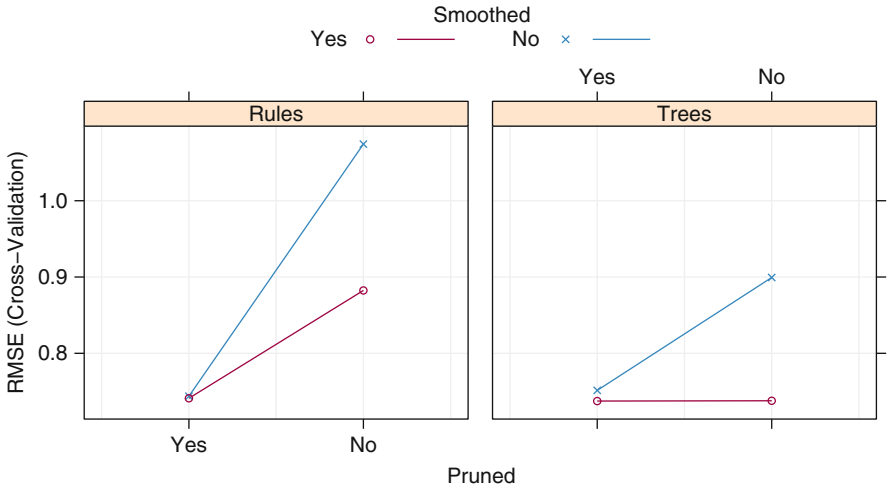

Fig. 8.12: Cross-validated RMSE profiles for the model trees before and after the conversion to rules

covered by at least two rules. The other rules used many of the same predictors: `SurfaceArea1` (five times), `MolWeight` (three times), and `NumCarbon` (also three times). Figure 8.13 shows the coefficients of the linear models for each rule (similar to Fig. 8.11 for the full model tree). Here, the linear models are more sparse; the number of terms in the linear models decreases as more rules are created. This makes sense because there are fewer data points to construct deep trees.

8.4 Bagged Trees

In the 1990s, ensemble techniques (methods that combine many models' predictions) began to appear. Bagging, short for *b*ootstrap *agg*regation, was originally proposed by Leo Breiman and was one of the earliest developed ensemble techniques (Breiman 1996a). Bagging is a general approach that uses bootstrapping (Sect. 4.4) in conjunction with any regression (or classification; see Sect. 14.3) model to construct an ensemble. The method is fairly simple in structure and consists of the steps in Algorithm 8.1. Each model in the ensemble is then used to generate a prediction for a new sample and these m predictions are averaged to give the bagged model's prediction.

Bagging models provide several advantages over models that are not bagged. First, bagging effectively reduces the variance of a prediction through its aggregation process (see Sect. 5.2 for a discussion of the bias-variance trade-off). For models that produce an unstable prediction, like regression trees, aggregating over many versions of the training data actually reduces

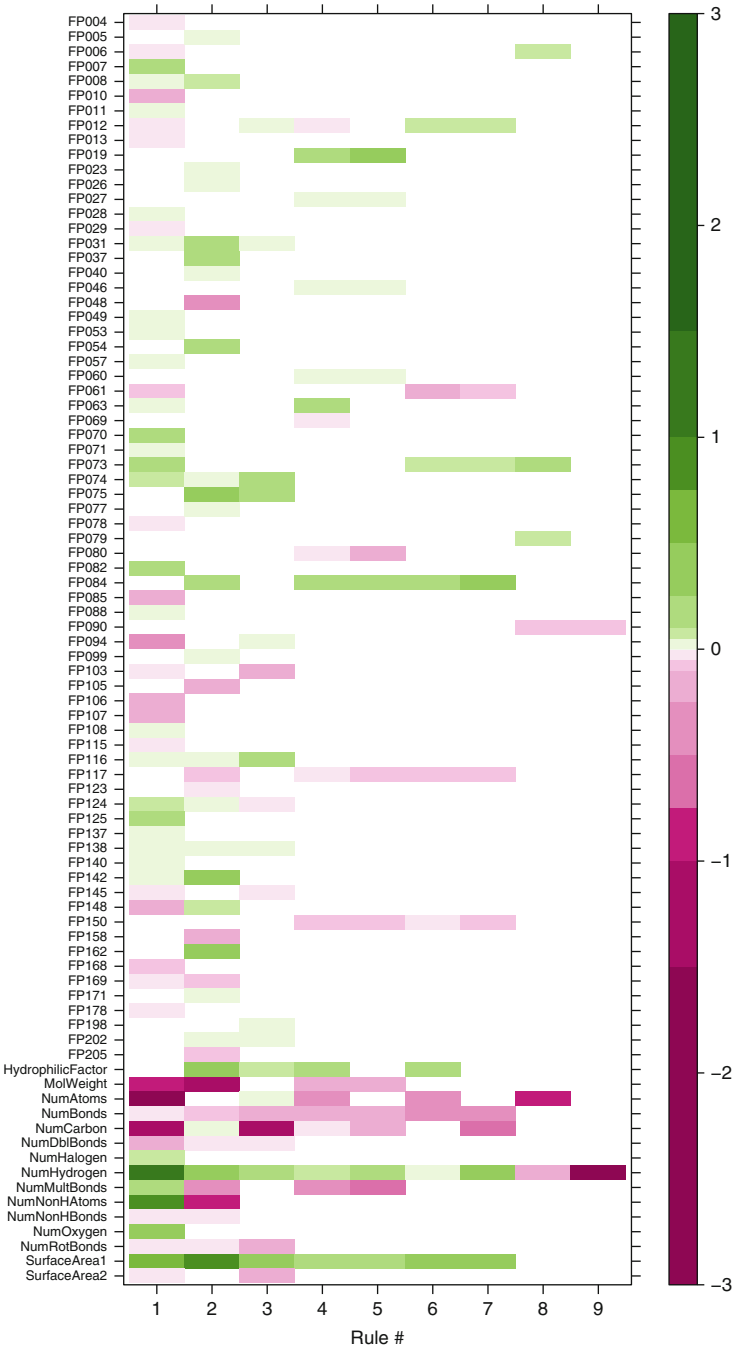

Fig. 8.13: Linear model coefficients for the rule-based version of M5. The coefficients have been normalized to be on the same scale as Fig. 8.11. *White blocks* indicate that the predictor was not involved in linear regression equation for that rule

```
for i = 1 to m do
    Generate a bootstrap sample of the original data
    Train an unpruned tree model on this sample
end
```

Algorithm 8.1: Bagging

the variance in the prediction and, hence, makes the prediction more stable. Consider the illustration of trees in Fig. 8.14. In this example, six bootstrap samples of the solubility data were generated and a tree of maximum depth was built for each sample. These trees vary in structure (compare Fig. 8.14b, d, which have different structures on the right- and left-hand sides of each tree), and hence the prediction for samples will vary from tree to tree. When the predictions for a sample are averaged across all of the single trees, the average prediction has lower variance than the variance across the individual predictions. This means that if we were to generate a different sequence of bootstrap samples, build a model on each of the bootstrap samples, and average the predictions across models, then we would likely get a very similar predicted value for the selected sample as with the previous bagging model. This characteristic also improves the predictive performance of a bagged model over a model that is not bagged. If the goal of the modeling effort is to find the best prediction, then bagging has a distinct advantage.

Bagging stable, lower variance models like linear regression and MARS, on the other hand, offers less improvement in predictive performance. Consider Fig. 8.15, in which bagging has been applied to trees, linear models, and MARS for the solubility data and also for data from a study of concrete mixtures (see Chap. 10). For each set of data, the test set performance based on $RMSE$ is plotted by number of bagging iterations. For the solubility data, the decrease in RMSE across iterations is similar for trees, linear regression, and MARS, which is not a typical result. This suggests that either the model predictions from linear regression and MARS have some inherent instability for these data which can be improved using a bagged ensemble or that trees are less effective at modeling the data. Bagging results for the concrete data are more typical, in which linear regression and MARS are least improved through the ensemble, while the predictions for regression trees are dramatically improved.

As a further demonstration of bagging's ability to reduce the variance of a model's prediction, consider the simulated *sin* wave in Fig. 5.2. For this illustration, 20 *sin* waves were simulated, and, for each data set, regression trees and MARS models were computed. The red lines in the panels show the true trend while the multiple black lines show the predictions for each model. Note that the CART panel has more noise around the true *sin* curve than the MARS model, which only shows variation at the change points of the pattern. This illustrates the high variance in the regression tree due to model

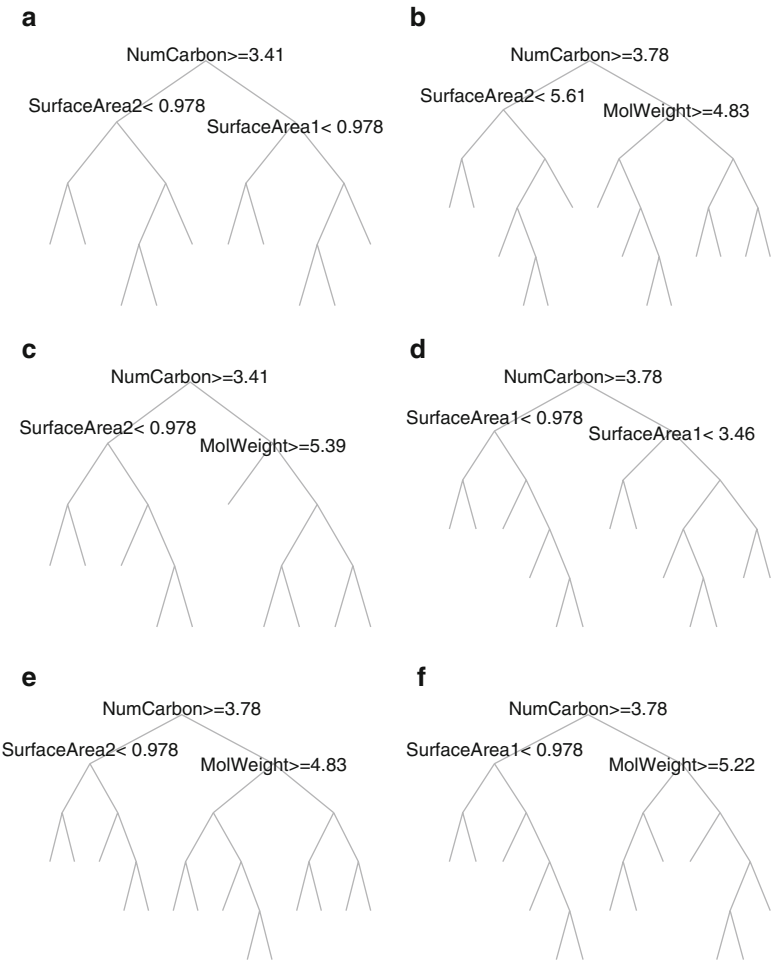

Fig. 8.14: Example of trees of maximum depth from bagging for the solubility data. Notice that the trees vary in structure, and hence the predictions will vary from tree to tree. The prediction variance for the ensemble of trees will be less than the variance of predictions from individual trees. (**a**) Sample 1. (**b**) Sample 2. (**c**) Sample 3. (**d**) Sample 4. (**e**) Sample 5. (**f**) Sample 6

instability. The bottom panels of the figure show the results for 20 bagged regression trees and MARS models (each with 50 model iterations). The variation around the true curve is greatly reduced for regression trees, and, for MARS, the variation is only reduced around the curvilinear portions on the pattern. Using a simulated test set for each model, the average reduction in RMSE by bagging the tree was 8.6 % while the more stable MARS model had a corresponding reduction of 2 % (Fig. 8.16).

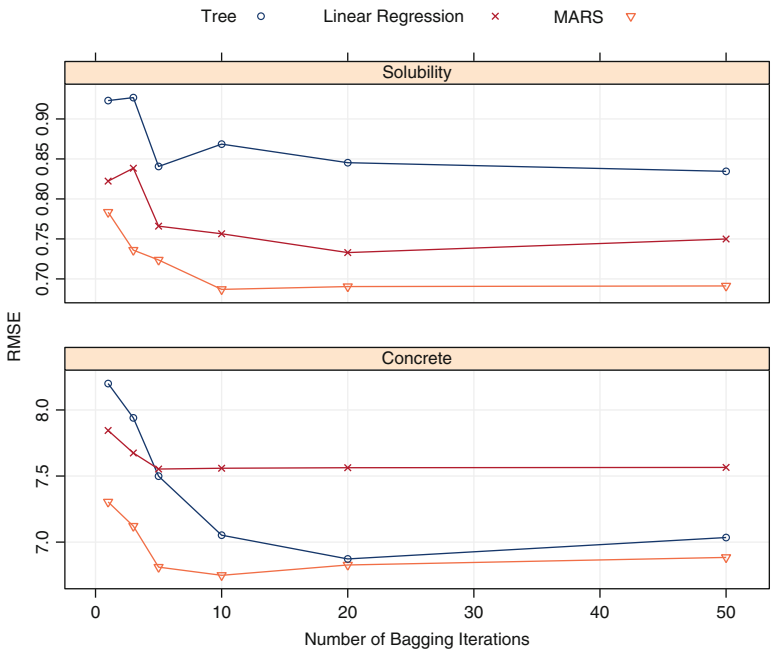

Fig. 8.15: Test set RMSE performance profiles for bagging trees, linear models, and MARS for the solubility data (*top plot*) and concrete data (*bottom plot*; see Chap. 10 for data details) by number of bootstrap samples. Bagging provides approximately the same magnitude of improvement in RMSE for all three methods for the solubility data, which is atypical. A more typical pattern of improvement from bagging is shown for the concrete data. In this case, trees benefit the most

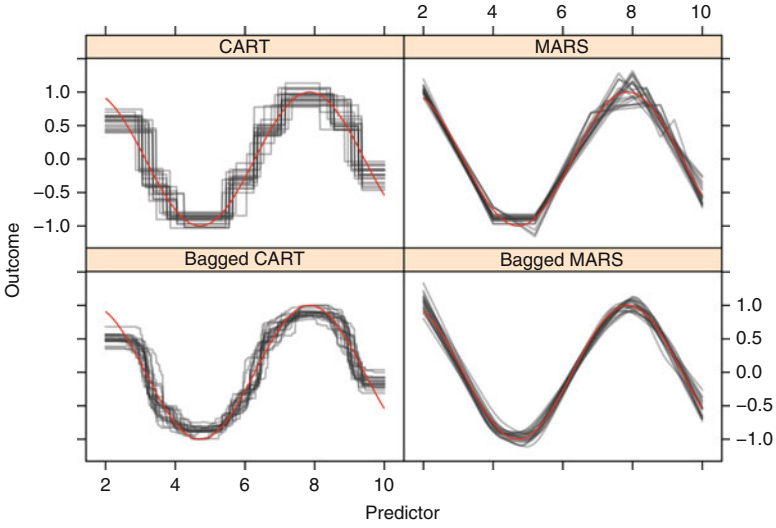

Fig. 8.16: The effect of bagging regression trees and MARS models

Another advantage of bagging models is that they can provide their own internal estimate of predictive performance that correlates well with either cross-validation estimates or test set estimates. Here's why: when constructing a bootstrap sample for each model in the ensemble, certain samples are left out. These samples are called *out-of-bag*, and they can be used to assess the predictive performance of that specific model since they were not used to build the model. Hence, every model in the ensemble generates a measure of predictive performance courtesy of the out-of-bag samples. The average of the out-of-bag performance metrics can then be used to gauge the predictive performance of the entire ensemble, and this value usually correlates well with the assessment of predictive performance we can get with either cross-validation or from a test set. This error estimate is usually referred to as the *out-of-bag* estimate.

In its basic form, the user has one choice to make for bagging: the number of bootstrap samples to aggregate, m. Often we see an exponential decrease in predictive improvement as the number of iterations increases; the most improvement in prediction performance is obtained with a small number of trees ($m < 10$). To illustrate this point, consider Fig. 8.17 which displays predictive performance ($RMSE$) for varying numbers of bootstrapped samples for CART trees. Notice predictive performance improves through ten trees and then tails off with very limited improvement beyond that point. In our experience, small improvements can still be made using bagging ensembles up to size 50. If performance is not at an acceptable level after 50 bagging iterations, then we suggest trying other more powerfully predictive ensemble methods such as random forests and boosting which will be described the following sections.

For the solubility data, CART trees without bagging produce an optimal cross-validated RMSE of 0.97 with a standard error of 0.021. Upon bagging, the performance improves and bottoms at an RMSE of 0.9, with a standard error of 0.019. Conditional inference trees, like CART trees, can also be bagged. As a comparison, conditional inference trees without bagging have an optimal RMSE and standard error of 0.93 and 0.034, respectively. Bagged conditional inference trees reduce the optimal RMSE to 0.8 with a standard error of 0.018. For both types of models, bagging improves performance and reduces variance of the estimate. In this specific example, bagging conditional inference trees appears to have a slight edge over CART trees in predictive performance as measured by RMSE. The test set R^2 values parallel the cross-validated RMSE performance with conditional inference trees doing slightly better (0.87) than CART trees (0.85).

Although bagging usually improves predictive performance for unstable models, there are a few caveats. First, computational costs and memory requirements increase as the number of bootstrap samples increases. This disadvantage can be mostly mitigated if the modeler has access to parallel computing because the bagging process can be easily parallelized. Recall that each bootstrap sample and corresponding model is independent of any other

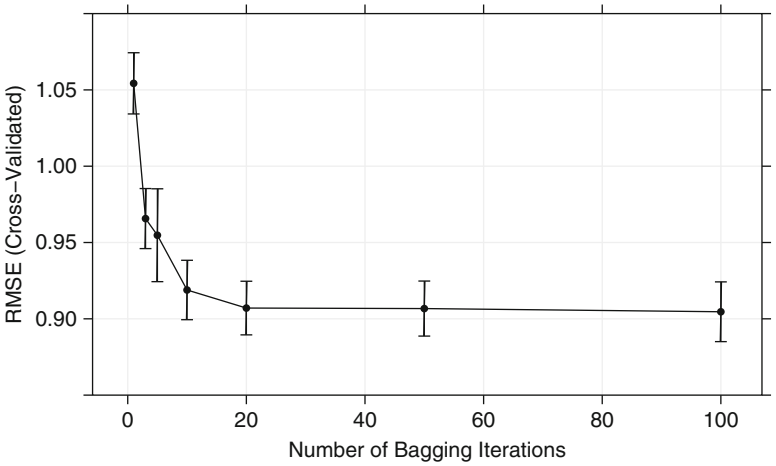

Fig. 8.17: Cross-validated performance profile for bagging CART trees for the solubility data by number of bootstrap samples. Vertical lines indicate ± one-standard error of $RMSE$. Most improvement in predictive performance is obtained aggregating across ten bootstrap replications

sample and model. This means that each model can be built separately and all models can be brought together in the end to generate the prediction.

Another disadvantage to this approach is that a bagged model is much less interpretable than a model that is not bagged. Convenient rules that we can get from a single regression tree like those displayed in Fig. 8.4 cannot be attained. However, measures of variable importance can be constructed by combining measures of importance from the individual models across the ensemble. More about variable importance will be discussed in the next section when we examine random forests.

8.5 Random Forests

As illustrated with the solubility data, bagging trees (or any high variance, low bias technique) improves predictive performance over a single tree by reducing variance of the prediction. Generating bootstrap samples introduces a random component into the tree building process, which induces a distribution of trees, and therefore also a distribution of predicted values for each sample. The trees in bagging, however, are not completely independent of each other since all of the original predictors are considered at every split of every tree. One can imagine that if we start with a sufficiently large number of original samples and a relationship between predictors and response that

can be adequately modeled by a tree, then trees from different bootstrap samples may have similar structures to each other (especially at the top of the trees) due to the underlying relationship. This characteristic is known as tree correlation and prevents bagging from optimally reducing variance of the predicted values. Figure 8.14 provides a direct illustration of this phenomenon. Despite taking bootstrap samples, each tree starts splitting on the number of carbon atoms at a scaled value of approximately 3.5. The second-level splits vary a bit more but are restricted to both of the surface area predictors and molecular weight. While each tree is ultimately unique—no two trees are exactly the same—they all begin with a similar structure and are consequently related to each other. Therefore, the variance reduction provided by bagging could be improved. For a mathematical explanation of the tree correlation phenomenon, see Hastie et al. (2008). Reducing correlation among trees, known as de-correlating trees, is then the next logical step to improving the performance of bagging.

From a statistical perspective, reducing correlation among predictors can be done by adding randomness to the tree construction process. After Breiman unveiled bagging, several authors tweaked the algorithm by adding randomness into the learning process. Because trees were a popular learner for bagging, Dietterich (2000) developed the idea of random split selection, where trees are built using a random subset of the top k predictors at each split in the tree. Another approach was to build entire trees based on random subsets of descriptors (Ho 1998; Amit and Geman 1997). Breiman (2000) also tried adding noise to the response in order to perturb tree structure. After carefully evaluating these generalizations to the original bagging algorithm, Breiman (2001) constructed a unified algorithm called *random forests*. A general random forests algorithm for a tree-based model can be implemented as shown in Algorithm 8.2.

Each model in the ensemble is then used to generate a prediction for a new sample and these m predictions are averaged to give the forest's prediction. Since the algorithm randomly selects predictors at each split, tree correlation will necessarily be lessened. As an example, the first splits for the first six trees in the random forest for the solubility data are `NumNonHBonds`, `NumCarbon`, `NumNonHAtoms`, `NumCarbon`, `NumCarbon`, and `NumCarbon`, which are different from the trees illustrated in Fig. 8.14.

Random forests' tuning parameter is the number of randomly selected predictors, k, to choose from at each split, and is commonly referred to as m_{try}. In the regression context, Breiman (2001) recommends setting m_{try} to be one-third of the number of predictors. For the purpose of tuning the m_{try} parameter, since random forests is computationally intensive, we suggest starting with five values of k that are somewhat evenly spaced across the range from 2 to P. The practitioner must also specify the number of trees for the forest. Breiman (2001) proved that random forests is protected from overfitting; therefore, the model will not be adversely affected if a large number of trees are built for the forest. Practically speaking, the larger the forest, the

```
Select the number of models to build, m
for i = 1 to m do
    Generate a bootstrap sample of the original data
    Train a tree model on this sample
    for each split do
        Randomly select k (< P) of the original predictors
        Select the best predictor among the k predictors and
        partition the data
    end
    Use typical tree model stopping criteria to determine when a
    tree is complete (but do not prune)
end
```

Algorithm 8.2: Basic Random Forests

more computational burden we will incur to train and build the model. As a starting point, we suggest using at least 1,000 trees. If the cross-validation performance profiles are still improving at 1,000 trees, then incorporate more trees until performance levels off.

Breiman showed that the linear combination of many independent learners reduces the variance of the overall ensemble relative to any individual learner in the ensemble. A random forest model achieves this variance reduction by selecting strong, complex learners that exhibit low bias. This ensemble of many independent, strong learners yields an improvement in error rates. Because each learner is selected independently of all previous learners, random forests is robust to a noisy response. We elaborate more on this point in Sect. 20.2 and provide an illustration of the effect of noise on random forests as well as many other models. At the same time, the independence of learners can underfit data when the response is not noisy (Fig. 5.1).

Compared to bagging, random forests is more computationally efficient on a tree-by-tree basis since the tree building process only needs to evaluate a fraction of the original predictors at each split, although more trees are usually required by random forests. Combining this attribute with the ability to parallel process tree building makes random forests more computationally efficient than boosting (Sect. 8.6).

Like bagging, CART or conditional inference trees can be used as the base learner in random forests. Both of these base learners were used, as well as 10-fold cross-validation and out-of-bag validation, to train models on the solubility data. The m_{try} parameter was evaluated at ten values from 10 to 228. The RMSE profiles for these combinations are presented in Fig. 8.18. Contrary to bagging, CART trees have better performance than conditional inference trees at all values of the tuning parameter. Each of the profiles

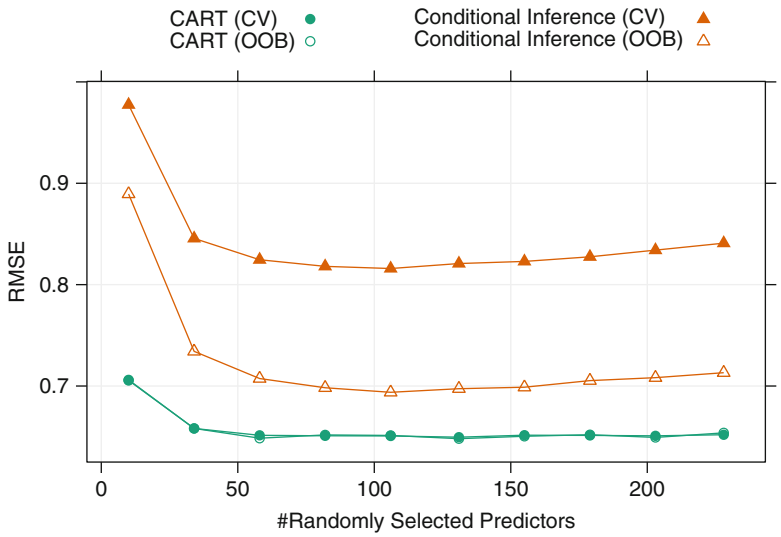

Fig. 8.18: Cross-validated RMSE profile for the CART and conditional inference approaches to random forests

shows a flat plateau between $m_{try} = 58$ and $m_{try} = 155$. The CART-based random forest model was numerically optimal at $m_{try} = 131$ regardless of the method of estimating the RMSE. Our experience is that the random forest tuning parameter does not have a drastic effect on performance. In these data, the only real difference in the RMSE comes when the smallest value is used (10 in this case). It is often the case that such a small value is not associated with optimal performance. However, we have seen rare examples where small tuning parameter values generate the best results. To get a quick assessment of how well the random forest model performs, the default tuning parameter value for regression ($m_{try} = P/3$) tends to work well. If there is a desire to maximize performance, tuning this value may result in a slight improvement.

In Fig. 8.18, also notice that random forest models built with CART trees had extremely similar RMSE results with the out-of-bag error estimate and cross-validation (when compared across tuning parameters). It is unclear whether the pattern seen in these data generalizes, especially under different circumstances such as small sample sizes. Using the out-of-bag error rate would drastically decrease the computational time to tune random forest models. For forests created using conditional inference trees, the out-of-bag error was much more optimistic than the cross-validated RMSE. Again, the reasoning behind this pattern is unclear.

The ensemble nature of random forests makes it impossible to gain an understanding of the relationship between the predictors and the response. However, because trees are the typical base learner for this method, it is possible to quantify the impact of predictors in the ensemble. Breiman (2000) originally proposed randomly permuting the values of each predictor for the out-of-bag sample of one predictor at a time for each tree. The difference in predictive performance between the non-permuted sample and the permuted sample for each predictor is recorded and aggregated across the entire forest. Another approach is to measure the improvement in node purity based on the performance metric for each predictor at each occurrence of that predictor across the forest. These individual improvement values for each predictor are then aggregated across the forest to determine the overall importance for the predictor.

Although this strategy to determine the relative influence of a predictor is very different from the approach described in Sect. 8 for single regression trees, it suffers from the same limitations related to bias. Also, Strobl et al. (2007) showed that the correlations between predictors can have a significant impact on the importance values. For example, uninformative predictors with high correlations to informative predictors had abnormally large importance values. In some cases, their importance was greater than or equal to weakly important variables. They also demonstrated that the m_{try} tuning parameter has a serious effect on the importance values.

Another impact of between-predictor correlations is to dilute the importances of key predictors. For example, suppose a critical predictor had an importance of X. If another predictor is just as critical but is almost perfectly correlated as the first, the importance of these two predictors will be roughly $X/2$. If three such predictors were in the model, the values would further decrease to $X/3$ and so on. This can have profound implications for some problems. For example, RNA expression profiling data tend to measure the same gene at many locations, and, as a result, the within-gene variables tend to have very high correlations. If this gene were important for predicting some outcome, adding all the variables to a random forest model would make the gene appear to be less important than it actually is.

Strobl et al. (2007) developed an alternative approach for calculating importance in random forest models that takes between-predictor correlations into account. Their methodology reduces the effect of between-predictor redundancy. It does not adjust for the aforementioned dilution effect.

Random forest variable importance values for the top 25 predictors of the solubility data are presented in Fig. 8.19. For this model, `MolWeight`, `NumCarbon`, `SurfaceArea2`, and `SurfaceArea1` percolate to the top of the importance metric, and importance values begin to taper with fingerprints. Importance values for fingerprints 116 and 75 are top fingerprint performers for importance, which may indicate that the structures represented by these fingerprints have an impact on a compound's solubility.

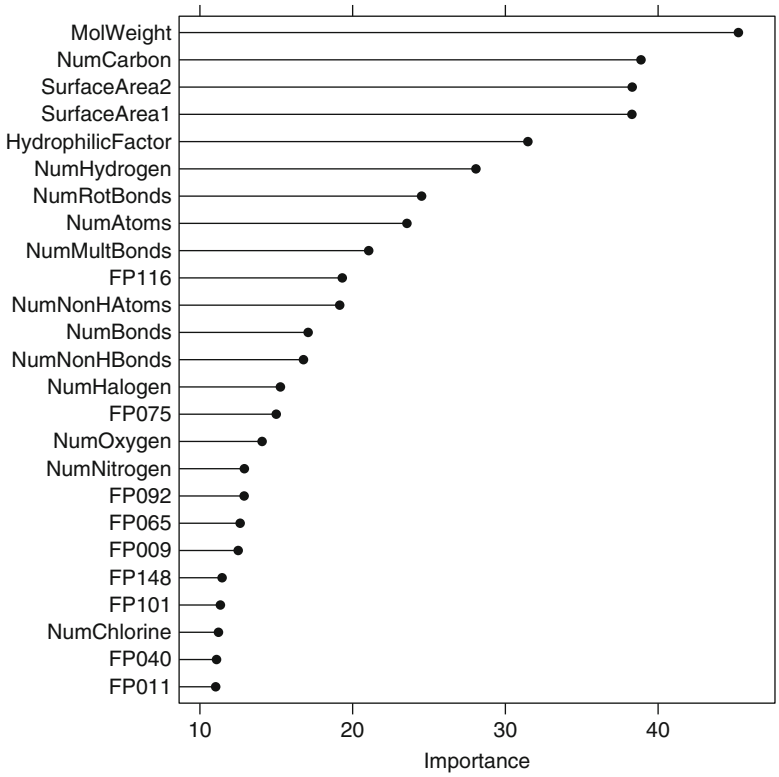

Fig. 8.19: Variable importance scores for the top 25 predictors used in the random forest CART tree model for solubility

Contrasting random forest importance results to a single CART tree (Fig. 8.6) we see that 2 of the top 4 predictors are the same (`SurfaceArea2` and `NumCarbon`) and 14 of the top 16 are the same. However, the importance orderings are much different. For example `NumNonHBonds` is the top predictor for a CART tree but ends up ranked 14th for random forests; random forests identify `MolWeight` as the top predictor, whereas a CART tree ranks it 5th. These differences should not be disconcerting; rather they emphasize that a single tree's greediness prioritizes predictors differently than a random forest.

8.6 Boosting

Boosting models were originally developed for classification problems and were later extended to the regression setting. Readers unfamiliar with boost-

ing may benefit by first reading about boosting for classification (Sect. 14.5) and then returning to this section. For completeness of this section, we will give a history of boosting to provide a bridge from boosting's original development in classification to its use in the regression context. This history begins with the AdaBoost algorithm and evolves to Friedman's stochastic gradient boosting machine, which is now widely accepted as the boosting algorithm of choice among practitioners.

In the early 1990s boosting algorithms appeared (Schapire 1990; Freund 1995; Schapire 1999), which were influenced by learning theory (Valiant 1984; Kearns and Valiant 1989), in which a number of weak classifiers (a classifier that predicts marginally better than random) are combined (or boosted) to produce an ensemble classifier with a superior generalized misclassification error rate. Researchers struggled for a time to find an effective implementation of boosting theory, until Freund and Schapire collaborated to produce the AdaBoost algorithm (Schapire 1999). AdaBoost (see Algorithm 14.2) provided a practical implementation of Kerns and Valiant's concept of boosting a weak learner into a strong learner (Kearns and Valiant 1989).

Boosting, especially in the form of the AdaBoost algorithm, was shown to be a powerful prediction tool, usually outperforming any individual model. Its success drew attention from the modeling community and its use became widespread with applications in gene expression (Dudoit et al. 2002; Ben-Dor et al. 2000), chemometrics (Varmuza et al. 2003), and music genre identification (Bergstra et al. 2006), to name a few.

The AdaBoost algorithm clearly worked, and after its successful arrival, several researchers (Friedman et al. 2000) connected the AdaBoost algorithm to statistical concepts of loss functions, additive modeling, and logistic regression and showed that boosting can be interpreted as a forward stagewise additive model that minimizes exponential loss. This fundamental understanding of boosting led to a new view of boosting that facilitated several algorithmic generalizations to classification problems (Sect. 14.5). Moreover, this new perspective also enabled the method to be extended to regression problems.

Friedman's ability to see boosting's statistical framework yielded a simple, elegant, and highly adaptable algorithm for different kinds of problems (Friedman 2001). He called this method "gradient boosting machines" which encompassed both classification and regression. The basic principles of gradient boosting are as follows: given a loss function (e.g., squared error for regression) and a weak learner (e.g., regression trees), the algorithm seeks to find an additive model that minimizes the loss function. The algorithm is typically initialized with the best guess of the response (e.g., the mean of the response in regression). The gradient (e.g., residual) is calculated, and a model is then fit to the residuals to minimize the loss function. The current model is added to the previous model, and the procedure continues for a user-specified number of iterations.

As described throughout this text, any modeling technique with tuning parameters can produce a range of predictive ability—from weak to strong. Because boosting requires a weak learner, almost any technique with tuning parameters can be made into a weak learner. Trees, as it turns out, make an excellent base learner for boosting for several reasons. First, they have the flexibility to be weak learners by simply restricting their depth. Second, separate trees can be easily added together, much like individual predictors can be added together in a regression model, to generate a prediction. And third, trees can be generated very quickly. Hence, results from individual trees can be directly aggregated, thus making them inherently suitable for an additive modeling process.

When regression tree are used as the base learner, simple gradient boosting for regression has two tuning parameters: tree depth and number of iterations. Tree depth in this context is also known as *interaction depth*, since each subsequential split can be thought of as a higher-level interaction term with all of the other previous split predictors. If squared error is used as the loss function, then a simple boosting algorithm using these tuning parameters can be found in Algorithm 8.3.

1 Select tree depth, D, and number of iterations, K
2 Compute the average response, $\overline{y}$, and use this as the initial predicted value for each sample
3 **for** $k = 1$ *to* K **do**
4 Compute the residual, the difference between the observed value and the *current* predicted value, for each sample
5 Fit a regression tree of depth, D, using the residuals as the response
6 Predict each sample using the regression tree fit in the previous step
7 Update the predicted value of each sample by adding the previous iteration's predicted value to the predicted value generated in the previous step
8 **end**

Algorithm 8.3: Simple Gradient Boosting for Regression

Clearly, the version of boosting presented in Algorithm 8.3 has similarities to random forests: the final prediction is based on an ensemble of models, and trees are used as the base learner. However, the way the ensembles are constructed differs substantially between each method. In random forests, all trees are created independently, each tree is created to have maximum depth,

and each tree contributes equally to the final model. The trees in boosting, however, are dependent on past trees, have minimum depth, and contribute unequally to the final model. Despite these differences, both random forests and boosting offer competitive predictive performance. Computation time for boosting is often greater than for random forests, since random forests can be easily parallel processed given that the trees are created independently.

Friedman recognized that his gradient boosting machine could be susceptible to over-fitting, since the learner employed—even in its weakly defined learning capacity—is tasked with optimally fitting the gradient. This means that boosting will select the optimal learner at each stage of the algorithm. Despite using weak learners, boosting still employs the greedy strategy of choosing the optimal weak learner at each stage. Although this strategy generates an optimal solution at the current stage, it has the drawbacks of not finding the optimal global model as well as over-fitting the training data. A remedy for greediness is to constrain the learning process by employing regularization, or shrinkage, in the same manner as illustrated in Sect. 6.4. In Algorithm 8.3, a regularization strategy can be injected into the final line of the loop. Instead of adding the predicted value for a sample to previous iteration's predicted value, only a fraction of the current predicted value is added to the previous iteration's predicted value. This fraction is commonly referred to as the *learning rate* and is parameterized by the symbol, λ. This parameter can take values between 0 and 1 and becomes another tuning parameter for the model. Ridgeway (2007) suggests that small values of the learning parameter (< 0.01) work best, but he also notes that the value of the parameter is inversely proportional to the computation time required to find an optimal model, because more iterations are necessary. Having more iterations also implies that more memory is required for storing the model.

After Friedman published his gradient boosting machine, he considered some of the properties of Breiman's bagging technique. Specifically, the random sampling nature of bagging offered a reduction in prediction variance for bagging. Friedman updated the boosting machine algorithm with a random sampling scheme and termed the new procedure *stochastic gradient boosting*. To do this, Friedman inserted the following step before line within the loop: randomly select a fraction of the training data. The residuals and models in the remaining steps of the current iteration are based only on the sample of data. The fraction of training data used, known as the bagging fraction, then becomes another tuning parameter for the model. It turns out that this simple modification improved the prediction accuracy of boosting while also reducing the required computational resources. Friedman suggests using a bagging fraction of around 0.5; this value, however, can be tuned like any other parameter.

Figure 8.20 presents the cross-validated RMSE results for boosted trees across tuning parameters of tree depth (1–7), number of trees (100–1,000), and shrinkage (0.01 or 0.1); the bagging fraction in this illustration was fixed at 0.5. When examining this figure, the larger value of shrinkage (right-hand

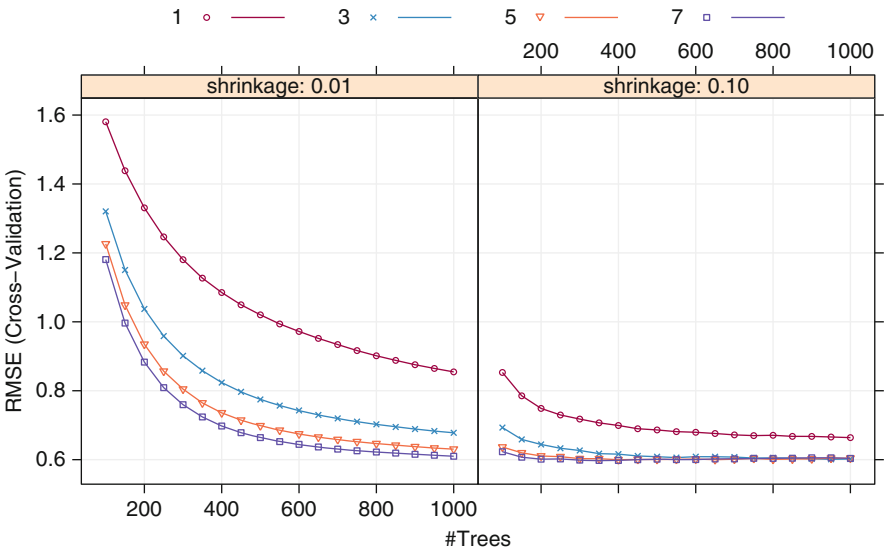

Fig. 8.20: Cross-validated RMSE profiles for the boosted tree model

plot) has an impact on reducing RMSE for all choices of tree depth and number of trees. Also, RMSE decreases as tree depth increases when shrinkage is 0.01. The same pattern holds true for RMSE when shrinkage is 0.1 and the number of trees is less than 300.

Using the one-standard-error rule, the optimal boosted tree has depth 3 with 400 trees and shrinkage of 0.1. These settings produce a cross-validated RMSE of 0.616.

Variable importance for boosting is a function of the reduction in squared error. Specifically, the improvement in squared error due to each predictor is summed within each tree in the ensemble (i.e., each predictor gets an improvement value for each tree). The improvement values for each predictor are then averaged across the entire ensemble to yield an overall importance value (Friedman 2002; Ridgeway 2007). The top 25 predictors for the model are presented in Fig. 8.21. `NumCarbon` and `MolWeight` stand out in this example as most important followed by `SurfaceArea1` and `SurfaceArea2`; importance values tail off after about 7 predictors. Comparing these results to random forests we see that both methods identify the same top 4 predictors, albeit in different order. The importance profile for boosting has a much steeper importance slope than the one for random forests. This is due to the fact that the trees from boosting are dependent on each other and hence will have correlated structures as the method follows by the gradient. Therefore many of the same predictors will be selected across the trees, increasing their contribution to the importance metric. Differences between variable importance

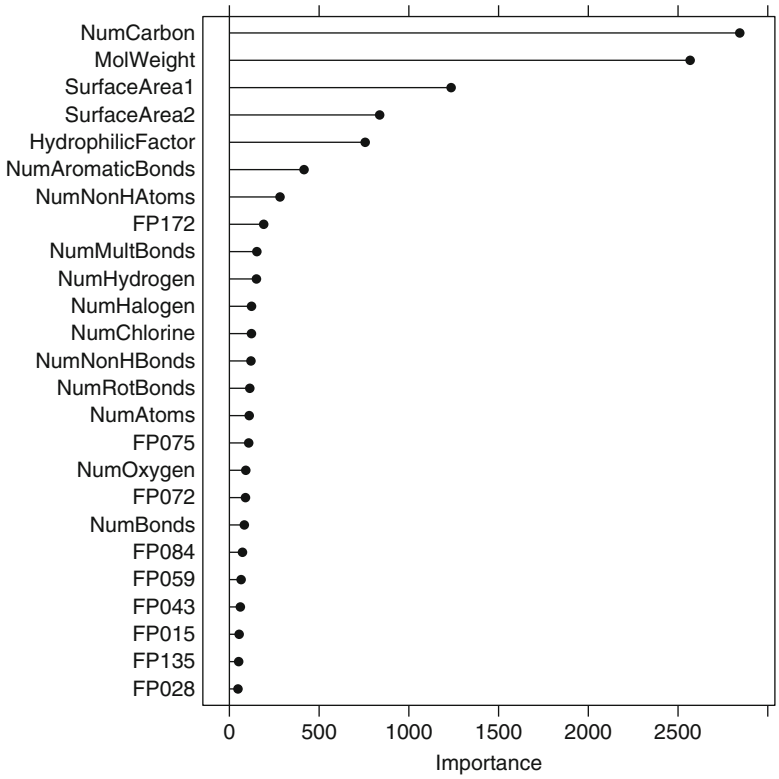

Fig. 8.21: Variable importance scores for the top 25 predictors used in the stochastic gradient boosting model for solubility

ordering and magnitude between random forests and boosting should not be disconcerting. Instead, one should consider these as two different perspectives of the data and use each view to provide some understanding of the gross relationships between predictors and the response.

8.7 Cubist

Cubist is a rule-based model that is an amalgamation of several methodologies published some time ago (Quinlan 1987, 1992, 1993a) but has evolved over this period. Previously, Cubist was only available in a commercial capacity, but in 2011 the source code was released under an open-source license. At this time, the full details of the current version of the model became public. Our description of the model stems from the open-source version

```
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

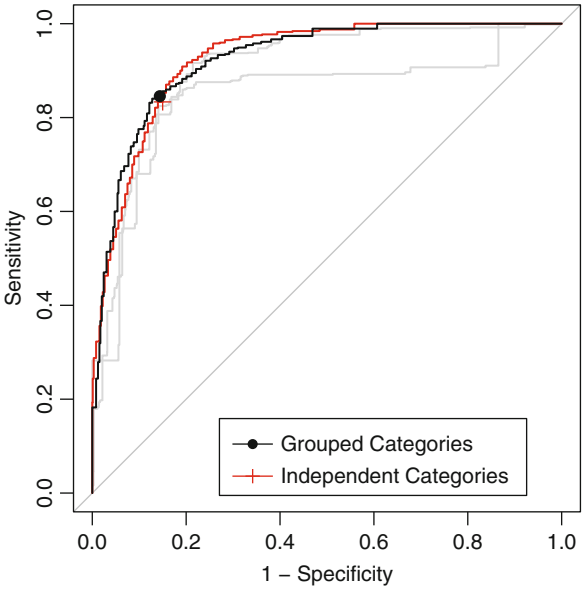

Fig. 14.7: The ROC curves for the bagged classification tree model. The area under the curves for both models was 0.92. The sensitivities and specificities were 82.98 and 85.71, respectively

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.

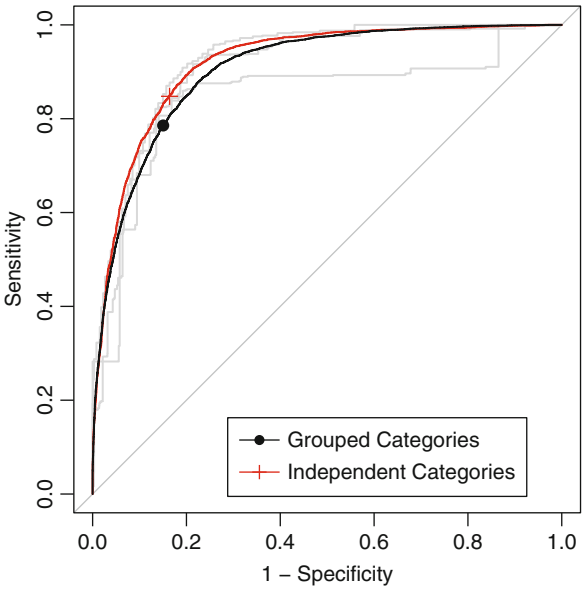

Fig. 14.8: The ROC curves for the random forest model. The area under the curve for independent categories was 0.92 and for the grouped category model the AUC was 0.9

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-