5
Divide and Conquer – Classification Using Decision Trees and Rules

To make a difficult decision, some people weigh their options by making lists of pros and cons for each possibility. Suppose a job seeker was deciding between several offers, some closer or further from home, with various levels of pay and benefits. He or she might create a list with the features of each position. Based on these features, rules can be created to eliminate some options. For instance, "if I have a commute longer than an hour, then I will be unhappy", or "if I make less than $50k, I won't be able to support my family." The difficult decision of predicting future happiness can be reduced to a series of small, but increasingly specific choices.

This chapter covers decision trees and rule learners—two machine learning methods that apply a similar strategy of dividing data into smaller and smaller portions to identify patterns that can be used for prediction. The knowledge is then presented in the form of logical structures that can be understood without any statistical knowledge. This aspect makes these models particularly useful for business strategy and process improvement.

By the end of this chapter, you will learn:

- The strategy each method employs to tackle the problem of partitioning data into interesting segments
- Several implementations of decision trees and classification rule learners, including the C5.0, 1R, and RIPPER algorithms

- How to use these algorithms for performing real-world classification tasks such as identifying risky bank loans and poisonous mushrooms

We will begin by examining decision trees and follow with a look at classification rules. Lastly, we'll wrap up with a summary of what we learned and preview later chapters, which discuss methods that use trees and rules as a foundation for other advanced machine learning techniques.

Understanding decision trees

As you might intuit from the name, decision tree learners build a model in the form of a **tree structure**. The model itself comprises a series of logical decisions, similar to a flowchart, with **decision nodes** that indicate a decision to be made on an attribute. These split into **branches** that indicate the decision's choices. The tree is terminated by **leaf nodes** (also known as terminal nodes) that denote the result of following a combination of decisions.

Data that is to be classified begin at the **root node** where it is passed through the various decisions in the tree according to the values of its features. The path that the data takes funnels each record into a leaf node, which assigns it a predicted class.

As the decision tree is essentially a flowchart, it is particularly appropriate for applications in which the classification mechanism needs to be transparent for legal reasons or the results need to be shared in order to facilitate decision making. Some potential uses include:

- Credit scoring models in which the criteria that causes an applicant to be rejected need to be well-specified
- Marketing studies of customer churn or customer satisfaction that will be shared with management or advertising agencies
- Diagnosis of medical conditions based on laboratory measurements, symptoms, or rate of disease progression

Although the previous applications illustrate the value of trees for informing decision processes, that is not to suggest that their utility ends there. In fact, decision trees are perhaps the single most widely used machine learning technique, and can be applied for modeling almost any type of data—often with unparalleled performance.

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 if the data has a large number of numeric features. These cases may result in a very large number of decisions and an overly complex tree.

Divide and conquer

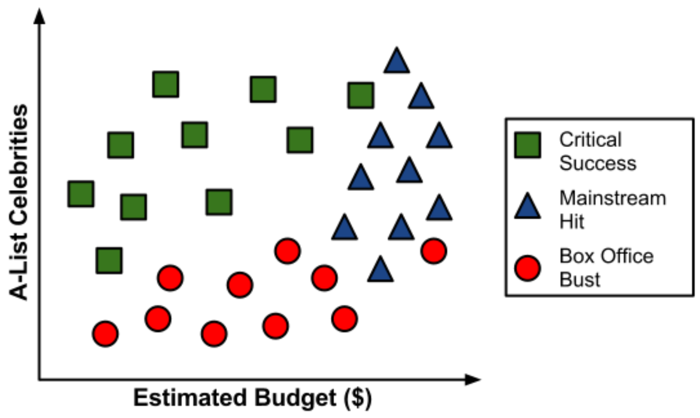
Decision trees are built using a heuristic called **recursive partitioning**. This approach is generally known as **divide and conquer** because it uses the feature values to split the data into smaller and smaller subsets of similar classes.

Beginning at the root node, which represents the entire dataset, the algorithm chooses a feature that is the most predictive of the target class. The examples are then partitioned into groups of distinct values of this feature; this decision forms the first set of tree branches. The algorithm continues to divide-and-conquer the nodes, choosing the best candidate feature each time until a stopping criterion is reached. This might occur at a node if:

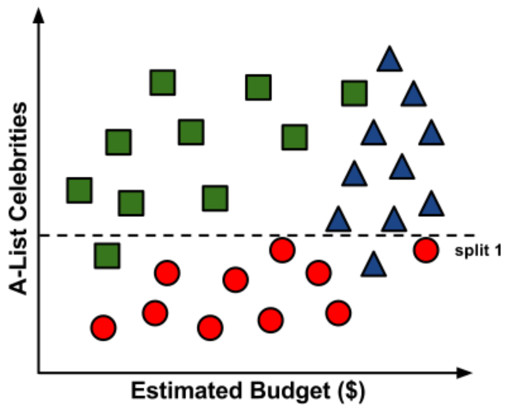
- All (or nearly all) of the examples at the node have the same class
- There are no remaining features to distinguish among examples
- The tree has grown to a predefined size limit

To illustrate the tree building process, let's consider a simple example. Imagine that you are working for a Hollywood film studio, and your desk is piled high with screenplays. Rather than read each one cover-to-cover, you decide to develop a decision tree algorithm to predict whether a potential movie would fall into one of three categories: mainstream hit, critic's choice, or box office bust.

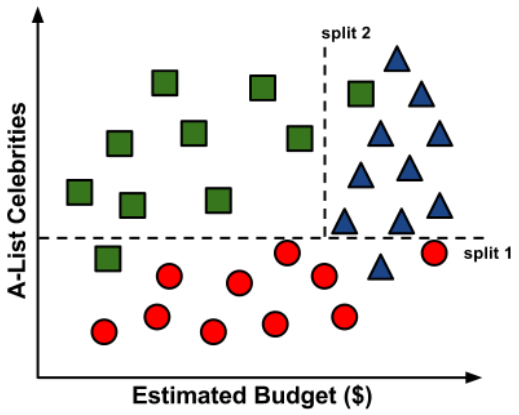
To gather data for your model, you turn to the studio archives to examine the previous ten years of movie releases. After reviewing the data for 30 different movie scripts, a pattern emerges. There seems to be a relationship between the film's proposed shooting budget, the number of A-list celebrities lined up for starring roles, and the categories of success. A scatter plot of this data might look something like the following diagram:



To build a simple decision tree using this data, we can apply a divide-and-conquer strategy. Let's first split the feature indicating the number of celebrities, partitioning the movies into groups with and without a low 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:

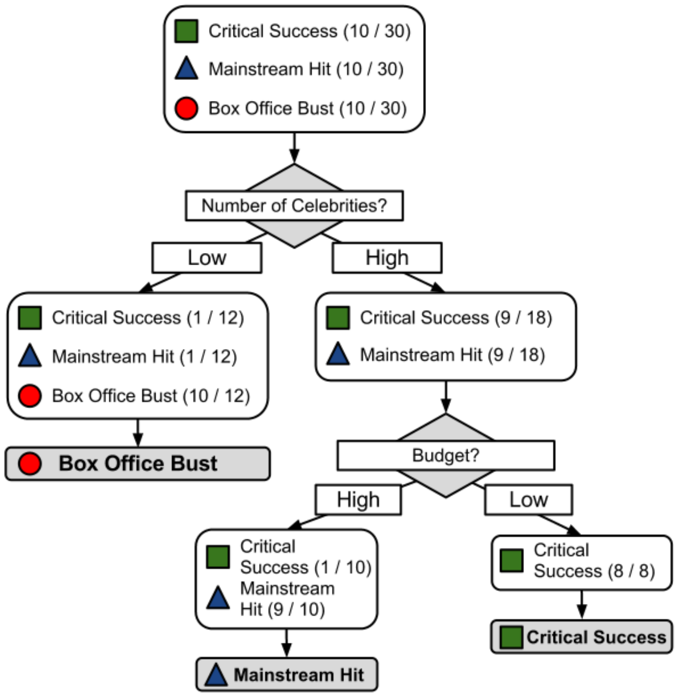


At this point we have partitioned the data into three groups. The group at the top-left corner of the diagram is composed entirely of critically-acclaimed films. This group is distinguished by a high number of celebrities and a relatively low budget. At the top-right corner, the majority of movies are 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 the data by splitting it based on increasingly specific ranges of budget and celebrity counts until each of the incorrectly classified values resides in its own, perhaps tiny partition. Since the data can continue to be split until there are no distinguishing features within a partition, a decision tree can be prone to be overfitting for the training data with overly-specific decisions. We'll avoid this by stopping the algorithm here since more than 80 percent of the examples in each group are from a single class.

> You might have noticed that diagonal lines could 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 decisions such as "if the number of celebrities is greater than the estimated budget, 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 its success or failure has been predicted. In no time, you will be able to classify the backlog of scripts and get back to more important work such as writing an 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 for building decision tree models automatically.

The C5.0 decision tree algorithm

There are numerous implementations of decision trees, but one of the most well-known 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 ID3 (Iterative Dichotomiser 3) 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 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 the 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 for producing 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 can handle numeric or nominal features, missing data • Uses only the most important features • Can be used on data with relatively few training examples or a very large number • 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 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

Earlier in this chapter, we followed a simple example illustrating how a decision tree models data using 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 feature values that split the data in such a way that partitions contained examples primarily of a single class. If the segments of data contain only a single class, they are considered **pure**. There are many different measurements of purity for identifying splitting criteria.

C5.0 uses **entropy** for measuring purity. The entropy of a sample of data indicates how mixed the class values are; the minimum value of 0 indicates that the sample is completely homogenous, while 1 indicates the maximum amount of disorder. The definition of entropy is specified by:

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

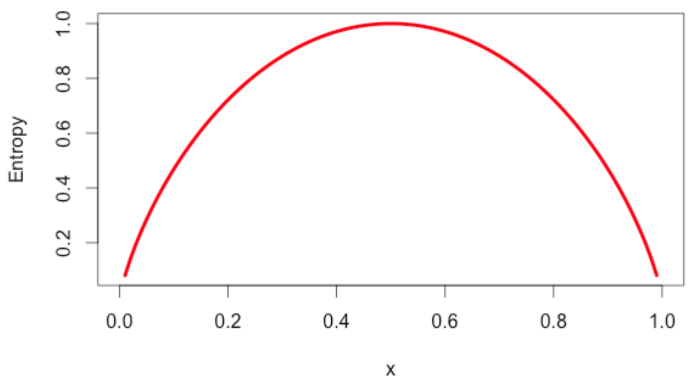
In the entropy formula, for a given segment of data (*S*), the term *c* refers to the number of different class levels, and p_i refers to the proportion of values falling into class level *i*. For example, suppose we have a partition of data with two classes: red (60 percent), and white (40 percent).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

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

We can examine the entropy for all possible two-class arrangements. If we know 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 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 the maximum entropy. As one class increasingly dominates the other, the entropy reduces to zero.

Given this measure of purity, the algorithm must still decide which feature to split upon. For this, the algorithm uses entropy to calculate the change in homogeneity resulting from a split on each possible feature. The calculation is 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):

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

The 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 that 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 entropy of each of the *n* partitions weighted by the proportion of examples falling in that partition (w_i).

The higher the information gain, the better a feature is at creating homogeneous groups after a split on that 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 the entropy after the split is zero, which means that the decision results in completely homogeneous groups.

The previous formulae assume nominal features, but decision trees use information gain for splitting on numeric features as well. A common practice is testing various splits that divide the values into groups greater than or less than a threshold. This reduces the numeric feature into a two-level categorical feature and information gain can be calculated easily. The numeric threshold 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: *An Empirical Comparison of Selection Measures for Decision-Tree Induction*, *Machine Learning, Vol. 3*, pp. 319-342, by *J. Mingers* (1989).

Pruning the decision tree

A decision tree can continue to grow indefinitely, choosing splitting features and dividing 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 have been 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 if 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 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 too large, then using pruning criteria based on the error rates at the 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 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: *A comparative analysis of methods for pruning decision trees. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 19*, pp. 476-491, by *F. Esposito, D. Malerba, and G. Semeraro* (1997).

One of the benefits of the C5.0 algorithm is that it is opinionated about pruning—it takes care of many of the decisions, automatically using fairly reasonable defaults. Its overall strategy is to postprune the tree. It first grows a large tree that overfits the training data. Later, 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 performance on the 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 has highlighted the importance of transparency and rigor in banking practices. As the availability of credit has been limited, banks are increasingly tightening their lending systems and turning 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 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 for instantly approving credit applications on the telephone and the 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 make an applicant at 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 about the applicant.

Data with these characteristics are 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. They represent loans obtained from a credit agency in Germany.

> The data presented in this chapter has been modified slightly from the original one for eliminating 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 of loans, plus a combination of numeric and nominal features indicating characteristics of the loan and the loan applicant. A class variable indicates whether the loan went into default. Let's see if we can determine any patterns that predict this outcome.

Step 2 – exploring and preparing the data

As we have done previously, we will import the data using the `read.csv()` function. We will ignore the `stringsAsFactors` option (and therefore use the default value, `TRUE`) as the majority of features in the data are nominal. We'll also look at the structure of the credit data frame we created:

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

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

```
'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 some of the `table()` output for a couple of features of loans that seem likely to predict a default. The `checking_balance` and `savings_balance` features indicate the applicant's checking and savings account balance, and 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
```

Since the loan data was obtained from Germany, the currency is recorded in Deutsche Marks (DM). It seems like a safe assumption that larger checking and savings account balances should be related to a reduced chance of loan default.

Some of the loan's features are numeric, such as its term (`months_loan_duration`), and the amount of credit requested (`amount`).

```
> 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 amount of 2,320 DM.

The `default` variable 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 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 likely to default, so that this number can be reduced.

Data preparation – creating random training and test datasets

As we have done in 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, our data here is not randomly ordered. Suppose that the bank had sorted the data by the loan amount, with the largest loans at the end of the file. If we use the first 90 percent for training and the remaining 10 percent for testing, we would be building 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 randomly ordering our credit data frame prior to splitting. The `order()` function is used to rearrange a list of items in ascending or descending order. If we combine this with a function to generate a list of random numbers, we can generate a randomly-ordered list. For random number generation, we'll use the `runif()` function, which by default generates a sequence of random numbers between 0 and 1.

> If you're trying to figure out where the `runif()` function gets its name, the answer is due to the fact that it chooses numbers from a uniform distribution, which we learned about in *Chapter 2, Managing and Understanding Data.*

The following command creates a randomly-ordered `credit` data frame. The `set.seed()` function is used to generate random numbers in a predefined sequence, starting from a position known as a **seed** (set here to the arbitrary value `12345`). It may seem that this defeats the purpose of generating random numbers, but there is a good reason for doing it this way. The `set.seed()` function ensures that if the analysis is repeated, an identical result is obtained.

```
> set.seed(12345)
> credit_rand <- credit[order(runif(1000)), ]
```

The `runif(1000)` command generates a list of 1,000 random numbers. We need exactly 1,000 random numbers because there are 1,000 records in the `credit` data frame. The `order()` function then returns a vector of numbers indicating the sorted position of the 1,000 random numbers. We then use these positions to select rows in the `credit` data frame and store in a new data frame named `credit_rand`.

> To better understand how this function works, note that `order(c(0.5, 0.25, 0.75, 0.1))` returns the sequence `4 1 2 3` because the smallest number (`0.1`) appears fourth, the second smallest (`0.25`) appears first, and so on.

To confirm that we have the same data frame sorted differently, we'll compare values on the `amount` feature across the two data frames. The following code shows the summary statistics:

```
> summary(credit$amount)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    250    1366    2320    3271    3972   18420
> summary(credit_rand$amount)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    250    1366    2320    3271    3972   18420
```

We can use the `head()` function to examine the first few values in each data frame:

```
> head(credit$amount)
[1] 1169 5951 2096 7882 4870 9055
> head(credit_rand$amount)
[1] 1199 2576 1103 4020 1501 1568
```

Since the summary statistics are identical while the first few values are different, this suggests that our random shuffle worked correctly.

> If your results do not match exactly with the previous ones, ensure that you run the command `set.seed(214805)` immediately prior to creating the `credit_rand` data frame.

Now, we can split into training (90 percent or 900 records), and test data (10 percent or 100 records) as we have done in previous analyses:

```
> credit_train <- credit_rand[1:900, ]
> credit_test  <- credit_rand[901:1000, ]
```

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

```
> prop.table(table(credit_train$default))
       no       yes
0.7022222 0.2977778
> prop.table(table(credit_test$default))
  no  yes
0.68 0.32
```

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

Step 3 – training a model on the data

We will use the C5.0 algorithm in the `C50` package for training 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 for building decision trees. Compared to the machine learning approaches we have used previously, the C5.0 algorithm offers many more ways to tailor the model to a particular learning problem, but even more options are available. 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 (by default, 1)
- `costs` is an optional matrix specifying costs associated with 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 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:

```
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 class variable, `default`, so we need to exclude it from the training data frame as an independent variable, 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 object. We can see some basic data about the tree by typing its name:

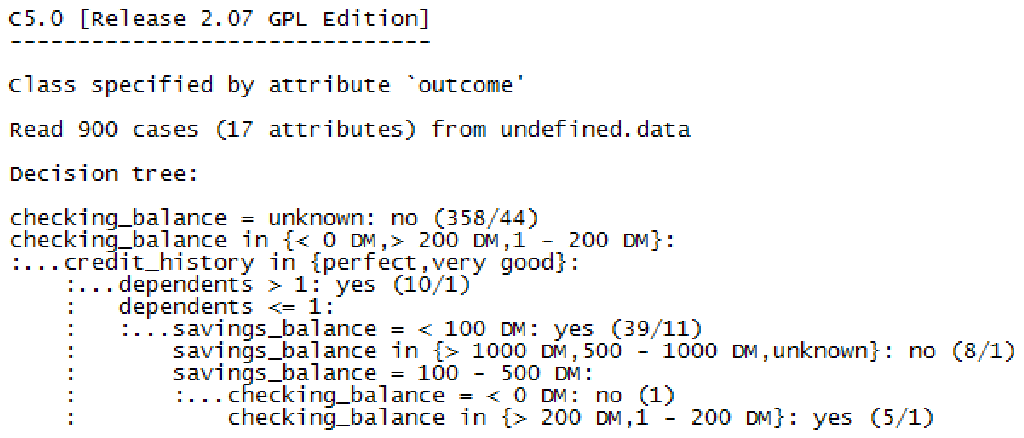
```
> credit_model


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


Classification Tree
Number of samples: 900
Number of predictors: 16


Tree size: 67
```

The preceding text shows some simple facts about the tree, including the function call that generated it, the number of features (that is, `predictors`), and examples (that is, `samples`) used to grow the tree. Also listed is the tree size of 67, which indicates that the tree is 67 decisions deep—quite a bit larger than the trees we've looked at so far!

To see the 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 = unknown: no (358/44)
checking_balance in {< 0 DM,> 200 DM,1 - 200 DM}:
:...credit_history in {perfect,very good}:
    :...dependents > 1: yes (10/1)
    :   dependents <= 1:
    :   :...savings_balance = < 100 DM: yes (39/11)
    :       savings_balance in {> 1000 DM,500 - 1000 DM,unknown}: no (8/1)
    :       savings_balance = 100 - 500 DM:
    :       :...checking_balance = < 0 DM: no (1)
    :           checking_balance in {> 200 DM,1 - 200 DM}: yes (5/1)
```

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

1. If the checking account balance is unknown, then classify as **not likely to default**.
2. Otherwise, if the checking account balance is less than zero DM, between one and 200 DM, or greater than 200 DM and…
3. The credit history is very good or perfect, and…
4. There is more than one dependent, 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, `(358/44)` indicates that of the 358 examples reaching the decision, 44 were incorrectly classified as `no`, that is, not likely to default. In other words, 44 applicants actually defaulted in spite of the model's prediction to the contrary.

Some of the tree's decisions do not seem to make logical sense. 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.

After the tree output, the `summary(credit_model)` 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
    66  125(13.9%)   <<


   (a)   (b)    <-classified as
  ----  ----
   609    23    (a): class no
   102   166    (b): class yes
```

The `Errors` field notes that the model correctly classified all but 125 of the 900 training instances for an error rate of 13.9 percent. A total of 23 actual `no` values were incorrectly classified as `yes` (false positives), while 102 `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

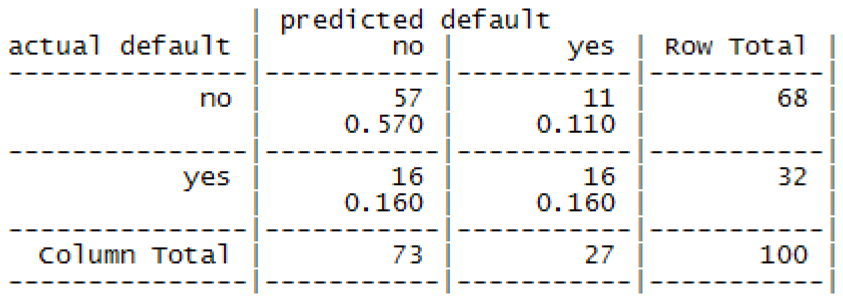
To apply our decision tree to the test dataset, we use the `predict()` function as shown in the following line of code:

```
> 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 |        57 |        11 |        68 |
               |     0.570 |     0.110 |           |
---------------|-----------|-----------|-----------|
           yes |        16 |        16 |        32 |
               |     0.160 |     0.160 |           |
---------------|-----------|-----------|-----------|
  Column Total |        73 |        27 |       100 |
---------------|-----------|-----------|-----------|
```

Out of the 100 test loan application records, our model correctly predicted that 57 did not default and 16 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 50 percent of the 32 loan defaults in the test data. Unfortunately, this type of error is a potentially very costly mistake. 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 68 percent of the time—a result not much worse than our model, 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 default. 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 mistakes.

Boosting the accuracy of decision trees

One way the C5.0 algorithm improved upon the C4.5 algorithm was by adding **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 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 Understanding Rule Learners* (The MIT Press, 2012).

As boosting can be applied more generally to any machine learning algorithm, it is covered in more 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 one of the learners alone. Each of the models has a unique set of strengths and weaknesses, and may be better or worse at 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: 56
```

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.

Let's take a look at the performance on our training data:

```
> summary(credit_boost10)


     (a)   (b)    <-classified as
    ----  ----
     626     6    (a): class no
      25   243    (b): class yes
```

The classifier made 31 mistakes on 900 training examples for an error rate of 3.4 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 |        60 |         8 |        68 |
               |     0.600 |     0.080 |           |
---------------|-----------|-----------|-----------|
           yes |        15 |        17 |        32 |
               |     0.150 |     0.170 |           |
---------------|-----------|-----------|-----------|
  Column Total |        75 |        25 |       100 |
---------------|-----------|-----------|-----------|
```

Here, we reduced the total error rate from 27 percent prior to boosting down to 23 percent in the boosted model. It does not seem like a large gain, but it is reasonably close to the 25 percent reduction we hoped for. On the other hand, the model is still not doing well at predicting defaults, getting *15 / 32 = 47%* wrong. 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.

That 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 some mistakes more costly 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. The few years' worth of interest that the bank would earn from a risky loan is far outweighed by the massive loss it would take if the money was never 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 many times more costly each error is, relative to any other. Suppose we believe that a loan default costs the bank four times as much as a missed opportunity. Our cost matrix then could be defined as:

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

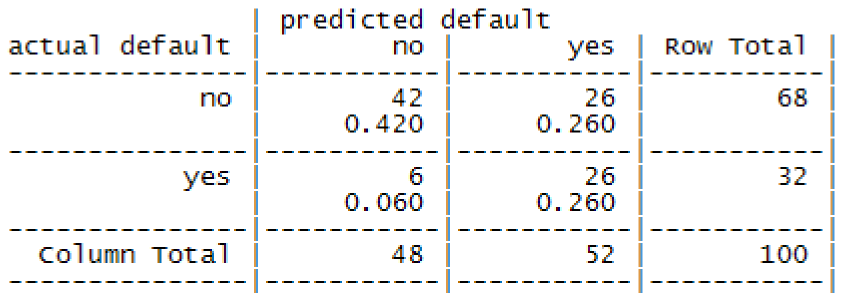
This creates a matrix with two rows and two columns, arranged somewhat differently than the confusion matrixes we have been working with. The value `1` indicates `no` and the value `2` indicates `yes`. Rows are for predicted values and columns are for actual values:

```
> error_cost
     [,1] [,2]
[1,]    0    4
[2,]    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 before:

```
> 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 |        42 |        26 |        68 |
               |     0.420 |     0.260 |           |
---------------|-----------|-----------|-----------|
           yes |         6 |        26 |        32 |
               |     0.060 |     0.260 |           |
---------------|-----------|-----------|-----------|
  Column Total |        48 |        52 |       100 |
---------------|-----------|-----------|-----------|
```

Compared to our best boosted model, this version makes more mistakes overall: 32 percent here versus 23 percent in the boosted case. However, the types of mistakes vary dramatically. Where the previous models incorrectly classiifed nearly half of the defaults incorrectly, in this model, only 25 percent of the 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.