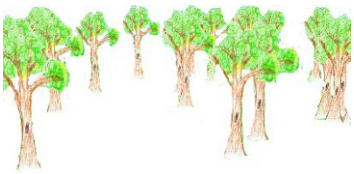
Chapter 12

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

12.1 Overview

The random forest algorithm tends to produce quite accurate models because the ensemble reduces the instability that we can observe when we build single decision trees. This can often be illustrated simply by removing a very small number of observations from the training dataset, to see quite a change in the resulting decision tree.

The random forest algorithm (and other ensemble algorithms) tends to be much more robust to changes in the data. Hence,it is very robust to noise (i.e., variables that have little relationship to the target variable). Being robust to noise means that small changes in the training dataset will have little, if any, impact on the final decisions made by the resulting model. Random forest models are generally very competitive with nonlinear classifiers such as artificial neural nets and support vector machines.

Random forests handle underrepresented classification tasks quite well. This is where, in the binary classification task, one class has very few (e.g., 5% or fewer) observations compared with the other class.

By building each decision tree to its maximal depth, as the random forest algorithm does (by not pruning the individual decision trees), we can end up with a model that is less biased. Each individual tree will overfit the data, but this is outweighed by the multiple trees using different variables and (over) fitting the data differently.

The randomness used by a random forest algorithm is in the selection of both observations and variables. It is this randomness that delivers considerable robustness to noise, outliers, and overfitting, when compared with a single-tree classifier.

The randomness also delivers substantial computational efficiencies. In building a single decision tree, the model builder may select a random subset of the observations available in the training dataset. Also, at each node in the process of building the decision tree, only a small fraction of all of the available variables are considered when determining how to best partition the dataset. This substantially reduces the computational requirement.

In the area of genetic marker selection and microarray data within bioinformatics, for example, random forests have been found to be particularly well suited. They perform well even when many of the input variables have little bearing on the target variable (i.e., they are noise variables). Random forests are also suitable when there are very many input variables and not so many observations.

In summary, a random forest model is a good choice for model building for a number of reasons. Often, very little preprocessing of the data needs to be performed, as the data does not need to be normalised and the approach is resilient to outliers. The need for variable selection is avoided because the algorithm effectively does its own. Because many trees are built using two levels of randomness (observations and variables), each tree is effectively an independent model and the resulting model tends not to overfit to the training dataset.

12.2 Knowledge Representation

The random forest algorithm is commonly presented in terms of decision trees as the primary form for representing knowledge. However, the random forest algorithm can be thought of as a meta-algorithm. It describes an approach to building models where the actual model builder could be a decision tree algorithm, a regression algorithm, or any one of many other kinds of model building algorithms. The general concepts apply to any of these approaches. We will stay with decision trees as the underlying model builder for our purposes here.

In any ensemble approach, the key extension to the knowledge representation is in the way that we combine the decisions that are made by the individual "experts" or models. Various approaches have been considered over the years. Many come from the knowledge-based and expert systems communities, which often need to consider the issue of combining expert knowledge from multiple experts. Approaches to aggregating decisions into one final decision include simple majority rules and a weighted score where the weights correspond to the quality of the expertise (e.g., the measured accuracy of the individual tree).

The random forest algorithms will often build from 100 to 500 trees. In deploying the model, the decisions made by each of the trees are combined by treating all trees as equals. The final decision of the ensemble will be the decision of the majority of the constituent trees. If 80 out of 100 trees in the random forest say that it will rain tomorrow, then we will go with that decision and take the appropriate action for rain. Even if 51 of the 100 trees say that it will rain, we might go with that, although perhaps with less certainty. In the context of regression rather than classification, the result is the average value over the ensemble of regression trees.

12.3 Algorithm

Chapter 11 covered the building of an individual tree, and the same algorithm can be used for building one or 500 trees. It is how the training set is selected and how the variables to use in the modelling are chosen that differs between the trees built for a random forest.

Sampling the Dataset

The random forest algorithm builds multiple decision trees, using a concept called *bagging*, to introduce random sampling into the whole process. Bagging is the idea of collecting a random sample of observations into a bag (though the term itself is an abbreviation of *bootstrap aggregation*). Multiple bags are made up of randomly selected observations obtained from the original observations from the training dataset.

The selection in bagging is made with replacement, meaning that a single observation has a chance of appearing multiple times within a single bag. The sample size is often the same as for the full dataset, and so in general about two-thirds of the observations will be included in the bag (with repeats) and one-third will be left out. Each bag of observations is then used as the training dataset for building a decision tree (and those left out can be used as an independent sample for performance evaluation purposes).

Sampling the Variables

A second key element of randomness relates to the choice of variables for partitioning the dataset. At each step in building a single decision node (i.e., at each split point of the tree), a random, and usually small, set of variables is chosen. Only these variables are considered when choosing a split point. For each node in building a decision tree, a different random set of variables is considered.

Randomness

By randomly sampling both the data and the variables, we introduce decision trees that purposefully have different performance behaviours for different subsets of the data. It is this variation that allows us to consider an ensemble of such trees as representing a team of experts with differing expertise working together to deliver a "better" answer.

Sampling also offers another significant advantage—computational efficiency. By considering only a small fraction of the total number of variables available, whilst considering split points, the amount of computation required is significantly reduced.

In building each decision tree, the random forest algorithm generally will not perform any pruning of the decision tree. When building a single decision tree, it was noted in Chapter 11 that pruning was necessary to avoid overfitting the data. Overfitted models tend not to perform well on new data. However, a random forest of overfitted trees can deliver a very good model that performs well on new data.

Ensemble Scoring

In deploying the multiple decision trees as a single model, each tree has equal weight in the final decision-making process. A simple majority might dictate the outcome. Thus, if 300 of 500 decision trees all predict that it will rain tomorrow, then we might be inclined to expect there to be rain tomorrow. If only 100 trees of the 500 predict rain tomorrow, then we might not expect rain.

12.4 Tutorial Example

Our task is again to predict the likelihood of rain tomorrow given today's weather conditions. We illustrate this using Rattle and directly through R. In both cases, **randomForest** (Liaw and Wiener, 2002) is used. This package provides direct access to the original implementation of the random forest algorithm by its authors.

Building a Model using Rattle

Rattle's Model tab provides the Forest option to build a forest of decision trees. Figure 12.1 displays the graphical interface to the options for building a random forest with the default values and also shows the top part of the results from building the random forest shown in the Textview area.

We now step through the output of the text view line by line. The first few lines note the number of observations used to build the model and then an indication that missing values in the training dataset are being imputed. If missing value imputation is not enabled, then the

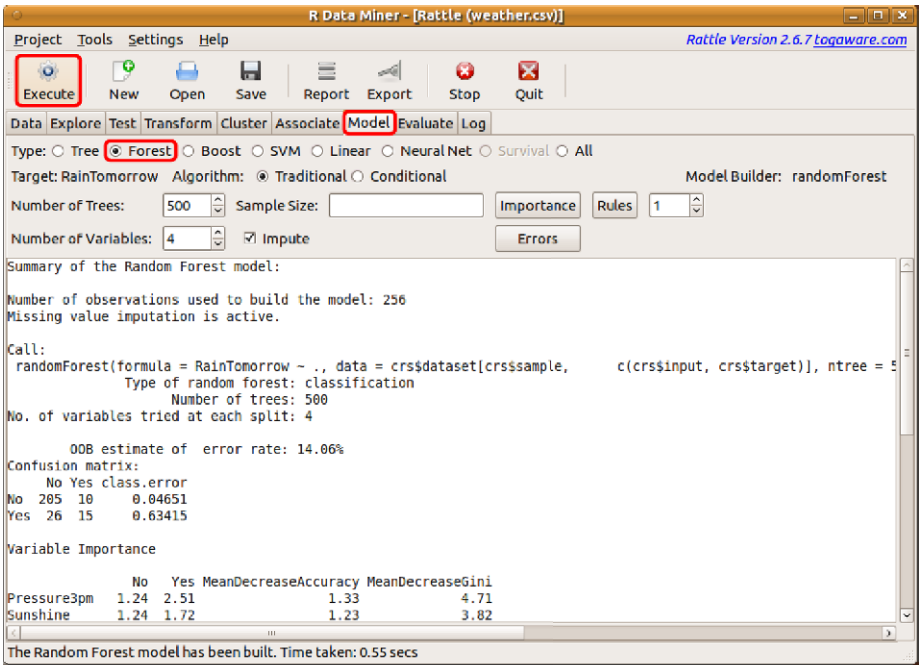


Figure 12.1: Building a random forest predictive model.

number of observations may be less than the number available, as the default is to drop observations containing missing values.

```
Summary of the Random Forest model:

Number of observations used to build the model: 256
Missing value imputation is active.
```

The next few lines record the actual function command line call that Rattle generated and passed onto R to be evaluated:

```
Call:
 randomForest(formula = RainTomorrow ~ .,
      data = crs$dataset[crs$sample,
                         c(crs$input, crs$target)],
      ntree = 500, mtry = 4, importance = TRUE,
      replace = FALSE, na.action = na.roughfix)
```

A more detailed dissection of the function call is presented later, but

in brief, 500 trees were asked for (`ntree=`) and just four variables were considered for the split point for each node (`mtry=`). An indication of the importance of variables is maintained (`importance=`), and any observations with missing values will have those values imputed (`na.action=`).

The next few lines summarise some of the same information in a more accessible form. Note that, due to numerical differences, specific results may vary slightly between 32 bit and 64 bit deployments of R. The following was performed on a 64 bit deployment of R:

```
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 4
```

Performance Evaluation

Next comes an indication of the performance of the resulting model. The out-of-bag (OOB) estimate of the error rate is calculated using the observations that are not included in the "bag"—the "bag" is the subset of the training dataset used for building the decision tree, hence the "out-of-bag" terminology.

This "unbiased" estimate of error suggests that when the resulting model is applied to new observations, the answers will be in error 14.06% of the time. That is, it is 85.94% accurate, which is a reasonably good model.

```
        OOB estimate of  error rate: 14.06%
```

This overall measure of accuracy is then followed by a confusion matrix that records the disagreement between the final model's predictions and the actual outcomes of the training observations. The actual observations are the rows of this table, whilst the columns correspond to what the model predicts for an observation and the cells count the number of observations in each category. That is, the model predicts `Yes` and the observation was `No` for 26 observations.

```
Confusion matrix:
     No Yes class.error
No  205  10     0.04651
Yes  26  15     0.63415
```

We see that the model and the training dataset agree that it won't rain for 205 of the observations. They agree that it will rain for 15 of the observations. However, there are 26 days for which the model predicts that it does not rain the following day and yet it does rain. Similarly, the model predicts that it will rain the following day for ten of the observations when in fact it does not rain.

The overall class errors, also calculated from the out-of-bag data, are included in the table. The model is wrong in predicting rain when there is none in only 63.41% of the observations when there is no rain. This is contrasted with the 4.65% error rate in predicting that it does rain tomorrow.

Underrepresented Classes

The acceptability of such errors (false positives versus false negatives) depends on many factors. Predicting that it will rain tomorrow and getting it wrong (false positive) might be an inconvenience in terms of carrying an umbrella around for the day. However, predicting that it won't rain and not being prepared for it (false negative) could result in a soggy dash for cover. The 63.41% error rate in predicting that it does not rain might be a concern.

One approach with random forests in addressing the "seriousness" associated with the false negatives might be to adjust the balance between the underrepresented class (66 observations have `RainTomorrow` as `Yes`) and the overrepresented class (300 observations have `RainTomorrow` as `No`). In the training dataset the observations are 41 and 215, respectively (after removing any observations with missing values).

We can use the Sample Size option to encourage the algorithm to be more aggressive in predicting that it will rain tomorrow. We will balance up the sampling so that equal numbers of observations with `Yes` and `No` are chosen. Specifying a value of `35,35` for the sample size will do this. The confusion matrix for the resulting random forest is:

```
        OOB estimate of  error rate: 28.52%
Confusion matrix:
     No Yes class.error
No  147  68      0.3163
Yes   5  36      0.1220
```

The error rate for when it does rain tomorrow is now 12.2%, and now we'll get wet 5 days out of 41 when it does rain, which is better than 26 days out of 41 days on which we'll end up getting wet.

The price we pay for this increased accuracy in predicting when it rains, is that we now have more days predicted as raining when in fact it does not rain. The "business problem" here indicates that carrying an umbrella with us unnecessarily is less of a burden than getting wet when it rains and we don't have our umbrella. We are also assuming that we don't want to carry our umbrella all the time.

Variable Importance

One of the problems with a random forest, compared with a single decision tree, is that it becomes quite a bit more difficult to readily understand the discovered knowledge—there are 500 trees here to try to understand. One way to get an idea of the knowledge being discovered is to consider the importance of the variables, as emerges from their use in the building of the 500 decision trees.

A variable importance table is the next piece of information that appears in the text view (we reformat it here to fit the limits of the page):

```
Variable Importance

                 No   Yes  Accu  Gini
Pressure3pm    1.24  2.51  1.33  4.71
Sunshine       1.24  1.72  1.23  3.82
Cloud3pm       1.13  1.90  1.16  3.19
WindGustSpeed  0.99  0.97  0.91  2.58
Pressure9am    1.03 -0.11  0.87  2.89
Temp3pm        0.83 -0.50  0.71  1.50
Humidity3pm    0.79  0.04  0.65  2.27
MaxTemp        0.61 -0.10  0.55  1.73
Temp9am        0.52  0.20  0.50  1.50
WindSpeed9am   0.56 -0.12  0.46  1.39
```

The table lists each input variable and then four measures of importance for each variable. Higher values indicate that the variable is relatively more important. The table is sorted by the Accuracy measure of importance.

A näive approach to measuring variable importance is to count the number of times the variable appears in the ensemble of decision trees. This is a rather blunt measure as, for example, variables can appear at different levels within a tree and thus have different levels of importance. Most measures thus incorporate some measure of the improvement made to the tree by each variable.

The third importance measure is a scaled average of the prediction accuracy of each variable. The calculation is based on a process of randomly permuting the values of a variable across the observations and measuring the impact on the predictive accuracy of the resulting tree. The larger the impact then the more important the variable is. Thus this measure reports the mean decrease in the accuracy of the model. The actual magnitude of the measure is not so relevant as the relative positioning of variables by the measure.

The final measure of importance is the total decrease in a decision tree node's impurity (the splitting criterion) when splitting on a variable. The splitting criterion used is the Gini index. This is measured for a variable over all trees giving a measure of the mean decrease in the Gini index of diversity relating to the variable.

The Importance button displays a visual plot of the accuracy and the Gini importance measures, as shown in Figure 12.2, and is more effective in illustrating the relative importance of the variables. Clearly, `Pressure3pm` is the most important variable, and then `Sunshine`. The accuracy measure then lists `Cloud3pm` and the next most important. This is consistent with the decision tree we built in Chapter 11. What we did not learn in building the decision tree is that `Pressure9am` is also quite important, and that the remainder are less so, at least according to the accuracy measure.

We also notice that the categoric variables (like the wind direction variables `WindGustDir`, `WindDir9am`, and `WindDir3pm`) have a higher importance according to the Gini measure than with the accuracy measure. This bias towards categoric variables with many categories, exhibited in the Gini measure, is discussed further in Section 12.6. It is noteworthy that this bias will mislead us about the importance of these categoric variables.

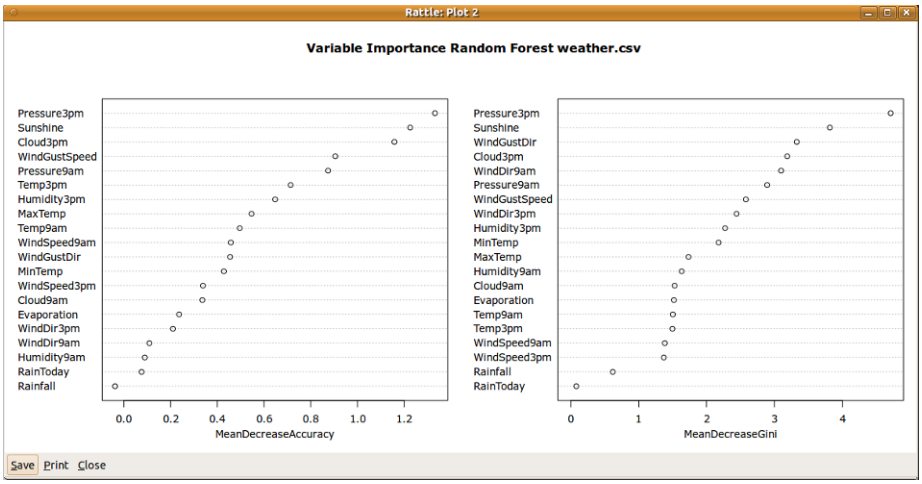


Figure 12.2: Two measures of variable importance as calculated by the random forest algorithm.

Time Taken

The tail of the textview provides information on how long it took to build the random forest of 500 trees. Note that even though we are building so many decision trees, the time taken is still less than 1 second.

Tuning Options

The Rattle interface provides a choice of Algorithm for building the random forest. 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 for building the decision trees. This is explained in more detail in Section 12.6. A small number of other tuning options are also provided, and they are discussed in some detail in Section 12.5.

Error Plots

A useful diagnostic tool is the error plot, obtained with a click of the Error button. Figure 12.3 shows the resulting error plot for our random forest model.

The plot reports the accuracy of the forest of trees (in terms of error rate on the y-axis) against the number of trees that have been included

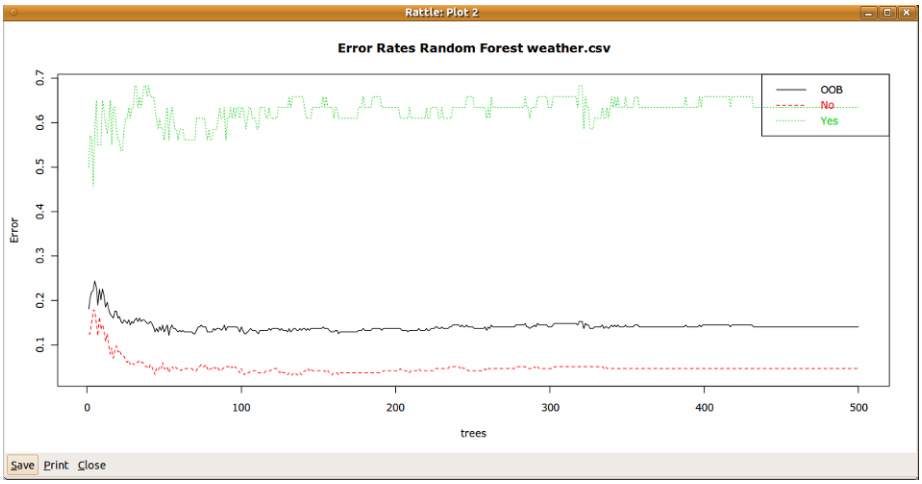


Figure 12.3: The error rate of the overall model gnerally decreases as each new tree is added to the ensemble.

in the forest (the x-axis). The key point we take from this plot is that after some number of trees there is actually very little that changes by adding further trees to the forest. From Figure 12.3 it would appear that going beyond about 20 trees in the forest adds very little value, when considering the out-of-bag (OOB) error rate.

The two other plots show the changes in error rate associated with the predictions of the model (here we have two classes predicted and so two additional lines). We also take these into account when deciding how many trees to add to the forest.

Conversion to Rules

Another button available with the Forest option is the Rules button, with an associated text entry box. Clicking this button will convert the specified tree into a set of rules. If the tree specified is 0 (rather than, for example, the default 1), then all trees will be converted to rules. Be careful, though, as that could take a very long time for 500 trees and 20 or more rules per tree (10,000 rules). The first two rules from tree 1 of the random forest are shown in the following code block.

```
Random Forest Model 1

Tree 1 Rule 1 Node 28 Decision No

1: Sunshine <= 6.45
2: Cloud9am <= 7.5
3: WindGustSpeed <= 43.5
4: Humidity3pm <= 36.5
5: MaxTemp <= 22.45

Tree 1 Rule 2 Node 29 Decision Yes

1: Sunshine <= 6.45
2: Cloud9am <= 7.5
3: WindGustSpeed <= 43.5
4: Humidity3pm <= 36.5
5: MaxTemp > 22.45
```

Building a Model Using R

As usual, we will create a container into which we place the relevant information for the modelling. We set up some useful variables within the container (using `evalq()`) as well as constructing the training and test datasets based on a random sample of 70% of the observations, including only those columns (i.e., dataset variables) that are not identified as being ignored (which is a list of negative indices, and thus indicates which columns not to include).

```
> library(rattle)
> weatherDS <- new.env()
> evalq({
    data <- na.omit(weather)
    nobs <- nrow(data)
    form <- formula(RainTomorrow ~ .)
    target <- all.vars(form)[1]
    vars <- -grep('^(Date|Location|RISK_)', names(data))
    set.seed(42)
    train <- sample(nobs, 0.7*nobs)
  }, weatherDS)
```

Considering the formula, the variable `RainTomorrow` is the target, with all remaining variables (`~ .`) from the provided dataset as the input variables.

Next we build the random forest. We first generate our training dataset as a random sample of 70% of the supplied dataset, noting that we reset the random number generator's seed back to a known number for repeatability. The data itself consists of the observations contained in the training dataset.

```
> library(randomForest)
> weatherRF <- new.env(parent=weatherDS)
> evalq({
    model <- randomForest(formula=form,
                          data=data[train, vars],
                          ntree=500, mtry=4,
                          importance=TRUE,
                          localImp=TRUE,
                          na.action=na.roughfix,
                          replace=FALSE)
  }, weatherRF)
```

The remaining arguments to the function are explained in Section 12.5.

Exploring the Model

The `model` object itself contains quite a lot of information about the model that has been built. The `str()` command gives the definitive list of all the components available within the object. An explanation is also available through the help page for `randomForest()`:

```
> str(weatherRF$model)
> ?randomForest
```

We consider some of the information stored within the object here.

The `predicted` component contains the values predicted for each observation in the training dataset based on the out-of-bag samples. If an observation is never in an out-of-bag sample then the prediction will be reported as `NA`. Here we show just the first ten predictions:

```
> head(weatherRF$model$predicted, 10)

336 342  94 304 227 173 265  44 230 245
 No  No  No  No  No  No  No  No  No  No
Levels: No Yes
```

The `importance` component records the information related to measures of variable importance as discussed in detail in Section 12.4, page 253. The information is reported for four measures (columns).

```
> head(weatherRF$model$importance)

                    No         Yes MeanDecreaseAccuracy
MinTemp      0.0031712  0.0056410             0.0036905
MaxTemp      0.0092405  0.0003834             0.0077143
Rainfall     0.0014129 -0.0033066             0.0005476
Evaporation  0.0006489 -0.0040790            -0.0002857
Sunshine     0.0211487  0.0445901             0.0251667
WindGustDir  0.0020603  0.0028510             0.0021905
             MeanDecreaseGini
MinTemp                2.2542
MaxTemp                1.8281
Rainfall               0.7377
Evaporation            1.3721
Sunshine               3.9320
WindGustDir            1.2739
```

The importance of each variable in predicting the outcome for each observation in the training dataset can also be available in the model object. This is accessible through the `localImp` component:

```
> head(weatherRF$model$localImp)[,1:4]

                 336        342         94       304
MinTemp      0.011834  0.016575  0.021053  0.00000
MaxTemp      0.000000  0.005525  0.010526 -0.07143
Rainfall     0.005917 -0.005525  0.005263  0.00000
Evaporation  0.000000  0.000000  0.000000 -0.02976
Sunshine     0.035503  0.038674 -0.010526  0.03571
WindGustDir  0.005917 -0.005525  0.005263  0.04762
```

The error rate data is stored as the `err.rate` component. This can be accessed from the `model` object as we see in the following code block:

```
> weatherRF$model$err.rate
```

In Rattle, we saw an error plot that showed the change in error rate as more trees are added to the forest. We can obtain the actual data behind the plot quite easily:

```
> round(head(weatherRF$model$err.rate, 15), 4)

         OOB     No    Yes
 [1,] 0.2738 0.2143 0.5714
 [2,] 0.2701 0.2261 0.5000
 [3,] 0.2560 0.2340 0.3704
 [4,] 0.2273 0.1728 0.4722
 [5,] 0.2067 0.1361 0.5128
 [6,] 0.1872 0.1061 0.5500
 [7,] 0.1570 0.0984 0.4250
 [8,] 0.1689 0.1081 0.4500
 [9,] 0.1404 0.0691 0.4750
[10,] 0.1223 0.0529 0.4500
[11,] 0.1223 0.0582 0.4250
[12,] 0.1048 0.0317 0.4500
[13,] 0.1310 0.0582 0.4750
[14,] 0.1354 0.0529 0.5250
[15,] 0.1223 0.0476 0.4750
```

Here we see that the OOB estimate decreases quickly and then starts to flatten out. We can find the minimum quite simply, together with a list of the indexes where each minimum occurs:

```
> evalq({
    min.err <- min(data.frame(model$err.rate)["OOB"])
    min.err.idx <- which(data.frame(model$err.rate)["OOB"]
                         == min.err)
  }, weatherRF)
```

The actual minimum value together with the indexes can be listed:

```
> weatherRF$min.err

[1] 0.1048

> weatherRF$min.err.idx

[1] 12 45 49 50 51
```

We can then list the actual models where the minimum occurs:

```
> weatherRF$model$err.rate[weatherRF$min.err.idx,]

        OOB      No   Yes
[1,] 0.1048 0.03175 0.450
[2,] 0.1048 0.02116 0.500
[3,] 0.1048 0.01587 0.525
[4,] 0.1048 0.01587 0.525
[5,] 0.1048 0.01587 0.525
```

We might thus decide that 12 (the first instance of the minimum OOB estimate) is a good number of trees to have in the forest.

Another interesting component is `votes`, which records the number of trees that vote `No` and `Yes` within the ensemble for a particular observation.

```
> head(weatherRF$model$votes)

        No      Yes
336 0.9467 0.053254
342 0.9779 0.022099
94  0.8263 0.173684
304 0.8690 0.130952
227 0.9943 0.005682
173 0.9950 0.005025
```

The numbers are reported as proportions and so add up to 1 for each observation, as we can confirm:

```
> head(apply(weatherRF$model$votes, 1, sum))

336 342  94 304 227 173
  1   1   1   1   1   1
```

12.5 Tuning Parameters

Rattle provides access to just a few basic tuning options (Figure 12.1) for the random forest algorithm. The user interface allows the number of trees, the number of variables, and the sample size to be specified. As is generally the case with Rattle, the defaults are a good starting point!

These result in 500 trees being built, choosing from the square root of the number of variables available for each node, and no sampling of the training dataset to balance the classes.

In Figure 12.1, we see that the number of variables has automatically been set to 4 for the *weather* dataset, which has 20 input variables. The user interface options correspond to the function arguments `ntree=`, `ntry=`, and `sampsize=`. Rattle also sets `importance=` to `TRUE`, `replace=` to `FALSE`, and `na.action=` to `na.roughfix()`.

A call to `randomForest()` including all arguments covered here will look like:

```
> evalq({
    model <- randomForest(formula=form,
                          data=data[train, vars],
                          ntree=500,
                          mtry=4,
                          replace=FALSE,
                          sampsize=.632*nobs,
                          importance=TRUE,
                          localImp=FALSE,
                          na.action=na.roughfix)
  }, weatherRF)
```

Number of Trees `ntree=`

This specifies how many trees are to be built to populate the random forest. The default value is 500, and a common recommendation is that a minimum of 100 trees be built. The performance of the resulting random forest model tends not to degrade as the number of trees increases, though computationally it will take longer and will be more complex to use when scoring, and often there is little to gain from adding too many trees to a forest. The error matrix and error plot provide a guide to a good number of trees to include in a forest. See Section 12.4 for examples.

Number of Variables `ntry=`

The number of variables to consider for splitting at every node is specified by `ntry=`. This many variables will be randomly selected from all of those available each time we look to partition a dataset in the process of

building the decision tree. The general default value is the square root of the total number of variables available, for classification tasks and one-third of the number of available variables for regression.

If there are many noise variables (i.e., variables that play little or no role in predicting the outcome), then we might consider increasing the number of variables considered at each node to ensure we have some relevant variables to choose from.

Sample Size `sampsize=`

The sample size argument can be used to force the algorithm to select a smaller sample size than the default or to sample the observations differently based on the output variable values (for classification tasks). For example, if our training dataset contains 5,000 observations for which it does not rain tomorrow and only 500 for which it does rain tomorrow, we can specify the sample size as `400,400`, for example, to have equal weight on both outcomes. This provides a mechanism for effectively setting the prior probabilities. See Section 12.4 for an example of doing this in Rattle.

Variable Importance `importance=`

The importance argument allows us to review the importance of each variable in determining the outcome. Two importance measures are calculated in addition to importance of the variable in relation to each outcome in a classification task. These have been described in Section 12.4, and issues with the measures are discussed in Section 12.6.

Sampling with Replacement `replace=`

By default, the sampling is performed when the training observations are sampled for building a particular tree within the forest samples with replacement. This means that any particular observation might appear multiple times within the sample, and thus some observations get over-represented in some datasets. This is a feature of the approach. The `replace=` argument set to `FALSE` will perform sampling without replacement.

Handling Missing Values `na.action=`

The implementation of the `randomForest()` algorithm does not directly handle missing values. A common approach on finding missing values is simply to ignore the observation with missing values by specifying `na.omit` as the value of `na.action=`. For some data, this could actually end up removing all observations from the training dataset. Another quick option is to replace missing values with the median (for numeric data) or the most frequent value (for categoric data) using `na.roughfix`.

12.6 Discussion

Brief History and Alternative Approaches

The concept of an ensemble of experts was something that the knowledge based and expert systems research communities were exploring in the 1980's. Some early work on building and combining multiple decision trees was undertaken at that time (Williams, 1988). Multiple decision trees were built by choosing different variables at nodes where the choice of variable was not clear. The resulting ensemble was found to produce a better predictive model.

Ho (1995, 1998) then developed the concept of randomly sampling variables to build the ensemble of decision trees. Half of the available variables were randomly chosen for building each of the decision tree. She noted that as more trees were added to the ensemble, the predictive performance increased, mostly monotonically.

Breiman (2001) built on the idea of randomly sampling variables by introducing random sampling of variables at each node as the decision tree is built. He also added the concept of bagging (Breiman, 1996) where different random samples of the dataset are chosen as the training dataset for each tree. His algorithm is in common use today, and his actual implementation can be accessed within R through **randomForest**.

In some situations we will have available a huge number of variables to choose from. Often only a small proportion of the available variables will have some influence on the target variable. By randomly selecting a small proportion of the available variables we will often miss the more relevant variables in building our trees.

An approach to address this situation introduces a weighted variable selection scheme to implement an enriched random forest (Amaratunga

et al., 2008). Weights can be based on the q-value, derived from the p-value for a two-sample t-test. We test for a group mean effect of a variable, testing how well the variable can separate the values of the binary target variable. The resulting weights then bias the random selection of variables toward those that have more influence on the target variable.

An extension to this method allows it to work when the target variable has more than two values. In that case we can use a chi-square or information gain measure. The approach can be shown to produce considerably more accuracte models, by ensuring each decision tree has a high degree of independence from the other trees and by weighting the sampling of the variables to ensure important variables are selected for each tree.

Using Other Random Forests

The `randomForest()` function can also be applied to regression tasks, survival analysis, and unsupervised clustering (Shi and Horvath, 2006).

Limitation on Categories

An issue with the implementation of random forests in R is that it can not handle categoric data with more than 32 categoric values. Statistical concerns also suggest that categoric variables with more than 32 categories don't make a lot of sense, and thus little effort has been made in the R package to rectify the issue.

Importance Measures

We introduced the idea of measures of variable importance in building a model in Section 12.4. There we looked at the mean decrease in accuracy and the mean decrease in the Gini index as two measures calculated whilst the trees of the random forest are being built.

These variable importance measures provided by `randomForest()` have been found to be unreliable under certain conditions. The issue particularly arises where there is a mix of numeric and categoric variables or the numeric variables have quite different scales (e.g., `Age` versus `Income`), or then categoric variables have very different numbers of categories (Strobl et al., 2007). Less important variables can end up having too high an importance according to the measures used, and thus we will

be misled into believing the measures provided. Indeed, the Gini measure can be quite biased, so that categorics with many categories obtain a higher importance.

The `cforest()` function of **party** (Hothorn et al., 2006) provides an improved importance measure. This newer measure can be applied to any dataset, using subsampling without replacement, to give a more reliable measure of variable importance. A key aspect is that rather than sampling the data with replacement to obtain a same size sample, a random subsample is used.

Underneath, `cforest()` builds conditional decision trees by using the `ctree()` function discussed in Chapter 11. In the following code block we first load **party** into the library and we create a new data structure to store our forest object, attaching the *weather* dataset to the object.

```
> library(party)
> weatherCFOREST <- new.env(parent=weatherDS)
```

Now we can build the model itself with a call to `cforest()`:

```
> evalq({
    model <- cforest(form,
                     data=data[vars],
                     controls=cforest_unbiased(ntree=50,
                       mtry=4))
  }, weatherCFOREST)
```

We could now explore the resulting forest, but here we will simply list the top few most important variables, according to the measure used by **party**:

```
> evalq({
    varimp <- as.data.frame(sort(varimp(model),
                                 decreasing=TRUE))
    names(varimp) <- "Importance"
    head(round(varimp, 4), 3)
  }, weatherCFOREST)
            Importance
Pressure3pm     0.0212
Sunshine        0.0163
Cloud3pm        0.0150
```

12.7 Summary

A random forest is an ensemble (i.e., a collection) of unpruned decision trees. Random forests are often used when we have very large training datasets and a very large number of input variables (hundreds or even thousands of input variables). A random forest model is typically made up of tens or hundreds of decision trees.

The generalisation error rate from random forests tends to compare favourably with boosting approaches (see Chapter 13), yet the approach tends to be more robust to noise in the training dataset and so tends to be a very stable model builder, as it does not suffer the sensitivity to noise in a dataset that single-decision-tree induction does. The general observation is that the random forest model builder is very competitive with nonlinear classifiers such as artificial neural nets and support vector machines. However, performance is often dataset-dependent, so it remains useful to try a suite of approaches.

Each decision tree is built from a random subset of the training dataset, using what is called replacement (thus it is doing what is known as bagging) in performing this sampling. That is, some observations will be included more than once in the sample, and others won't appear at all. Generally, about two-thirds of the observations will be included in the subset of the training dataset and one-third will be left out.

In building each decision tree model based on a different random subset of the training dataset a random subset of the available variables is used to choose how best to partition the dataset at each node. Each decision tree is built to its maximum size, with no pruning performed. Together, the resulting decision tree models of the forest represent the final ensemble model, where each decision tree votes for the result, and the majority wins. (For a regression model, the result is the average value over the ensemble of regression trees.)

In building the random forest model, we have options to choose the number of trees, the training dataset sample size for building each decision tree, and the number of variables to randomly select when considering how to partition the training dataset at each node. The random forest model builder can also report on the input variables that are actually most important in determining the values of the output variable.

By building each decision tree to its maximal depth (i.e., by not pruning the decision tree), we can end up with a model that is less biased. The randomness introduced by the random forest model builder in selecting

the dataset and the variable delivers considerable robustness to noise, outliers, and overfitting when compared with a single tree classifier.

The randomness also delivers substantial computational efficiencies. In building a single decision tree, the model builder may select a random subset of the training dataset. Also, at each node in the process of building the decision tree, only a small fraction of all of the available variables are considered when determining how best to partition the dataset. This substantially reduces the computational requirement.

In summary, a random forest model is a good choice for model building for a number of reasons. First, just like decision trees, very little, if any, preprocessing of the data needs to be performed. The data does not need to be normalised and the approach is resilient to outliers. Second, if we have many input variables, we generally do not need to do any variable selection before we begin model building. The random forest model builder is able to target the most useful variables. Third, because many trees are built and there are two levels of randomness, and each tree is effectively an independent model, the model builder tends not to overfit to the training dataset. A key factor about a random forest being a collection of many decision trees is that each decision tree is not influenced by the other decision trees when constructed.

12.8 Command Summary

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

`cforest()`	function	Build a conditional random forest.
`ctree()`	function	Build a conditional inference tree.
`evalq()`	function	Access environment for storing data.
`na.roughfix()`	function	Impute missing values.
party	package	Conditional inference trees.
`randomForest()`	function	Implementation of random forests.
randomForest	package	Build ensemble of decision trees.
`str()`	function	Show the structure of an object.
weather	dataset	Sample dataset from **rattle**.