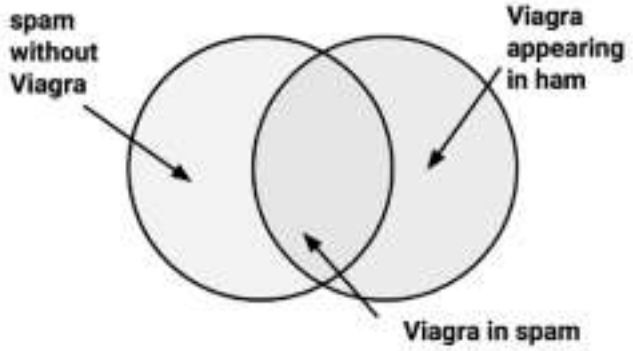
Understanding joint probability

Often, we are interested in monitoring several nonmutually exclusive events for the same trial. If certain events occur with the event of interest, we may be able to use them to make predictions. Consider, for instance, a second event based on the outcome that an e-mail message contains the word Viagra. In most cases, this word is likely to appear only in a spam message; its presence in an incoming e-mail is therefore a very strong piece of evidence that the message is spam. The preceding diagram, updated for this second event, might appear as shown in the following diagram:



Notice in the diagram that the Viagra circle does not completely fill the spam circle, nor is it completely contained by the spam circle. This implies that not all spam messages contain the word Viagra and not every e-mail with the word Viagra is spam.

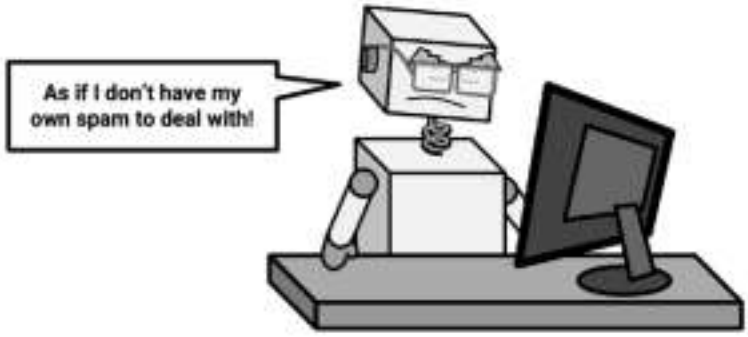
To zoom in for a closer look at the overlap between the spam and Viagra circles, we'll employ a visualization known as a **Venn diagram**. First used in the late 19th century by John Venn, the diagram uses circles to illustrate the overlap between sets of items. In most Venn diagrams, the size of the circles and the degree of the overlap is not meaningful. Instead, it is used as a reminder to allocate probability to all possible combinations of events:



We know that 20 percent of all messages were spam (the left circle) and 5 percent of all messages contained the word Viagra (the right circle). We would like to quantify the degree of overlap between these two proportions. In other words, we hope to estimate the probability that both *P(spam)* and *P(Viagra)* occur, which can be written as *P(spam ∩ Viagra)*. The upside down 'U' symbol signifies the **intersection** of the two events; the notation *A ∩ B* refers to the event in which both *A* and *B* occur.

Calculating *P(spam ∩ Viagra)* depends on the **joint probability** of the two events or how the probability of one event is related to the probability of the other. If the two events are totally unrelated, they are called **independent events**. This is not to say that independent events cannot occur at the same time; event independence simply implies that knowing the outcome of one event does not provide any information about the outcome of the other. For instance, the outcome of a heads result on a coin flip is independent from whether the weather is rainy or sunny on any given day.

If all events were independent, it would be impossible to predict one event by observing another. In other words, **dependent events** are the basis of predictive modeling. Just as the presence of clouds is predictive of a rainy day, the appearance of the word Viagra is predictive of a spam e-mail.



Calculating the probability of dependent events is a bit more complex than for independent events. If *P(spam)* and *P(Viagra)* were independent, we could easily calculate *P(spam ∩ Viagra)*, the probability of both events happening at the same time. Because 20 percent of all the messages are spam, and 5 percent of all the e-mails contain the word Viagra, we could assume that 1 percent of all messages are spam with the term Viagra. This is because *0.05 * 0.20 = 0.01*. More generally, for independent events *A* and *B*, the probability of both happening can be expressed as *P(A ∩ B) = P(A) * P(B)*.

This said, we know that *P(spam)* and *P(Viagra)* are likely to be highly dependent, which means that this calculation is incorrect. To obtain a reasonable estimate, we need to use a more careful formulation of the relationship between these two events, which is based on advanced Bayesian methods.

Computing conditional probability with Bayes' theorem

The relationships between dependent events can be described using **Bayes' theorem**, as shown in the following formula. This formulation provides a way of thinking about how to revise an estimate of the probability of one event in light of the evidence provided by another event:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

The notation *P(A | B)* is read as the probability of event *A*, given that event *B* occurred. This is known as **conditional probability**, since the probability of *A* is dependent (that is, conditional) on what happened with event *B*. Bayes' theorem tells us that our estimate of *P(A | B)* should be based on *P(A ∩ B)*, a measure of how often *A* and *B* are observed to occur together, and *P(B)*, a measure of how often *B* is observed to occur in general.

Bayes' theorem states that the best estimate of *P(A | B)* is the proportion of trials in which *A* occurred with *B* out of all the trials in which *B* occurred. In plain language, this tells us that if we know event *B* occurred, the probability of event *A* is higher the more often that *A* and *B* occur together each time *B* is observed. In a way, this adjusts *P(A ∩ B)* for the probability of *B* occurring; if *B* is extremely rare, *P(B)* and *P(A ∩ B)* will always be small; however, if *A* and *B* almost always happen together, *P(A | B)* will be high regardless of the probability of *B*.

By definition, *P(A ∩ B) = P(A | B) * P(B)*, a fact that can be easily derived by applying a bit of algebra to the previous formula. Rearranging this formula once more with the knowledge that *P(A ∩ B) = P(B ∩ A)* results in the conclusion that *P(A ∩ B) = P(B | A) * P(A)*, which we can then use in the following formulation of Bayes' theorem:

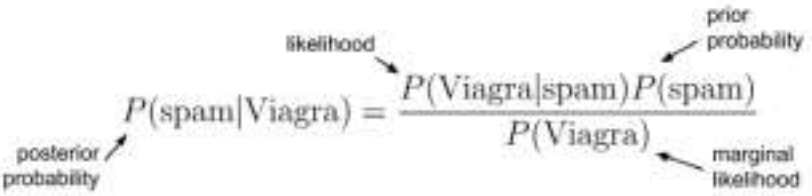
$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)}$$

In fact, this is the traditional way in which Bayes' theorem has been specified, for reasons that will become clear as we apply it to machine learning. First, to better understand how Bayes' theorem works in practice, let's revisit our hypothetical spam filter.

Without knowledge of an incoming message's content, the best estimate of its spam status would be *P(spam)*, the probability that any prior message was spam, which we calculated previously to be 20 percent. This estimate is known as the **prior probability**.

Suppose that you obtained additional evidence by looking more carefully at the set of previously received messages to examine the frequency that the term Viagra appeared. The probability that the word Viagra was used in previous spam messages, or *P(Viagra | spam)*, is called the **likelihood**. The probability that Viagra appeared in any message at all, or *P(Viagra)*, is known as the **marginal likelihood**.

By applying Bayes' theorem to this evidence, we can compute a **posterior probability** that measures how likely the message is to be spam. If the posterior probability is greater than 50 percent, the message is more likely to be spam than ham and it should perhaps be filtered. The following formula shows how Bayes' theorem is applied to the evidence provided by the previous e-mail messages:



To calculate these components of Bayes' theorem, it helps to construct a **frequency table** (shown on the left in the following diagram) that records the number of times Viagra appeared in spam and ham messages. Just like a two-way cross-tabulation, one dimension of the table indicates levels of the class variable (spam or ham), while the other dimension indicates levels for features (Viagra: yes or no). The cells then indicate the number of instances having the particular combination of class value and feature value. The frequency table can then be used to construct a **likelihood table**, as shown on right in the following diagram. The rows of the likelihood table indicate the conditional probabilities for Viagra (yes/no), given that an e-mail was either spam or ham:

	Viagra		
Frequency	Yes	No	Total
spam	4	16	20
ham	1	79	80
Total	5	95	100

	Viagra		
Likelihood	Yes	No	Total
spam	4 / 20	16 / 20	20
ham	1 / 80	79 / 80	80
Total	5 / 100	95 / 100	100

The likelihood table reveals that $P(Viagra=Yes \mid spam) = 4/20 = 0.20$, indicating that the probability is 20 percent that a message contains the term Viagra, given that the message is spam. Additionally, since $P(A \cap B) = P(B \mid A) * P(A)$, we can calculate $P(spam \cap Viagra)$ as $P(Viagra \mid spam) * P(spam) = (4/20) * (20/100) = 0.04$. The same result can be found in the frequency table, which notes that 4 out of the 100 messages were spam with the term Viagra. Either way, this is four times greater than the previous estimate of 0.01 we calculated as $P(A \cap B) = P(A) * P(B)$ under the false assumption of independence. This, of course, illustrates the importance of Bayes' theorem while calculating joint probability.

To compute the posterior probability, *P(spam | Viagra)*, we simply take *P(Viagra | spam) * P(spam) / P(Viagra)* or *(4/20) * (20/100) / (5/100) = 0.80*. Therefore, the probability is 80 percent that a message is spam, given that it contains the word Viagra. In light of this result, any message containing this term should probably be filtered.

This is very much how commercial spam filters work, although they consider a much larger number of words simultaneously while computing the frequency and likelihood tables. In the next section, we'll see how this concept is put to use when additional features are involved.

The Naive Bayes algorithm

The **Naive Bayes** algorithm describes a simple method to apply Bayes' theorem to classification problems. Although it is not the only machine learning method that utilizes Bayesian methods, it is the most common one. This is particularly true for text classification, where it has become the de facto standard. The strengths and weaknesses of this algorithm are as follows:

Strengths	Weaknesses
• Simple, fast, and very effective • Does well with noisy and missing data • Requires relatively few examples for training, but also works well with very large numbers of examples • Easy to obtain the estimated probability for a prediction	• Relies on an often-faulty assumption of equally important and independent features • Not ideal for datasets with many numeric features • Estimated probabilities are less reliable than the predicted classes

The Naive Bayes algorithm is named as such because it makes some "naive" assumptions about the data. In particular, Naive Bayes assumes that all of the features in the dataset are equally important and independent. These assumptions are rarely true in most real-world applications.

For example, if you were attempting to identify spam by monitoring e-mail messages, it is almost certainly true that some features will be more important than others. For example, the e-mail sender may be a more important indicator of spam than the message text. Additionally, the words in the message body are not independent from one another, since the appearance of some words is a very good indication that other words are also likely to appear. A message with the word Viagra will probably also contain the words prescription or drugs.

However, in most cases when these assumptions are violated, Naive Bayes still performs fairly well. This is true even in extreme circumstances where strong dependencies are found among the features. Due to the algorithm's versatility and accuracy across many types of conditions, Naive Bayes is often a strong first candidate for classification learning tasks.

The exact reason why Naive Bayes works well in spite of its faulty assumptions has been the subject of much speculation. One explanation is that it is not important to obtain a precise estimate of probability, so long as the predictions are accurate. For instance, if a spam filter correctly identifies spam, does it matter whether it was 51 percent or 99 percent confident in its prediction? For one discussion of this topic, refer to: Domingos P, Pazzani M. On the optimality of the simple Bayesian classifier under zero-one loss. *Machine Learning*. 1997; 29:103-130.

Classification with Naive Bayes

Let's extend our spam filter by adding a few additional terms to be monitored in addition to the term Viagra: Money, Groceries, and Unsubscribe. The Naive Bayes learner is trained by constructing a likelihood table for the appearance of these four words (labeled W_1, W_2, W_3, and W_4), as shown in the following diagram for 100 e-mails:

	Viagra (W_1)		Money (W_2)		Groceries (W_3)		Unsubscribe (W_4)		
Likelihood	**Yes**	**No**	**Yes**	**No**	**Yes**	**No**	**Yes**	**No**	**Total**
spam	4 / 20	16 / 20	10 / 20	10 / 20	0 / 20	20 / 20	12 / 20	8 / 20	20
ham	1 / 80	79 / 80	14 / 80	66 / 80	8 / 80	71 / 80	23 / 80	57 / 80	80
Total	5 / 100	95 / 100	24 / 100	76 / 100	8 / 100	91 / 100	35 / 100	65 / 100	100

As new messages are received, we need to calculate the posterior probability to determine whether they are more likely to be spam or ham, given the likelihood of the words found in the message text. For example, suppose that a message contains the terms Viagra and Unsubscribe, but does not contain either Money or Groceries.

Using Bayes' theorem, we can define the problem as shown in the following formula. It captures the probability that a message is spam, given that *Viagra* = *Yes*, *Money* = *No*, *Groceries* = *No*, and *Unsubscribe* = *Yes*:

$$P(\text{spam}|W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4) = \frac{P(W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4|\text{spam})P(\text{spam})}{P(W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4)}$$

For a number of reasons, this formula is computationally difficult to solve. As additional features are added, tremendous amounts of memory are needed to store probabilities for all of the possible intersecting events; imagine the complexity of a Venn diagram for the events for four words, let alone for hundreds or more.

The work becomes much easier if we can exploit the fact that Naive Bayes assumes independence among events. Specifically, it assumes **class-conditional independence**, which means that events are independent so long as they are conditioned on the same class value. Assuming conditional independence allows us to simplify the formula using the probability rule for independent events, which states that $P(A \cap B) = P(A) * P(B)$. Because the denominator does not depend on the class (spam or ham), it is treated as a constant value and can be ignored for the time being. This means that the conditional probability of spam can be expressed as:

$$P(\text{spam}|W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4) \propto P(W_1|\text{spam})P(\neg W_2|\text{spam})P(\neg W_3|\text{spam})P(W_4|\text{spam})P(\text{spam})$$

And the probability that the message is ham can be expressed as:

$$P(\text{ham}|W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4) \propto P(W_1|\text{ham})P(\neg W_2|\text{ham})P(\neg W_3|\text{ham})P(W_4|\text{ham})P(\text{ham})$$

Note that the equals symbol has been replaced by the proportional-to symbol (similar to a sideways, open-ended '8') to indicate the fact that the denominator has been omitted.

Using the values in the likelihood table, we can start filling numbers in these equations. The overall likelihood of spam is then:

$$(4/20) * (10/20) * (20/20) * (12/20) * (20/100) = 0.012$$

While the likelihood of ham is:

$$(1/80) * (66/80) * (71/80) * (23/80) * (80/100) = 0.002$$

Because $0.012/0.002 = 6$, we can say that this message is six times more likely to be spam than ham. However, to convert these numbers into probabilities, we need to perform one last step to reintroduce the denominator that had been excluded. Essentially, we must rescale the likelihood of each outcome by dividing it by the total likelihood across all possible outcomes.

In this way, the probability of spam is equal to the likelihood that the message is spam divided by the likelihood that the message is either spam or ham:

$$0.012/(0.012 + 0.002) = 0.857$$

Similarly, the probability of ham is equal to the likelihood that the message is ham divided by the likelihood that the message is either spam or ham:

$$0.002/(0.012 + 0.002) = 0.143$$

Given the pattern of words found in this message, we expect that the message is spam with 85.7 percent probability and ham with 14.3 percent probability. Because these are mutually exclusive and exhaustive events, the probabilities sum to 1.

The Naive Bayes classification algorithm we used in the preceding example can be summarized by the following formula. The probability of level L for class C, given the evidence provided by features F_1 through F_n, is equal to the product of the probabilities of each piece of evidence conditioned on the class level, the prior probability of the class level, and a scaling factor $1/Z$, which converts the likelihood values into probabilities:

$$P(C_L|F_1, ..., F_n) = \frac{1}{Z} p(C_L) \prod_{i=1}^{n} p(F_i|C_L)$$

Although this equation seems intimidating, as the prior example illustrated, the series of steps is fairly straightforward. Begin by building a frequency table, use this to build a likelihood table, and multiply the conditional probabilities according to the Naive Bayes' rule. Finally, divide by the total likelihood to transform each class likelihood into a probability. After attempting this calculation a few times by hand, it will become second nature.

The Laplace estimator

Before we employ Naive Bayes in more complex problems, there are some nuances to consider. Suppose that we received another message, this time containing all four terms: Viagra, Groceries, Money, and Unsubscribe. Using the Naive Bayes algorithm as before, we can compute the likelihood of spam as:

$$(4/20) * (10/20) * (0/20) * (12/20) * (20/100) = 0$$

The likelihood of ham is:

$$(1/80) * (14/80) * (8/80) * (23/80) * (80/100) = 0.00005$$

Therefore, the probability of spam is:

$$0/(0 + 0.00005) = 0$$

The probability of ham is:

$$0.00005/(0 + 0.\ 0.00005) = 1$$

These results suggest that the message is spam with 0 percent probability and ham with 100 percent probability. Does this prediction make sense? Probably not. The message contains several words usually associated with spam, including Viagra, which is rarely used in legitimate messages. It is therefore very likely that the message has been incorrectly classified.

This problem might arise if an event never occurs for one or more levels of the class. For instance, the term Groceries had never previously appeared in a spam message. Consequently, *P(spam | groceries) = 0%*.

Because probabilities in the Naive Bayes formula are multiplied in a chain, this 0 percent value causes the posterior probability of spam to be zero, giving the word Groceries the ability to effectively nullify and overrule all of the other evidence. Even if the e-mail was otherwise overwhelmingly expected to be spam, the absence of the word Groceries in spam will always veto the other evidence and result in the probability of spam being zero.

A solution to this problem involves using something called the **Laplace estimator**, which is named after the French mathematician Pierre-Simon Laplace. The Laplace estimator essentially adds a small number to each of the counts in the frequency table, which ensures that each feature has a nonzero probability of occurring with each class. Typically, the Laplace estimator is set to 1, which ensures that each class-feature combination is found in the data at least once.

The Laplace estimator can be set to any value and does not necessarily even have to be the same for each of the features. If you were a devoted Bayesian, you could use a Laplace estimator to reflect a presumed prior probability of how the feature relates to the class. In practice, given a large enough training dataset, this step is unnecessary and the value of 1 is almost always used.

Let's see how this affects our prediction for this message. Using a Laplace value of 1, we add one to each numerator in the likelihood function. The total number of 1 values must also be added to each conditional probability denominator. The likelihood of spam is therefore:

$$(5/24) * (11/24) * (1/24) * (13/24) * (20/100) = 0.0004$$

The likelihood of ham is:

$$(2/84) * (15/84) * (9/84) * (24/84) * (80/100) = 0.0001$$

This means that the probability of spam is 80 percent, and the probability of ham is 20 percent, which is a more plausible result than the one obtained when the term Groceries alone determined the result.

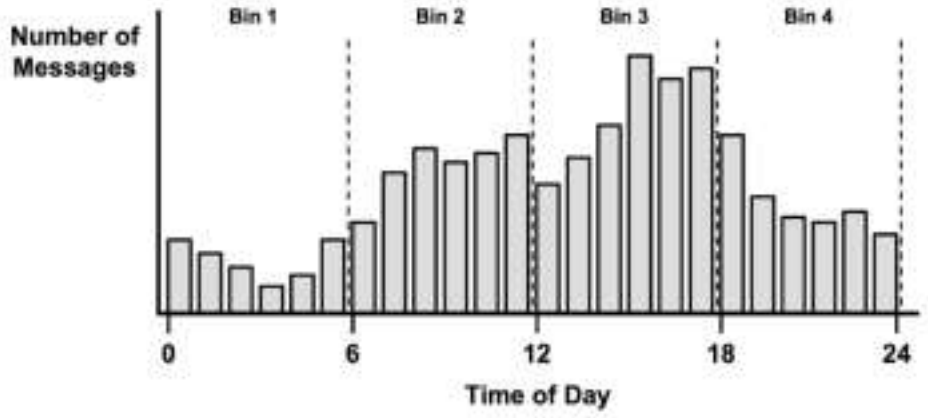
Using numeric features with Naive Bayes

Because Naive Bayes uses frequency tables to learn the data, each feature must be categorical in order to create the combinations of class and feature values comprising of the matrix. Since numeric features do not have categories of values, the preceding algorithm does not work directly with numeric data. There are, however, ways that this can be addressed.

One easy and effective solution is to **discretize** numeric features, which simply means that the numbers are put into categories known as **bins**. For this reason, discretization is also sometimes called **binning**. This method is ideal when there are large amounts of training data, a common condition while working with Naive Bayes.

There are several different ways to discretize a numeric feature. Perhaps the most common is to explore the data for natural categories or **cut points** in the distribution of data. For example, suppose that you added a feature to the spam dataset that recorded the time of night or day the e-mail was sent, from 0 to 24 hours past midnight.

Depicted using a histogram, the time data might look something like the following diagram. In the early hours of the morning, the message frequency is low. The activity picks up during business hours and tapers off in the evening. This seems to create four natural bins of activity, as partitioned by the dashed lines indicating places where the numeric data are divided into levels of a new nominal feature, which could then be used with Naive Bayes:



Keep in mind that the choice of four bins was somewhat arbitrary based on the natural distribution of data and a hunch about how the proportion of spam might change throughout the day. We might expect that spammers operate in the late hours of the night or they may operate during the day, when people are likely to check their e-mail. This said, to capture these trends, we could have just as easily used three bins or twelve.

If there are no obvious cut points, one option will be to discretize the feature using quantiles. You could divide the data into three bins with tertiles, four bins with quartiles, or five bins with quintiles.

One thing to keep in mind is that discretizing a numeric feature always results in a reduction of information as the feature's original granularity is reduced to a smaller number of categories. It is important to strike a balance here. Too few bins can result in important trends being obscured. Too many bins can result in small counts in the Naive Bayes frequency table, which can increase the algorithm's sensitivity to noisy data.

Example – filtering mobile phone spam with the Naive Bayes algorithm

As the worldwide use of mobile phones has grown, a new avenue for electronic junk mail has opened for disreputable marketers. These advertisers utilize Short Message Service (SMS) text messages to target potential consumers with unwanted advertising known as SMS spam. This type of spam is particularly troublesome because, unlike e-mail spam, many cellular phone users pay a fee per SMS received. Developing a classification algorithm that could filter SMS spam would provide a useful tool for cellular phone providers.

Since Naive Bayes has been used successfully for e-mail spam filtering, it seems likely that it could also be applied to SMS spam. However, relative to e-mail spam, SMS spam poses additional challenges for automated filters. SMS messages are often limited to 160 characters, reducing the amount of text that can be used to identify whether a message is junk. The limit, combined with small mobile phone keyboards, has led many to adopt a form of SMS shorthand lingo, which further blurs the line between legitimate messages and spam. Let's see how a simple Naive Bayes classifier handles these challenges.

Step 1 – collecting data

To develop the Naive Bayes classifier, we will use data adapted from the SMS Spam Collection at `http://www.dt.fee.unicamp.br/~tiago/smsspamcollection/`.

To read more about how the SMS Spam Collection was developed, refer to: Gómez JM, Almeida TA, Yamakami A. On the validity of a new SMS spam collection. *Proceedings of the 11th IEEE International Conference on Machine Learning and Applications*. 2012.

This dataset includes the text of SMS messages along with a label indicating whether the message is unwanted. Junk messages are labeled spam, while legitimate messages are labeled ham. Some examples of spam and ham are shown in the following table:

Sample SMS ham	Sample SMS spam
• Better. Made up for Friday and stuffed myself like a pig yesterday. Now I feel bleh. But, at least, its not writhing pain kind of bleh. • If he started searching, he will get job in few days. He has great potential and talent. • I got another job! The one at the hospital, doing data analysis or something, starts on Monday! Not sure when my thesis will finish.	• Congratulations ur awarded 500 of CD vouchers or 125 gift guaranteed & Free entry 2 100 wkly draw txt MUSIC to 87066. • December only! Had your mobile 11mths+? You are entitled to update to the latest colour camera mobile for Free! Call The Mobile Update Co FREE on 08002986906. • Valentines Day Special! Win over £1000 in our quiz and take your partner on the trip of a lifetime! Send GO to 83600 now. 150 p/msg rcvd.

Looking at the preceding messages, did you notice any distinguishing characteristics of spam? One notable characteristic is that two of the three spam messages use the word "free," yet the word does not appear in any of the ham messages. On the other hand, two of the ham messages cite specific days of the week, as compared to zero in spam messages.

Our Naive Bayes classifier will take advantage of such patterns in the word frequency to determine whether the SMS messages seem to better fit the profile of spam or ham. While it's not inconceivable that the word "free" would appear outside of a spam SMS, a legitimate message is likely to provide additional words explaining the context. For instance, a ham message might state "are you free on Sunday?" Whereas, a spam message might use the phrase "free ringtones." The classifier will compute the probability of spam and ham, given the evidence provided by all the words in the message.

Step 2 – exploring and preparing the data

The first step towards constructing our classifier involves processing the raw data for analysis. Text data are challenging to prepare, because it is necessary to transform the words and sentences into a form that a computer can understand. We will transform our data into a representation known as **bag-of-words**, which ignores word order and simply provides a variable indicating whether the word appears at all.

The data used here has been modified slightly from the original in order to make it easier to work with in R. If you plan on following along with the example, download the `sms_spam.csv` file from the Packt website and save it in your R working directory.

We'll begin by importing the CSV data and saving it in a data frame:

```
> sms_raw <- read.csv("sms_spam.csv", stringsAsFactors = FALSE)
```

Using the `str()` function, we see that the `sms_raw` data frame includes 5,559 total SMS messages with two features: `type` and `text`. The SMS type has been coded as either `ham` or `spam`. The `text` element stores the full raw SMS text.

```
> str(sms_raw)
'data.frame':   5559 obs. of  2 variables:
 $ type: chr  "ham" "ham" "ham" "spam" ...
 $ text: chr  "Hope you are having a good week. Just checking in"
"K..give back my thanks." "Am also doing in cbe only. But have to
pay." "complimentary 4 STAR Ibiza Holiday or £10,000 cash needs
your URGENT collection. 09066364349 NOW from Landline not to lose
out"| __truncated__ ...
```

The `type` element is currently a character vector. Since this is a categorical variable, it would be better to convert it into a factor, as shown in the following code:

```
> sms_raw$type <- factor(sms_raw$type)
```

Examining this with the `str()` and `table()` functions, we see that `type` has now been appropriately recoded as a factor. Additionally, we see that 747 (about 13 percent) of SMS messages in our data were labeled as spam, while the others were labeled as ham:

```
> str(sms_raw$type)
 Factor w/ 2 levels "ham","spam": 1 1 1 2 2 1 1 1 2 1 ...
> table(sms_raw$type)
 ham spam
4812  747
```

For now, we will leave the message text alone. As you will learn in the next section, processing the raw SMS messages will require the use of a new set of powerful tools designed specifically to process text data.

Data preparation – cleaning and standardizing text data

SMS messages are strings of text composed of words, spaces, numbers, and punctuation. Handling this type of complex data takes a lot of thought and effort. One needs to consider how to remove numbers and punctuation; handle uninteresting words such as *and*, *but*, and *or*; and how to break apart sentences into individual words. Thankfully, this functionality has been provided by the members of the R community in a text mining package titled `tm`.

The `tm` package was originally created by Ingo Feinerer as a dissertation project at the Vienna University of Economics and Business. To learn more, see: Feinerer I, Hornik K, Meyer D. Text Mining Infrastructure in R. *Journal of Statistical Software*. 2008; 25:1-54.

The `tm` package can be installed via the `install.packages("tm")` command and loaded with the `library(tm)` command. Even if you already have it installed, it may be worth re-running the install process to ensure that your version is up-to-date, as the `tm` package is still being actively developed. This occasionally results in changes to its functionality.

This chapter was written and tested using tm version 0.6-2, which was current as of July 2015. If you see differences in the output or if the code does not work, you may be using a different version. The Packt Publishing support page for this book will post solutions for future `tm` packages if significant changes are noted.

The first step in processing text data involves creating a **corpus**, which is a collection of text documents. The documents can be short or long, from individual news articles, pages in a book or on the web, or entire books. In our case, the corpus will be a collection of SMS messages.

In order to create a corpus, we'll use the `VCorpus()` function in the `tm` package, which refers to a volatile corpus—volatile as it is stored in memory as opposed to being stored on disk (the `PCorpus()` function can be used to access a permanent corpus stored in a database). This function requires us to specify the source of documents for the corpus, which could be from a computer's filesystem, a database, the Web, or elsewhere. Since we already loaded the SMS message text into R, we'll use the `VectorSource()` reader function to create a source object from the existing `sms_raw$text` vector, which can then be supplied to `VCorpus()` as follows:

```
> sms_corpus <- VCorpus(VectorSource(sms_raw$text))
```

The resulting corpus object is saved with the name `sms_corpus`.

By specifying an optional `readerControl` parameter, the `VCorpus()` function offers functionality to import text from sources such as PDFs and Microsoft Word files. To learn more, examine the *Data Import* section in the `tm` package vignette using the `vignette("tm")` command.

By printing the corpus, we see that it contains documents for each of the 5,559 SMS messages in the training data:

```
> print(sms_corpus)
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 5559
```

Because the `tm` corpus is essentially a complex list, we can use list operations to select documents in the corpus. To receive a summary of specific messages, we can use the `inspect()` function with list operators. For example, the following command will view a summary of the first and second SMS messages in the corpus:

```
> inspect(sms_corpus[1:2])
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 2
```

```
[[1]]
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 49

[[2]]
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 23
```

To view the actual message text, the `as.character()` function must be applied to the desired messages. To view one message, use the `as.character()` function on a single list element, noting that the double-bracket notation is required:

```
> as.character(sms_corpus[[1]])
[1] "Hope you are having a good week. Just checking in"
```

To view multiple documents, we'll need to use `as.character()` on several items in the `sms_corpus` object. To do so, we'll use the `lapply()` function, which is a part of a family of R functions that applies a procedure to each element of an R data structure. These functions, which include `apply()` and `sapply()` among others, are one of the key idioms of the R language. Experienced R coders use these much like the way `for` or `while` loops are used in other programming languages, as they result in more readable (and sometimes more efficient) code. The `lapply()` command to apply `as.character()` to a subset of corpus elements is as follows:

```
> lapply(sms_corpus[1:2], as.character)
$`1`
[1] "Hope you are having a good week. Just checking in"

$`2`
[1] "K..give back my thanks."
```

As noted earlier, the corpus contains the raw text of 5,559 text messages. In order to perform our analysis, we need to divide these messages into individual words. But first, we need to clean the text, in order to standardize the words, by removing punctuation and other characters that clutter the result. For example, we would like the strings *Hello!*, *HELLO*, and *hello* to be counted as instances of the same word.

The `tm_map()` function provides a method to apply a transformation (also known as mapping) to a `tm` corpus. We will use this function to clean up our corpus using a series of transformations and save the result in a new object called `corpus_clean`.

Our first order of business will be to standardize the messages to use only lowercase characters. To this end, R provides a `tolower()` function that returns a lowercase version of text strings. In order to apply this function to the corpus, we need to use the `tm` wrapper function `content_transformer()` to treat `tolower()` as a transformation function that can be used to access the corpus. The full command is as follows:

```
> sms_corpus_clean <- tm_map(sms_corpus,
    content_transformer(tolower))
```

To check whether the command worked as advertised, let's inspect the first message in the original corpus and compare it to the same in the transformed corpus:

```
> as.character(sms_corpus[[1]])
[1] "Hope you are having a good week. Just checking in"
> as.character(sms_corpus_clean[[1]])
[1] "hope you are having a good week. just checking in"
```

As expected, uppercase letters have been replaced by lowercase versions of the same.

The `content_transformer()` function can be used to apply more sophisticated text processing and cleanup processes, such as `grep` pattern matching and replacement. Simply write a custom function and wrap it before applying via `tm_map()` as done earlier.

Let's continue our cleanup by removing numbers from the SMS messages. Although some numbers may provide useful information, the majority would likely be unique to individual senders and thus will not provide useful patterns across all messages. With this in mind, we'll strip all the numbers from the corpus as follows:

```
> sms_corpus_clean <- tm_map(sms_corpus_clean, removeNumbers)
```

Note that the preceding code did not use the `content_transformer()` function. This is because `removeNumbers()` is built into `tm` along with several other mapping functions that do not need to be wrapped. To see the other built-in transformations, simply type `getTransformations()`.

Our next task is to remove filler words such as *to*, *and*, *but*, and *or* from our SMS messages. These terms are known as **stop words** and are typically removed prior to text mining. This is due to the fact that although they appear very frequently, they do not provide much useful information for machine learning.

Rather than define a list of stop words ourselves, we'll use the `stopwords()` function provided by the `tm` package. This function allows us to access various sets of stop words, across several languages. By default, common English language stop words are used. To see the default list, type `stopwords()` at the command line. To see the other languages and options available, type `?stopwords` for the documentation page.

Even within a single language, there is no single definitive list of stop words. For example, the default English list in `tm` includes about 174 words while another option includes 571 words. You can even specify your own list of stop words if you prefer. Regardless of the list you choose, keep in mind the goal of this transformation, which is to eliminate all useless data while keeping as much useful information as possible.

The stop words alone are not a useful transformation. What we need is a way to remove any words that appear in the stop words list. The solution lies in the `removeWords()` function, which is a transformation included with the `tm` package. As we have done before, we'll use the `tm_map()` function to apply this mapping to the data, providing the `stopwords()` function as a parameter to indicate exactly the words we would like to remove. The full command is as follows:

```
> sms_corpus_clean <- tm_map(sms_corpus_clean,
    removeWords, stopwords())
```

Since `stopwords()` simply returns a vector of stop words, had we chosen so, we could have replaced it with our own vector of words to be removed. In this way, we could expand or reduce the list of stop words to our liking or remove a completely different set of words entirely.

Continuing with our cleanup process, we can also eliminate any punctuation from the text messages using the built-in `removePunctuation()` transformation:

```
> sms_corpus_clean <- tm_map(sms_corpus_clean, removePunctuation)
```

The `removePunctuation()` transformation strips punctuation characters from the text blindly, which can lead to unintended consequences. For example, consider what happens when it is applied as follows:

```
> removePunctuation("hello...world")
[1] "helloworld"
```

As shown, the lack of blank space after the ellipses has caused the words *hello* and *world* to be joined as a single word. While this is not a substantial problem for our analysis, it is worth noting for the future.

To work around the default behavior of `removePunctuation()`, simply create a custom function that replaces rather than removes punctuation characters:

```
> replacePunctuation <- function(x) {
    gsub("[[:punct:]]+", " ", x)
}
```

Essentially, this uses R's `gsub()` function to substitute any punctuation characters in `x` with a blank space. The `replacePunctuation()` function can then be used with `tm_map()` as with other transformations.

Another common standardization for text data involves reducing words to their root form in a process called **stemming**. The stemming process takes words like *learned*, *learning*, and *learns*, and strips the suffix in order to transform them into the base form, *learn*. This allows machine learning algorithms to treat the related terms as a single concept rather than attempting to learn a pattern for each variant.

The `tm` package provides stemming functionality via integration with the `SnowballC` package. At the time of this writing, `SnowballC` was not installed by default with `tm`. Do so with `install.packages("SnowballC")` if it is not installed already.

The `SnowballC` package is maintained by Milan Bouchet-Valat and provides an R interface to the C-based `libstemmer` library, which is based on M.F. Porter's "Snowball" word stemming algorithm, a widely used open source stemming method. For more detail, see `http://snowball.tartarus.org`.

The `SnowballC` package provides a `wordStem()` function, which for a character vector, returns the same vector of terms in its root form. For example, the function correctly stems the variants of the word *learn*, as described previously:

```
> library(SnowballC)
> wordStem(c("learn", "learned", "learning", "learns"))
[1] "learn"   "learn"   "learn"   "learn"
```

In order to apply the `wordStem()` function to an entire corpus of text documents, the `tm` package includes a `stemDocument()` transformation. We apply this to our corpus with the `tm_map()` function exactly as done earlier:

```
> sms_corpus_clean <- tm_map(sms_corpus_clean, stemDocument)
```

If you receive an error message while applying the `stemDocument()` transformation, please confirm that you have the `SnowballC` package installed. If after installing the package you still encounter the message that `all scheduled cores encountered errors`, you can also try forcing the `tm_map()` command to a single core, by adding an additional parameter to specify `mc.cores=1`.

After removing numbers, stop words, and punctuation as well as performing stemming, the text messages are left with the blank spaces that previously separated the now-missing pieces. The final step in our text cleanup process is to remove additional whitespace, using the built-in `stripWhitespace()` transformation:

```
> sms_corpus_clean <- tm_map(sms_corpus_clean, stripWhitespace)
```

The following table shows the first three messages in the SMS corpus before and after the cleaning process. The messages have been limited to the most interesting words, and punctuation and capitalization have been removed:

SMS messages before cleaning	SMS messages after cleaning

`> as.character(sms_corpus[1:3])`
 	
`[[1]] Hope you are having a good week. Just checking in`
 	
`[[2]] K..give back my thanks.`
 	
`[[3]] Am also doing in cbe only. But have to pay.`
	`> as.character(sms_corpus_clean[1:3])`
	
	`[[1]] hope good week just check`
	
	`[[2]] kgive back thank`
	
	`[[3]] also cbe pay`

Data preparation – splitting text documents into words

Now that the data are processed to our liking, the final step is to split the messages into individual components through a process called **tokenization**. A token is a single element of a text string; in this case, the tokens are words.

As you might assume, the `tm` package provides functionality to tokenize the SMS message corpus. The `DocumentTermMatrix()` function will take a corpus and create a data structure called a **Document Term Matrix** (**DTM**) in which rows indicate documents (SMS messages) and columns indicate terms (words).

The `tm` package also provides a data structure for a **Term Document Matrix (TDM)**, which is simply a transposed DTM in which the rows are terms and the columns are documents. Why the need for both? Sometimes, it is more convenient to work with one or the other. For example, if the number of documents is small, while the word list is large, it may make sense to use a TDM because it is generally easier to display many rows than to display many columns. This said, the two are often interchangeable.

Each cell in the matrix stores a number indicating a count of the times the word represented by the column appears in the document represented by the row. The following illustration depicts only a small portion of the DTM for the SMS corpus, as the complete matrix has 5,559 rows and over 7,000 columns:

message #	balloon	balls	bam	bambling	band
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0

The fact that each cell in the table is zero implies that none of the words listed on the top of the columns appear in any of the first five messages in the corpus. This highlights the reason why this data structure is called a **sparse matrix**; the vast majority of the cells in the matrix are filled with zeros. Stated in real-world terms, although each message must contain at least one word, the probability of any one word appearing in a given message is small.

Creating a DTM sparse matrix, given a `tm` corpus, involves a single command:

```
> sms_dtm <- DocumentTermMatrix(sms_corpus_clean)
```

This will create an `sms_dtm` object that contains the tokenized corpus using the default settings, which apply minimal processing. The default settings are appropriate because we have already prepared the corpus manually.

On the other hand, if we hadn't performed the preprocessing, we could do so here by providing a list of `control` parameter options to override the defaults. For example, to create a DTM directly from the raw, unprocessed SMS corpus, we can use the following command:

```
> sms_dtm2 <- DocumentTermMatrix(sms_corpus, control = list(
    tolower = TRUE,
```

```
    removeNumbers = TRUE,
    stopwords = TRUE,
    removePunctuation = TRUE,
    stemming = TRUE
  ))
```

This applies the same preprocessing steps to the SMS corpus in the same order as done earlier. However, comparing `sms_dtm` to `sms_dtm2`, we see a slight difference in the number of terms in the matrix:

```
> sms_dtm
<<DocumentTermMatrix (documents: 5559, terms: 6518)>>
Non-/sparse entries: 42113/36191449
Sparsity           : 100%
Maximal term length: 40
Weighting          : term frequency (tf)

> sms_dtm2
<<DocumentTermMatrix (documents: 5559, terms: 6909)>>
Non-/sparse entries: 43192/38363939
Sparsity           : 100%
Maximal term length: 40
Weighting          : term frequency (tf)
```

The reason for this discrepancy has to do with a minor difference in the ordering of the preprocessing steps. The `DocumentTermMatrix()` function applies its cleanup functions to the text strings only after they have been split apart into words. Thus, it uses a slightly different stop words removal function. Consequently, some words split differently than when they are cleaned before tokenization.

To force the two prior document term matrices to be identical, we can override the default stop words function with our own that uses the original replacement function. Simply replace `stopwords = TRUE` with the following:

```
stopwords = function(x) { removeWords(x, stopwords()) }
```

The differences between these two cases illustrate an important principle of cleaning text data: the order of operations matters. With this in mind, it is very important to think through how early steps in the process are going to affect later ones. The order presented here will work in many cases, but when the process is tailored more carefully to specific datasets and use cases, it may require rethinking. For example, if there are certain terms you hope to exclude from the matrix, consider whether you should search for them before or after stemming. Also, consider how the removal of punctuation—and whether the punctuation is eliminated or replaced by blank space—affects these steps.

Data preparation – creating training and test datasets

With our data prepared for analysis, we now need to split the data into training and test datasets, so that once our spam classifier is built, it can be evaluated on data it has not previously seen. But even though we need to keep the classifier blinded as to the contents of the test dataset, it is important that the split occurs after the data have been cleaned and processed; we need exactly the same preparation steps to occur on both the training and test datasets.

We'll divide the data into two portions: 75 percent for training and 25 percent for testing. Since the SMS messages are sorted in a random order, we can simply take the first 4,169 for training and leave the remaining 1,390 for testing. Thankfully, the DTM object acts very much like a data frame and can be split using the standard `[row, col]` operations. As our DTM stores SMS messages as rows and words as columns, we must request a specific range of rows and all columns for each:

```
> sms_dtm_train <- sms_dtm[1:4169, ]
> sms_dtm_test  <- sms_dtm[4170:5559, ]
```

For convenience later on, it is also helpful to save a pair of vectors with labels for each of the rows in the training and testing matrices. These labels are not stored in the DTM, so we would need to pull them from the original `sms_raw` data frame:

```
> sms_train_labels <- sms_raw[1:4169, ]$type
> sms_test_labels  <- sms_raw[4170:5559, ]$type
```

To confirm that the subsets are representative of the complete set of SMS data, let's compare the proportion of spam in the training and test data frames:

```
> prop.table(table(sms_train_labels))

      ham       spam
0.8647158 0.1352842

> prop.table(table(sms_test_labels))

      ham       spam
0.8683453 0.1316547
```

Both the training data and test data contain about 13 percent spam. This suggests that the spam messages were divided evenly between the two datasets.

Visualizing text data – word clouds

A **word cloud** is a way to visually depict the frequency at which words appear in text data. The cloud is composed of words scattered somewhat randomly around the figure. Words appearing more often in the text are shown in a larger font, while less common terms are shown in smaller fonts. This type of figures grew in popularity recently, since it provides a way to observe trending topics on social media websites.

The `wordcloud` package provides a simple R function to create this type of diagrams. We'll use it to visualize the types of words in SMS messages, as comparing the clouds for spam and ham will help us gauge whether our Naive Bayes spam filter is likely to be successful. If you haven't already done so, install and load the package by typing `install.packages("wordcloud")` and `library(wordcloud)` at the R command line.

The `wordcloud` package was written by Ian Fellows. For more information on this package, visit his blog at `http://blog.fellstat.com/?cat=11`.

A word cloud can be created directly from a `tm` corpus object using the syntax:

```
> wordcloud(sms_corpus_clean, min.freq = 50, random.order = FALSE)
```

This will create a word cloud from our prepared SMS corpus. Since we specified `random.order = FALSE`, the cloud will be arranged in a nonrandom order with higher frequency words placed closer to the center. If we do not specify `random.order`, the cloud would be arranged randomly by default. The `min.freq` parameter specifies the number of times a word must appear in the corpus before it will be displayed in the cloud. Since a frequency of 50 is about 1 percent of the corpus, this means that a word must be found in at least 1 percent of the SMS messages to be included in the cloud.

You might get a warning message noting that R was unable to fit all of the words in the figure. If so, try increasing `min.freq` to reduce the number of words in the cloud. It might also help to use the `scale` parameter to reduce the font size.

The resulting word cloud should appear similar to the following figure:

A perhaps more interesting visualization involves comparing the clouds for SMS spam and ham. Since we did not construct separate corpora for spam and ham, this is an appropriate time to note a very helpful feature of the `wordcloud()` function. Given a vector of raw text strings, it will automatically apply common text preparation processes before displaying the cloud.

Let's use R's `subset()` function to take a subset of the `sms_raw` data by the SMS `type`. First, we'll create a subset where the message `type` is `spam`:

```
> spam <- subset(sms_raw, type == "spam")
```

Next, we'll do the same thing for the `ham` subset:

```
> ham <- subset(sms_raw, type == "ham")
```

Be careful to note the double equals sign. Like many programming languages, R uses `==` to test equality. If you accidently use a single equals sign, you'll end up with a subset much larger than you expected!

We now have two data frames, `spam` and `ham`, each with a `text` feature containing the raw text strings for SMSes. Creating word clouds is as simple as before. This time, we'll use the `max.words` parameter to look at the 40 most common words in each of the two sets. The scale parameter allows us to adjust the maximum and minimum font size for words in the cloud. Feel free to adjust these parameters as you see fit. This is illustrated in the following commands:

```
> wordcloud(spam$text, max.words = 40, scale = c(3, 0.5))
> wordcloud(ham$text, max.words = 40, scale = c(3, 0.5))
```

The resulting word clouds are shown in the following diagram:

Do you have a hunch about which one is the spam cloud and which represents ham?

Because of the randomization process, each word cloud may look slightly different. Running the `wordcloud()` function several times allows you to choose the cloud that is the most visually appealing for presentation purposes.

As you probably guessed, the spam cloud is on the left. Spam messages include words such as *urgent*, *free*, *mobile*, *claim*, and *stop*; these terms do not appear in the ham cloud at all. Instead, ham messages use words such as *can*, *sorry*, *need*, and *time*. These stark differences suggest that our Naive Bayes model will have some strong key words to differentiate between the classes.

Data preparation – creating indicator features for frequent words

The final step in the data preparation process is to transform the sparse matrix into a data structure that can be used to train a Naive Bayes classifier. Currently, the sparse matrix includes over 6,500 features; this is a feature for every word that appears in at least one SMS message. It's unlikely that all of these are useful for classification. To reduce the number of features, we will eliminate any word that appear in less than five SMS messages, or in less than about 0.1 percent of the records in the training data.

Finding frequent words requires use of the `findFreqTerms()` function in the `tm` package. This function takes a DTM and returns a character vector containing the words that appear for at least the specified number of times. For instance, the following command will display the words appearing at least five times in the `sms_dtm_train` matrix:

```
> findFreqTerms(sms_dtm_train, 5)
```

The result of the function is a character vector, so let's save our frequent words for later on:

```
> sms_freq_words <- findFreqTerms(sms_dtm_train, 5)
```

A peek into the contents of the vector shows us that there are 1,136 terms appearing in at least five SMS messages:

```
> str(sms_freq_words)

 chr [1:1136] "abiola" "abl" "abt" "accept" "access" "account"
"across" "act" "activ" ...
```

We now need to filter our DTM to include only the terms appearing in a specified vector. As done earlier, we'll use the data frame style `[row, col]` operations to request specific portions of the DTM, noting that the columns are named after the words the DTM contains. We can take advantage of this to limit the DTM to specific words. Since we want all the rows, but only the columns representing the words in the `sms_freq_words` vector, our commands are:

```
> sms_dtm_freq_train<- sms_dtm_train[ , sms_freq_words]
> sms_dtm_freq_test <- sms_dtm_test[ , sms_freq_words]
```

The training and test datasets now include 1,136 features, which correspond to words appearing in at least five messages.

The Naive Bayes classifier is typically trained on data with categorical features. This poses a problem, since the cells in the sparse matrix are numeric and measure the number of times a word appears in a message. We need to change this to a categorical variable that simply indicates yes or no depending on whether the word appears at all.

The following defines a `convert_counts()` function to convert counts to `Yes`/`No` strings:

```
> convert_counts <- function(x) {
    x <- ifelse(x > 0, "Yes", "No")
  }
```

By now, some of the pieces of the preceding function should look familiar. The first line defines the function. The `ifelse(x > 0, "Yes", "No")` statement transforms the values in `x`, so that if the value is greater than `0`, then it will be replaced by `"Yes"`, otherwise it will be replaced by a `"No"` string. Lastly, the newly transformed `x` vector is returned.

We now need to apply `convert_counts()` to each of the columns in our sparse matrix. You may be able to guess the R function to do exactly this. The function is simply called `apply()` and is used much like `lapply()` was used previously.

The `apply()` function allows a function to be used on each of the rows or columns in a matrix. It uses a `MARGIN` parameter to specify either rows or columns. Here, we'll use `MARGIN = 2`, since we're interested in the columns (`MARGIN = 1` is used for rows). The commands to convert the training and test matrices are as follows:

```
> sms_train <- apply(sms_dtm_freq_train, MARGIN = 2,
                                         convert_counts)
> sms_test <- apply(sms_dtm_freq_test, MARGIN = 2,
                                        convert_counts)
```

The result will be two character type matrixes, each with cells indicating `"Yes"` or `"No"` for whether the word represented by the column appears at any point in the message represented by the row.

Step 3 – training a model on the data

Now that we have transformed the raw SMS messages into a format that can be represented by a statistical model, it is time to apply the Naive Bayes algorithm. The algorithm will use the presence or absence of words to estimate the probability that a given SMS message is spam.

The Naive Bayes implementation we will employ is in the `e1071` package. This package was developed in the statistics department of the Vienna University of Technology (TU Wien), and includes a variety of functions for machine learning. If you have not done so already, be sure to install and load the package using the `install.packages("e1071")` and `library(e1071)` commands before continuing.

Many machine learning approaches are implemented in more than one R package, and Naive Bayes is no exception. One other option is `NaiveBayes()` in the `klaR` package, which is nearly identical to the one in the `e1071` package. Feel free to use whichever option you prefer.

Unlike the k-NN algorithm we used for classification in the previous chapter, a Naive Bayes learner is trained and used for classification in separate stages. Still, as shown in the following table, these steps are is fairly straightforward:

Naive Bayes classification syntax

using the `naiveBayes()` function in the `e1071` package

Building the classifier:

```
m <- naiveBayes(train, class, laplace = 0)
```

- `train` is a data frame or matrix containing training data
- `class` is a factor vector with the class for each row in the training data
- `laplace` is a number to control the Laplace estimator (by default, 0)

The function will return a naive Bayes model object that can be used to make predictions.

Making predictions:

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

- `m` is a model trained by the `naiveBayes()` function
- `test` is a data frame or matrix containing test data with the same features as the training data used to build the classifier
- `type` is either `"class"` or `"raw"` and specifies whether the predictions should be the most likely class value or the raw predicted probabilities

The function will return a vector of predicted class values or raw predicted probabilities depending upon the value of the `type` parameter.

Example:

```
sms_classifier <- naiveBayes(sms_train, sms_type)
sms_predictions <- predict(sms_classifier, sms_test)
```

To build our model on the `sms_train` matrix, we'll use the following command:

```
> sms_classifier <- naiveBayes(sms_train, sms_train_labels)
```

The `sms_classifier` object now contains a `naiveBayes` classifier object that can be used to make predictions.

Step 4 – evaluating model performance

To evaluate the SMS classifier, we need to test its predictions on unseen messages in the test data. Recall that the unseen message features are stored in a matrix named `sms_test`, while the class labels (spam or ham) are stored in a vector named `sms_test_labels`. The classifier that we trained has been named `sms_classifier`. We will use this classifier to generate predictions and then compare the predicted values to the true values.

The `predict()` function is used to make the predictions. We will store these in a vector named `sms_test_pred`. We will simply supply the function with the names of our classifier and test dataset, as shown:

```
> sms_test_pred <- predict(sms_classifier, sms_test)
```

To compare the predictions to the true values, we'll use the `CrossTable()` function in the `gmodels` package, which we used previously. This time, we'll add some additional parameters to eliminate unnecessary cell proportions and use the `dnn` parameter (dimension names) to relabel the rows and columns, as shown in the following code:

```
> library(gmodels)
> CrossTable(sms_test_pred, sms_test_labels,
    prop.chisq = FALSE, prop.t = FALSE,
    dnn = c('predicted', 'actual'))
```

This produces the following table:

```
Total Observations in Table:  1390

             | actual
   predicted |       ham |      spam | Row Total |
-------------|-----------|-----------|-----------|
         ham |      1201 |        30 |      1231 |
             |     0.995 |     0.164 |           |
-------------|-----------|-----------|-----------|
        spam |         6 |       153 |       159 |
             |     0.005 |     0.836 |           |
-------------|-----------|-----------|-----------|
Column Total |      1207 |       183 |      1390 |
             |     0.868 |     0.132 |           |
-------------|-----------|-----------|-----------|
```

Looking at the table, we can see that a total of only *6 + 30* = 36 of the 1,390 SMS messages were incorrectly classified (2.6 percent). Among the errors were 6 out of 1,207 ham messages that were misidentified as spam, and 30 of the 183 spam messages were incorrectly labeled as ham. Considering the little effort we put into the project, this level of performance seems quite impressive. This case study exemplifies the reason why Naive Bayes is the standard for text classification; directly out of the box, it performs surprisingly well.

On the other hand, the six legitimate messages that were incorrectly classified as spam could cause significant problems for the deployment of our filtering algorithm, because the filter could cause a person to miss an important text message. We should investigate to see whether we can slightly tweak the model to achieve better performance.

Step 5 – improving model performance

You may have noticed that we didn't set a value for the Laplace estimator while training our model. This allows words that appeared in zero spam or zero ham messages to have an indisputable say in the classification process. Just because the word "ringtone" only appeared in the spam messages in the training data, it does not mean that every message with this word should be classified as spam.

We'll build a Naive Bayes model as done earlier, but this time set `laplace = 1`:

```
> sms_classifier2 <- naiveBayes(sms_train, sms_train_labels,
    laplace = 1)
```

Next, we'll make predictions:

```
> sms_test_pred2 <- predict(sms_classifier2, sms_test)
```

Finally, we'll compare the predicted classes to the actual classifications using a cross tabulation:

```
> CrossTable(sms_test_pred2, sms_test_labels,
    prop.chisq = FALSE, prop.t = FALSE, prop.r = FALSE,
    dnn = c('predicted', 'actual'))
```

This results in the following table:

```
Total Observations in Table:  1390

             | actual
   predicted |       ham |      spam | Row Total |
-------------|-----------|-----------|-----------|
         ham |      1202 |        28 |      1230 |
             |     0.996 |     0.153 |           |
-------------|-----------|-----------|-----------|
        spam |         5 |       155 |       160 |
             |     0.004 |     0.847 |           |
-------------|-----------|-----------|-----------|
Column Total |      1207 |       183 |      1390 |
             |     0.868 |     0.132 |           |
-------------|-----------|-----------|-----------|
```

Adding the Laplace estimator reduced the number of false positives (ham messages erroneously classified as spam) from six to five and the number of false negatives from 30 to 28. Although this seems like a small change, it's substantial considering that the model's accuracy was already quite impressive. We'd need to be careful before tweaking the model too much in order to maintain the balance between being overly aggressive and overly passive while filtering spam. Users would prefer that a small number of spam messages slip through the filter than an alternative in which ham messages are filtered too aggressively.

Summary

In this chapter, we learned about classification using Naive Bayes. This algorithm constructs tables of probabilities that are used to estimate the likelihood that new examples belong to various classes. The probabilities are calculated using a formula known as Bayes' theorem, which specifies how dependent events are related. Although Bayes' theorem can be computationally expensive, a simplified version that makes so-called "naive" assumptions about the independence of features is capable of handling extremely large datasets.

The Naive Bayes classifier is often used for text classification. To illustrate its effectiveness, we employed Naive Bayes on a classification task involving spam SMS messages. Preparing the text data for analysis required the use of specialized R packages for text processing and visualization. Ultimately, the model was able to classify over 97 percent of all the SMS messages correctly as spam or ham.

In the next chapter, we will examine two more machine learning methods. Each performs classification by partitioning data into groups of similar values.

5

Divide and Conquer – Classification Using Decision Trees and Rules

While deciding between several job offers with various levels of pay and benefits, many people begin by making lists of pros and cons, and eliminate options based on simple rules. For instance, "if I have to commute for more than an hour, I will be unhappy." Or, "if I make less than $50k, I won't be able to support my family." In this way, the complex and difficult decision of predicting one's future happiness can be reduced to a series of simple decisions.

This chapter covers decision trees and rule learners—two machine learning methods that also make complex decisions from sets of simple choices. These methods then present their knowledge in the form of logical structures that can be understood with no statistical knowledge. This aspect makes these models particularly useful for business strategy and process improvement.

By the end of this chapter, you will learn:

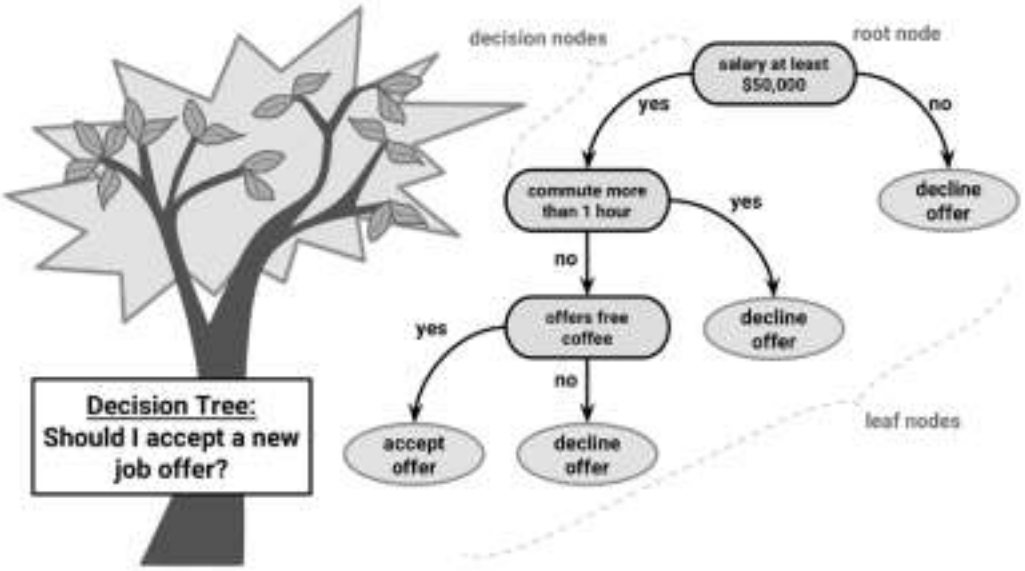
- How trees and rules "greedily" partition data into interesting segments
- The most common decision tree and classification rule learners, including the C5.0, 1R, and RIPPER algorithms
- How to use these algorithms to perform real-world classification tasks, such as identifying risky bank loans and poisonous mushrooms

We will begin by examining decision trees, followed by a look at classification rules. Then, we will summarize what we've learned by previewing later chapters, which discuss methods that use trees and rules as a foundation for more advanced machine learning techniques.

Understanding decision trees

Decision tree learners are powerful classifiers, which utilize a **tree structure** to model the relationships among the features and the potential outcomes. As illustrated in the following figure, this structure earned its name due to the fact that it mirrors how a literal tree begins at a wide trunk, which if followed upward, splits into narrower and narrower branches. In much the same way, a decision tree classifier uses a structure of branching decisions, which channel examples into a final predicted class value.

To better understand how this works in practice, let's consider the following tree, which predicts whether a job offer should be accepted. A job offer to be considered begins at the **root node**, where it is then passed through **decision nodes** that require choices to be made based on the attributes of the job. These choices split the data across **branches** that indicate potential outcomes of a decision, depicted here as yes or no outcomes, though in some cases there may be more than two possibilities.
In the case a final decision can be made, the tree is terminated by **leaf nodes** (also known as **terminal nodes**) that denote the action to be taken as the result of the series of decisions. In the case of a predictive model, the leaf nodes provide the expected result given the series of events in the tree.



A great benefit of decision tree algorithms is that the flowchart-like tree structure is not necessarily exclusively for the learner's internal use. After the model is created, many decision tree algorithms output the resulting structure in a human-readable format. This provides tremendous insight into how and why the model works or doesn't work well for a particular task. This also makes decision trees particularly appropriate for applications in which the classification mechanism needs to be transparent for legal reasons, or in case the results need to be shared with others in order to inform future business practices. With this in mind, some potential uses include:

- Credit scoring models in which the criteria that causes an applicant to be rejected need to be clearly documented and free from bias
- Marketing studies of customer behavior such as satisfaction or churn, which will be shared with management or advertising agencies
- Diagnosis of medical conditions based on laboratory measurements, symptoms, or the rate of disease progression

Although the previous applications illustrate the value of trees in informing decision processes, this is not to suggest that their utility ends here. In fact, decision trees are perhaps the single most widely used machine learning technique, and can be applied to model almost any type of data—often with excellent out-of-the-box applications.

This said, in spite of their wide applicability, it is worth noting some scenarios where trees may not be an ideal fit. One such case might be a task where the data has a large number of nominal features with many levels or it has a large number of numeric features. These cases may result in a very large number of decisions and an overly complex tree. They may also contribute to the tendency of decision trees to overfit data, though as we will soon see, even this weakness can be overcome by adjusting some simple parameters.

Divide and conquer

Decision trees are built using a heuristic called **recursive partitioning**. This approach is also commonly known as **divide and conquer** because it splits the data into subsets, which are then split repeatedly into even smaller subsets, and so on and so forth until the process stops when the algorithm determines the data within the subsets are sufficiently homogenous, or another stopping criterion has been met.

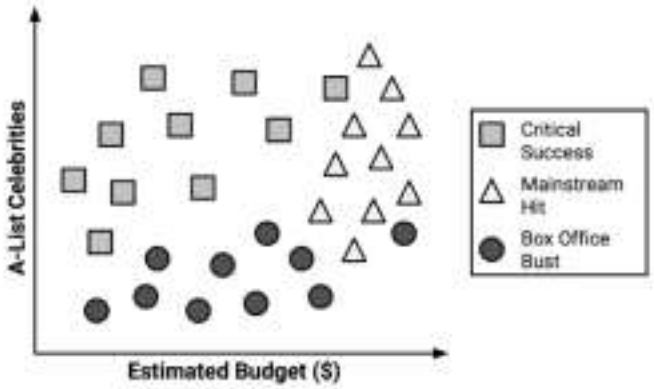
To see how splitting a dataset can create a decision tree, imagine a bare root node that will grow into a mature tree. At first, the root node represents the entire dataset, since no splitting has transpired. Next, the decision tree algorithm must choose a feature to split upon; ideally, it chooses the feature most predictive of the target class. The examples are then partitioned into groups according to the distinct values of this feature, and the first set of tree branches are formed.

Working down each branch, the algorithm continues to divide and conquer the data, choosing the best candidate feature each time to create another decision node, until a stopping criterion is reached. Divide and conquer might stop at a node in a case that:

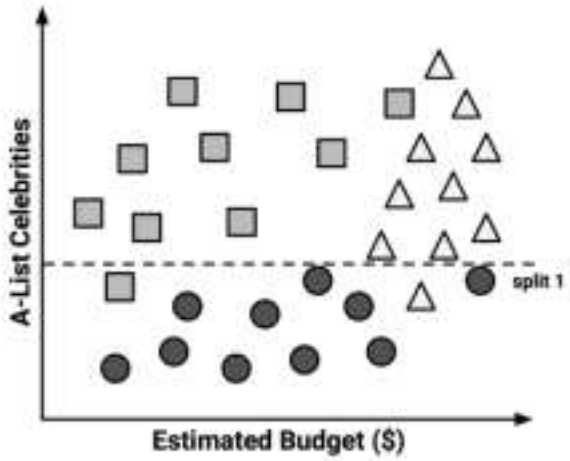
- All (or nearly all) of the examples at the node have the same class
- There are no remaining features to distinguish among the examples
- The tree has grown to a predefined size limit

To illustrate the tree building process, let's consider a simple example. Imagine that you work for a Hollywood studio, where your role is to decide whether the studio should move forward with producing the screenplays pitched by promising new authors. After returning from a vacation, your desk is piled high with proposals. Without the time to read each proposal cover-to-cover, you decide to develop a decision tree algorithm to predict whether a potential movie would fall into one of three categories: **Critical Success**, **Mainstream Hit**, or **Box Office Bust**.

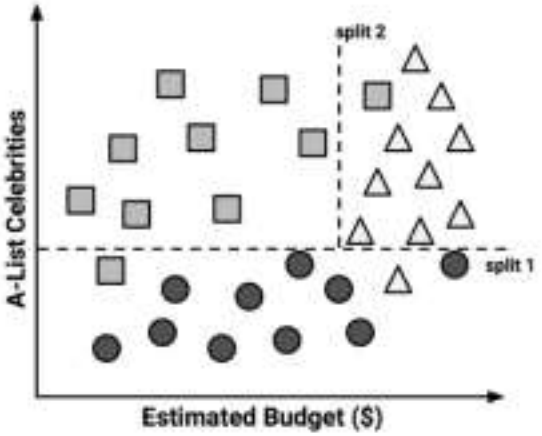
To build the decision tree, you turn to the studio archives to examine the factors leading to the success and failure of the company's 30 most recent releases. You quickly notice a relationship between the film's estimated shooting budget, the number of A-list celebrities lined up for starring roles, and the level of success. Excited about this finding, you produce a scatterplot to illustrate the pattern:



Using the divide and conquer strategy, we can build a simple decision tree from this data. First, to create the tree's root node, we split the feature indicating the number of celebrities, partitioning the movies into groups with and without a significant number of A-list stars:



Next, among the group of movies with a larger number of celebrities, we can make another split between movies with and without a high budget:

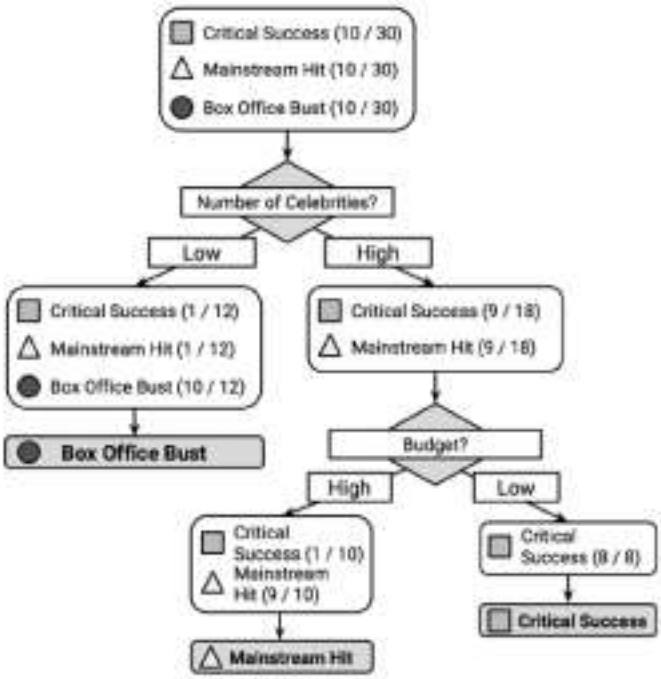


At this point, we have partitioned the data into three groups. The group at the top-left corner of the diagram is composed entirely of critically acclaimed films. This group is distinguished by a high number of celebrities and a relatively low budget. At the top-right corner, majority of movies are box office hits with high budgets and a large number of celebrities. The final group, which has little star power but budgets ranging from small to large, contains the flops.

If we wanted, we could continue to divide and conquer the data by splitting it based on the increasingly specific ranges of budget and celebrity count, until each of the currently misclassified values resides in its own tiny partition, and is correctly classified. However, it is not advisable to overfit a decision tree in this way. Though there is nothing to stop us from splitting the data indefinitely, overly specific decisions do not always generalize more broadly. We'll avoid the problem of overfitting by stopping the algorithm here, since more than 80 percent of the examples in each group are from a single class. This forms the basis of our stopping criterion.

You might have noticed that diagonal lines might have split the data even more cleanly. This is one limitation of the decision tree's knowledge representation, which uses **axis-parallel splits**. The fact that each split considers one feature at a time prevents the decision tree from forming more complex decision boundaries. For example, a diagonal line could be created by a decision that asks, "is the number of celebrities is greater than the estimated budget?" If so, then "it will be a critical success."

Our model for predicting the future success of movies can be represented in a simple tree, as shown in the following diagram. To evaluate a script, follow the branches through each decision until the script's success or failure has been predicted. In no time, you will be able to identify the most promising options among the backlog of scripts and get back to more important work, such as writing an Academy Awards acceptance speech.



Since real-world data contains more than two features, decision trees quickly become far more complex than this, with many more nodes, branches, and leaves. In the next section, you will learn about a popular algorithm to build decision tree models automatically.

The C5.0 decision tree algorithm

There are numerous implementations of decision trees, but one of the most well-known implementations is the **C5.0 algorithm**. This algorithm was developed by computer scientist J. Ross Quinlan as an improved version of his prior algorithm, **C4.5**, which itself is an improvement over his **Iterative Dichotomiser 3** (**ID3**) algorithm. Although Quinlan markets C5.0 to commercial clients (see `http://www.rulequest.com/` for details), the source code for a single-threaded version of the algorithm was made publically available, and it has therefore been incorporated into programs such as R.

To further confuse matters, a popular Java-based open source alternative to C4.5, titled **J48**, is included in R's `RWeka` package. Because the differences among C5.0, C4.5, and J48 are minor, the principles in this chapter will apply to any of these three methods, and the algorithms should be considered synonymous.

The C5.0 algorithm has become the industry standard to produce decision trees, because it does well for most types of problems directly out of the box. Compared to other advanced machine learning models, such as those described in *Chapter 7, Black Box Methods – Neural Networks and Support Vector Machines*, the decision trees built by C5.0 generally perform nearly as well, but are much easier to understand and deploy. Additionally, as shown in the following table, the algorithm's weaknesses are relatively minor and can be largely avoided:

Strengths	Weaknesses
• An all-purpose classifier that does well on most problems • Highly automatic learning process, which can handle numeric or nominal features, as well as missing data • Excludes unimportant features • Can be used on both small and large datasets • Results in a model that can be interpreted without a mathematical background (for relatively small trees) • More efficient than other complex models	• Decision tree models are often biased toward splits on features having a large number of levels • It is easy to overfit or underfit the model • Can have trouble modeling some relationships due to reliance on axis-parallel splits • Small changes in the training data can result in large changes to decision logic • Large trees can be difficult to interpret and the decisions they make may seem counterintuitive

To keep things simple, our earlier decision tree example ignored the mathematics involved in how a machine would employ a divide and conquer strategy. Let's explore this in more detail to examine how this heuristic works in practice.

Choosing the best split

The first challenge that a decision tree will face is to identify which feature to split upon. In the previous example, we looked for a way to split the data such that the resulting partitions contained examples primarily of a single class. The degree to which a subset of examples contains only a single class is known as **purity**, and any subset composed of only a single class is called **pure**.

There are various measurements of purity that can be used to identify the best decision tree splitting candidate. C5.0 uses **entropy**, a concept borrowed from information theory that quantifies the randomness, or disorder, within a set of class values. Sets with high entropy are very diverse and provide little information about other items that may also belong in the set, as there is no apparent commonality. The decision tree hopes to find splits that reduce entropy, ultimately increasing homogeneity within the groups.

Typically, entropy is measured in **bits**. If there are only two possible classes, entropy values can range from 0 to 1. For n classes, entropy ranges from 0 to $log_2(n)$. In each case, the minimum value indicates that the sample is completely homogenous, while the maximum value indicates that the data are as diverse as possible, and no group has even a small plurality.

In the mathematical notion, entropy is specified as follows:

$$\text{Entropy}(S) = \sum_{i=1}^{c} -p_i \log_2(p_i)$$

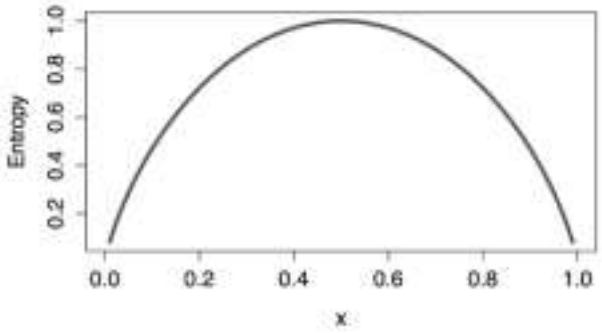
In this formula, for a given segment of data (S), the term c refers to the number of class levels and p_i refers to the proportion of values falling into class level i. For example, suppose we have a partition of data with two classes: red (60 percent) and white (40 percent). We can calculate the entropy as follows:

```
> -0.60 * log2(0.60) - 0.40 * log2(0.40)
[1] 0.9709506
```

We can examine the entropy for all the possible two-class arrangements. If we know that the proportion of examples in one class is x, then the proportion in the other class is $(1 - x)$. Using the `curve()` function, we can then plot the entropy for all the possible values of x:

```
> curve(-x * log2(x) - (1 - x) * log2(1 - x),
        col = "red", xlab = "x", ylab = "Entropy", lwd = 4)
```

This results in the following figure:



As illustrated by the peak in entropy at $x = 0.50$, a 50-50 split results in maximum entropy. As one class increasingly dominates the other, the entropy reduces to zero.

To use entropy to determine the optimal feature to split upon, the algorithm calculates the change in homogeneity that would result from a split on each possible feature, which is a measure known as **information gain**. The information gain for a feature *F* is calculated as the difference between the entropy in the segment before the split *(S_1)* and the partitions resulting from the split *(S_2)*:

$$\text{InfoGain}(F) = \text{Entropy}(S_1) - \text{Entropy}(S_2)$$

One complication is that after a split, the data is divided into more than one partition. Therefore, the function to calculate *Entropy(S_2)* needs to consider the total entropy across all of the partitions. It does this by weighing each partition's entropy by the proportion of records falling into the partition. This can be stated in a formula as:

$$\text{Entropy}(S) = \sum_{i=1}^{n} w_i \, \text{Entropy}(P_i)$$

In simple terms, the total entropy resulting from a split is the sum of the entropy of each of the *n* partitions weighted by the proportion of examples falling in the partition (w_i).

The higher the information gain, the better a feature is at creating homogeneous groups after a split on this feature. If the information gain is zero, there is no reduction in entropy for splitting on this feature. On the other hand, the maximum information gain is equal to the entropy prior to the split. This would imply that the entropy after the split is zero, which means that the split results in completely homogeneous groups.

The previous formulae assume nominal features, but decision trees use information gain for splitting on numeric features as well. To do so, a common practice is to test various splits that divide the values into groups greater than or less than a numeric threshold. This reduces the numeric feature into a two-level categorical feature that allows information gain to be calculated as usual. The numeric cut point yielding the largest information gain is chosen for the split.

Though it is used by C5.0, information gain is not the only splitting criterion that can be used to build decision trees. Other commonly used criteria are **Gini index**, **Chi-Squared statistic**, and **gain ratio**. For a review of these (and many more) criteria, refer to Mingers J. *An Empirical Comparison of Selection Measures for Decision-Tree Induction*. Machine Learning. 1989; 3:319-342.

Pruning the decision tree

A decision tree can continue to grow indefinitely, choosing splitting features and dividing the data into smaller and smaller partitions until each example is perfectly classified or the algorithm runs out of features to split on. However, if the tree grows overly large, many of the decisions it makes will be overly specific and the model will be overfitted to the training data. The process of **pruning** a decision tree involves reducing its size such that it generalizes better to unseen data.

One solution to this problem is to stop the tree from growing once it reaches a certain number of decisions or when the decision nodes contain only a small number of examples. This is called **early stopping** or **pre-pruning** the decision tree. As the tree avoids doing needless work, this is an appealing strategy. However, one downside to this approach is that there is no way to know whether the tree will miss subtle, but important patterns that it would have learned had it grown to a larger size.

An alternative, called **post-pruning**, involves growing a tree that is intentionally too large and pruning leaf nodes to reduce the size of the tree to a more appropriate level. This is often a more effective approach than pre-pruning, because it is quite difficult to determine the optimal depth of a decision tree without growing it first. Pruning the tree later on allows the algorithm to be certain that all the important data structures were discovered.

The implementation details of pruning operations are very technical and beyond the scope of this book. For a comparison of some of the available methods, see Esposito F, Malerba D, Semeraro G. *A Comparative Analysis of Methods for Pruning Decision Trees*. IEEE Transactions on Pattern Analysis and Machine Intelligence. 1997;19: 476-491.

One of the benefits of the C5.0 algorithm is that it is opinionated about pruning—it takes care of many decisions automatically using fairly reasonable defaults. Its overall strategy is to post-prune the tree. It first grows a large tree that overfits the training data. Later, the nodes and branches that have little effect on the classification errors are removed. In some cases, entire branches are moved further up the tree or replaced by simpler decisions. These processes of grafting branches are known as **subtree raising** and **subtree replacement**, respectively.

Balancing overfitting and underfitting a decision tree is a bit of an art, but if model accuracy is vital, it may be worth investing some time with various pruning options to see if it improves the performance on test data. As you will soon see, one of the strengths of the C5.0 algorithm is that it is very easy to adjust the training options.

Example – identifying risky bank loans using C5.0 decision trees

The global financial crisis of 2007-2008 highlighted the importance of transparency and rigor in banking practices. As the availability of credit was limited, banks tightened their lending systems and turned to machine learning to more accurately identify risky loans.

Decision trees are widely used in the banking industry due to their high accuracy and ability to formulate a statistical model in plain language. Since government organizations in many countries carefully monitor lending practices, executives must be able to explain why one applicant was rejected for a loan while the others were approved. This information is also useful for customers hoping to determine why their credit rating is unsatisfactory.

It is likely that automated credit scoring models are employed to instantly approve credit applications on the telephone and web. In this section, we will develop a simple credit approval model using C5.0 decision trees. We will also see how the results of the model can be tuned to minimize errors that result in a financial loss for the institution.

Step 1 – collecting data

The idea behind our credit model is to identify factors that are predictive of higher risk of default. Therefore, we need to obtain data on a large number of past bank loans and whether the loan went into default, as well as information on the applicant.

Data with these characteristics is available in a dataset donated to the UCI Machine Learning Data Repository (`http://archive.ics.uci.edu/ml`) by Hans Hofmann of the University of Hamburg. The dataset contains information on loans obtained from a credit agency in Germany.

The dataset presented in this chapter has been modified slightly from the original in order to eliminate some preprocessing steps. To follow along with the examples, download the `credit.csv` file from Packt Publishing's website and save it to your R working directory.

The credit dataset includes 1,000 examples on loans, plus a set of numeric and nominal features indicating the characteristics of the loan and the loan applicant. A class variable indicates whether the loan went into default. Let's see whether we can determine any patterns that predict this outcome.

Step 2 – exploring and preparing the data

As we did previously, we will import data using the `read.csv()` function. We will ignore the `stringsAsFactors` option and, therefore, use the default value of `TRUE`, as the majority of the features in the data are nominal:

```
> credit <- read.csv("credit.csv")
```

The first several lines of output from the `str()` function are as follows:

```
> str(credit)
'data.frame':1000 obs. of  17 variables:
 $ checking_balance : Factor w/ 4 levels "< 0 DM","> 200 DM",..
 $ months_loan_duration: int  6 48 12 ...
 $ credit_history     : Factor w/ 5 levels "critical","good",..
 $ purpose            : Factor w/ 6 levels "business","car",..
 $ amount             : int  1169 5951 2096 ...
```

We see the expected 1,000 observations and 17 features, which are a combination of factor and integer data types.

Let's take a look at the `table()` output for a couple of loan features that seem likely to predict a default. The applicant's checking and savings account balance are recorded as categorical variables:

```
> table(credit$checking_balance)
    < 0 DM    > 200 DM 1 - 200 DM    unknown
```

```
       274          63         269         394
> table(credit$savings_balance)
     < 100 DM > 1000 DM  100 - 500 DM 500 - 1000 DM   unknown
           603         48         103          63         183
```

The checking and savings account balance may prove to be important predictors of loan default status. Note that since the loan data was obtained from Germany, the currency is recorded in Deutsche Marks (DM).

Some of the loan's features are numeric, such as its duration and the amount of credit requested:

```
> summary(credit$months_loan_duration)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    4.0    12.0    18.0    20.9    24.0    72.0
> summary(credit$amount)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    250    1366    2320    3271    3972   18420
```

The loan amounts ranged from 250 DM to 18,420 DM across terms of 4 to 72 months with a median duration of 18 months and an amount of 2,320 DM.

The `default` vector indicates whether the loan applicant was unable to meet the agreed payment terms and went into default. A total of 30 percent of the loans in this dataset went into default:

```
> table(credit$default)
 no yes
700 300
```

A high rate of default is undesirable for a bank, because it means that the bank is unlikely to fully recover its investment. If we are successful, our model will identify applicants that are at high risk to default, allowing the bank to refuse credit requests.

Data preparation – creating random training and test datasets

As we have done in the previous chapters, we will split our data into two portions: a training dataset to build the decision tree and a test dataset to evaluate the performance of the model on new data. We will use 90 percent of the data for training and 10 percent for testing, which will provide us with 100 records to simulate new applicants.

As prior chapters used data that had been sorted in a random order, we simply divided the dataset into two portions, by taking the first 90 percent of records for training, and the remaining 10 percent for testing. In contrast, the credit dataset is not randomly ordered, making the prior approach unwise. Suppose that the bank had sorted the data by the loan amount, with the largest loans at the end of the file. If we used the first 90 percent for training and the remaining 10 percent for testing, we would be training a model on only the small loans and testing the model on the big loans. Obviously, this could be problematic.

We'll solve this problem by using a **random sample** of the credit data for training. A random sample is simply a process that selects a subset of records at random. In R, the `sample()` function is used to perform random sampling. However, before putting it in action, a common practice is to set a **seed** value, which causes the randomization process to follow a sequence that can be replicated later on if desired. It may seem that this defeats the purpose of generating random numbers, but there is a good reason for doing it this way. Providing a seed value via the `set.seed()` function ensures that if the analysis is repeated in the future, an identical result is obtained.

You may wonder how a so-called random process can be seeded to produce an identical result. This is due to the fact that computers use a mathematical function called a **pseudorandom number generator** to create random number sequences that appear to act very random, but are actually quite predictable given knowledge of the previous values in the sequence. In practice, modern pseudorandom number sequences are virtually indistinguishable from true random sequences, but have the benefit that computers can generate them quickly and easily.

The following commands use the `sample()` function to select 900 values at random out of the sequence of integers from 1 to 1000. Note that the `set.seed()` function uses the arbitrary value `123`. Omitting this seed will cause your training and testing split to differ from those shown in the remainder of this chapter:

```
> set.seed(123)
> train_sample <- sample(1000, 900)
```

As expected, the resulting `train_sample` object is a vector of 900 random integers:

```
> str(train_sample)
 int [1:900] 288 788 409 881 937 46 525 887 548 453 ...
```

By using this vector to select rows from the credit data, we can split it into the 90 percent training and 10 percent test datasets we desired. Recall that the dash operator used in the selection of the test records tells R to select records that are not in the specified rows; in other words, the test data includes only the rows that are not in the training sample.

```
> credit_train <- credit[train_sample, ]
> credit_test  <- credit[-train_sample, ]
```

If all went well, we should have about 30 percent of defaulted loans in each of the datasets:

```
> prop.table(table(credit_train$default))

       no       yes
0.7033333 0.2966667


> prop.table(table(credit_test$default))

  no  yes
0.67 0.33
```

This appears to be a fairly even split, so we can now build our decision tree.

If your results do not match exactly, ensure that you ran the command `set.seed(123)` immediately prior to creating the `train_sample` vector.

Step 3 – training a model on the data

We will use the C5.0 algorithm in the `C50` package to train our decision tree model. If you have not done so already, install the package with `install.packages("C50")` and load it to your R session, using `library(C50)`.

The following syntax box lists some of the most commonly used commands to build decision trees. Compared to the machine learning approaches we used previously, the C5.0 algorithm offers many more ways to tailor the model to a particular learning problem, but more options are available. Once the `C50` package has been loaded, the `?C5.0Control` command displays the help page for more details on how to finely-tune the algorithm.

C5.0 decision tree syntax

using the `C5.0()` function in the `C50` package

Building the classifier:

```
m <- C5.0(train, class, trials = 1, costs = NULL)
```

- `train` is a data frame containing training data
- `class` is a factor vector with the class for each row in the training data
- `trials` is an optional number to control the number of boosting iterations (set to 1 by default)
- `costs` is an optional matrix specifying costs associated with various types of errors

The function will return a C5.0 model object that can be used to make predictions.

Making predictions:

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

- `m` is a model trained by the `C5.0()` 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 `"class"` or `"prob"` and specifies whether the predictions should be the most probable class value or the raw predicted probabilities

The function will return a vector of predicted class values or raw predicted probabilities depending upon the value of the `type` parameter.

Example:

```
credit_model <- C5.0(credit_train, loan_default)
credit_prediction <- predict(credit_model,
  credit_test)
```

For the first iteration of our credit approval model, we'll use the default C5.0 configuration, as shown in the following code. The 17th column in `credit_train` is the `default` class variable, so we need to exclude it from the training data frame, but supply it as the target factor vector for classification:

```
> credit_model <- C5.0(credit_train[-17], credit_train$default)
```

The `credit_model` object now contains a C5.0 decision tree. We can see some basic data about the tree by typing its name:

```
> credit_model

Call:
C5.0.default(x = credit_train[-17], y = credit_train$default)

Classification Tree
Number of samples: 900
```

```
Number of predictors: 16


Tree size: 57


Non-standard options: attempt to group attributes
```

The preceding text shows some simple facts about the tree, including the function call that generated it, the number of features (labeled `predictors`), and examples (labeled `samples`) used to grow the tree. Also listed is the tree size of 57, which indicates that the tree is 57 decisions deep—quite a bit larger than the example trees we've considered so far!

To see the tree's decisions, we can call the `summary()` function on the model:

```
> summary(credit_model)
```

This results in the following output:

```
C5.0 [Release 2.07 GPL Edition]
-------------------------------

Class specified by attribute `outcome'

Read 900 cases (17 attributes) from undefined.data

Decision tree:

checking_balance in {> 200 DM,unknown}: no (412/50)
checking_balance in {< 0 DM,1 - 200 DM}:
:...credit_history in {perfect,very good}: yes (59/18)
    credit_history in {critical,good,poor}:
    :...months_loan_duration <= 22:
        :...credit_history = critical: no (72/14)
        :   credit_history = poor:
        :   :...dependents > 1: no (5)
        :   :   dependents <= 1:
        :   :   :...years_at_residence <= 3: yes (4/1)
        :   :       years_at_residence > 3: no (5/1)
```

The preceding output shows some of the first branches in the decision tree. The first three lines could be represented in plain language as:

1. If the checking account balance is unknown or greater than 200 DM, then classify as "not likely to default."
2. Otherwise, if the checking account balance is less than zero DM or between one and 200 DM.
3. And the credit history is perfect or very good, then classify as "likely to default."

The numbers in parentheses indicate the number of examples meeting the criteria for that decision, and the number incorrectly classified by the decision. For instance, on the first line, `412/50` indicates that of the 412 examples reaching the decision, 50 were incorrectly classified as not likely to default. In other words, 50 applicants actually defaulted, in spite of the model's prediction to the contrary.

Sometimes a tree results in decisions that make little logical sense. For example, why would an applicant whose credit history is very good be likely to default, while those whose checking balance is unknown are not likely to default? Contradictory rules like this occur sometimes. They might reflect a real pattern in the data, or they may be a statistical anomaly. In either case, it is important to investigate such strange decisions to see whether the tree's logic makes sense for business use.

After the tree, the `summary(credit_model)` output displays a confusion matrix, which is a cross-tabulation that indicates the model's incorrectly classified records in the training data:

```
Evaluation on training data (900 cases):

     Decision Tree
   ----------------
   Size      Errors

     56  133(14.8%)   <<


    (a)   (b)    <-classified as
   ----  ----
    598    35    (a): class no
     98   169    (b): class yes
```

The Errors output notes that the model correctly classified all but 133 of the 900 training instances for an error rate of 14.8 percent. A total of 35 actual no values were incorrectly classified as yes (false positives), while 98 yes values were misclassified as no (false negatives).

Decision trees are known for having a tendency to overfit the model to the training data. For this reason, the error rate reported on training data may be overly optimistic, and it is especially important to evaluate decision trees on a test dataset.

Step 4 – evaluating model performance

To apply our decision tree to the test dataset, we use the `predict()` function, as shown in the following line of code:

```
> credit_pred <- predict(credit_model, credit_test)
```

This creates a vector of predicted class values, which we can compare to the actual class values using the `CrossTable()` function in the `gmodels` package. Setting the `prop.c` and `prop.r` parameters to `FALSE` removes the column and row percentages from the table. The remaining percentage (`prop.t`) indicates the proportion of records in the cell out of the total number of records:

```
> library(gmodels)
> CrossTable(credit_test$default, credit_pred,
             prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
             dnn = c('actual default', 'predicted default'))
```

This results in the following table:

```
                | predicted default
 actual default |        no |       yes | Row Total |
----------------|-----------|-----------|-----------|
             no |        59 |         8 |        67 |
                |     0.590 |     0.080 |           |
----------------|-----------|-----------|-----------|
            yes |        19 |        14 |        33 |
                |     0.190 |     0.140 |           |
----------------|-----------|-----------|-----------|
   Column Total |        78 |        22 |       100 |
----------------|-----------|-----------|-----------|
```

Out of the 100 test loan application records, our model correctly predicted that 59 did not default and 14 did default, resulting in an accuracy of 73 percent and an error rate of 27 percent. This is somewhat worse than its performance on the training data, but not unexpected, given that a model's performance is often worse on unseen data. Also note that the model only correctly predicted 14 of the 33 actual loan defaults in the test data, or 42 percent. Unfortunately, this type of error is a potentially very costly mistake, as the bank loses money on each default. Let's see if we can improve the result with a bit more effort.

Step 5 – improving model performance

Our model's error rate is likely to be too high to deploy it in a real-time credit scoring application. In fact, if the model had predicted "no default" for every test case, it would have been correct 67 percent of the time—a result not much worse than our model's, but requiring much less effort! Predicting loan defaults from 900 examples seems to be a challenging problem.

Making matters even worse, our model performed especially poorly at identifying applicants who do default on their loans. Luckily, there are a couple of simple ways to adjust the C5.0 algorithm that may help to improve the performance of the model, both overall and for the more costly type of mistakes.

Boosting the accuracy of decision trees

One way the C5.0 algorithm improved upon the C4.5 algorithm was through the addition of **adaptive boosting**. This is a process in which many decision trees are built and the trees vote on the best class for each example.

The idea of boosting is based largely upon the research by Rob Schapire and Yoav Freund. For more information, try searching the web for their publications or their recent textbook *Boosting: Foundations and Algorithms*. The MIT Press (2012).

As boosting can be applied more generally to any machine learning algorithm, it is covered in detail later in this book in *Chapter 11, Improving Model Performance*. For now, it suffices to say that boosting is rooted in the notion that by combining a number of weak performing learners, you can create a team that is much stronger than any of the learners alone. Each of the models has a unique set of strengths and weaknesses and they may be better or worse in solving certain problems. Using a combination of several learners with complementary strengths and weaknesses can therefore dramatically improve the accuracy of a classifier.

The `C5.0()` function makes it easy to add boosting to our C5.0 decision tree. We simply need to add an additional `trials` parameter indicating the number of separate decision trees to use in the boosted team. The `trials` parameter sets an upper limit; the algorithm will stop adding trees if it recognizes that additional trials do not seem to be improving the accuracy. We'll start with 10 trials, a number that has become the de facto standard, as research suggests that this reduces error rates on test data by about 25 percent:

```
> credit_boost10 <- C5.0(credit_train[-17], credit_train$default,
                         trials = 10)
```

While examining the resulting model, we can see that some additional lines have been added, indicating the changes:

```
> credit_boost10
Number of boosting iterations: 10
Average tree size: 47.5
```

Across the 10 iterations, our tree size shrunk. If you would like, you can see all 10 trees by typing `summary(credit_boost10)` at the command prompt. It also lists the model's performance on the training data:

```
> summary(credit_boost10)

     (a)   (b)    <-classified as
    ----  ----
     629     4    (a): class no
      30   237    (b): class yes
```

The classifier made 34 mistakes on 900 training examples for an error rate of 3.8 percent. This is quite an improvement over the 13.9 percent training error rate we noted before adding boosting! However, it remains to be seen whether we see a similar improvement on the test data. Let's take a look:

```
> credit_boost_pred10 <- predict(credit_boost10, credit_test)
> CrossTable(credit_test$default, credit_boost_pred10,
             prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
             dnn = c('actual default', 'predicted default'))
```

The resulting table is as follows:

	predicted default		
actual default	no	yes	Row Total
no	62	5	67
	0.620	0.050	
yes	13	20	33
	0.130	0.200	
Column Total	75	25	100

Here, we reduced the total error rate from 27 percent prior to boosting down to 18 percent in the boosted model. It does not seem like a large gain, but it is in fact larger than the 25 percent reduction we expected. On the other hand, the model is still not doing well at predicting defaults, predicting only *20/33 = 61%* correctly. The lack of an even greater improvement may be a function of our relatively small training dataset, or it may just be a very difficult problem to solve.

This said, if boosting can be added this easily, why not apply it by default to every decision tree? The reason is twofold. First, if building a decision tree once takes a great deal of computation time, building many trees may be computationally impractical. Secondly, if the training data is very noisy, then boosting might not result in an improvement at all. Still, if greater accuracy is needed, it's worth giving it a try.

Making mistakes more costlier than others

Giving a loan out to an applicant who is likely to default can be an expensive mistake. One solution to reduce the number of false negatives may be to reject a larger number of borderline applicants, under the assumption that the interest the bank would earn from a risky loan is far outweighed by the massive loss it would incur if the money is not paid back at all.

The C5.0 algorithm allows us to assign a penalty to different types of errors, in order to discourage a tree from making more costly mistakes. The penalties are designated in a **cost matrix**, which specifies how much costlier each error is, relative to any other prediction.

To begin constructing the cost matrix, we need to start by specifying the dimensions. Since the predicted and actual values can both take two values, `yes` or `no`, we need to describe a 2 x 2 matrix, using a list of two vectors, each with two values. At the same time, we'll also name the matrix dimensions to avoid confusion later on:

```
> matrix_dimensions <- list(c("no", "yes"), c("no", "yes"))
> names(matrix_dimensions) <- c("predicted", "actual")
```

Examining the new object shows that our dimensions have been set up correctly:

```
> matrix_dimensions
$predicted
[1] "no"  "yes"

$actual
[1] "no"  "yes"
```

Next, we need to assign the penalty for the various types of errors by supplying four values to fill the matrix. Since R fills a matrix by filling columns one by one from top to bottom, we need to supply the values in a specific order:

- Predicted no, actual no
- Predicted yes, actual no
- Predicted no, actual yes
- Predicted yes, actual yes

Suppose we believe that a loan default costs the bank four times as much as a missed opportunity. Our penalty values could then be defined as:

```
> error_cost <- matrix(c(0, 1, 4, 0), nrow = 2,
    dimnames = matrix_dimensions)
```

This creates the following matrix:

```
> error_cost
          actual
predicted no yes
      no   0   4
      yes  1   0
```

As defined by this matrix, there is no cost assigned when the algorithm classifies a no or yes correctly, but a false negative has a cost of 4 versus a false positive's cost of 1. To see how this impacts classification, let's apply it to our decision tree using the `costs` parameter of the `C5.0()` function. We'll otherwise use the same steps as we did earlier:

```
> credit_cost <- C5.0(credit_train[-17], credit_train$default,
                                costs = error_cost)
> credit_cost_pred <- predict(credit_cost, credit_test)
> CrossTable(credit_test$default, credit_cost_pred,
             prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
             dnn = c('actual default', 'predicted default'))
```

This produces the following confusion matrix:

	predicted default		
actual default	no	yes	Row Total
no	37	30	67
	0.370	0.300	
yes	7	26	33
	0.070	0.260	
Column Total	44	56	100

Compared to our boosted model, this version makes more mistakes overall: 37 percent error here versus 18 percent in the boosted case. However, the types of mistakes are very different. Where the previous models incorrectly classified only 42 and 61 percent of defaults correctly, in this model, 79 percent of the actual defaults were predicted to be non-defaults. This trade resulting in a reduction of false negatives at the expense of increasing false positives may be acceptable if our cost estimates were accurate.

Understanding classification rules

Classification rules represent knowledge in the form of logical if-else statements that assign a class to unlabeled examples. They are specified in terms of an **antecedent** and a **consequent**; these form a hypothesis stating that "if this happens, then that happens." A simple rule might state, "if the hard drive is making a clicking sound, then it is about to fail." The antecedent comprises certain combinations of feature values, while the consequent specifies the class value to assign when the rule's conditions are met.

Rule learners are often used in a manner similar to decision tree learners. Like decision trees, they can be used for applications that generate knowledge for future action, such as:

- Identifying conditions that lead to a hardware failure in mechanical devices
- Describing the key characteristics of groups of people for customer segmentation
- Finding conditions that precede large drops or increases in the prices of shares on the stock market

On the other hand, rule learners offer some distinct advantages over trees for some tasks. Unlike a tree, which must be applied from top-to-bottom through a series of decisions, rules are propositions that can be read much like a statement of fact. Additionally, for reasons that will be discussed later, the results of a rule learner can be more simple, direct, and easier to understand than a decision tree built on the same data.

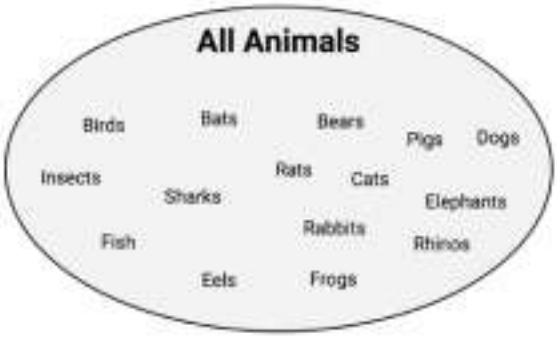
As you will see later in this chapter, rules can be generated using decision trees. So, why bother with a separate group of rule learning algorithms? The reason is that decision trees bring a particular set of biases to the task that a rule learner avoids by identifying the rules directly.

Rule learners are generally applied to problems where the features are primarily or entirely nominal. They do well at identifying rare events, even if the rare event occurs only for a very specific interaction among feature values.

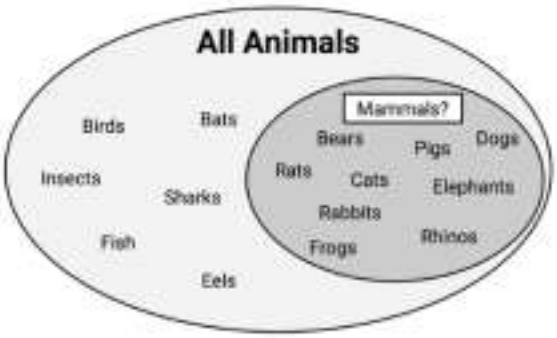
Separate and conquer

Classification rule learning algorithms utilize a heuristic known as **separate and conquer**. The process involves identifying a rule that covers a subset of examples in the training data, and then separating this partition from the remaining data. As the rules are added, additional subsets of the data are separated until the entire dataset has been covered and no more examples remain.

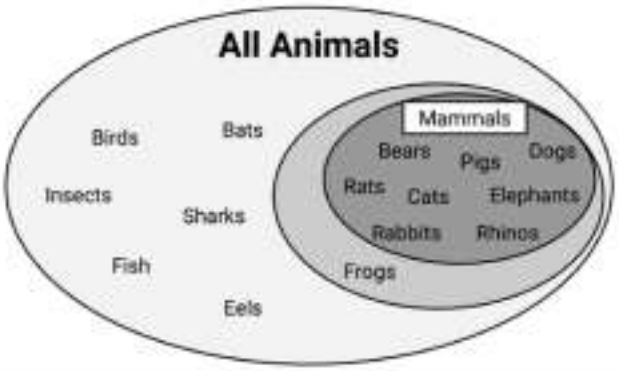
One way to imagine the rule learning process is to think about drilling down into the data by creating increasingly specific rules to identify class values. Suppose you were tasked with creating rules to identify whether or not an animal is a mammal. You could depict the set of all animals as a large space, as shown in the following diagram:



A rule learner begins by using the available features to find homogeneous groups. For example, using a feature that indicates whether the species travels via land, sea, or air, the first rule might suggest that any land-based animals are mammals:

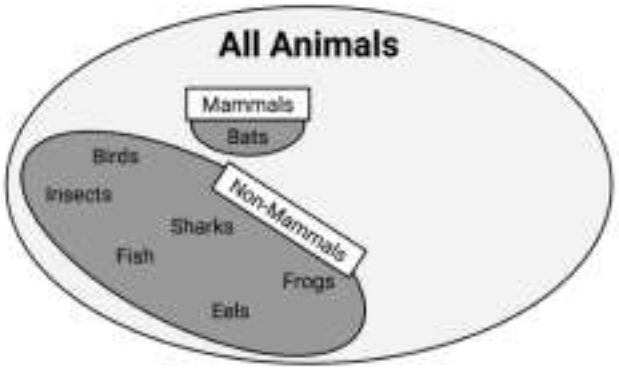


Do you notice any problems with this rule? If you're an animal lover, you might have realized that frogs are amphibians, not mammals. Therefore, our rule needs to be a bit more specific. Let's drill down further by suggesting that mammals must walk on land and have a tail:



An additional rule can be defined to separate out the bats, the only remaining mammal. Thus, this subset can be separated from the other data.

An additional rule can be defined to separate out the bats, the only remaining mammal. A potential feature distinguishing bats from the other remaining animals would be the presence of fur. Using a rule built around this feature, we have then correctly identified all the animals:



At this point, since all of the training instances have been classified, the rule learning process would stop. We learned a total of three rules:

- Animals that walk on land and have tails are mammals
- If the animal does not have fur, it is not a mammal
- Otherwise, the animal is a mammal

The previous example illustrates how rules gradually consume larger and larger segments of data to eventually classify all instances.

As the rules seem to cover portions of the data, separate and conquer algorithms are also known as **covering algorithms**, and the resulting rules are called covering rules. In the next section, we will learn how covering rules are applied in practice by examining a simple rule learning algorithm. We will then examine a more complex rule learner, and apply both to a real-world problem.

The 1R algorithm

Suppose a television game show has a wheel with ten evenly sized colored slices. Three of the segments were colored red, three were blue, and four were white. Prior to spinning the wheel, you are asked to choose one of these colors. When the wheel stops spinning, if the color shown matches your prediction, you will win a large cash prize. What color should you pick?

If you choose white, you are, of course, more likely to win the prize—this is the most common color on the wheel. Obviously, this game show is a bit ridiculous, but it demonstrates the simplest classifier, **ZeroR**, a rule learner that literally learns no rules (hence the name). For every unlabeled example, regardless of the values of its features, it predicts the most common class.

The **1R algorithm** (**One Rule** or **OneR**), improves over ZeroR by selecting a single rule. Although this may seem overly simplistic, it tends to perform better than you might expect. As demonstrated in empirical studies, the accuracy of this algorithm can approach that of much more sophisticated algorithms for many real-world tasks.

For an in-depth look at the surprising performance of 1R, see Holte RC. *Very simple classification rules perform well on most commonly used datasets.* Machine Learning. 1993; 11:63-91.

The strengths and weaknesses of the 1R algorithm are shown in the following table:

Strengths	Weaknesses
• Generates a single, easy-to-understand, human-readable rule of thumb • Often performs surprisingly well • Can serve as a benchmark for more complex algorithms	• Uses only a single feature • Probably overly simplistic

The way this algorithm works is simple. For each feature, 1R divides the data into groups based on similar values of the feature. Then, for each segment, the algorithm predicts the majority class. The error rate for the rule based on each feature is calculated and the rule with the fewest errors is chosen as the one rule.

The following tables show how this would work for the animal data we looked at earlier in this section:

Animal	Travels By	Has Fur	Mammal
Bats	Air	Yes	Yes
Bears	Land	Yes	Yes
Birds	Air	No	No
Cats	Land	Yes	Yes
Dogs	Land	Yes	Yes
Eels	Sea	No	No
Elephants	Land	No	Yes
Fish	Sea	No	No
Frogs	Land	No	No
Insects	Air	No	No
Pigs	Land	No	Yes
Rabbits	Land	Yes	Yes
Rats	Land	Yes	Yes
Rhinos	Land	No	Yes
Sharks	Sea	No	No

Full Dataset

Travels By	Predicted	Mammal	
Air	No	Yes	✖
Air	No	No	
Air	No	No	
Land	Yes	Yes	
Land	Yes	Yes	
Land	Yes	Yes	
Land	Yes	Yes	
Land	Yes	No	✖
Land	Yes	Yes	
Land	Yes	Yes	
Land	Yes	Yes	
Land	Yes	Yes	
Sea	No	No	
Sea	No	No	
Sea	No	No	

Rule for "Travels By"
Error Rate = 2 / 15

Has Fur	Predicted	Mammal	
No	No	No	
No	No	No	
No	No	Yes	✖
No	No	No	
No	No	No	
No	No	No	
No	No	Yes	✖
No	No	Yes	✖
No	No	No	
Yes	Yes	Yes	
Yes	Yes	Yes	
Yes	Yes	Yes	
Yes	Yes	Yes	
Yes	Yes	Yes	
Yes	Yes	Yes	

Rule for "Has Fur"
Error Rate = 3 / 15

For the **Travels By** feature, the dataset was divided into three groups: **Air**, **Land**, and **Sea**. Animals in the **Air** and **Sea** groups were predicted to be non-mammal, while animals in the **Land** group were predicted to be mammals. This resulted in two errors: bats and frogs. The **Has Fur** feature divided animals into two groups. Those with fur were predicted to be mammals, while those without fur were not predicted to be mammals. Three errors were counted: pigs, elephants, and rhinos. As the **Travels By** feature results in fewer errors, the 1R algorithm will return the following "one rule" based on **Travels By**:

- If the animal travels by air, it is not a mammal
- If the animal travels by land, it is a mammal
- If the animal travels by sea, it is not a mammal

The algorithm stops here, having found the single most important rule.

Obviously, this rule learning algorithm may be too basic for some tasks. Would you want a medical diagnosis system to consider only a single symptom, or an automated driving system to stop or accelerate your car based on only a single factor? For these types of tasks, a more sophisticated rule learner might be useful. We'll learn about one in the following section.

The RIPPER algorithm

Early rule learning algorithms were plagued by a couple of problems. First, they were notorious for being slow, which made them ineffective for the increasing number of large datasets. Secondly, they were often prone to being inaccurate on noisy data.

A first step toward solving these problems was proposed in 1994 by Johannes Furnkranz and Gerhard Widmer. Their **Incremental Reduced Error Pruning (IREP) algorithm** uses a combination of pre-pruning and post-pruning methods that grow very complex rules and prune them before separating the instances from the full dataset. Although this strategy helped the performance of rule learners, decision trees often still performed better.

For more information on IREP, see Furnkranz J, Widmer G. *Incremental Reduced Error Pruning*. Proceedings of the 11th International Conference on Machine Learning. 1994: 70-77.

Rule learners took another step forward in 1995 when William W. Cohen introduced the **Repeated Incremental Pruning to Produce Error Reduction (RIPPER) algorithm**, which improved upon IREP to generate rules that match or exceed the performance of decision trees.

For more detail on RIPPER, see Cohen WW. *Fast effective rule induction*. Proceedings of the 12th International Conference on Machine Learning. 1995:115-123.

As outlined in the following table, the strengths and weaknesses of RIPPER are generally comparable to decision trees. The chief benefit is that they may result in a slightly more parsimonious model:

Strengths	Weaknesses
• Generates easy-to-understand, human-readable rules • Efficient on large and noisy datasets • Generally produces a simpler model than a comparable decision tree	• May result in rules that seem to defy common sense or expert knowledge • Not ideal for working with numeric data • Might not perform as well as more complex models

Having evolved from several iterations of rule learning algorithms, the RIPPER algorithm is a patchwork of efficient heuristics for rule learning. Due to its complexity, a discussion of the technical implementation details is beyond the scope of this book. However, it can be understood in general terms as a three-step process:

1. Grow
2. Prune
3. Optimize

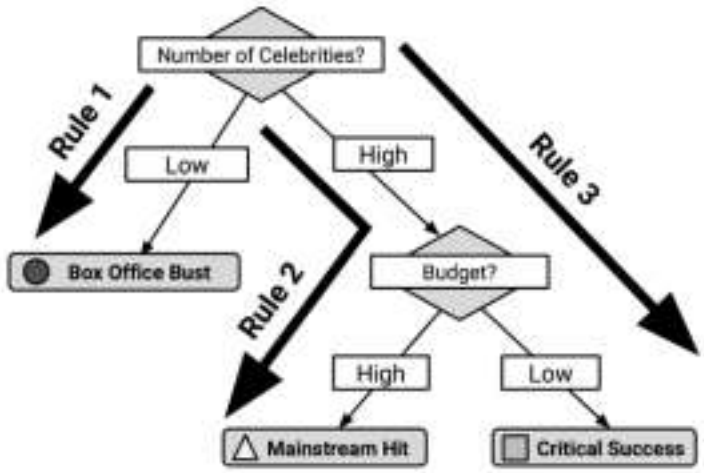
The growing phase uses the separate and conquer technique to greedily add conditions to a rule until it perfectly classifies a subset of data or runs out of attributes for splitting. Similar to decision trees, the information gain criterion is used to identify the next splitting attribute. When increasing a rule's specificity no longer reduces entropy, the rule is immediately pruned. Steps one and two are repeated until it reaches a stopping criterion, at which point the entire set of rules is optimized using a variety of heuristics.

The RIPPER algorithm can create much more complex rules than can the 1R algorithm, as in can consider more than one feature. This means that it can create rules with multiple antecedents such as "if an animal flies and has fur, then it is a mammal." This improves the algorithm's ability to model complex data, but just like decision trees, it means that the rules can quickly become more difficult to comprehend.

The evolution of classification rule learners didn't stop with RIPPER. New rule learning algorithms are being proposed rapidly. A survey of literature shows algorithms called IREP++, SLIPPER, TRIPPER, among many others.

Rules from decision trees

Classification rules can also be obtained directly from decision trees. Beginning at a leaf node and following the branches back to the root, you will have obtained a series of decisions. These can be combined into a single rule. The following figure shows how rules could be constructed from the decision tree to predict movie success:



Following the paths from the root node down to each leaf, the rules would be:

1. If the number of celebrities is low, then the movie will be a **Box Office Bust**.
2. If the number of celebrities is high and the budget is high, then the movie will be a **Mainstream Hit**.
3. If the number of celebrities is high and the budget is low, then the movie will be a **Critical Success**.

For reasons that will be made clear in the following section, the chief downside to using a decision tree to generate rules is that the resulting rules are often more complex than those learned by a rule learning algorithm. The divide and conquer strategy employed by decision trees biases the results differently than that of a rule learner. On the other hand, it is sometimes more computationally efficient to generate rules from trees.

The `C5.0()` function in the `C50` package will generate a model using classification rules if you specify `rules = TRUE` when training the model.

What makes trees and rules greedy?

Decision trees and rule learners are known as **greedy learners** because they use data on a first-come, first-served basis. Both the divide and conquer heuristic used by decision trees and the separate and conquer heuristic used by rule learners attempt to make partitions one at a time, finding the most homogeneous partition first, followed by the next best, and so on, until all examples have been classified.

The downside to the greedy approach is that greedy algorithms are not guaranteed to generate the optimal, most accurate, or smallest number of rules for a particular dataset. By taking the low-hanging fruit early, a greedy learner may quickly find a single rule that is accurate for one subset of data; however, in doing so, the learner may miss the opportunity to develop a more nuanced set of rules with better overall accuracy on the entire set of data. However, without using the greedy approach to rule learning, it is likely that for all but the smallest of datasets, rule learning would be computationally infeasible.

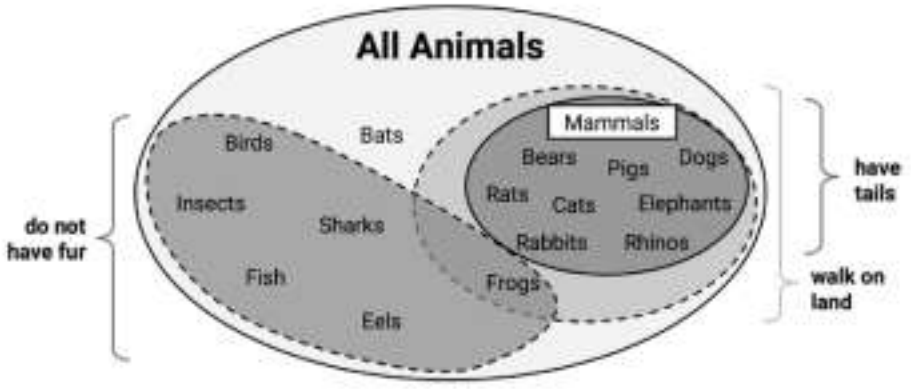
Though both trees and rules employ greedy learning heuristics, there are subtle differences in how they build rules. Perhaps the best way to distinguish them is to note that once divide and conquer splits on a feature, the partitions created by the split may not be re-conquered, only further subdivided. In this way, a tree is permanently limited by its history of past decisions. In contrast, once separate and conquer finds a rule, any examples not covered by all of the rule's conditions may be re-conquered.

To illustrate this contrast, consider the previous case in which we built a rule learner to determine whether an animal was a mammal. The rule learner identified three rules that perfectly classify the example animals:

- Animals that walk on land and have tails are mammals (bears, cats, dogs, elephants, pigs, rabbits, rats, rhinos)
- If the animal does not has fur, it is not a mammal (birds, eels, fish, frogs, insects, sharks)
- Otherwise, the animal is a mammal (bats)

In contrast, a decision tree built on the same data might have come up with four rules to achieve the same perfect classification:

- If an animal walks on land and has fur, then it is a mammal (bears, cats, dogs, elephants, pigs, rabbits, rats, rhinos)
- If an animal walks on land and does not have fur, then it is not a mammal (frogs)
- If the animal does not walk on land and has fur, then it is a mammal (bats)
- If the animal does not walk on land and does not have fur, then it is not a mammal (birds, insects, sharks, fish, eels)



The different result across these two approaches has to do with what happens to the frogs after they are separated by the "walk on land" decision. Where the rule learner allows frogs to be re-conquered by the "does not have fur" decision, the decision tree cannot modify the existing partitions, and therefore must place the frog into its own rule.

On one hand, because rule learners can reexamine cases that were considered but ultimately not covered as part of prior rules, rule learners often find a more parsimonious set of rules than those generated from decision trees. On the other hand, this reuse of data means that the computational cost of rule learners may be somewhat higher than for decision trees.

Example – identifying poisonous mushrooms with rule learners

Each year, many people fall ill and sometimes even die from ingesting poisonous wild mushrooms. Since many mushrooms are very similar to each other in appearance, occasionally even experienced mushroom gatherers are poisoned.

Unlike the identification of harmful plants such as a poison oak or poison ivy, there are no clear rules such as "leaves of three, let them be" to identify whether a wild mushroom is poisonous or edible. Complicating matters, many traditional rules, such as "poisonous mushrooms are brightly colored," provide dangerous or misleading information. If simple, clear, and consistent rules were available to identify poisonous mushrooms, they could save the lives of foragers.

Because one of the strengths of rule learning algorithms is the fact that they generate easy-to-understand rules, they seem like an appropriate fit for this classification task. However, the rules will only be as useful as they are accurate.

Step 1 – collecting data

To identify rules for distinguishing poisonous mushrooms, we will utilize the Mushroom dataset by Jeff Schlimmer of Carnegie Mellon University. The raw dataset is available freely at the UCI Machine Learning Repository (`http://archive.ics.uci.edu/ml`).

The dataset includes information on 8,124 mushroom samples from 23 species of gilled mushrooms listed in *Audubon Society Field Guide to North American Mushrooms* (1981). In the Field Guide, each of the mushroom species is identified "definitely edible," "definitely poisonous," or "likely poisonous, and not recommended to be eaten." For the purposes of this dataset, the latter group was combined with the "definitely poisonous" group to make two classes: poisonous and nonpoisonous. The data dictionary available on the UCI website describes the 22 features of the mushroom samples, including characteristics such as cap shape, cap color, odor, gill size and color, stalk shape, and habitat.

This chapter uses a slightly modified version of the mushroom data. If you plan on following along with the example, download the `mushrooms.csv` file from the Packt Publishing website and save it in your R working directory.

Step 2 – exploring and preparing the data

We begin by using `read.csv()`, to import the data for our analysis. Since all the 22 features and the target class are nominal, in this case, we will set `stringsAsFactors = TRUE` and take advantage of the automatic factor conversion:

```
> mushrooms <- read.csv("mushrooms.csv", stringsAsFactors = TRUE)
```

The output of the `str(mushrooms)` command notes that the data contain 8,124 observations of 23 variables as the data dictionary had described. While most of the `str()` output is unremarkable, one feature is worth mentioning. Do you notice anything peculiar about the `veil_type` variable in the following line?

```
$ veil_type : Factor w/ 1 level "partial": 1 1 1 1 1 1 ...
```

If you think it is odd that a factor has only one level, you are correct. The data dictionary lists two levels for this feature: partial and universal. However, all the examples in our data are classified as partial. It is likely that this data element was somehow coded incorrectly. In any case, since the veil type does not vary across samples, it does not provide any useful information for prediction. We will drop this variable from our analysis using the following command:

```
> mushrooms$veil_type <- NULL
```

By assigning `NULL` to the veil type vector, R eliminates the feature from the `mushrooms` data frame.

Before going much further, we should take a quick look at the distribution of the mushroom `type` class variable in our dataset:

```
> table(mushrooms$type)

   edible poisonous
     4208      3916
```

About 52 percent of the mushroom samples (*N = 4,208*) are edible, while 48 percent (*N = 3,916*) are poisonous.

For the purposes of this experiment, we will consider the 8,214 samples in the mushroom data to be an exhaustive set of all the possible wild mushrooms. This is an important assumption, because it means that we do not need to hold some samples out of the training data for testing purposes. We are not trying to develop rules that cover unforeseen types of mushrooms; we are merely trying to find rules that accurately depict the complete set of known mushroom types. Therefore, we can build and test the model on the same data.

Step 3 – training a model on the data

If we trained a hypothetical ZeroR classifier on this data, what would it predict? Since ZeroR ignores all of the features and simply predicts the target's mode, in plain language, its rule would state that all the mushrooms are edible. Obviously, this is not a very helpful classifier, because it would leave a mushroom gatherer sick or dead with nearly half of the mushroom samples bearing the possibility of being poisonous. Our rules will need to do much better than this in order to provide safe advice that can be published. At the same time, we need simple rules that are easy to remember.

Since simple rules can often be extremely predictive, let's see how a very simple rule learner performs on the mushroom data. Toward the end, we will apply the 1R classifier, which will identify the most predictive single feature of the target class and use it to construct a set of rules.

We will use the 1R implementation in the `RWeka` package called `OneR()`. You may recall that we had installed `RWeka` in *Chapter 1, Introducing Machine Learning*, as a part of the tutorial on installing and loading packages. If you haven't installed the package per these instructions, you will need to use the `install.packages("RWeka")` command and have Java installed on your system (refer to the installation instructions for more details). With these steps complete, load the package by typing `library(RWeka)`:

1R classification rule syntax

using the `OneR()` function in the `RWeka` package

Building the classifier:

```
m <- OneR(class ~ predictors, data = mydata)
```

- `class` is the column in the `mydata` data frame to be predicted
- `predictors` is an R formula specifying the features in the `mydata` data frame to use for prediction
- `data` is the data frame in which `class` and `predictors` can be found

The function will return a 1R model object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test)
```

- `m` is a model trained by the `OneR()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier.

The function will return a vector of predicted class values.

Example:

```
mushroom_classifier <- OneR(type ~ odor + cap_color,
                            data = mushroom_train)
mushroom_prediction <- predict(mushroom_classifier,
                               mushroom_test)
```

The `OneR()` implementation uses the R formula syntax to specify the model to be trained. The formula syntax uses the `~` operator (known as the tilde) to express the relationship between a target variable and its predictors. The class variable to be learned goes to the left of the tilde, and the predictor features are written on the right, separated by `+` operators. If you like to model the relationship between the `y` class and predictors `x1` and `x2`, you could write the formula as `y ~ x1 + x2`. If you like to include all the variables in the model, the special term `.` can be used. For example, `y ~ .` specifies the relationship between `y` and all the other features in the dataset.

The R formula syntax is used across many R functions and offers some powerful features to describe the relationships among predictor variables. We will explore some of these features in the later chapters. However, if you're eager for a sneak peek, feel free to read the documentation using the `?formula` command.

Using the `type ~ .` formula, we will allow our first `OneR()` rule learner to consider all the possible features in the mushroom data while constructing its rules to predict type:

```
> mushroom_1R <- OneR(type ~ ., data = mushrooms)
```

To examine the rules it created, we can type the name of the classifier object, in this case, `mushroom_1R`:

```
> mushroom_1R

odor:
  almond  -> edible
  anise  -> edible
  creosote  -> poisonous
  fishy  -> poisonous
  foul  -> poisonous
  musty  -> poisonous
  none  -> edible
  pungent  -> poisonous
  spicy  -> poisonous
(8004/8124 instances correct)
```

In the first line of the output, we see that the odor feature was selected for rule generation. The categories of odor, such as almond, anise, and so on, specify rules for whether the mushroom is likely to be edible or poisonous. For instance, if the mushroom smells fishy, foul, musty, pungent, spicy, or like creosote, the mushroom is likely to be poisonous. On the other hand, mushrooms with more pleasant smells like almond and anise, and those with no smell at all are predicted to be edible. For the purposes of a field guide for mushroom gathering, these rules could be summarized in a simple rule of thumb: "if the mushroom smells unappetizing, then it is likely to be poisonous."

Step 4 – evaluating model performance

The last line of the output notes that the rules correctly predicted the edibility of 8,004 of the 8,124 mushroom samples or nearly 99 percent of the mushroom samples. We can obtain additional details about the classifier using the `summary()` function, as shown in the following example:

```
> summary(mushroom_1R)

=== Summary ===
Correctly Classified Instances        8004  98.5229 %
Incorrectly Classified Instances       120  1.4771 %
Kappa statistic                          0.9704
Mean absolute error                      0.0148
Root mean squared error                  0.1215
Relative absolute error                  2.958  %
Root relative squared error             24.323  %
Coverage of cases (0.95 level)          98.5229 %
Mean rel. region size (0.95 level)      50      %
Total Number of Instances             8124

=== Confusion Matrix ===
    a    b   <-- classified as
 4208    0 |    a = edible
  120 3796 |    b = poisonous
```

The section labeled `Summary` lists a number of different ways to measure the performance of our 1R classifier. We will cover many of these statistics later on in *Chapter 10, Evaluating Model Performance*, so we will ignore them for now.

The section labeled `Confusion Matrix` is similar to those used before. Here, we can see where our rules went wrong. The key is displayed on the right, with `a = edible` and `b = poisonous`. Table columns indicate the predicted class of the mushroom while the table rows separate the 4,208 edible mushrooms from the 3,916 poisonous mushrooms. Examining the table, we can see that although the 1R classifier did not classify any edible mushrooms as poisonous, it did classify 120 poisonous mushrooms as edible—which makes for an incredibly dangerous mistake!

Considering that the learner utilized only a single feature, it did reasonably well; if one avoids unappetizing smells when foraging for mushrooms, they will almost avoid a trip to the hospital. That said, close does not cut it when lives are involved, not to mention the field guide publisher might not be happy about the prospect of a lawsuit when its readers fall ill. Let's see if we can add a few more rules and develop an even better classifier.

Step 5 – improving model performance

For a more sophisticated rule learner, we will use `JRip()`, a Java-based implementation of the RIPPER rule learning algorithm. As with the 1R implementation we used previously, `JRip()` is included in the `RWeka` package. If you have not done so yet, be sure to load the package using the `library(RWeka)` command:

RIPPER classification rule syntax

using the `JRip()` function in the `RWeka` package

Building the classifier:

```
m <- JRip(class ~ predictors, data = mydata)
```

- `class` is the column in the `mydata` data frame to be predicted
- `predictors` is an R formula specifying the features in the `mydata` data frame to use for prediction
- `data` is the data frame in which `class` and `predictors` can be found

The function will return a RIPPER model object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test)
```

- `m` is a model trained by the `JRip()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier.

The function will return a vector of predicted class values.

Example:

```
mushroom_classifier <- JRip(type ~ odor + cap_color,
                            data = mushroom_train)
mushroom_prediction <- predict(mushroom_classifier,
                               mushroom_test)
```

As shown in the syntax box, the process of training a `JRip()` model is very similar to how we previously trained a `OneR()` model. This is one of the pleasant benefits of the functions in the `RWeka` package; the syntax is consistent across algorithms, which makes the process of comparing a number of different models very simple.

Let's train the `JRip()` rule learner as we did with `OneR()`, allowing it to choose rules from all the available features:

```
> mushroom_JRip <- JRip(type ~ ., data = mushrooms)
```

To examine the rules, type the name of the classifier:

```
> mushroom_JRip

JRIP rules:
===========
(odor = foul) => type=poisonous (2160.0/0.0)
(gill_size = narrow) and (gill_color = buff) => type=poisonous
(1152.0/0.0)
(gill_size = narrow) and (odor = pungent) => type=poisonous (256.0/0.0)
(odor = creosote) => type=poisonous (192.0/0.0)
(spore_print_color = green) => type=poisonous (72.0/0.0)
(stalk_surface_below_ring = scaly) and (stalk_surface_above_ring = silky)
=> type=poisonous (68.0/0.0)
(habitat = leaves) and (cap_color = white) => type=poisonous (8.0/0.0)
(stalk_color_above_ring = yellow) => type=poisonous (8.0/0.0)
 => type=edible (4208.0/0.0)
Number of Rules : 9
```

The `JRip()` classifier learned a total of nine rules from the mushroom data. An easy way to read these rules is to think of them as a list of if-else statements, similar to programming logic. The first three rules could be expressed as:

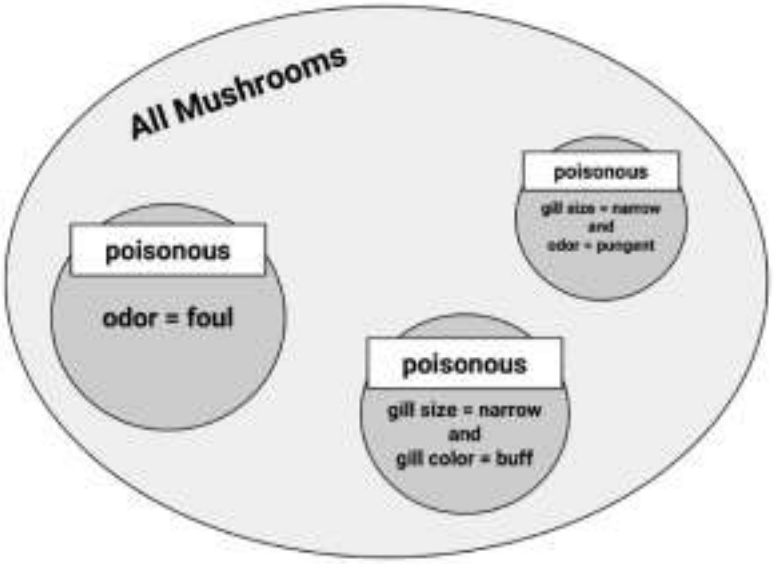
- If the odor is foul, then the mushroom type is poisonous
- If the gill size is narrow and the gill color is buff, then the mushroom type is poisonous
- If the gill size is narrow and the odor is pungent, then the mushroom type is poisonous

Finally, the ninth rule implies that any mushroom sample that was not covered by the preceding eight rules is edible. Following the example of our programming logic, this can be read as:

- Else, the mushroom is edible

The numbers next to each rule indicate the number of instances covered by the rule and a count of misclassified instances. Notably, there were no misclassified mushroom samples using these nine rules. As a result, the number of instances covered by the last rule is exactly equal to the number of edible mushrooms in the data (*N = 4,208*).

The following figure provides a rough illustration of how the rules are applied to the mushroom data. If you imagine everything within the oval as all the species of mushroom, the rule learner identified features or sets of features, which separate homogeneous segments from the larger group. First, the algorithm found a large group of poisonous mushrooms uniquely distinguished by their foul odor. Next, it found smaller and more specific groups of poisonous mushrooms. By identifying covering rules for each of the varieties of poisonous mushrooms, all of the remaining mushrooms were found to be edible. Thanks to Mother Nature, each variety of mushrooms was unique enough that the classifier was able to achieve 100 percent accuracy.



Summary

This chapter covered two classification methods that use so-called "greedy" algorithms to partition the data according to feature values. Decision trees use a divide and conquer strategy to create flowchart-like structures, while rule learners separate and conquer data to identify logical if-else rules. Both methods produce models that can be interpreted without a statistical background.

One popular and highly configurable decision tree algorithm is C5.0. We used the C5.0 algorithm to create a tree to predict whether a loan applicant will default. Using options for boosting and cost-sensitive errors, we were able to improve our accuracy and avoid risky loans that would cost the bank more money.

We also used two rule learners, 1R and RIPPER, to develop rules to identify poisonous mushrooms. The 1R algorithm used a single feature to achieve 99 percent accuracy in identifying potentially fatal mushroom samples. On the other hand, the set of nine rules generated by the more sophisticated RIPPER algorithm correctly identified the edibility of each mushroom.

This chapter merely scratched the surface of how trees and rules can be used. In *Chapter 6, Forecasting Numeric Data – Regression Methods*, we will learn techniques known as regression trees and model trees, which use decision trees for numeric prediction rather than classification. In *Chapter 11, Improving Model Performance*, we will discover how the performance of decision trees can be improved by grouping them together in a model known as a random forest. In *Chapter 8, Finding Patterns – Market Basket Analysis Using Association Rules*, we will see how association rules—a relative of classification rules—can be used to identify groups of items in transactional data.

6

Forecasting Numeric Data – Regression Methods

Mathematical relationships help us to understand many aspects of everyday life. For example, body weight is a function of one's calorie intake, income is often related to education and job experience, and poll numbers help us estimate a presidential candidate's odds of being re-elected.

When such relationships are expressed with exact numbers, we gain additional clarity. For example, an additional 250 kilocalories consumed daily may result in nearly a kilogram of weight gain per month; each year of job experience may be worth an additional $1,000 in yearly salary; and a president is more likely to be re-elected when the economy is strong. Obviously, these equations do not perfectly fit every situation, but we expect that they are reasonably correct, on average.

This chapter extends our machine learning toolkit by going beyond the classification methods covered previously and introducing techniques for estimating relationships among numeric data. While examining several real-world numeric prediction tasks, you will learn:

- The basic statistical principles used in regression, a technique that models the size and the strength of numeric relationships
- How to prepare data for regression analysis, and estimate and interpret a regression model
- A pair of hybrid techniques known as regression trees and model trees, which adapt decision tree classifiers for numeric prediction tasks

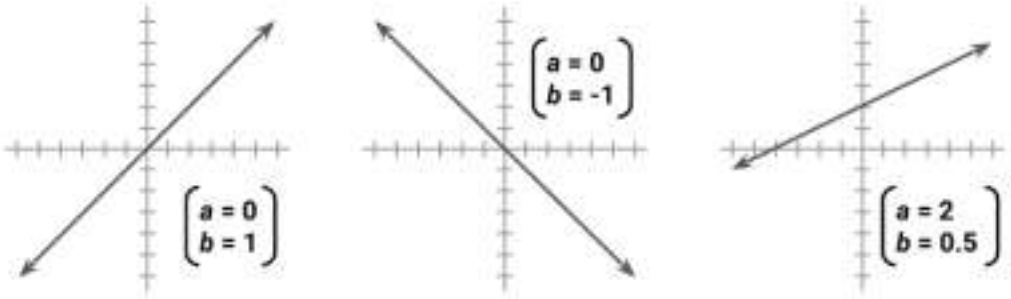
Based on a large body of work in the field of statistics, the methods used in this chapter are a bit heavier on math than those covered previously, but don't worry! Even if your algebra skills are a bit rusty, R takes care of the heavy lifting.

Understanding regression

Regression is concerned with specifying the relationship between a single numeric **dependent variable** (the value to be predicted) and one or more numeric **independent variables** (the predictors). As the name implies, the dependent variable depends upon the value of the independent variable or variables. The simplest forms of regression assume that the relationship between the independent and dependent variables follows a straight line.

The origin of the term "regression" to describe the process of fitting lines to data is rooted in a study of genetics by Sir Francis Galton in the late 19th century. He discovered that fathers who were extremely short or extremely tall tended to have sons whose heights were closer to the average height. He called this phenomenon "regression to the mean".

You might recall from basic algebra that lines can be defined in a **slope-intercept form** similar to $y = a + bx$. In this form, the letter y indicates the dependent variable and x indicates the independent variable. The **slope** term b specifies how much the line rises for each increase in x. Positive values define lines that slope upward while negative values define lines that slope downward. The term a is known as the **intercept** because it specifies the point where the line crosses, or intercepts, the vertical y axis. It indicates the value of y when $x = 0$.



Regression equations model data using a similar slope-intercept format. The machine's job is to identify values of a and b so that the specified line is best able to relate the supplied x values to the values of y. There may not always be a single function that perfectly relates the values, so the machine must also have some way to quantify the margin of error. We'll discuss this in depth shortly.

Regression analysis is commonly used for modeling complex relationships among data elements, estimating the impact of a treatment on an outcome, and extrapolating into the future. Although it can be applied to nearly any task, some specific use cases include:

- Examining how populations and individuals vary by their measured characteristics, for use in scientific research across fields as diverse as economics, sociology, psychology, physics, and ecology
- Quantifying the causal relationship between an event and the response, such as those in clinical drug trials, engineering safety tests, or marketing research
- Identifying patterns that can be used to forecast future behavior given known criteria, such as predicting insurance claims, natural disaster damage, election results, and crime rates

Regression methods are also used for **statistical hypothesis testing**, which determines whether a premise is likely to be true or false in light of the observed data. The regression model's estimates of the strength and consistency of a relationship provide information that can be used to assess whether the observations are due to chance alone.

Hypothesis testing is extremely nuanced and falls outside the scope of machine learning. If you are interested in this topic, an introductory statistics textbook is a good place to get started.

Regression analysis is not synonymous with a single algorithm. Rather, it is an umbrella for a large number of methods that can be adapted to nearly any machine learning task. If you were limited to choosing only a single method, regression would be a good choice. One could devote an entire career to nothing else and perhaps still have much to learn.

In this chapter, we'll focus only on the most basic **linear regression** models—those that use straight lines. When there is only a single independent variable it is known as **simple linear regression**. In the case of two or more independent variables, this is known as **multiple linear regression**, or simply "multiple regression". Both of these techniques assume that the dependent variable is measured on a continuous scale.

Regression can also be used for other types of dependent variables and even for some classification tasks. For instance, **logistic regression** is used to model a binary categorical outcome, while **Poisson regression**—named after the French mathematician Siméon Poisson—models integer count data. The method known as **multinomial logistic regression** models a categorical outcome; thus, it can be used for classification. The same basic principles apply across all the regression methods, so after understanding the linear case, it is fairly simple to learn the others.

Many of the specialized regression methods fall into a class of **Generalized Linear Models (GLM)**. Using a GLM, linear models can be generalized to other patterns via the use of a **link function**, which specifies more complex forms for the relationship between x and y. This allows regression to be applied to almost any type of data.

We'll begin with the basic case of simple linear regression. Despite the name, this method is not too simple to address complex problems. In the next section, we'll see how the use of a simple linear regression model might have averted a tragic engineering disaster.

Simple linear regression

On January 28, 1986, seven crew members of the United States space shuttle *Challenger* were killed when a rocket booster failed, causing a catastrophic disintegration. In the aftermath, experts focused on the launch temperature as a potential culprit. The rubber O-rings responsible for sealing the rocket joints had never been tested below 40°F (4°C) and the weather on the launch day was unusually cold and below freezing.

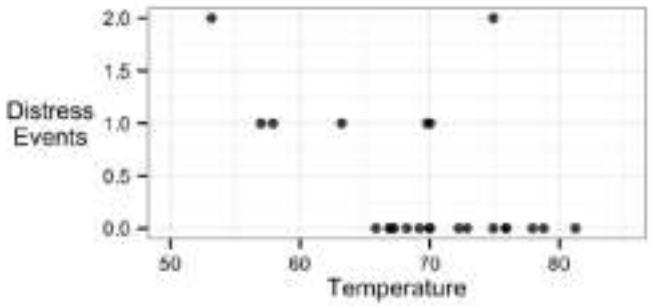
With the benefit of hindsight, the accident has been a case study for the importance of data analysis and visualization. Although it is unclear what information was available to the rocket engineers and decision makers leading up to the launch, it is undeniable that better data, utilized carefully, might very well have averted this disaster.

This section's analysis is based on data presented in Dalal SR, Fowlkes EB, Hoadley B. *Risk analysis of the space shuttle: pre-Challenger prediction of failure*. Journal of the American Statistical Association. 1989; 84:945-957. For one perspective on how data may have changed the result, see Tufte ER. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press; 1997. For a counterpoint, see Robison W, Boisioly R, Hoeker D, Young, S. *Representation and misrepresentation: Tufte and the Morton Thiokol engineers on the Challenger*. Science and Engineering Ethics. 2002; 8:59-81.

The rocket engineers almost certainly knew that cold temperatures could make the components more brittle and less able to seal properly, which would result in a higher chance of a dangerous fuel leak. However, given the political pressure to continue with the launch, they needed data to support this hypothesis. A regression model that demonstrated a link between temperature and O-ring failure, and could forecast the chance of failure given the expected temperature at launch, might have been very helpful.

To build the regression model, scientists might have used the data on launch temperature and component distresses from 23 previous successful shuttle launches. A component distress indicates one of the two types of problems. The first problem, called erosion, occurs when excessive heat burns up the O-ring. The second problem, called blowby, occurs when hot gases leak through or "blow by" a poorly sealed O-ring. Since the shuttle has a total of six primary O-rings, up to six distresses can occur per flight. Though the rocket can survive one or more distress events, or fail with as few as one, each additional distress increases the probability of a catastrophic failure.

The following scatterplot shows a plot of primary O-ring distresses detected for the previous 23 launches, as compared to the temperature at launch:



Examining the plot, there is an apparent trend. Launches occurring at higher temperatures tend to have fewer O-ring distress events.

Additionally, the coldest launch (53° F) had two distress events, a level which had only been reached in one other launch. With this information at hand, the fact that the Challenger was scheduled to launch at a temperature over 20 degrees colder seems concerning. But exactly how concerned should we be? To answer this question, we can turn to simple linear regression.

A simple linear regression model defines the relationship between a dependent variable and a single independent predictor variable using a line defined by an equation in the following form:

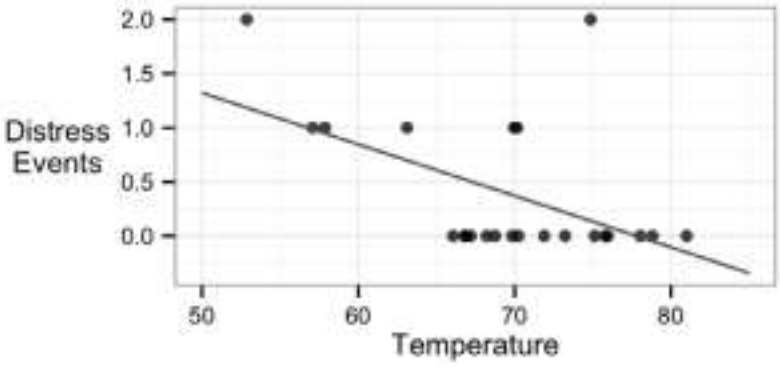
$$y = \alpha + \beta x$$

Don't be alarmed by the Greek characters, this equation can still be understood using the slope-intercept form described previously. The intercept, α (alpha), describes where the line crosses the y axis, while the slope, β (beta), describes the change in y given an increase of x. For the shuttle launch data, the slope would tell us the expected reduction in the number of O-ring failures for each degree the launch temperature increases.

Greek characters are often used in the field of statistics to indicate variables that are parameters of a statistical function. Therefore, performing a regression analysis involves finding **parameter estimates** for α and β. The parameter estimates for alpha and beta are often denoted using a and b, although you may find that some of this terminology and notation is used interchangeably.

Suppose we know that the estimated regression parameters in the equation for the shuttle launch data are: $a = 3.70$ and $b = -0.048$.

Hence, the full linear equation is $y = 3.70 - 0.048x$. Ignoring for a moment how these numbers were obtained, we can plot the line on the scatterplot like this:

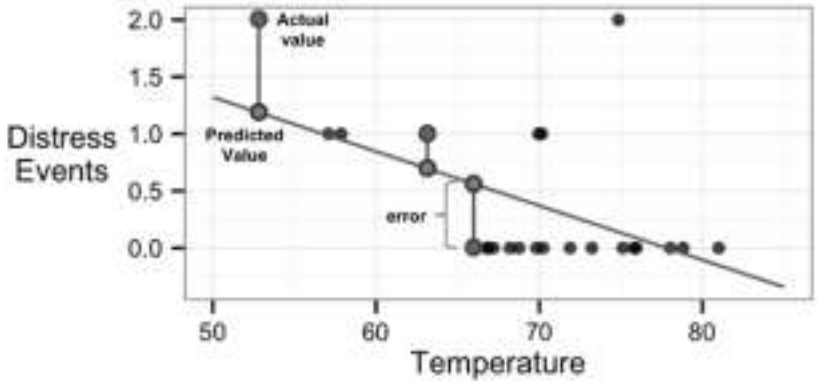


As the line shows, at 60 degrees Fahrenheit, we predict just under one O-ring distress. At 70 degrees Fahrenheit, we expect around 0.3 failures. If we extrapolate our model, all the way to 31 degrees—the forecasted temperature for the Challenger launch—we would expect about $3.70 - 0.048 * 31 = 2.21$ O-ring distress events. Assuming that each O-ring failure is equally likely to cause a catastrophic fuel leak means that the Challenger launch at 31 degrees was nearly three times more risky than the typical launch at 60 degrees, and over eight times more risky than a launch at 70 degrees.

Notice that the line doesn't pass through each data point exactly. Instead, it cuts through the data somewhat evenly, with some predictions lower or higher than the line. In the next section, we will learn about why this particular line was chosen.

Ordinary least squares estimation

In order to determine the optimal estimates of α and β, an estimation method known as **Ordinary Least Squares (OLS)** was used. In OLS regression, the slope and intercept are chosen so that they minimize the sum of the squared errors, that is, the vertical distance between the predicted y value and the actual y value. These errors are known as **residuals**, and are illustrated for several points in the following diagram:



In mathematical terms, the goal of OLS regression can be expressed as the task of minimizing the following equation:

$$\sum (y_i - \hat{y}_i)^2 = \sum e_i^2$$

In plain language, this equation defines e (the error) as the difference between the actual y value and the predicted y value. The error values are squared and summed across all the points in the data.

The caret character (ˆ) above the y term is a commonly used feature of statistical notation. It indicates that the term is an estimate for the true y value. This is referred to as the y-hat, and is pronounced exactly like the hat you'd wear on your head.

The solution for *a* depends on the value of *b*. It can be obtained using the following formula:

$$a = \bar{y} - b\bar{x}$$

To understand these equations, you'll need to know another bit of statistical notation. The horizontal bar appearing over the *x* and *y* terms indicates the mean value of *x* or *y*. This is referred to as the *x*-bar or *y*-bar, and is pronounced just like the establishment you'd go to for an alcoholic drink.

Though the proof is beyond the scope of this book, it can be shown using calculus that the value of *b* that results in the minimum squared error is:

$$b = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}$$

If we break this equation apart into its component pieces, we can simplify it a bit. The denominator for *b* should look familiar; it is very similar to the variance of *x*, which is denoted as *Var(x)*. As we learned in *Chapter 2, Managing and Understanding Data*, the variance involves finding the average squared deviation from the mean of *x*. This can be expressed as:

$$\mathrm{Var}(x) = \frac{\sum(x_i - \bar{x})^2}{n}$$

The numerator involves taking the sum of each data point's deviation from the mean *x* value multiplied by that point's deviation away from the mean *y* value. This is similar to the **covariance** function for *x* and *y*, denoted as *Cov(x, y)*. The covariance formula is:

$$\mathrm{Cov}(x, y) = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{n}$$

If we divide the covariance function by the variance function, the n terms get cancelled and we can rewrite the formula for *b* as:

$$b = \frac{\mathrm{Cov}(x, y)}{\mathrm{Var}(x)}$$

Given this restatement, it is easy to calculate the value of *b* using built-in R functions. Let's apply it to the rocket launch data to estimate the regression line.

If you would like to follow along with these examples, download the `challenger.csv` file from the Packt Publishing website and load to a data frame using the `launch <- read.csv("challenger.csv")` command.

Assume that our shuttle launch data is stored in a data frame named `launch`, the independent variable *x* is temperature, and the dependent variable *y* is `distress_ct`. We can then use R's `cov()` and `var()` functions to estimate *b*:

```
> b <- cov(launch$temperature, launch$distress_ct) /
        var(launch$temperature)
> b
[1] -0.04753968
```

From here we can estimate *a* using the `mean()` function:

```
> a <- mean(launch$distress_ct) - b * mean(launch$temperature)
> a
[1] 3.698413
```

Estimating the regression equation by hand is not ideal, so R provides functions for performing this calculation automatically. We will use such methods shortly. First, we will expand our understanding of regression by learning a method for measuring the strength of a linear relationship, and then we will see how linear regression can be applied to data having more than one independent variable.

Correlations

The **correlation** between two variables is a number that indicates how closely their relationship follows a straight line. Without additional qualification, correlation typically refers to **Pearson's correlation coefficient**, which was developed by the 20th century mathematician Karl Pearson. The correlation ranges between -1 and +1. The extreme values indicate a perfectly linear relationship, while a correlation close to zero indicates the absence of a linear relationship.

The following formula defines Pearson's correlation:

$$\rho_{x,y} = \text{Corr}(x, y) = \frac{\text{Cov}(x, y)}{\sigma_x \sigma_y}$$

More Greek notation has been introduced here. The first symbol (which looks like a lowercase p) is *rho*, and it is used to denote the Pearson correlation statistic. The characters that look like q turned sideways are the Greek letter *sigma*, and they indicate the standard deviation of x or y.

Using this formula, we can calculate the correlation between the launch temperature and the number of O-ring distress events. Recall that the covariance function is `cov()` and the standard deviation function is `sd()`. We'll store the result in `r`, a letter that is commonly used to indicate the estimated correlation:

```
> r <- cov(launch$temperature, launch$distress_ct) /
        (sd(launch$temperature) * sd(launch$distress_ct))
> r
[1] -0.5111264
```

Alternatively, we can use R's correlation function, `cor()`:

```
> cor(launch$temperature, launch$distress_ct)
[1] -0.5111264
```

The correlation between the temperature and the number of distressed O-rings is -0.51. The negative correlation implies that increases in temperature are related to decreases in the number of distressed O-rings. To the NASA engineers studying the O-ring data, this would have been a very clear indicator that a low temperature launch could be problematic. The correlation also tells us about the relative strength of the relationship between temperature and O-ring distress. Because -0.51 is halfway to the maximum negative correlation of -1, this implies that there is a moderately strong negative linear association.

There are various rules of thumb used to interpret correlation strength. One method assigns a status of "weak" to values between 0.1 and 0.3, "moderate" to the range of 0.3 to 0.5, and "strong" to values above 0.5 (these also apply to similar ranges of negative correlations). However, these thresholds may be too lax for some purposes. Often, the correlation must be interpreted in context. For data involving human beings, a correlation of 0.5 may be considered extremely high, while for data generated by mechanical processes, a correlation of 0.5 may be weak.

You have probably heard the expression "correlation does not imply causation." This is rooted in the fact that a correlation only describes the association between a pair of variables, yet there could be other unmeasured explanations. For example, there may be a strong association between mortality and time per day spent matching movies, but before doctors should start recommending that we all watch more movies, we need to rule out another explanation—younger people watch more movies and are less likely to die.

Measuring the correlation between two variables gives us a way to quickly gauge the relationships among the independent and dependent variables. This will be increasingly important as we start defining the regression models with a larger number of predictors.

Multiple linear regression

Most real-world analyses have more than one independent variable. Therefore, it is likely that you will be using **multiple linear regression** for most numeric prediction tasks. The strengths and weaknesses of multiple linear regression are shown in the following table:

Strengths	Weaknesses
• By far the most common approach for modeling numeric data • Can be adapted to model almost any modeling task • Provides estimates of both the strength and size of the relationships among features and the outcome	• Makes strong assumptions about the data • The model's form must be specified by the user in advance • Does not handle missing data • Only works with numeric features, so categorical data requires extra processing • Requires some knowledge of statistics to understand the model

We can understand multiple regression as an extension of simple linear regression. The goal in both cases is similar—find values of beta coefficients that minimize the prediction error of a linear equation. The key difference is that there are additional terms for additional independent variables.

Multiple regression equations generally follow the form of the following equation. The dependent variable *y* is specified as the sum of an intercept term *a* plus the product of the estimated β value and the *x* values for each of the *i* features. An error term (denoted by the Greek letter *epsilon*) has been added here as a reminder that the predictions are not perfect. This represents the **residual** term noted previously:

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_i x_i + \varepsilon$$

Let's consider for a moment the interpretation of the estimated regression parameters. You will note that in the preceding equation, a coefficient is provided for each feature. This allows each feature to have a separate estimated effect on the value of *y*. In other words, *y* changes by the amount β_i for each unit increase in x_i. The intercept *a* is then the expected value of *y* when the independent variables are all zero.

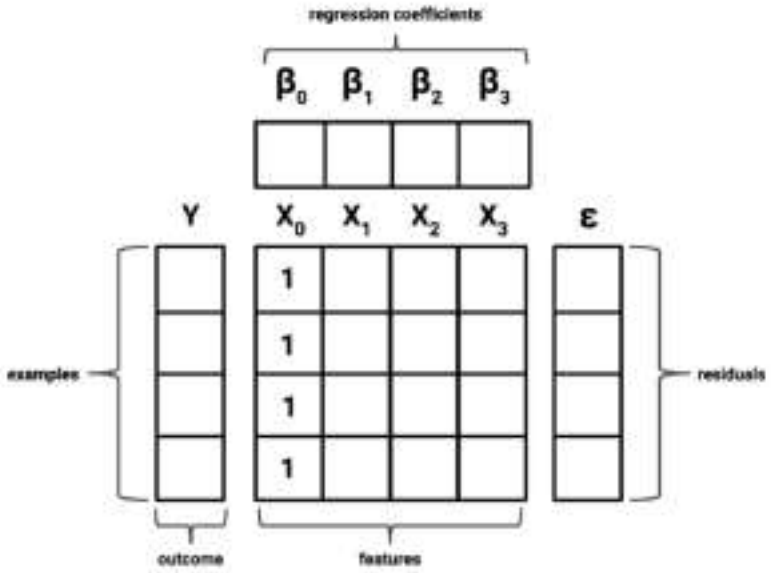
Since the intercept term *a* is really no different than any other regression parameter, it is also sometimes denoted as β_0 (pronounced beta-naught), as shown in the following equation:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_i x_i + \varepsilon$$

Just like before, the intercept is unrelated to any of the independent *x* variables. However, for reasons that will become clear shortly, it helps to imagine β_0 as if it were being multiplied by a term x_0, which is a constant with the value 1:

$$y = \beta_0 x_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_i x_i + \varepsilon$$

In order to estimate the values of the regression parameters, each observed value of the dependent variable *y* must be related to the observed values of the independent *x* variables using the regression equation in the previous form. The following figure illustrates this structure:



The many rows and columns of data illustrated in the preceding figure can be described in a condensed formulation using bold font **matrix notation** to indicate that each of the terms represents multiple values:

$$\mathbf{Y} = \beta\mathbf{X} + \varepsilon$$

The dependent variable is now a vector, **Y**, with a row for every example. The independent variables have been combined into a matrix, **X**, with a column for each feature plus an additional column of '1' values for the intercept term. Each column has a row for every example. The regression coefficients **β** and residual errors **ε** are also now vectors.

The goal is now to solve for **β**, the vector of regression coefficients that minimizes the sum of the squared errors between the predicted and actual **Y** values. Finding the optimal solution requires the use of matrix algebra; therefore, the derivation deserves more careful attention than can be provided in this text. However, if you're willing to trust the work of others, the best estimate of the vector **β** can be computed as:

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}$$

This solution uses a pair of matrix operations—the **T** indicates the **transpose** of matrix **X**, while the negative exponent indicates the **matrix inverse**. Using R's built-in matrix operations, we can thus implement a simple multiple regression learner. Let's apply this formula to the Challenger launch data.

If you are unfamiliar with the preceding matrix operations, the Wikipedia pages for transpose and matrix inverse provide a thorough introduction and are quite understandable, even without a strong mathematics background.

Using the following code, we can create a basic regression function named `reg()`, which takes a parameter `y` and a parameter `x` and returns a vector of estimated beta coefficients:

```
reg <- function(y, x) {
  x <- as.matrix(x)
  x <- cbind(Intercept = 1, x)
  b <- solve(t(x) %*% x) %*% t(x) %*% y
  colnames(b) <- "estimate"
  print(b)
}
```

The `reg()` function created here uses several R commands that we have not used previously. First, since we will be using the function with sets of columns from a data frame, the `as.matrix()` function is used to convert the data frame into matrix form. Next, the `cbind()` function is used to bind an additional column onto the `x` matrix; the command `Intercept = 1` instructs R to name the new column `Intercept` and to fill the column with repeating 1 values. Then, a number of matrix operations are performed on the `x` and `y` objects:

- `solve()` takes the inverse of a matrix
- `t()` is used to transpose a matrix
- `%*%` multiplies two matrices

By combining these as shown, our function will return a vector `b`, which contains the estimated parameters for the linear model relating `x` to `y`. The final two lines in the function give the `b` vector a name and print the result on screen.

Let's apply our function to the shuttle launch data. As shown in the following code, the dataset includes three features and the distress count (`distress_ct`), which is the outcome of interest:

```
> str(launch)
'data.frame':       23 obs. of  4 variables:
 $ distress_ct         : int  0 1 0 0 0 0 0 0 1 1 ...
 $ temperature         : int  66 70 69 68 67 72 73 70 57 63 ...
 $ field_check_pressure: int  50 50 50 50 50 50 100 100 200 ...
 $ flight_num          : int  1 2 3 4 5 6 7 8 9 10 ...
```

We can confirm that our function is working correctly by comparing its result to the simple linear regression model of O-ring failures versus temperature, which we found earlier to have parameters $a = 3.70$ and $b = -0.048$. Since temperature is in the third column of the launch data, we can run the `reg()` function as follows:

```
> reg(y = launch$distress_ct, x = launch[2])
               estimate
Intercept    3.69841270
temperature -0.04753968
```

These values exactly match our prior result, so let's use the function to build a multiple regression model. We'll apply it just as before, but this time specifying three columns of data instead of just one:

```
> reg(y = launch$distress_ct, x = launch[2:4])
                           estimate
Intercept                3.527093383
temperature             -0.051385940
field_check_pressure     0.001757009
flight_num               0.014292843
```

This model predicts the number of O-ring distress events versus temperature, field check pressure, and the launch ID number. As with the simple linear regression model, the coefficient for the temperature variable is negative, which suggests that as temperature increases, the number of expected O-ring events decreases. The field check pressure refers to the amount of pressure applied to the O-ring to test it prior to launch. Although the check pressure had originally been 50 psi, it was raised to 100 and 200 psi for some launches, which led some to believe that it may be responsible for O-ring erosion. The coefficient is positive, but small. The flight number is included to account for the shuttle's age. As it gets older, its parts may be more brittle or prone to fail. The small positive association between flight number and distress count may reflect this fact.

So far we've only scratched the surface of linear regression modeling. Although our work was useful to help us understand exactly how regression models are built, R's functions also include some additional functionality necessary for the more complex modeling tasks and diagnostic output that are needed to aid model interpretation and assess fit. Let's apply our knowledge of regression to a more challenging learning task.

Example – predicting medical expenses using linear regression

In order for a health insurance company to make money, it needs to collect more in yearly premiums than it spends on medical care to its beneficiaries. As a result, insurers invest a great deal of time and money in developing models that accurately forecast medical expenses for the insured population.

Medical expenses are difficult to estimate because the most costly conditions are rare and seemingly random. Still, some conditions are more prevalent for certain segments of the population. For instance, lung cancer is more likely among smokers than non-smokers, and heart disease may be more likely among the obese.

The goal of this analysis is to use patient data to estimate the average medical care expenses for such population segments. These estimates can be used to create actuarial tables that set the price of yearly premiums higher or lower, depending on the expected treatment costs.

Step 1 – collecting data

For this analysis, we will use a simulated dataset containing hypothetical medical expenses for patients in the United States. This data was created for this book using demographic statistics from the US Census Bureau, and thus, approximately reflect real-world conditions.

If you would like to follow along interactively, download the `insurance.csv` file from the Packt Publishing website and save it to your R working folder.

The `insurance.csv` file includes 1,338 examples of beneficiaries currently enrolled in the insurance plan, with features indicating characteristics of the patient as well as the total medical expenses charged to the plan for the calendar year. The features are:

- `age`: An integer indicating the age of the primary beneficiary (excluding those above 64 years, since they are generally covered by the government).
- `sex`: The policy holder's gender, either male or female.
- `bmi`: The body mass index (BMI), which provides a sense of how over- or under-weight a person is relative to their height. BMI is equal to weight (in kilograms) divided by height (in meters) squared. An ideal BMI is within the range of 18.5 to 24.9.
- `children`: An integer indicating the number of children/dependents covered by the insurance plan.
- `smoker`: A yes or no categorical variable that indicates whether the insured regularly smokes tobacco.
- `region`: The beneficiary's place of residence in the US, divided into four geographic regions: northeast, southeast, southwest, or northwest.

It is important to give some thought to how these variables may be related to billed medical expenses. For instance, we might expect that older people and smokers are at higher risk of large medical expenses. Unlike many other machine learning methods, in regression analysis, the relationships among the features are typically specified by the user rather than being detected automatically. We'll explore some of these potential relationships in the next section.

Step 2 – exploring and preparing the data

As we have done before, we will use the `read.csv()` function to load the data for analysis. We can safely use `stringsAsFactors = TRUE` because it is appropriate to convert the three nominal variables to factors:

```
> insurance <- read.csv("insurance.csv", stringsAsFactors = TRUE)
```

The `str()` function confirms that the data is formatted as we had expected:

```
> str(insurance)
'data.frame':     1338 obs. of  7 variables:
 $ age     : int  19 18 28 33 32 31 46 37 37 60 ...
 $ sex     : Factor w/ 2 levels "female","male": 1 2 2 2 2 1 ...
 $ bmi     : num  27.9 33.8 33 22.7 28.9 25.7 33.4 27.7 ...
 $ children: int  0 1 3 0 0 0 1 3 2 0 ...
```

```
$ smoker   : Factor w/ 2 levels "no","yes": 2 1 1 1 1 1 1 1 ...
$ region   : Factor w/ 4 levels "northeast","northwest",..: ...
$ expenses: num  16885 1726 4449 21984 3867 ...
```

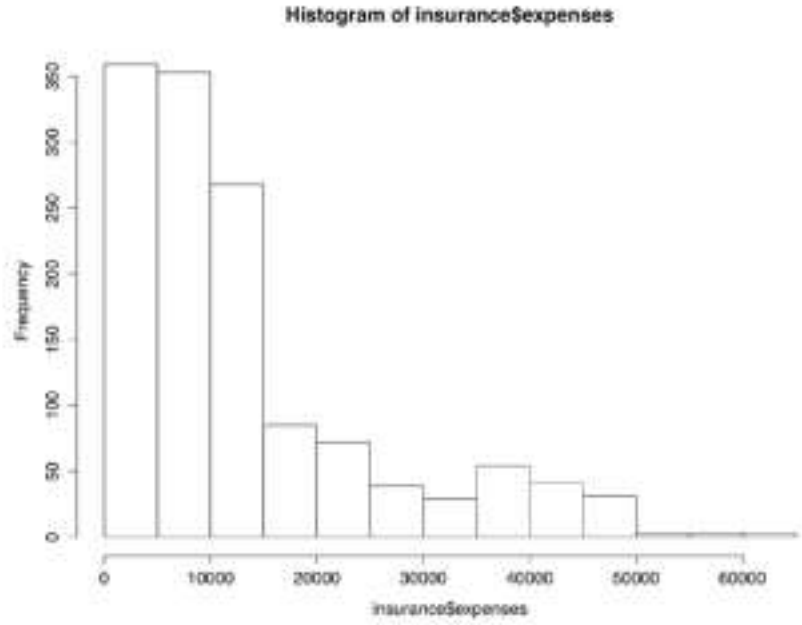
Our model's dependent variable is `expenses`, which measures the medical costs each person charged to the insurance plan for the year. Prior to building a regression model, it is often helpful to check for normality. Although linear regression does not strictly require a normally distributed dependent variable, the model often fits better when this is true. Let's take a look at the summary statistics:

```
> summary(insurance$expenses)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1122    4740    9382   13270   16640   63770
```

Because the mean value is greater than the median, this implies that the distribution of insurance expenses is right-skewed. We can confirm this visually using a histogram:

```
> hist(insurance$expenses)
```

The output is shown as follows:



As expected, the figure shows a right-skewed distribution. It also shows that the majority of people in our data have yearly medical expenses between zero and $15,000, in spite of the fact that the tail of the distribution extends far past these peaks. Although this distribution is not ideal for a linear regression, knowing this weakness ahead of time may help us design a better-fitting model later on.

Before we address that issue, another problem is at hand. Regression models require that every feature is numeric, yet we have three factor-type features in our data frame. For instance, the sex variable is divided into male and female levels, while smoker is divided into yes and no. From the `summary()` output, we know that the `region` variable has four levels, but we need to take a closer look to see how they are distributed:

```
> table(insurance$region)

northeast northwest southeast southwest
      324       325       364       325
```

Here, we see that the data has been divided nearly evenly among four geographic regions. We will see how R's linear regression function handles these factor variables shortly.

Exploring relationships among features – the correlation matrix

Before fitting a regression model to data, it can be useful to determine how the independent variables are related to the dependent variable and each other. A **correlation matrix** provides a quick overview of these relationships. Given a set of variables, it provides a correlation for each pairwise relationship.

To create a correlation matrix for the four numeric variables in the insurance data frame, use the `cor()` command:

```
> cor(insurance[c("age", "bmi", "children", "expenses")])
               age        bmi   children   expenses
age      1.0000000 0.10934101 0.04246900 0.29900819
bmi      0.1093410 1.00000000 0.01264471 0.19857626
children 0.0424690 0.01264471 1.00000000 0.06799823
expenses 0.2990082 0.19857626 0.06799823 1.00000000
```

At the intersection of each row and column pair, the correlation is listed for the variables indicated by that row and column. The diagonal is always `1.0000000` since there is always a perfect correlation between a variable and itself. The values above and below the diagonal are identical since correlations are symmetrical. In other words, `cor(x, y)` is equal to `cor(y, x)`.

None of the correlations in the matrix are considered strong, but there are some notable associations. For instance, `age` and `bmi` appear to have a weak positive correlation, meaning that as someone ages, their body mass tends to increase. There is also a moderate positive correlation between `age` and `expenses`, `bmi` and `expenses`, and `children` and `expenses`. These associations imply that as age, body mass, and number of children increase, the expected cost of insurance goes up. We'll try to tease out these relationships more clearly when we build our final regression model.

Visualizing relationships among features – the scatterplot matrix

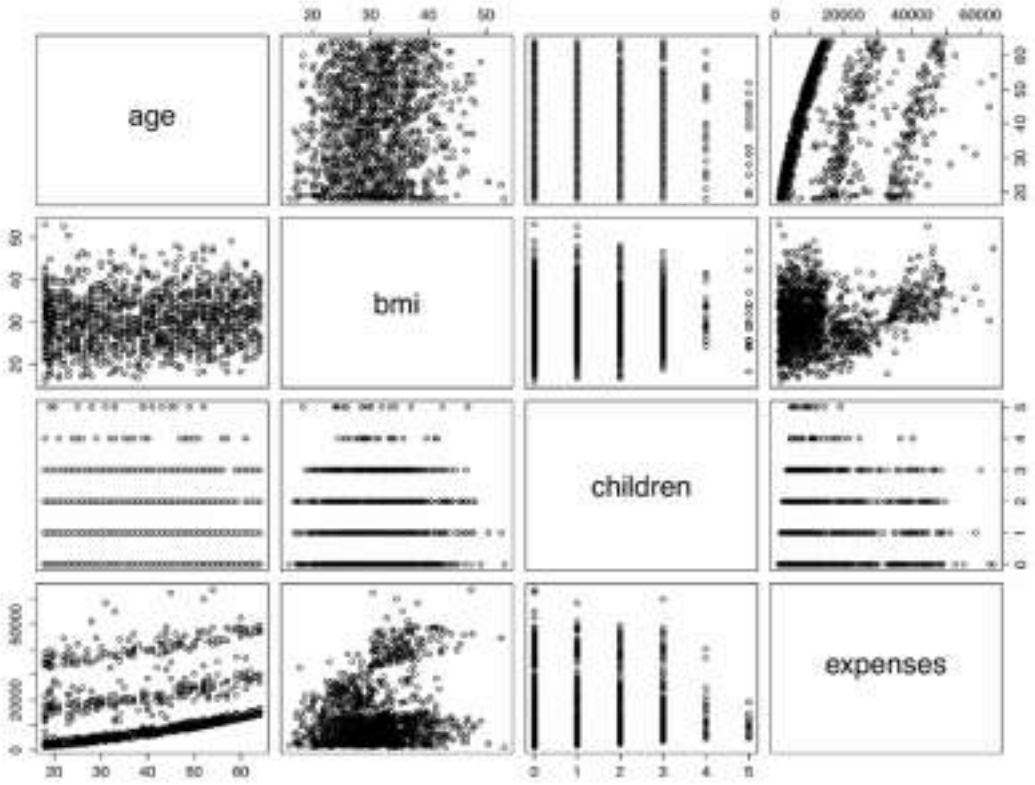
It can also be helpful to visualize the relationships among numeric features by using a scatterplot. Although we could create a scatterplot for each possible relationship, doing so for a large number of features might become tedious.

An alternative is to create a **scatterplot matrix** (sometimes abbreviated as **SPLOM**), which is simply a collection of scatterplots arranged in a grid. It is used to detect patterns among three or more variables. The scatterplot matrix is not a true multidimensional visualization because only two features are examined at a time. Still, it provides a general sense of how the data may be interrelated.

We can use R's graphical capabilities to create a scatterplot matrix for the four numeric features: `age`, `bmi`, `children`, and `expenses`. The `pairs()` function is provided in a default R installation and provides basic functionality for producing scatterplot matrices. To invoke the function, simply provide it the data frame to present. Here, we'll limit the `insurance` data frame to the four numeric variables of interest:

```
> pairs(insurance[c("age", "bmi", "children", "expenses")])
```

This produces the following diagram:



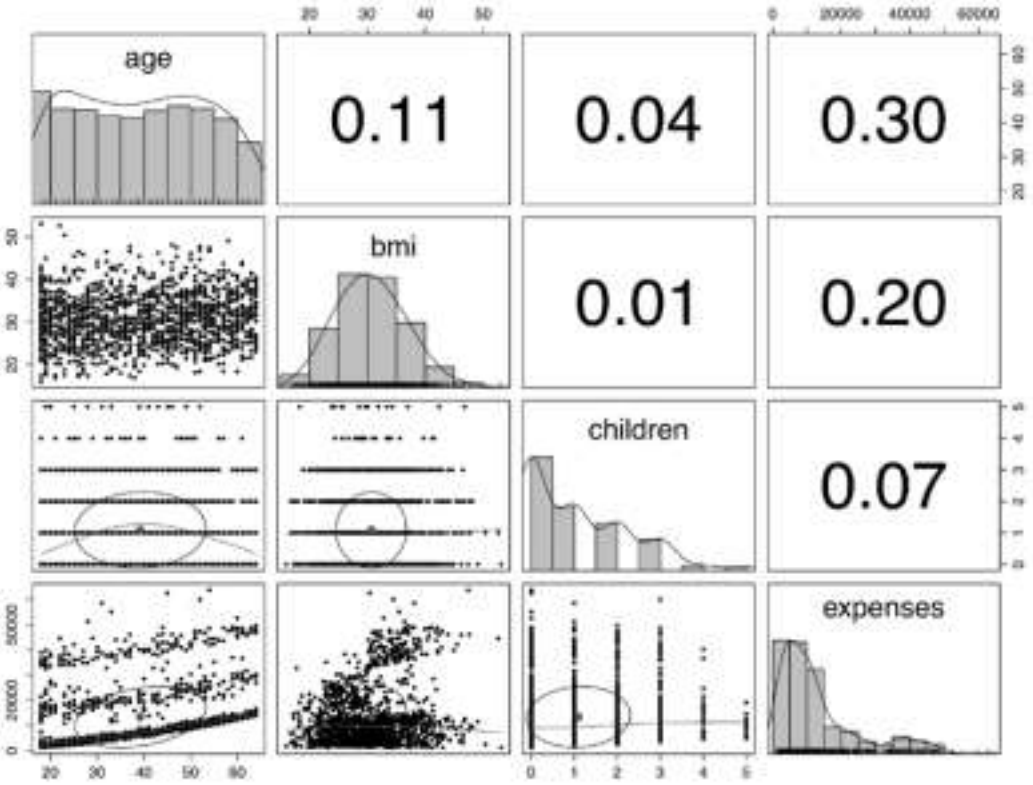
In the scatterplot matrix, the intersection of each row and column holds the scatterplot of the variables indicated by the row and column pair. The diagrams above and below the diagonal are transpositions since the *x* axis and *y* axis have been swapped.

Do you notice any patterns in these plots? Although some look like random clouds of points, a few seem to display some trends. The relationship between `age` and `expenses` displays several relatively straight lines, while the `bmi` versus `expenses` plot has two distinct groups of points. It is difficult to detect trends in any of the other plots.

If we add more information to the plot, it can be even more useful. An enhanced scatterplot matrix can be created with the `pairs.panels()` function in the `psych` package. If you do not have this package installed, type `install.packages("psych")` to install it on your system and load it using the `library(psych)` command. Then, we can create a scatterplot matrix as we had done previously:

```
> pairs.panels(insurance[c("age", "bmi", "children", "expenses")])
```

This produces a slightly more informative scatterplot matrix, as shown here:



Above the diagonal, the scatterplots have been replaced with a correlation matrix. On the diagonal, a histogram depicting the distribution of values for each feature is shown. Finally, the scatterplots below the diagonal are now presented with additional visual information.

The oval-shaped object on each scatterplot is a **correlation ellipse**. It provides a visualization of correlation strength. The dot at the center of the ellipse indicates the point at the mean values for the *x* and *y* axis variables. The correlation between the two variables is indicated by the shape of the ellipse; the more it is stretched, the stronger the correlation. An almost perfectly round oval, as with `bmi` and `children`, indicates a very weak correlation (in this case, it is 0.01).

The curve drawn on the scatterplot is called a **loess curve**. It indicates the general relationship between the *x* and *y* axis variables. It is best understood by example. The curve for `age` and `children` is an upside-down U, peaking around middle age. This means that the oldest and youngest people in the sample have fewer children on the insurance plan than those around middle age. Because this trend is non-linear, this finding could not have been inferred from the correlations alone. On the other hand, the loess curve for `age` and `bmi` is a line sloping gradually up, implying that body mass increases with age, but we had already inferred this from the correlation matrix.

Step 3 – training a model on the data

To fit a linear regression model to data with R, the `lm()` function can be used. This is included in the `stats` package, which should be included and loaded by default with your R installation. The `lm()` syntax is as follows:

Multiple regression modeling syntax

using the `lm()` function in the `stats` package

Building the model:

```
m <- lm(dv ~ iv, data = mydata)
```

- `dv` is the dependent variable in the `mydata` data frame to be modeled
- `iv` is an R formula specifying the independent variables in the `mydata` data frame to use in the model
- `data` specifies the data frame in which the `dv` and `iv` variables can be found

The function will return a regression model object that can be used to make predictions. Interactions between independent variables can be specified using the `*` operator.

Making predictions:

```
p <- predict(m, test)
```

- `m` is a model trained by the `lm()` function
- `test` is a data frame containing test data with the same features as the training data used to build the model.

The function will return a vector of predicted values.

Example:

```
ins_model <- lm(charges ~ age + sex + smoker,
                data = insurance)
ins_pred <- predict(ins_model, insurance_test)
```

The following command fits a linear regression model relating the six independent variables to the total medical expenses. The R formula syntax uses the tilde character ~ to describe the model; the dependent variable `expenses` goes to the left of the tilde while the independent variables go to the right, separated by + signs. There is no need to specify the regression model's intercept term as it is assumed by default:

```
> ins_model <- lm(expenses ~ age + children + bmi + sex +
    smoker + region, data = insurance)
```

Because the . character can be used to specify all the features (excluding those already specified in the formula), the following command is equivalent to the preceding command:

```
> ins_model <- lm(expenses ~ ., data = insurance)
```

After building the model, simply type the name of the model object to see the estimated beta coefficients:

```
> ins_model

Call:
lm(formula = expenses ~ ., data = insurance)

Coefficients:
    (Intercept)              age          sexmale
      -11941.6             256.8            -131.4
            bmi         children        smokeryes
          339.3             475.7          23847.5
regionnorthwest  regionsoutheast  regionsouthwest
         -352.8           -1035.6            -959.3
```

Understanding the regression coefficients is fairly straightforward. The intercept is the predicted value of `expenses` when the independent variables are equal to zero. As is the case here, quite often the intercept is of little value alone because it is impossible to have values of zero for all features. For example, since no person exists with age zero and BMI zero, the intercept has no real-world interpretation. For this reason, in practice, the intercept is often ignored.

The beta coefficients indicate the estimated increase in expenses for an increase of one in each of the features, assuming all other values are held constant. For instance, for each additional year of age, we would expect $256.80 higher medical expenses on average, assuming everything else is equal. Similarly, each additional child results in an average of $475.70 in additional medical expenses each year, and each unit increase in BMI is associated with an average increase of $339.30 in yearly medical expenses, all else equal.

You might notice that although we only specified six features in our model formula, there are eight coefficients reported in addition to the intercept. This happened because the `lm()` function automatically applied a technique known as **dummy coding** to each of the factor-type variables we included in the model.

Dummy coding allows a nominal feature to be treated as numeric by creating a binary variable, often called a **dummy variable**, for each category of the feature. The dummy variable is set to `1` if the observation falls into the specified category or `0` otherwise. For instance, the `sex` feature has two categories: `male` and `female`. This will be split into two binary variables, which R names `sexmale` and `sexfemale`. For observations where `sex = male`, then `sexmale = 1` and `sexfemale = 0`; conversely, if `sex = female`, then `sexmale = 0` and `sexfemale = 1`. The same coding applies to variables with three or more categories. For example, R split the four-category feature `region` into four dummy variables: `regionnorthwest`, `regionsoutheast`, `regionsouthwest`, and `regionnortheast`.

When adding a dummy variable to a regression model, one category is always left out to serve as the reference category. The estimates are then interpreted relative to the reference. In our model, R automatically held out the `sexfemale`, `smokerno`, and `regionnortheast` variables, making female non-smokers in the northeast region the reference group. Thus, males have $131.40 less medical expenses each year relative to females and smokers cost an average of $23,847.50 more than non-smokers per year. The coefficient for each of the three regions in the model is negative, which implies that the reference group, the northeast region, tends to have the highest average expenses.

By default, R uses the first level of the factor variable as the reference. If you would prefer to use another level, the `relevel()` function can be used to specify the reference group manually. Use the `?relevel` command in R for more information.

The results of the linear regression model make logical sense: old age, smoking, and obesity tend to be linked to additional health issues, while additional family member dependents may result in an increase in physician visits and preventive care such as vaccinations and yearly physical exams. However, we currently have no sense of how well the model is fitting the data. We'll answer this question in the next section.

Step 4 – evaluating model performance

The parameter estimates we obtained by typing `ins_model` tell us about how the independent variables are related to the dependent variable, but they tell us nothing about how well the model fits our data. To evaluate the model performance, we can use the `summary()` command on the stored model:

```
> summary(ins_model)
```

This produces the following output. Note that the output has been labeled for illustrative purposes:

```
Call:
lm(formula = expenses ~ ., data = insurance)

Residuals:
     Min       1Q   Median       3Q      Max 
-11302.7  -2850.9   -979.6   1383.9  29981.7 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept)  -11941.6      987.8 -12.089  < 2e-16 ***
age             256.8       11.9  21.586  < 2e-16 ***
sexmale        -131.3      332.9  -0.395 0.693255    
bmi             339.3       28.6  11.864  < 2e-16 ***
children        475.7      137.8   3.452 0.000574 ***
smokeryes     23847.5      413.1  57.723  < 2e-16 ***
regionnorthwest  -352.8      476.3  -0.741 0.458976    
regionsoutheast -1035.6      478.7  -2.163 0.030685 *  
regionsouthwest  -959.3      477.9  -2.007 0.044921 *  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6062 on 1329 degrees of freedom
Multiple R-squared:  0.7509,  Adjusted R-squared:  0.7494 
F-statistic: 500.9 on 8 and 1329 DF,  p-value: < 2.2e-16
```

The `summary()` output may seem confusing at first, but the basics are easy to pick up. As indicated by the numbered labels in the preceding output, the output provides three key ways to evaluate the performance, or fit, of our model:

1. The **residuals** section provides summary statistics for the errors in our predictions, some of which are apparently quite substantial. Since a residual is equal to the true value minus the predicted value, the maximum error of 29981.7 suggests that the model under-predicted expenses by nearly $30,000 for at least one observation. On the other hand, 50 percent of errors fall within the 1Q and 3Q values (the first and third quartile), so the majority of predictions were between $2,850.90 over the true value and $1,383.90 under the true value.

2. For each estimated regression coefficient, the **p-value**, denoted `Pr(>|t|)`, provides an estimate of the probability that the true coefficient is zero given the value of the estimate. Small p-values suggest that the true coefficient is very unlikely to be zero, which means that the feature is extremely unlikely to have no relationship with the dependent variable. Note that some of the p-values have stars (`***`), which correspond to the footnotes to indicate the **significance level** met by the estimate. This level is a threshold, chosen prior to building the model, which will be used to indicate "real" findings, as opposed to those due to chance alone; p-values less than the significance level are considered **statistically significant**. If the model had few such terms, it may be cause for concern, since this would indicate that the features used are not very predictive of the outcome. Here, our model has several highly significant variables, and they seem to be related to the outcome in logical ways.
3. The **multiple R-squared value** (also called the coefficient of determination) provides a measure of how well our model as a whole explains the values of the dependent variable. It is similar to the correlation coefficient, in that the closer the value is to 1.0, the better the model perfectly explains the data. Since the R-squared value is 0.7494, we know that the model explains nearly 75 percent of the variation in the dependent variable. Because models with more features always explain more variation, the **adjusted R-squared value** corrects R-squared by penalizing models with a large number of independent variables. It is useful for comparing the performance of models with different numbers of explanatory variables.

Given the preceding three performance indicators, our model is performing fairly well. It is not uncommon for regression models of real-world data to have fairly low R-squared values; a value of 0.75 is actually quite good. The size of some of the errors is a bit concerning, but not surprising given the nature of medical expense data. However, as shown in the next section, we may be able to improve the model's performance by specifying the model in a slightly different way.

Step 5 – improving model performance

As mentioned previously, a key difference between the regression modeling and other machine learning approaches is that regression typically leaves feature selection and model specification to the user. Consequently, if we have subject matter knowledge about how a feature is related to the outcome, we can use this information to inform the model specification and potentially improve the model's performance.

Model specification – adding non-linear relationships

In linear regression, the relationship between an independent variable and the dependent variable is assumed to be linear, yet this may not necessarily be true. For example, the effect of age on medical expenditure may not be constant throughout all the age values; the treatment may become disproportionately expensive for oldest populations.

If you recall, a typical regression equation follows a form similar to this:

$$y = \alpha + \beta_1 x$$

To account for a non-linear relationship, we can add a higher order term to the regression model, treating the model as a polynomial. In effect, we will be modeling a relationship like this:

$$y = \alpha + \beta_1 x + \beta_2 x^2$$

The difference between these two models is that an additional beta will be estimated, which is intended to capture the effect of the *x*-squared term. This allows the impact of age to be measured as a function of age squared.

To add the non-linear age to the model, we simply need to create a new variable:

```
> insurance$age2 <- insurance$age^2
```

Then, when we produce our improved model, we'll add both `age` and `age2` to the `lm()` formula using the `expenses ~ age + age2` form. This will allow the model to separate the linear and non-linear impact of age on medical expenses.

Transformation – converting a numeric variable to a binary indicator

Suppose we have a hunch that the effect of a feature is not cumulative, rather it has an effect only after a specific threshold has been reached. For instance, BMI may have zero impact on medical expenditures for individuals in the normal weight range, but it may be strongly related to higher costs for the obese (that is, BMI of 30 or above).

We can model this relationship by creating a binary obesity indicator variable that is 1 if the BMI is at least 30, and 0 if less. The estimated beta for this binary feature would then indicate the average net impact on medical expenses for individuals with BMI of 30 or above, relative to those with BMI less than 30.

To create the feature, we can use the `ifelse()` function, which for each element in a vector tests a specified condition and returns a value depending on whether the condition is true or false. For BMI greater than or equal to 30, we will return `1`, otherwise `0`:

```
> insurance$bmi30 <- ifelse(insurance$bmi >= 30, 1, 0)
```

We can then include the `bmi30` variable in our improved model, either replacing the original `bmi` variable or in addition, depending on whether or not we think the effect of obesity occurs in addition to a separate linear BMI effect. Without good reason to do otherwise, we'll include both in our final model.

If you have trouble deciding whether or not to include a variable, a common practice is to include it and examine the p-value. If the variable is not statistically significant, you have evidence to support excluding it in the future.

Model specification – adding interaction effects

So far, we have only considered each feature's individual contribution to the outcome. What if certain features have a combined impact on the dependent variable? For instance, smoking and obesity may have harmful effects separately, but it is reasonable to assume that their combined effect may be worse than the sum of each one alone.

When two features have a combined effect, this is known as an **interaction**. If we suspect that two variables interact, we can test this hypothesis by adding their interaction to the model. Interaction effects are specified using the R formula syntax. To have the obesity indicator (`bmi30`) and the smoking indicator (`smoker`) interact, we would write a formula in the form `expenses ~ bmi30*smoker`.

The `*` operator is shorthand that instructs R to model `expenses ~ bmi30 + smokeryes + bmi30:smokeryes`. The `:` (colon) operator in the expanded form indicates that `bmi30:smokeryes` is the interaction between the two variables. Note that the expanded form also automatically included the `bmi30` and `smoker` variables as well as the interaction.

Interactions should never be included in a model without also adding each of the interacting variables. If you always create interactions using the `*` operator, this will not be a problem since R will add the required components automatically.

Putting it all together – an improved regression model

Based on a bit of subject matter knowledge of how medical costs may be related to patient characteristics, we developed what we think is a more accurately specified regression formula. To summarize the improvements, we:

- Added a non-linear term for age
- Created an indicator for obesity
- Specified an interaction between obesity and smoking

We'll train the model using the `lm()` function as before, but this time we'll add the newly constructed variables and the interaction term:

```
> ins_model2 <- lm(expenses ~ age + age2 + children + bmi + sex +
                     bmi30*smoker + region, data = insurance)
```

Next, we summarize the results:

```
> summary(ins_model2)
```

The output is shown as follows:

```
Call:
lm(formula = expenses ~ age + age2 + children + bmi + sex + bmi30 *
    smoker + region, data = insurance)

Residuals:
     Min       1Q   Median       3Q      Max
-17297.1  -1656.0  -1262.7   -727.8  24161.6

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)       139.0053  1363.1359   0.102 0.918792
age               -32.6181    59.8250  -0.545 0.585690
age2                3.7387     0.7463   4.999 6.54e-07 ***
children          678.6017   105.8855   6.409 2.03e-10 ***
bmi               119.7715    34.2796   3.494 0.000492 ***
sexmale          -496.7690   244.3713  -2.033 0.042267 *
bmi30            -997.9355   422.9607  -2.359 0.018449 *
smokeryes       13404.5952   439.9591  30.468  < 2e-16 ***
regionnorthwest  -279.1661   349.2826  -0.799 0.424285
regionsoutheast  -828.0345   351.6484  -2.355 0.018682 *
regionsouthwest -1222.1619   350.5314  -3.487 0.000505 ***
bmi30:smokeryes 19810.1534   604.6769  32.762  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4445 on 1326 degrees of freedom
Multiple R-squared:  0.8664,  Adjusted R-squared:  0.8653
F-statistic: 781.7 on 11 and 1326 DF,  p-value: < 2.2e-16
```

The model fit statistics help to determine whether our changes improved the performance of the regression model. Relative to our first model, the R-squared value has improved from 0.75 to about 0.87. Similarly, the adjusted R-squared value, which takes into account the fact that the model grew in complexity, also improved from 0.75 to 0.87. Our model is now explaining 87 percent of the variation in medical treatment costs. Additionally, our theories about the model's functional form seem to be validated. The higher-order `age2` term is statistically significant, as is the obesity indicator, `bmi30`. The interaction between obesity and smoking suggests a massive effect; in addition to the increased costs of over $13,404 for smoking alone, obese smokers spend another $19,810 per year. This may suggest that smoking exacerbates diseases associated with obesity.

Strictly speaking, regression modeling makes some strong assumptions about the data. These assumptions are not as important for numeric forecasting, as the model's worth is not based upon whether it truly captures the underlying process—we simply care about the accuracy of its predictions. However, if you would like to make firm inferences from the regression model coefficients, it is necessary to run diagnostic tests to ensure that the regression assumptions have not been violated. For an excellent introduction to this topic, see Allison PD. *Multiple regression: A primer*. Pine Forge Press; 1998.

Understanding regression trees and model trees

If you recall from *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, a decision tree builds a model much like a flowchart in which decision nodes, leaf nodes, and branches define a series of decisions that are used to classify examples. Such trees can also be used for numeric prediction by making only small adjustments to the tree-growing algorithm. In this section, we will consider only the ways in which trees for numeric prediction differ from trees used for classification.

Trees for numeric prediction fall into two categories. The first, known as **regression trees**, were introduced in the 1980s as part of the seminal **Classification and Regression Tree** (**CART**) algorithm. Despite the name, regression trees do not use linear regression methods as described earlier in this chapter, rather they make predictions based on the average value of examples that reach a leaf.

The CART algorithm is described in detail in Breiman L, Friedman JH, Stone CJ, Olshen RA. *Classification and Regression Trees*. Belmont, CA: Chapman and Hall; 1984.

The second type of trees for numeric prediction are known as **model trees**. Introduced several years later than regression trees, they are lesser-known, but perhaps more powerful. Model trees are grown in much the same way as regression trees, but at each leaf, a multiple linear regression model is built from the examples reaching that node. Depending on the number of leaf nodes, a model tree may build tens or even hundreds of such models. This may make model trees more difficult to understand than the equivalent regression tree, with the benefit that they may result in a more accurate model.

The earliest model tree algorithm, **M5**, is described in Quinlan JR. *Learning with continuous classes*. Proceedings of the 5th Australian Joint Conference on Artificial Intelligence. 1992:343-348.

Adding regression to trees

Trees that can perform numeric prediction offer a compelling yet often overlooked alternative to regression modeling. The strengths and weaknesses of regression trees and model trees relative to the more common regression methods are listed in the following table:

Strengths	Weaknesses
• Combines the strengths of decision trees with the ability to model numeric data • Does not require the user to specify the model in advance • Uses automatic feature selection, which allows the approach to be used with a very large number of features • May fit some types of data much better than linear regression • Does not require knowledge of statistics to interpret the model	• Not as well-known as linear regression • Requires a large amount of training data • Difficult to determine the overall net effect of individual features on the outcome • Large trees can become more difficult to interpret than a regression model

Though traditional regression methods are typically the first choice for numeric prediction tasks, in some cases, numeric decision trees offer distinct advantages. For instance, decision trees may be better suited for tasks with many features or many complex, non-linear relationships among features and outcome. These situations present challenges for regression. Regression modeling also makes assumptions about how numeric data is distributed that are often violated in real-world data. This is not the case for trees.

Trees for numeric prediction are built in much the same way as they are for classification. Beginning at the root node, the data is partitioned using a divide-and-conquer strategy according to the feature that will result in the greatest increase in homogeneity in the outcome after a split is performed. In classification trees, you will recall that homogeneity is measured by entropy, which is undefined for numeric data. Instead, for numeric decision trees, homogeneity is measured by statistics such as variance, standard deviation, or absolute deviation from the mean.

One common splitting criterion is called the **Standard Deviation Reduction** (**SDR**). It is defined by the following formula:

$$\text{SDR} = sd(T) - \sum_i \frac{|T_i|}{|T|} \times sd(T_i)$$

In this formula, the *sd(T)* function refers to the standard deviation of the values in set *T*, while T_1, T_2, ..., T_n are the sets of values resulting from a split on a feature. The $|T|$ term refers to the number of observations in set *T*. Essentially, the formula measures the reduction in standard deviation by comparing the standard deviation pre-split to the weighted standard deviation post-split.

For example, consider the following case in which a tree is deciding whether or not to perform a split on binary feature A or B:

original data	1	1	1	2	2	3	4	5	5	6	6	7	7	7	7
split on feature A	1	1	1	2	2	3	4	5	5	6	6	7	7	7	7
split on feature B	1	1	1	2	2	3	4	5	5	6	6	7	7	7	7
						T_1						T_2			

Using the groups that would result from the proposed splits, we can compute the SDR for A and B as follows. The `length()` function used here returns the number of elements in a vector. Note that the overall group T is named `tee` to avoid overwriting R's built-in `T()` and `t()` functions:

```
> tee <- c(1, 1, 1, 2, 2, 3, 4, 5, 5, 6, 6, 7, 7, 7, 7)
> at1 <- c(1, 1, 1, 2, 2, 3, 4, 5, 5)
> at2 <- c(6, 6, 7, 7, 7, 7)
> bt1 <- c(1, 1, 1, 2, 2, 3, 4)
> bt2 <- c(5, 5, 6, 6, 7, 7, 7, 7)
> sdr_a <- sd(tee) - (length(at1) / length(tee) * sd(at1) +
              length(at2) / length(tee) * sd(at2))
> sdr_b <- sd(tee) - (length(bt1) / length(tee) * sd(bt1) +
              length(bt2) / length(tee) * sd(bt2))
```

Let's compare the SDR of A against the SDR of B:

```
> sdr_a
[1] 1.202815
> sdr_b
[1] 1.392751
```

The SDR for the split on feature A was about 1.2 versus 1.4 for the split on feature B. Since the standard deviation was reduced more for the split on B, the decision tree would use B first. It results in slightly more homogeneous sets than with A.

Suppose that the tree stopped growing here using this one and only split. A regression tree's work is done. It can make predictions for new examples depending on whether the example's value on feature B places the example into group T_1 or T_2. If the example ends up in T_1, the model would predict *mean(bt1)* = 2, otherwise it would predict *mean(bt2)* = 6.25.

In contrast, a model tree would go one step further. Using the seven training examples falling in group T_1 and the eight in T_2, the model tree could build a linear regression model of the outcome versus feature A. Note that Feature B is of no help in building the regression model because all examples at the leaf have the same value of B—they were placed into T_1 or T_2 according to their value of B. The model tree can then make predictions for new examples using either of the two linear models.

To further illustrate the differences between these two approaches, let's work through a real-world example.

Example – estimating the quality of wines with regression trees and model trees

Winemaking is a challenging and competitive business that offers the potential for great profit. However, there are numerous factors that contribute to the profitability of a winery. As an agricultural product, variables as diverse as the weather and the growing environment impact the quality of a varietal. The bottling and manufacturing can also affect the flavor for better or worse. Even the way the product is marketed, from the bottle design to the price point, can affect the customer's perception of taste.

As a consequence, the winemaking industry has heavily invested in data collection and machine learning methods that may assist with the decision science of winemaking. For example, machine learning has been used to discover key differences in the chemical composition of wines from different regions, or to identify the chemical factors that lead a wine to taste sweeter.

More recently, machine learning has been employed to assist with rating the quality of wine—a notoriously difficult task. A review written by a renowned wine critic often determines whether the product ends up on the top or bottom shelf, in spite of the fact that even the expert judges are inconsistent when rating a wine in a blinded test.

In this case study, we will use regression trees and model trees to create a system capable of mimicking expert ratings of wine. Because trees result in a model that is readily understood, this can allow the winemakers to identify the key factors that contribute to better-rated wines. Perhaps more importantly, the system does not suffer from the human elements of tasting, such as the rater's mood or palate fatigue. Computer-aided wine testing may therefore result in a better product as well as more objective, consistent, and fair ratings.

Step 1 – collecting data

To develop the wine rating model, we will use data donated to the UCI Machine Learning Data Repository (`http://archive.ics.uci.edu/ml`) by P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis. The data include examples of red and white Vinho Verde wines from Portugal—one of the world's leading wine-producing countries. Because the factors that contribute to a highly rated wine may differ between the red and white varieties, for this analysis we will examine only the more popular white wines.

To follow along with this example, download the `whitewines.csv` file from the Packt Publishing website and save it to your R working directory. The `redwines.csv` file is also available in case you would like to explore this data on your own.

The white wine data includes information on 11 chemical properties of 4,898 wine samples. For each wine, a laboratory analysis measured characteristics such as acidity, sugar content, chlorides, sulfur, alcohol, pH, and density. The samples were then rated in a blind tasting by panels of no less than three judges on a quality scale ranging from zero (very bad) to 10 (excellent). In the case of judges disagreeing on the rating, the median value was used.

The study by Cortez evaluated the ability of three machine learning approaches to model the wine data: multiple regression, artificial neural networks, and support vector machines. We covered multiple regression earlier in this chapter, and we will learn about neural networks and support vector machines in *Chapter 7, Black Box Methods – Neural Networks and Support Vector Machines*. The study found that the support vector machine offered significantly better results than the linear regression model. However, unlike regression, the support vector machine model is difficult to interpret. Using regression trees and model trees, we may be able to improve the regression results while still having a model that is easy to understand.

To read more about the wine study described here, please refer to Cortez P, Cerdeira A, Almeida F, Matos T, Reis J. *Modeling wine preferences by data mining from physicochemical properties*. Decision Support Systems. 2009; 47:547-553.

Step 2 – exploring and preparing the data

As usual, we will use the `read.csv()` function to load the data into R. Since all of the features are numeric, we can safely ignore the `stringsAsFactors` parameter:

```
> wine <- read.csv("whitewines.csv")
```

The wine data includes 11 features and the quality outcome, as follows:

```
> str(wine)
'data.frame':  4898 obs. of  12 variables:
 $ fixed.acidity       : num  6.7 5.7 5.9 5.3 6.4 7 7.9 ...
 $ volatile.acidity    : num  0.62 0.22 0.19 0.47 0.29 0.12 ...
 $ citric.acid         : num  0.24 0.2 0.26 0.1 0.21 0.41 ...
 $ residual.sugar      : num  1.1 16 7.4 1.3 9.65 0.9 ...
```

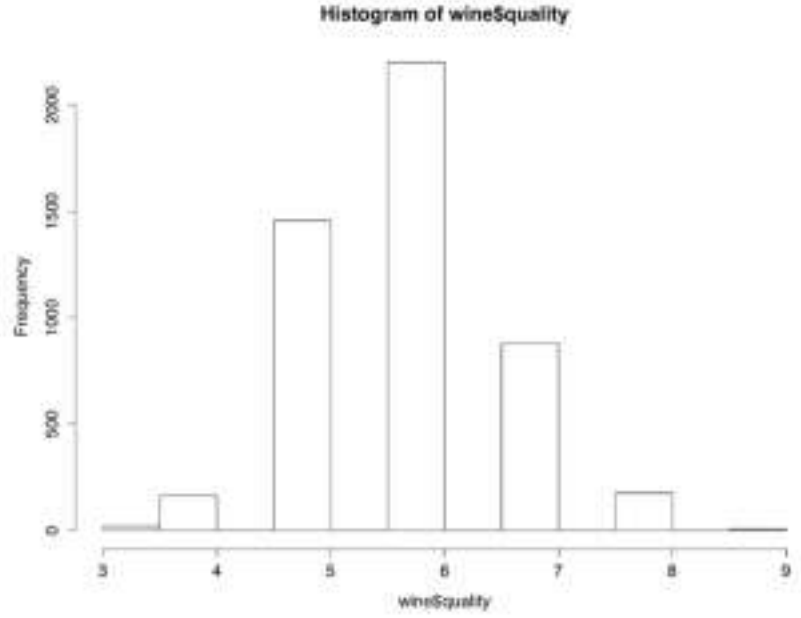
```
$ chlorides           : num  0.039 0.044 0.034 0.036 0.041 ...
$ free.sulfur.dioxide : num  6 41 33 11 36 22 33 17 34 40 ...
$ total.sulfur.dioxide: num  62 113 123 74 119 95 152 ...
$ density             : num  0.993 0.999 0.995 0.991 0.993 ...
$ pH                  : num  3.41 3.22 3.49 3.48 2.99 3.25 ...
$ sulphates           : num  0.32 0.46 0.42 0.54 0.34 0.43 ...
$ alcohol             : num  10.4 8.9 10.1 11.2 10.9 ...
$ quality             : int  5 6 6 4 6 6 6 6 6 7 ...
```

Compared with other types of machine learning models, one of the advantages of trees is that they can handle many types of data without preprocessing. This means we do not need to normalize or standardize the features.

However, a bit of effort to examine the distribution of the outcome variable is needed to inform our evaluation of the model's performance. For instance, suppose that there was a very little variation in quality from wine-to-wine, or that wines fell into a bimodal distribution: either very good or very bad. To check for such extremes, we can examine the distribution of quality using a histogram:

```
> hist(wine$quality)
```

This produces the following figure:



The wine quality values appear to follow a fairly normal, bell-shaped distribution, centered around a value of six. This makes sense intuitively because most wines are of average quality; few are particularly bad or good. Although the results are not shown here, it is also useful to examine the `summary(wine)` output for outliers or other potential data problems. Even though trees are fairly robust with messy data, it is always prudent to check for severe problems. For now, we'll assume that the data is reliable.

Our last step then is to divide into training and testing datasets. Since the `wine` data set was already sorted into random order, we can partition into two sets of contiguous rows as follows:

```
> wine_train <- wine[1:3750, ]
> wine_test <- wine[3751:4898, ]
```

In order to mirror the conditions used by Cortez, we used sets of 75 percent and 25 percent for training and testing, respectively. We'll evaluate the performance of our tree-based models on the testing data to see if we can obtain results comparable to the prior research study.

Step 3 – training a model on the data

We will begin by training a regression tree model. Although almost any implementation of decision trees can be used to perform regression tree modeling, the `rpart` (recursive partitioning) package offers the most faithful implementation of regression trees as they were described by the CART team. As the classic R implementation of CART, the `rpart` package is also well-documented and supported with functions for visualizing and evaluating the `rpart` models.

Install the `rpart` package using the `install.packages("rpart")` command. It can then be loaded into your R session using the `library(rpart)` command. The following syntax will train a tree using the default settings, which typically work fairly well. If you need more finely-tuned settings, refer to the documentation for the control parameters using the `?rpart.control` command.

Regression trees syntax

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

Building the model:

```
m <- rpart(dv ~ iv, data = mydata)
```

- `dv` is the dependent variable in the `mydata` data frame to be modeled
- `iv` is an R formula specifying the independent variables in the `mydata` data frame to use in the model
- `data` specifies the data frame in which the `dv` and `iv` variables can be found

The function will return a regression tree model object that can be used to make predictions.

Making predictions:

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

- `m` is a model trained by the `rpart()` function
- `test` is a data frame containing test data with the same features as the training data used to build the model
- `type` specifies the type of prediction to return, either `"vector"` (for predicted numeric values), `"class"` for predicted classes, or `"prob"` (for predicted class probabilities)

The function will return a vector of predictions depending on the `type` parameter.

Example:

```
wine_model <- rpart(quality ~ alcohol + sulfates,
                    data = wine_train)
wine_predictions <- predict(wine_model, wine_test)
```

Using the R formula interface, we can specify `quality` as the outcome variable and use the dot notation to allow all the other columns in the `wine_train` data frame to be used as predictors. The resulting regression tree model object is named `m.rpart` to distinguish it from the model tree that we will train later:

```
> m.rpart <- rpart(quality ~ ., data = wine_train)
```

For basic information about the tree, simply type the name of the model object:

```
> m.rpart
n= 3750

node), split, n, deviance, yval
      * denotes terminal node

 1) root 3750 2945.53200 5.870933
   2) alcohol< 10.85 2372 1418.86100 5.604975
     4) volatile.acidity>=0.2275 1611  821.30730 5.432030
```

```
      8) volatile.acidity>=0.3025 688  278.97670 5.255814 *
      9) volatile.acidity< 0.3025 923  505.04230 5.563380 *
    5) volatile.acidity< 0.2275 761  447.36400 5.971091 *
  3) alcohol>=10.85 1378 1070.08200 6.328737
    6) free.sulfur.dioxide< 10.5 84   95.55952 5.369048 *
    7) free.sulfur.dioxide>=10.5 1294  892.13600 6.391036
     14) alcohol< 11.76667 629  430.11130 6.173291
       28) volatile.acidity>=0.465 11   10.72727 4.545455 *
       29) volatile.acidity< 0.465 618  389.71680 6.202265 *
     15) alcohol>=11.76667 665  403.99400 6.596992 *
```

For each node in the tree, the number of examples reaching the decision point is listed. For instance, all 3,750 examples begin at the root node, of which 2,372 have `alcohol < 10.85` and 1,378 have `alcohol >= 10.85`. Because alcohol was used first in the tree, it is the single most important predictor of wine quality.

Nodes indicated by `*` are terminal or leaf nodes, which means that they result in a prediction (listed here as `yval`). For example, node 5 has a `yval` of 5.971091. When the tree is used for predictions, any wine samples with `alcohol < 10.85` and `volatile.acidity < 0.2275` would therefore be predicted to have a quality value of 5.97.

A more detailed summary of the tree's fit, including the mean squared error for each of the nodes and an overall measure of feature importance, can be obtained using the `summary(m.rpart)` command.

Visualizing decision trees

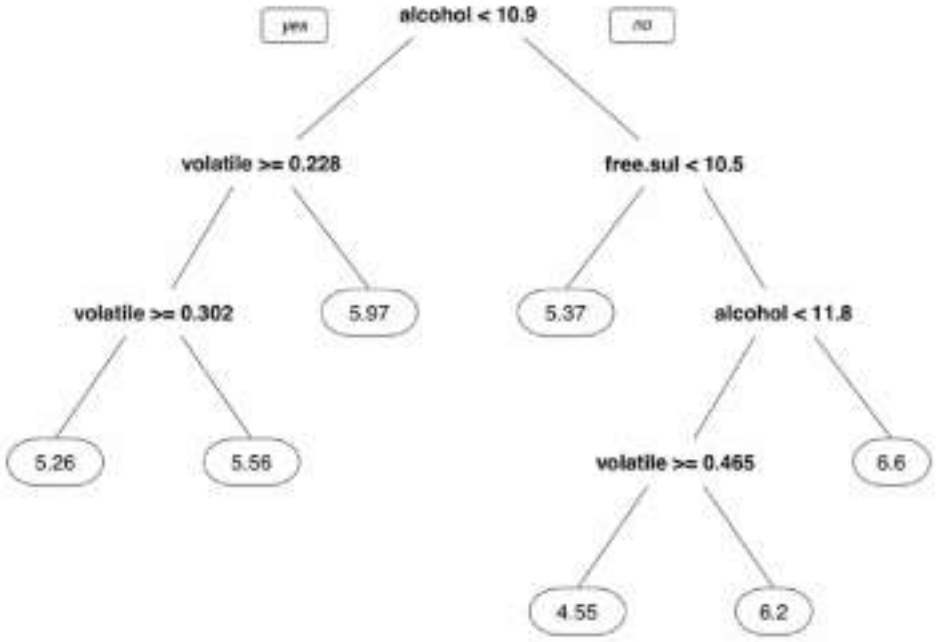
Although the tree can be understood using only the preceding output, it is often more readily understood using visualization. The `rpart.plot` package by Stephen Milborrow provides an easy-to-use function that produces publication-quality decision trees.

For more information on `rpart.plot`, including additional examples of the types of decision tree diagrams that the function can produce, refer to the author's website at `http://www.milbo.org/rpart-plot/`.

After installing the package using the `install.packages("rpart.plot")` command, the `rpart.plot()` function produces a tree diagram from any `rpart` model object. The following commands plot the regression tree we built earlier:

```
> library(rpart.plot)
> rpart.plot(m.rpart, digits = 3)
```

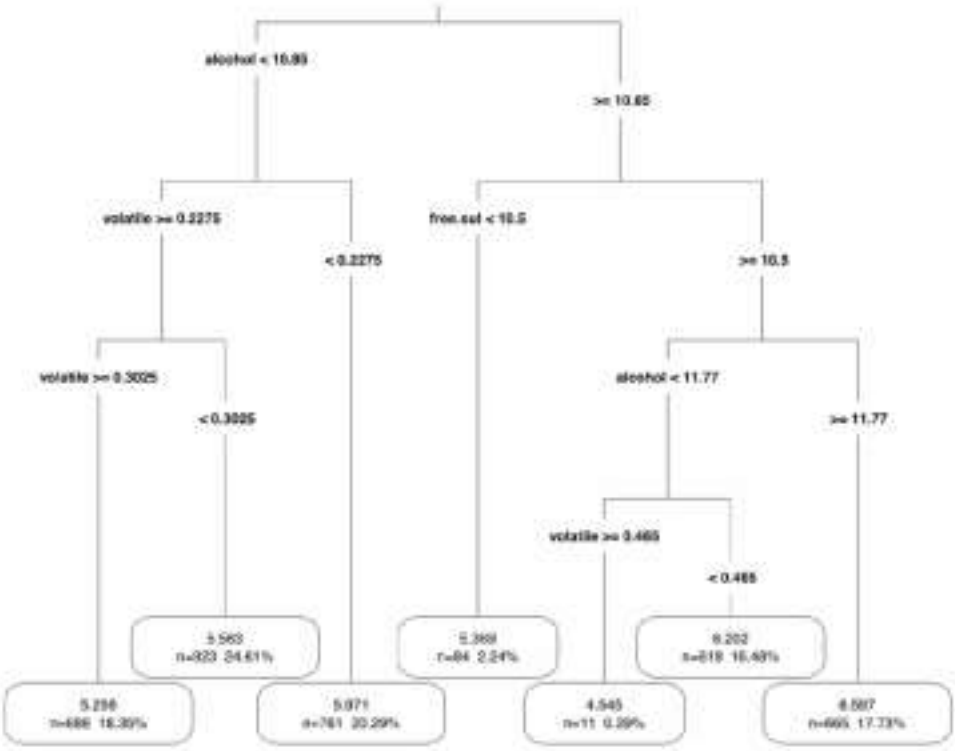
The resulting tree diagram is as follows:



In addition to the `digits` parameter that controls the number of numeric digits to include in the diagram, many other aspects of the visualization can be adjusted. The following command shows just a few of the useful options: The `fallen.leaves` parameter forces the leaf nodes to be aligned at the bottom of the plot, while the `type` and `extra` parameters affect the way the decisions and nodes are labeled:

```
> rpart.plot(m.rpart, digits = 4, fallen.leaves = TRUE,
             type = 3, extra = 101)
```

The result of these changes is a very different looking tree diagram:



Visualizations like these may assist with the dissemination of regression tree results, as they are readily understood even without a mathematics background. In both cases, the numbers shown in the leaf nodes are the predicted values for the examples reaching that node. Showing the diagram to the wine producers may thus help to identify the key factors that predict the higher rated wines.

Step 4 – evaluating model performance

To use the regression tree model to make predictions on the test data, we use the `predict()` function. By default, this returns the estimated numeric value for the outcome variable, which we'll save in a vector named `p.rpart`:

```
> p.rpart <- predict(m.rpart, wine_test)
```

A quick look at the summary statistics of our predictions suggests a potential problem; the predictions fall on a much narrower range than the true values:

```
> summary(p.rpart)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.545   5.563   5.971   5.893   6.202   6.597
> summary(wine_test$quality)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.000   5.000   6.000   5.901   6.000   9.000
```

This finding suggests that the model is not correctly identifying the extreme cases, in particular the best and worst wines. On the other hand, between the first and third quartile, we may be doing well.

The correlation between the predicted and actual quality values provides a simple way to gauge the model's performance. Recall that the `cor()` function can be used to measure the relationship between two equal-length vectors. We'll use this to compare how well the predicted values correspond to the true values:

```
> cor(p.rpart, wine_test$quality)
[1] 0.5369525
```

A correlation of 0.54 is certainly acceptable. However, the correlation only measures how strongly the predictions are related to the true value; it is not a measure of how far off the predictions were from the true values.

Measuring performance with the mean absolute error

Another way to think about the model's performance is to consider how far, on average, its prediction was from the true value. This measurement is called the **mean absolute error** (**MAE**). The equation for MAE is as follows, where n indicates the number of predictions and e_i indicates the error for prediction i:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|e_i$$

As the name implies, this equation takes the mean of the absolute value of the errors. Since the error is just the difference between the predicted and actual values, we can create a simple `MAE()` function as follows:

```
> MAE <- function(actual, predicted) {
    mean(abs(actual - predicted))
}
```

The MAE for our predictions is then:

```
> MAE(p.rpart, wine_test$quality)
[1] 0.5872652
```

This implies that, on average, the difference between our model's predictions and the true quality score was about 0.59. On a quality scale from zero to 10, this seems to suggest that our model is doing fairly well.

On the other hand, recall that most wines were neither very good nor very bad; the typical quality score was around five to six. Therefore, a classifier that did nothing but predict the mean value may still do fairly well according to this metric.

The mean quality rating in the training data is as follows:

```
> mean(wine_train$quality)
[1] 5.870933
```

If we predicted the value 5.87 for every wine sample, we would have a mean absolute error of only about 0.67:

```
> MAE(5.87, wine_test$quality)
[1] 0.6722474
```

Our regression tree (*MAE = 0.59*) comes closer on average to the true quality score than the imputed mean (*MAE = 0.67*), but not by much. In comparison, Cortez reported an MAE of 0.58 for the neural network model and an MAE of 0.45 for the support vector machine. This suggests that there is room for improvement.

Step 5 – improving model performance

To improve the performance of our learner, let's try to build a model tree. Recall that a model tree improves on regression trees by replacing the leaf nodes with regression models. This often results in more accurate results than regression trees, which use only a single value for prediction at the leaf nodes.

The current state-of-the-art in model trees is the **M5' algorithm (M5-prime)** by Y. Wang and I.H. Witten, which is a variant of the original M5 model tree algorithm proposed by J.R. Quinlan in 1992.

For more information on the M5' algorithm, see Wang Y, Witten IH. *Induction of model trees for predicting continuous classes*. Proceedings of the Poster Papers of the European Conference on Machine Learning. 1997.

The M5 algorithm is available in R via the `RWeka` package and the `M5P()` function. The syntax of this function is shown in the following table. Be sure to install the `RWeka` package if you haven't already. Because of its dependence on Java, the installation instructions are included in *Chapter 1, Introducing Machine Learning*.

Model trees syntax

using the `M5P()` function in the `RWeka` package

Building the model:

```
m <- M5P(dv ~ iv, data = mydata)
```

- `dv` is the dependent variable in the `mydata` data frame to be modeled
- `iv` is an R formula specifying the independent variables in the `mydata` data frame to use in the model
- `data` specifies the data frame in which the `dv` and `iv` variables can be found

The function will return a model tree object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test)
```

- `m` is a model trained by the `M5P()` function
- `test` is a data frame containing test data with the same features as the training data used to build the model

The function will return a vector of predicted numeric values.

Example:

```
wine_model <- M5P(quality ~ alcohol + sulfates,
                  data = wine_train)
wine_predictions <- predict(wine_model, wine_test)
```

We'll fit the model tree using essentially the same syntax as we used for the regression tree:

```
> library(RWeka)
> m.m5p <- M5P(quality ~ ., data = wine_train)
```

The tree itself can be examined by typing its name. In this case, the tree is very large and only the first few lines of output are shown:

```
> m.m5p
M5 pruned model tree:
(using smoothed linear models)

alcohol <= 10.85 :
|   volatile.acidity <= 0.238 :
|   |   fixed.acidity <= 6.85 : LM1 (406/66.024%)
|   |   fixed.acidity >  6.85 :
|   |   |   free.sulfur.dioxide <= 24.5 : LM2 (113/87.697%)
```

You will note that the splits are very similar to the regression tree that we built earlier. Alcohol is the most important variable, followed by volatile acidity and free sulfur dioxide. A key difference, however, is that the nodes terminate not in a numeric prediction, but a linear model (shown here as `LM1` and `LM2`).

The linear models themselves are shown later in the output. For instance, the model for `LM1` is shown in the forthcoming output. The values can be interpreted exactly the same as the multiple regression models we built earlier in this chapter. Each number is the net effect of the associated feature on the predicted wine quality. The coefficient of `0.266` for fixed acidity implies that for an increase of 1 unit of acidity, the wine quality is expected to increase by `0.266`:

```
LM num: 1
quality =
  0.266 * fixed.acidity
  - 2.3082 * volatile.acidity
  - 0.012 * citric.acid
  + 0.0421 * residual.sugar
  + 0.1126 * chlorides
  + 0 * free.sulfur.dioxide
  - 0.0015 * total.sulfur.dioxide
  - 109.8813 * density
```

```
  + 0.035 * pH
  + 1.4122 * sulphates
  - 0.0046 * alcohol
  + 113.1021
```

It is important to note that the effects estimated by `LM1` apply only to wine samples reaching this node; a total of 36 linear models were built in this model tree, each with different estimates of the impact of fixed acidity and the other 10 features.

For statistics on how well the model fits the training data, the `summary()` function can be applied to the M5P model. However, note that since these statistics are based on the training data, they should be used only as a rough diagnostic:

```
> summary(m.m5p)

=== Summary ===

Correlation coefficient                  0.6666
Mean absolute error                      0.5151
Root mean squared error                  0.6614
Relative absolute error                 76.4921 %
Root relative squared error             74.6259 %
Total Number of Instances             3750
```

Instead, we'll look at how well the model performs on the unseen test data. The `predict()` function gets us a vector of predicted values:

```
> p.m5p <- predict(m.m5p, wine_test)
```

The model tree appears to be predicting a wider range of values than the regression tree:

```
> summary(p.m5p)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.389   5.430   5.863   5.874   6.305   7.437
```

The correlation also seems to be substantially higher:

```
> cor(p.m5p, wine_test$quality)
[1] 0.6272973
```

Furthermore, the model has slightly reduced the mean absolute error:

```
> MAE(wine_test$quality, p.m5p)
[1] 0.5463023
```

Although we did not improve a great deal beyond the regression tree, we surpassed the performance of the neural network model published by Cortez, and we are getting closer to the published mean absolute error value of 0.45 for the support vector machine model, all by using a much simpler learning method.

Not surprisingly, we have confirmed that predicting the quality of wines is a difficult problem; wine tasting, after all, is inherently subjective. If you would like additional practice, you may try revisiting this problem after reading *Chapter 11, Improving Model Performance*, which covers additional techniques that may lead to better results.

Summary

In this chapter, we studied two methods for modeling numeric data. The first method, linear regression, involves fitting straight lines to data. The second method uses decision trees for numeric prediction. The latter comes in two forms: regression trees, which use the average value of examples at leaf nodes to make numeric predictions; and model trees, which build a regression model at each leaf node in a hybrid approach that is, in some ways, the best of both worlds.

We used linear regression modeling to calculate the expected medical costs for various segments of the population. Because the relationship between the features and the target variable are well-described by the estimated regression model, we were able to identify certain demographics, such as smokers and the obese, who may need to be charged higher insurance rates to cover the higher-than-average medical expenses.

Regression trees and model trees were used to model the subjective quality of wines from measureable characteristics. In doing so, we learned how regression trees offer a simple way to explain the relationship between features and a numeric outcome, but the more complex model trees may be more accurate. Along the way, we learned several methods for evaluating the performance of numeric models.

In stark contrast to this chapter, which covered machine learning methods that result in a clear understanding of the relationships between the input and the output, the next chapter covers methods that result in nearly-incomprehensible models. The upside is that they are extremely powerful techniques—among the most powerful stock classifiers—that can be applied to both classification and numeric prediction problems.