**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.

*Chapter 5*

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.

**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.

**[ 129 ]**

page146image3768816page146image3769024page146image3769648

*Divide and Conquer – Classification Using Decision Trees and Rules*

**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**

**$ purpose**

**$ amount**

**: Factor w/ 5 levels "critical","good",..**

**: Factor w/ 6 levels "business","car",..**

**: 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.

**[ 130 ]**

page147image3774224

*Chapter 5*

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**

**4.0 12.0 18.0**

**> summary(credit$amount)**

**Min. 1st Qu. Median**

**250 1366 2320**

**Mean 3rd Qu. Max.**

**20.9 24.0 72.0**

**Mean 3rd Qu. Max.**

**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**

simulate new applicants. *Divide and Conquer – Classification Using Decision Trees and Rules*

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**

**[ 132 ]**

page149image1686640page149image1686848page149image1687056page149image1687680

*Chapter 5*

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**

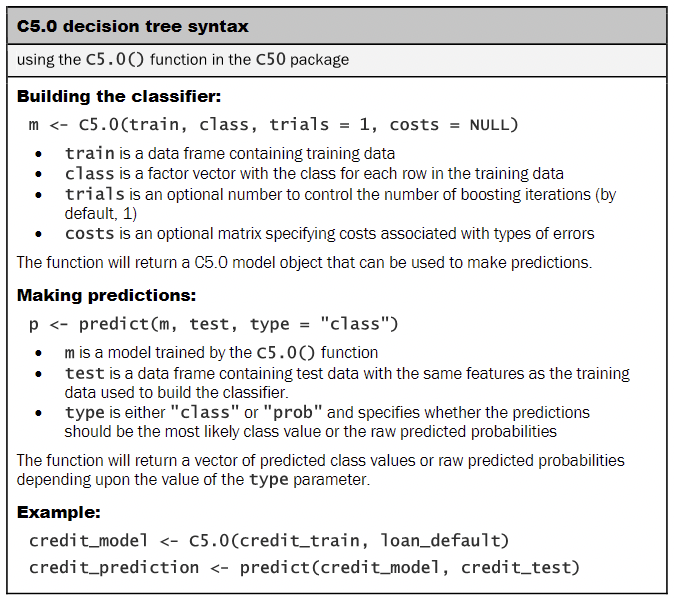
This appears to be a fairly equal split, so we can now build our decision tree.

**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).

*Divide and Conquer – Classification Using Decision Trees and Rules*

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.



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)**

**[ 134 ]**

page151image1689344

*Chapter 5*

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:

**[ 135 ]**

page152image1690176

*Divide and Conquer – Classification Using Decision Trees and Rules*

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**

**----------------**

page153image3793984page153image3794192

**Size 66**

**Errors**

**125(13.9%) <<**

**(a)  
---- ----**

**609 23**

102 166

**<-classified as**

**(a): class no**

(b): class yes

**(b)**

**[ 136 ]**

page153image3794816

*Chapter 5*

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).

**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)**

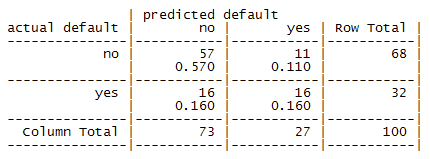
This creates a vector of predicted class values, which we can compare to the actual class values using the CrossTable() function in the gmodels package. Setting the prop.c and prop.r parameters to FALSE removes the column and row percentages from the table. The remaining percentage (prop.t) indicates 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'))**



**[ 137 ]**

page154image1697040

*Divide and Conquer – Classification Using Decision Trees and Rules*

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).

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)**

**---- ----**

**626 6**

**25 243**

**<-classified as**

**(a): class no**

**(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:

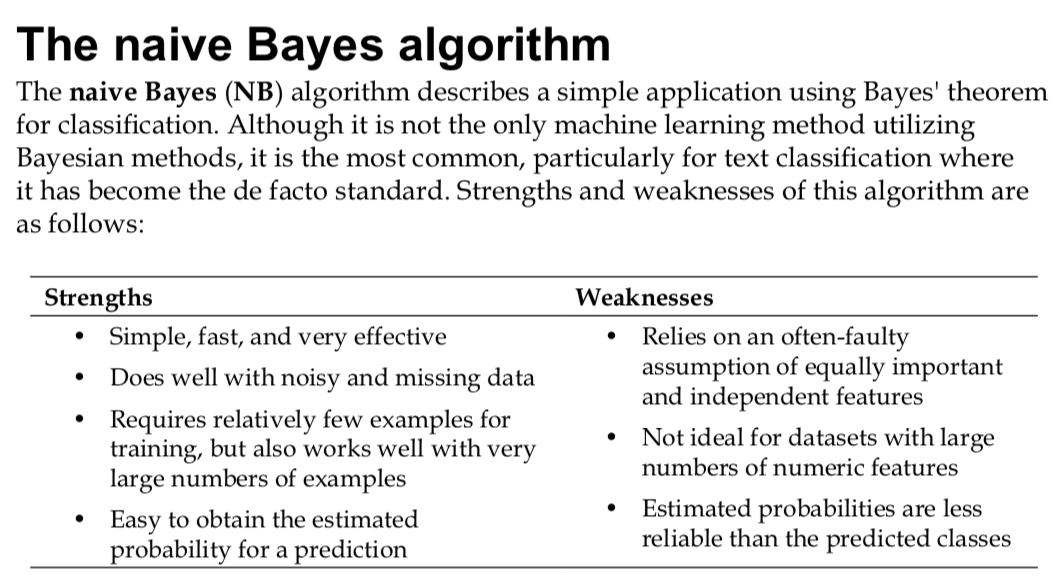
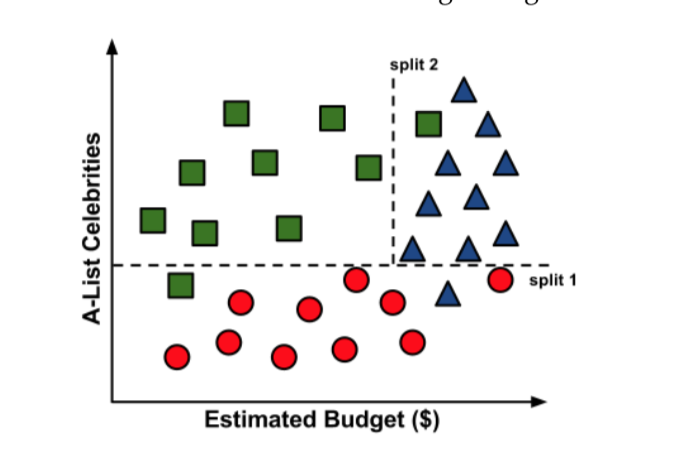
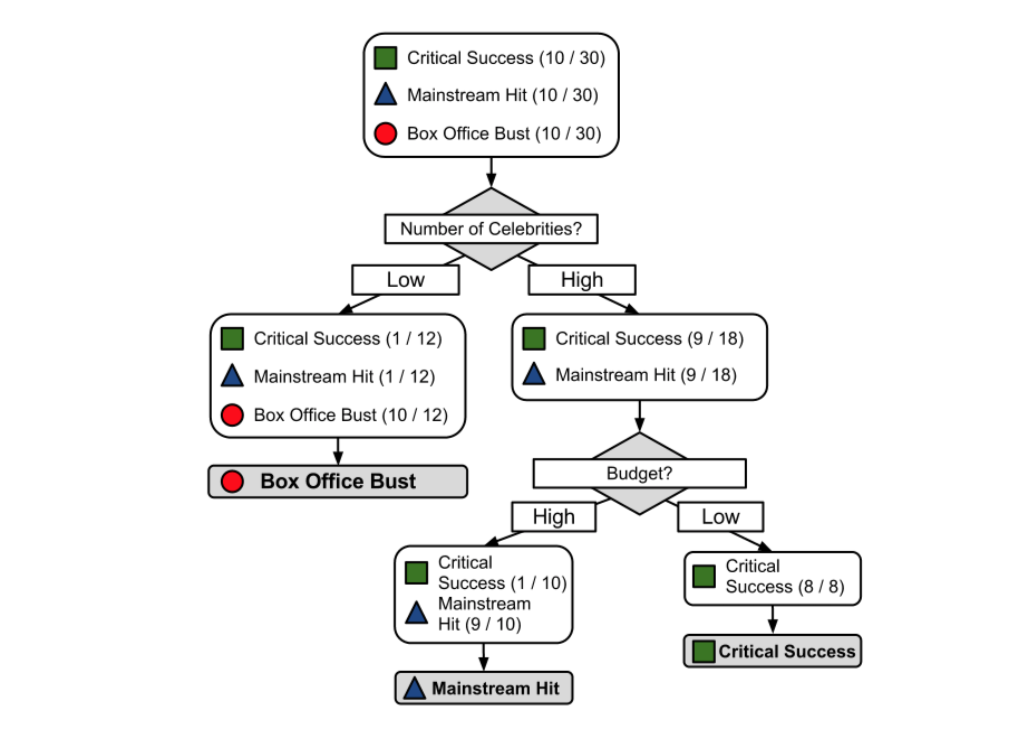
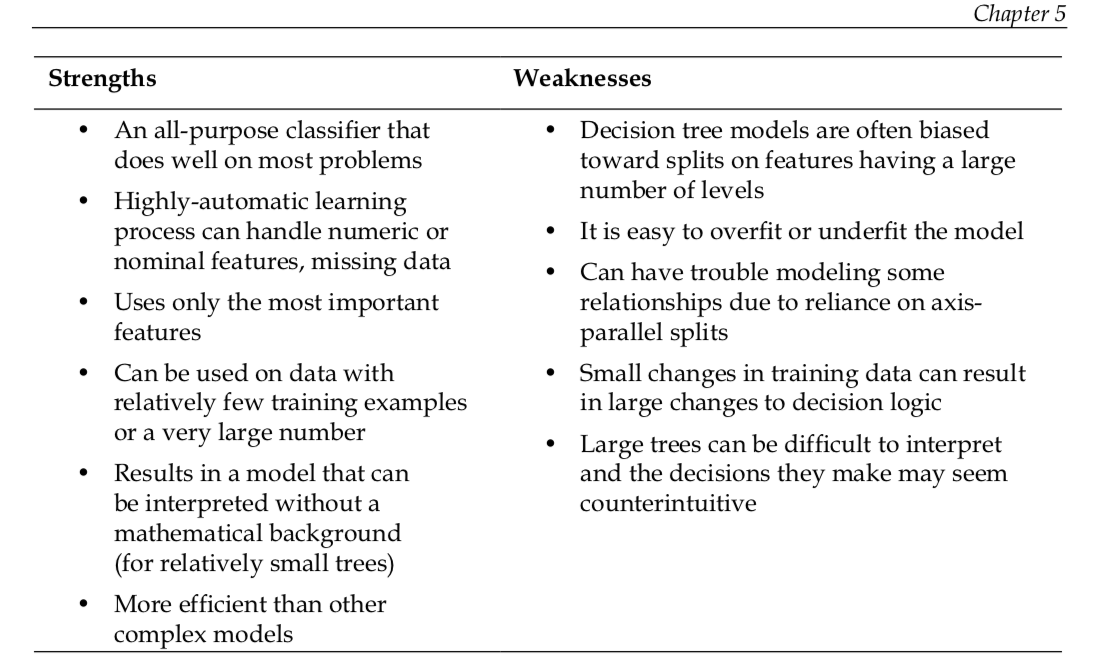
**> credit\_boost\_pred10 <- predict(credit\_boost10, credit\_test)**

**> CrossTable(credit\_test$default, credit\_boost\_pred10,**

**prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,**

**dnn = c('actual default', 'predicted default'))**

**[**



The first step in processing text data involves creating a **corpus**, which refers to a collection of text documents. In our project, a text document refers to a single SMS message. We'll build a corpus containing the SMS messages in the training data using the following command:

**> sms\_corpus <- Corpus(VectorSource(sms\_raw$text))**

This command uses two functions. First, the Corpus() function creates an R object to store text documents. This function takes a parameter specifying the format of  
the text documents to be loaded. Since we have already read the SMS messages and stored them in an R vector, we specify VectorSource(), which tells Corpus() to use the messages in the vector sms\_train$text. The Corpus() function stores the result in an object named sms\_corpus.

To look at the contents of the corpus, we can use the inspect() function.

First, we will convert all of the SMS messages to lowercase and remove any numbers:

**> corpus\_clean <- tm\_map(sms\_corpus, tolower)**

**> corpus\_clean <- tm\_map(corpus\_clean, removeNumbers)**

A common practice when analyzing text data is to remove filler words such as **to**, **and**, **but**, and **or**. These are known as **stop words**. Rather than define a list of stop words ourselves, we will use the stopwords() function provided by the tm package. It contains a set of numerous stop words. To see them all, type stopwords() at

the command line. As we did before, we'll use the tm\_map() function to apply this function to the data:

**> corpus\_clean <- tm\_map(corpus\_clean, removeWords, stopwords())**

We'll also remove punctuation:

**> corpus\_clean <- tm\_map(corpus\_clean, removePunctuation)**

Now that we have removed numbers, stop words, and punctuation, the text messages are left with blank spaces where these characters used to be. The last step then is to remove additional whitespace, leaving only a single space between words.

**> corpus\_clean <- tm\_map(corpus\_clean, stripWhitespace)**

Creating a sparse matrix given a tm corpus involves a single command: **> sms\_dtm <- DocumentTermMatrix(corpus\_clean)**

This will tokenize the corpus and return the sparse matrix with the name sms\_dtm. From here, we'll be able to perform analyses involving word frequency.

A **descriptive model** is used for tasks that would benefit from the insight gained from summarizing data in new and interesting ways. As opposed to predictive models that predict a target of interest; in a descriptive model, no single feature is more important than any other. In fact, because there is no target to learn, the process of training a descriptive model is called **unsupervised learning**. Although it can be more difficult to think of applications for descriptive models—after all, what good is a learner that isn't learning anything in particular—they are used quite regularly for data mining.

A **predictive model** is used for tasks that involve, as the name implies, the prediction of one value using other values in the dataset. The learning algorithm attempts to discover and model the relationship among the **target** feature (the feature being predicted) and the other features. Despite the common use of the word "prediction" to imply forecasting predictive models need not necessarily foresee future events. For instance, a predictive model could be used to predict past events such as the date of a baby's conception using the mother's hormone levels; or, predictive models could be used in real time to control traffic lights during rush hours.

Because predictive models are given clear instruction on what they need to learn and how they are intended to learn it, the process of training a predictive model is known as **supervised learning**. The supervision does not refer to human involvement, but rather the fact that the target values provide a supervisory role, which indicates to the learner the task it needs to learn. Specifically, given a set of data, the learning algorithm attempts to optimize a function (the model) to find the combination of feature values that result in the target output.

The often used supervised machine learning task of predicting which category an example belongs to is known as **classification**. I

1. The target feature to be predicted is a categorical feature known as the **class** and is divided into categories called **levels**.