Chapter 14
Classification Trees and Rule-Based Models

Classification trees fall within the family of tree-based models and, similar to regression trees, consist of nested `if-then` statements. For the familiar two-class problem shown in the last two chapters, a simple classification tree might be

```
if Predictor B >= 0.197 then
|   if Predictor A >= 0.13 then Class = 1
|   else Class = 2
else Class = 2
```

In this case, two-dimensional predictor space is cut into three regions (or terminal nodes) and, within each region, the outcome categorized into either "Class 1" or "Class 2." Figure 14.1 presents the tree in the predictor space. Just like in the regression setting, the nested `if-then` statements could be collapsed into rules such as

```
if Predictor A >= 0.13 and Predictor B >= 0.197 then Class = 1
if Predictor A >= 0.13 and Predictor B <  0.197 then Class = 2
if Predictor A <  0.13 then Class = 2
```

Clearly, the structure of trees and rules is similar to the structure we saw in the regression setting. And the benefits and weaknesses of trees in the classification setting are likewise similar: they can be highly interpretable, can handle many types of predictors as well as missing data, but suffer from model instability and may not produce optimal predictive performance. The process for finding the optimal splits and rules, however, is slightly different due to a change in the optimization criteria, which will be described below.

Random forests, boosting, and other ensemble methodologies using classification trees or rules are likewise extended to this setting and are discussed in Sects. 14.3 through 14.6.

M. Kuhn and K. Johnson, *Applied Predictive Modeling*,
DOI 10.1007/978-1-4614-6849-3_14,

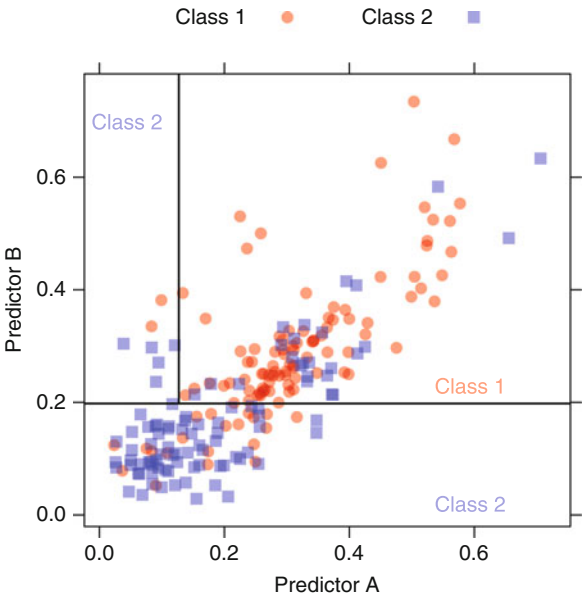

Fig. 14.1: An example of the predicted classes within regions defined by a tree-based model

14.1 Basic Classification Trees

As with regression trees, the aim of classification trees is to partition the data into smaller, more homogeneous groups. Homogeneity in this context means that the nodes of the split are more pure (i.e., contain a larger proportion of one class in each node). A simple way to define purity in classification is by maximizing accuracy or equivalently by minimizing misclassification error. Accuracy as a measure of purity, however, is a bit misleading since the measure's focus is on partitioning the data in a way that minimizes misclassification rather than a focus on partitioning the data in a way that place samples primarily in one class.

Two alternative measures, the Gini index (Breiman et al. 1984) and cross entropy, which is also referred to as deviance or information (defined later in this section), shift the focus from accuracy to purity. For the two-class problem, the Gini index for a given node is defined as

$$p_1 (1 - p_1) + p_2 (1 - p_2), \tag{14.1}$$

where p_1 and p_2 are the Class 1 and Class 2 probabilities, respectively. Since this is a two-class problem $p_1 + p_2 = 1$, and therefore Eq. 14.1 can equivalently

be written as $2p_1p_2$. It is easy to see that the Gini index is minimized when either of the class probabilities is driven towards zero, meaning that the node is pure with respect to one of the classes. Conversely, the Gini index is maximized when $p_1 = p_2$, the case in which the node is least pure.

When working with a continuous predictor and a categorical response, the process for finding the optimal split point is similar to the process we saw in Sect. 8.1. First, the samples are sorted based on their predictor values. The split points are then the midpoints between each unique predictor value. If the response is binary, then this process generates a 2×2 contingency table at each split point. This table can be generally represented as

	Class 1	Class 2	
$>$ split	n_{11}	n_{12}	n_{+1}
$\leq$ split	n_{21}	n_{22}	n_{+2}
	n_{1+}	n_{2+}	n

The Gini index prior to the split would be

$$Gini(\text{prior to split}) = 2\left(\frac{n_{1+}}{n}\right)\left(\frac{n_{2+}}{n}\right).$$

And the Gini index can be calculated after the split within each of the new nodes with values $2\left(\frac{n_{11}}{n_{+1}}\right)\left(\frac{n_{12}}{n_{+1}}\right)$ and $2\left(\frac{n_{21}}{n_{+2}}\right)\left(\frac{n_{22}}{n_{+2}}\right)$ for greater than and less than or equal to the split, respectively. These values are combined using the proportion of samples in each part of the split as weights with $\left(\frac{n_{+1}}{n}\right)$ and $\left(\frac{n_{+2}}{n}\right)$ representing the respective weights for greater than and less than or equal to the split. After some simplification, the Gini index to evaluate the split would be:

$$Gini(\text{after split}) = 2\left[\left(\frac{n_{11}}{n}\right)\left(\frac{n_{12}}{n_{+1}}\right) + \left(\frac{n_{21}}{n}\right)\left(\frac{n_{22}}{n_{+2}}\right)\right].$$

Now consider the simple example presented in Fig. 14.1, where the contingency table for the Predictor B split is as follows:

	Class 1	Class 2	
$B > 0.197$	91	30	121
$B \leq 0.197$	20	67	87

The Gini index for the samples in the $B > 0.197$ split would be 0.373 and for the samples with $B \leq 0.197$ would be 0.354. To determine if this is a good overall split, these values must be combined which is done by weighting each purity value by the proportion of samples in the node relative to the total number of samples in the parent node. In this case, the weight for the $B > 0.197$ split would be 0.582 and 0.418 when $B \leq 0.197$. The overall Gini index measure for this split would then be $(0.582)(0.373) + (0.418)(0.354) = 0.365$.

Here we have evaluated just one possible split point; partitioning algorithms, however, evaluate nearly all split points[1] and select the split point value that minimizes the purity criterion. The splitting process continues within each newly created partition, therefore increasing the depth of the tree, until the stopping criteria is met (such as the minimum number of samples in a node or the maximum tree depth).

Trees that are constructed to have the maximum depth are notorious for over-fitting the training data. A more generalizable tree is one that is a pruned version of the initial tree and can be determined by cost-complexity tuning, in which the purity criterion is penalized by a factor of the total number of terminal nodes in the tree. The cost-complexity factor is called the complexity parameter and can be incorporated into the tuning process so that an optimal value can be estimated. More details about this process can be found in Sect. 8.1.

After the tree has been pruned, it can be used for prediction. In classification, each terminal node produces a vector of class probabilities based on the training set which is then used as the prediction for a new sample. In the simple example above, if a new sample has a value of Predictor $B = 0.10$, then predicted class probability vector would be $(0.23, 0.77)$ for Class 1 and Class 2, respectively.

Similar to regression trees, classification trees can handle missing data. In tree construction, only samples with non-missing information are considered for creating the split. In prediction, surrogate splits can be used in place of the split for which there are missing data. Likewise, variable importance can be computed for classification trees by assessing the overall improvement in the optimization criteria for each predictor. See Sect. 8.1 for the parallel explanation in regression.

When the predictor is continuous, the partitioning process for determining the optimal split point is straightforward. When the predictor is categorical, the process can take a couple of equally justifiable paths, one of which differs from the traditional statistical modeling approach. For example, consider a logistic regression model which estimates slopes and intercepts associated with the predictors. For categorical predictors, a set of binary dummy variables (Sect. 3.6) is created that decomposes the categories to independent bits of information. Each of these dummy variables is then included separately in the model. Tree models can also bin categorical predictors. Evaluating purity for each of these new predictors is then simple, since each predictor has exactly one split point.

For tree models, the splitting procedure may be able to make more dynamic splits of the data, such as groups of two or more categories on either side of the split. However, to do this, the algorithm must treat the categorical predictors as an ordered set of bits. Therefore, when fitting trees and rule-based models, the practitioner must make a choice regarding the treatment of categorical predictor data:

[1] See Breiman (1996c) for a discussion of the technical nuances of splitting algorithms.

1. Each categorical predictor can be entered into the model as a single entity so that the model decides how to group or split the values. In the text, this will be referred to as using *grouped categories*.
2. Categorical predictors are first decomposed into binary dummy variables. In this way, the resulting dummy variables are considered independently, forcing binary splits for the categories. In effect, splitting on a binary dummy variable prior to modeling imposes a "one-versus-all" split of the categories. This approach will be labelled as using *independent categories*.

Which approach is more appropriate depends on the data and the model. For example, if a subset of the categories are highly predictive of the outcome, the first approach is probably best. However, as we will see later, this choice can have a significant effect on the complexity of the model and, as a consequence, the performance. In the following sections, models will be created using *both* approaches described above to assess which approach is model advantageous. A summary of the differences in the two approaches are summarized in Fig. 14.14 on p. 402 of this chapter.

To illustrate the partitioning process for a categorical predictor, consider the CART model of the grant data illustrated in Fig. 14.3. The first split for these data is on contract value band, which has 17 possible categories, and places values `I`, `J`, `P`, and `Unknown` into one partition and the remaining categories in the other. From a combinatorial standpoint, as the number of possible categories increase, the number of possible category orderings increases factorially. The algorithmic approach must therefore take a rational but greedy path to ordering the categories prior to determining the optimal split. One approach is to order the categories based on the proportion of samples in a selected class. The top plot in Fig. 14.2 displays the probability of successful grant application within each contract value band, ordered from least successful to most successful. To calculate the Gini index, the split points are the divisions between each of the ordered categories, with the categories to the left placed into one group and the categories to the right placed into the other group. The results from these sequential partitions are presented in the bottom plot. Clearly, adding samples from the Unknown category to the samples from categories `P` and `J` greatly reduces the Gini index. While it is difficult to see from the figure, the minimum value occurs at the split point between categories `I` and `M`. Therefore, the algorithm chooses to place samples from contract value band `I`, `J`, `P`, and Unknown into one partition and the remaining samples into the other. Using only this split, the model would classify a new sample as unsuccessful if it had a contract value band of `I`, `J`, `P`, or `Unknown` and successful otherwise.

Continuing the tree building process treating the predictors as grouped categories and pruning via cost complexity produces the tree in Fig. 14.3. Because the predictors are encoded, it is difficult to interpret the tree without an in-depth knowledge of the data. However, it is still possible to use the tree structure to gain insight to the relevance of the predictors to the response. We can also see that grouped category variables such as sponsor

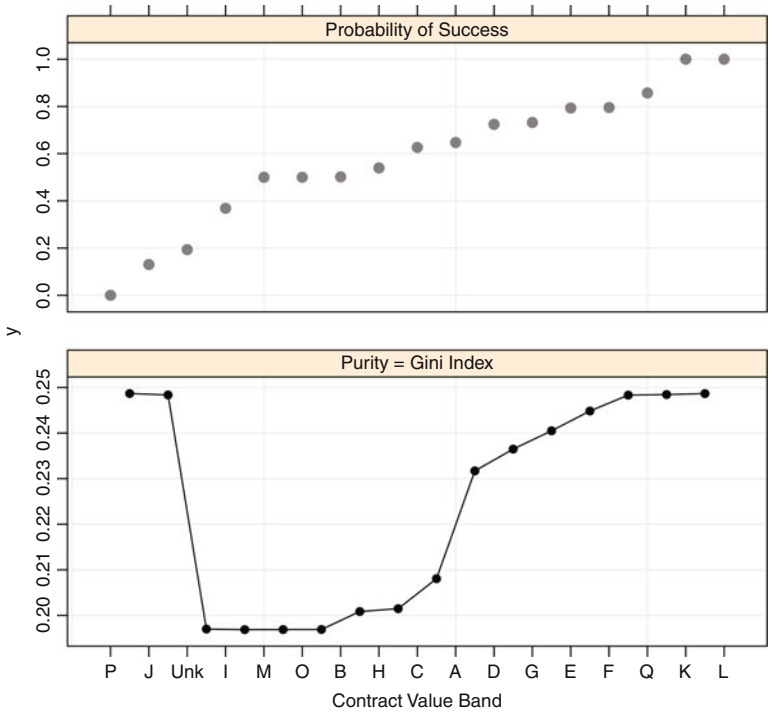

Fig. 14.2: *Top*: A scatter plot of the ordered probability of success (y-axis) for each contract value band. *Bottom*: The Gini index profile across each ordered split. The Gini index for the split points between categories `Unknown`, `I`, `M`, `O`, and `B` are nearly equivalent, with the minimum occurring between categories `I` and `M`

code, weekday, and month are relevant to the success of grant funding. The grouped categories model has an area under the ROC curve of 0.91 using 16 terminal nodes.

A CART model was also built using independent category predictors. Because this approach creates many more predictors, we would expect that the pruned tree would have more terminal nodes. Counter to intuition, the final pruned tree has 16 nodes and is illustrated in Fig. 14.4. This tree has an AUC of 0.912, and Fig. 14.5 compares its performance with the grouped category predictors. For classification trees using CART, there is no practical difference in predictive performance when using grouped categories or independent categories predictors for the grant data.

A comparison of Figs. 14.3 and 14.4 highlights a few interesting similarities and differences between how a tree model handles grouped category versus independent predictors. First, notice that the upper levels of the trees are generally the same with each selecting contract value band, sponsor code, and

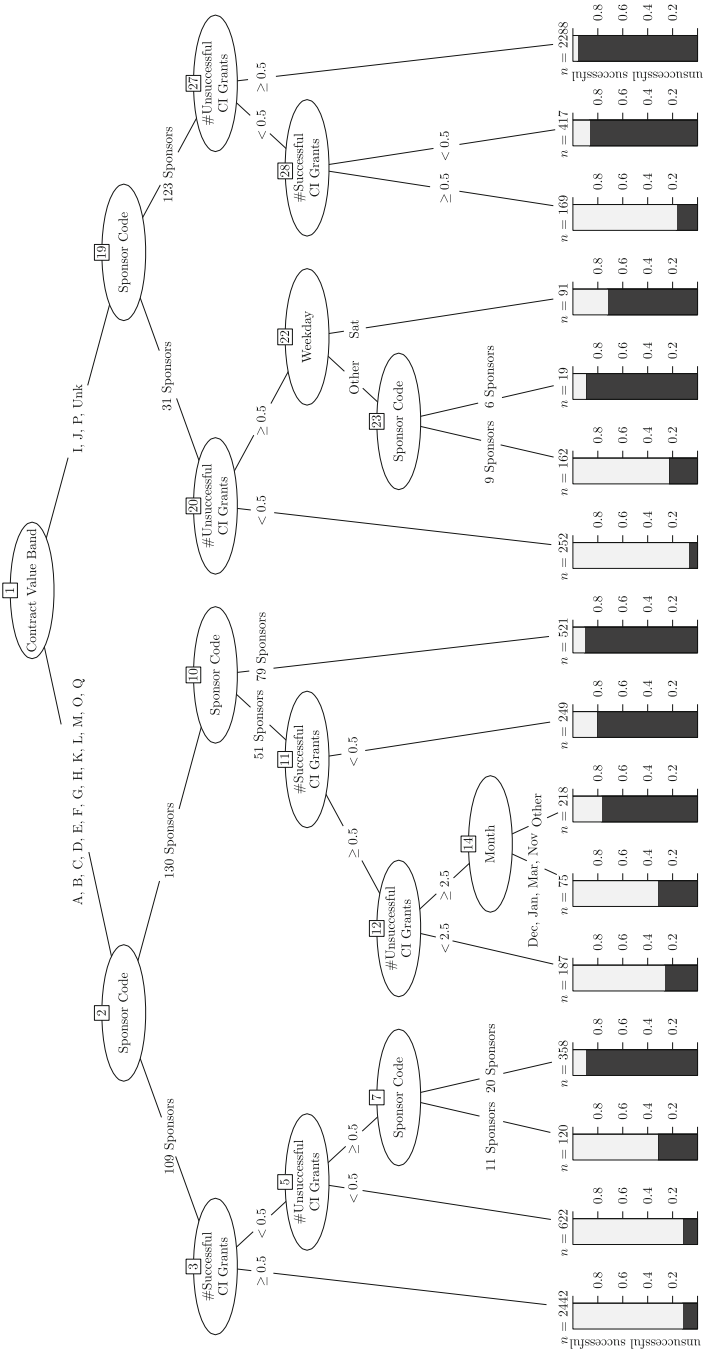

Fig. 14.3: The final CART model for the grant data using grouped category predictors

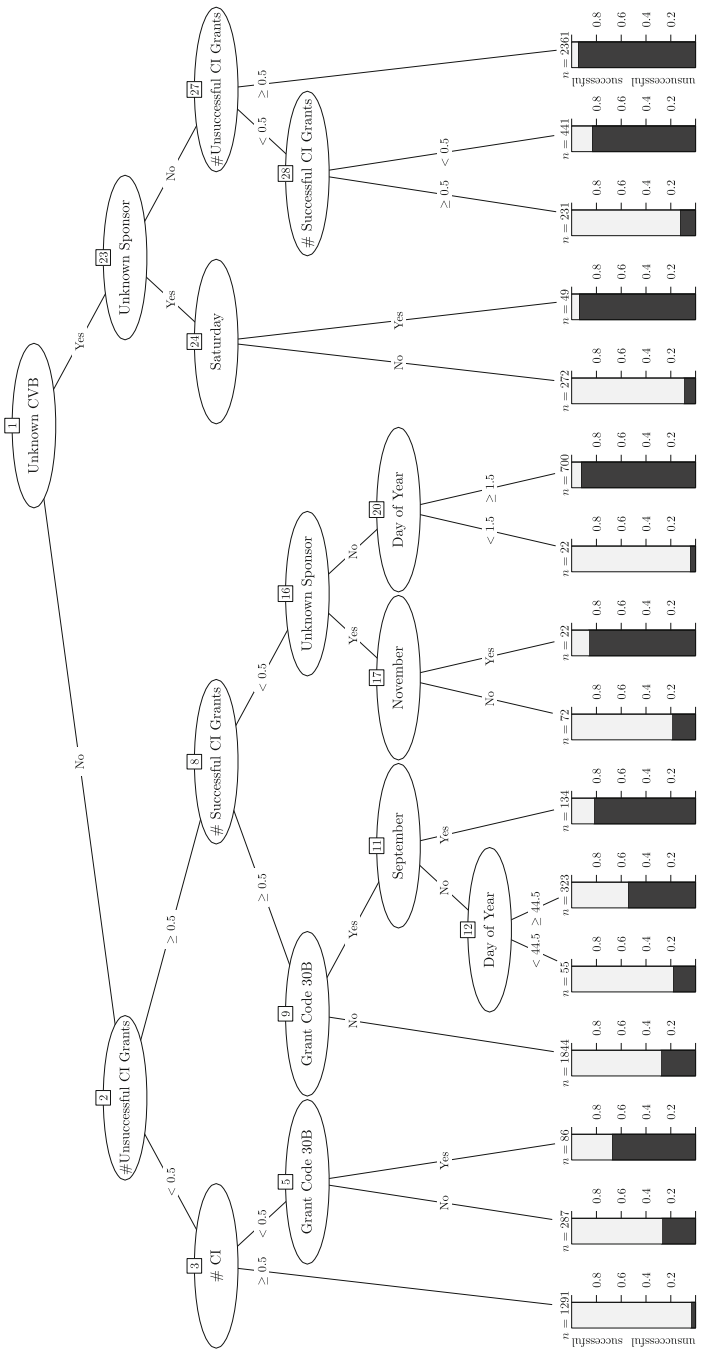

Fig. 14.4: The final CART model for the grant data using independent category predictors

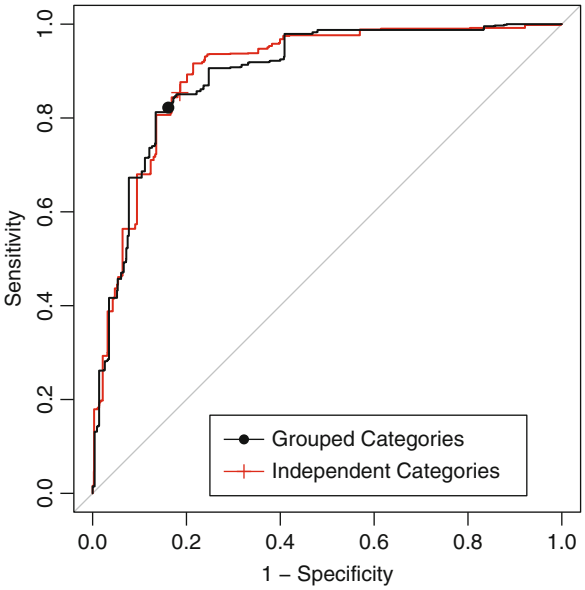

Fig. 14.5: The CART ROC curves for the holdout data. When using grouped categories, the area under the curve was 0.89. With independent categories, the AUC was also 0.89

number of unsuccessful and successful grants by chief investigators within the first four levels. Although the trees are identifying similarly important information, the independent category tree is much easier to interpret than the grouped category tree. For example, while the contract value band predictor is chosen as the first split in each tree, the independent category tree indicates that the value of Unknown is most critical for creating subsequent nodes that are more pure. Without producing a purity plot of the ordered categories, the importance of the Unknown band is masked within the grouping of bands I, J, P, and Unknown for the grouped category tree. Similar contrasts can be made with predictors of Month and Weekday, where the independent category tree provides further insight into the importance of specific months and weekdays. In the case of trees, therefore, creating independent category predictors may provide valuable interpretation about the relationship between predictors and the response that is not readily available when treating predictors as grouped categories.

Another approach for classification trees is the C4.5 model (Quinlan 1993b). Here, the splitting criteria is based on information theory (Wallace 2005; Cover and Thomas 2006). Suppose we want to communicate some piece of information, such as the probability distribution of the classes in the terminal node of a tree, in a series of messages. If the probability distribution is

extremely unbalanced, there is a high likelihood of the sample belonging to the majority class, thus less uncertainty when guessing. However, if the class probabilities in the node were even, there is high uncertainty of a sample's true class. If we were trying to communicate the content of the probability distribution in a series of messages, on average, more information needs to be conveyed when there is a high degree of uncertainty in the message. Shannon (1948) and others developed a theory for the communication of information. The quantity that they call the *information statistic* represents the average number of bits needed to communicate in a message.

In our context, suppose there are $C = 2$ classes and the probability of the first class is p. The formal definition of the information statistic is

$$info = -[p\, log_2 p + (1-p)\, log_2(1-p)].$$

When $p = 0$, it is customary to have $0\, log_2(0) = 0$. As previously mentioned, the units are called *bits*.

For the two class data shown in Fig. 14.1, the classes are almost even. If p is the proportion of samples in the first class, then $p = 0.53$. From this, the average number of bits of information to guess the true class (i.e., the information) would be 0.997. Now consider an unbalanced situation where fewer of the samples were in class 1 ($p = 0.10$). In this case, the information would be 0.46 bits, which is smaller because the class imbalance makes it easier to randomly guess the true class.[2] This metric has been previously discussed twice: as an objective function for neural networks (Eq. 13.3) and logistic regression (in Eq. 12.1 with a single data point).

How does this relate to determining splits? Using the general contingency table notation from above, the total information content of the data prior to splitting would be

$$info(\text{prior to split}) = -\left[\frac{n_{1+}}{n} \times log2\left(\frac{n_{1+}}{n}\right)\right] - \left[\frac{n_{2+}}{n} \times log2\left(\frac{n_{2+}}{n}\right)\right].$$

Again, when $n_{1+} = 0$ or $n_{2+} = 0$, it is traditional to set the terms inside the brackets to zero.

We can measure the improvement in the information criteria that would be induced by creating splits in a classification tree. The *information gain*[3] (or simply the *gain*) would be

$$gain(split) = info(\text{prior to split}) - info(\text{after split}).$$

[2] An alternate way to think of this is in terms of *entropy*, a measure of uncertainty. When the classes are balanced 50/50, we have no real ability to guess the outcome: it is as uncertain as possible. However, if ten samples were in class 1, we would have less uncertainty since it is more likely that a random data point would be in class 1.

[3] Also known as the *mutual information* statistic. This statistic is discussed again in Chap. 18.

Splits with larger information gains are more attractive than those with smaller gains.

For the binary split shown in the table above, the information after the split would be the sum of the information values from each of the resulting partitions. For example, the information for the data with values greater than the split value is

$$info(\text{greater}) = -\left[\frac{n_{11}}{n_{+1}} \times log2\left(\frac{n_{11}}{n_{+1}}\right)\right] - \left[\frac{n_{12}}{n_{+1}} \times log2\left(\frac{n_{12}}{n_{+1}}\right)\right].$$

The formula for the data on the other side of the split is analogous. The total information after the split is a weighted average of these values where the weights are related to the number of samples in the leaves of the split

$$info(\text{after split}) = \frac{n_{+1}}{n}\, info(\text{greater}) + \frac{n_{+2}}{n}\, info(\text{less than}).$$

Going back to the two class data, consider the predictor B split at a value of 0.197. The information when $B > 0.197$ is 0.808 and, on the other side of the split, the value is 0.778 when weighted by the proportion of samples on each side of the split, the total information is 0.795, a gain of $0.997 - 0.795 = 0.201$ Suppose, on the other hand, another split chosen that was completely non-informative, the information after the split would be the same as prior to the split, so the gain would be zero.

For continuous predictors, a tree could be constructed by searching for the predictor and single split that maximizes the information gain.[4] For these data, this gain is the largest when splitting predictor B at 0.197 and this is the split shown in Fig. 14.1. It also turns out that this split is also the best split for the Gini criterion used by CART.

There is one issue with this strategy. Since the predictors might have different numbers of possible values, the information gain criteria is biased against predictors that have a large number of possible outcomes (i.e., would favor categorical predictors with only a few distinct values over continuous predictors). This phenomenon is similar to the previously discussed bias for regression trees in Sect. 8.1. In this case, the bias is related to the ability of the algorithm to split the categorical predictors many ways (instead of a binary split on continuous predictors). The multi-way splits are likely to have larger gains. To correct for the bias, the *gain ratio* is used, which divides the gain by a measure of the amount of information in the split itself. Quinlan (1993b) shows additional examples of these calculations while Quinlan (1996b) describes refinements to this procedure for continuous predictors using the minimum description length (MDL) principle.

[4] By default, C4.5 uses simple binary split of continuous predictors. However, Quinlan (1993b) also describes a technique called *soft thresholding* that treats values near the split point differently. For brevity, this is not discussed further here.

When evaluating splits of categorical predictors, one strategy is to represent the predictor using multi-way splits such that there is a separate split for each category. When a predictor has a large number of possible values, this can lead to overly complex trees. For example, the sponsor code predictor in the grant data have 298 unique values. If this predictor were considered important, an initial 298-way split of the data would be created (prior to pruning). After the pruning process described below, some of these splits are likely to be combined and simplified.

Chapter 7 of Quinlan (1993b) describes a modified approach for creating multi-way splits that have the ability to group two or more categories. Prior to evaluating a categorical predictor as a split variable, the model first enumerates the gain ratio when the predictor is represented as:

- A multi-way split with as many splits as distinct values (i.e., the default approach where each category is a separate split).
- Multi-way splits for all possible combinations when two categories are grouped together and the others are split separately.

Based on the results of these representations of the predictor, a greedy algorithm is used to find the best categories to merge. As a result, there are many possible representations of the categorical predictor. Once the model constructs the final groupings, the gain ratio is calculated for this configuration. The ratio is compared to the other predictors when searching for the best split variable. This process is repeated each time the model conducts a search for a new split variable. This option is computationally expensive and may have a minimal impact on the tree if the categorical predictors have only a few possible levels. Unfortunately, this option is not available in the implementation of C4.5 that is currently available (in the Weka software suite under the name J48). The effect of this option on the data cannot be directly demonstrated here, but will be shown later when describing C5.0 (the descendent of C4.5). Since this can have a profound impact on the model, we will label this version of C4.5 as J48 to differentiate the versions.

When constructing trees with training sets containing missing predictor values, C4.5 makes several adjustments to the training process:

- When calculating the information gain, the information statistics are calculated using the non-missing data then scaled by the fraction of non-missing data at the split.
- Recall that C4.5 deals with selection bias by adjusting the gain statistic by the information value for the predictor. When the predictor contains missing values, the number of branches is increased by one; missing data are treated as an "extra" category or value of the predictor.
- Finally, when the class distribution is determined for the resulting splits, missing predictor values contribute *fractionally* to each class. The fractional contribution of the data points are based on the class distribution of the non-missing values. For example, suppose 11 samples are being split and one value was missing. If three samples are Class #1 and the rest are

Class #2, the missing value would contribute 0.30 to Class #1 and 0.70 to Class #2 (on both sides of the split).

Because of this accounting, the class frequency distribution in each node may not contain whole numbers. Also, the number of errors in the terminal node can be fractional.

Like CART, C4.5 builds a large tree that is likely to over-fit the data then prunes the tree back with two different strategies:

- Simple elimination of a sub-tree.
- *Raising* a sub-tree so that it replaces a node further up the tree.

Whereas CART uses cost complexity pruning, *pessimistic pruning* evaluates whether the tree should be simplified. Consider the case where a sub-tree is a candidate for removal. Pessimistic pruning estimates the number of errors with and without the sub-tree. However, it is well-known that the apparent error rate is extremely optimistic. To counteract this, pessimistic pruning calculates an upper confidence bound on the number of errors—this is the *pessimistic estimate* of the number of errors. This is computed with and without the sub-tree. If the estimated number of errors without the sub-tree is lower than the tree that includes it, the sub-tree is pruned from the model.

When determining the estimated error rate, C4.5 uses a default confidence level for the interval of 0.25 (called the *confidence factor*). This can be considered a tuning parameter for the model, as increasing the confidence factor leads larger trees. While intuitive, this approach stands on shaky statistical grounds, Quinlan (1993b) acknowledges this, saying that the approach

> "does violence to statistical notions of sampling and confidence limits, so the reasoning should be taken with a grain of salt."

That said, this technique can be very effective and is more computationally efficient than using cross-validation to determine the appropriate size of the tree.

Once the tree has been grown and pruned, a new sample is classified by moving down the appropriate path until it reaches the terminal node. Here, the majority class for the training set data falling into the terminal node is used to predict a new sample. A *confidence value*, similar to a class probability, can also be calculated on the basis of the class frequencies associated with the terminal nodes. Quinlan (1993b) describes how upper and lower ranges for the confidence factors can be derived from calculations similar to the pessimistic pruning algorithm described above.

When *predicting* a sample with one or more missing values, the sample is again treated fractionally. When a split is encountered for a variable that is missing in the data, each possible path down the tree is determined. Ordinarily, the predicted class would be based on the class with the largest frequency from a single terminal node. Since the missing value could have possibly landed in more than one terminal node, each class receives a weighted vote to determine the final predicted class. The class weights for all the relevant

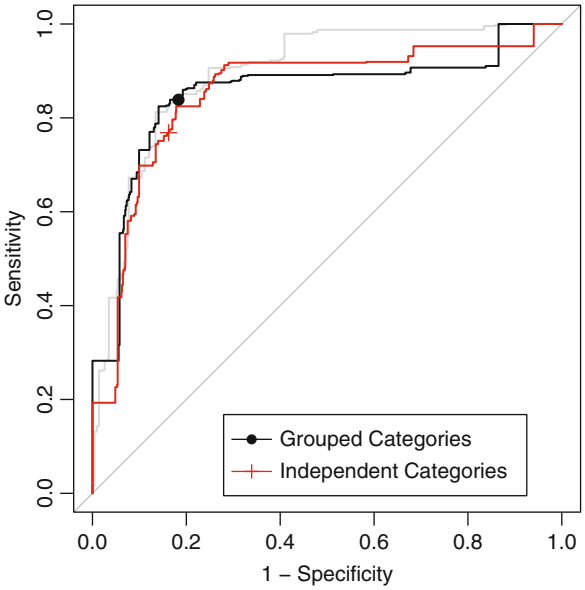

Fig. 14.6: The J48 ROC curves for the holdout data using two different approaches for handling categorical predictors. The symbols (*filled circle* and *plus*) represent the 50 % probability cutoff. The areas under the curves were 0.835 when using grouped categories and 0.842 when using independent categories. The *grey line* corresponds to the previous CART model

terminal nodes are aggregated and the class associated with the largest total weight is used to predict the sample. In this way, each terminal node with possible associations with the sample contributes to the overall prediction.

J48 trees were created for the grant application data. Although the confidence factor could be treated as a tuning parameter, our experience is that the default value (0.25) works well. Two models were fit using the two different approaches for representing the categorical predictors. Based on the prior discussion, there is the expectation that treating the categories as a cohesive set will results in a much larger tree than one using independent categories. This is exactly the case for these data. Grouping the categories resulted in a pruned tree with 2,918 terminal nodes. This was primarily due to a large number of splits using the sponsor code; 2,384 splits out of 2,918 (82 %) involve this predictor. When using independent categories, the tree was much smaller (821 terminal nodes).

The area under the ROC curve for the large model was 0.835, compared to 0.842 when using independent categories. Figure 14.6 shows the two ROC curves and the points on each curve corresponding to the default 50 % probability cutoff. From this, it is clear that the specificities are about the same for each approach (81.7 % for the larger model vs. 83.8 %), but there is a

significant difference in the sensitivity of the models; the more complex model resulted in a sensitivity of 83.9 % while the independent category model had relatively poor ability to predict successful grants (with a sensitivity of 76.8 %). However, these statistics are based on the nominal 50 % cutoff for success. The curves overlap considerably and alternate cutoffs would produce almost identical results (see Sect. 16.4).

While CART and C4.5 classification trees are the most widely used, there has been extensive research in this area and many other proposals for tree-based models. For example, as discussed in the section on regression trees, conditional inference trees (Hothorn et al. 2006) avoid selection bias during splitting. Also, several techniques exist (Frank et al. 1998; Loh 2002; Chan and Loh 2004; Zeileis et al. 2008) that use more complex models in the terminal nodes, similar to M5 and Cubist. Other types of splits can be employed. For example, Breiman et al. (1984) introduced the idea of splitting on a linear combination of the predictors. These *oblique trees* may be beneficial when the classes are linearly separable, which traditional splits have a difficult time approximating. Menze et al. (2011) discusses tree ensemble models with oblique trees.

14.2 Rule-Based Models

As previously discussed, rule-based models consist of one or more independent conditional statements. Unlike trees, a sample may be predicted from a set of rules. Rules have a long history as classifiers and this section will discuss approaches for creating classification rules.

C4.5Rules

There are several different philosophies and algorithms for creating rule-based models from classification trees. Some of the first were described by Quinlan (1987) and Quinlan (1993b). This model, called C4.5Rules, builds on the C4.5 tree methodology described in the last section. To start, an unpruned tree is created, then each path through the tree is collapsed into an individual rule.

Given this initial set, each rule is evaluated individually to assess whether it can be generalized by eliminating terms in the conditional statement. The pruning process here is similar to the one used to prune C4.5 trees. For a rule, the model first calculates a baseline pessimistic error rate, then removes each condition in the rule in isolation. Once a condition is removed, the pessimistic error rate is recomputed. If any error rate is smaller than the baseline, the condition associated with the smallest error rate is removed. The process is repeated until all conditions are above the baseline rate or all conditions are

removed. In the latter case, the rule is completely pruned from the model. The table below shows the pruning process with a five condition rule for the grant data:

Condition	Pessimistic error rate		
	Pass 1	Pass 2	Pass 3
Baseline	*14.9*	*5.8*	*5.2*
First day of year	12.9	**5.2**	
Zero unsuccessful grants (CI)	77.3	53.5	50.7
Number of CI	42.0	21.6	19.7
Number of SCI	18.0	8.1	7.3
Zero successful grants (CI)	**5.8**		

On the first pass, removing the condition associated with zero successful grants by a chief investigator has the least impact on the error rate, so this condition is deleted from the rule. Three passes of pruning were needed until none of the error rates were below the baseline rate. Also, note that the pessimistic error rate decreases with each iteration. Finally, the condition related to zero unsuccessful grants for a chief investigator appears to have the most importance to the rule since the error rate is the largest when the condition is removed from the rule.

After the conditions have been pruned *within* each rule, the set of rules associated with each class are processed separately to reduce and order the rules. First, redundant or ineffective rules are removed using the MDL principle [see Quinlan and Rivest (1989) and Chap. 5 of Quinlan (1993b)]. An MDL metric is created that encapsulates a ruleset's performance and complexity—for two rulesets with equivalent performance, the simpler collection of rules is favored by the metric. Within each class, an initial group of groups is assembled such that every training set sample is covered by at least one rule. These are combined into the initial ruleset. Starting with this set, search methods (such as greedy hill climbing or simulated annealing) are used to add and remove rules until no further improvements can be made on the ruleset. The second major operation within a class is to order the rules from most to least accurate.

Once the rulesets within each class have been finalized, the classes are ordered based on accuracy and a default class is chosen for samples that have no relevant rules. When predicting a new sample, each rule is evaluated in order until one is satisfied. The predicted class corresponds to the class for the first active rule.

```
repeat
    Create a pruned classification tree
    Determine the path through the tree with the largest coverage
    Add this path as a rule to the rule set
    Remove the training set samples covered by the rule
until all training set samples are covered by a rule
```

Algorithm 14.1: The PART algorithm for constructing rule-based models (Frank and Witten 1998)

PART

C4.5Rules follows the philosophy that the initial set of candidate rules are developed simultaneously then post-processed into an improved model. Alternatively, rules can be created incrementally. In this way, a new rule can adapt to the previous set of rules and may more effectively capture important trends in the data.

Frank and Witten (1998) describe another rule model called PART shown in Algorithm 14.1. Here, a pruned C4.5 tree is created from the data and the path through the tree that covers the most samples is retained as a rule. The samples covered by the rule are discarded from the data set and the process is repeated until all samples are covered by at least one rule. Although the model uses trees to create the rules, each rule is created separately and has more potential freedom to adapt to the data.

The PART model for the grant data slightly favored the grouped category model. For this model, the results do not show an improvement above and beyond the previous models: the estimated sensitivity was 77.9 %, the specificity was 80.2 %, and the area under the ROC curve (not shown) was 0.809. The model contained 360 rules. Of these, 181 classify grants as successful while the other 179 classify grants as unsuccessful. Here, the five most prolific predictors were sponsor code (332 rules), contract value band (30 rules), the number of unsuccessful grants by chief investigators (27 rules), the number of successful grants by chief investigators (26 rules), and the number of chief investigators (23 rules).

14.3 Bagged Trees

Bagging for classification is a simple modification to bagging for regression (Sect. 8.4). Specifically, the regression tree in Algorithm 8.1 is replaced with an unpruned classification tree for modeling C classes. Like the regression

Table 14.1: The 2008 holdout set confusion matrix for the random forest model

	Observed class	
	Successful	Unsuccessful
Successful	491	144
Unsuccessful	79	843

This model had an overall accuracy of 85.7 %, a sensitivity of 86.1 %, and a specificity of 85.4 %

setting, each model in the ensemble is used to predict the class of the new sample. Since each model has equal weight in the ensemble, each model can be thought of as casting a vote for the class it thinks the new sample belongs to. The total number of votes within each class are then divided by the total number of models in the ensemble (M) to produce a predicted probability vector for the sample. The new sample is then classified into the group that has the most votes, and therefore the highest probability.

For the grant data, bagging models were built using both strategies for categorical predictors. As discussed in the regression trees chapter, bagging performance often plateaus with about 50 trees, so 50 was selected as the number of trees for each of these models. Figure 14.7 illustrates the bagging ensemble performance using either independent or grouped categories. Both of these ROC curves are smoother than curves produced with classification trees or J48, which is an indication of bagging's ability to reduce variance via the ensemble. Additionally, both bagging models have better AUCs (0.92 for both) than either of the previous models. For these data, there seems to be no obvious difference in performance for bagging when using either independent or grouped categories; the ROC curves, sensitivities, and specificities are all nearly identical. The holdout set performance in Fig. 14.7 shows an improvement over the J48 results (Fig. 14.6).

Similar to the regression setting, variable importance measures can be calculated by aggregating variable importance values from the individual trees in the ensemble. Variable importance of the top 16 predictors for both the independent and grouped category bagged models set are presented in Fig. 14.15, and a comparison of these results is left to the reader in Exercise 14.1.

14.4 Random Forests

Random forests for classification requires a simple tweak to the random forest regression algorithm (Algorithm 8.2): a classification tree is used in place of

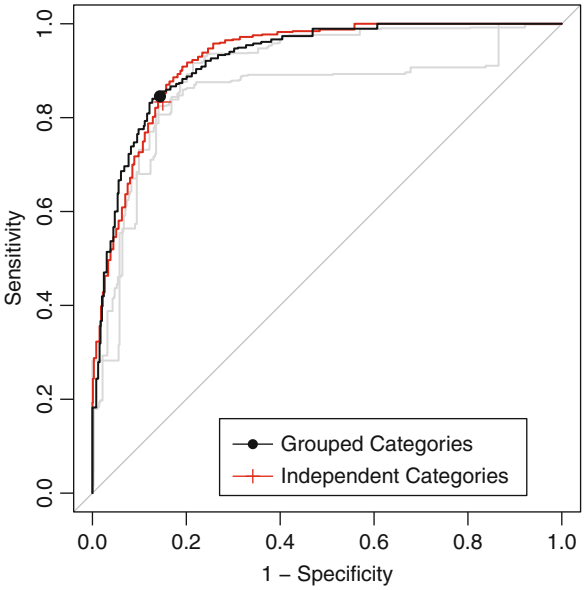

Fig. 14.7: The ROC curves for the bagged classification tree model. The area under the curves for both models was 0.92. The sensitivities and specificities were 82.98 and 85.71, respectively

a regression tree. As with bagging, each tree in the forest casts a vote for the classification of a new sample, and the proportion of votes in each class across the ensemble is the predicted probability vector.

While the type of tree changes in the algorithm, the tuning parameter of number of randomly selected predictors to choose from at each split is the same (denoted as m_{try}). As in regression, the idea behind randomly sampling predictors during training is to de-correlate the trees in the forest. For classification problems, Breiman (2001) recommends setting m_{try} to the square root of the number of predictors. To tune m_{try}, we recommend starting with five values that are somewhat evenly spaced across the range from 2 to P, where P is the number of predictors. We likewise recommend starting with an ensemble of 1,000 trees and increasing that number if performance is not yet close to a plateau.

For the most part, random forest for classification has very similar properties to the regression analog discussed previously, including:

- The model is relatively insensitive to values of m_{try}.
- As with most trees, the data pre-processing requirements are minimal.
- Out-of-bag measures of performance can be calculated, including accuracy, sensitivity, specificity, and confusion matrices.

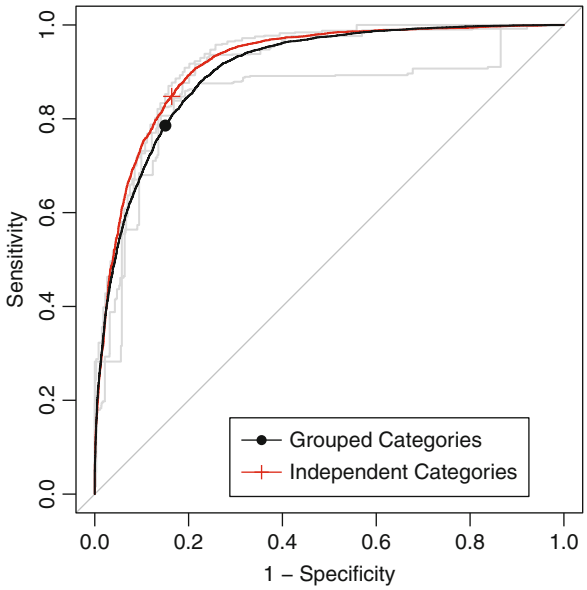

Fig. 14.8: The ROC curves for the random forest model. The area under the curve for independent categories was 0.92 and for the grouped category model the AUC was 0.9

One difference is the ability to weight classes differentially. This aspect of the model is discussed more in Chap. 16.

Random forest models were built on both independent and grouped category models. The tuning parameter, m_{try}, was evaluated at values ranging from 5 to 1,000. For independent categories, the optimal tuned value of m_{try} was 100, and for grouped categories the value was also 250. Figure 14.8 presents the results, and in this case the independent categories have a slightly higher AUC (0.92) than the grouped category approach (0.9). The binary predictor model also has better sensitivity (86.1 % vs. 84.7 %) but slightly worse specificity (85.4 % vs. 87.2 %).

For single trees, variable importance can be determined by aggregating the improvement in the optimization objective for each predictor. For random forests, the improvement criteria (default is typically the Gini index) is aggregated across the ensemble to generate an overall variable importance measure. Alternatively, predictors' impact on the ensemble can be calculated using a permutation approach (Breiman 2000) as discussed in Sect. 8.5. Variable importance values based on aggregated improvement have been computed for the grant data for both types of predictors and the most important

predictors are presented in Fig. 14.15. The interpretation is left to the reader in Exercise 14.1.

Conditional inference trees can also be used as the base learner for random forests. But current implementations of the methodology are computationally burdensome for problems that are the relative size of the grant data. A comparison of the performance of random forests using CART trees and conditional inference trees is explored in Exercise 14.3.

14.5 Boosting

Although we have already discussed boosting in the regression setting, the method was originally developed for classification problems (Valiant 1984; Kearns and Valiant 1989), in which many weak classifiers (e.g., a classifier that predicts marginally better than random) were combined into a strong classifier. There are many species of boosting algorithms, and here we discuss the major ones.

AdaBoost

In the early 1990s several boosting algorithms appeared (Schapire 1990; Freund 1995) to implement the original theory. Freund and Schapire (1996) finally provided the first practical implementation of boosting theory in their famous AdaBoost algorithm; an intuitive version is provided in Algorithm 14.2.

To summarize the algorithm, AdaBoost generates a sequence of weak classifiers, where at each iteration the algorithm finds the best classifier based on the current sample weights. Samples that are incorrectly classified in the kth iteration receive more weight in the $(k+1)$st iteration, while samples that are correctly classified receive less weight in the subsequent iteration. This means that samples that are difficult to classify receive increasingly larger weights until the algorithm identifies a model that correctly classifies these samples. Therefore, each iteration of the algorithm is required to learn a different aspect of the data, focusing on regions that contain difficult-to-classify samples. At each iteration, a *stage weight* is computed based on the error rate at that iteration. The nature of the stage weight described in Algorithm 14.2 implies that more accurate models have higher positive values and less accurate models have lower negative values.[5] The overall sequence of weighted classifiers is then combined into an ensemble and has a strong potential to classify better than any of the individual classifiers.

[5] Because a weak classifier is used, the stage values are often close to zero.

1 Let one class be represented with a value of +1 and the other with a value of -1
2 Let each sample have the same starting weight $(1/n)$
3 **for** $k = 1$ *to* K **do**
4 Fit a weak classifier using the weighted samples and compute the kth model's misclassification error (err_k)
5 Compute the kth stage value as $\ln\left(\left(1 - err_k\right)/err_k\right)$.
6 Update the sample weights giving more weight to incorrectly predicted samples and less weight to correctly predicted samples
7 **end**
8 Compute the boosted classifier's prediction for each sample by multiplying the kth stage value by the kth model prediction and adding these quantities across k. If this sum is positive, then classify the sample in the +1 class, otherwise the -1 class.

Algorithm 14.2: AdaBoost algorithm for two-class problems

Boosting can be applied to any classification technique, but classification trees are a popular method for boosting since these can be made into weak learners by restricting the tree depth to create trees with few splits (also known as stumps). Breiman (1998) gives an explanation for why classification trees work particularly well for boosting. Since classification trees are a low bias/high variance technique, the ensemble of trees helps to drive down variance, producing a result that has low bias and low variance. Working through the lens of the AdaBoost algorithm, Johnson and Rayens (2007) showed that low variance methods cannot be greatly improved through boosting. Therefore, boosting methods such as LDA or KNN will not show as much improvement as boosting methods such as neural networks (Freund and Schapire 1996) or naïve Bayes (Bauer and Kohavi 1999).

Stochastic Gradient Boosting

As mentioned in Sect. 8.6, Friedman et al. (2000) worked to provide statistical insight of the AdaBoost algorithm. For the classification problem, they showed that it could be interpreted as a forward stagewise additive model that minimizes an exponential loss function. This framework led to algorithmic generalizations such as Real AdaBoost, Gentle AdaBoost, and LogitBoost. Subsequently, these generalizations were put into a unifying framework called gradient boosting machines which was previously discussed in the regression trees chapter.

1 Initialized all predictions to the sample log-odds: $f_i^{(0)} = \log \frac{\widehat{p}}{1-\widehat{p}}$.
2 **for** *iteration* $j = 1 \ldots M$ **do**
3 Compute the residual (i.e. gradient) $z_i = y_i - \widehat{p}_i$
4 Randomly sample the training data
5 Train a tree model on the random subset using the residuals as the outcome
6 Compute the terminal node estimates of the Pearson residuals: $r_i = \frac{1/n \sum_i^n (y_i - \widehat{p}_i)}{1/n \sum_i^n \widehat{p}_i(1-\widehat{p}_i)}$
7 Update the current model using $f_i = f_i + \lambda f_i^{(j)}$
8 **end**

Algorithm 14.3: Simple gradient boosting for classification (2-class)

Akin to the regression setting, when trees are used as the base learner, basic gradient boosting has two tuning parameters: tree depth (or *interaction depth*) and number of iterations. One formulation of stochastic gradient boosting models an event probability, similar to what we saw in logistic regression, by

$$\widehat{p}_i = \frac{1}{1 + exp\left[-f(x)\right]},$$

where $f(x)$ is a model prediction in the range of $[-\infty, \infty]$. For example, an initial estimate of the model could be the sample log odds, $f_i^{(0)} = \log \frac{\widehat{p}}{1-\widehat{p}}$, where p is the sample proportion of one class from the training set.

Using the Bernoulli distribution, the algorithm for stochastic gradient boosting for two classes is shown in Algorithm 14.3.

The user can tailor the algorithm more specifically by selecting an appropriate loss function and corresponding gradient (Hastie et al. 2008). Shrinkage can be implemented in the final step of Algorithm 14.3. Furthermore, this algorithm can be placed into the stochastic gradient boosting framework by adding a random sampling scheme prior to the first step in the inner `For` loop. Details about this process can be found in Sect. 8.6.

For the grant data a tuning parameter grid was constructed where interaction depth ranged from 1 to 9, number of trees ranged from 100 to 2,000, and shrinkage ranged from 0.01 to 0.1. This grid was applied to constructing a boosting model where the categorical variables were treated as independent categories and separately as grouped categories. For the independent category model, the optimal area under the ROC curve was 0.94, with an interaction depth of 9, number of trees 1,300, and shrinkage 0.01. For the grouped category model, the optimal area under the ROC curve was 0.92, with an interaction depth of 7, number of trees 100, and shrinkage 0.01 (see

Fig. 14.9). In this case, the independent category model performs better than the grouped category model on the basis of ROC. However, the number of trees in each model was substantially different, which logically follows since the binary predictor set is larger than the grouped categories.

An examination of the tuning parameter profiles for the grouped category and independent category predictors displayed in Figs. 14.10 and 14.11 reveals some interesting contrasts. First, boosting independent category predictors has almost uniformly better predictive performance across tuning parameter settings relative to boosting grouped category predictors. This pattern is likely because only one value for many of the important grouped category predictors contains meaningful predictive information. Therefore, trees using the independent category predictors are more easily able to find that information quickly which then drives the boosting process. Within the grouped category predictors, increasing the shrinkage parameter almost uniformly degrades predictive performance across tree depth. These results imply that for the grouped category predictors, boosting obtains most of its predictive information from a moderately sized initial tree, which is evidenced by comparable AUCs between a single tree (0.89) and the optimal boosted tree (0.92).

Boosting independent category predictors shows that as the number of trees increases, model performance improves for low values of shrinkage and degrades for higher values of shrinkage. But, whether a lower or higher value of shrinkage is selected, each approach finds peak predictive performance at an ROC of approximately 0.94. This result implies, for these data, that boosting can find an optimal setting fairly quickly without the need for too much shrinkage.

Variable importance for boosting in the classification setting is calculated in a similar manner to the regression setting: within each tree in the ensemble, the improvement based on the splitting criteria for each predictor is aggregated. These importance values are then averaged across the entire boosting ensemble.

14.6 C5.0

C5.0 is a more advanced version of Quinlan's C4.5 classification model that has additional features, such as boosting and unequal costs for different types of errors. Like C4.5, it has tree- and rule-based versions and shares much of its core algorithms with its predecessor. Unlike C4.5 or Cubist, there is very little literature on the improvements and our description comes largely from evaluating the program source code, which was made available to the public in 2011.

The model has many features and options and our discussion is broken down into four separate areas: creating a single classification tree, the cor-

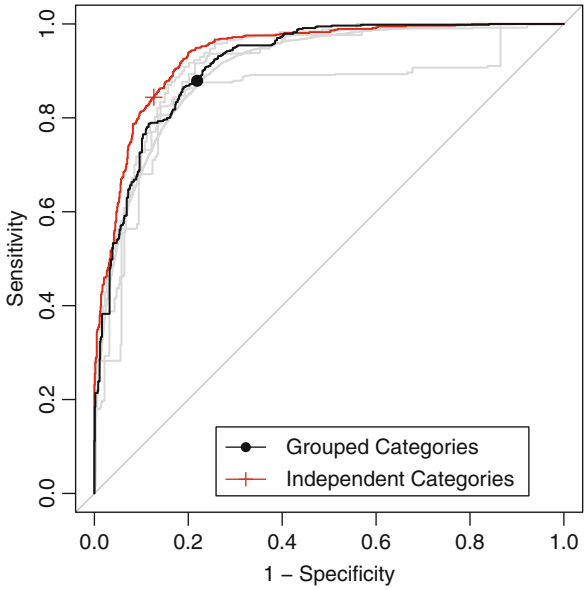

Fig. 14.9: The ROC curves for the boosted tree model. The area under the curve for independent categories was 0.936 and for the grouped category model the AUC was 0.916

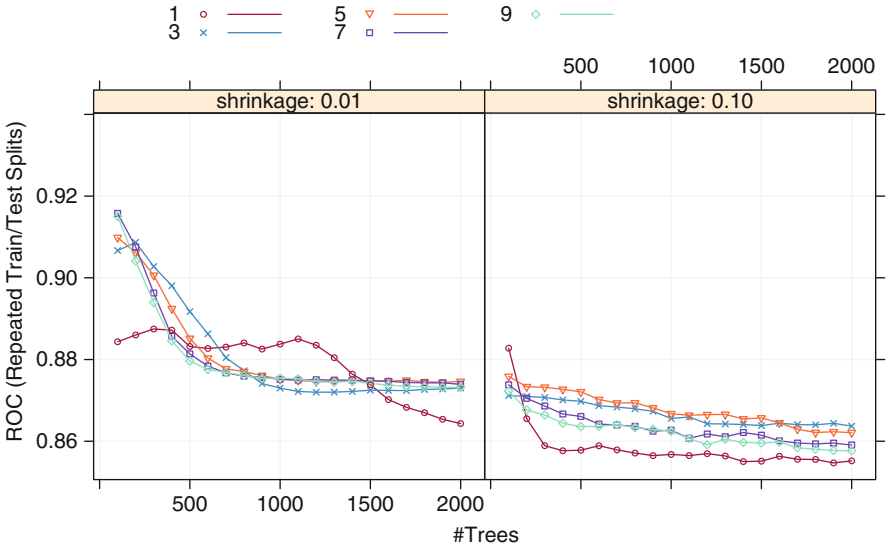

Fig. 14.10: Tuning parameter profiles for the boosted tree model using grouped categories

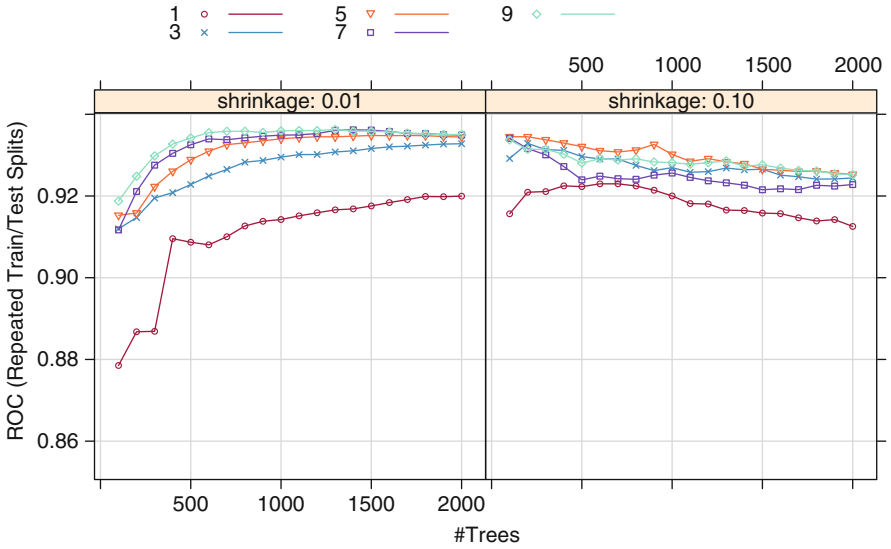

Fig. 14.11: Tuning parameter profiles for the boosted tree model using independent categories

responding rule-based model, C5.0's boosting procedure, and miscellaneous features of the algorithm (e.g., variable importance etc).

Classification Trees

C5.0 trees have several basic improvements that are likely to generate smaller trees. For example, the algorithm will combine nonoccurring conditions for splits with several categories. It also conducts a final global pruning procedure that attempts to remove the sub-trees with a cost-complexity approach. Here, sub-trees are removed until the error rate exceeds one standard error of the baseline rate (i.e., no pruning). Initial experimentation suggests that these additional procedures tend to create simpler trees than the previous algorithm.

The nominal C5.0 tree was fit to the grant data with the categorical predictors treated as cohesive sets. The tree had 86 terminal nodes and resulted in an area under the ROC curve of 0.685. The five most prolific predictors in the tree were contract value band (six splits), numeric day of the year (six splits), sponsor code (five splits), category code (four splits), and day of the week (four splits). Recall that the analogous J48 tree had many more terminal nodes (2,918), which was primarily due to how splits were made on categor-

ical variables with many possible values, such as the sponsor code. The C5.0 tree avoids this issue using the heuristic algorithm described in Sect. 14.1 that attempts to consolidate the categories into two or more smaller groups. If this option is turned off in C5.0, the tree is much larger (213 terminal nodes) due to the categorical predictors. However, the area under the ROC curve for the larger tree (0.685) is nearly the same as the smaller tree.

Neither C5.0 model approaches the size of the previously described J48 tree. For J48 and C5.0 (without grouping), categorical predictors with many values are used in more splits, and, at each split, they tend to result in more than two branches when the grouping option is not used.

Classification Rules

The process used for creating rules is similar to C4.5; an initial tree is grown, collapsed into rules, then the individual rules are simplified via pruning and a global procedure is used on the entire set to potentially reduce the number of constituent rules. The process for pruning conditions within a rule and simplifying the ruleset follows C4.5, but C5.0 does not order the rules. Instead, when predicting new samples, C5.0 uses *all* active rules, each of which votes for the most likely class. The votes for each class are weighted by the confidence values and the class associated with the highest vote is used. However, the predicted confidence value is the one associated with the most specific active rule. Recall that C4.5 sorts the rules, and uses the first active rule for prediction.

The grant data were analyzed with this algorithm. The rule-based model consists of 22 rules with an estimated area under the ROC curve of 0.675. The complexity of the model is much simpler than PART. When ordered by the confidence value of the rule, the top three rules to predict a successful grant are:

1. (First day of the year)
2. (The number of chief investigators > 0) and (the number of principal supervisors ≤ 0) and (the number of student chief investigators ≤ 0) and (the number of unsuccessful grants by chief investigators ≤ 0) and (SEO code $\neq 730106$) and (numeric day of the year ≤ 209)
3. (The number of external chief investigators ≤ 0) and (the number of chief investigators born around $1975 \leq 0$) and (the number of successful grants by chief investigators ≤ 0) and (numeric day of the year > 109) and (unknown category code) and (day of the week in Tues, Fri, Mon, Wed, Thurs)

Similarly, the top three rules for unsuccessful grants are:

1. (The number of unsuccessful grants by chief investigators > 0) and (numeric day of the year > 327) and (sponsor code in 2B, 4D, 24D, 60D,

90B, 32D, 176D, 7C, 173A, 269A) and (contract value band in Unk, J) and (CategoryCode in 10A, 30B, 30D, 30C)
2. (The number of chief investigators ≤ 1) and (the number of unsuccessful grants by chief investigators > 0) and (the number of B journal papers by chief investigators > 3) and (sponsor code = 4D) and (contract value band in B, Unk, J) and (Month in Nov, Dec, Feb, Mar, May, Jun)
3. (The number of chief investigators > 0) and (the number of chief investigators born around $1945 \leq 0$) and (the number of successful grants by chief investigators ≤ 0) and (numeric day of the year > 209) and (sponsor code in 21A, 60D, 172D, 53A, 103C, 150B, 175C, 93A, 207C, 294B)

There were 11 rules to predict successful grants and 11 for unsuccessful outcomes. The predictors involved in the most rules were the number of unsuccessful grants by chief investigators (11 rules), contract value band (9 rules), category code (8 rules), numeric day of the year (8 rules), and Month (5 rules).

C5.0 has other features for rule-based models. For example, the model can create *utility bands*. Here, the utility is measured as the increase in error that occurs when the rule is removed from the set. The rules are ordered with an iterative algorithm: the model removes the rule with the smallest utility and recomputes the utilities for the other rules. The sequence in which the rules are removed defines their importance. For example, the first rule that is removed is associated with the lowest utility and the last rule with the highest utility. The bands are groups of rules of roughly equal size based on the utility order (highest to smallest). The relationship between the cumulative error rate can be profiled as the groups of rules are added to the model.

Boosting

C5.0's boosting procedure is similar to the previously described AdaBoost algorithm in the basic sense: models are fit sequentially and each iteration adjusts the case weights based on the accuracy of a sample's prediction. There are, however, some notable differences. First, C5.0 attempts to create trees that are about the same size as the first tree by coercing the trees to have about the same number of terminal nodes per case as the initial tree. Previous boosting techniques treated the tree complexity as a tuning parameter. Secondly, the model combines the predictions from the constituent trees differently than AdaBoost. Each boosted model calculates the confidence values for each class as described above and a simple average of these values is calculated. The class with the largest confidence value is selected. Stage weights are not calculated during the model training process. Third, C5.0 conducts two sorts of "futility analysis" during model training. The model will automatically stop boosting if the model is very effective (i.e., the sum of the

weights for the misclassified samples is less than 0.10) or if it is highly ineffective (e.g., the average weight of incorrect samples is greater than 50 %). Also, after half of the requested boosting iterations, each sample is assessed to determine if a correct prediction is possible. If it is not, the case is dropped from further computations.

Finally, C5.0 uses a different weighting scheme during model training. First, some notation:

$$\begin{aligned} N &= \text{training set size} \\ N_- &= \text{number of incorrectly classified samples} \\ w_k &= \text{case weight for sample at the } k\text{th boosting iteration} \\ S_+ &= \text{sum of weights for correctly classified samples} \\ S_- &= \text{sum of weights for incorrectly classified samples} \end{aligned}$$

The algorithm begins by determining the midpoint between the sum of the weights for misclassified samples and half of the overall sum of the weights

$$midpoint = \frac{1}{2}\left[\frac{1}{2}(S_- + S_+) - S_-\right] = \frac{1}{4}(S_+ - S_-).$$

From this, the correctly classified samples are adjusted with the equation

$$w_k = w_{k-1} \times \frac{S_+ - midpoint}{S_+}$$

and the misclassified samples are updated using

$$w_k = w_{k-1} + \frac{midpoint}{N_-}.$$

This updating scheme gives a large positive jump in the weights when a sample is incorrectly predicted. When a sample is correctly predicted, the multiplicative nature of the equation makes the weights drop more slowly and with a decreasing rate as the sample continues to be correctly predicted. Figure 14.12 shows an example of the change in weights for a single sample over several boosting iterations.

Quinlan (1996a) describes several experiments with boosting and bagging tree-based models including several where boosting C4.5 resulted in a less effective model.

Other Aspects of the Model

C5.0 measures predictor importance by determining the percentage of training set samples that fall into all the terminal nodes after the split. For

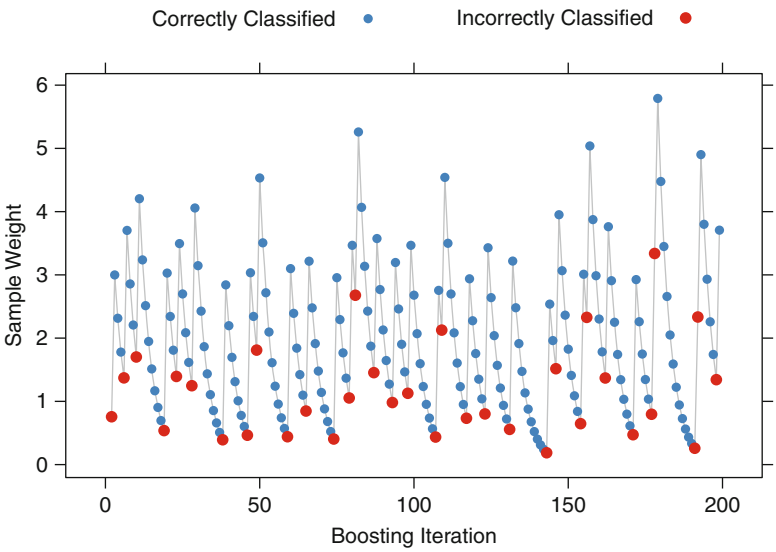

Fig. 14.12: An example of the sample weighting scheme using C5.0 when boosting

example, the predictor in the first split automatically has an importance measurement of 100 % since all samples are affected by this split. Other predictors may be used frequently in splits, but if the terminal nodes cover only a handful of training set samples, the importance scores may be close to zero. The same strategy is applied to rule-based models and boosted versions of the model.

C5.0 also has an option to *winnow* or remove predictors: an initial algorithm uncovers which predictors have a relationship with the outcome, and the final model is created from only the important predictors. To do this, the training set is randomly split in half and a tree is created for the purpose of evaluating the utility of the predictors (call this the "winnowing tree"). Two procedures characterize the importance of each predictor to the model:

1. Predictors are considered unimportant if they are not in any split in the winnowing tree.
2. The half of the training set samples not included to create the winnowing tree are used to estimate the error rate of the tree. The error rate is also estimated without each predictor and compared to the error rate when all the predictors are used. If the error rate improves without the predictor, it is deemed to be irrelevant and is provisionally removed.

Once the tentative list of non-informative predictors is established, C5.0 recreates the tree. If the error rate has become worse, the winnowing process is disabled and no predictors are excluded.

After the important predictors are established (if any), the conventional C5.0 training process is used with the full training set but with *only* the predictors that survived the winnowing procedure.

For example, C5.0 split the grant data into roughly equal parts, built a tree on one-half of the data, and used the second half to estimate the error rate to be about 14.6 %. When the predictor related to the number of student chief investigators was excluded, the error rate decreased slightly to 14.2 %. Given this, the number of student chief investigators was excluded from further consideration. Conversely, when the contract value band was excluded, the error rate rose to 24.8 %. This predictor was retained for subsequent C5.0 models.

Grant Data

For the grant data, several variations of the C5.0 model were evaluated:

- Single tree- and rule-based models
- Tree and rules with boosting (up to 100 iterations)
- All predictors and the winnowed set
- The two previously described approaches for handling categorical predictors

For the last set of model conditions, there was very little difference in the models. Figure 14.13 shows the ROC curves for the two methods of encoding the categorical predictors. The curves are almost identical.

The top panel of Fig. 14.13 shows the tuning profile for the C5.0 models with grouped categories. There was a slight decrease in performance when the winnowing algorithm was applied, although this is likely to be within the experimental noise of the data. Boosting clearly had a positive effect for these models and there is marginal improvement after about 50 iterations. Although single rules did worse than single trees, boosting showed the largest impact on the rule-based models, which ended up having superior performance. The optimal area under the ROC curve for this model was 0.942, the best seen among the models.

What predictors were used in the model? First, it may be helpful to know how often each predictor was used in a rule across all iterations of boosting. The vast majority of the predictors were used rarely; 99 % of the predictors were used in less than 0.71 % of the rules. The ten most frequent predictors were: contract value band (9.2 %), the number of unsuccessful grants by chief investigators (8.3 %), the number of successful grants by chief investigators (7.1 %), numeric day of the year (6.3 %), category code (6 %), Month (3.5 %), day of the week (3.1 %), sponsor code (2.8 %), the number of external chief investigators (1.1 %), and the number of C journal papers by chief investigators (0.9 %). As previously described, the predictors can be ranked by their

importance values, as measured by the aggregate percentage of samples covered by the predictor. With boosting, this metric is less informative since the predictor in the first split is calculated to have 100 % importance. In this model, where a significant number of boosting iterations were used, 40 predictors had importance values of 100 %. This model only used 357 predictors (24 %).

14.7 Comparing Two Encodings of Categorical Predictors

All of the models fit in this chapter used two methods for encoding categorical predictors. Figure 14.14 shows the results of the holdout set for each model and approach. In general, large differences in the area under the ROC curve were not seen between the two encodings. J48 saw a loss in sensitivity with separate binary dummy variables, while stochastic gradient boosting and PART have losses in specificity when using grouped variables. In some cases, the encodings did have an effect on the complexity of the model. For the boosted trees, the choice of encodings resulted in very different tuning profiles, as demonstrated in Figs. 14.10 and 14.11. It is difficult to extrapolate these findings to other models and other data sets, and, for this reason, it may be worthwhile to try both encodings during the model training phase.

14.8 Computing

This section uses functions from the following packages: C50, caret, gbm, ipred, partykit, pROC, randomForest, and RWeka. This section also uses the same R objects created in Sect. 12.7 that contain the Grant Applications data (such as the data frame `training`).

In addition to the sets of dummy variables described in Sect. 12.7, several of the categorical predictors are encoded as R factors: `SponsorCode`, `ContractValueBand`, `CategoryCode`, and `Weekday`. When fitting models with independent categories for these predictors, the character vector `fullSet` is used. When treating the categorical predictors as a cohesive set, an alternate list of predictors is contained in the vector `factorPredictors`, which contains the factor versions of the relevant data. Additionally, the character string `factorForm` is an R formula created using all the predictors contained in `factorPredictors` (and is quite long).

A good deal of the syntax shown in this section is similar to other computing sections, especially the previous one related to regression trees. The focus here will be on the nuances of individual model functions and interpreting their output. Some code is shown to recreate the analyses in this chapter.

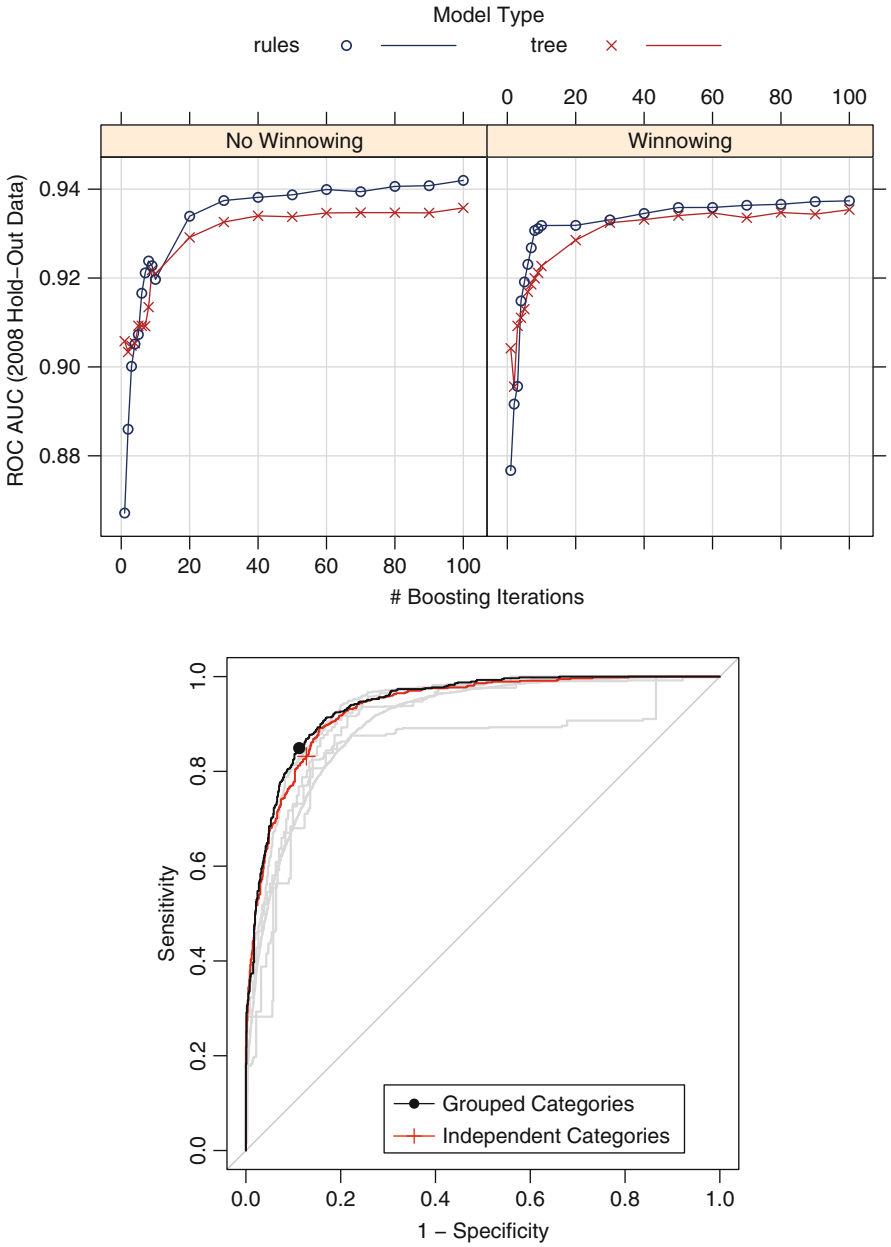

Fig. 14.13: *Top*: The parameter tuning profile for the C5.0 model using grouped categories. *Bottom*: The ROC curves for the boosted C5.0 models. The grouped and independent categories versions of the model are almost identical, with an area under the ROC curve of 0.942

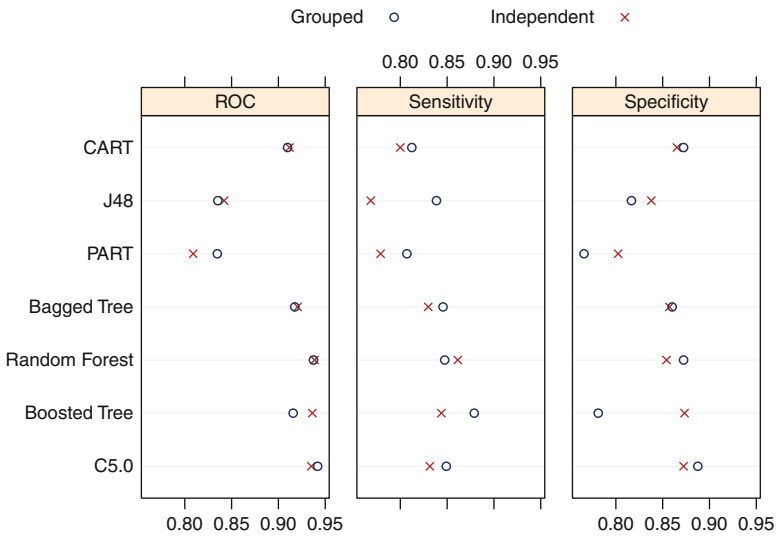

Fig. 14.14: The effect of different methods of representing categorical predictors in tree- and rule-based models. "Grouped" indicates that the categories for a predictor were treated as a cohesive set, while "independent" indicates that the categories were converted to independent dummy variables prior to modeling

A comprehensive program for the models shown is contained in the `Chapter` directory of the AppliedPredictiveModeling package.

Classification Trees

There are a number of R packages to build single classification trees. The primary package is rpart. As discussed in regression, the function takes only the formula method for specifying the exact form of the model.

There are a large number of predictors for the grant data, and, as previously mentioned, an R formula was created programmatically to model the classes for grouped categories. The following syntax fits a CART model to these predictors with our data splitting strategy:

```
> library(rpart)
> cartModel <- rpart(factorForm, data = training[pre2008,])
```

This automatically grows and prunes the tree using the internal cross-validation procedure. One important argument for classification is `parms`. Here, several alterations to the model training process can be declared, such as the prior probabilities of the outcome and the type of splitting (either

the Gini coefficient or the information statistic). These values should be in a list.[6] See `?rpart` for the details. Also, the `control` argument can customize the fitting procedure in terms of the numerical methods (such as the tree depth).

The model output is somewhat different than in regression trees. To show this we generate a smaller model with two predictors:

```
> rpart(Class ~ NumCI + Weekday, data = training[pre2008,])
  n= 6633

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

  1) root 6633 3200 unsuccessful (0.49 0.51)
    2) Weekday=Sun 223    0 successful (1.00 0.00) *
    3) Weekday=Fri,Mon,Sat,Thurs,Tues,Wed 6410 3000 unsuccessful (0.47 0.53)
      6) Weekday=Mon,Thurs,Tues 2342 1000 successful (0.57 0.43) *
      7) Weekday=Fri,Sat,Wed 4068 1700 unsuccessful (0.41 0.59) *
```

The output shows the split variable/value, along with how many samples were partitioned into the branch (223 for the second node in the output above). The majority class is also printed (successful for node 2) and the predicted class probabilities for samples that terminate in this node.

Prediction syntax is nearly the same as other models in R. The `predict` function, by default, produces probabilities for each class. Using `predict(object, type = "class")` generates a factor vector of the winning class.

The R implementation of C4.5 is in the RWeka package in a function called `J48`. The function also takes a model formula:

```
> library(RWeka)
> J48(Class ~ NumCI + Weekday, data = training[pre2008,])
  J48 pruned tree
  ------------------

  Weekday = Fri: unsuccessful (1422.0/542.0)
  Weekday = Mon: successful (1089.0/455.0)
  Weekday = Sat
  |   NumCI <= 1: unsuccessful (1037.0/395.0)
  |   NumCI > 1
  |   |   NumCI <= 3: unsuccessful (378.0/185.0)
  |   |   NumCI > 3: successful (61.0/26.0)
  Weekday = Sun: successful (223.0)
  Weekday = Thurs
  |   NumCI <= 0: unsuccessful (47.0/21.0)
  |   NumCI > 0: successful (520.0/220.0)
  Weekday = Tues
  |   NumCI <= 2
  |   |   NumCI <= 0: unsuccessful (45.0/21.0)
  |   |   NumCI > 0: successful (585.0/251.0)
```

[6] An example of this type of argument is shown in Sect. 16.9 where `rpart` is fit using with differential costs for different types of errors.

```
|   NumCI > 2: unsuccessful (56.0/22.0)
Weekday = Wed: unsuccessful (1170.0/521.0)

Number of Leaves  :     12

Size of the tree :  18
```

Recall that this implementation of C4.5 does not attempt to group the categories prior to pruning. The prediction function automatically produces the winning classes and the class probabilities can be obtained from `predict(object, type = "prob")`.

When visualizing CART or J48 trees, the `plot` function from the partykit package can create detailed displays. The objects must be converted to the appropriate class with `as.party`, followed by the `plot` function.

A single C5.0 tree can be created from the C50 package:

```
> library(C50)
> C5tree <- C5.0(Class ~ NumCI + Weekday, data = training[pre2008,])
> C5tree

Call:
C5.0.formula(formula = Class ~ NumCI + Weekday, data
 = training[pre2008, ])

Classification Tree
Number of samples: 6633
Number of predictors: 2

Tree size: 2

Non-standard options: attempt to group attributes
> summary(C5tree)

Call:
C5.0.formula(formula = Class ~ NumCI + Weekday, data
 = training[pre2008, ])


C5.0 [Release 2.07 GPL Edition]     Thu Dec  6 13:53:14 2012
-------------------------------

Class specified by attribute `outcome'

Read 6633 cases (3 attributes) from undefined.data

Decision tree:

Weekday in Tues,Mon,Thurs,Sun: successful (2565/1010)
Weekday in Fri,Wed,Sat: unsuccessful (4068/1678)


Evaluation on training data (6633 cases):

        Decision Tree
```

```
        ----------------
        Size      Errors

           2 2688(40.5%)   <<


         (a)   (b)    <-classified as
        ----  ----
        1555  1678    (a): class successful
        1010  2390    (b): class unsuccessful


      Attribute usage:

      100.00% Weekday

  Time: 0.0 secs
```

Note that, unlike `J48`, this function is able to split the weekday values from groups of values. The control function for this model (`C5.0Control`) turns this feature off (`subset = FALSE`). Other options are available here, such as winnowing and the confidence factor for splitting. Like `J48`, the default prediction function produces classes and `type = "prob"` produces the probabilities.

There are wrappers for these models using the caret function `train`. For example, to fit the grouped category model for CART, we used:

```
> set.seed(476)
> rpartGrouped <- train(x = training[,factorPredictors],
+                       y = training$Class,
+                       method = "rpart",
+                       tuneLength = 30,
+                       metric = "ROC",
+                       trControl = ctrl)
```

Recall that the `ctrl` object specifies which data are in the holdout set and what performance measures should be calculated (e.g., sensitivity, specificity, and the area under the ROC curve). The model codes for J48 and C5.0 trees are `J48` and `C5.0Tree`, respectively. The main differences here between `train` and the original model function are a unified interface to the models and the ability to tune the models with alternative metrics, such as the area under the ROC curve.

Note that `rpart`, `C5.0`, and `J48` use the formula method differently than most other functions. Usually, the formula method automatically decomposes any categorical predictors to a set of binary dummy variables. These functions respect the categorical nature of the data and treat these predictors as grouped sets of categories (unless the data are already converted to dummy variables). The `train` function follows the more common convention in R, which is to create dummy variables prior to modeling. This is the main reason the code snippet above is written with the non-formula method when invoking `train`.

Rules

There are several rule-based models in the RWeka package. The `PART` function creates models based on Frank and Witten (1998). Its syntax is similar to `J48`:

```
> PART(Class ~ NumCI + Weekday, data = training[pre2008,])
  PART decision list
  ------------------

  Weekday = Fri: unsuccessful (1422.0/542.0)

  Weekday = Sat AND
  NumCI <= 1: unsuccessful (1037.0/395.0)

  Weekday = Mon: successful (1089.0/455.0)

  Weekday = Thurs AND
  NumCI > 0: successful (520.0/220.0)

  Weekday = Wed: unsuccessful (1170.0/521.0)

  Weekday = Tues AND
  NumCI <= 2 AND
  NumCI > 0: successful (585.0/251.0)

  Weekday = Sat AND
  NumCI <= 3: unsuccessful (378.0/185.0)

  Weekday = Sun: successful (223.0)

  Weekday = Tues: unsuccessful (101.0/43.0)

  Weekday = Sat: successful (61.0/26.0)

  : unsuccessful (47.0/21.0)

  Number of Rules  :  11
```

Other RWeka functions for rules can be found on the help page `?Weka_classifier_rules`.

C5.0 rules are created using the `C5.0` function in the same manner as trees, but with the `rules = TRUE` option:

```
> C5rules <- C5.0(Class ~ NumCI + Weekday, data = training[pre2008,],
+                 rules = TRUE)
> C5rules
  Call:
  C5.0.formula(formula = Class ~ NumCI + Weekday, data
   = training[pre2008, ], rules = TRUE)

  Rule-Based Model
  Number of samples: 6633
```

```
  Number of predictors: 2

  Number of Rules: 2

  Non-standard options: attempt to group attributes
> summary(C5rules)

  Call:
  C5.0.formula(formula = Class ~ NumCI + Weekday, data
   = training[pre2008, ], rules = TRUE)


  C5.0 [Release 2.07 GPL Edition]     Thu Dec  6 13:53:14 2012
  -------------------------------

  Class specified by attribute `outcome'

  Read 6633 cases (3 attributes) from undefined.data

  Rules:

  Rule 1: (2565/1010, lift 1.2)
      Weekday in Tues, Mon, Thurs, Sun
      ->  class successful  [0.606]

  Rule 2: (4068/1678, lift 1.1)
      Weekday in Fri, Wed, Sat
      ->  class unsuccessful  [0.587]

  Default class: unsuccessful


  Evaluation on training data (6633 cases):

              Rules
        ----------------
          No      Errors

           2 2688(40.5%)   <<


         (a)   (b)    <-classified as
        ----  ----
        1555  1678    (a): class successful
        1010  2390    (b): class unsuccessful


      Attribute usage:

      100.00% Weekday


  Time: 0.0 secs
```

Prediction follows the same syntax as above. The variable importance scores for C5.0 trees and rules is calculated using the `C5imp` function or the `varImp` function in the caret package.

When working with the `train` function, model codes `C5.0Rules` and `PART` are available.

Other packages for single trees include party (conditional inference trees), tree (CART trees), oblique.tree (oblique trees), partDSA (for the model of Molinaro et al. (2010)), and evtree (trees developed using genetic algorithms). Another class of partitioning methods not discussed here called Logic Regression (Ruczinski et al. 2003) are implemented in several packages, including LogicReg.

Bagged Trees

The primary tree bagging package is ipred. The `bagging` function creates bagged versions of rpart trees using the formula method (another function, `ipredbagg`, uses the non-formula method). The syntax is familiar:

```
> bagging(Class ~ Weekday + NumCI, data = training[pre2008,])
```

The argument `nbagg` controls how many trees are in the ensemble (25 by default). The default for the standard `predict` method is to determine the winning class and `type = "prob"` will produce the probabilities.

Another function in the caret package, called `bag`, creates bag models more generally (i.e., models other than trees).

Random Forest

The R port of the original random forest program is contained in the randomForest package and its basic syntax is identical to the regression tree code shown on p. 215. The default value of $m_{try} \approx \sqrt{p}$ is different than in regression. One option, `cutoff`, is specific to classification and controls the voting cutoff(s) for determining the winning class from the ensemble of trees. This particular option is also available when using random forest's `predict` function.

The model takes the formula and non-formula syntax. In either case, any categorical predictors encoded as R factor variables are treated as a group. The `predict` syntax defaults to generating the winning class, but the `type` argument allows for predicting other quantities such as the class probabilities (`type = "prob"`) or the actual vote counts `type = "votes"`.

A basic example for the grant data, with output, is:

```
> library(randomForest)
> randomForest(Class ~ NumCI + Weekday, data = training[pre2008,])
  Call:
   randomForest(formula = Class ~ NumCI + Weekday, data = training[pre2008, ])
                 Type of random forest: classification
                       Number of trees: 500
  No. of variables tried at each split: 1

          OOB estimate of  error rate: 40.06%
  Confusion matrix:
               successful unsuccessful class.error
  successful         1455         1778   0.5499536
  unsuccessful        879         2521   0.2585294
```

Since only two predictors are included, only a single predictor is randomly selected at each split.

The function prints the out-of-bag error estimate, as well as the analogous confusion matrix. Out-of-bag estimates of the sensitivity and the false positive rate (i.e., 1—specificity) are shown under the column `class.error`.

The model code for tuning a random forest model with `train` is `"rf"`.

Other random forests functions are `cforest` (in the party package), `obliqueRF` (forests from oblique trees in the obliqueRF package), `rFerns` (for the random fern model of Ozuysal et al. (2010) in the rFerns package), and `RRF` (regularized random forest models in the RRF package).

Boosted Trees

The primary boosted tree package in R is gbm, which implements stochastic gradient boosting. The primary difference between boosting regression and classification trees is the choice of the distribution of the data. The `gbm` function can only accommodate two class problems and using `distribution = "bernoulli"` is an appropriate choice here. Another option is `distribution = "adaboost"` to replicate the loss function used by that methodology.

One complication when using `gbm` for classification is that it expects that the outcome is coded as 0/1. An example of a simple model for the grant data would be

```
> library(gbm)
> forGBM <- training
> forGBM$Class <- ifelse(forGBM$Class == "successful", 1, 0)
> gbmModel <- gbm(Class ~ NumCI + Weekday,
+                 data = forGBM[pre2008,],
+                 distribution = "bernoulli",
+                 interaction.depth = 9,
+                 n.trees = 1400,
+                 shrinkage = 0.01,
+                 ## The function produces copious amounts
```

```
+                 ## of output by default.
+                 verbose = FALSE)
```

The prediction function for this model does not predict the winning class. Using `predict(gbmModel, type = "response")` will calculate the class probability for the class encoded as a 1 (in our example, a successful grant was encoded as a 1). This can be converted to a factor variable with the winning class:

```
> gbmPred <- predict(gbmModel,
+                    newdata = head(training[-pre2008,]),
+                    type = "response",
+                    ## The number of trees must be
+                    ## explicitly set
+                    n.trees = 1400)
> gbmPred
  [1] 0.5697346 0.5688882 0.5688882 0.5688882 0.5697346 0.5688882
> gbmClass <- ifelse(gbmPred > .5, "successful", "unsuccessful")
> gbmClass <- factor(gbmClass, levels = levels(training$Class))
> gbmClass
  [1] successful successful successful successful successful successful
  Levels: successful unsuccessful
```

Fitting this model with `train` simplifies the process considerably. For example, a factor variable can be used as the outcome format (`train` automatically does the conversion). When predicting the winning class, a factor is produced. If the class probabilities are required, then specify `predict(object, type = "prob")` (`train`'s prediction function automatically uses the number of trees that were found to be optimal during model tuning).

The original AdaBoost algorithm is available in the ada package. Another function for boosting trees is `blackboost` in the mboost package. This package also contains functions for boosting other types of models (such as logistic regression) as does the bst package.

To train boosted versions of C5.0, the `trials` argument is used (with values between 1 and 100).

```
> library(C50)
> C5Boost <- C5.0(Class ~ NumCI + Weekday, data = training[pre2008,],
+                 trials = 10)
> C5Boost

  Call:
  C5.0.formula(formula = Class ~ NumCI + Weekday, data
   = training[pre2008, ], trials = 10)

  Classification Tree
  Number of samples: 6633
  Number of predictors: 2

  Number of boosting iterations: 10 requested;  6 used due to early stopping
  Average tree size: 2.5

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

By default, the algorithm has internal tests that assess whether the boosting is effective and will halt the model training when it diagnoses that it is no longer effective (note the message that ten iterations were requested but only six were used due to early stopping). This feature can be negated using `C5.0Control(earlyStopping = FALSE)`.

These models can be tuned by `train` using `method` values of `gbm`, `ada`, or `C5.0`.

Exercises

14.1. Variable importance for the bagging, random forests, and boosting has been computed for both the independent categories and the factor model predictors. The top 16 important predictors for each method and predictor set are presented in Fig. 14.15.

(a) Within each modeling technique, which factors are in common between the independent category and factor models?
(b) How do these results compare with the most prolific predictors found in the PART model results discussed in Sect. 14.2?

14.2. For the churn data described in Exercise 12.3:

(a) Fit a few basic trees to the training set. Should the area code be encoded as independent dummy variables or as a grouped set of values?
(b) Does bagging improve the performance of the trees? What about boosting?
(c) Apply rule-based models to the data. How is the performance? Do the rules make any sense?
(d) Use lift charts to compare tree or rule models to the best techniques from previous chapters.

14.3. Exercise 12.1 gives a detailed description of the hepatic injury data set, where the primary scientific objective for these data is to construct a model to predict hepatic injury status. Recall that random forests can be performed with CART trees or conditional inference trees. Start R and use these commands to load the data:

```
> library(AppliedPredictiveModeling)
> data(hepatic)
```

(a) Fit a random forest model using both CART trees and conditional inference trees to the chemistry predictors, using the Kappa statistic as the metric as follows:

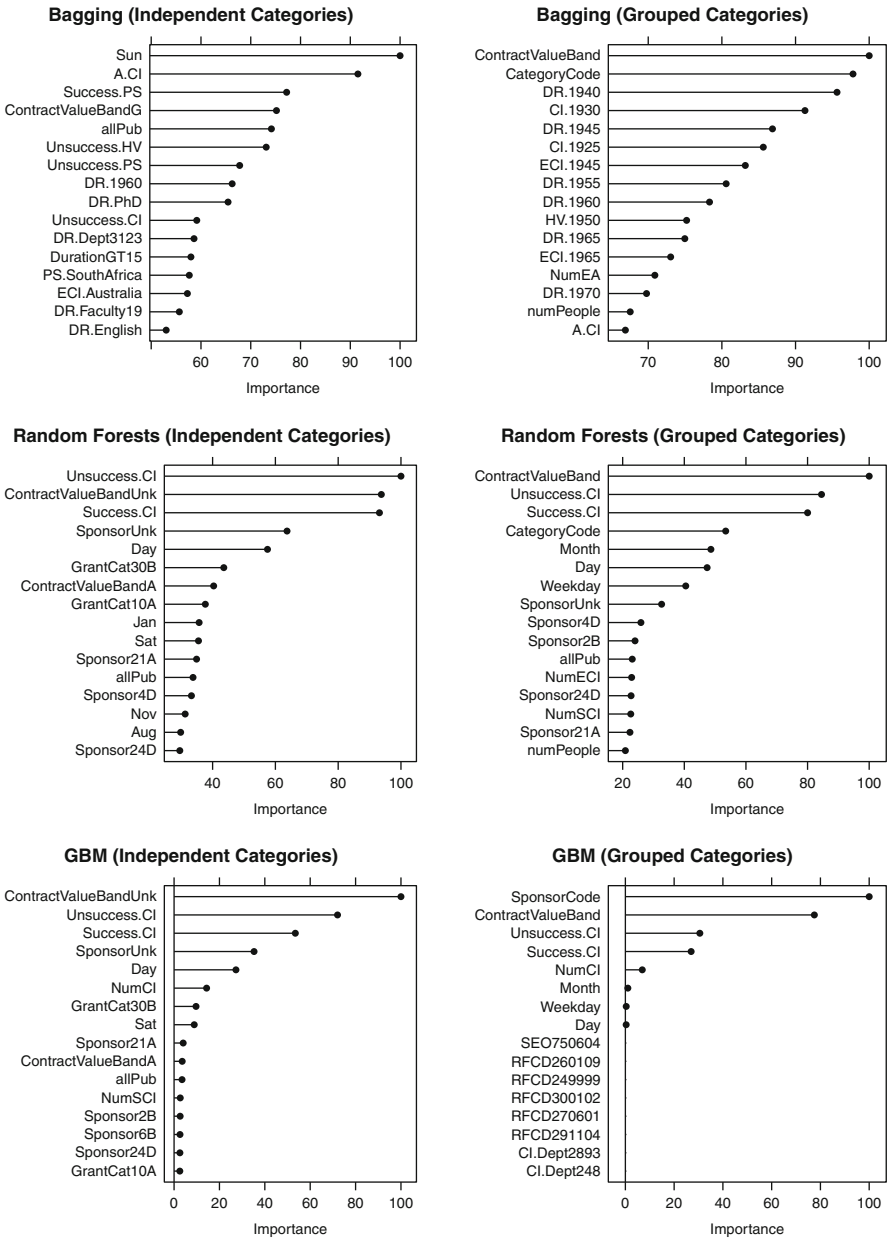

Fig. 14.15: A comparison of variable importance for the ensemble methods of bagging, random forests, and boosting for both the independent categories and grouped categories predictors

```
> library(caret)
> set.seed(714)
> indx <- createFolds(injury, returnTrain = TRUE)
> ctrl <- trainControl(method = "cv", index = indx)
> mtryValues <- c(5, 10, 25, 50, 75, 100)
> rfCART <- train(chem, injury,
+                 method = "rf",
+                 metric = "Kappa",
+                 ntree = 1000,
+                 tuneGrid = data.frame(.mtry = mtryValues))
> rfcForest <- train(chem, injury,
+                    method = "cforest",
+                    metric = "Kappa",
+                    tuneGrid = data.frame(.mtry = mtryValues))
```

Which model has better performance, and what are the corresponding tuning parameters?

(b) Use the following syntax to obtain the computation time for each model:

```
> rfCART$times$everything
> rfcForest$times$everything
```

Which model takes less computation time? Given the trade-off between performance and computation time, which model do you prefer?

(c) Use the following syntax to obtain the variable importance for the top ten predictors for each model:

```
> varImp(rfCART)
> varImp(rfcForest)
```

Are there noticeable differences in variable importance between the top ten predictors for each model? Explain possible reasons for the differences.