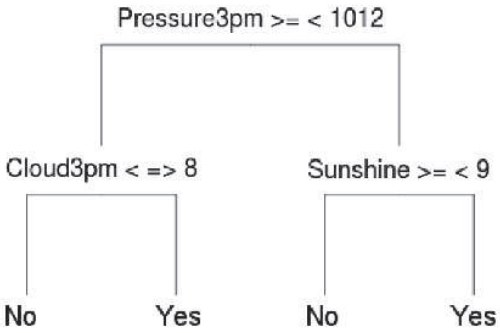
Chapter 11

Decision Trees



Decision trees (also referred to as classification and regression trees) are the traditional building blocks of data mining and the classic machine learning algorithm. Since their development in the 1980s, decision trees have been the most widely deployed machine-learning based data mining model builder. Their attraction lies in the simplicity of the resulting model, where a decision tree (at least one that is not too large) is quite easy to view, understand, and, importantly, explain. Decision trees do not always deliver the best performance, and represent a trade-off between performance and simplicity of explanation. The decision tree structure can represent both classification and regression models.

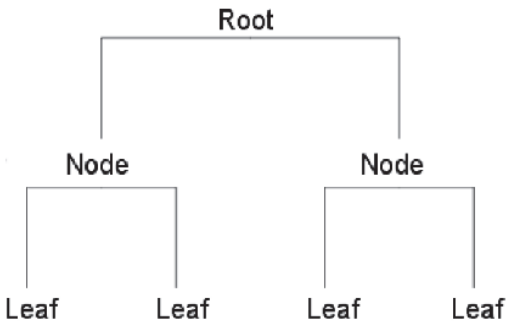
We introduce the decision tree as a knowledge representation language in Section 11.1. A search algorithm for finding a good decision tree is presented in Section 11.2. The measures used to identify a good tree are discussed in Section 11.3. Section 11.4 then illustrates the building of a decision tree in Rattle and directly through R. The options for building a decision tree are covered in Section 11.5.

G. Williams, *Data Mining with Rattle and R: The Art of Excavating Data for Knowledge Discovery*, Use R, DOI 10.1007/978-1-4419-9890-3_11, © Springer Science+Business Media, LLC 2011

11.1 Knowledge Representation

The tree structure is used in many different fields, such as medicine, logic, problem solving, and management science. It is also a traditional computer science structure for organising data. We generally present the tree upside down, with the *root* at the top and the leaves at the bottom. Starting from the root, the tree splits from the single trunk into two or more branches. Each branch itself might further split into two or more branches. This continues until we reach a leaf, which is a node that is not further split. We refer to the split of a branch as a *node* of the tree. The root and leaves are also referred to as *nodes*.

A *decision tree* uses this traditional structure. It starts with a single root node that splits into multiple branches, leading to further nodes, each of which may further split or else terminate as a leaf node. Associated with each nonleaf node will be a test or question that determines which branch to follow. The leaf nodes contain the "decisions."



Consider the decision tree drawn on page 205 (which is the same tree as Figure 2.5 on page 30). This represents knowledge about observing weather conditions one day and the observation of rain on the following day. The `No` and `Yes` values at the leaves of the decision tree represent the decisions.

The root node of the example decision tree tests the mean sea level pressure at 3 pm (`Pressure3pm`). When this variable, for an observation, has a value greater than or equal to 1012 hPa, then we will continue down the left side of the tree. The next test down this left side of the tree is on the amount of cloud cover observed at 3 pm (`Cloud3pm`). If this is less than 8 oktas (i.e., anything but a fully overcast sky), then it is observed that on the following day it generally does not rain (`No`). If we observe that it is overcast today at 3 pm (i.e., `Cloud3pm` is 8 oktas, the maximum value of this variable—see Section 5.2.9, page 127) then generally we observe that it rains the following day (`Yes`). Thus we would be inclined to think that it might rain tomorrow if we observe these same conditions today.

Resuming our interpretation of the model from the root node of the

tree, if `Pressure3pm` is less than 1012 hPa and `Sunshine` is greater than or equal to 9 (i.e., we observe at least 9 hours of sunshine during the day), then we do not expect to observe rain tomorrow. If we record 9 or less hours of sunshine, then we expect it to rain tomorrow.

The decision tree is a very convenient and efficient representation of knowledge. Generally, models expressed in one language can be translated to another language—and so it is with a decision tree. One simple and useful translation is into a rule set. The decision tree above translates to the following rules, where each rule corresponds to one pathway through the decision tree, starting at the root node and terminating at a leaf node:

```
Rule number: 7 [RainTomorrow=Yes cover=27 (11%) prob=0.74]
  Pressure3pm< 1012
  Sunshine< 8.85

Rule number: 5 [RainTomorrow=Yes cover=9 (4%) prob=0.67]
  Pressure3pm>=1012
  Cloud3pm>=7.5

Rule number: 6 [RainTomorrow=No cover=25 (10%) prob=0.20]
  Pressure3pm< 1012
  Sunshine>=8.85

Rule number: 4 [RainTomorrow=No cover=195 (76%) prob=0.05]
  Pressure3pm>=1012
  Cloud3pm< 7.5
```

A rule representation has its advantages. In reviewing the knowledge that has been captured, we can consider each rule separately rather than being distracted by the more complex structure of a large decision tree. It is also easy to see how each rule could be translated into a programming language statement like R, Python, C, VisualBasic, or SQL. The structure is as simple, and clear, as an If-Then statement. We now explain the information provided for each rule.

In building a decision tree, often a larger tree is built and then cut back (or pruned) so that it is not so complex and also to improve its accuracy. As a consequence, we will often see node numbers (and rule numbers) that are not sequential. The node numbers do not have any specific meaning other than as a reference.

Although it is not shown in the tree representation at the beginning of the chapter, we see in the rules above the probabilities that are typically recorded for each leaf node of the decision tree. The probabilities can be used to provide an indication of the strength of the decision we derive from the model. Thus, rule number 7 indicates that for 74% of the observations (`prob=0.74`), when the observed pressure at 3 pm is less than 1012 hPa and the hours of sunshine are less than 8.85 hours, there is rainfall recorded on the following day (`RainTomorrow=Yes`). The other information provided with the rule is that 27 observations from the training dataset (i.e., 11% of the training dataset observations) are covered by this rule—they satisfy the two conditions.

There exist variations to the basic decision tree structure we have presented here for representing knowledge. Some approaches, as here, limit trees to two splits at any one node to generate a *binary decision tree*. For categoric data this might involve partitioning the values (levels) of the variable into two groups. Another approach is to have a branch corresponding to each of the levels of a categoric variable. From a representation point of view, what can be represented using a multiway tree can also be represented as a binary tree and vice versa. Other variations, for example, allow multiple variables to be tested at a node. We generally stay with the simpler representation, though, sometimes at the cost of the resulting model being a little more complex than if we used a more complex decision tree structure.

11.2 Algorithm

Identifying Alternative Models

The decision tree structure, as described above, is the "language" we use to express our knowledge. A sentence (or model) in this language is a particular decision tree. For any dataset, there will be very many, or even infinite, possible decision trees (sentences).

Consider the simple decision tree discussed above. Instead of the variable `Pressure3pm` being tested against the value 1012, it could have been tested against the value 1011, or 1013, or 1020, etc. Each would, when the rest of the tree has been built, represent a different sentence in the language, representing a slightly different capture of the knowledge. There are very many possible values to choose from for just this one

variable, even before we begin to consider values for the other variables that appear in the decision tree.

Alternatively, we might choose to test the value of a different variable at the root node (or any other node). Perhaps we could test the value of `Humidity3pm` instead of `Pressure3pm`. This again introduces a large collection of alternative sentences that we might generate within the constraints of the language we have defined. Each sentence is a candidate for the capture of knowledge that is consistent with the observations represented in our training dataset.

As we saw in Section 8.2, this wealth of possible sentences presents a challenge—which is the best sentence or equivalently which is the best model that fits the data? Our task is to identify the sentence (or perhaps sentences) that best captures the knowledge that can be obtained from the observations that we have available to us.

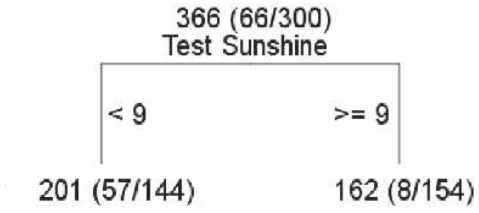
We generally have an infinite collection of possible sentences to choose from. Enumerating every possible sentence, and testing whether it is a good model, will generally be too computationally expensive. This could well involve days, weeks, months, or even more of our computer time. Our task is to use the observations (the training dataset) to narrow down this search task so that we can find a good model in a reasonable amount of time.

Partitioning the Dataset

The algorithm that has been developed for decision tree induction is referred to as the top-down induction of decision trees, using a divide-and-conquer, or recursive partitioning, approach. We will describe the algorithm intuitively.

We continue here with the *weather* dataset to describe the algorithm. The distribution of the observations, with respect to the target variable `RainTomorrow`, is of particular interest. There are 66 observations that have the target as `Yes` (18%) and 300 observations with `No` (82%).

We want to find any input variable that can be used to split the dataset into two smaller datasets. The goal is to increase the homogeneity of each of the two datasets with respect to the target variable. That is, for one of the datasets, we would be looking for it to have an increased proportion of observations with `Yes` and so the other dataset would have an increased proportion of observations with `No`.



We might, for example, decide to construct a partition of the original dataset using the variable `Sunshine` with a split value of 9. Every observation that has a value of `Sunshine` less than 9 goes into one subset and those remaining (with `Sunshine` equal to 9) into a second subset. These new datasets will have 201 and 162 observations, respectively (noting that three observations have missing values for this variable).

Now we consider the proportions of `Yes` and `No` observations within the two new datasets. For the subset of observations with `Sunshine` less than 9, the proportions are 28% `Yes` and 72% `No`. For the subset of observations with `Sunshine` greater than or equal to 9 the proportions are 5% `Yes` and 95% `No`.

By splitting on this variable, we have made an improvement in the homogeneity of the target variable values. In particular, the right dataset (`Sunshine` ≥ 9) results in a collection of observations that are very much in favour of no rain on the following day (95% `No`). This is what we are aiming to do. It allows us to observe that when the amount of sunshine on any day is quite high (i.e., at least 9 hours), then there is very little chance of rain on the following day (only a 5% chance based on our observations from the particular weather station).

The story for the other dataset is not quite so clear. The proportions have certainly changed, with a higher proportion of `Yes` observations than the original dataset, but the `No` observations still outnumber the `Yes` observations. Nonetheless, we can say that when we observe `Sunshine` $<$ 9 there is an increased likelihood of rain the following day based on our historic observations. There is a 28% chance of rain compared with an 18% over all observations.

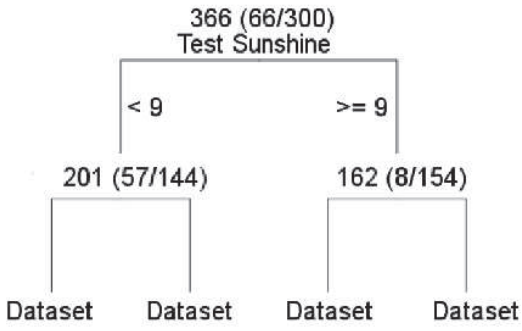
Choosing the value 9 for the variable `Sunshine` is just one possibility from amongst very many choices. If we had chosen the value 5 for the variable `Sunshine` we would have two new datasets with the `Yes`/`No` proportions 41%/59% and 12%/88%. Choosing a different variable altogether (`Cloud3pm`) with a split of 6, we would have two new datasets with the `Yes`/`No` proportions 8%/92% and 34%/66%. Another choice might be `Pressure3pm` with a split of 1012. This gives the `Yes`/`No` proportions as 47%/53% and 10%/90%.

We now have a collection of choices for how we might partition our

training dataset: which of these is the best split? We come back to answer that question formally in Section 11.3. For now, we assume we choose one of them. With whichever choice we make, the result is that we now have two new smaller datasets.

Recursive Partitioning

The process is now repeated again separately for the two new datasets. That is, for the left dataset above (observations having `Sunshine` < 9), we consider all possible variables and splits to partition that dataset into two smaller datasets. Independently, for the right dataset (observations having `Sunshine` ≥ 9) we consider all possible variables and splits to partition that dataset into two smaller datasets as well.



Now we have four even smaller datasets—and the process continues. For each of the four datasets, we again consider all possible variables and splits, choosing the "best" at each stage, partitioning the data, and so on, repeating the process until we decide that we should stop. In general, we might stop when we run out of variables, run out of data, or when partitioning the dataset does not improve the proportions or the outcome.

We can see now why this process is called divide-and-conquer or recursive partitioning. At each step, we have identified a question, that we use to partition the data. The resulting two datasets then correspond to the two branches of the tree emanating from that node. For each branch, we identify a new question and partition appropriately, building our representation of the knowledge we are discovering from the data. We continually divide the dataset and conquer each of the smaller datasets more easily. We are also repeatedly partitioning the dataset and applying the same process, independently, to each of the smaller datasets; thus it is recursive partitioning.

At each stage of the process, we make a decision as to the best variable and split to partition the data. That decision may not be the best to make in the overall context of building this decision tree, but once we make that decision, we stay with it for the rest of the tree. This is generally referred to as a *greedy* approach.

A greedy algorithm is generally quite efficient, whilst possibly sacrificing our opportunity to find the very best decision tree. There remains quite a bit of searching for the one variable and split point for each of the datasets we produce. However, this heuristic approach reduces our search space considerably by fixing the variable/split once it has been chosen.

11.3 Measures

In describing the basic algorithm above, it was indicated that we need to measure how good a particular partition of the dataset is. Such a measure will allow us to choose from amongst a collection of possibilities. We now consider how to measure the different splits of the dataset.

Information Gain

Rattle uses an *information gain* measure for deciding between alternative splits. The concept comes from information theory and uses a formulation of the concept of entropy from physics (i.e., the concept of the amount of disorder in a system). We discuss the concepts here in terms of a binary target variable, but the concept generalises to multiple classes and even to numeric target variables for regression tasks.

For our purposes, the concept of disorder relates to how "mixed" our dataset is with respect to the values of the target variable. If the dataset contains only observations that all have the same value for the target variable (e.g., it contains only observations where it rains the following day), then there is no disorder—i.e., no entropy or zero entropy. If the two values of the target variable are equally distributed across the observations (i.e., 50% of the dataset are observations where it rains tomorrow and the other 50% are observations where it does not rain tomorrow), then the dataset contains the maximum amount of disorder. We identify the maximum amount of entropy as 1. Datasets containing different mixtures of the values of the target variable will have a measure of entropy between 0 and 1.

From an information theory perspective, we interpret a measure of 0 (i.e., an entropy of 0) as indicating that we need no further information in order to classify a specific observation within the dataset—all observations belong to the same class. Conversely, a measure of 1 suggests we need the maximal amount of extra information in order to classify our

observations into one of the two available classes. If the split between the observations where it rains tomorrow and where it does not rain tomorrow is not 50%/50% but perhaps 75%/25%, then we need less extra information in order to classify our observations—the dataset already contains some information about which way the classification is going to go. Like entropy, our measure of "required information" is thus between 0 and 1.

In both cases, we will use the mathematical logarithm function for base 2 ($\log_2$) to transform our proportions (the proportions being 0.5, 0.75, 1.00, etc.). Base 2 is chosen since we use binary digits (bits) to encode information. However, we can use any base since in the end it is the relative measure rather than the exact measure, that we are interested in and the logarithm functions have identical behaviour in this respect. The default R implementation (as we will see in Section 11.4) uses the natural logarithm, for example.

The formula we use to capture the entropy of a dataset, or equivalently the information needed to classify an observation, is

$$info(\mathcal{D}) = -p\log_2(p) - n\log_2(n)$$

We now delve into the nature of this formula to understand why this is a useful measure. We can easily plot this function, as in Figure 11.1, with the x-axis showing the possible values of p and the y-axis showing the values of $info$.

From the plot, we can see that the maximum value of the measure is 1. This occurs when there is the most amount of disorder in the data or when the most amount of additional information is required to classify an observation. This occurs when the observations are equally distributed across the values of the target variable. For a binary target, as here, this occurs when $p = 0.5$ and $n = 0.5$.

Likewise, the minimum value of the measure is 0. This occurs at the extremes, where $p = 1$ (i.e., all observations are positive —`RainTomorrow` has the value `Yes` for each) or $p = 0$ (i.e., all observations are negative —`RainTomorrow` has the value `No` for each). This is interpreted as either no entropy or as requiring no further information in order to classify the observations.

This then provides a mechanism for measuring some aspect of the training dataset, capturing something about the knowledge content. As

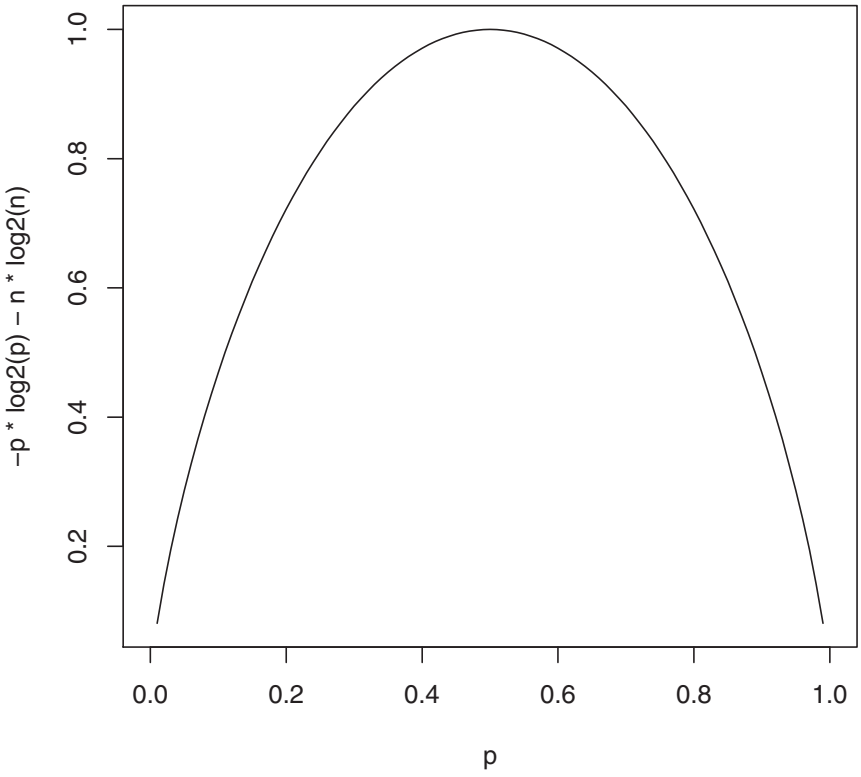


Figure 11.1: Plotting the relationship between the proportion of positive observations in the data and the measure of information/entropy.

we now see, we use this formulation to help choose the "best" split from among the very many possible splits we identified in Section 11.2.

Each choice of a split results in a binary partition of the training dataset. We will call these $\mathcal{D}_1$ and $\mathcal{D}_2$, noting that $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2$. The information measure can be applied to each of these subsets to give $\mathcal{I}_1$ and $\mathcal{I}_2$. If we add these together, weighted by the sizes of the two subsets, we get a measure of the combined information, or entropy:

$$info(\mathcal{D}, \mathcal{S}) = \frac{|\mathcal{D}_1|}{|\mathcal{D}|}\mathcal{I}_1 + \frac{|\mathcal{D}_2|}{|\mathcal{D}|}\mathcal{I}_2$$

Comparing this with the original information, or entropy, we get a measure of the gain in "knowledge" obtained by using the particular split point:

$$gain(\mathcal{D}, \mathcal{S}) = info(\mathcal{D}) - info(\mathcal{D}, \mathcal{S})$$

This can then be calculated for each of the possible splits. The split that provides the greatest gain in information (and equivalently the greatest reduction in entropy) is the split we choose.

Other Measures

A variety of measures can be used as alternatives to the information measure. The most common alternative is the Gini index of diversity. This was introduced into decision tree building through the original CART (classification and regression tree) algorithm (Breiman et al., 1984). The plot of the function is very similar to the $p * \log_2(p)$ curve and typically will give the same split points.

11.4 Tutorial Example

The *weather* dataset is used to illustrate the building of a decision tree. We saw our first decision tree in Chapter 2. We can build a decision tree using Rattle's Tree option, found on the Model tab or directly in R through `rpart()` of **rpart** (Therneau and Atkinson, 2011).

Building a Model Using Rattle

We build a decision tree using Rattle's Model tab's Tree option. After loading our dataset and identifying the Input variables and the Target variable, an Execute of the Model tab will result in a decision tree. We can see the result for the *weather* dataset in Figure 11.2, which shows the resulting tree in the text view and also highlights the key interface widgets that we need to deal with to build a tree.

The text view includes much information, and we will work our way through its contents. However, before doing so, we can get a quick view of the resulting decision tree by using the Draw button of the interface. A window will pop up, displaying the tree, as we saw in Figure 2.5 on page 30.

Working our way through the textual summary of the decision tree, we start with a report of the number of observations that were used to build the tree (i.e., 256):

```
Summary of the Decision Tree model for Classification ...

n= 256
```

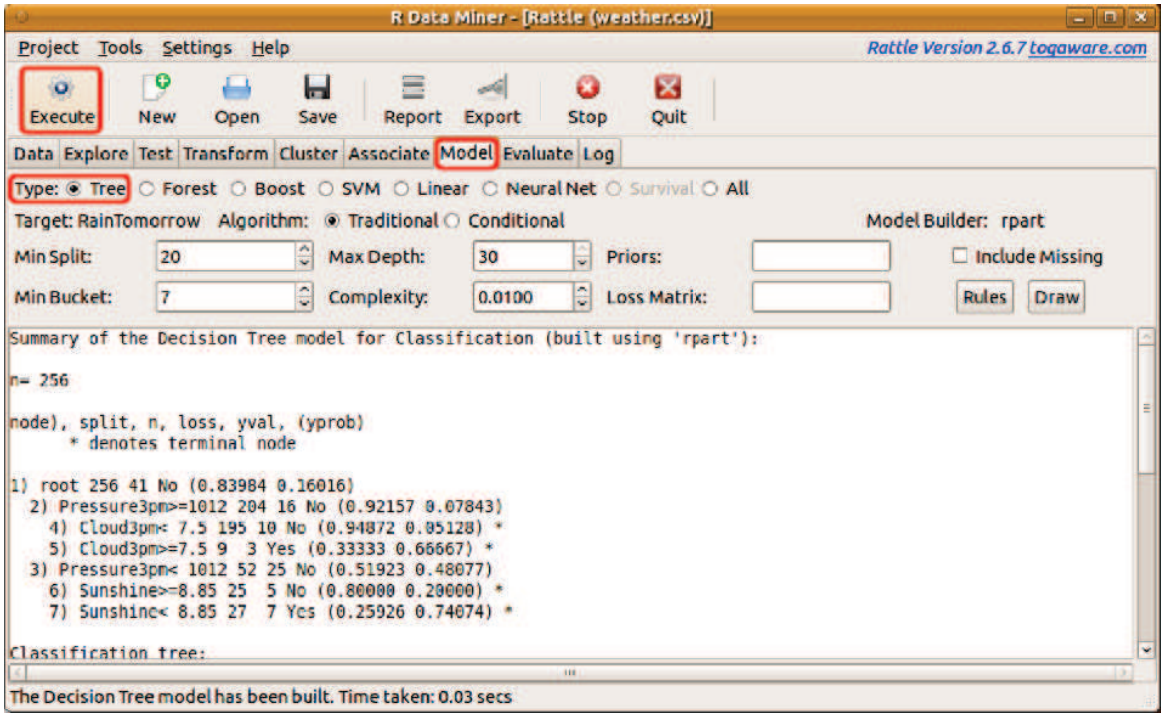


Figure 11.2: Building a decision tree predictive model using the *weather* dataset.

Tree Structure

We now look at the structure of the tree as it is presented in the text view. A legend is provided to assist in reading the tree structure:

```
node), split, n, loss, yval, (yprob)
      * denotes terminal node
```

The legend indicates that a `node` number will be provided, followed by a `split` (which will usually be in the form of a *variable operation value*), the number of entities `n` at that node, the number of entities that are incorrectly classified (the `loss`), the default classification for the node (the `yval`), and then the distribution of classes in that node (the `yprobs`). The distribution is ordered by class and the order is the same for all nodes. The next line indicates that a "*" denotes a terminal node of the tree (i.e., a leaf node—the tree is not split any further at that node).

The first node of any tree is always the root node. We work our way into the tree itself through the root node. The root node is numbered as node number 1:

```
1) root 256 41 No (0.83984 0.16016)
```

The root node represents all observations. By itself the node represents

a model that simply classifies every observation into the class that is associated with the majority from the training dataset. The information provided tells us that the majority class for the root node (the `yval`) is `No`. The 41 tells us how many of the 256 observations will be incorrectly classified as `Yes`. This is technically called the `loss`.

The `yprob` component then reports on the distribution of the classes across the observations. We know the classes to be No, and Yes. Thus, 84% (i.e., 0.83984375 as a proportion) of the observations have the target variable `RainTomorrow` as No, and 16% of the observations have it as Yes.

If the root node itself were treated as a model, it would always decide that it won't rain tomorrow. Based on the training dataset, the model would be 84% correct. That is quite a good level of accuracy, but the model is not particularly useful since we are really interested in whether it is going to rain tomorrow.

The root node is split into two subnodes. The split is based on the variable `Pressure3pm` with a split value of `1011.9`. Node 2 has the *split* expressed as `Pressure3pm>=1011.9`. That is, there are 204 observations with a 3 pm pressure reading of more than 1011.9 hPa:.

```
2) Pressure3pm>=1012 204 16 No (0.92157 0.07843)
```

Only 16 of these 204 observations are misclassified, with the classification associated with this node being No. This represents an accuracy of 92% in predicting that it does not rain tomorrow.

Node 3 contains the remaining 52 observations which have a 3 pm pressure of less than `1011.9`. Whilst the decision is No, it is pretty close to a 50/50 split in this partition:

```
3) Pressure3pm< 1012 52 25 No (0.51923 0.48077)
```

We've skipped ahead a little to jump to node 3, so we now have a look again at node 2 and its split into subnodes. The algorithm has chosen `Cloud3pm` for the next split, with a split value of `7.5`. Node 4 has 195 observations. These are the 195 observations for which the 3 pm pressure is greater than or equal to 1011.9 and the cloud coverage at 3 pm is less than 7.5. Under these circumstances, there is no rain the following day 95% of the time.

```
2) Pressure3pm>=1012 204 16 No (0.92157 0.07843)
  4) Cloud3pm< 7.5 195 10 No (0.94872 0.05128) *
  5) Cloud3pm>=7.5 9  3 Yes (0.33333 0.66667) *
```

Node 5, at last, predicts that it will rain on the following day—at least based on the available historic observations. There are only nine observations here, and the frequency of observing rain on the following day is 67%. Thus we say there is a 67% probability of rain when the pressure at 3 pm is at least 1011.9 and the cloud cover at 3 pm is at least 7.5. Both node 4 and node 5 are marked with an asterisk (*), indicating that they are terminal nodes—they are not further split. The remaining nodes, 6 and 7, split node 3 using the variable `Sunshine` and a split point of `8.85`:

```
  3) Pressure3pm< 1012 52 25 No (0.51923 0.48077)
    6) Sunshine>=8.85 25  5 No (0.80000 0.20000) *
    7) Sunshine< 8.85 27  7 Yes (0.25926 0.74074) *
```

Node 3 has almost equal numbers of No and Yes observations (52% and 48%, respectively). However, splitting on the number of hours of sunshine has quite nicely partitioned the observations into two groups that are quite a bit more homogeneous with respect to the target variable. Node 6 represents only a 20% chance of rain tomorrow, whilst node 7 represents a 74% chance of rain tomorrow.

That then is the model that has been built. It is a relatively simple decision tree with just seven nodes and four leaf nodes, with a maximum depth of 2 (in fact, each leaf node is at a depth of exactly 2).

Function Call

The next segment lists the underlying R command line that is used to build the decision tree. This was automatically generated based on the information provided through the interface. We could have directly entered this at the prompt in the R Console:

```
Classification tree:
rpart(formula=RainTomorrow ~ .,data=crs$dataset[crs$train,
 c(crs$input,crs$target)],method="class",
 parms=list(split="information"),
 control=rpart.control(usesurrogate=0,maxsurrogate=0))
```

The *formula* notes that we want to build a model to predict the value of the variable `RainTomorrow` based on the remainder of the variables in the dataset supplied (notated as the "~ ."). The dataset supplied consists of the `crs$dataset` data frame indexed to include the rows listed in the

variable `crs$train`. This is the training dataset. The columns from 3 to 22, and then column 24, are included in the dataset from which the model is built.

Following the specification of the formula and dataset are the tuning parameters for the algorithm. These are explained in detail in Section 11.5, but we briefly summarise them here. The method used is based on classification. The method for choosing the best split uses the information measure. Surrogates (for dealing with missing values) are not used by default in Rattle.

Variables Used

In general, only a subset of the available variables will be used in the resulting decision tree model. The next segment lists those variables that do appear in the tree. Of the 20 input variables, only three are used in the final model.

```
Variables actually used in tree construction:
[1] Cloud3pm     Pressure3pm Sunshine
```

Performance Evaluation

The next segment summarises the process of building the tree, and in particular the iterations and associated change in the accuracy of the model as new levels are added to the tree. The complexity table is discussed in more detail in Section 11.5.

Briefly, though, we are most likely interested in the cross-validated error (refer to Section 15.1 for a discussion of cross-validation), which is the `xerror` column of the table. The error over the whole dataset (i.e., if we were to classify every observation as `No`) is 0.16, or 16%. Treating this as the baseline error (i.e., 1.00), the table shows the relative reduction in the error (and cross-validation-based error) as we build the tree.

From line 2, we see that after the first split of the dataset, we have reduced the cross-validation based error to 80% of the original amount (i.e., 0.1289, or 13%). Notice that the cross-validation is being reduced more slowly than the error on the training dataset (`error`). This is typical.

The CP value (the complexity parameter) is explained further in Section 11.5, but for now we note that as the tree splits into more nodes,

the complexity parameter is reduced. But we also note that the cross-validation error starts to increase as we further split the decision tree. This tells the algorithm to stop partitioning, as the error rate (at least the unbiased estimate of it—refer to Section 15.1) is not improving:

```
Root node error: 41/256 = 0.16

n= 256

     CP nsplit rel error xerror xstd
1 0.159      0      1.00   1.00 0.14
2 0.073      2      0.68   0.80 0.13
3 0.010      3      0.61   0.95 0.14
```

Time Taken

Finally, we see how long it took to build the tree. Decision trees are generally very quick to build.

```
Time taken: 0.03 secs
```

Tuning Options

The Rattle interface provides a choice of Algorithm for building the decision tree. The Traditional option is chosen by default, and that is what we have presented here. The Conditional option uses a more recent conditional inference tree algorithm, which is explained in more detail in Section 11.6. A variety of other tuning options are also provided, and they are discussed in some detail in Section 11.5.

Displaying Trees

The Rules and Draw buttons provide alternative views of the decision tree. Clicking on the Rules button will translate the decision tree into a set of rules and list those rules at the bottom of the text view. We need to scroll down the text view in order to see the rules. The rules in this form can be more easily extracted and used to generate code in other languages. A common example is to generate a query in SQL to extract the corresponding observations from a database.

The Draw button will pop up a separate window to display a more visually appealing representation of the decision tree. We have seen the pictorial representation of a decision tree a number of times now, and they were generated from this button, as was Figure 11.3.

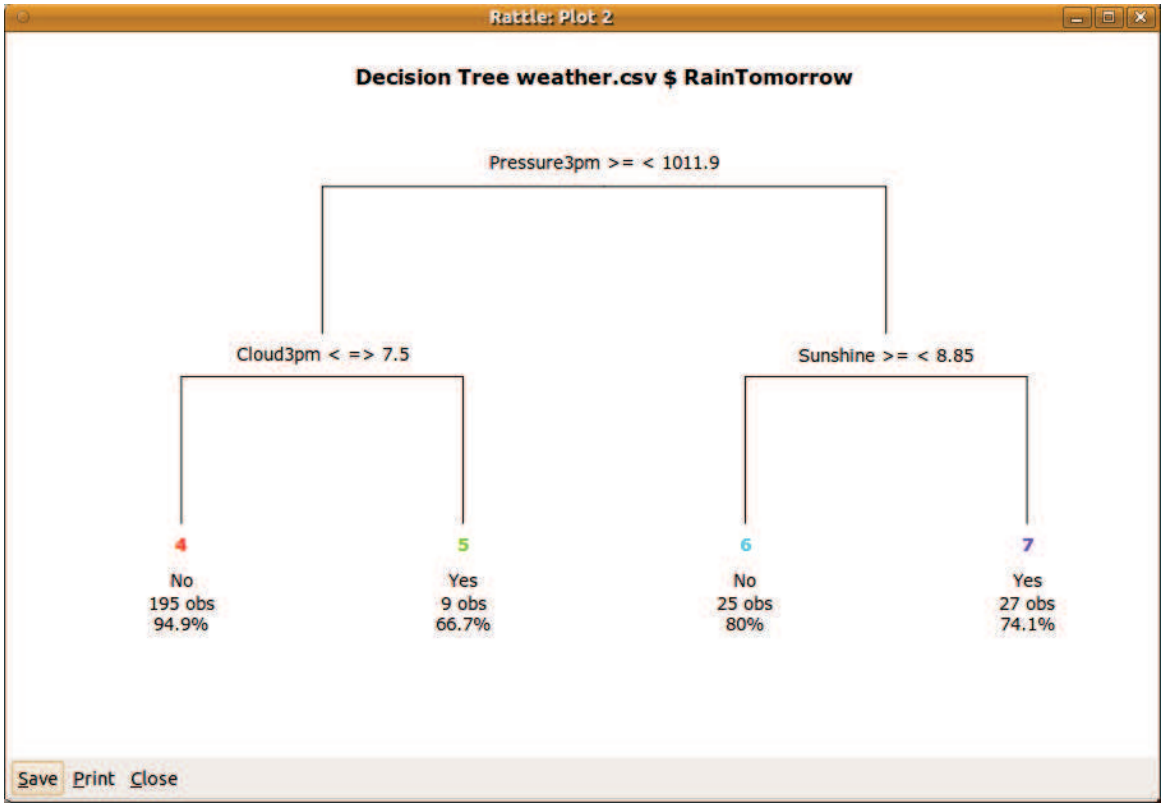


Figure 11.3: Typical Rattle decision tree.

Scoring

We can now use the model to predict the outcome for new observations—something we often call scoring. The Evaluate tab provides the Score option and the choice to Enter some data manually and have that data scored by the model. Executing this setup will result in a popup window in which to enter the data, and, on closing the window, the data is passed on to the model and the predictions are displayed in the Textview.

Building a Model using R

Underneath Rattle's GUI, we are relying on a collection of R commands and functions. The Log tab will expose them, and it is instructive to review the Log tab regularly to gain insight and understanding that will

be helpful in using R itself. We effectively lift the bonnet on the hood here so that we can directly build decision trees using R.

To use the traditional decision-tree-building algorithm, we use **rpart**, This provides `rpart()` which is an implementation of the standard classification and regression tree algorithms. The implementation is very robust and reliable.

```
> library(rpart)
```

As we saw in Section 2.9, we will create the variable `weatherDS` (using `new.env()`—new environment) to act as a container for the *weather* dataset and related information. We will access data within this container through the use of `evalq()` below.

```
> weatherDS <- new.env()
```

The *weather* dataset from **rattle** will be used for the modelling. Three columns from the dataset are ignored in our analyses, as they play no role in the model building. The three variables are the two that serve to identify the observations (`Date` and `Location`) and the risk variable (`RISK_MM`—the amount of rain recorded on the next day). Below we identify the index of these variables and record the negative index in `vars`, which is stored within the container:

```
> library(rattle)
> evalq({
    data <- weather
    nobs <- nrow(data)
    vars <- -grep('^(Date|Locat|RISK)', names(weather))
  }, weatherDS)
```

A random subset of 70% of the observations is chosen and will be used to identify a training dataset. The random number seed is set, using `set.seed()`, so that we will always obtain the same random sample for illustrative purposes and repeatability. Choosing different random sample seeds is also useful, providing empirically an indication of how stable the models are.

```
> evalq({
    set.seed(42)
    train <- sample(nobs, 0.7*nobs)
  }, weatherDS)
```

We add to the weatherDS container the formula to describe the model that is to be built based on this dataset:

```
> evalq({
    form <- formula(RainTomorrow ~ .)
  }, weatherDS)
```

We now create a model container for the information relevant to the decision tree model that we will build. The container includes the weatherDS container (identifying it as parent= in the call to new.env()):

```
> weatherRPART <- new.env(parent=weatherDS)
```

The command to build a model is then straight forward. The variables data, train, and vars are obtained from the weatherDS container, and the result will be stored as the variable model within the weatherRPART container. We explain rpart() in detail below.

```
> evalq({
    model <- rpart(formula=form, data=data[train, vars])
  }, weatherRPART)
```

Here we use rpart(), passing to it a formula and the data. We don't need to include the formula= and the data= in the formal arguments to the function, as they will also be determined from their position in the argument list. It doesn't hurt to include them either to provide more clarity for others reading the code.

The formula= argument identifies the model that is to be built. In this case, we pass to the function the variable form that we previously defined. The target variable (to the left of the tilde in form) is RainTomorrow, and the input variables consist of all of the remaining variables in the dataset (denoted by the period to the right of the tilde in form). We are requesting a model that predicts a value for RainTomorrow based on today's observations.

The data= argument identifies the training dataset. Once again, we pass to the function the variable data that we previously defined. The training dataset subset consists of the observation numbers listed in the variable train. The variables of the dataset that we wish to include are specified by vars, which in this case actually lists as negative integers the variables to ignore. Together, train and vars identify the observations and variables to include in the training of the model.

The result of building the model is assigned into the variable `model` inside the environment `weatherRPART` and so can be independently referred to as `weatherRPART$model`.

Exploring the Model

Towards the end of Section 11.4, we explained the textual presentation of the results of building a decision tree model. The output we saw there can be reproduced in R using `print()` and `printcp()`. The output from `print()` is:

```
> print(weatherRPART$model)

n= 256

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 256 41 No (0.83984 0.16016)
  2) Pressure3pm>=1012 204 16 No (0.92157 0.07843)
    4) Cloud3pm< 7.5 195 10 No (0.94872 0.05128) *
    5) Cloud3pm>=7.5 9  3 Yes (0.33333 0.66667) *
  3) Pressure3pm< 1012 52 25 No (0.51923 0.48077)
    6) Sunshine>=8.85 25  5 No (0.80000 0.20000) *
    7) Sunshine< 8.85 27  7 Yes (0.25926 0.74074) *
```

We briefly discussed the output of this and the `printcp()` below in the previous section. We mentioned there how the CP (complexity parameter) is used to guide how large a decision tree to build. We might choose to stop when the cross-validated error (`xerror`) begins to increase. This is displayed in the output of `printcp()`. We can also obtain a useful graphical representation of the complexity parameter using `plotcp()` instead.

```
> printcp(weatherRPART$model)

Classification tree:
rpart(formula = form, data = data[train, vars])

Variables actually used in tree construction:
[1] Cloud3pm    Pressure3pm Sunshine

Root node error: 41/256 = 0.16

n= 256

     CP nsplit rel error xerror xstd
1 0.159      0      1.00   1.00 0.14
2 0.073      2      0.68   0.83 0.13
3 0.010      3      0.61   0.80 0.13
```

Another command useful for providing information about the resulting model is `summary()`:

```
> summary(weatherRPART$model)
```

This command provides quite a bit more information about the model building process, beginning with the function call and data size. This is followed by the same complexity table we saw above:

```
Call:
rpart(formula=form, data=data[train, vars])
  n= 256

       CP nsplit rel error xerror   xstd
1 0.15854      0    1.0000 1.0000 0.1431
2 0.07317      2    0.6829 0.8293 0.1324
3 0.01000      3    0.6098 0.8049 0.1308
```

The summary goes on to provide information related to each node of the decision. Node number 1 is the root node of the decision tree. Its information appears first (note that the text here is modified to fit the page):

```
Node number 1: 256 observations, complexity param=0.1585
  predicted class=No   expected loss=0.1602
    class counts:   215    41
   probabilities: 0.840 0.160
  left son=2 (204 obs) right son=3 (52 obs)
  Primary splits:
   Pressure3pm   < 1012  right, improve=13.420, (0 missing)
   Cloud3pm      < 7.5   left,  improve= 9.492, (0 missing)
   Pressure9am   < 1016  right, improve= 9.143, (0 missing)
   Sunshine      < 6.45  right, improve= 8.990, (2 missing)
   WindGustSpeed < 64    left,  improve= 7.339, (2 missing)
  Surrogate splits:
   Pressure9am   < 1013  right, agree=0.938, adj=0.692,...
   MinTemp       < 16.15 left,  agree=0.824, adj=0.135,...
   Temp9am       < 20.35 left,  agree=0.816, adj=0.096,...
   WindGustSpeed < 64    left,  agree=0.812, adj=0.077,...
   WindSpeed3pm  < 34    left,  agree=0.812, adj=0.077,...
```

We see that node number 1 has 256 observations to work with. It has a complexity parameter of 0.1585366, which is discussed in Section 11.5.

The next line identifies the default class for this node (`No` in this case) which corresponds to the class that occurs most frequently in the training dataset. With this class as the decision associated with this node, the error rate (or expected loss) is 16% (or 0.1601562).

The table that follows then reports the frequency of observations by the target variable. There are 215 observations with `No` for `RainTomorrow` (84%) and 41 with `Yes` (16%).

The remainder of the information relates to deciding how to split the node into two subsets. The resulting split has a left branch (labelled as node number 2) with 204 observations. The right branch (labelled as node number 3) has 52 observations.

The actual variable used to split the dataset into these two subsets is `Pressure3pm`, with the test being on the value `1011.9`. Any observation with `Pressure3pm` < 1011.9 goes to the right branch, and so ≥ 1011.9 goes to the left. The measure (the improvement) associated with this split of the dataset is 13.42.

We then see a collection of alternative splits and their associated measures. Clearly, `Pressure3pm` offers the best improvement, with the nearest competitor offering an improvement of 9.49.

The surrogate splits that are then presented relate to the handling of missing values in the data. Consider the situation where we apply the model to new data but have an observation with `Pressure3pm` missing. We could instead use `Pressure9am`. The information here indicates that 93.8% of the observations in the split based on $Pressure9am < 1013.3$ are the same as that based on $Pressure3pm < 1011.9$. The `adj` value is an indication of what is gained by using this surrogate split over simply giving up at this node and assigning the majority decision to the new observation. Thus, in using `Pressure9am` we gain a 69% improvement by using the surrogate.

The other nodes are then listed in the summary. They include the same kind of information, and we see at the beginning of node number 2 here:

```
Node number 2: 204 observations, complexity param=0.07317
  predicted class=No   expected loss=0.07843
    class counts:   188    16
   probabilities: 0.922 0.078
  left son=4 (195 obs) right son=5 (9 obs)
  Primary splits:
   Cloud3pm    < 7.5   left,  improve=6.516, (0 missing)
   Sunshine    < 6.4   right, improve=2.937, (2 missing)
   Cloud9am    < 7.5   left,  improve=2.795, (0 missing)
   Humidity3pm < 71    left,  improve=1.465, (0 missing)
   WindDir9am  splits as  RRRRR...LLLL, improve=1.391,...

...
```

Note how categoric variables are reported. `WindDir9am` has 16 levels:

```
> levels(weather$WindDir9am)

 [1] "N"   "NNE" "NE"  "ENE" "E"   "ESE" "SE"  "SSE" "S"
[10] "SSW" "SW"  "WSW" "W"   "WNW" "NW"  "NNW"
```

All possible binary combinations of levels will have been considered and the one reported above offers the best improvement. Here the first five levels (`N` to `E`) correspond to the right (`R`) branch and the remainder to the left (`L`) branch.

The leaf nodes of the decision tree (nodes 4, 5, 6, and 7) will have just the relevant information—thus no information on splits or surrogates. An

example is node 7. The following text again comes from the output of `summary()`:

```
...
Node number 7: 27 observations
  predicted class=Yes  expected loss=0.2593
    class counts:     7    20
   probabilities: 0.259 0.741
```

Node 7 is a leaf node that predicts `Yes` as the outcome. The error/loss is 7 out of 27, or 0.2593 or 26%, and the probability of `Yes` is 74%.

Miscellaneous Functions

We have covered above the main functions and commands in R for building and displaying a decision tree. **Rpart** and **rattle** also provide a collection of utility functions for exploring the model.

First, the `where=` component of the decision tree object records the leaf node of the decision tree in which each observation in the training dataset ends up:

```
> head(weatherRPART$model$where, 12)

335 343 105 302 233 188 266  49 236 252 163 256
  3   3   7   3   3   3   3   4   3   3   3   3
```

The `plot()` command and the related `text()` command will display a decision tree labelled appropriately:

```
> opar <- par(xpd=TRUE)
> plot(weatherRPART$model)
> text(weatherRPART$model)
> par(opar)
```

We notice that the default plot (Figure 11.4) looks different from the plot we obtain through Rattle. Rattle provides `drawTreeNodes()` as a variation of `plot()` based on `draw.tree()` from **maptree** (White, 2010). The plot here is a basic plot. The length of each line within the tree branches gives a visual indication of the error down that branch of the tree. The plot and text can be further tuned through addition arguments to the two commands. There are very many tuning options available,

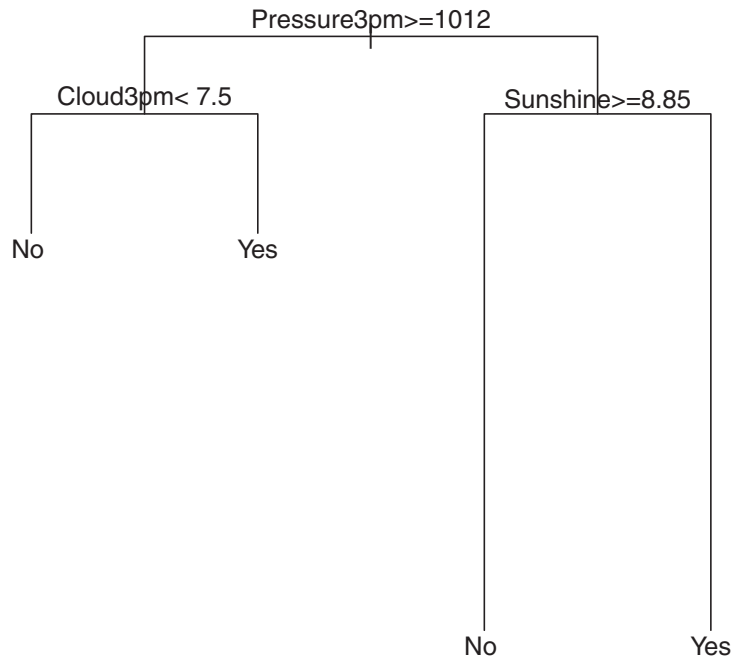


Figure 11.4: Typical R decision tree.

and they are listed in the manuals for the commands (`?plot.rpart` and `?text.rpart`). The `path.rpart()` command is then a useful adjunct to `plot()`:

```
> path.rpart(weatherRPART$model)
```

Running this command allows us to use the left mouse button to click on a node on the plot to list the path to that node. For example, clicking the left mouse button on the bottom right node results in:

```
node number: 7
  root
  Pressure3pm< 1012
  Sunshine< 8.85
```

Click on the middle or right mouse button to finish interacting with the plot.

11.5 Tuning Parameters

Any implementation of the decision tree algorithm provides a collection of parameters for tuning how the tree is built. The defaults in Rattle (based on **rpart**'s defaults) often provide a basically good tree. They are certainly a very good starting point and may be a satisfactory end point, too. However, tuning will be necessary where, for example, the target variable has very few examples of the particular class of interest or we would like to explore a number of alternative trees.

Whilst many tuning parameters are introduced here in some level of detail, the R documentation provides much more information. Use `?rpart` to start exploring further. The `rpart()` function has two arguments for tuning the algorithm, each being a structure containing other options. They are `control=` and `parms=`. We use these as in the following example:

```
> evalq({
    control <- rpart.control(minsplit=10,
                             minbucket=5,
                             maxdepth=20,
                             usesurrogate=0,
                             maxsurrogate=0)
    model <- rpart(formula=form,
                   data=data[train, vars],
                   method="class",
                   parms=list(split="information"),
                   control=control)
  }, weatherRPART)
```

We have already discussed the `formula=` and `data=` arguments. The remaining arguments are now discussed.

Modelling Method (`method=`)

The `method=` argument indicates the type of model to be built and is dependent on the target variable. For categoric targets, we generally build classification models, and so we use `method="class"`. If the target is a numeric variable, then the argument would be `method="anova"` for an "analysis of variance," building a regression tree.

Splitting Function (split=)

The split= argument is used to choose between different splitting functions (measures). The argument appears within the parms argument of rpart(), which is built up as a named list. The split="information" directs rpart to use the information gain measure we introduced above. The default choice of split="gini" (in R, though Rattle's default is "information") uses the Gini index of diversity. The choice makes no difference in this case, as we can verify by reviewing the output of the following two commands (though here we show just the one set of output):

```
> evalq({
    rpart(formula=form,
          data=data[train, vars],
          parms=list(split="information"))
  }, weatherRPART)
n= 256

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 256 41 No (0.83984 0.16016)
  2) Pressure3pm>=1012 204 16 No (0.92157 0.07843)
    4) Cloud3pm< 7.5 195 10 No (0.94872 0.05128) *
    5) Cloud3pm>=7.5 9  3 Yes (0.33333 0.66667) *
  3) Pressure3pm< 1012 52 25 No (0.51923 0.48077)
    6) Sunshine>=8.85 25  5 No (0.80000 0.20000) *
    7) Sunshine< 8.85 27  7 Yes (0.25926 0.74074) *
```

```
> evalq({
    rpart(formula=form,
          data=data[train, vars],
          parms=list(split="gini"))
  }, weatherRPART)
```

Minimum Split (`minsplit=`)

The `minsplit=` argument specifies the minimum number of observations that must exist at a node in the tree before it is considered for splitting. A node is not considered for splitting if it has fewer than `minsplit` observations. The `minsplit=` argument appears within the `control=` argument of `rpart()`. The default value of `minsplit=` is 20.

In the following example, we illustrate the boundary between splitting and not splitting the root node of our decision tree. This is often an issue in building a decision tree, and an inconvenience when all we obtain is a root node. Here the example shows that with a `minsplit=` of 53 the tree building will not proceed past the root node:

```
> evalq({
    rpart(formula=form,
          data=data[train, vars],
          control=rpart.control(minsplit=53))
  }, weatherRPART)
n= 256

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 256 41 No (0.8398 0.1602) *
```

Setting `minsplit=` to 52 results in a split on `Pressure3pm` (and further splitting) being considered and chosen, as we see in the code block below. Splitting on `Pressure3pm` splits the dataset into two datasets, one with 204 observations and the other with 52 observations. We can then see why, with `minsplit=` set to of 53, the tree building does not proceed past the root node.

Changing the value of `minsplit=` allows us to eliminate some computation, as nodes with a small number of observations will generally play less of a role in our models. Leaf nodes can still be constructed that have fewer observations than the `minsplit=`, as that is controlled by the `minbucket=` argument.

```
> evalq({
    rpart(formula=form,
          data=data[train, vars],
          control=rpart.control(minsplit=52))
  }, weatherRPART)
n= 256

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 256 41 No (0.83984 0.16016)
  2) Pressure3pm>=1012 204 16 No (0.92157 0.07843) *
  3) Pressure3pm< 1012 52 25 No (0.51923 0.48077)
    6) Sunshine>=8.85 25  5 No (0.80000 0.20000) *
    7) Sunshine< 8.85 27  7 Yes (0.25926 0.74074) *
```

Minimum Bucket Size (`minbucket=`)

The `minbucket=` argument is the minimum number of observations in any leaf node. The default value is 7, or about one-third of the default value of `minsplit=`. If either of these two arguments is specified but not the other, then the default of the unspecified one is taken to be a value such that this relationship holds (i.e., `minbucket=` is one-third of `minsplit=`)

Once again we will see two examples of using `minbucket=`. The first example limits the minimum bucket size to be 10, resulting in the same model we obtained above. The second example reduces the limit down to just 5 observations in the bucket. The result will generally be a larger decision tree, since we are allowing leaf nodes with a smaller number of observations to be considered, and hence the option to split a node into smaller nodes will often be exercised by the tree building algorithm.

```
> ops <- options(digits=2)
> evalq({
    rpart(formula=form,
          data=data[train, vars],
          control=rpart.control(minbucket=10))
  }, weatherRPART)
n= 256

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 256 41 No (0.840 0.160)
  2) Pressure3pm>=1e+03 204 16 No (0.922 0.078) *
  3) Pressure3pm< 1e+03 52 25 No (0.519 0.481)
    6) Sunshine>=8.9 25  5 No (0.800 0.200) *
    7) Sunshine< 8.9 27  7 Yes (0.259 0.741) *
> evalq({
    rpart(formula=form,
          data=data[train, vars],
          control=rpart.control(minbucket=5))
  }, weatherRPART)
n= 256

node), split, n, loss, yval, (yprob)
      * denotes terminal node

 1) root 256 41 No (0.840 0.160)
   2) Pressure3pm>=1e+03 204 16 No (0.922 0.078)
     4) Cloud3pm< 7.5 195 10 No (0.949 0.051) *
     5) Cloud3pm>=7.5 9  3 Yes (0.333 0.667) *
   3) Pressure3pm< 1e+03 52 25 No (0.519 0.481)
     6) Sunshine>=8.9 25  5 No (0.800 0.200) *
     7) Sunshine< 8.9 27  7 Yes (0.259 0.741)
      14) Evaporation< 5.5 15  7 Yes (0.467 0.533)
        28) WindGustSpeed< 58 10  3 No (0.700 0.300) *
        29) WindGustSpeed>=58 5  0 Yes (0.000 1.000) *
      15) Evaporation>=5.5 12  0 Yes (0.000 1.000) *
> options(ops)
```

Note that changing the value of `minbucket=` can have an impact on the choice of variable for the split. This will occur when one choice with a higher improvement results in a node with too few observations, leading to another choice being taken to meet the minimum requirements for the number of observations in a split.

Whilst the default is to set `minbucket=` to be one-third of `minsplit=`, there is no requirement for `minbucket=` to be less than `minsplit=`. A node will always have at least `minbucket=` entities, and it will be considered for splitting if it has at least `minsplit=` observations and if on splitting each of its children has at least `minbucket=` observations.

Complexity Parameter (`cp=`)

The complexity parameter is used to control the size of the decision tree and to select an optimal tree size. The complexity parameter controls the process of pruning a decision tree. As we will discuss in Chapter 15, without pruning, a decision tree model can overfit the training data and then not perform very well on new data. In general, the more complex a model, the more likely it is to match the data on which it has been trained and the less likely it is to match new, previously unseen data.

On the other hand, decision tree models are very interpretable, and thus building a more complex tree (i.e., having many branches) is sometimes tempting (and useful). It can provide insights that we can then test statistically.

Using `cp=` governs the minimum "benefit" that must be gained at each split of the decision tree in order to make a split worthwhile. This therefore saves on computing time by eliminating splits that appear to add little value to the model. The default is 0.01. A value of 0 will build a "complete" decision tree to maximum depth depending on the values of `minplit=` and `minbucket=`. This is useful if we want to look at the values for CP for various tree sizes. We look for the number of splits where the sum of the xerror (cross-validation error relative to the root node error) and xstd is minimum (as discussed in Section 11.4). This is usually early in the list.

The `plotcp()` command is useful in visualising the progression of the CP values. In the following example,[1] we build a full decision tree with

[1]Note that the *cptable* may vary slightly between different deployments of R, particularly between 64 bit R, as here, and 32 bit R.

both `cp=` and `minbucket=` set to zero. We also show the CP table. The corresponding plot is shown in Figure 11.5.

```
> set.seed(41)
> evalq({
    control <- rpart.control(cp=0, minbucket=0)
    model <- rpart(formula=form,
                   data=data[train, vars],
                   control=control)
  }, weatherRPART)
> print(weatherRPART$model$cptable)

       CP nsplit rel error xerror   xstd
1 0.15854      0   1.00000 1.0000 0.1431
2 0.07317      2   0.68293 0.9024 0.1372
3 0.04878      3   0.60976 0.9024 0.1372
4 0.03659      7   0.41463 0.8780 0.1357
5 0.02439     10   0.29268 0.8780 0.1357
6 0.01829     13   0.21951 1.0488 0.1459
7 0.01220     21   0.02439 1.1463 0.1511
8 0.00000     23   0.00000 1.2439 0.1559

> plotcp(weatherRPART$model)
> grid()
```

The figure illustrates a typical behaviour of model building. As we proceed to build a complex model, the error rate (the y-axis) initially decreases. It then flattens out and, as the model becomes more complex, the error rate begins to again increase. We will want to choose a model where it has flattened out. Based on the principle of favouring simpler models, we might choose the first of the similarly performing bottom points and thus we might set `cp=` 0.1, for example.

As a script, we could automate the selection with the following:

```
> xerr <- weatherRPART$model$cptable[,"xerror"]
> minxerr <- which.min(xerr)
> mincp <- weatherRPART$model$cptable[minxerr, "CP"]
> weatherRPART$model.prune <- prune(weatherRPART$model,
                                    cp=mincp)
```

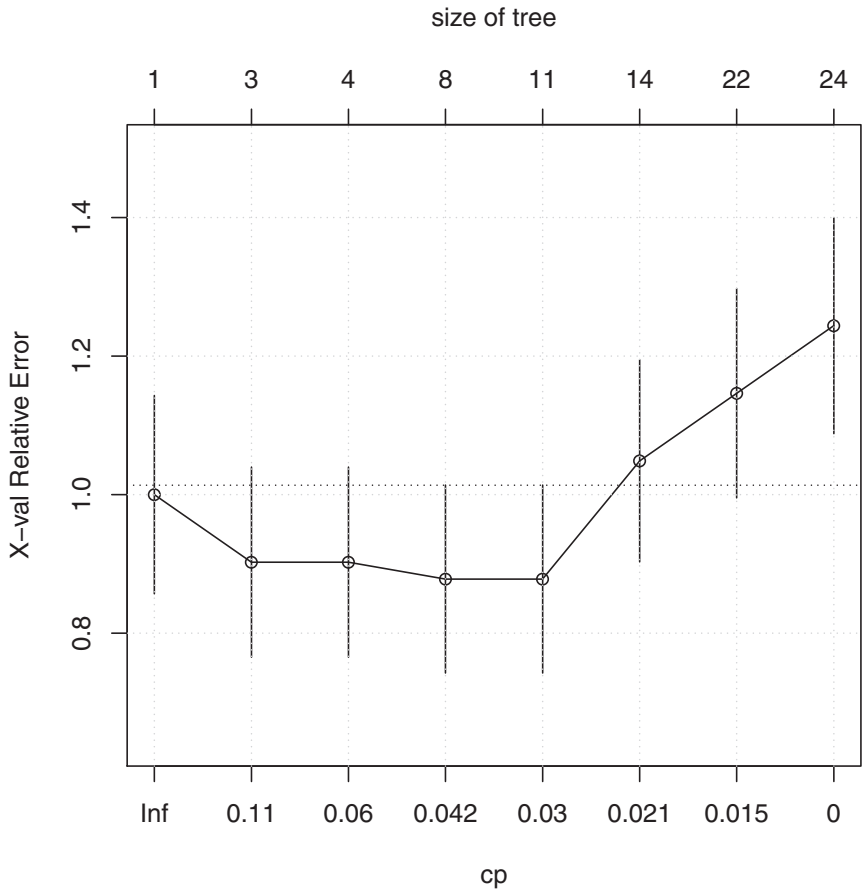


Figure 11.5: Error rate versus complexity/tree size.

Priors (`prior=`)

Sometimes the proportions of classes in a training set do not reflect their true proportions in the population. We can inform Rattle and R of the population proportions, and the resulting model will reflect them. All probabilities will be modified to reflect the prior probabilities of the classes rather than the actual proportions exhibited in the training dataset.

The priors can also be used to "boost" a particularly important class, by giving it a higher prior probability, although this might best be done through the loss matrix (Section 11.5).

In Rattle, the priors are expressed as a list of numbers that sum to 1. The list must be of the same length as the number of unique classes in the training dataset. An example for binary classification is `0.6,0.4`. This translates into `prior=c(0.6,0.4)` for the call to `rpart()`.

The following example illustrates how we might use the priors to

favour a particular target class that was otherwise not being predicted by the resulting model (because the resulting model turns out to be only a root node always predicting `No`). We begin by creating the dataset object, consisting of the larger Australian weather dataset, *weatherAUS*:

```
> wausDS <- new.env()
> evalq({
    data <- weatherAUS
    nobs <- nrow(data)
    form <- formula(RainTomorrow ~ RainToday)
    target <- all.vars(form)[1]
    set.seed(42)
    train <- sample(nobs, 0.5*nobs)
  }, wausDS)
```

A decision tree model is then built and displayed:

```
> wausRPART <- new.env(parent=wausDS)
> evalq({
    model <- rpart(formula=form, data=data[train,])
    model
  }, wausRPART)
n=19509 (489 observations deleted due to missingness)

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 19509 4491 No (0.7698 0.2302) *
```

A table shows the proportion of observations assigned to each class in the training dataset.

```
> evalq({
    freq <- table(data[train, target])
    round(100*freq/length(train), 2)
  }, wausRPART)
   No   Yes
75.66 22.74
```

Now we build a decision tree model but with different prior probabilities:

```
> evalq({
    model <- rpart(formula=form,
                   data=data[train,],
                   parm=list(prior=c(0.5, 0.5)))
    model
  }, wausRPART)
n=19509 (489 observations deleted due to missingness)

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 19509 9754 Yes (0.5000 0.5000)
  2) RainToday=No 15042 5098 No (0.6180 0.3820) *
  3) RainToday=Yes 4467 1509 Yes (0.2447 0.7553) *
```

The default priors when using `raprt()` without the `prior=` option are set to be the class proportions as found in the training dataset supplied.

Loss Matrix (`loss=`)

The loss matrix is used to weight different kinds of errors (or loss) differently. This refers to what are commonly known as false positives (or type I errors) and false negatives (or type II errors) when we talk about a two-class problem.

Often, one type of error is more significant than another type of error. In fraud, for example, a model that identifies too many false positives is probably better than a model that identifies too many false negatives (because we then miss too many real frauds). In medicine, a false positive means that we diagnose a healthy patient with a disease, whilst a false negative means that we diagnose an ill patient as being healthy.

The default loss for each of the true/false positives/negatives is 1—they are all of equal impact or loss. In the case of a rare, and under-represented class (like fraud) we might consider false negatives to be four or even ten times worse than a false positive. Thus, we communicate this to the algorithm so that it will work harder to build a model to find all of the positive cases.

The loss matrix records these relative weights for the two class case only. The following table illustrates the terminology (showing predicted

versus observed):

$Pr\ v\ Ob$	0	1
0	TN	FN
1	FP	TP

Noting that we do not specify any weights in the loss matrix for the true positives (TP) and the true negatives (TN), we supply weights of 0 for them in the matrix. To specify the matrix in the Rattle interface, we supply a list of the form: $0, FN, FP, 0$.

In general, the loss matrix must have the same dimensions as the number of classes (i.e., the number of levels of the target variable) in the training dataset. For binary classification, we must supply four numbers with the diagonals as zeros.

An example is the string of numbers $0, 10, 1, 0$, which might be interpreted as saying that an actual 1 predicted as 0 (i.e., a false negative) is ten times more unwelcome than a false positive. This is used to construct, row-wise, the loss matrix which is passed through to `rpart()` as `loss=loss=matrix(c(0,10,1,0), byrow=TRUE, nrow=2)`.

The loss matrix is used to alter the priors, which will affect the choice of variable on which to split the dataset on at each node, giving more weight where appropriate.

Using the loss matrix is often indicated when we build a decision tree that ends up being just a single root node (often because the positive class represents less than 5% of the population—and so the most accurate model would predict everyone to be a negative).

Other Options

The `rpart()` function provides many other tuning parameters that are not exposed through the Rattle GUI. These include `maxdepth=` to limit the depth of a tree and `maxcompete=` to limit the number of competing alternative splits for each node that is retained in the resulting model.

A number of options relate to the handling of surrogates. As indicated above, surrogates in the model allow for the handling of missing values. The `surrogatestyle=` argument indicates how surrogates are given preference. The default is to prefer variables with fewer missing values in the training dataset, with the alternative being to sort them by the percentage correct over the number of nonmissing values.

The `usesurrogate=` argument controls how surrogates are made use of in the model. The default for the `usesurrogate=` argument is 2.

This is also set when Rattle's Include Missing check button is active. The behaviour here is to try each of the surrogates whenever the main variable has a missing value, but if all surrogates are also missing, then follow the path with the majority of cases. If `usesurrogate=` is set to 1, the behaviour is to try each of the surrogates whenever the main variable has a missing value, but if all surrogates are also missing, then go no further. When the argument is set to 0 (the case when Rattle's Include Missing check button is not active), the observation with a missing value for the main variable is not used any further in the tree building.

The `maxsurrogate=` argument simply limits the number of surrogates considered for each node.

11.6 Discussion

Decision trees have been around for a long time. They present a mechanism for structuring a series of questions. The next question to ask, at any time, is based on the answer to a previous question.

In data mining, we commonly identify decision trees as the knowledge representation scheme targeted by the family of techniques originating from ID3 within the machine learning community (Quinlan, 1986) and from CART within the statistics community. The original ID3 algorithm was extended to become the commercially available C4.5 software. This was made available together with a book by Quinlan (1993) that served as a guide to using the code.

Traditional decision tree algorithms can suffer from overfitting and can exhibit a bias towards selecting variables with many possible splits (i.e., categoric variables). The algorithms do not use any statistical significance concepts and thus, as noted by Mingers (1989), cannot distinguish between significant and insignificant improvements in the information measure. The use of a cross-validated relative error measure, as in the implementation in `rpart()` does guard against overfitting.

Hothorn et al. (2006) introduced an improvement to the approach presented here for building a decision tree, called conditional inference trees. Rattle offers the choice of traditional and conditional algorithms. Conditional inference trees address overfitting and variable selection biases by using a conditional distribution to measure the association between the output and the input variables. They take into account distributional properties.

Conditional inference trees can be built using `ctree()` from **party** (Hothorn et al., 2006). Within Rattle, we can choose the Conditional option to build a conditional inference tree. From the command line, we would use the following call to `ctree()`:

```
> library(party)
> weatherCTREE <- new.env(parent=weatherDS)
> evalq({
    model <- ctree(formula=form, data=data[train, vars])
  }, weatherCTREE)
```

We can review just lines 8 to 17 of the resulting output, which is the tree itself:

```
> cat(paste(capture.output(weatherCTREE$model)[8:17],
            collapse="\n"))

1) Pressure3pm <= 1012; criterion = 1, statistic = 39.281
  2) Sunshine <= 8.8; criterion = 0.99, statistic = 12.099
    3)*  weights = 27
  2) Sunshine > 8.8
    4)*  weights = 25
1) Pressure3pm > 1012
  5) Cloud3pm <= 7; criterion = 1, statistic = 20.825
    6)*  weights = 195
  5) Cloud3pm > 7
    7)*  weights = 9
```

A plot of the tree is presented in Figure 11.6. The plot is quite informative and primarily self-explanatory. Node 3, for example, predicts rain relatively accurately, whilst node 6 describes conditions under which there is almost never any rain on the following day.

11.7 Summary

Decision tree algorithms handle mixed types of variables and missing values, and are robust to outliers and monotonic transformations of the input and to irrelevant inputs. The predictive power of decision trees tends to be poorer than for other techniques that we will introduce. However, the algorithm is generally straightforward, and the resulting

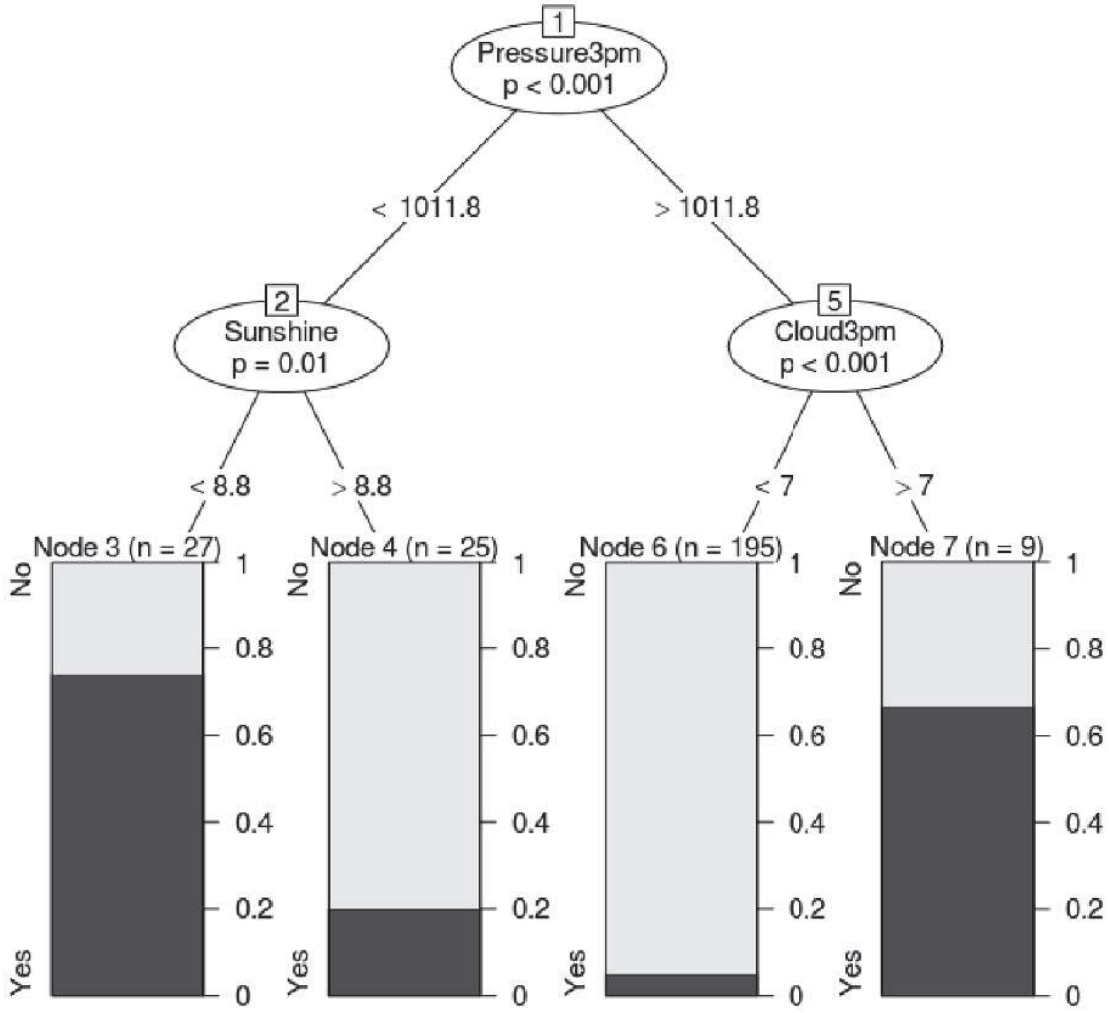


Figure 11.6: A conditional inference tree.

models are generally easily interpretable. This last characteristic has made decision tree induction very popular for over 30 years.

This chapter has introduced the basic concept of representing knowledge as a decision tree and presented a measure for choosing a good decision tree and an algorithm for building one.

11.8 Command Summary

This chapter has referenced the following R packages, commands, functions, and datasets:

`ctree()`	function	Build a conditional inference tree.
`draw.tree()`	command	Enhanced graphic decision tree.
maptree	package	Provides `draw.tree()`.
party	package	Conditional inference trees.
`path.rpart()`	function	Identify paths through decision tree.
`plot()`	command	Graphic display of the tree.
`plotcp()`	command	Plot complexity parameter.
`print()`	command	Textual version of the decision tree.
`printcp()`	command	Complexity parameter table.
rattle	package	The *weather* dataset and GUI.
`rpart()`	function	Build a decision tree predictive model.
rpart	package	Provides decision tree functions.
rpart.control	function	Organise `rpart` control arguments.
`set.seed()`	function	Initiate random seed number sequence.
`summary()`	command	Summary of the tree building process.
`text()`	command	Add labels to decision tree graphic.
weather	dataset	Sample dataset from **rattle**.

Chapter 12

Random Forests

Building a single decision tree provides a simple model of the world, but it is often too simple or too specific. Over many years of experience in data mining, it has become clear that many models working together are better than one model doing it all. We have now become familiar with the idea of combining multiple models (like decision trees) into a single ensemble of models (to build a forest of trees).

Compare this to how we might bring together panels of experts to ponder an issue and to then come up with a consensus decision. Governments, industry, and universities all manage their business processes in this way. It can often result in better decisions compared to simply relying on the expertise of a single authority on a topic.

The idea of building multiple trees arose early on with the development of the multiple inductive learning (MIL) algorithm (Williams, 1987, 1988). In building a single decision tree, it was noted that often there was very little difference in choosing between alternative variables. For example, two or more variables might not be distinguishable in terms of their ability to partition the data into more homogeneous datasets. The MIL algorithm builds all "equally" good models and then combines them into one model, resulting in a better overall model.

Today we see a number of algorithms generating ensembles, including boosting, bagging, and random forests. In this chapter, we introduce the random forest algorithm, which builds hundreds of decision trees and combines them into a single model.

G. Williams, *Data Mining with Rattle and R: The Art of Excavating Data for Knowledge Discovery*, Use R, DOI 10.1007/978-1-4419-9890-3_12, © Springer Science+Business Media, LLC 2011